ListBox, you must add the element-typed ScrollBar style to the resources collection of the ListBox itself. And finally, it you want to change the way scroll bars look in your entire application, you can add it to the resources collection in the App.xaml file.

The ScrollBar control is surprisingly sophisticated. It's actually built out of a collection of smaller pieces, as shown in Figure 17-10.

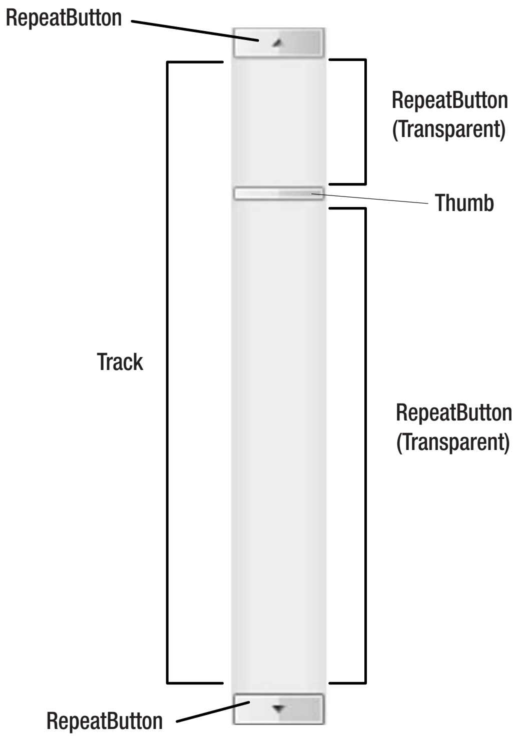


Figure 17-10. *Dissecting the scroll bar*

The background of the scroll bar is represented by the Track class—it's usually a shaded rectangle that's stretched out over the length of the scroll bar. At the far ends of the scroll bar are buttons that allow you to move one increment up or down (or to the left or right). These are instances of the RepeatButton class, which derives from ButtonBase. The key difference between a RepeatButton and the ordinary Button class is that if you hold the mouse down on a RepeatButton, the Click event fires over and over again (which is handy for scrolling).

In the middle of the scroll bar is a Thumb that represents the current position in the scrollable content. And, most interestingly of all, the blank space on either side of the thumb is actually made up of two more RepeatButton objects, which are transparent. When you click either one of these, the scroll bar scrolls an entire *page* (a page is defined as the amount that fits in the visible window of the scrollable content). This gives you the familiar ability to jump quickly through scrollable content by clicking the bar on either side of the thumb.

Here's the template for a vertical scroll bar:

```
<ControlTemplate x:Key="VerticalScrollBar" TargetType="{x:Type ScrollBar}">
  <Grid>
    <Grid.RowDefinitions>
      <RowDefinition MaxHeight="18"/>
      <RowDefinition Height="*"/>
      <RowDefinition MaxHeight="18"/>
    </Grid.RowDefinitions>

    <RepeatButton Grid.Row="0" Height="18"
      Style="{StaticResource ScrollBarLineButtonStyle}"
      Command="ScrollBar.LineUpCommand" >
      <Path Fill="{StaticResource GlyphBrush}"
       Data="M 0 4 L 8 4 L 4 0 Z"></Path>
    </RepeatButton>

    <Track Name="PART_Track" Grid.Row="1"
      IsDirectionReversed="True" ViewportSize="0">
      <Track.DecreaseRepeatButton>
        <RepeatButton Command="ScrollBar.PageUpCommand"
         Style="{StaticResource ScrollBarPageButtonStyle}">
        </RepeatButton>
      </Track.DecreaseRepeatButton>
      <Track.Thumb>
        <Thumb Style="{StaticResource ScrollBarThumbStyle}">
        </Thumb>
      </Track.Thumb>
      <Track.IncreaseRepeatButton>
        <RepeatButton Command="ScrollBar.PageDownCommand"
         Style="{StaticResource ScrollBarPageButtonStyle}">
        </RepeatButton>
      </Track.IncreaseRepeatButton>
    </Track>

    <RepeatButton
      Grid.Row="3" Height="18"
      Style="{StaticResource ScrollBarLineButtonStyle}"
      Command="ScrollBar.LineDownCommand"
      Content="M 0 0 L 4 4 L 8 0 Z">
    </RepeatButton>

    <RepeatButton
      Grid.Row="3" Height="18"
      Style="{StaticResource ScrollBarLineButtonStyle}"
      Command="ScrollBar.LineDownCommand">
      <Path Fill="{StaticResource GlyphBrush}"
        Data="M 0 0 L 4 4 L 8 0 Z"></Path>
    </RepeatButton>
  </Grid>
</ControlTemplate>
```

This template is fairly straightforward, once you understand the multipart structure of the scroll bar (as shown in Figure 17-10). There are a few key points to note:

- The vertical scroll bar consists of a three-row grid. The top and bottom rows hold the buttons at either end (and appear as arrows). They're fixed at 18 units. The middle section, which holds the track, takes the rest of the space.
- The RepeatButton at both ends use the same style. The only difference is the Content property that contains a Path that draws the arrow because the top button has an up arrow while the bottom button has a down arrow. For conciseness, these arrows are represented using the path mini-language described in Chapter 13. The other details, such as the background fill and the circle that appears around the arrow are defined in the control template, which is set out in the ScrollButtonLineStyle.
- Both buttons are linked to a command in the ScrollBar class (LineUpCommand and LineDownCommand). This is how they do their work. As long as you provide a button that's linked to this command, it doesn't matter what its name is, what it looks like, or what specific class it uses. (Commands are covered in detail in Chapter 9.)
- The Track has the name PART_Track. You must use this name in order for the ScrollBar class to hook up its code successfully. If you look at the default template for the ScrollBar class (which is similar, but lengthier), you'll see it appears there as well.

■ **Note** If you're examining a control with reflection (or using a tool such as Reflector), you can look for the TemplatePart attributes attached to the class declaration. There should be one TemplatePart attribute for each named part. The TemplatePart attribute indicates the name of the expected element (through the Name property) and its class (through the Type property). In Chapter 18, you'll see how to apply the TemplatePart attribute to your own custom control classes.

- The Track.ViewportSize property is set to 0. This is a specific implementation detail in this template. It ensures that the Thumb always has the same size. (Ordinarily, the thumb is sized proportionately based on the content so that if you're scrolling through content that mostly fits in the window, the thumb becomes much larger.)
- The Track wraps two RepeatButton objects (whose style is defined separately) and the Thumb. Once again, these buttons are wired up to the appropriate functionality using commands.

You'll also notice that the template uses a key name that specifically identifies it as a vertical scroll bar. As you learned in Chapter 11, when you set a key name on a style, you ensure that it isn't applied automatically, even if you've also set the TargetType property. The reason this example uses this approach is because the template is suitable only for scroll bars in the vertical orientation.

Another, element-typed style uses a trigger to automatically apply the control template if the ScrollBar.Orientation property is set to Vertical:

```
<Style TargetType="{x:Type ScrollBar}">
  <Setter Property="SnapsToDevicePixels" Value="True"/>
  <Setter Property="OverridesDefaultStyle" Value="true"/>
  <Style.Triggers>
    <Trigger Property="Orientation" Value="Vertical">
      <Setter Property="Width" Value="18"/>
      <Setter Property="Height" Value="Auto" />
      <Setter Property="Template" Value="{StaticResource VerticalScrollBar}" />
    </Trigger>
  </Style.Triggers>
</Style>
```

Although you could easily build a horizontal scroll bar out of the same basic pieces, this example doesn't take that step (and so retains the normally styled horizontal scroll bar).

The final task is to fill in the styles that format the various RepeatButton objects and the Thumb. These styles are relatively modest, but they do change the standard look of the scroll bar. First, the Thumb is shaped like an ellipse:

```
<Style x:Key="ScrollBarThumbStyle" TargetType="{x:Type Thumb}">
  <Setter Property="IsTabStop" Value="False"/>
  <Setter Property="Focusable" Value="False"/>
  <Setter Property="Margin" Value="1,0,1,0" />
  <Setter Property="Background" Value="{StaticResource StandardBrush}" />
  <Setter Property="BorderBrush" Value="{StaticResource StandardBorderBrush}" />
  <Setter Property="Template">
    <Setter.Value>
      <ControlTemplate TargetType="{x:Type Thumb}">
        <Ellipse Stroke="{StaticResource StandardBorderBrush}"
         Fill="{StaticResource StandardBrush}"></Ellipse>
      </ControlTemplate>
    </Setter.Value>
  </Setter>
</Style>
```

Next, the arrows at either end are drawn inside nicely rounded circles. The circles are defined in the control template, while the arrows are provided from the content of the RepeatButton and inserted into the control template using the ContentPresenter:

```
<Style x:Key="ScrollBarLineButtonStyle" TargetType="{x:Type RepeatButton}">
  <Setter Property="Focusable" Value="False"/>
  <Setter Property="Template">
    <Setter.Value>
      <ControlTemplate TargetType="{x:Type RepeatButton}">
        <Grid Margin="1">
          <Ellipse Name="Border" StrokeThickness="1"
           Stroke="{StaticResource StandardBorderBrush}"
           Fill="{StaticResource StandardBrush}"></Ellipse>
          <ContentPresenter HorizontalAlignment="Center"
           VerticalAlignment="Center"></ContentPresenter>
        </Grid>
        <ControlTemplate.Triggers>
          <Trigger Property="IsPressed" Value="true">
            <Setter TargetName="Border" Property="Fill"
             Value="{StaticResource PressedBrush}" />
          </Trigger>
        </ControlTemplate.Triggers>
      </ControlTemplate>
    </Setter.Value>
  </Setter>
</Style>
```

The RepeatButton objects that are displayed over the track aren't changed. They simply use a transparent background so the track shows through:

```
<Style x:Key="ScrollBarPageButtonStyle" TargetType="{x:Type RepeatButton}">
  <Setter Property="IsTabStop" Value="False"/>
  <Setter Property="Focusable" Value="False"/>
  <Setter Property="Template">
    <Setter.Value>
      <ControlTemplate TargetType="{x:Type RepeatButton}">
        <Border Background="Transparent" />
      </ControlTemplate>
    </Setter.Value>
  </Setter>
</Style>
```

Unlike the normal scroll bar, in this template no background is assigned to the Track, which leaves it transparent. That way, the gently shaded gradient of the list box shows through. Figure 17-11 shows the final list box.

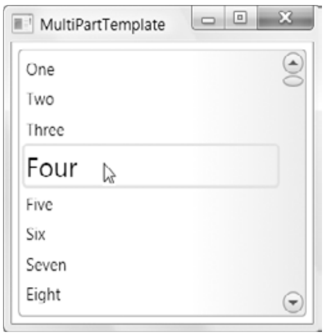


Figure 17-11. A list box with a customized scroll bar

The Control Template Examples

As you've seen, giving a new template to a common control can be a detailed task. That's because all the requirements of a control template aren't always obvious. For example, a typical ScrollBar requires a combination of two RepeatButton objects and a Track. Other control templates need elements with specific PART_ names. In the case of a custom window, you need to make sure the adorner layer is defined because some controls will require it.

Although you can discover these details by exploring the default template for a control, these default templates are often complicated and include details that aren't important and bindings that you probably won't support anyway. Fortunately, there's a better place to get started: the ControlTemplateExamples sample project (formerly known as the "Simple Styles").

The control template examples provide a collection of simple, streamlined templates for all WPF's standard controls, which makes them a useful jumping-off point for any custom control designer. Unlike the default control templates, these use standard colors, perform all their work declaratively (with no chrome classes), and leave out optional parts such as template bindings for less commonly used properties. The goal of control template examples is to give developers a practical starting point that they can use to design their own graphically enhanced control templates. Figure 17-12 shows about half of the control template examples.

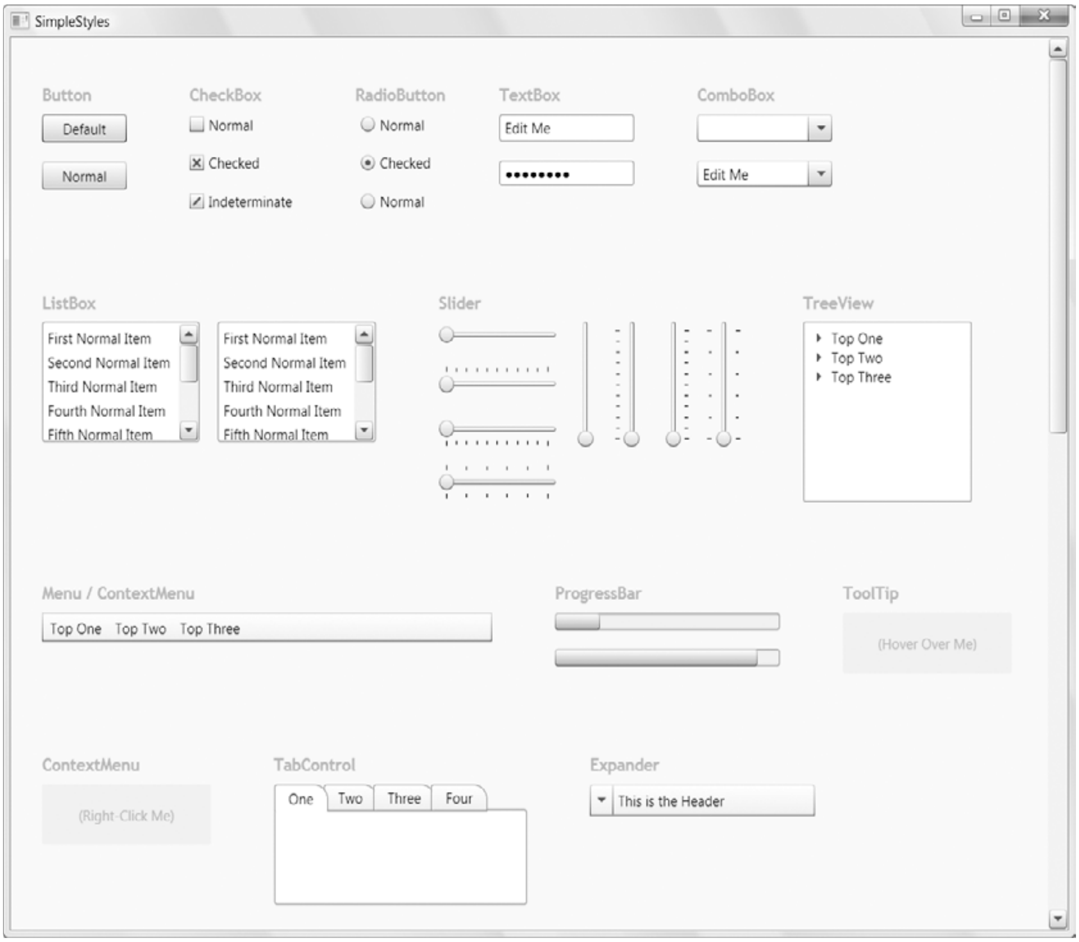


Figure 17-12. WPF controls with bare-bones styles

The SimpleStyles examples are included with the .NET Framework SDK. The easiest way to get them is to download them directly from `http://code.msdn.microsoft.com/wpfsamples#controlcustomization`.

■ **Tip** The SimpleStyles are one of the hidden gems of WPF. They provide templates that are easier to understand and enhance than the default control templates. If you need to enhance a common control with a custom look, this project should be your first stop.

Visual States

So far, you've learned the most direct (and most popular) way to craft a control template: using a mix of elements, binding expressions, and triggers. The elements create the overall visual structure of the control. The bindings pull information from the properties of the control class and apply it to the elements inside. And the triggers create the interactivity, allowing the control to change its appearance when its state changes.

The advantage of this model is that it's extremely powerful and extremely flexible. You can do just about anything you want. This isn't immediately apparent in the button example, because the control template relies on built-in properties like IsMouseOver and IsPressed. But even if these properties weren't available, you could still craft a control template that changes itself in response to mouse movements and button clicks. The trick would be to use event triggers that apply animations. For example, you could add an event trigger that reacts to the Border.MouseOver by starting an animation that changes the border background color. This animation doesn't even need to look like an animation—if you give it a duration of 0 seconds, it will apply itself immediately, just like the property triggers you're using now. In fact, this exact technique is used in a number of professional template examples.

Despite their capabilities, trigger-based templates have a downside: they require that the template designer has a detailed understanding of the way the control works. In the button example, the template designer needs to know about the existence of the IsMouseOver and IsPressed properties, for example, and how to use them. And these aren't the only details—for example, most controls need to react visually to mouse movements, being disabled, getting focus, and many other state changes. When these states are applied in combination, it can be difficult to determine exactly how the control should look. The trigger-based model is also notoriously awkward with *transitions*. For example, imagine you want to create a button that pulses while the mouse is over it. To get a professional result, you may need two animations—one that changes the state of the button from normal to mouseover and one that applies the continuous pulsing effect immediately after that. Managing all these details with a trigger-based template can be a challenge.

In WPF 4, Microsoft adds a new feature called *visual states*, which addresses this challenge. Using named parts (which you've already seen) and visual states, a control can provide a standardized visual contract. Rather than understanding the entire control, a template designer simply needs to understand the rules of the visual contract. As a result, it's much easier to design a simple control template—especially when it's for a control you've never worked with before.

Much as controls can use the TemplatePart attribute to indicate specific named elements (or parts) that the control template should include, they can use the TemplateVisualState attribute to indicate the visual states they support. For example, an ordinary button would provide a set of visual states like this:

```
[TemplateVisualState(Name="Normal", GroupName="CommonStates")]
[TemplateVisualState(Name="MouseOver", GroupName="CommonStates")]
[TemplateVisualState(Name="Pressed", GroupName="CommonStates")]
[TemplateVisualState(Name="Disabled", GroupName="CommonStates")]
[TemplateVisualState(Name="Unfocused", GroupName="FocusStates")]
[TemplateVisualState(Name="Focused", GroupName="FocusStates")]
public class Button : ButtonBase
{ ... }
```

States are placed together in *groups*. Groups are mutually exclusive, which means a control has one state in each group. For example, the button shown here has two state groups: CommonStates and FocusStates. At any given time, the button has one of the states from the CommonStates group *and* one of the states from the FocusStates group.

For example, if you tab to the button, its states will be Normal (from CommonStates) and Focused (from FocusStates). If you then move the mouse over the button, its states will be MouseOver (from CommonStates) and Focused (from FocusStates). Without state groups, you'd have trouble dealing with this situation. You'd either be forced to make some states dominate others (so a button in the MouseOver state would lose its focus indicator) or need to create many more states (such as FocusedNormal, UnfocusedNormal, FocusedMouseOver, UnfocusedMouseOver, and so on).

At this point, you can already see the appeal of the visual states model. From the template, it's immediately clear that a control template needs to address six different state possibilities. You also know the name of each state, which is the only essential detail. You don't need know what properties the Button class provides or understand the inner workings of the control. Best of all, if you Expression Blend, you'll get enhanced design-time support when creating control templates for a control that supports visual states. Blend will show you the named parts and visual states the control supports (as defined with the TemplatePart and TemplateVisualState attributes), and you can then add the corresponding elements and storyboards.

In the next chapter, you'll see a custom control named the FlipPanel that puts the visual state model into practice.

The Last Word

In this chapter, you learned how to use basic template building techniques to re-skin core WPF controls like the button, without being forced to reimplement any core button functionality. These custom buttons support all the normal button behavior—you can tab from one to the next, you can click them to fire an event, you can use access keys, and so on. Best of all, you can reuse your button template throughout your application and still replace it with a whole new design at a moment's notice.

So, what more do you need to know before you can skin all the basic WPF controls? To get the snazzy look you probably want, you might need to spend more time studying the details of WPF drawing (Chapter 12 and Chapter 13) and animation (Chapter 15 and Chapter 16). It might surprise you to know that you can use the shapes and brushes you've already learned about to build sophisticated controls with glass-style blurs and soft glow effects. The secret is in combining multiple layers of shapes, each with a different gradient brush. The best way to get this sort of effect is to learn from the control template examples others have created. Here are two good examples to check out:

- There are plenty of handcrafted, shaded buttons with glass and soft glow effects on the Web. You can find a complete tutorial that walks you through the process of creating a snazzy glass button in Expression Blend at `http://blogs.msdn.com/mgrayson/archive/2007/02/16/creating-a-glass-button-the-complete-tutorial.aspx`.
- An MSDN Magazine article about control templates provides examples of templates that incorporate simple drawings in innovative ways. For example, a CheckBox is replaced by an up-down lever, a slider is rendered with a three-dimensional tab, a ProgressBar is changed into a thermometer, and so on. Check it out at `http://msdn.microsoft.com/en-us/magazine/cc163497.aspx`.

If you don't want to type these links in by hand, you can find them listed on the page for this book at `http://www.prosetech.com`.

CHAPTER 18

Custom Elements

In previous Windows development frameworks, custom controls played a central role. But in WPF, the emphasis has shifted. Custom controls are still a useful way to build custom widgets that you can share between applications, but they're no longer a requirement when you want to enhance and customize core controls. (To understand how remarkable this change is, it helps to point out that this book's predecessor, *Pro .NET 2.0 Windows Forms and Custom Controls* in *C#*, had nine complete chapters about custom controls and additional examples in other chapters. But in this book, you've made it to Chapter 18 without a single custom control sighting!)

WPF de-emphasizes custom controls because of its support for styles, content controls, and templates. These features give every developer several ways to refine and extend standard controls without deriving a new control class. Here are your possibilities:

- **Styles.** You can use a style to painlessly reuse a combination of control properties. You can even apply effects using triggers. To get the same effect in Windows Forms, developers needed to copy and paste code (which was impractical) or derive a custom control with hardwired property setting logic in the constructor.
- **Content controls.** Any control that derives from ContentControl supports nested content. Using content controls, you can quickly create compound controls that aggregate other elements. (For example, you can transform a button into an image button or a list box into an image list.)
- **Control templates.** All WPF controls are *lookless*, which means they have hardwired functionality but the appearance is defined separately through the control template. Replace the default template with something new, and you can revamp basic controls such as buttons, check boxes, radio buttons, and even windows.
- **Data templates.** All ItemsControl-derived classes support data templates, which allow you to create a rich list representation of some type of data object. Using the right data template, you can display each item using a combination of text, images, and even editable controls, all in a layout container of your choosing.

If possible, you should pursue these avenues before you decide to create a custom control or another type of custom element. That's because these solutions are simpler, easier to implement, and often easier to reuse.

So, when *should* you create a custom element? Custom elements aren't the best choice when you want to fine-tune the appearance of an element, but they do make sense when you want to change its underlying functionality. For example, there's a reason that WPF has separate classes for the TextBox and PasswordBox classes. They handle key presses differently, store their data internally in a different way, interact with other components such as the clipboard differently, and so on. Similarly, if you want to design a control that has its own distinct set of properties, methods, and events, you'll need to build it yourself.

In this chapter, you'll learn how to create custom elements and how to make them into first-class WPF citizens. That means you'll outfit them with dependency properties and routed events to get support for essential WPF services such as data binding, styles, and animation. You'll also learn how to create a *lookless* control—a template-driven control that allows the control consumer to supply different visuals for greater flexibility.

■ **What's New** This chapter features a full-scale example of the new visual state model, which was introduced in Chapter 17. In the "Supporting Visual States" section, you'll learn about a custom FlipPanel that uses states and transitions.

Understanding Custom Elements in WPF

Although you can code a custom element in any WPF project, you'll usually want to place custom elements in a dedicated class library (DLL) assembly. That way, you can share your work with multiple WPF applications.

To make sure you have the right assembly references and namespace imports, you should choose the Custom Control Library (WPF) project type when you create your application in Visual Studio. Inside your class library, you can create as many or as few controls as you like.

■ **Tip** As with all class library development, it's often a good practice to place both your class library and the application that uses your class library in the same Visual Studio solution. That way you can easily modify and debug both pieces at once.

The first step in creating a custom control is choosing the right base class to inherit from. Table 18-1 lists some commonly used classes for creating custom controls, and Figure 18-1 shows where they fit into the element hierarchy.

Table 18-1. Base Classes for Creating a Custom Element

Name	Description
FrameworkElement	This is the lowest level you'll typically use when creating a custom element. Usually, you'll take this approach only if you want to draw your content from scratch by overriding OnRender() and using the System.Windows.Media.DrawingContext. It's similar to the approach you saw in Chapter 14, where a user interface was constructed using Visual objects. The FrameworkElement class provides the basic set of properties and events for elements that aren't intended to interact with the user.
Control	This is the most common starting point when building a control from scratch. It's the base class for all user-interactive widgets. The Control class adds properties for setting the background and foreground, as well as the font and alignment of content. The control class also places itself into the tab order (through the IsTabStop property) and introduces the notion of double-clicking (through the MouseDoubleClick and PreviewMouseDoubleClick events). But most important, the Control class defines the Template property that allows its appearance to be swapped out with a customized element tree for endless flexibility.
ContentControl	This is the base class for controls that can display a single piece of arbitrary content. That content can be an element or a custom object that's used in conjunction with a template. (The content is set through the Content property, and an optional template can be provided in the ContentTemplate property.) Many controls wrap a specific, limited type of content (like a string of text in a text box). Because these controls don't support all elements, they shouldn't be defined as content controls.
UserControl	This is a content control that can be configured using a design-time surface. Although a user control isn't that different from an ordinary content control, it's typically used when you want to quickly reuse an unchanging block of user interface in more than one window (rather than create a true stand-alone control that can be transported from one application to another).
ItemsControl or Selector	ItemsControl is the base class for controls that wrap a list of items but don't support selection, while Selector is the more specialized base class for controls that do support selection. These classes aren't often used to create custom controls, because the data templating features of the ListBox, ListView, and TreeView provide a great deal of flexibility.
Panel	This is the base class for controls with layout logic. A layout control can hold multiple children and arranges them according to specific layout semantics. Often, panels include attached properties that can be set on the children to configure how the children are arranged.

Name	Description
Decorator	This is the base class for elements that wrap another element and provide a graphical effect or specific feature. Two prominent examples are the Border, which draws a line around an element, and the Viewbox, which scales its content dynamically using a transform. Other decorators include the chrome classes used to give the familiar border and background to common controls like the button.
A specific control class	If you want to introduce a refinement to an existing control, you can derive directly from that control. For example, you can create a TextBox with built-in validation logic. However, before you take this step, consider whether you could accomplish the same thing using event handling code or a separate component. Both approaches allow you to decouple your logic from the control and reuse it with other controls.

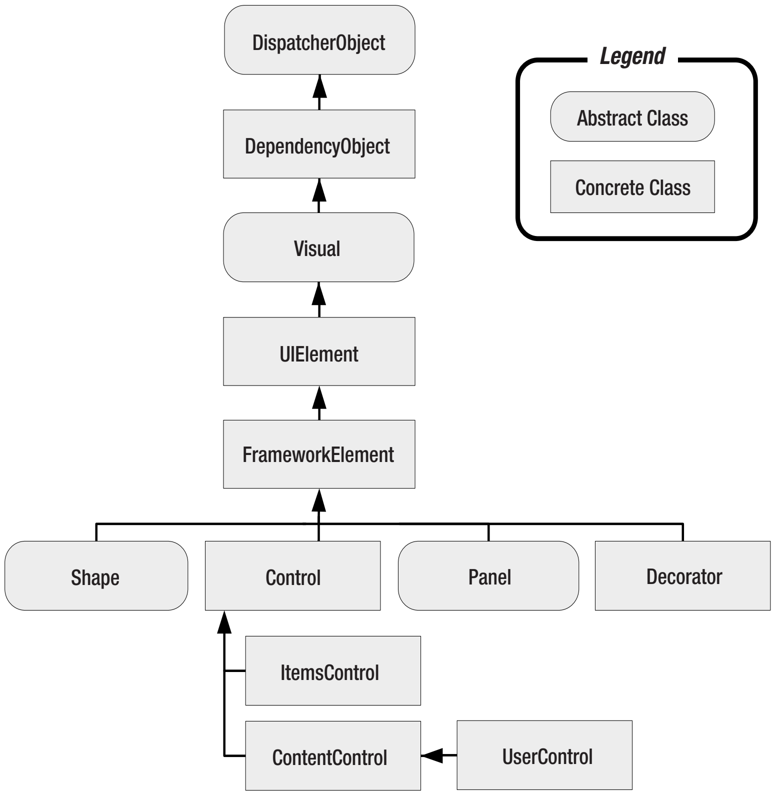


Figure 18-1. Element and control base classes

■ **Note** Although you can create a custom element that isn't a control, most custom elements you create in WPF will be controls—that is to say they'll be able to receive focus, and they'll interact with the user's key presses and mouse actions. For that reason, the terms *custom elements* and *custom controls* are sometimes used interchangeably in WPF development.

In this chapter, you'll see a user control, a lookless color picker that derives directly from the Control class, a lookless FlipPanel that uses visual states, a custom layout panel, and a custom-drawn element that derives from FrameworkElement and overrides OnRender(). Many of the examples are quite lengthy. Although you'll walk through almost all of the code in this chapter, you'll probably want to follow up by downloading the samples and playing with the custom controls yourself.

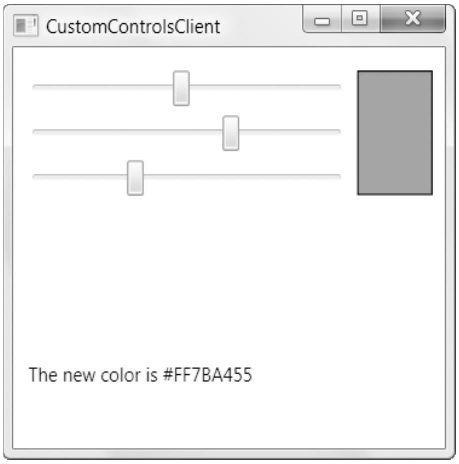
Building a Basic User Control

A good way to get started with custom controls is to take a crack at creating a straightforward user control. In this section, we'll begin by creating a basic color picker. Later, you'll see how to refactor this control into a more capable template-based control.

Creating a basic color picker is easy—in fact, several examples are available online, including one with the .NET Framework SDK (available at `http://code.msdn.microsoft.com/wpfsamples`). However, creating a custom color picker is still a worthy exercise. Not only does it demonstrate a variety of important control building concepts, but it also gives you a practical piece of functionality.

You could create a custom dialog box for your color picker, such as the kind that's included with Windows Forms. But if you want to create a color picker that you can integrate into different windows, a custom control is a far better choice. The most straightforward type of custom control is a *user control*, which allows you to assemble a combination of elements in the same way as when you design a window or page. Because the color picker appears to be little more than a fairly straightforward grouping of existing controls with added functionality, a user control seems like a perfect choice.

A typical color picker allows a user to select a color by clicking somewhere in a color gradient or specifying individual red, green, and blue components. Figure 18-2 shows the basic color picker you'll create in this section (at the top of the window). It consists of three Slider controls for adjusting color components, along with a Rectangle that shows a preview of the selected color.



The new color is #FF7BA455

Figure 18-2. *A color picker user control*

■ **Note** The user control approach has one significant flaw—it limits your ability to customize the appearance of your color picker to suit different windows, applications, and uses. Fortunately, it's not much harder to step up to a more template-based control, as you'll see a bit later.

Defining Dependency Properties

The first step in creating the color picker is to add a user control to your custom control library project. When you do, Visual Studio creates a XAML markup file and a corresponding custom class to hold your initialization and event handling code. This is the same experience as when you create a new window or page—the only difference is that the top-level container is the UserControl class.

```
public partial class ColorPicker : System.Windows.Controls.UserControl
{ ... }
```

The easiest starting point is to design the public interface that the user control exposes to the outside world. In other words, it's time to create the properties, methods, and events that the control consumer (the application that uses the control) will rely on to interact with the color picker.

The most fundamental detail is the Color property—after all, the color picker is nothing more than a specialized tool for displaying and choosing a color value. To support WPF features such as data binding, styles, and animation, writeable control properties are almost always dependency properties.

As you learned in Chapter 4, the first step to creating a dependency property is to define a static field for it, with the word *Property* added to the end of your property name:

```
public static DependencyProperty ColorProperty;
```

The Color property will allow the control consumer to set or retrieve the color value programmatically. However, the sliders in the color picker also allow the user to modify one aspect of the current color. To implement this design, you could use event handlers that respond when a slider value is changed and update the Color property accordingly. But it's cleaner to wire the sliders up using data binding. To make this possible, you need to define each of the color components as a separate dependency property:

```
public static DependencyProperty RedProperty;
public static DependencyProperty GreenProperty;
public static DependencyProperty BlueProperty;
```

Although the Color property will store a System.Windows.Media.Color object, the Red, Green, and Blue properties will store individual byte values that represent each color component. (You could also add a slider and a property for managing the alpha value, which allows you to create a partially transparent color, but this example doesn't add this detail.)

Defining the static fields for your properties is just the first step. You also need a static constructor in your user control that registers them, specifying the property name, the data type, and the control class that owns the property. As you learned in Chapter 4, this is the point where you can opt in to specific property features (such as value inheritance) by passing a FrameworkPropertyMetadata object with the right flags set. It's also the point where you can attach callbacks for validation, value coercion, and property change notifications.

In the color picker, you have just one consideration—you need to attach callbacks that respond when the various properties are changed. That's because the Red, Green, and Blue properties are really a different representation of the Color property, and if one property changes, you need to make sure the others stay synchronized.

Here's the static constructor code that registers the four dependency properties of the color picker:

```
static ColorPicker()
{
    ColorProperty =  DependencyProperty.Register(
      "Color", typeof(Color), typeof(ColorPicker),
      new FrameworkPropertyMetadata(Colors.Black,
        new PropertyChangedCallback(OnColorChanged)));

     RedProperty = DependencyProperty.Register(
      "Red", typeof(byte), typeof(ColorPicker),
      new FrameworkPropertyMetadata(
        new PropertyChangedCallback(OnColorRGBChanged)));

     GreenProperty = DependencyProperty.Register(
       "Green", typeof(byte), typeof(ColorPicker),
       new FrameworkPropertyMetadata(
         new PropertyChangedCallback(OnColorRGBChanged)));

     BlueProperty = DependencyProperty.Register(
       "Blue", typeof(byte), typeof(ColorPicker),
      new FrameworkPropertyMetadata(
        new PropertyChangedCallback(OnColorRGBChanged)));
}
```

Now that you have your dependency properties defined, you can add standard property wrappers that make them easier to access and usable in XAML.

```
public Color Color
{
    get { return (Color)GetValue(ColorProperty); }
    set { SetValue(ColorProperty, value); }
}

public byte Red
{
    get { return (byte)GetValue(RedProperty); }
    set { SetValue(RedProperty, value); }
}

public byte Green
{
    get { return (byte)GetValue(GreenProperty); }
    set { SetValue(GreenProperty, value); }
}

public byte Blue
{
    get { return (byte)GetValue(BlueProperty); }
    set { SetValue(BlueProperty, value); }
}
```

Remember, the property wrappers shouldn't contain any logic, because properties may be set and retrieved directly using the SetValue() and GetValue() methods of the base DependencyObject class. For example, the property synchronization logic in this example is implemented using callbacks that fire when the property changes through the property wrapper or a direct SetValue() call.

The property change callbacks are responsible for keeping the Color property consistent with the Red, Green, and Blue properties. Whenever the Red, Green, or Blue property is changed, the Color property is adjusted accordingly:

```
private static void OnColorRGBChanged(DependencyObject sender,
  DependencyPropertyChangedEventArgs e)
{
    ColorPicker colorPicker = (ColorPicker)sender;
    Color color = colorPicker.Color;

    if (e.Property == RedProperty)
        color.R = (byte)e.NewValue;
    else if (e.Property == GreenProperty)
        color.G = (byte)e.NewValue;
    else if (e.Property == BlueProperty)
        color.B = (byte)e.NewValue;

    colorPicker.Color = color;
}
```

and when the Color property is set, the Red, Green, and Blue values are also updated:

```
private static void OnColorChanged(DependencyObject sender,
  DependencyPropertyChangedEventArgs e)
{
    Color newColor = (Color)e.NewValue;
    Color oldColor = (Color)e.OldValue;

    ColorPicker colorPicker = (ColorPicker)sender;
    colorPicker.Red = newColor.R;
    colorPicker.Green = newColor.G;
    colorPicker.Blue = newColor.B;
}
```

Despite its appearances, this code won't cause an infinite series of calls as each property tries to change the other. That's because WPF doesn't allow reentrancy in the property change callbacks. For example, if you change the Color property, the OnColorChanged() method will be triggered. The OnColorChanged() method will modify the Red, Green, and Blue properties, triggering the OnColorRGBChanged() callback three times (once for each property). However, the OnColorRBGChanged() will not trigger the OnColorChanged() method again.

■ **Tip** It might occur to you to use the coercion callbacks discussed in Chapter 4 to deal with the color properties. However, this approach isn't appropriate. Property coercion callbacks are designed for properties that are interrelated and may override or influence one another. They don't make sense for properties that expose the same data in different ways. If you used property coercion in this example, it would be possible to set different values in the Red, Green, and Blue properties and have that color information *override* the Color property. The behavior you really want is to set the Red, Green, and Blue properties and use that color information to permanently *change* the value of the Color property.

Defining Routed Events

You might also want to add routed events that can be used to notify the control consumer when something happens. In the color picker example, it's useful to have an event that fires when the color is changed. Although you could define this event as an ordinary .NET event, using a routed event allows you to provide event bubbling and tunneling, so the event can be handled in a higher-level parent, such as the containing window.

As with the dependency properties, the first step to defining a routed event is to create a static property for it, with the word *Event* added to the end of the event name:

```
public static readonly RoutedEvent ColorChangedEvent;
```

You can then register the event in the static constructor. At this point you specify the event name, the routing strategy, the signature, and the owning class:

```
ColorChangedEvent = EventManager.RegisterRoutedEvent(
  "ColorChanged", RoutingStrategy.Bubble,
  typeof(RoutedPropertyChangedEventHandler<Color>), typeof(ColorPicker));
```

Rather than going to the work of creating a new delegate for your event signature, you can sometimes reuse existing delegates. The two useful delegates are RoutedEventHandler (for a routed event that doesn't pass along any extra information) and RoutedPropertyChangedEventHandler (for a routed event that provides the old and new values after a property has been changed). The RoutedPropertyChangedEventHandler, which is used in the previous example, is a generic delegate that's parameterized by type. As a result, you can use it with any property data type without sacrificing type safety.

Once you've defined and registered the event, you need to create a standard .NET event wrapper that exposes your event. This event wrapper can be used to attach (and remove) event listeners:

```
public event RoutedPropertyChangedEventHandler<Color> ColorChanged
{
    add { AddHandler(ColorChangedEvent, value); }
    remove { RemoveHandler(ColorChangedEvent, value); }
}
```

The final detail is the code that raises the event at the appropriate time. This code must call the RaiseEvent() method that's inherited from the base DependencyObject class.

In the color picker example, you simply need to add these lines of code to the end of the OnColorChanged() method:

```
Color oldColor = (Color)e.OldValue;
RoutedPropertyChangedEventArgs<Color> args =
  new RoutedPropertyChangedEventArgs<Color>(oldColor, newColor);
args.RoutedEvent = ColorPicker.ColorChangedEvent;

colorPicker.RaiseEvent(args);
```

Remember, the OnColorChanged() callback is triggered whenever the Color property is modified, either directly or by modifying the Red, Green, and Blue color components.

Adding Markup

Now that your user control's public interface is in place, all you need is the markup that creates the control's appearance. In this case, a basic Grid is all that's needed to bring together the three Slider controls and the Rectangle with the color preview. The trick is the data binding expressions that tie these controls to the appropriate properties, with no event handling code required.

All in all, four data binding expressions are at work in the color picker. The three sliders are bound to the Red, Green, and Blue properties and are allowed to range from 0 to 255 (the acceptable values for a byte). The Rectangle.Fill property is set using a SolidColorBrush, and the Color property of that brush is bound to the Color property of the user control.

Here's the complete markup:

```
<UserControl x:Class="CustomControls.ColorPicker"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml" Name="colorPicker">
  <Grid>
    <Grid.RowDefinitions>
      <RowDefinition Height="Auto"></RowDefinition>
      <RowDefinition Height="Auto"></RowDefinition>
      <RowDefinition Height="Auto"></RowDefinition>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
      <ColumnDefinition></ColumnDefinition>
      <ColumnDefinition Width="Auto"></ColumnDefinition>
    </Grid.ColumnDefinitions>

    <Slider Name="sliderRed" Minimum="0" Maximum="255"
     Value="{Binding ElementName=colorPicker,Path=Red}"></Slider>
    <Slider Grid.Row="1" Name="sliderGreen" Minimum="0" Maximum="255"
     Value="{Binding ElementName=colorPicker,Path=Green}"></Slider>
    <Slider Grid.Row="2" Name="sliderBlue" Minimum="0" Maximum="255"
     Value="{Binding ElementName=colorPicker,Path=Blue}"></Slider>

    <Rectangle Grid.Column="1" Grid.RowSpan="3"
     Width="50" Stroke="Black" StrokeThickness="1">
      <Rectangle.Fill>
        <SolidColorBrush Color="{Binding ElementName=colorPicker,Path=Color}">
        </SolidColorBrush>
      </Rectangle.Fill>
    </Rectangle>

  </Grid>
</UserControl>
```

The markup for a user control plays the same role as the control template for a lookless control. If you want to make some of the details in your markup configurable, you can use binding expressions that link them to control properties. For example, currently the Rectangle's width is hard-coded at 50 units. However, you could replace this detail with a data binding expression that pulls the value from a dependency property in your user control. That way, the control consumer could modify that property to choose a different width. Similarly, you could make the stroke color and thickness variable. However, if you want to make a control with real flexibility, you're much better off to create a lookless control and define the markup in a template, as described later in this chapter.

Occasionally, you might choose to use binding expressions to repurpose one of the core properties that's already defined in your control. For example, the UserControl class uses its Padding property to add space between the outer edge and the inner content that you define. (This detail is implemented through the control template for the UserControl.) However, you could also use the Padding property to set the spacing around each slider, as shown here:

```
<Slider Name="sliderRed" Minimum="0" Maximum="255"
  Margin="{Binding ElementName=colorPicker,Path=Padding}"
  Value="{Binding ElementName=colorPicker,Path=Red}"></Slider>
```

Similarly, you could grab the border settings for the Rectangle from the BorderThickness and BorderBrush properties of the UserControl. Once again, this is a shortcut that may make perfect sense for creating simple controls but can be improved by introducing additional properties (for example, SliderMargin, PreviewBorderBrush, and PreviewBorderThickness) or creating a full-fledged template-based control.