PART II

Deeper Into WPF

CHAPTER 6

Controls

Now that you've learned the fundamentals of layout, content, and event handling, you're ready to take a closer look at WPF's family of elements. In this chapter, you'll consider *controls*—elements that derive from the System.Windows.Control class. You'll begin by examining the base Control class, and learning how it supports brushes and fonts. Then you'll explore the full catalog of WPF controls, including the following:

Content controls: These controls can contain nested elements, giving them nearly unlimited display capabilities. They include the Label, Button, ToolTip, and ScrollViewer classes.

Headered content controls: These are content controls that allow you to add a main section of content and a separate title portion. They are usually intended to wrap larger blocks of user interface. They include the TabItem, GroupBox, and Expander classes.

Text controls: This is the small set of controls that allow users to enter input. The text controls support ordinary text (the TextBox), passwords (the PasswordBox), and formatted text (the RichTextBox, which is discussed in Chapter 28).

List controls: These controls show collections of items in a list. They include the ListBox and ComboBox classes.

Range-based controls: These controls have just one thing in common: a Value property that can be set to any number in a prescribed range. Examples include the Slider and ProgressBar classes.

Date controls: This category includes two controls that allow users to select dates: the Calendar and DatePicker.

There are several types of controls that you won't see in this chapter, including those that create menus, toolbars, and ribbons; those that show grids and trees of bound data; and those that allow rich document viewing and editing. You'll consider these more advanced controls throughout this book, as you explore the related WPF features.

The Control Class

WPF windows are filled with elements, but only some of these elements are *controls*. In the world of WPF, a control is generally described as a *user-interactive* element—that is, an element that can receive focus and accept input from the keyboard or mouse. Obvious examples include text boxes and buttons. However, the distinction is sometimes a bit blurry. A tooltip is considered to be a control because it appears and disappears depending on the user's mouse movements. A label is considered to be a control because of its support for *mnemonics* (shortcut keys that transfer the focus to related controls).

All controls derive from the System.Windows.Control class, which adds a bit of basic infrastructure:

- The ability to set the alignment of content inside the control
- The ability to set the tab order
- Support for painting a background, foreground, and border
- Support for formatting the size and font of text content

Background and Foreground Brushes

All controls include the concept of a background and foreground. Usually, the background is the surface of the control (think of the white or gray area inside the borders of a button), and the foreground is the text. In WPF, you set the color of these two areas (but not the content) by using the Background and Foreground properties.

It's natural to expect that the Background and Foreground properties would use color objects. However, these properties actually use something much more versatile: a Brush object. That gives you the flexibility to fill your background and foreground content with a solid color (by using the SolidColorBrush) or something more exotic (for example, by using a LinearGradientBrush or TileBrush). In this chapter, you'll consider only the simple SolidColorBrush, but you'll try fancier brushwork in Chapter 12.

Setting Colors in Code

Imagine you want to set a blue surface area inside a button named cmd. Here's the code that does the trick:

```
cmd.Background = new SolidColorBrush(Colors.AliceBlue);
```

This code creates a new SolidColorBrush by using a ready-made color via a static property of the handy Colors class. (The names are based on the color names supported by most web browsers.) It then sets the brush as the background brush for the button, which causes its background to be painted a light shade of blue.

■ **Note** This method of styling a button isn't completely satisfactory. If you try it, you'll find that it configures the background color for a button in its normal (unpressed) state, but it doesn't change the color that appears when you press the button (which is a darker gray). To really customize every aspect of a button's appearance, you need to delve into templates, as discussed in Chapter 17.

You can also grab system colors (which may be based on user preferences) from the System.Windows.SystemColors enumeration. Here's an example:

```
cmd.Background = new SolidColorBrush(SystemColors.ControlColor);
```

Because system brushes are used frequently, the SystemColors class also provides ready-made properties that return SolidColorBrush objects. Here's how to use them:

```
cmd.Background = SystemColors.ControlBrush;
```

As written, both of these examples suffer from a minor problem. If the system color is changed *after* you run this code, your button won't be updated to use the new color. In essence, this code grabs a snapshot of the current color or brush. To make sure your program can update itself in response to configuration changes, you need to use dynamic resources, as described in Chapter 10.

The Colors and SystemColors classes offer handy shortcuts, but they're not the only way to set a color. You can also create a Color object by supplying the R, G, B values (red, green, and blue). Each one of these values is a number from 0 to 255:

```
int red = 0; int green = 255; int blue = 0;
cmd.Foreground = new SolidColorBrush(Color.FromRgb(red, green, blue));
```

You can also make a color partly transparent by supplying an alpha value and calling the Color.FromArgb() method. An alpha value of 255 is completely opaque, while 0 is completely transparent.

RGB AND SCRGB

The RGB standard is useful because it's used in many other programs. For example, you can get the RGB value of a color in a graphic in a paint program and use the same color in your WPF application. However, other devices (such as printers) might support a richer range of colors. Therefore, an alternative scRGB standard has been created. This standard uses 64-bit values to represent each color component (alpha, red, green, and blue).

The WPF Color structure supports either approach. It includes a set of standard RGB properties (A, R, G, and B) and a set of properties for scRGB (ScA, ScR, ScG, and ScB). These properties are linked, so that if you set the R property, the ScR property is changed accordingly.

The relationship between the RGB values and the scRGB values is not linear. A 0 value in the RGB system is 0 in scRGB, 255 in RGB becomes 1 in scRGB, and all values between 0 and 255 in RGB are represented as decimal values between 0 and 1 in scRGB.

Setting Colors in XAML

When you set the background or foreground in XAML, you can use a helpful shortcut. Rather than define a Brush object, you can supply a color name or color value. The WPF parser will automatically create a SolidColorBrush object using the color you specify, and it will use that brush object for the foreground or background. Here's an example that uses a color name:

```
<Button Background="Red">A Button</Button>
```

It's equivalent to this more verbose syntax:

```
<Button>A Button
  <Button.Background>
    <SolidColorBrush Color="Red" />
  </Button.Background>
</Button>
```

You need to use the longer form if you want to create a different type of brush, such as a LinearGradientBrush, and use that to paint the background.

If you want to use a color code, you need to use a slightly less convenient syntax that puts the R, G, and B values in hexadecimal notation. You can use one of two formats—either #rrggbb or #aarrggbb (the difference being that the latter includes the alpha value). You need only two digits to supply the A, R, G, and B values because they're all in hexadecimal notation. Here's an example that uses the #aarrggbb notation to create the same color as in the previous code snippets:

```
<Button Background="#FFFF0000">A Button</Button>
```

Here, the alpha value is FF (255), the red value is FF (255), and the green and blue values are 0.

■ **Note** Brushes support automatic change notification. In other words, if you attach a brush to a control and change the brush, the control updates itself accordingly. This works because brushes derive from the System.Windows.Freezable class. The name stems from the fact that all freezable objects have two states—a readable state and a read-only (or "frozen") state.

The Background and Foreground properties are not the only details you can set with a brush. You can also paint a border around controls (and some other elements, such as the Border element) by using the BorderBrush and BorderThickness properties. BorderBrush takes a brush of your choosing, and BorderThickness takes the width of the border in device-independent units. You need to set both properties before you'll see the border.

■ **Note** Some controls don't respect the BorderBrush and BorderThickness properties. The Button object ignores them completely because it defines its background and border by using the ButtonChrome decorator. However, you can give a button a new face (with a border of your choosing) by using templates, as described in Chapter 17.

Fonts

The Control class defines a small set of font-related properties that determine how text appears in a control. These properties are outlined in Table 6-1.

Table 6-1. *Font-Related Properties of the Control Class*

Name	Description
FontFamily	The name of the font you want to use.
FontSize	The size of the font in device-independent units (each of which is 1/96 inch). This is a bit of a change from tradition that's designed to support WPF's new resolution-independent rendering model. Ordinary Windows applications measure fonts by using *points*, which are assumed to be 1/72 inch on a standard PC monitor. If you want to turn a WPF font size into a more familiar point size, you can use a handy trick—just multiply by 3/4. For example, a traditional 38-point font is equivalent to 48 units in WPF.

FontStyle	The angling of the text, as represented as a FontStyle object. You get the FontSyle preset you need from the static properties of the FontStyles class, which includes Normal, Italic, or Oblique lettering. (*Oblique* is an artificial way to create italic text on a computer that doesn't have the required italic font. Letters are taken from the normal font and slanted by using a transform. This usually creates a poor result.)
FontWeight	The heaviness of text, as represented as a FontWeight object. You get the FontWeight preset you need from the static properties of the FontWeights class. Bold is the most obvious of these, but some typefaces provide other variations, such as Heavy, Light, ExtraBold, and so on.
FontStretch	The amount that text is stretched or compressed, as represented by a FontStretch object. You get the FontStretch preset you need from the static properties of the FontStretches class. For example, UltraCondensed reduces fonts to 50% of their normal width, while UltraExpanded expands them to 200%. Font stretching is an OpenType feature that is not supported by many typefaces. (To experiment with this property, try using the Rockwell font, which does support it.)

■ **Note** The Control class doesn't define any properties that use its font. While many controls include a property such as Text, that isn't defined as part of the base Control class. Obviously, the font properties don't mean anything unless they're used by the derived class.

Font Family

A *font family* is a collection of related typefaces. For example, Arial Regular, Arial Bold, Arial Italic, and Arial Bold Italic are all part of the Arial font family. Although the typographic rules and characters for each variation are defined separately, the operating system realizes they are related. As a result, you can configure an element to use Arial Regular, set the FontWeight property to Bold, and be confident that WPF will switch over to the Arial Bold typeface.

When choosing a font, you must supply the full family name, as shown here:

```
<Button Name="cmd" FontFamily="Times New Roman" FontSize="18">A Button</Button>
```

It's much the same in code:

```
cmd.FontFamily = "Times New Roman";
cmd.FontSize = "18";
```

When identifying a FontFamily, a shortened string is not enough. That means you can't substitute Times or Times New instead of the full name Times New Roman.

Optionally, you can use the full name of a typeface to get italic or bold, as shown here:

```
<Button FontFamily="Times New Roman Bold">A Button</Button>
```

However, it's clearer and more flexible to use just the family name and set other properties (such as FontStyle and FontWeight) to get the variant you want. For example, the following markup sets the FontFamily to Times New Roman and sets the FontWeight to FontWeights.Bold:

```
<Button FontFamily="Times New Roman" FontWeight="Bold">A Button</Button>
```

Text Decorations and Typography

Some elements also support more-advanced text manipulation through the TextDecorations and Typography properties. These allow you to add embellishments to text. For example, you can set the TextDecorations property by using a static property from the TextDecorations class. It provides just four decorations, each of which allows you to add some sort of line to your text. They include Baseline, OverLine, Strikethrough, and Underline. The Typography property is more advanced—it lets you access specialized typeface variants that only some fonts will provide. Examples include different number alignments, ligatures (connections between adjacent letters), and small caps.

For the most part, the TextDecorations and Typography features are found only in flow document content—which you use to create rich, readable documents. (Chapter 28 describes documents in detail.) However, the frills also turn up on the TextBox class. Additionally, they're supported by the TextBlock, which is a lighter-weight version of the Label that's perfect for showing small amounts of wrappable text content. Although you're unlikely to use text decorations with the TextBox or change its typography, you may want to use underlining in the TextBlock, as shown here:

```
<TextBlock TextDecorations="Underline">Underlined text</TextBlock>
```

If you're planning to place a large amount of text content in a window and you want to format individual portions (for example, underline important words), you should refer to Chapter 28, where you'll learn about many more flow elements. Although flow elements are designed for use with documents, you can nest them directly inside a TextBlock.

Font Inheritance

When you set any of the font properties, the values flow through to nested objects. For example, if you set the FontFamily property for the top-level window, every control in that window gets the same FontFamily value (unless the control explicitly sets a different font). This works because the font properties are dependency properties, and one of the features that dependency properties can provide is property value inheritance—the magic that passes your font settings down to nested controls.

It's worth noting that property value inheritance can flow through elements that don't even support that property. For example, imagine you create a window that holds a StackPanel, inside of which are three Label controls. You can set the FontSize property of the window because the Window class derives from the Control class. You *can't* set the FontSize property for the StackPanel because it isn't a control. However, if you set the FontSize property of the window, your property value is still able to flow through the StackPanel to get to your labels inside and change their font sizes.

Along with the font settings, several other base properties use property value inheritance. In the Control class, the Foreground property uses inheritance. The Background property does not. (However, the default background is a null reference that's rendered by most controls as a transparent background. That means the parent's background will still show through.) In the UIElement class, AllowDrop, IsEnabled, and IsVisible use property inheritance. In the FrameworkElement, the CultureInfo and FlowDirection properties do.

■ **Note** A dependency property supports inheritance only if the FrameworkPropertyMetadata.Inherits flag is set to true, which is not the default. Chapter 4 discusses the FrameworkPropertyMetadata class and property registration in detail.

Font Substitution

When you're setting fonts, you need to be careful to choose a font that you know will be present on the user's computer. However, WPF does give you a little flexibility with a font fallback system. You can set FontFamily to a comma-separated list of font options. WPF will then move through the list in order, trying to find one of the fonts you've indicated.

Here's an example that attempts to use Technical Italic font but falls back to Comic Sans MS or Arial if that isn't available:

```
<Button FontFamily="Technical Italic, Comic Sans MS, Arial">A Button</Button>
```

If a font family really does contain a comma in its name, you'll need to escape the comma by including it twice in a row.

Incidentally, you can get a list of all the fonts that are installed on the current computer by using the static SystemFontFamilies collection of the System.Windows.Media.Fonts class. Here's an example that uses the collection to add fonts to a list box:

```
foreach (FontFamily fontFamily in Fonts.SystemFontFamilies)
{
    lstFonts.Items.Add(fontFamily.Source);
}
```

The FontFamily object also allows you to examine other details, such as the line spacing and associated typefaces.

■ **Note** One of the ingredients that WPF doesn't include is a dialog box for choosing a font. The WPF Text team has posted two much more attractive WPF font pickers, including a no-code version that uses data binding (`http://blogs.msdn.com/text/archive/2006/06/20/592777.aspx`) and a more sophisticated version that supports the optional typographic features that are found in some OpenType fonts (`http://blogs.msdn.com/text/archive/2006/11/01/sample-font-chooser.aspx`).

Font Embedding

Another option for dealing with unusual fonts is to embed them in your application. That way, your application never has a problem finding the font you want to use.

The embedding process is simple. First, you add the font file (typically, a file with the extension .ttf) to your application and set the Build Action option to Resource. (You can do this in Visual Studio by selecting the font file in the Solution Explorer and changing its Build Action in the Properties window.)

Next, when you use the font, you need to add the character sequence `./#` before the font family name, as shown here:

```
<Label FontFamily="./#Bayern" FontSize="20">This is an embedded font</Label>
```

The `./` characters are interpreted by WPF to mean *the current folder*. To understand what this means, you need to know a little more about XAML's packaging system.

As you learned in Chapter 2, you can run stand-alone (known as *loose*) XAML files directly in your browser without compiling them. The only limitation is that your XAML file can't use a code-behind file. In this scenario, the current folder is exactly that, and WPF looks at the font files that are in the same directory as the XAML file and makes them available to your application.

More commonly, you'll compile your WPF application to a .NET assembly before you run it. In this case, the current folder is still the location of the XAML document, only now that document has been compiled and embedded in your assembly. WPF refers to compiled resources by using a specialized URI syntax that's discussed in Chapter 7. All application URIs start with `pack://application`. If you create a project named ClassicControls and add a window named EmbeddedFont.xaml, the URI for that window is this:

```
pack://application:,,,/ClassicControls/embeddedfont.xaml
```

This URI is made available in several places, including through the FontFamily.BaseUri property. WPF uses this URI to base its font search. Thus, when you use the `./` syntax in a compiled WPF application, WPF looks for fonts that are embedded as resources alongside your compiled XAML.

After the `./` character sequence, you can supply the file name, but you'll usually just add the number sign (#) and the font's real family name. In the previous example, the embedded font is named Bayern.

■ **Note** Setting up an embedded font can be a bit tricky. You need to make sure you get the font family name exactly right, and you need to make sure you choose the correct build action for the font file. To see an example of the correct setup, refer to the sample code for this chapter.

Embedding fonts raises obvious licensing concerns. Unfortunately, most font vendors allow their fonts to be embedded in documents (such as PDF files) but not applications (such as WPF assemblies), even though an embedded WPF font isn't directly accessible to the end user. WPF doesn't make any attempt to enforce font licensing, but you should make sure you're on solid legal ground before you redistribute a font.

You can check a font's embedding permissions by using Microsoft's free font properties extension utility, which is available at `www.microsoft.com/typography/TrueTypeProperty21.mspx`. After you install this utility, right-click any font file and choose Properties to see more-detailed information about it. In particular, check the Embedding tab for information about the allowed embedding for this font. Fonts marked with Installed Embedding Allowed are suitable for WPF applications; fonts with Editable Embedding Allowed may not be. Consult with the font vendor for licensing information about a specific font.

Text Formatting Mode

The text rendering in WPF is significantly different from the rendering in older GDI-based applications. A large part of the difference is due to WPF's device-independent display system, but there are also significant enhancements that allow text to appear clearer and crisper, particularly on LCD monitors.

However, WPF text rendering has one well-known shortcoming. At small text sizes, text can become blurry and show undesirable artifacts (such as color fringing around the edges). These problems don't occur with GDI text display, because GDI uses tricks to optimize the clarity of small text. For example, GDI can change the shapes of small letters, adjust their positions, and line up everything on pixel boundaries. These steps cause the typeface to lose its distinctive character, but they make for a better onscreen reading experience when dealing with very small text.

So how can you fix WPF's small-text display problem? The best solution is to scale up your text (on a 96 dpi monitor, the effect should disappear at a text size of about 15 device-independent units) or use a high-dpi monitor that has enough resolution to show sharp text at any size. But because these options often aren't practical, WPF also has the ability to selectively use GDI-like text rendering.

To use GDI-style text rendering, you add the TextOptions.TextFormattingMode attached property to a text-displaying element such as the TextBlock or Label, and set it to Display (rather than the standard value, Ideal). Here's an example:

```
<TextBlock FontSize="12" Margin="5">
This is a Test. Ideal text is blurry at small sizes.
</TextBlock>

<TextBlock FontSize="12" Margin="5" TextOptions.TextFormattingMode="Display">
This is a Test. Display text is crisp at small sizes.
</TextBlock>
```

It's important to remember that the TextFormattingMode property is a solution for small text only. If you use it on larger text (text above 15 points), the text will not be as clear, the spacing will not be as even, and the typeface will not be rendered as accurately. And if you use text in conjunction with a transform (discussed in Chapter 12) that rotates, resizes, or otherwise changes its appearance, you should always use WPF's standard text display mode. That's because the GDI-style optimization for display text is applied before any transforms. After a transform is applied, the result will no longer be aligned on pixel boundaries, and the text will appear blurry.

Mouse Cursors

A common task in any application is to adjust the mouse cursor to show when the application is busy or to indicate how different controls work. You can set the mouse pointer for any element by using the Cursor property, which is inherited from the FrameworkElement class.

Every cursor is represented by a System.Windows.Input.Cursor object. The easiest way to get a Cursor object is to use the static properties of the Cursors class (from the System.Windows.Input namespace). The cursors include all the standard Windows cursors, such as the hourglass, the hand, resizing arrows, and so on. Here's an example that sets the hourglass for the current window:

```
this.Cursor = Cursors.Wait;
```

Now when you move the mouse over the current window, the mouse pointer changes to the familiar swirl.

■ **Note** The properties of the Cursors class draw on the cursors that are defined on the computer. If the user has customized the set of standard cursors, the application you create will use those customized cursors.

If you set the cursor in XAML, you don't need to use the Cursors class directly. That's because the TypeConverter for the Cursor property is able to recognize the property names and retrieve the corresponding Cursor object from the Cursors class. That means you can write markup like this to show the help cursor (a combination of an arrow and a question mark) when the mouse is positioned over a button:

```
<Button Cursor="Help">Help</Button>
```

It's possible to have overlapping cursor settings. In this case, the most specific cursor wins. For example, you could set a different cursor on a button and on the window that contains the button. The button's cursor will be shown when you move the mouse over the button, and the window's cursor will be used for every other region in the window.

However, there's one exception. A parent can override the cursor settings of its children by using the ForceCursor property. When this property is set to true, the child's Cursor property is ignored, and the parent's Cursor property applies everywhere inside.

If you want to apply a cursor setting to every element in every window of an application, the FrameworkElement.Cursor property won't help you. Instead, you need to use the static Mouse.OverrideCursor property, which overrides the Cursor property of every element:

```
Mouse.OverrideCursor = Cursors.Wait;
```

To remove this application-wide cursor override, set the Mouse.OverrideCursor property to null.

Lastly, WPF supports custom cursors without any fuss. You can use both ordinary .cur cursor files (which are essentially small bitmaps) and .ani animated cursor files. To use a custom cursor, you pass the file name of your cursor file or a stream with the cursor data to the constructor of the Cursor object:

```
Cursor customCursor = new Cursor(Path.Combine(applicationDir, "stopwatch.ani");
this.Cursor = customCursor;
```

The Cursor object doesn't directly support the URI resource syntax that allows other WPF elements (such as the Image) to use files that are stored in your compiled assembly. However, it's still quite easy to add a cursor file to your application as a resource and then retrieve it as a stream that you can use to construct a Cursor object. The trick is using the Application.GetResourceStream() method:

```
StreamResourceInfo sri = Application.GetResourceStream(
  new Uri("stopwatch.ani", UriKind.Relative));
Cursor customCursor = new Cursor(sri.Stream);
this.Cursor = customCursor;
```

This code assumes that you've added a file named stopwatch.ani to your project and set its Build Action to Resource. You'll learn more about the GetResourceStream() method in Chapter 7.

Content Controls

A content control is a still more specialized type of control that is able to hold (and display) a piece of content. Technically, a content control is a control that can contain a single nested element. The one-child limit is what differentiates content controls from layout containers, which can hold as many nested elements as you want.

■ **Tip** Of course, you can still pack a lot of content in a single content control. The trick is to wrap everything in a single container, such as a StackPanel or a Grid. For example, the Window class is itself a content control. Obviously, windows often hold a great deal of content, but it's all wrapped in one top-level container (typically a Grid).

As you learned in Chapter 3, all WPF layout containers derive from the abstract Panel class, which gives the support for holding multiple elements. Similarly, all content controls derive from the abstract ContentControl class. Figure 6-1 shows the class hierarchy.

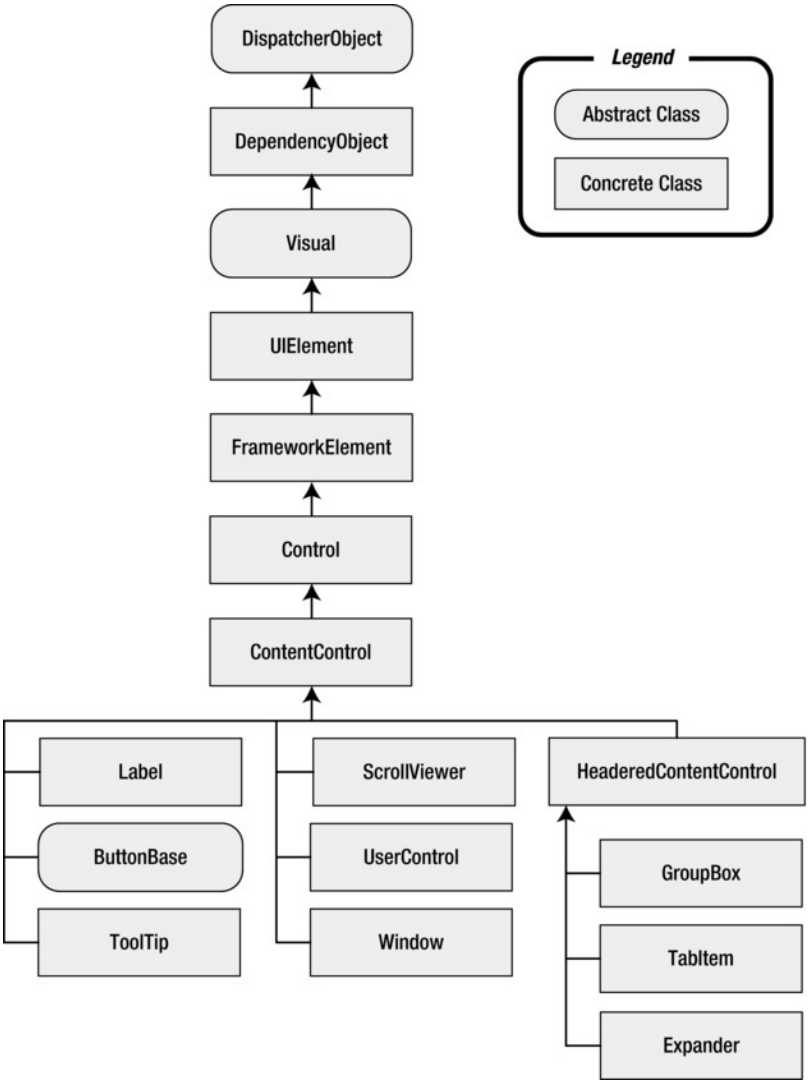

Figure 6-1. *The hierarchy of content controls*

As Figure 6-1 shows, several common controls are content controls, including the Label and the ToolTip. Additionally, all types of buttons are content controls, including the familiar Button, the RadioButton, and the CheckBox. There are also a few more specialized content controls, such as ScrollViewer (which allows you to create a scrollable panel) and UserControl class (which allows you to reuse a custom grouping of controls). The Window class, which is used to represent each window in your application, is itself a content control.

Finally, there is a subset of content controls that goes through one more level of inheritance by deriving from the HeaderedContentControl class. These controls have both a content region and a header region, which can be used to display some sort of title. They include the GroupBox, TabItem (a page in a TabControl), and Expander controls.

■ **Note** Figure 6-1 leaves out just a few elements. It doesn't show the Frame element, which is used for navigation (discussed in Chapter 24), and it omits a few elements that are used inside other controls (such as list box and status bar items).

The Content Property

Whereas the Panel class adds the Children collection to hold nested elements, the ContentControl class adds a Content property, which accepts a single object. The Content property supports any type of object, but it separates objects into two groups and gives each group different treatment:

Objects that don't derive from UIElement: The content control calls ToString() to get the text for these controls and then displays that text.

Objects that derive from UIElement: These objects (which include all the visual elements that are a part of WPF) are displayed inside the content control by using the UIElement.OnRender() method.

■ **Note** Technically, the OnRender() method doesn't draw the object immediately. It simply generates a graphical representation, which WPF paints on the screen as needed.

To understand how this works, consider the humble button. So far, the examples that you've seen that include buttons have simply supplied a string:

```
<Button Margin="3">Text content</Button>
```

This string is set as the button content and displayed on the button surface. However, you can get more ambitious by placing other elements inside the button. For example, you can place an image inside a button by using the Image class:

```
<Button Margin="3">
  <Image Source="happyface.jpg" Stretch="None" />
</Button>
```

Or you could combine text and images by wrapping them all in a layout container such as the StackPanel:

```
<Button Margin="3">
  <StackPanel>
    <TextBlock Margin="3">Image and text button</TextBlock>
    <Image Source="happyface.jpg" Stretch="None" />
    <TextBlock Margin="3">Courtesy of the StackPanel</TextBlock>
  </StackPanel>
</Button>
```

■ **Note** It's acceptable to place text content inside a content control because the XAML parser converts that to a string object and uses that to set the Content property. However, you can't place string content directly in a layout container. Instead, you need to wrap it in a class that derives from UIElement, such as TextBlock or Label.

If you wanted to create a truly exotic button, you could even place other content controls such as text boxes and buttons inside the button (and still nest elements inside these). It's doubtful that such an interface would make much sense, but it's possible. Figure 6-2 shows some sample buttons.

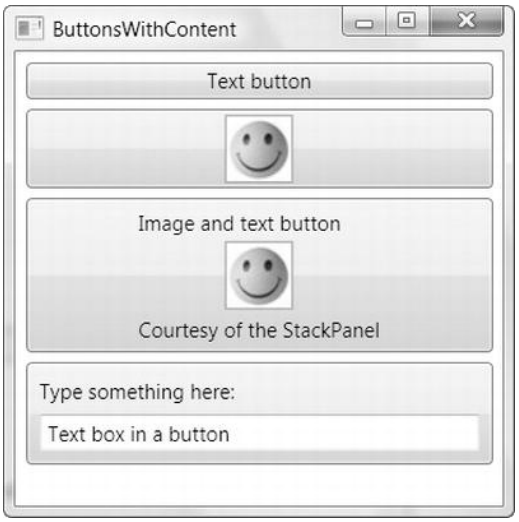

Figure 6-2. *Buttons with different types of nested content*

This is the same content model you saw with windows. Just like the Button class, the Window class allows a single nested element, which can be a piece of text, an arbitrary object, or an element.

■ **Note** One of the few elements that is not allowed inside a content control is the Window. When you create a Window, it checks to see whether it's the top-level container. If it's placed inside another element, the Window throws an exception.

Aside from the Content property, the ContentControl class adds very little. It includes a HasContent property that returns true if there is content in the control, and a ContentTemplate that allows you to build a template telling the control how to display an otherwise unrecognized object. Using a ContentTemplate, you can display non-UIElement-derived objects more intelligently. Instead of just calling ToString() to get a string, you can take various property values and arrange them into more-complex markup. You'll learn more about data templates in Chapter 20.

Aligning Content

In Chapter 3, you learned how to align different controls in a container by using the HorizontalAlignment and VerticalAlignment properties, which are defined in the base FrameworkElement class. However, once a control contains content, you need to consider another level of organization. You need to decide how the content inside your content control is aligned with its borders. This is accomplished by using the HorizontalContentAlignment and VerticalContentAlignment properties.

HorizontalContentAlignment and VerticalContentAlignment support the same values as HorizontalAlignment and VerticalAlignment. That means you can line up content on the inside of any edge (Top, Bottom, Left, or Right), you can center it (Center), or you can stretch it to fill the available space (Stretch). These settings are applied directly to the nested content element, but you can use multiple levels of nesting to create a sophisticated layout. For example, if you nest a StackPanel in a Label element, the Label.HorizontalContentAlignment property determines where the StackPanel is placed, but the alignment and sizing options of the StackPanel and its children will determine the rest of the layout.

In Chapter 3, you also learned about the Margin property, which allows you to add whitespace between adjacent elements. Content controls use a complementary property named Padding, which inserts space between the edges of the control and the edges of the content. To see the difference, compare the following two buttons:

```
<Button>Absolutely No Padding</Button>
<Button Padding="3">Well Padded</Button>
```

The button that has no padding (the default) has its text crowded against the button edge. The button that has a padding of 3 units on each side gets a more respectable amount of breathing space. Figure 6-3 highlights the difference.

Figure 6-3. *Padding the content of the button*

■ **Note** The HorizontalContentAlignment, VerticalContentAlignment, and Padding properties are defined as part of the Control class, not the more specific ContentControl class. That's because some controls that aren't content controls could still have some sort of content. One example is the TextBox—its contained text (stored in the Text property) is adjusted by using the alignment and padding settings you've applied.

The WPF Content Philosophy

At this point, you might be wondering if the WPF content model is really worth all the trouble. After all, you might choose to place an image inside a button, but you're unlikely to embed other controls and entire layout panels. However, there are a few important reasons driving the shift in perspective.

Consider the example shown in Figure 6-2, which includes a simple image button that places an Image element inside the Button control. This approach is less than ideal, because bitmaps are not resolution-independent. On a high-dpi display, the bitmap may appear blurry because WPF must add more pixels by interpolation to make sure the image stays the correct size. More-sophisticated WPF interfaces avoid bitmaps and use a combination of vector shapes to create custom-drawn buttons and other graphical frills (as you'll see in Chapter 12).

This approach integrates nicely with the content control model. Because the Button class is a content control, you are not limited to filling it with a fixed bitmap; instead, you can include other content. For example, you can use the classes in the System.Windows.Shapes namespace to draw a vector image inside a button. Here's an example that creates a button with two diamond shapes (as shown in Figure 6-4):

```
<Button Margin="3">
  <Grid>
    <Polygon Points="100,25 125,0 200,25 125,50"
     Fill="LightSteelBlue" />
    <Polygon Points="100,25 75,0 0,25 75,50"
     Fill="White"/>
  </Grid>
</Button>
```

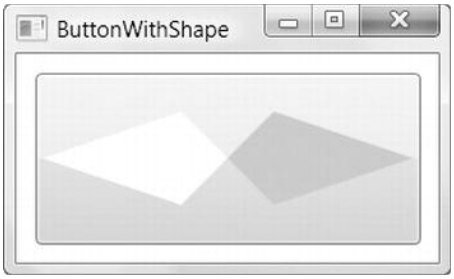

Figure 6-4. *A button with shape content*

Clearly, in this case, the nested content model is simpler than adding extra properties to the Button class to support the different types of content. Not only is the nested content model more flexible, but it also allows the Button class to expose a simpler interface. And because all content controls support content nesting in the same way, there's no need to add different content properties to multiple classes.

In essence, the nested content model is a trade-off. It simplifies the class model for elements because there's no need to use additional layers of inheritance to add properties for different types of content. However, you need to use a slightly more complex *object* model—elements that can be built from other nested elements.

■ **Note** You can't always get the effect you want by changing the content of a control. For example, even though you can place any content in a button, a few details never change, such as the button's shaded background, its rounded border, and the mouse-over effect that makes it glow when you move the mouse pointer over it. However,

another way to change these built-in details is to apply a new control template. Chapter 17 shows how you can change all aspects of a control's look and feel by using a control template.

Labels

The simplest of all content controls is the Label control. Like any other content control, it accepts any single piece of content you want to place inside. But what distinguishes the Label control is its support for *mnemonics*, which are essentially shortcut keys that set the focus to a linked control.

To support this functionality, the Label control adds a single property, named Target. To set the Target property, you need to use a binding expression that points to another control. Here's the syntax you must use:

```
<Label Target="{Binding ElementName=txtA}">Choose _A</Label>
<TextBox Name="txtA"></TextBox>
<Label Target="{Binding ElementName=txtB}">Choose _B</Label>
<TextBox Name="txtB"></TextBox>
```

The underscore in the label text indicates the shortcut key. (If you really *do* want an underscore to appear in your label, you must add two underscores instead.) All mnemonics work with Alt and the shortcut key you've identified. For example, if the user presses Alt+A in this example, the first label transfers focus to the linked control, which is txtA. Similarly, Alt+B takes the user to txtB.

Usually, the shortcut letters are hidden until the user presses Alt, at which point they appear as underlined letters (Figure 6-5). However, this behavior depends on system settings.

■ **Tip** If all you need to do is display content without support for mnemonics, you may prefer to use the more lightweight TextBlock element. Unlike the Label, the TextBlock also supports wrapping through its TextWrapping property.

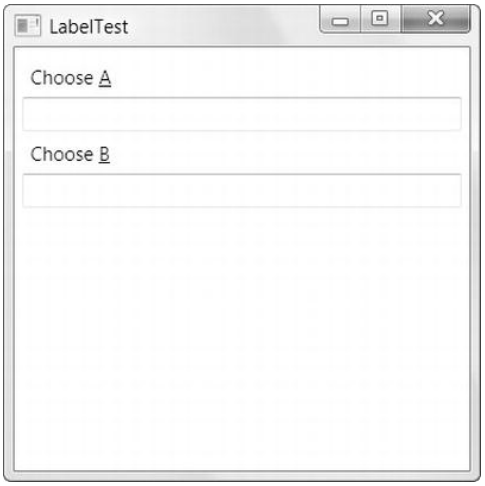

Figure 6-5. *Shortcuts in a label*

Buttons

WPF recognizes three types of button controls: the familiar Button, the CheckBox, and the RadioButton. All of these controls are content controls that derive from ButtonBase.

The ButtonBase class includes only a few members. It defines the Click event and adds support for commands, which allow you to wire buttons to higher-level application tasks (a feat you'll consider in Chapter 9). Finally, the ButtonBase class adds a ClickMode property, which determines when a button fires its Click event in response to mouse actions. The default value is ClickMode.Release, which means the Click event fires when the mouse is clicked and released. However, you can also choose to fire the Click event when the mouse button is first pressed (ClickMode.Press) or, oddly enough, whenever the mouse moves over the button and pauses there (ClickMode.Hover).

■ **Note** All button controls support access keys, which work similarly to mnemonics in the Label control. You add the underscore character to identify the access key. If the user presses Alt and the access key, a button click is triggered.

The Button

The Button class represents the ever-present Windows push button. It adds just two writeable properties, IsCancel and IsDefault:

- *When IsCancel is true*, this button is designated as the cancel button for a window. If you press the Esc key while positioned anywhere on the current window, this button is triggered.
- *When IsDefault is true*, this button is designated as the default button (also known as the *accept button*). Its behavior depends on your current location in the window. If you're positioned on a non-Button control (such as a TextBox, RadioButton, CheckBox, and so on), the default button is given a blue shading, almost as though it has focus. If you press Enter, this button is triggered. However, if you're positioned on another Button control, the current button gets the blue shading, and pressing Enter triggers that button, not the default button.

Many users rely on these shortcuts (particularly the Esc key to close an unwanted dialog box), so it makes sense to take the time to define these details in every window you create. It's still up to you to write the event-handling code for the cancel and default buttons, because WPF won't supply this behavior.

In some cases, it may make sense for the same button to be the cancel button *and* the default button for a window. One example is the OK button in an About box. However, there should be only a single cancel button and a single default button in a window. If you designate more than one cancel button, pressing Esc will simply move the focus to the next default button, but it won't trigger that button. If you have more than one default button, pressing Enter has a somewhat more confusing behavior. If you're on a non-Button control, pressing Enter moves you to the next default button. If you're on a Button control, pressing Enter triggers it.

ISDEFAULT AND ISDEFAULTED

The Button class also includes the horribly confusing IsDefaulted property, which is read-only. IsDefaulted returns true for a default button if another control has focus and that control doesn't accept the Enter key. In this situation, pressing the Enter key will trigger the button.

For example, a TextBox does not accept the Enter key, unless you've set TextBox.AcceptsReturn to true. When a TextBox with an AcceptsReturn value of true has focus, IsDefaulted is false for the default button. When a TextBox with an AcceptsReturns value of false has focus, the default button has IsDefaulted set to true. If this isn't confusing enough, the IsDefaulted property returns false when the button itself has focus, even though hitting Enter at this point will trigger the button.

Although it's unlikely that you'll want to use the IsDefaulted property, this property does allow you to write certain types of style triggers, as you'll see in Chapter 11. If that doesn't interest you, just add it to your list of obscure WPF trivia, which you can use to puzzle your colleagues.

The ToggleButton and RepeatButton

Along with Button, three more classes derive from ButtonBase. These include the following:

- *GridViewColumnHeader* represents the clickable header of a column when you use a grid-based ListView. The ListView is described in Chapter 22.
- *RepeatButton* fires Click events continuously, as long as the button is held down. Ordinary buttons fire one Click event per user click.
- *ToggleButton* represents a button that has two states (pushed or unpushed). When you click a ToggleButton, it stays in its pushed state until you click it again to release it. This is sometimes described as *sticky click* behavior.

Both RepeatButton and ToggleButton are defined in the System.Windows.Controls.Primitives namespace, which indicates they aren't often used on their own. Instead, they're used to build more-complex controls by composition, or extended with features through inheritance. For example, the RepeatButton is used to build the higher-level ScrollBar control (which, ultimately, is a part of the even higher-level ScrollViewer). The RepeatButton gives the arrow buttons at the ends of the scrollbar their trademark behavior—scrolling continues as long as you hold it down. Similarly, ToggleButton is used to derive the more useful CheckBox and RadioButton classes described next.

However, neither RepeatButton nor ToggleButton is an abstract class, so you can use both of them directly in your user interfaces. The ToggleButton is genuinely useful inside a ToolBar, which you'll use in Chapter 25.

The CheckBox

Both the CheckBox and the RadioButton are buttons of a different sort. They derive from ToggleButton, which means they can be switched on or off by the user, hence their "toggle" behavior. In the case of the CheckBox, switching the control on means placing a check mark in it.

The CheckBox class doesn't add any members, so the basic CheckBox interface is defined in the ToggleButton class. Most important, ToggleButton adds an IsChecked property. IsChecked is a nullable Boolean, which means it can be set to true, false, or null. Obviously, true represents a checked box, while false represents an empty one. The null value is a little trickier—it represents an indeterminate state, which is displayed as a shaded box. The indeterminate state is commonly used to represent values that haven't

been set or areas where some discrepancy exists. For example, if you have a check box that allows you to apply bold formatting in a text application, and the current selection includes both bold and regular text, you might set the check box to null to show an indeterminate state.

To assign a null value in WPF markup, you need to use the null markup extension, as shown here:

```
<CheckBox IsChecked="{x:Null}">A check box in indeterminate state</CheckBox>
```

Along with the IsChecked property, the ToggleButton class adds a property named IsThreeState, which determines whether the user is able to place the check box into an indeterminate state. If IsThreeState is false (the default), clicking the check box alternates its state between checked and unchecked, and the only way to place it in an indeterminate state is through code. If IsThreeState is true, clicking the check box cycles through all three possible states.

The ToggleButton class also defines three events that fire when the check box enters specific states: Checked, Unchecked, and Indeterminate. In most cases, it's easier to consolidate this logic into one event handler by handling the Click event that's inherited from ButtonBase. The Click event fires whenever the button changes state.

The RadioButton

The RadioButton also derives from ToggleButton and uses the same IsChecked property and the same Checked, Unchecked, and Indeterminate events. Along with these, the RadioButton adds a single property named GroupName, which allows you to control how radio buttons are placed into groups.

Ordinarily, radio buttons are grouped by their container. That means if you place three RadioButton controls in a single StackPanel, they form a group from which you can select just one of the three. On the other hand, if you place a combination of radio buttons in two separate StackPanel controls, you have two independent groups on your hands.

The GroupName property allows you to override this behavior. You can use it to create more than one group in the same container or to create a single group that spans multiple containers. Either way, the trick is simple—just give all the radio buttons that belong together the same group name.

Consider this example:

```
<StackPanel>
  <GroupBox Margin="5">
    <StackPanel>
      <RadioButton>Group 1</RadioButton>
      <RadioButton>Group 1</RadioButton>
      <RadioButton>Group 1</RadioButton>
      <RadioButton Margin="0,10,0,0" GroupName="Group2">Group 2</RadioButton>
    </StackPanel>
  </GroupBox>
  <GroupBox Margin="5">
    <StackPanel>
      <RadioButton>Group 3</RadioButton>
      <RadioButton>Group 3</RadioButton>
      <RadioButton>Group 3</RadioButton>
      <RadioButton Margin="0,10,0,0" GroupName="Group2">Group 2</RadioButton>
    </StackPanel>
  </GroupBox>
</StackPanel>
```

Here, there are two containers holding radio buttons, but three groups. The final radio button at the bottom of each group box is part of a third group. In this example, it makes for a confusing design, but at

times you might want to separate a specific radio button from the pack in a subtle way without causing it to lose its group membership.

■ **Tip** You don't need to use the GroupBox container to wrap your radio buttons, but it's a common convention. The GroupBox shows a border and gives you a caption that you can apply to your group of buttons.

Tooltips

WPF has a flexible model for *tooltips* (those infamous yellow boxes that pop up when you hover over something interesting). Because tooltips in WPF are content controls, you can place virtually anything inside a tooltip. You can also tweak various timing settings to control how quickly tooltips appear and disappear.

The easiest way to show a tooltip doesn't involve using the ToolTip class directly. Instead, you simply set the ToolTip propertyof your element. The ToolTip property is defined in the FrameworkElement class, so it's available on anything you'll place in a WPF window.

For example, here's a button that has a basic tooltip:

```
<Button ToolTip="This is my tooltip">I have a tooltip</Button>
```

When you hover over this button, the text *This is my tooltip* appears in the familiar yellow box.

If you want to supply more-ambitious tooltip content, such as a combination of nested elements, you need to break the ToolTip property out into a separate element. Here's an example that sets the ToolTip property of a button by using more-complex nested content:

```
<Button>
  <Button.ToolTip>
    <StackPanel>
      <TextBlock Margin="3" >Image and text</TextBlock>
      <Image Source="happyface.jpg" Stretch="None" />
      <TextBlock Margin="3" >Image and text</TextBlock>
    </StackPanel>
  </Button.ToolTip>
  <Button.Content>I have a fancy tooltip</Button.Content>
</Button>
```

As in the previous example, WPF implicitly creates a ToolTip object. The difference is that, in this case, the ToolTip object contains a StackPanel rather than a simple string. Figure 6-6 shows the result.

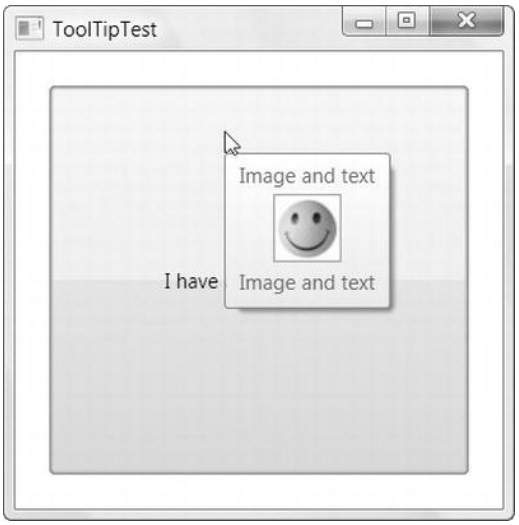

Figure 6-6. *A fancy tooltip*

If more than one tooltip overlaps, the most specific tooltip wins. For example, if you add a tooltip to the StackPanel container in the previous example, this tooltip appears when you hover over an empty part of the panel or a control that doesn't have its own tooltip.

■ **Note** Don't put user-interactive controls in a tooltip because the ToolTip window can't accept focus. For example, if you place a button in a ToolTip control, the button will appear, but it isn't clickable. (If you attempt to click it, your mouse click will just pass through to the window underneath.) If you want a tooltip-like window that can hold other controls, consider using the Popup control instead, which is discussed shortly, in the section named "The Popup."

Setting ToolTip Properties

The previous example shows how you can customize the content of a tooltip, but what if you want to configure other ToolTip-related settings? You have two options. The first technique you can use is to explicitly define the ToolTip object. That gives you the chance to directly set a variety of ToolTip properties.

The ToolTip is a content control, so you can adjust standard properties such as the Background (so it isn't a yellow box), Padding, and Font. You can also modify the members that are defined in the ToolTip class (listed in Table 6-2). Most of these properties are designed to help you place the tooltip exactly where you want it.

Table 6-2. *ToolTip Properties*

Name	Description
HasDropShadow	Determines whether the tooltip has a diffuse black drop shadow that makes it stand out from the window underneath.
Placement	Determines how the tooltip is positioned, using one of the values from the PlacementMode enumeration. The default value is Mouse, which means that the top-left corner of the tooltip is placed relative to the current mouse position. (The actual position of the tooltip may be offset from this starting point based on the HorizontalOffset and VerticalOffset properties.) Other possibilities allow you to place the tooltip by using absolute screen coordinates or place it relative to some element (which you indicate using the PlacementTarget property).
HorizontalOffset and VerticalOffset	Allow you to nudge the tooltip into the exact position you want. You can use positive or negative values.
PlacementTarget	Allows you to place a tooltip relative to another element. In order to use this property, the Placement property must be set to Left, Right, Top, Bottom, or Center. (This is the edge of the element to which the tooltip is aligned.)
PlacementRectangle	Allows you to offset the position of the tooltip. This works in much the same way as the HorizontalOffset and VerticalOffest properties. This property doesn't have an effect if Placement property is set to Mouse.
CustomPopupPlacementCallback	Allows you to position a tooltip dynamically using code. If the Placement property is set to Custom, this property identifies the method that will be called by the ToolTip to get the position where the ToolTip should be placed. Your callback method receives three pieces of information: popupSize (the size of the ToolTip), targetSize (the size of the PlacementTarget, if it's used), and offset (a point that's created based on HorizontalOffset and VerticalOffset properties). The method returns a CustomPopupPlacement object that tells WPF where to place the tooltip.
StaysOpen	Has no effect in practice. The intended purpose of this property is to allow you to create a tooltip that remains open until the user clicks somewhere else. However, the ToolTipService.ShowDuration property overrides the StaysOpen property. As a result, tooltips always disappear after a configurable amount of time (usually about 5 seconds) or when the user moves the mouse away. If you want to create a tooltip-like window that stays open indefinitely, the easiest approach is to use the Popup control.
IsEnabled and IsOpen	Allow you to control the tooltip in code. IsEnabled allows you to temporarily disable a ToolTip. IsOpen allows you to programmatically show or hide a tooltip (or just check whether the tooltip is open).

Using the ToolTip properties, the following markup creates a tooltip that has no drop shadow but uses a transparent red background, which lets the underlying window (and controls) show through:

```
<Button>
  <Button.ToolTip>
    <ToolTip Background="#60AA4030" Foreground="White"
      HasDropShadow="False" >
      <StackPanel>
        <TextBlock Margin="3" >Image and text</TextBlock>
        <Image Source="happyface.jpg" Stretch="None" />
        <TextBlock Margin="3" >Image and text</TextBlock>
      </StackPanel>
    </ToolTip>
  </Button.ToolTip>
  <Button.Content>I have a fancy tooltip</Button.Content>
</Button>
```

In most cases, you'll be happy enough to use the standard tooltip placement, which puts it at the current mouse position. However, the various ToolTip properties give you many more options. Here are some strategies you can use to place a tooltip:

Based on the current position of the mouse: This is the standard behavior, which relies on Placement being set to Mouse. The top-left corner of the tooltip box is lined up with the bottom-left corner of the invisible bounding box around the mouse pointer.

Based on the position of the moused-over element: Set the Placement property to Left, Right, Top, Bottom, or Center, depending on the edge of the element you want to use. The top-left corner of the tooltip box will be lined up with that edge.

Based on the position of another element (or the window): Set the Placement property as if you were lining up the tooltip with the current element. (Use the value Left, Right, Top, Bottom, or Center.) Then choose the element by setting the PlacementTarget property. Remember to use the {Binding ElementName=*Name*} syntax to identify the element you want to use.

With an offset: Use any of the strategies described previously, but set the HorizontalOffset and VerticalOffset properties to add a little extra space.

Using absolute coordinates: Set Placement to Absolute and use the HorizontalOffset and VerticalOffset properties (or the PlacementRectangle) to set some space between the tooltip and the top-left corner of the window.

Using a calculation at runtime: Set Placement to Custom. Set the CustomPopupPlacementCallback property to point to a method that you've created.

Figure 6-7 shows how different placement properties stack up. Note that when lining up a tooltip against an element along the tooltip's bottom or right edge, you'll end up with a bit of extra space. That's because of the way that the ToolTip measures its content.

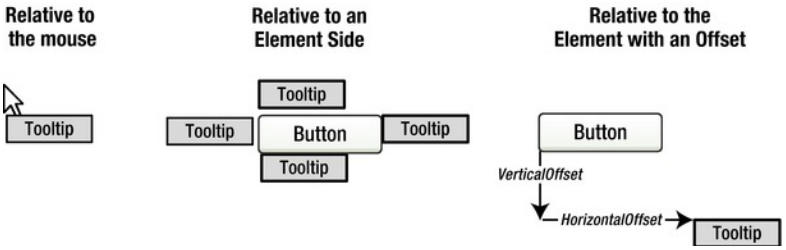

Figure 6-7. *Placing a tooltip explicitly*

Setting ToolTipService Properties

There are some tooltip properties that can't be configured using the properties of the ToolTip class. In this case, you need to use a different class, which is named ToolTipService. ToolTipService allows you to configure the time delays associated with the display of a tooltip. All the properties of the ToolTipService class are attached properties, so you can set them directly in your control tag, as shown here:

```
<Button ToolTipService.InitialShowDelay="1">
  ...
</Button>
```

The ToolTipService class defines many of the same properties as ToolTip. This allows you to use a simpler syntax when you're dealing with text-only tooltips. Rather than adding a nested ToolTip element, you can set everything you need using attributes:

```
<Button ToolTip="This tooltip is aligned with the bottom edge"
  ToolTipService.Placement="Bottom">I have a tooltip</Button>
```

Table 6-3 lists the properties of the ToolTipService class. The ToolTipService class also provides two routed events: ToolTipOpening and ToolTipClosing. You can react to these events to fill a tooltip with just-in-time content or to override the way tooltips work. For example, if you set the handled flag in both events, tooltips will no longer be shown or hidden automatically. Instead, you'll need to show and hide them manually by setting the IsOpen property.

■ **Tip** It makes little sense to duplicate the same tooltip settings for several controls. If you plan to adjust the way tooltips are handled in your entire application, use styles so that your settings are applied automatically, as described in Chapter 11. Unfortunately, the ToolTipService property values are not inherited, which means if you set them at the window or container level, they don't flow through to the nested elements.

Table 6-3. *ToolTipService Properties*

Name	Description
InitialShowDelay	Sets the delay (in milliseconds) before this tooltip is shown when the mouse hovers over the element.
ShowDuration	Sets the amount of time (in milliseconds) that this tooltip is shown before it disappears, if the user does not move the mouse.

BetweenShowDelay	Sets a time window (in milliseconds) during which the user can move between tooltips without experiencing the InitialShowDelay. For example, if BetweenShowDelay is 5000, the user has 5 seconds to move to another control that has a tooltip. If the user moves to another control within that time period, the new tooltip is shown immediately. If the user takes longer, the BetweenShowDelay window expires, and the InitialShowDelay kicks into action. In this case, the second tooltip isn't shown until after the InitialShowDelay period.
ToolTip	Sets the content for the tooltip. Setting ToolTipService.ToolTip is equivalent to setting the FrameworkElement.ToolTip property of an element.
HasDropShadow	Determines whether the tooltip has a diffuse black drop shadow that makes it stand out from the window underneath.
ShowOnDisabled	Determines the tooltip behavior when the associated element is disabled. If true, the tooltip will appear for disabled elements (elements that have their IsEnabled property set to false). The default is false, in which case the tooltip appears only if the associated element is enabled.
Placement, PlacementTarget, PlacementRectangle, and VerticalOffset	Allow you to control the placement of the tooltip. These properties work in the same way as the matching properties of the ToolTipHorizontalOffset class.

The Popup

The Popup control has a great deal in common with the ToolTip, although neither one derives from the other. Like the ToolTip, the Popup can hold a single piece of content, which can include any WPF element. (This content is stored in the Popup.Child property, unlike the ToolTip content, which is stored in the ToolTip.Content property.) Also, like the ToolTip, the content in the Popup can extend beyond the bounds of the window. Lastly, the Popup can be placed using the same placement properties and shown or hidden using the same IsOpen property.

The differences between the Popup and ToolTip are more important. They include the following:

- The Popup is never shown automatically. You must set the IsOpen property for it to appear.
- By default, the Popup.StaysOpen property is set to true, and the Popup does not disappear until you explicitly set its IsOpen property to false. If you set StaysOpen to false, the Popup disappears when the user clicks somewhere else.

■ **Note** A pop-up that stays open can be a bit jarring because it behaves like a separate stand-alone window. If you move the window underneath, the pop-up remains fixed in its original position. You won't witness this behavior with the ToolTip or with a Popup that sets StaysOpen to false, because as soon as you click to move the window, the tooltip or pop-up window disappears.

- The Popup provides a PopupAnimation property that lets you control how it comes into view when you set IsOpen to true. Your options include None (the default), Fade

(the opacity of the pop-up gradually increases), Scroll (the pop-up slides in from the upper-left corner of the window, space permitting), and Slide (the pop-up slides down into place, space permitting). In order for any of these animations to work, you must also set the AllowsTransparency property to true.

- The Popup can accept focus. Thus, you can place user-interactive controls in it, such as a Button. This functionality is one of the key reasons to use the Popup instead of the ToolTip.
- The Popup control is defined in the System.Windows.Controls.Primitives namespace because it is most commonly used as a building block for more-complex controls. You'll find that the Popup is not quite as polished as other controls. Notably, you must set the Background property if you want to see your content, because it won't be inherited from your window and you need to add the border yourself (the Border element works perfectly well for this purpose).

Because the Popup must be shown manually, you may choose to create it entirely in code. However, you can define it just as easily in XAML markup—just make sure to include the Name property so you can manipulate it in code.

Figure 6-8 shows an example. Here, when the user moves the mouse over an underlined word, a pop-up appears with more information and a link that opens an external web browser window.

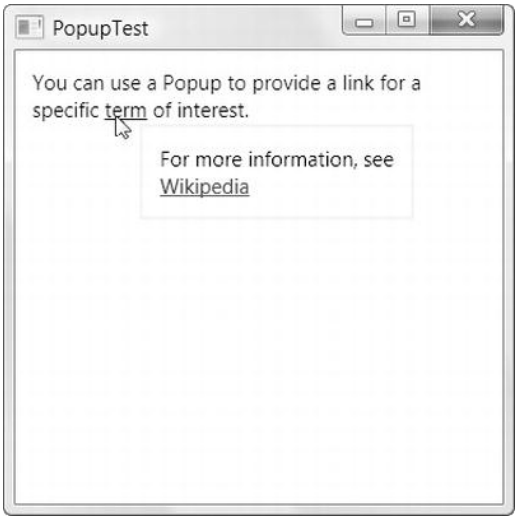

Figure 6-8. *A pop-up with a hyperlink*

To create this window, you need to include a TextBlock with the initial text and a Popup with the additional content that you'll show when the user moves the mouse into the correct place. Technically, it doesn't matter where you define the Popup tag, because it's not associated with any particular control. Instead, it's up to you to set the placement properties to position the Popup in the correct spot. In this example, the Popup appears at the current mouse position, which is the simplest option.

```
<TextBlock TextWrapping="Wrap">You can use a Popup to provide a link for a
 specific <Run TextDecorations="Underline" MouseEnter="run_MouseEnter">term</Run>
 of interest.</TextBlock>
```

```
<Popup Name="popLink" StaysOpen="False" Placement="Mouse" MaxWidth="200"
 PopupAnimation="Slide" AllowsTransparency="True">
  <Border BorderBrush="Beige" BorderThickness="2" Background="White">
    <TextBlock Margin="10" TextWrapping="Wrap">
      For more information, see
      <Hyperlink NavigateUri="http://en.wikipedia.org/wiki/Term"
       Click="lnk_Click">Wikipedia</Hyperlink>
    </TextBlock>
  </Border>
</Popup>
```

This example presents two elements that you might not have seen before. The Run element allows you to apply formatting to a specific part of a TextBlock—it's a piece of flow content that you'll learn about in Chapter 28 when you consider documents. The Hyperlink allows you to provide a clickable piece of text. You'll take a closer look at it in Chapter 24, when you consider page-based applications.

The only remaining details are the relatively trivial code that shows the Popup when the mouse moves over the correct word and the code that launches the web browser when the link is clicked:

```
private void run_MouseEnter(object sender, MouseEventArgs e)
{
    popLink.IsOpen = true;
}

private void lnk_Click(object sender, RoutedEventArgs e)
{
    Process.Start(((Hyperlink)sender).NavigateUri.ToString());
}
```

■ **Note** You can show and hide a Popup by using a trigger—an action that takes place automatically when a specific property hits a specific value. You simply need to create a trigger that reacts when Popup.IsMouseOver is true and sets the Popup.IsOpen property to true. Chapter 11 has the details.

Specialized Containers

Content controls aren't just for basics such as labels, buttons, and tooltips. They also include specialized containers that allow you to shape large portions of your user interface.

In the following sections, you'll meet these more ambitious content controls. You'll start with the ScrollViewer control, which derives directly from ContentControl and provides a virtual surface that lets users scroll around a much larger element. And although the ScrollViewer can only hold a single element (like all content controls), you can place a layout container inside to hold any assortment of elements you need.

Next you'll look at three more controls that go through an extra layer of inheritance: the GroupBox, TabItem, and Expander. All of these controls derive from HeaderedContentControl (which, in turn, derives from ContentControl). The role of HeaderedContentControl is simple—it represents a container that has both single-element content (as stored in the Content property) and a single-element header (as stored in the Header property). The addition of the header is what distinguishes HeaderedContentControl from the

content controls you've seen so far. Once again, you can pack the content into a HeaderedContentControl using a layout container for its content, its header, or both.

The ScrollViewer

Scrolling is a key feature if you want to fit large amounts of content in a limited amount of space. In order to get scrolling support in WPF, you need to wrap the content you want to scroll inside a ScrollViewer.

Although the ScrollViewer can hold anything, you'll typically use it to wrap a layout container. For example, in Chapter 3, you saw an example that used a Grid element to create a three-column display of text, text boxes, and buttons. To make this Grid scrollable, you simply need to wrap the Grid in a ScrollViewer, as shown in this slightly shortened markup:

```
<ScrollViewer>
  <Grid Margin="3,3,10,3">
    <Grid.RowDefinitions>
      ...
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
      ...
    </Grid.ColumnDefinitions>

    <Label Grid.Row="0" Grid.Column="0" Margin="3"
      VerticalAlignment="Center">Home:</Label>
    <TextBox Grid.Row="0" Grid.Column="1" Margin="3"
      Height="Auto" VerticalAlignment="Center"></TextBox>
    <Button Grid.Row="0" Grid.Column="2" Margin="3" Padding="2">
      Browse</Button>
    ...

  </Grid>
</ScrollViewer>
```

The result is shown in Figure 6-9.

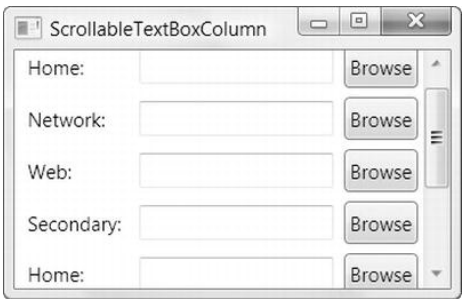

Figure 6-9. *A scrollable window*

If you resize the window in this example so that it's large enough to fit all its content, the scrollbar becomes disabled. However, the scrollbar will still be visible. You can control this behavior by setting the VerticalScrollBarVisibility property, which takes a value from the ScrollBarVisibility enumeration. The

default value of Visible makes sure the vertical scrollbar is always present. Use Auto if you want the scrollbar to appear when it's needed and disappear when it's not. Or use Disabled if you don't want the scrollbar to appear at all.

■ **Note** You can also use Hidden, which is similar to Disabled but subtly different. First, content with a hidden scrollbar is still scrollable. (For example, you can scroll through the content by using the arrow keys.) Second, the content in a ScrollViewer is laid out differently. When you use Disabled, you tell the content in the ScrollViewer that it has only as much space as the ScrollViewer itself. On the other hand, if you use Hidden, you tell the content that it has an infinite amount of space. That means it can overflow and stretch off into the scrollable region. Ordinarily, you'll use Hidden only if you plan to allow scrolling by another mechanism (such as the custom scrolling buttons described next). You'll use Disabled only if you want to temporarily prevent the ScrollViewer from doing anything at all.

The ScrollViewer also supports horizontal scrolling. However, the HorizontalScrollBarVisibility property is Hidden by default. To use horizontal scrolling, you need to change this value to Visible or Auto.

Programmatic Scrolling

To scroll through the window shown in Figure 6-9, you can click the scrollbar with the mouse, you can move over the grid and use a mouse scroll wheel, you can tab through the controls, or you can click somewhere on the blank surface of the grid and use the up and down arrow keys. If this still doesn't give you the flexibility you crave, you can use the methods of the ScrollViewer class to scroll your content programmatically:

- The most obvious are LineUp() and LineDown(), which are equivalent to clicking the arrow buttons on the vertical scrollbar to move up or down once.
- You can also use PageUp() and PageDown(), which scroll an entire screenful up or down and are equivalent to clicking the surface of the scrollbar, above or below the scrollbar thumb.
- Similar methods allow horizontal scrolling, including LineLeft(), LineRight(), PageLeft(), and PageRight().
- Finally, you can use the ScrollTo*Xxx*() methods to go somewhere specific. For vertical scrolling, they include ScrollToEnd() and ScrollToHome(), which take you to the top or bottom of the scrollable content, and ScrollToVerticalOffset(), which takes you to a specific position. There are horizontal versions of the same methods, including ScrollToLeftEnd(), ScrollToRightEnd(), and ScrollToHorizontalOffset().

In the example in Figure 6-10, several custom buttons allow you to move through the ScrollViewer. Each button triggers a simple event handler that uses one of the methods in the previous list.

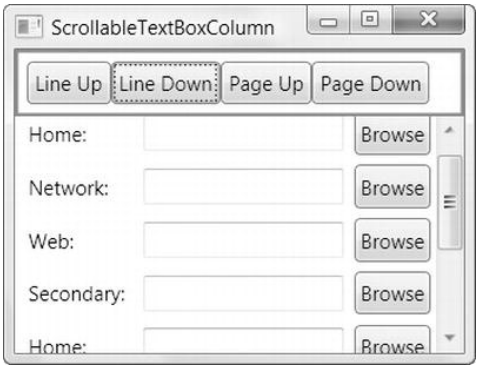

Figure 6-10. *Programmatic scrolling*

Custom Scrolling

The built-in scrolling in the ScrollViewer is quite useful. It allows you to scroll slowly through any content, from a complex vector drawing to a grid of elements. However, one of the most intriguing features of the ScrollViewer is its ability to let its content participate in the scrolling process. Here's how it works:

1. You place a scrollable element inside the ScrollViewer. This is any element that implements IScrollInfo.
2. You tell the ScrollViewer that the content knows how to scroll itself by setting the ScrollViewer.CanContentScroll property to true.
3. When you interact with the ScrollViewer (by using the scrollbar, the mouse wheel, the scrolling methods, and so on), the ScrollViewer calls the appropriate methods on your element by using the IScrollInfo interface. The element then performs its own custom scrolling.

■ **Note** The IScrollInfo interface defines a set of methods that react to different scrolling actions. For example, it includes many of the scrolling methods exposed by the ScrollViewer, such as LineUp(), LineDown(), PageUp(), and PageDown(). It also defines methods that handle the mouse wheel.

Very few elements implement IScrollInfo. One element that does is the StackPanel container. Its IScrollInfo implementation uses *logical scrolling*, which is scrolling that moves from element to element, rather than from line to line.

If you place a StackPanel in a ScrollViewer and you don't set the CanContentScroll property, you get the ordinary behavior. Scrolling up and down moves you a few pixels at a time. However, if you set CanContentScroll to true, each time you click down, you scroll to the beginning of the next element:

```
<ScrollViewer CanContentScroll="True">
  <StackPanel>
    <Button Height="100">1</Button>
    <Button Height="100">2</Button>
    <Button Height="100">3</Button>
    <Button Height="100">4</Button>
```

```
  </StackPanel>
</ScrollViewer>
```

You may or may not find that the StackPanel's logical scrolling system is useful in your application. However, it's indispensable if you want to create a custom panel with specialized scrolling behavior.

The GroupBox

The GroupBox is the simplest of the three controls that derives from HeaderedContentControl. It's displayed as a box with rounded corners and a title. Here's an example (shown in Figure 6-11):

```
<GroupBox Header="A GroupBox Test" Padding="5"
  Margin="5" VerticalAlignment="Top">
  <StackPanel>
    <RadioButton Margin="3">One</RadioButton>
    <RadioButton Margin="3">Two</RadioButton>
    <RadioButton Margin="3">Three</RadioButton>
    <Button Margin="3">Save</Button>
  </StackPanel>
</GroupBox>
```

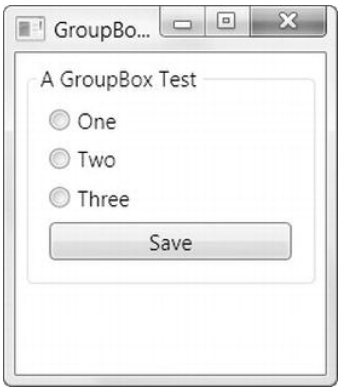

Figure 6-11. *A basic group box*

Notice that the GroupBox still requires a layout container (such as a StackPanel) to arrange its contents. The GroupBox is often used to group small sets of related controls, such as radio buttons. However, the GroupBox has no built-in functionality, so you can use it however you want. (RadioButton objects are grouped by placing them into any panel. A GroupBox is not required, unless you want the rounded, titled border.)

The TabItem

The TabItem represents a page in a TabControl. The only significant member that the TabItem class adds is the IsSelected property, which indicates whether the tab is currently being shown in the TabControl. Here's the markup that's required to create the simple example shown in Figure 6-12:

```
<TabControl Margin="5">
  <TabItem Header="Tab One">
    <StackPanel Margin="3">
      <CheckBox Margin="3">Setting One</CheckBox>
      <CheckBox Margin="3">Setting Two</CheckBox>
      <CheckBox Margin="3">Setting Three</CheckBox>
    </StackPanel>
  </TabItem>
  <TabItem Header="Tab Two">
    ...
  </TabItem>
</TabControl>
```

■ **Tip** You can use the TabStripPlacement property to make the tabs appear on the side of the tab control, rather than in their normal location at the top.

Figure 6-12. *A set of tabs*

As with the Content property, the Header property can accept any type of object. It displays UIElement-derived classes by rendering them and uses the ToString() method for inline text and all other objects. That means you can create a group box or a tab with graphical content or arbitrary elements in its title. Here's an example:

```
<TabControl Margin="5">
  <TabItem>
    <TabItem.Header>
      <StackPanel>
        <TextBlock Margin="3" >Image and Text Tab Title</TextBlock>
        <Image Source="happyface.jpg" Stretch="None" />
      </StackPanel>
```

```
    </TabItem.Header>

    <StackPanel Margin="3">
      <CheckBox Margin="3">Setting One</CheckBox>
      <CheckBox Margin="3">Setting Two</CheckBox>
      <CheckBox Margin="3">Setting Three</CheckBox>
    </StackPanel>
  </TabItem>

  <TabItem Header="Tab Two"></TabItem>
</TabControl>
```

Figure 6-13 shows the somewhat garish result.

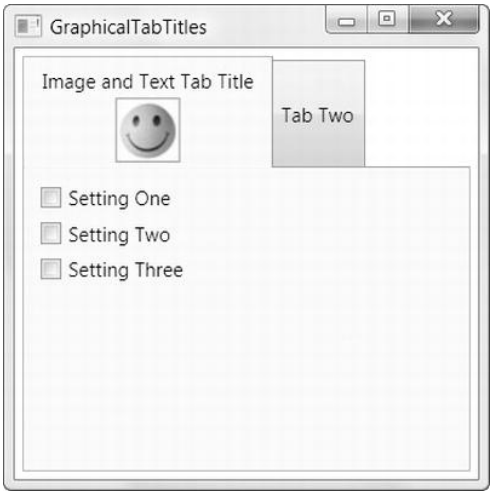

Figure 6-13. *An exotic tab title*

The Expander

The most exotic headered content control is the Expander. It wraps a region of content that the user can show or hide by clicking a small arrow button. This technique is used frequently in online help and on web pages, to allow them to include large amounts of content without overwhelming users with information they don't want to see.

Figure 6-14 shows two views of a window with three expanders. In the version on the left, all three expanders are collapsed. In the version on the right, all the regions are expanded. (Of course, users are free to expand or collapse any combination of expanders individually.)

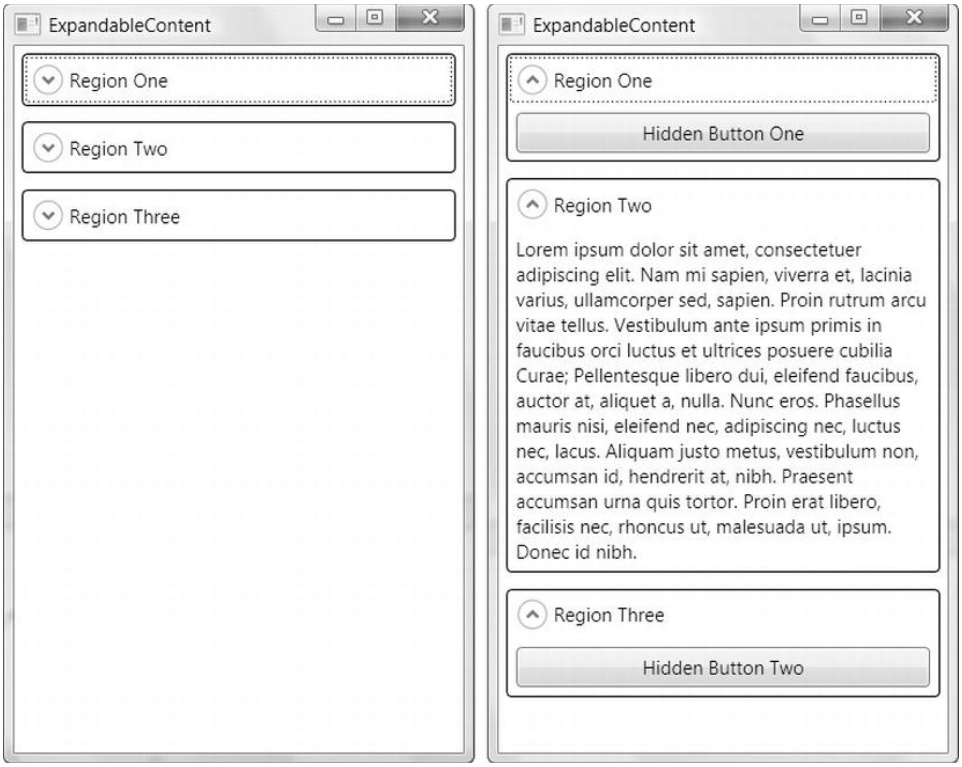

Figure 6-14. *Hiding content with expandable regions*

Using an Expander is extremely simple—you just need to wrap the content you want to make collapsible inside. Ordinarily, each Expander begins collapsed, but you can change this in your markup (or in your code) by setting the IsExpanded property. Here's the markup that creates the example shown in Figure 6-14:

```
<StackPanel>
  <Expander Margin="5" Padding="5" Header="Region One">
    <Button Padding="3">Hidden Button One</Button>
  </Expander>
  <Expander Margin="5" Padding="5" Header="Region Two" >
    <TextBlock TextWrapping="Wrap">
      Lorem ipsum dolor sit amet, consectetuer adipiscing elit ...
    </TextBlock>
  </Expander>
  <Expander Margin="5" Padding="5" Header="Region Three">
    <Button Padding="3">Hidden Button Two</Button>
  </Expander>
</StackPanel>
```

You can also choose in which direction the expander expands. In Figure 6-14, the standard value (Down) is used, but you can also set the ExpandDirection property to Up, Left, or Right. When the Expander is collapsed, the arrow always points in the direction where it will expand.

Life gets a little interesting when using different ExpandDirection values, because the effect on the rest of your user interface depends on the type of container. Some containers, such as the WrapPanel, simply bump other elements out of the way. Others, such as Grid, have the option of using proportional or automatic sizing. Figure 6-15 shows an example with a four-cell grid in various degrees of expansion. In each cell is an Expander with a different ExpandDirection. The columns are sized proportionately, which forces the text in the Expander to wrap. (An autosized column would simply stretch to fit the text, making it larger than the window.) The rows are set to automatic sizing, so they expand to fit the extra content.

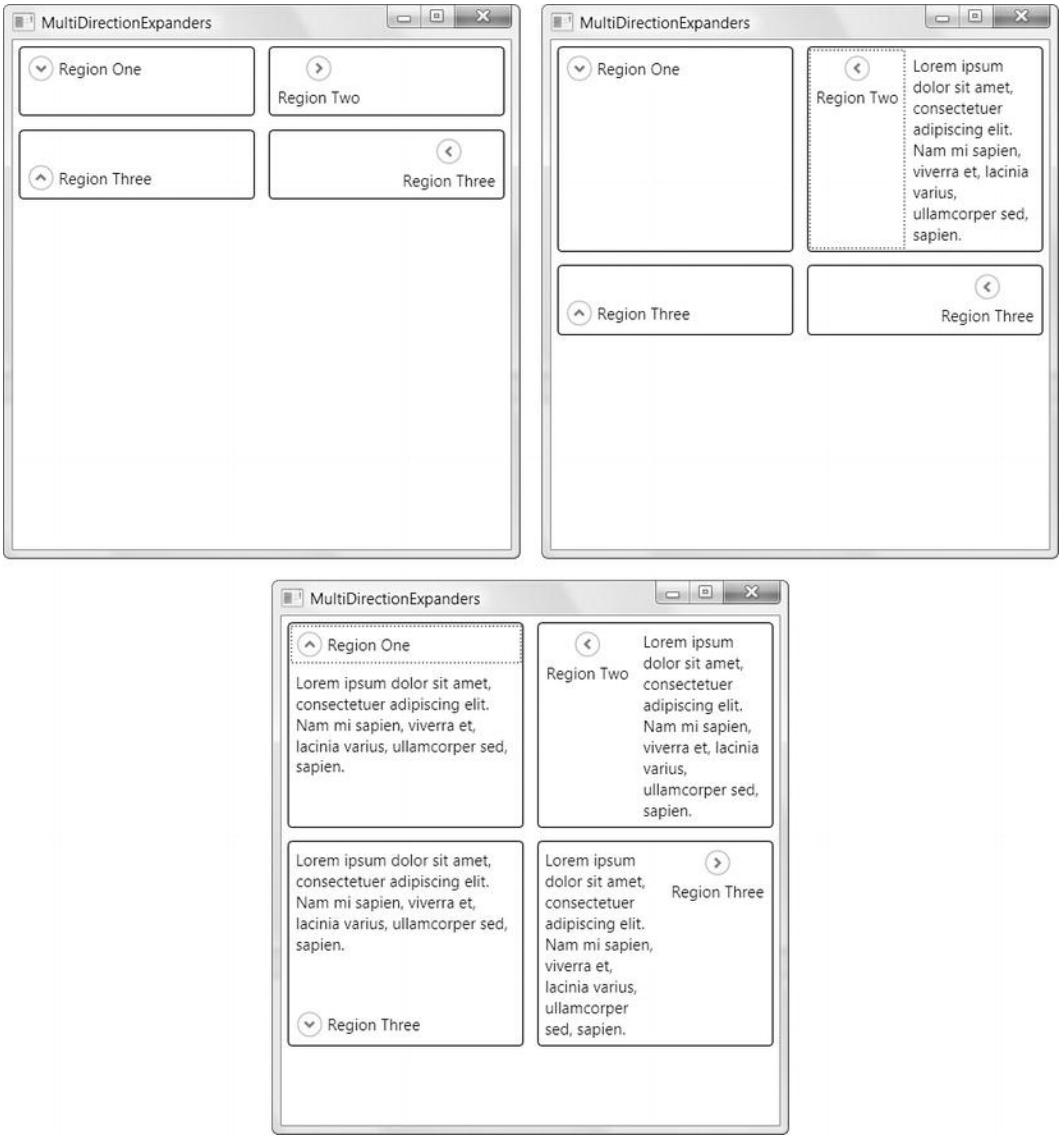

Figure 6-15. *Expanding in different directions*

The Expander is a particularly nice fit in WPF because WPF encourages you to use a flowing layout model that can easily handle content areas that grow or shrink dynamically.

If you need to synchronize other controls with an Expander, you can handle the Expanded and Collapsed events. Contrary to what the naming of these events implies, they fire just *before* the content appears or disappears. This gives you a useful way to implement a lazy load. For example, if the content in an Expander is expensive to create, you might wait until it's shown to retrieve it. Or perhaps you want to update the content just before it's shown. Either way, you can react to the Expanded event to perform your work.

■ **Note** If you like the functionality of the Expander but aren't impressed with the built-in appearance, don't worry. Using the template system in WPF, you can completely customize the expand and collapse arrows so they match the style of the rest of your application. You'll learn how in Chapter 17.

Ordinarily, when you expand an Expander, it grows to fit its content. This may create a problem if your window isn't large enough to fit all the content when everything is expanded. You can use several strategies to handle this problem:

- Set a minimum size for the window (using MinWidth and MinHeight) to make sure it will fit everything even at its smallest.
- Set the SizeToContent property of the window so that it expands automatically to fit the exact dimensions you need when you open or close an Expander. Ordinarily, SizeToContent is set to Manual, but you can use Width or Height to make it expand or contract in either dimension to accommodate its content.
- Limit the size of the Expander by hard-coding its Height and Width. Unfortunately, this is likely to truncate the content that's inside if it's too large.
- Create a scrollable expandable region by using the ScrollViewer.

For the most part, these techniques are quite straightforward. The only one that requires any further exploration is the combination of an Expander and a ScrollViewer. In order for this approach to work, you need to hard-code the size for the ScrollViewer. Otherwise, it will simply expand to fit its content. Here's an example:

```
<Expander Margin="5" Padding="5" Header="Region Two">
  <ScrollViewer Height="50">
    <TextBlock TextWrapping="Wrap">
     ...
    </TextBlock>
  </ScrollViewer>
</Expander>
```

It would be nice to have a system in which an Expander could set the size of its content region based on the available space in a window. However, this would present obvious complexities. (For example, how would space be shared between multiple regions when an Expander expands?) The Grid layout container might seem like a potential solution, but unfortunately, it doesn't integrate well with the Expander. If you try it out, you'll end up with oddly spaced rows that don't update their heights properly when an Expander is collapsed.

Text Controls

WPF includes three text-entry controls: TextBox, RichTextBox, and PasswordBox. The PasswordBox derives directly from Control. The TextBox and RichTextBox controls go through another level and derive from TextBoxBase.

Unlike the content controls you've seen, the text boxes are limited in the type of content they can contain. The TextBox always stores a string (provided by the Text property). The PasswordBox also deals with string content (provided by the Password property), although it uses a SecureString internally to mitigate against certain types of attacks. Only the RichTextBox has the ability to store more-sophisticated content: a FlowDocument that can contain a complex combination of elements.

In the following sections, you'll consider the core features of the TextBox. You'll end by taking a quick look at the security features of the PasswordBox.

■ **Note** The RichTextBox is an advanced control design for displaying FlowDocument objects. You'll learn how to use it when you tackle documents in Chapter 28.

Multiple Lines of Text

Ordinarily, the TextBox control stores a single line of text. (You can limit the allowed number of characters by setting the MaxLength property.) However, in many cases you'll want to create a multiline text box for dealing with large amounts of content. In this case, set the TextWrapping property to Wrap or WrapWithOverflow. Wrap always breaks at the edge of the control, even if it means severing an extremely long word in two. WrapWithOverflow allows some lines to stretch beyond the right edge if the line-break algorithm can't find a suitable place (such as a space or a hyphen) to break the line.

For multiple lines to be visible in a text box, it needs to be sized large enough. Rather than setting a hard-coded height (which won't adapt to different font sizes and may cause layout problems), you can use the handy MinLines and MaxLines properties. MinLines is the minimum number of lines that must be visible in the text box. For example, if MinLines is 2, the text box will grow to be at least two lines tall. If its container doesn't have enough room, part of the text box may be clipped. MaxLines sets the maximum number of lines that will be displayed. Even if a text box expands to fit its container (for example, a proportionally sized Grid row or the last element in a DockPanel), it won't grow beyond this limit.

■ **Note** The MinLines and MaxLines properties have no effect on the amount of content you can place in a text box. They simply help you size the text box. In your code, you can examine the LineCount property to find out exactly how many lines are in a text box.

If your text box supports wrapping, the odds are good that the user can enter more text than can be displayed at once in the visible lines. For this reason, it usually makes sense to add an always-visible or on-demand scrollbar by setting the VerticalScrollBarVisibility property to Visible or Auto. (You can also set the HorizontalScrollBarVisibility property to show a less common horizontal scrollbar.)

You may want to allow the user to enter hard returns in a multiline text box by pressing the Enter key. (Ordinarily, pressing the Enter key in a text box triggers the default button.) To make sure a text box supports the Enter key, set AcceptsReturn to true. You can also set AcceptsTab to allow the user to insert tabs. Otherwise, the Tab key moves to the next focusable control in the tab sequence.

■ **Tip** The TextBox class also includes a host of methods that let you move through the text content programmatically in small or large steps. They include LineUp(), LineDown(), PageUp(), PageDown(), ScrollToHome(), ScrollToEnd(), and ScrollToLine().

Sometimes, you'll create a text box purely for the purpose of displaying text. In this case, set the IsReadOnly property to true to prevent editing. This is preferable to disabling the text box by setting IsEnabled to false, because a disabled text box shows grayed-out text (which is more difficult to read), does not support selection (or copying to the clipboard), and does not support scrolling.

Text Selection

As you already know, you can select text in any text box by clicking and dragging with the mouse or holding down Shift while you move through the text with the arrow keys. The TextBox class also gives you the ability to determine or change the currently selected text programmatically, using the SelectionStart, SelectionLength, and SelectedText properties.

SelectionStart identifies the zero-based position where the selection begins. For example, if you set this property to 10, the first selected character is the 11th character in the text box. SelectionLength indicates the total number of selected characters. (A value of 0 indicates no selected characters.) Finally, the SelectedText property allows you to quickly examine or change the selected text in the text box. You can react to the selection being changed by handling the SelectionChanged event. Figure 6-16 shows an example that reacts to this event and displays the current selection information.

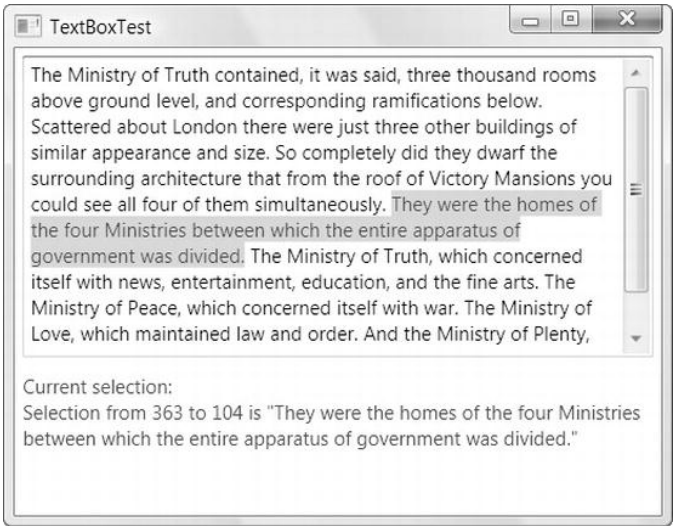

Figure 6-16. *Selecting text*

The TextBox class also includes one property that lets you control its selection behavior: AutoWordSelection. If this is true, the text box selects entire words at a time as you drag through the text.

Another useful feature of the TextBox control is Undo, which allows the user to reverse recent changes. The Undo feature is available programmatically (using the Undo() method), and it's available using the Ctrl+Z keyboard shortcut, as long as the CanUndo property has not been set to false.

■ **Tip** When manipulating text in the text box programmatically, you can use the BeginChange() and EndChange() methods to bracket a series of actions that the TextBox will treat as a single block of changes. These actions can then be undone in a single step.

Spell Checking

The TextBox includes an unusual frill: an integrated spell-check feature, which underlines unrecognized words with a red squiggly line. The user can right-click an unrecognized word and choose from a list of possibilities, as shown in Figure 6-17.

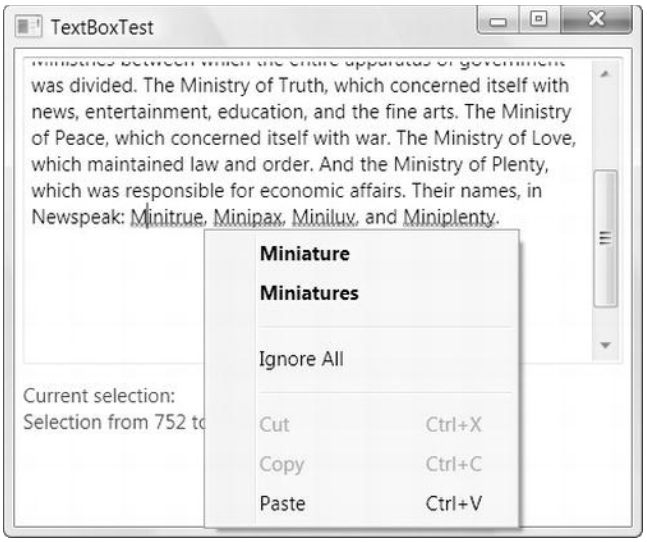

Figure 6-17. *Spell-checking a text box*

To turn on the spell-check functionality for the TextBox control, you simply need to set the SpellCheck.IsEnabled dependency property, as shown here:

```
<TextBox SpellCheck.IsEnabled="True">...</TextBox>
```

The spelling checker is WPF-specific and doesn't depend on any other software (such as Office). The spelling checker determines which dictionary to use based on the input language that's configured for the keyboard. You can override this default by setting the Language property of the TextBox, which is inherited from the FrameworkElement class, or you can set the xml:lang attribute on the <TextBox> element. However, the WPF spelling checker is currently limited to just four languages: English, Spanish, French, and German. You can use the SpellingReform property to set whether post-1990 spelling rule changes are applied to French and German languages.

WPF allows you to customize the dictionary by adding a list of words that will not be treated as errors (and will be used as right-click suggestions, when appropriate). To do so, you must first create a lexicon file, which is nothing more than a text file with the extension .lex. In the lexicon file, you add the list of words. Place each word on a separate line, in any order, as shown here:

```
acantholysis
atypia
bulla
chromonychia
dermatoscopy
desquamation
...
```

In this example, the words are used regardless of the current language setting. However, you can specify that a lexicon should be used only for a specific language by adding a locale ID. Here's how you would specify that the custom words should be used only when the current language is English:

```
#LID 1033
acantholysis
atypia
bulla
chromonychia
dermatoscopy
desquamation
...
```

The other supported locale IDs are 3082 (Spanish), 1036 (French), and 1031 (German).

■ **Note** The custom dictionary feature is not designed to allow you to use additional languages. Instead, it simply augments an already supported language (such as English) with the words you supply. For example, you can use a custom dictionary to recognize proper names or to allow medical terms in a medical application.

Once you've created the lexicon file, make sure the SpellCheck.IsEnabled property is set to true for your TextBox. The final step is to attach a Uri object that points to your custom dictionary, using the SpellCheck.CustomDictionaries property. If you choose to specify it in XAML, as in the following example, you must first import the System namespace so that you can declare a Uri object in markup:

```
<Window xmlns:sys="clr-namespace:System;assembly=system" ... >
```

You can use multiple custom dictionaries at once, as long as you add a Uri object for each one. Each Uri can use a hard-coded path to the file on a local drive or network share. But the safest approach is to use an application resource. For example, if you've added the file CustomWords.lex to a project named SpellTest, and you've set the Build Action of that file to Resource (using the Solution Explorer), you will use markup like this:

```
<TextBox TextWrapping="Wrap" SpellCheck.IsEnabled="True"
 Text="Now the spell checker recognizes acantholysis and offers the right correction
for acantholysi">
  <SpellCheck.CustomDictionaries>
    <sys:Uri>pack://application:,,,/SpellTest;component/CustomWords.lex</sys:Uri>
  </SpellCheck.CustomDictionaries>
</TextBox>
```

The odd `pack://application:,,,/` portion at the beginning of the URI is the pack URI syntax that WPF uses to refer to an assembly resource. You'll take a closer look at it when you consider resources in detail in Chapter 7.

If you need to load the lexicon file from the application directory, the easiest option is to create the URI you need by using code, and add it to the SpellCheck.CustomDictionaries collection when the window is initialized.

The PasswordBox

The PasswordBox looks like a TextBox, but it displays a string of circle symbols to mask the characters it shows. (You can choose a different mask character by setting the PasswordChar property.) Additionally, the PasswordBox does not support the clipboard, so you can't copy the text inside.

Compared to the TextBox class, the PasswordBox has a much simpler, stripped-down interface. Much like the TextBox class, it provides a MaxLength property; Clear(), Paste() and SelectAll() methods; and an event that fires when the text is changed (named PasswordChanged). But that's it. Still, the most important difference between the TextBox and the PasswordBox is on the inside. Although you can set text and read it as an ordinary string by using the Password property, internally the PasswordBox uses a System.Security.SecureString object exclusively.

A SecureString is a text-only object much like the ordinary string. The difference is how it's stored in memory. A SecureString is stored in memory in an encrypted form. The key that's used to encrypt the string is generated randomly and stored in a portion of memory that's never written to disk. The end result is that even if your computer crashes, malicious users won't be able to examine the paging file to retrieve the password data. At best, they will find the encrypted form.

The SecureString class also includes on-demand disposal. When you call SecureString.Dispose(), the in-memory password data is overwritten. This guarantees that all password information has been wiped out of memory and is no longer subject to any kind of exploit. As you would expect, the PasswordBox is conscientious enough to call Dispose() on the SecureString that it stores internally when the control is destroyed.

List Controls

WPF includes many controls that wrap a collection of items, ranging from the simple ListBox and ComboBox that you'll examine here to more specialized controls such as the ListView, the TreeView, and the ToolBar, which are covered in future chapters. All of these controls derive from the ItemsControl class (which itself derives from Control).

The ItemsControl class fills in the basic plumbing that's used by all list-based controls. Notably, it gives you two ways to fill the list of items. The most straightforward approach is to add them directly to the Items collection, using code or XAML. However, in WPF, it's more common to use data binding. In this case, you set the ItemsSource property to the object that has the collection of data items you want to display. (You'll learn more about data binding with a list in Chapter 19.)

The class hierarchy that leads from ItemsControls is a bit tangled. One major branch is the *selectors*, which includes the ListBox, the ComboBox, and the TabControl. These controls derive from Selector and have properties that let you track down the currently selected item (SelectedItem) or its position (SelectedIndex). Separate from these are controls that wrap lists of items but don't support selection in the same way. These include the classes for menus, toolbars, and trees—all of which are ItemsControls but aren't selectors.

In order to unlock most of the features of any ItemsControl, you'll need to use data binding. This is true even if you aren't fetching your data from a database or an external data source. WPF data binding is general enough to work with data in a variety of forms, including custom data objects and collections. But

you won't consider the details of data binding just yet. For now, you'll take only a quick look at the ListBox and ComboBox.

The ListBox

The ListBox class represents a common staple of Windows design—the variable-length list that allows the user to select an item.

■ **Note** The ListBox class also allows multiple selection if you set the SelectionMode property to Multiple or Extended. In Multiple mode, you can select or deselect any item by clicking it. In Extended mode, you need to hold down the Ctrl key to select additional items or the Shift key to select a range of items. In either type of multiple-selection list, you use the SelectedItems collection instead of the SelectedItem property to get all the selected items.

To add items to the ListBox, you can nest ListBoxItem elements inside the ListBox element. For example, here's a ListBox that contains a list of colors:

```
<ListBox>
  <ListBoxItem>Green</ListBoxItem>
  <ListBoxItem>Blue</ListBoxItem>
  <ListBoxItem>Yellow</ListBoxItem>
  <ListBoxItem>Red</ListBoxItem>
</ListBox>
```

As you'll remember from Chapter 2, different controls treat their nested content in different ways. The ListBox stores each nested object in its Items collection.

The ListBox is a remarkably flexible control. Not only can it hold ListBoxItem objects, but it can also host any arbitrary element. This works because the ListBoxItem class derives from ContentControl, which gives it the ability to hold a single piece of nested content. If that piece of content is a UIElement-derived class, it will be rendered in the ListBox. If it's some other type of object, the ListBoxItem will call ToString() and display the resulting text.

For example, if you decided you want to create a list with images, you could create markup like this:

```
<ListBox>
  <ListBoxItem>
    <Image Source="happyface.jpg"></Image>
  </ListBoxItem>
  <ListBoxItem>
    <Image Source="happyface.jpg"></Image>
  </ListBoxItem>
</ListBox>
```

The ListBox is intelligent enough to create the ListBoxItem objects it needs implicitly. That means you can place your objects directly inside the ListBox element. Here's a more ambitious example that uses nested StackPanel objects to combine text and image content:

```
<ListBox>
  <StackPanel Orientation="Horizontal">
    <Image Source="happyface.jpg"  Width="30" Height="30"></Image>
    <Label VerticalContentAlignment="Center">A happy face</Label>
  </StackPanel>
```

```
  <StackPanel Orientation="Horizontal">
    <Image Source="redx.jpg" Width="30" Height="30"></Image>
    <Label VerticalContentAlignment="Center">A warning sign</Label>
  </StackPanel>
  <StackPanel Orientation="Horizontal">
    <Image Source="happyface.jpg"  Width="30" Height="30"></Image>
    <Label VerticalContentAlignment="Center">A happy face</Label>
  </StackPanel>
</ListBox>
```

In this example, the StackPanel becomes the item that's wrapped by the ListBoxItem. This markup creates the rich list shown in Figure 6-18.

Figure 6-18. *A list of images*

■ **Note** One flaw in the current design is that the text color doesn't change when the item is selected. This isn't ideal, because it's difficult to read the black text with a blue background. To solve this problem, you need to use a data template, as described in Chapter 20.

This ability to nest arbitrary elements inside list box items allows you to create a variety of list-based controls without needing to use other classes. For example, the Windows Forms toolkit includes a CheckedListBox class that's displayed as a list with a check box next to every item. No such specialized class is required in WPF because you can quickly build one by using the standard ListBox:

```
<ListBox Name="lst" SelectionChanged="lst_SelectionChanged"
  CheckBox.Click="lst_SelectionChanged">
  <CheckBox Margin="3">Option 1</CheckBox>
  <CheckBox Margin="3">Option 2</CheckBox>
</ListBox>
```

There's one caveat to be aware of when you use a list with different elements inside. When you read the SelectedItem value (and the SelectedItems and Items collections), you won't see ListBoxItem objects; instead, you'll see whatever objects you placed in the list. In the CheckedListBox example, that means SelectedItem provides a CheckBox object.

For example, here's some code that reacts when the SelectionChanged event fires. It then gets the currently selected CheckBox and displays whether that item has been selected:

```
private void lst_SelectionChanged(object sender, SelectionChangedEventArgs e)
{
    if (lst.SelectedItem == null) return;
    txtSelection.Text = String.Format(
      "You chose item at position {0}.\r\nChecked state is {1}.",
      lst.SelectedIndex,
      ((CheckBox)lst.SelectedItem).IsChecked);
}
```

■ **Tip** If you want to find the current selection, you can read it directly from the SelectedItem or SelectedItems property, as shown here. If you want to determine which item (if any) was *unselected*, you can use the RemovedItems property of the SelectionChangedEventArgs object. Similarly, the AddedItems property tells you which items were added to the selection. In single-selection mode, one item is always added and one item is always removed whenever the selection changes. In multiple or extended mode, this isn't necessarily the case.

In the following code snippet, similar code loops through the collection of items to determine which ones are selected. (You could write similar code that loops through the collection of selected items in a multiple-selection list with check boxes.)

```
private void cmd_ExamineAllItems(object sender, RoutedEventArgs e)
{
    StringBuilder sb = new StringBuilder();
    foreach (CheckBox item in lst.Items)
    {
        if (item.IsChecked == true)
        {
            sb.Append(item.Content);
            sb.Append(" is checked.");
            sb.Append("\r\n");
        }
    }
    txtSelection.Text = sb.ToString();
}
```

Figure 6-19 shows the list box that uses this code.

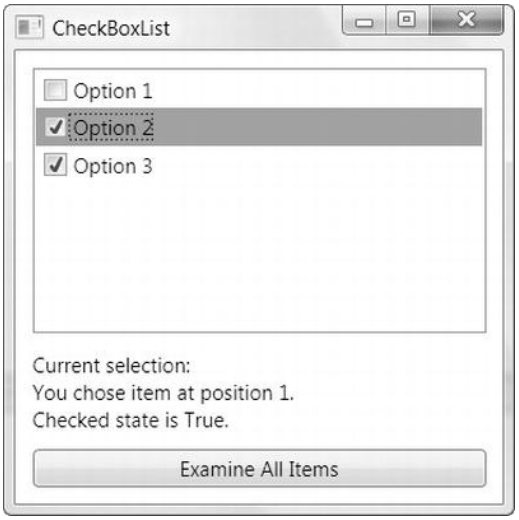

Figure 6-19. *A check box list*

When manually placing items in a list, it's up to you whether you want to insert the items directly or explicitly wrap each one in a ListBoxItem object. The second approach is often cleaner, albeit more tedious. The most important consideration is to be consistent. For example, if you place StackPanel objects in your list, the ListBox.SelectedItem object will be a StackPanel. If you place StackPanel objects wrapped by ListBoxItem objects, the ListBox.SelectedItem object will be a ListBoxItem, so code accordingly.

The ListBoxItem offers a little extra functionality from what you get with directly nested objects. Namely, it defines an IsSelected property that you can read (or set) and a Selected and Unselected event that tells you when that item is highlighted. However, you can get similar functionality by using the members of the ListBox class, such as the SelectedItem (or SelectedItems) property, and the SelectionChanged event.

Interestingly, there's a technique to retrieve a ListBoxItem wrapper for a specific object when you use the nested object approach. The trick is the often overlooked ContainerFromElement() method. Here's the code that uses this technique to check whether the first item is selected in a list:

```
ListBoxItem item = (ListBoxItem)lst.ContainerFromElement(
  (DependencyObject)lst.SelectedItems[0]);
MessageBox.Show("IsSelected: " + item.IsSelected.ToString());
```

The ComboBox

The ComboBox is similar to the ListBox control. It holds a collection of ComboBoxItem objects, which are created either implicitly or explicitly. As with the ListBoxItem, the ComboBoxItem is a content control that can contain any nested element.

The key difference between the ComboBox and ListBox classes is the way they render themselves in a window. The ComboBox control uses a drop-down list, which means only one item can be selected at a time.

If you want to allow the user to type text in the combo box to select an item, you must set the IsEditable property to true, and you must make sure you are storing ordinary text-only ComboBoxItem objects or an object that provides a meaningful ToString() representation. For example, if you fill an

editable combo box with Image objects, the text that appears in the upper portion is simply the fully qualified Image class name, which isn't much use.

One limitation of the ComboBox is the way it sizes itself when you use automatic sizing. The ComboBox widens itself to fit its content, which means that it changes size as you move from one item to the next. Unfortunately, there's no easy way to tell the ComboBox to take the size of its largest contained item. Instead, you may need to supply a hard-coded value for the Width property, which isn't ideal.

Range-Based Controls

WPF includes three controls that use the concept of a *range*. These controls take a numeric value that falls between a specific minimum and maximum value. These controls—ScrollBar, ProgressBar, and Slider—all derive from the RangeBase class (which itself derives from the Control class). But although they share an abstraction (the range), they work quite differently.

The RangeBase class defines the properties shown in Table 6-4.

Table 6-4. *Properties of the RangeBase Class*

Name	Description
Value	The current value of the control (which must fall between the minimum and maximum). By default, it starts at 0. Contrary to what you might expect, Value isn't an integer—it's a double, so it accepts fractional values. You can react to the ValueChanged event if you want to be notified when the value is changed.
Maximum	The upper limit (the largest allowed value).
Minimum	The lower limit (the smallest allowed value).
SmallChange	The amount the Value property is adjusted up or down for a small change. The meaning of a "small change" depends on the control (and may not be used at all). For the ScrollBar and Slider, this is the amount the value changes when you use the arrow keys. For the ScrollBar, you can also use the arrow buttons at either end of the bar.
LargeChange	The amount the Value property is adjusted up or down for a large change. The meaning of a "large change" depends on the control (and may not be used at all). For the ScrollBar and Slider, this is the amount the value changes when you use the Page Up and Page Down keys or when you click the bar on either side of the thumb (which indicates the current position).

Ordinarily, there's no need to use the ScrollBar control directly. The higher-level ScrollViewer control, which wraps two ScrollBar controls, is typically much more useful. The Slider and ProgressBar are more practical, and are often useful on their own.

The Slider

The Slider is a specialized control that's occasionally useful—for example, you might use it to set numeric values when the number itself isn't particularly significant. For example, it makes sense to set the volume in a media player by dragging the thumb in a slider bar from side to side. The general position of the thumb indicates the relative loudness (normal, quiet, or loud), but the underlying number has no meaning to the user.

The key Slider properties are defined in the RangeBase class. Along with these, you can use all the properties listed in Table 6-5.

Table 6-5. *Additional Properties in the Slider Class*

Name	Description
Orientation	Switches between a vertical and a horizontal slider.
Delay and Interval	Control how fast the thumb moves along the track when you click and hold down either side of the slider. Both are millisecond values. The Delay is the time before the thumb moves one (small change) unit after you click, and the Interval is the time before it moves again if you continue holding down the mouse button.
TickPlacement	Determines where the tick marks appear. (*Tick marks* are notches that appear near the bar to help you visualize the scale.) By default, TickPlacement is set to None, and no tick marks appear. If you have a horizontal slider, you can place the tick marks above (TopLeft) or below (BottomRight) the track. With a vertical slider, you can place them on the left (TopLeft) and right (BottomRight). (The TickPlacement names are a bit confusing because two values cover four possibilities, depending on the orientation of the slider.)
TickFrequency	Sets the interval between ticks, which determines how many ticks appear. For example, you could place them every 5 numeric units, every 10, and so on.
Ticks	If you want to place ticks in specific, irregular positions, you can use the Ticks collection. Simply add one number (as a double) to this collection for each tick mark. For example, you could place ticks at the positions 1, 1.5, 2, and 10 on the scale by adding these numbers.
IsSnapToTickEnabled	If true, when you move the slider, it automatically snaps into place, jumping to the nearest tick mark. The default is false.
IsSelectionRangeEnabled	If true, you can use a selection range to shade in a portion of the slider bar. You set the position selection range by using the SelectionStart and SelectionEnd properties. The selection range has no intrinsic meaning, but you can use it for whatever purpose makes sense. For example, media players sometimes use a shaded background bar to indicate the download progress for a media file.

Figure 6-20 compares Slider controls with different tick settings.

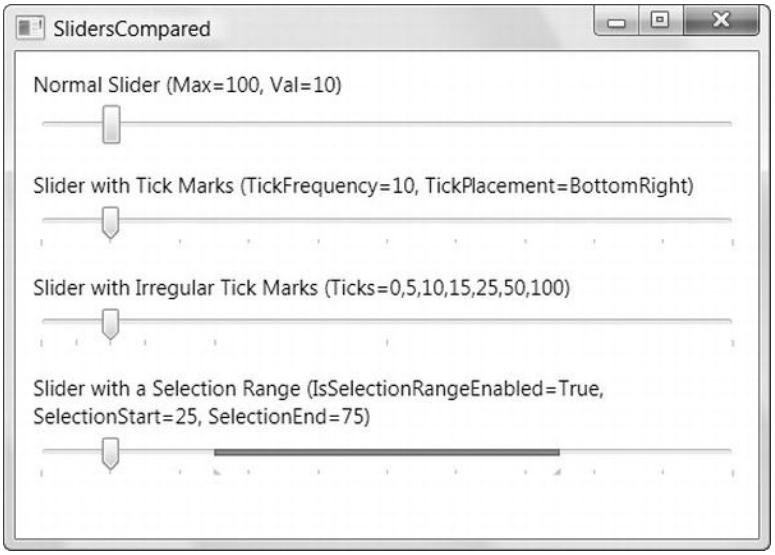

Figure 6-20. *Adding ticks to a slider*

The ProgressBar

The ProgressBar indicates the progress of a long-running task. Unlike the slider, the ProgressBar isn't user-interactive. Instead, it's up to your code to periodically increment the Value property. (Technically speaking, WPF rules suggest that the ProgressBar shouldn't be a control because it doesn't respond to mouse actions or keyboard input.) The ProgressBar has a minimum height of four device-independent units. It's up to you to set the Height property (or put it in the appropriate fixed-size container) if you want to see a larger, more traditional bar.

One neat trick that you can perform with the ProgressBar is using it to show a long-running status indicator, even if you don't know how long the task will take. Interestingly (and oddly), you do this by setting the IsIndeterminate property to true:

```
<ProgressBar Height="18"  Width="200" IsIndeterminate="True"></ProgressBar>
```

When setting IsIndeterminate, you no longer use the Minimum, Maximum, and Value properties. Instead, this ProgressBar shows a periodic green pulse that travels from left to right, which is the universal Windows convention indicating that there's work in progress. This sort of indicator makes sense in an application's status bar. For example, you could use it to indicate that you're contacting a remote server for information.

Date Controls

WPF includes two date controls: the Calendar and the DatePicker. Both are designed to allow the user to choose a single date.

The Calendar control displays a calendar that's similar to what you see in the Windows operating system (for example, when you configure the system date). It shows a single month at a time and allows you to step through from month to month (by clicking the arrow buttons) or jump to a specific month (by clicking the month header to view an entire year, and then clicking the month).

The DatePicker requires less space. It's modeled after a simple text box, which holds a date string in long or short date format. The DatePicker provides a drop-down arrow that, when clicked, pops open a full calendar view that's identical to that shown by the Calendar control. This pop-up is displayed on top of any other content, just like a drop-down combo box.

Figure 6-21 shows the two display modes that the Calendar supports, as well as the two date formats that the DatePicker allows.

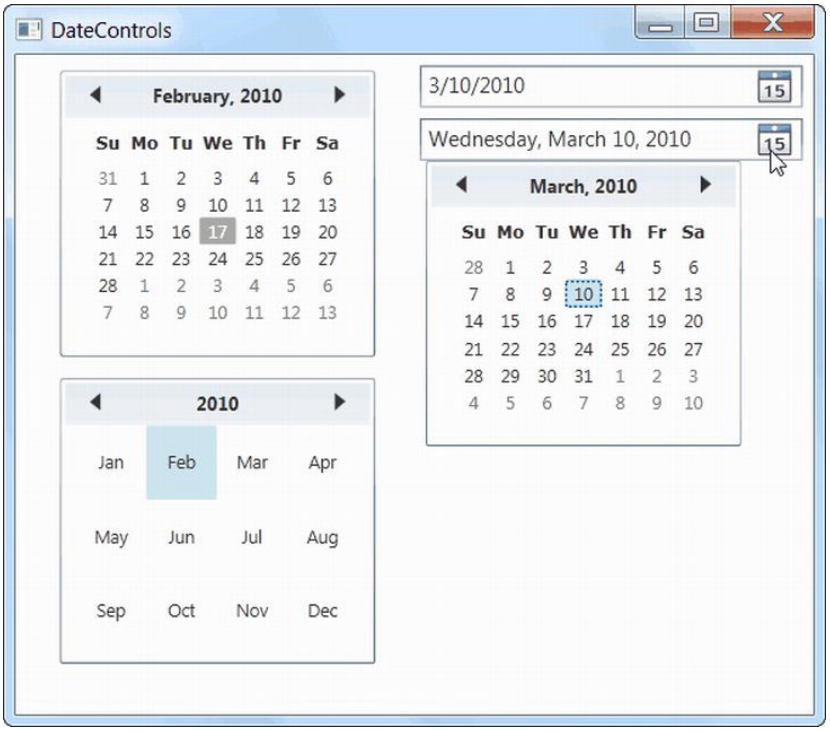

Figure 6-21. *The Calendar and DatePicker*

The Calendar and DatePicker include properties that allow you to determine which dates are shown and which dates are selectable (provided they fall in a contiguous range). Table 6-6 lists the properties you can use.

Table 6-6. *Properties of the Calendar and DatePicker Classes*

Property	Description
DisplayDateStart and DisplayDateEnd	Set the range of dates that are displayed in the calendar view, from the first, earliest date (DisplayDateStart) to the last, most recent date (DisplayDateEnd). The user won't be able to navigate to months that don't have any displayable dates. To show all dates, set DisplayDateStart to DateTime.MinValue and DisplayDateEnd to DateTime.MaxValue.
BlackoutDates	Holds a collection of dates that will be disabled in the calendar and won't be selectable. If these dates are not in the range of displayed dates, or if one of these dates is already selected, you'll receive an exception. To prevent selection of any date in the past, call the BlackoutDates.AddDatesInPast() method.
SelectedDate	Provides the selected date as a DateTime object (or a null value if no date is selected). It can be set programmatically, by the user clicking the date in the calendar, or by the user typing in a date string (in the DatePicker). In the calendar view, the selected date is marked by a shaded square, which is visible only when the date control has focus.
SelectedDates	Provides the selected dates as a collection of DateTime objects. This property is supported by the Calendar, and it's useful only if you've changed the SelectionMode property to allow multiple date selection.
DisplayDate	Determines the date that's displayed initially in the calendar view (using a DateTime object). If it's a null, the SelectedDate is shown. If DisplayDate and SelectedDate are both null, the current date is used. The display date determines the initial month page of the calendar view. When the date control has focus, a square outline is displayed around the appropriate day in that month (which is different from the shaded square used for the currently selected date).
FirstDayOfWeek	Determines the day of the week that will be displayed at the start of each calendar row, in the leftmost position.
IsTodayHighlighted	Determines whether the calendar view uses highlighting to point out the current date.
DisplayMode (Calendar only)	Determines the initial display month of the calendar. If set to Month, the Calendar shows the standard single-month view. If set to Year, the Calendar shows the months in the current year (similar to when the user clicks the month header). After the user clicks a month, the Calendar shows the full calendar view for that month.
SelectionMode (Calendar only)	Determines the type of date selections that are allowed. The default is SingleDate, which allows a single date to be selected. Other options include None (selection is disabled entirely), SingleRange (a contiguous group of dates can be selected), and MultipleRange (any combination of dates can be selected). In SingleRange or MultipleRange modes, the user can drag to select multiple dates, or click while holding down the Ctrl key. You can use the SelectedDates property to get a collection with all the selected dates.
IsDropDownOpen (DatePicker only)	Determines whether the calendar view drop-down is open in the DatePicker. You can set this property programmatically to show or hide the calendar.
SelectedDateFormat (DatePicker only)	Determines how the selected date will be displayed in the text part of the DatePicker. You can choose Short or Long. The actual display format is based on the client computer's regional settings. For example, if you use Short, the date might be rendered in the yyyy/mm/dd format or dd/mm/yyyy. The long format generally includes the month and day names.

The date controls also provide a few events. Most useful is SelectedDateChanged (in the DatePicker) or the similar SelectedDatesChanged (in the Calendar), which adds support for multiple date selection. You can react to these events to reject specific date selections, such as dates that fall on a weekend:

```
private void Calendar_SelectedDatesChanged (object sender,
  CalendarDateChangedEventArgs e)
{
    // Check all the newly added items.
    foreach (DateTime selectedDate in e.AddedItems)
    {
        if ((selectedDate.DayOfWeek == DayOfWeek.Saturday) ||
          (selectedDate.DayOfWeek == DayOfWeek.Sunday))
        {
            lblError.Text = "Weekends are not allowed";

            // Remove the selected date.
            ((Calendar)sender).SelectedDates.Remove(selectedDate);
        }
    }
}
```

You can try this out with a Calendar that supports single or multiple selection. If it supports multiple selection, try dragging the mouse over an entire week of dates. All the dates will remain highlighted except for the disallowed weekend dates, which will be unselected automatically.

The Calendar also adds a DisplayDateChanged event (when the user browses to a new month). The DatePicker adds CalendarOpened and CalendarClosed events (which fire when the calendar drop-down is displayed and closed) and a DateValidationError event (which fires when the user types a value in the text-entry portion that can't be interpreted as a valid date). Ordinarily, invalid values are discarded when the user opens the calendar view, but here's an option that fills in some text to inform the user of the problem:

```
private void DatePicker_DateValidationError(object sender,
  DatePickerDateValidationErrorEventArgs e)
{
     lblError.Text = "'" + e.Text +
       "' is not a valid value because " + e.Exception.Message;
}
```

The Last Word

In this chapter, you took a tour of the fundamental WPF controls, including basic ingredients such as labels, buttons, text boxes, and lists. Along the way, you learned about some important WPF concepts that underlie the control model, such as brushes, fonts, and the content model. Although most WPF controls are quite easy to use, developers who have this additional understanding—and know how all the different branches of WPF elements relate together—will have an easier time creating well-designed windows.