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 abilities. They include the Label, Button, and ToolTip 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.

■ **What's New** Although WPF is continually making minor refinements to its control classes, WPF 4 adds just a few significant changes to the controls covered in this chapter. The most important is WPF 4's ability to display clearer text at small sizes (see the section "Text Formatting Mode"). WPF 4 also gives the TextBox control the ability to use a custom spell-check dictionary, and it adds two entirely new date controls: the Calendar and DatePicker.

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) using the Background and Foreground properties.

It's natural to expect that the Background and Foreground properties would use color objects, as they do in a Windows Forms application. 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 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, it's possible that other devices (such as printers) might support a richer range of colors. For this reason, an alternative scRGB standard has been created. This standard represents each color component (alpha, red, green, and blue) using 64-bit values.

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 in between 0 and 255 in RGB are represented as decimal values in 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 creates the same color as in the previous code snippets using #aarrggbb notation:

```
<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) 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 using the ButtonChrome decorator. However, you can give a button a new face (with a border of your choosing) 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 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 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 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 feature is similar to the Windows Forms concept of *ambient properties*, but the underlying plumbing is different. It 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 using the static SystemFontFamilies collection of the System.Windows.Media.Fonts class. Here's an example that uses it 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 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 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. Furthermore, Visual Studio doesn't currently provide design support for embedded fonts (meaning your control text won't appear in the correct font until you run your application). 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 using Microsoft's free font properties extension utility, which is available at `http://www.microsoft.com/typography/TrueTypeProperty21.mspx`. Once 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 (like color fringing around the edges). These problems don't occur with GDI text display, because GDI uses a number of 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 on-screen 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 4 introduces a new feature: 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 like 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. Once 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 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 hourglass icon (in Windows XP) or the swirl (in Windows Vista and Windows 7).

■ **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 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 controls 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.

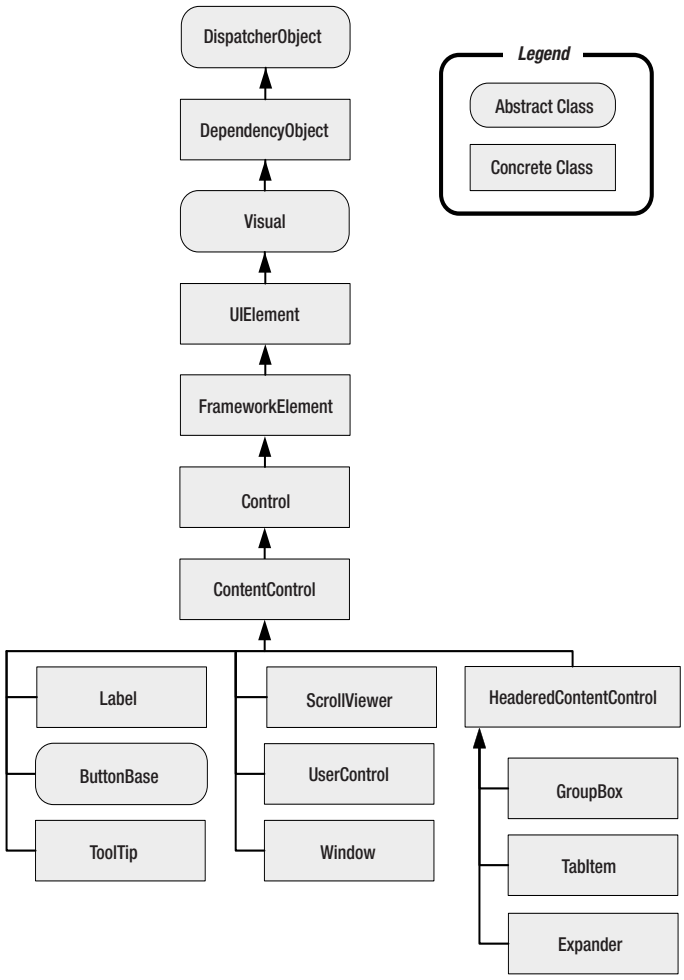

Figure 6-1. *The hierarchy of content controls*

As Figure 6-1 shows, several common controls are actually 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 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 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 like 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.

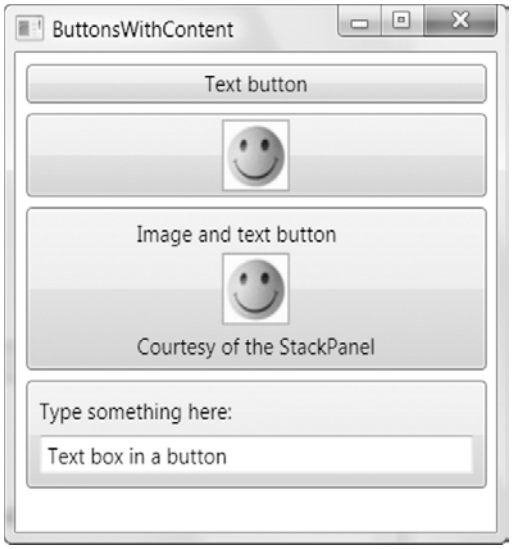

Figure 6-2. *Buttons with different types of nested content*

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 if 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 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 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 there may be controls that aren't content controls but still have some sort of content. One example is the TextBox—its contained text (stored in the Text property) is adjusted 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>
```

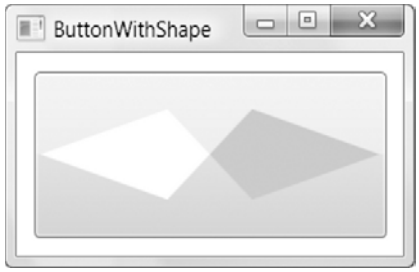

Figure 6-4. *A button with shape content*

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. (Windows Forms ran into this issue in .NET 2.0, while enhancing the Button and Label class to better support images and mixed image and text content.)

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 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.

■ **Note** If you've programmed with Windows Forms, you're probably used to using the ampersand (&) character to identify a shortcut key. XAML uses the underscore instead because the ampersand character can't be entered directly in XML; instead, you need to use the clunkier character entity & in its place.

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 mouse 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 Escape 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 Escape 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 Escape 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.