To understand how you can bind an element to another element, consider the simple window shown in Figure 8-1. It contains two controls: a Slider and a TextBlock with a single line of text. If you pull the thumb in the slider to the right, the font size of the text is increased immediately. If you pull it to the left, the font size is reduced.

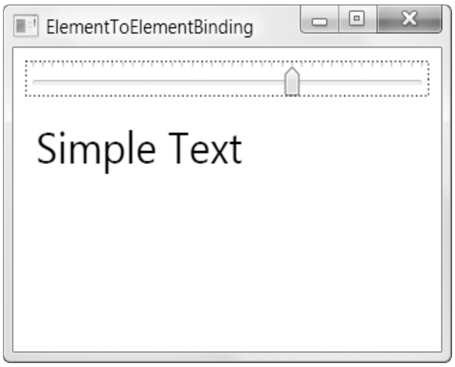

***Figure 8-1.** Linked controls through data binding*

Clearly, it wouldn't be difficult to create this behavior using code. You would simply react to the Slider.ValueChanged event and copy the current value from the slider to the TextBlock. However, data binding makes it even easier.

■ **Tip** Data binding also has another benefit: it allows you to create simple XAML pages that you can run in the browser without compiling them into applications. (As you learned in Chapter 2, if your XAML file has a linked code-behind file, it can't be opened in a browser.)

The Binding Expression

When using data binding, you don't need to make any changes to your source object (which is the Slider in this example). Just configure it to take the correct range of values, as you would usually:

```
<Slider Name="sliderFontSize" Margin="3"
 Minimum="1" Maximum="40" Value="10"
 TickFrequency="1" TickPlacement="TopLeft">
</Slider>
```

The binding is defined in the TextBlock element. Instead of setting the FontSize using a literal value, you use a binding expression, as shown here:

```
<TextBlock Margin="10" Text="Simple Text" Name="lblSampleText"
 FontSize="{Binding ElementName=sliderFontSize, Path=Value}" >
</TextBlock>
```

Data binding expressions use a XAML markup extension (and hence have curly braces). You begin with the word *Binding*, because you're creating an instance of the System.Windows.Data.Binding class. Although you can configure a Binding object in several ways, in this situation, you need to set just two properties: the ElementName that indicates the source element and a Path that indicates the property in the source element.

The name Path is used instead of Property because the Path might point to a property of a property (for example, FontFamily.Source) or an indexer used by a property (for example, Content.Children[0]). You can build up a path with multiple periods to dig into a property of a property of a property, and so on.

If you want to refer to an attached property (a property that's defined in another class but applied to the bound element), you need to wrap the property name in parentheses. For example, if you're binding to an element that's placed in a Grid, the path (Grid.Row) retrieves the row number where you've placed it.

Binding Errors

WPF doesn't raise exceptions to notify you about data binding problems. If you specify an element or a property that doesn't exist, you won't receive any indication; instead, the data will simply fail to appear in the target property.

At first glance, this seems like a debugging nightmare. Fortunately, WPF *does* output trace information that details binding failures. This information appears in Visual Studio's Output window when you're debugging the application. For example, if you try to bind to a nonexistent property, you'll see a message like this in the Output window:

```
System.Windows.Data Error: 35 : BindingExpression path error:
 'Tex' property not found on 'object' ''TextBox' (Name='txtFontSize')'.
 BindingExpression:Path=Tex; DataItem='TextBox' (Name='txtFontSize');
 target element is 'TextBox' (Name='');
 target property is 'Text' (type 'String')
```

WPF also ignores any exception that's thrown when you attempt to read the source property and quietly swallows the exception that occurs if the source data can't be cast to the data type of the target property. However, there is another option when dealing with these problems—you can tell WPF to change the appearance of the source element to indicate that an error has occurred. For example, this allows you to flag invalid input with an exclamation icon or a red outline. You'll learn more about validation in Chapter 19.

Binding Modes

One of the neat features of data binding is that your target is updated automatically, no matter how the source is modified. In this example, the source can be modified in only one way—by the user's interaction with the slider thumb. However, consider a slightly revamped version of this example that adds a few buttons, each of which applies a preset value to the slider. Figure 8-2 shows the new window.

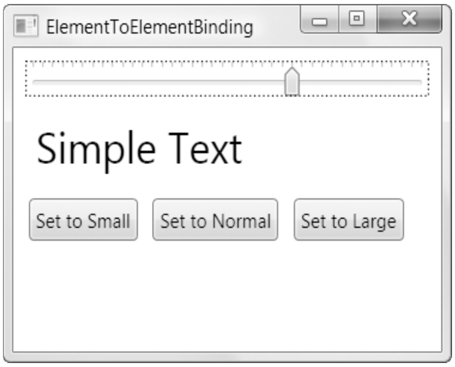

Figure 8-2. Modifying the data binding source programmatically

When you click the Set to Large button, this code runs:

```
private void cmd_SetLarge(object sender, RoutedEventArgs e)
{
    sliderFontSize.Value = 30;
}
```

This code sets the value of the slider, which in turn forces a change to the font size of the text through data binding. It's the same as if you had moved the slider thumb yourself.

However, this code doesn't work as well:

```
private void cmd_SetLarge(object sender, RoutedEventArgs e)
{
    lblSampleText.FontSize = 30;
}
```

It sets the font of the text box directly. As a result, the slider position isn't updated to match. Even worse, this has the effect of wiping out your font size binding and replacing it with a literal value. If you move the slider thumb now, the text block won't change at all.

Interestingly, there's a way to force values to flow in both directions: from the source to the target *and* from the target to the source. The trick is to set the Mode property of the Binding. Here's a revised bidirectional binding that allows you to apply changes to either the source or the target and have the other piece of the equation update itself automatically:

```
<TextBlock Margin="10" Text="Simple Text" Name="lblSampleText"
 FontSize="{Binding ElementName=sliderFontSize, Path=Value, Mode=TwoWay}" >
</TextBlock>
```

In this example, you have no reason to use a two-way binding (which requires more overhead) because you can solve the problem by using the correct code. However, consider a variation of this example that includes a text box where the user can set the font size precisely. This text box needs to use a two-way binding, so it can both apply the user's changes and display the most recent size value in the text box when it's changed through another avenue. You'll see this example in the next section.

WPF allows you to use one of five values from the System.Windows.Data.BindingMode enumeration when setting the Binding.Mode property. Table 8-1 has the full list.

Table 8-1. Values from the BindingMode Enumeration

Name	Description
OneWay	The target property is updated when the source property changes.
TwoWay	The target property is updated when the source property changes, and the source property is updated when the target property changes.
OneTime	The target property is set initially based on the source property value. However, changes are ignored from that point onward (unless the binding is set to a completely different object or you call BindingExpression.UpdateTarget(), as described later in this chapter). Usually, you'll use this mode to reduce overhead if you know the source property won't change.
OneWayToSource	Similar to OneWay but in reverse. The source property is updated when the target property changes (which might seem a little backward), but the target property is never updated.
Default	The type of binding depends on the target property. It's either TwoWay (for user-settable properties, such as the TextBox.Text) or OneWay (for everything else). All bindings use this approach unless you specify otherwise.

Figure 8-3 illustrates the difference. You've already seen OneWay and TwoWay. OneTime is fairly straightforward. The other two choices bear some additional investigation.

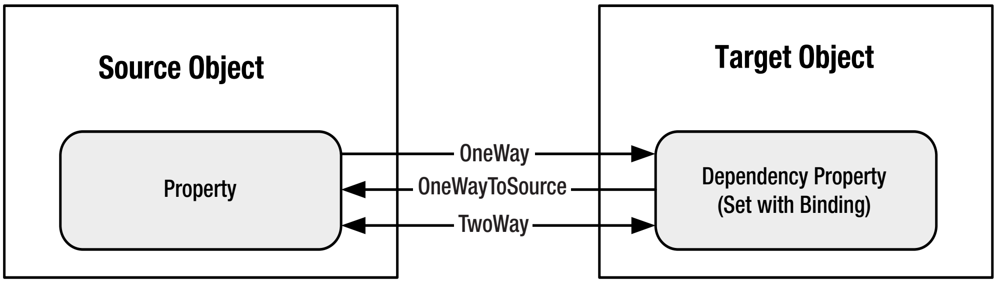

Figure 8-3. Different ways to bind two properties

OneWayToSource

You might wonder why there's both a OneWay and a OneWayToSource option—after all, both values create a one-way binding that works in the same way. The only difference is where the binding expression is placed. Essentially, OneWayToSource allows you to flip the source and target by placing the expression in what would ordinarily be considered the binding source.

The most common reason to use this trick is to set a property that isn't a dependency property. As you learned at the beginning of this chapter, binding expressions can be used only to set dependency properties. But by using OneWayToSource, you can overcome this limitation, provided the property that's supplying the value is itself a dependency property.

Default

Initially, it seems logical to assume that all bindings are one-way unless you explicitly specify otherwise. (After all, that's the way the simple slider example works.) However, this actually isn't the case. To demonstrate this fact to yourself, return to the example with the bound text box that allows you to edit the current font size. If you remove the Mode=TwoWay setting, this example still works just as well. That's because WPF uses a different Mode default depending on the property you're binding. (Technically, there's a bit of metadata on every dependency property—the FrameworkPropertyMetadata.BindsTwoWayByDefault flag—that indicates whether that property should use one-way or two-way binding.)

Often, the default is exactly what you want. However, you can imagine an example with a read-only text box that the user can't change. In this case, you can reduce the overhead slightly by setting the mode to use one-way binding.

As a general rule of thumb, it's never a bad idea to explicitly set the mode. Even in the case of a text box, it's worth emphasizing that you want a two-way binding by including the Mode property.

Creating Bindings with Code

When you're building a window, it's usually most efficient to declare your binding expression in the XAML markup using the Binding markup extension. However, it's also possible to create a binding using code.

Here's how you could create the binding for the TextBlock shown in the previous example:

```
Binding binding = new Binding();
binding.Source = sliderFontSize;
binding.Path = new PropertyPath("Value");
binding.Mode = BindingMode.TwoWay;
lblSampleText.SetBinding(TextBlock.FontSizeProperty, binding);
```

You can also remove a binding with code using two static methods of the BindingOperations class. The ClearBinding() method takes a reference to the dependency property that has the binding you want to remove, while ClearAllBindings() removes all the data binding for an element:

```
BindingOperations.ClearAllBindings(lblSampleText);
```

Both ClearBinding() and ClearAllBindings() use the ClearValue() method that every element inherits from the based DependencyObject class. ClearValue() simply removes a property's local value (which, in this case, is a data binding expression).

Markup-based binding is far more common than programmatic binding, because it's cleaner and requires less work. In this chapter, all the examples use markup to create their bindings. However, you will want to use code to create a binding in some specialized scenarios:

- **Creating a dynamic binding.** If you want to tailor a binding based on other runtime information or create a different binding depending on the circumstances, it often makes sense to create your binding in code. (Alternatively, you could define every binding you might want to use in the Resources collection of your window and just add the code that calls SetBinding() with the appropriate binding object.)
- **Removing a binding.** If you want to remove a binding so that you can set a property in the usual way, you need the help of the ClearBinding() or ClearAllBindings() method. It isn't enough to simply apply a new value to the property. If you're using a two-way binding, the value you set is propagated to the linked object, and both properties remain synchronized.

■ **Note** You can remove any binding using the ClearBinding() and ClearAllBindings() methods. It doesn't matter whether the binding was applied programmatically or in XAML markup.

- **Creating custom controls.** To make it easier for other people to modify the visual appearance of a custom control you build, you'll need to move certain details (such as event handlers and data binding expressions) into your code and out of your markup. Chapter 18 includes a custom color-picking control that uses code to create its bindings.

Multiple Bindings

Although the previous example includes just a single binding, you don't need to stop there. If you wanted, you could set the TextBlock up to draw its text from a text box, its current foreground and background color from separate lists of colors, and so on. Here's an example:

```
<TextBlock Margin="3" Name="lblSampleText"
  FontSize="{Binding ElementName=sliderFontSize, Path=Value}"
  Text="{Binding ElementName=txtContent, Path=Text}"
  Foreground="{Binding ElementName=lstColors, Path=SelectedItem.Tag}" >
</TextBlock>
```

Figure 8-4 shows the triple-bound TextBlock.

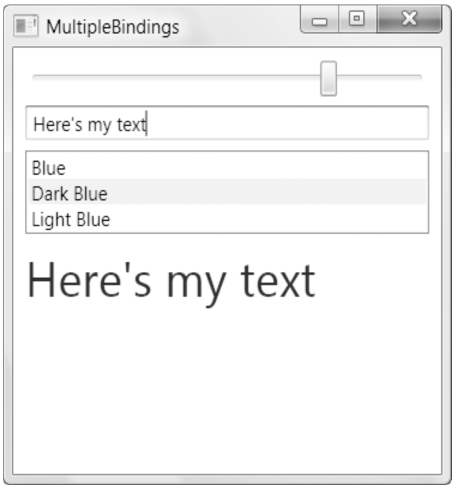

Figure 8-4. *A TextBlock that's bound to three elements*

You can also chain data bindings. For example, you could create a binding expression for the TextBox.Text property that links to the TextBlock.FontSize property, which contains a binding expression that links to the Slider.Value property. In this case, when the user drags the slider thumb to a new position, the value flows from the Slider to the TextBlock and then from the TextBlock to the TextBox. Although this works seamlessly, a cleaner approach is to bind your elements as closely as possible to the data they use. In the example described here, you should consider binding both the TextBlock and the TextBox directly to the Slider.Value property.

Life becomes a bit more interesting if you want a target property to be influenced by more than one source—for example, if you want there to be two equally legitimate bindings that set its property. At first glance, this doesn't seem possible. After all, when you create a binding, you can point to only a single target property. However, you can get around this limitation in several ways.

The easiest approach is to change the data binding mode. As you've already learned, the Mode property allows you to change the way a binding works so that values aren't just pushed from the source to the target but also from the target to the source. Using this technique, you can create multiple binding expressions that set the same property. The last-set property is the one that comes into effect.

To understand how this works, consider a variation of the slider bar example that introduces a text box where you can set the exact font size. In this example (shown in Figure 8-5), you can set the TextBlock.FontSize property in two ways: by dragging the slider thumb or by typing a font size into the text box. All the controls are synchronized so that if you type a new number in the text box, the font size of the sample text is adjusted *and* the slider thumb is moved to the corresponding position.

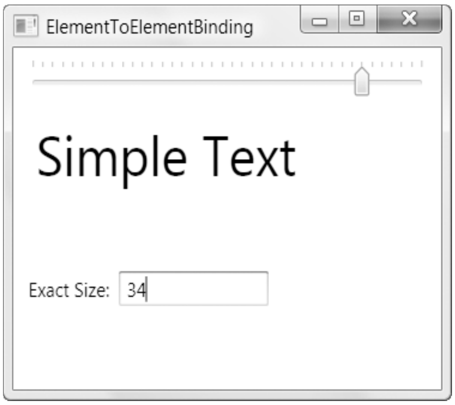

Figure 8-5. *Linking two properties to the font size*

As you know, you can apply only a single data binding to the TextBlock.FontSize property. It makes sense to leave the TextBlock.FontSize property as is, so that it binds directly to the slider:

```
<TextBlock Margin="10" Text="Simple Text" Name="lblSampleText"
 FontSize="{Binding ElementName=sliderFontSize, Path=Value, Mode=TwoWay}" >
</TextBlock>
```

Although you can't add another binding to the FontSize property, you *can* bind the new control—the TextBox—to the TextBlock.FontSize property. Here's the markup you need:

```
<TextBox Text="{Binding ElementName=lblSampleText, Path=FontSize, Mode=TwoWay}">
</TextBox>
```

Now, whenever the TextBlock.FontSize property changes, the current value is inserted into the text box. Even better, you can edit the value in the text box to apply a specific size. Notice that in order for this example to work, the TextBox.Text property must use a two-way binding so that values travel both ways. Otherwise, the text box will be able to display the TextBlock.FontSize value but won't be able to change it.

This example has a few quirks:

- Because the Slider.Value property is a double, you'll end up with a fractional font size when you drag the slider thumb. You can constrain the slider to whole numbers by setting the TickFrequency property to 1 (or some other whole number interval) and setting the IsSnapToTickEnabled property to true.
- The text box allows letters and other non-numeric characters. If you enter any, the text box value can no longer be interpreted as a number. As a result, the data binding silently fails, and the font size is set to 0. Another approach would be to handle key presses in the text box to prevent invalid input altogether or to use validation, as discussed in Chapter 19.
- The changes you make in the text box aren't applied until the text box loses focus (for example, when you tab to another control). If this isn't the behavior you want, you can get an instantaneous refresh using the UpdateSourceTrigger property of the Binding object, as you'll learn shortly in the "Binding Updates" section.

Interestingly, the solution shown here isn't the only way to connect the text box. It's just as reasonable to configure the text box so that it changes the Slider.Value property instead of the TextBlock.FontSize property:

```
<TextBox Text="{Binding ElementName=sliderFontSize, Path=Value, Mode=TwoWay}">
</TextBox>
```

Now changing the text box triggers a change in the slider, which then applies the new font to the text. Once again, this approach works only if you use two-way data binding.

And lastly, you can swap the roles of the slider and text box so that the slider binds to the text box. To do this, you need to create an unbound TextBox and give it a name:

```
<TextBox Name="txtFontSize" Text="10">
</TextBox>
```

Then you can bind the Slider.Value property, as shown here:

```
<Slider Name="sliderFontSize" Margin="3"
 Minimum="1" Maximum="40"
 Value="{Binding ElementName=txtFontSize, Path=Text, Mode=TwoWay}"
 TickFrequency="1" TickPlacement="TopLeft">
</Slider>
```

Now the slider is in control. When the window is first shown, it retrieves the TextBox.Text property and uses that to set its Value property. When the user drags the slider thumb to a new position, it uses the binding to update the text box. Alternatively, the user can update the slider value (and the font size of the sample text) by typing in the text box.

■ **Note** If you bind the Slider.Value property, the text box behaves slightly differently than the previous two examples. Any edits you make in the text box are applied immediately, rather than waiting until the text box loses focus. You'll learn more about controlling when an update takes place in the next section.

As this example demonstrates, two-way bindings give you remarkable flexibility. You can use them to apply changes from the source to the target and from the target to the source. You can also apply them in combination to create a surprisingly complex code-free window.

Usually, the decision of where to place a binding expression is driven by the logic of your coding model. In the previous example, it probably makes more sense to place the binding in the TextBox.Text property rather than the Slider.Value property, because the text box is an optional add-on to an otherwise complete example, not a core ingredient that the slider relies on. It also makes more sense to bind the text box directly to the TextBlock.FontSize property rather than the Slider.Value property. (Conceptually, you're interested in reporting the current font size, and the slider is just one of the ways this font size can be set. Even though the slider position is the same as the font size, it's an unnecessary extra detail if you're trying to write the cleanest possible markup.) Of course, these decisions are subjective and a matter of coding style. The most important lesson is that all three approaches can give you the same behavior.

In the following sections, you'll explore two details that apply to this example. First, you'll consider your choices for setting the direction of a binding. Then you'll see how you can tell WPF exactly when it should update the source property in a two-way binding.

Binding Updates

In the example shown in Figure 8-5 (which binds TextBox.Text to TextBlock.FontSize), there's another quirk. As you change the displayed font size by typing in the text box, nothing happens. The change is not applied until you tab to another control. This behavior is different from the behavior you see with the slider control. With that control, the new font size is applied as you drag the slider thumb—there's no need to tab away.

To understand this difference, you need to take a closer look at the binding expressions used by these two controls. When you use OneWay or TwoWay binding, the changed value is propagated from the source to the target immediately. In the case of the slider, there's a one-way binding expression in the TextBlock. Thus, changes in the Slider.Value property are immediately applied to the TextBlock.FontSize property. The same behavior takes place in the text box example—changes to the source (which is TextBlock.FontSize) affect the target (TextBox.Text) immediately.

However, changes that flow in the reverse direction—from the target to the source—don't necessarily happen immediately. Instead, their behavior is governed by the Binding.UpdateSourceTrigger property, which takes one of the values listed in Table 8-2. When the text is taken from the text box and used to update the TextBlock.FontSize property, you're witnessing an example of a target-to-source update that uses the UpdateSourceTrigger.LostFocus behavior.

Table 8-2. *Values from the UpdateSourceTrigger Enumeration*

Name	Description
PropertyChanged	The source is updated immediately when the target property changes.
LostFocus	The source is updated when the target property changes and the target loses focus.
Explicit	The source is not updated unless you call the BindingExpression.UpdateSource() method.
Default	The updating behavior is determined by the metadata of the target property (technically, its FrameworkPropertyMetadata.DefaultUpdateSourceTrigger property). For most properties, the default behavior is PropertyChanged, although the TextBox.Text property has a default behavior of LostFocus.

Remember that the values in Table 8-2 have no effect over how the target is updated. They simply control how the *source* is updated in a TwoWay or OneWayToSource binding.

With this knowledge, you can improve the text box example so that changes are applied to the font size as the user types in the text box. Here's how:

```
<TextBox Text="{Binding ElementName=txtSampleText, Path=FontSize, Mode=TwoWay,
 UpdateSourceTrigger=PropertyChanged}" Name="txtFontSize"></TextBox>
```

■ **Tip** The default behavior of the TextBox.Text property is LostFocus, simply because the text in a text box will change repeatedly as the user types, causing multiple refreshes. Depending on how the source control updates itself, the PropertyChanged update mode can make the application feel more sluggish. Additionally, it might cause the source object to refresh itself before an edit is complete, which can cause problems for validation.

For complete control over when the source object is updated, you can choose the UpdateSourceTrigger.Explicit mode. If you use this approach in the text box example, nothing happens when the text box loses focus. Instead, it's up to your code to manually trigger the update. For example, you could add an Apply button that calls the BindingExpression.UpdateSource() method, triggering an immediate refresh and updating the font size.

Of course, before you can call BindingExpression.UpdateSource(), you need a way to get a BindingExpression object. A BindingExpression object is a slim package that wraps together two things: the Binding object you've already learned about (provided through the BindingExpression.ParentBinding property) and the object that's being bound from the source (BindingExpression.DataItem). In addition, the BindingExpression object provides two methods for triggering an immediate update for one part of the binding: UpdateSource() and UpdateTarget().

To get a BindingExpression object, you use the GetBindingExpression() method, which every element inherits from the base FrameworkElement class, and pass in the target property that has the binding. Here's an example that changes the font size in the TextBlock based on the current text in the text box:

```
// Get the binding that's applied to the text box.
BindingExpression binding = txtFontSize.GetBindingExpression(TextBox.TextProperty);

// Update the linked source (the TextBlock).
binding.UpdateSource();
```

Binding to Objects That Aren't Elements

So far, you've focused on adding bindings that link two elements. But in data-driven applications, it's more common to create binding expressions that draw their data from a nonvisual object. The only requirement is that the information you want to display must be stored in *public properties*. The WPF data binding infrastructure won't pick up private information or public fields.

When binding to an object that isn't an element, you need to give up the Binding.ElementName property and use one of the following properties instead:

- **Source.** This is a reference that points to the source object—in other words, the object that's supplying the data.
- **RelativeSource.** This points to the source object using a RelativeSource object, which allows you to base your reference on the current element. This is a specialized tool that's handy when writing control templates and data templates.

- **DataContext.** If you don't specify a source using the Source or RelativeSource property, WPF searches up the element tree, starting at the current element. It examines the DataContext property of each element and uses the first one that isn't null. The DataContext property is extremely useful if you need to bind several properties of the same object to different elements, because you can set the DataContext property of a higher-level container object, rather than setting it directly on the target element.

The following sections fill in a few more details about these three options.

Source

The Source property is quite straightforward. The only catch is that you need to have your data object handy in order to bind it. As you'll see, you can use several approaches for getting the data object: pull it out of a resource, generate it programmatically, or get it with the help of a data provider.

The simplest option is to point the Source to some static object that's readily available. For example, you could create a static object in your code and use that. Or, you could use an ingredient from the .NET class library, as shown here:

```
<TextBlock Text="{Binding Source={x:Static SystemFonts.IconFontFamily},
 Path=Source}"></TextBlock>
```

This binding expression gets the FontFamily object that's provided by the static SystemFonts.IconFontFamily property. (Notice that you need the help of the static markup extension to set the Binding.Source property.) It then sets the Binding.Path property to the FontFamily.Source property, which gives the name of the font family. The result is a single line of text. In Windows Vista or Windows 7, the font name Segoe UI appears.

Another option is to bind to an object that you've previously created as a resource. For example, this markup creates a FontFamily object that points to the Calibri font:

```
<Window.Resources>
  <FontFamily x:Key="CustomFont">Calibri</FontFamily>
</Window.Resources>
```

And here's a TextBlock that binds to this resource:

```
<TextBlock Text="{Binding Source={StaticResource CustomFont},
 Path=Source}"></TextBlock>
```

Now the text you'll see is "Calibri."

RelativeSource

The RelativeSource property allows you to point to a source object based on its relation to the target object. For example, you can use RelativeSource property to bind an element to itself or to bind to a parent element that's found an unknown number of steps up the element tree.

To set the Binding.RelativeSource property, you use a RelativeSource object. This makes the syntax a little more convoluted, because you need to create a Binding object and create a nested RelativeSource object inside. One option is to use the property-setting syntax instead of the Binding markup extension.

For example, the following code creates a Binding object for the TextBlock.Text property. The Binding object uses a RelativeSource that searches out the parent window and displays the window title.

```
<TextBlock>
  <TextBlock.Text>
    <Binding Path="Title">
      <Binding.RelativeSource>
        <RelativeSource Mode="FindAncestor" AncestorType="{x:Type Window}" />
      </Binding.RelativeSource>
    </Binding>
  </TextBlock.Text>
</TextBlock>
```

The RelativeSource object uses the FindAncestor mode, which tells it to search up the element tree until it finds the type of element defined by the AncestorType property.

The more common way to write this binding is to combine it into one string using the Binding and RelativeSource markup extensions, as shown here:

```
<TextBlock Text="{Binding Path=Title,
   RelativeSource={RelativeSource FindAncestor, AncestorType={x:Type Window}} }">
</TextBlock>
```

The FindAncestor mode is only one of four options when you create a RelativeSource object. Table 8-3 lists all four modes.

Table 8-3. *Values from the RelativeSourceMode Enumeration*

Name	Description
Self	The expression binds to another property in the same element.
FindAncestor	The expression binds to a parent element. WPF will search up the element tree until it finds the parent you want. To specify the parent, you must also set the AncestorType property to indicate the type of parent element you want to find. Optionally, you can use the AncestorLevel property to skip a certain number of occurrences of the specified element. For example, if you want to bind to the third element of type ListBoxItem when going up the tree, you would set AncestorType={x:Type ListBoxItem} and AncestorLevel=3, thereby skipping the first two ListBoxItems. By default, AncestorLevel is 1, and the search stops at the first matching element.
PreviousData	The expression binds to the previous data item in a data-bound list. You would use this in a list item.
TemplatedParent	The expression binds to the element on which the template is applied. This mode works only if your binding is located inside a control template or data template.

At first glance, the RelativeSource property seems like a way to unnecessarily complicate your markup. After all, why not bind directly to the source you want using the Source or ElementName property? However, this isn't always possible, usually because the source and target objects are in different chunks of markup. This happens when you're creating control templates and data templates. For example, if you're building a data template that changes the way items are presented in a list, you might need to access the top-level ListBox object to read a property.

DataContext

In some cases, you'll have a number of elements that bind to the same object. For example, consider the following group of TextBlock elements, each of which uses a similar binding expression to pull out different details about the default icon font, including its line spacing and the style and weight of the first typeface it provides (both of which are simply Regular). You can use the Source property for each one, but this results in fairly lengthy markup:

```
<StackPanel>
  <TextBlock Text="{Binding Source={x:Static SystemFonts.IconFontFamily},
   Path=Source}"></TextBlock>
  <TextBlock Text="{Binding Source={x:Static SystemFonts.IconFontFamily},
   Path=LineSpacing}"></TextBlock>
  <TextBlock Text="{Binding Source={x:Static SystemFonts.IconFontFamily},
   Path=FamilyTypefaces[0].Style}"></TextBlock>
  <TextBlock Text="{Binding Source={x:Static SystemFonts.IconFontFamily},
   Path=FamilyTypefaces[0].Weight}"></TextBlock>
</StackPanel>
```

In this situation, it's cleaner and more flexible to define the binding source once using the FrameworkElement.DataContext property. In this example, it makes sense to set the DataContext property of the StackPanel that contains all the TextBlock elements. (You could also set the DataContext property at an even higher level—for example, the entire window—but its better to define it as narrowly as possible to make your intentions clear.)

You can set the DataContext property of an element in the same way that you set the Binding.Source property. In other words, you can supply your object inline, pull it out of a static property, or pull it out of a resource, as shown here:

```
<StackPanel DataContext="{x:Static SystemFonts.IconFontFamily}">
```

Now you can streamline your binding expressions by leaving out the source information:

```
<TextBlock Margin="5" Text="{Binding Path=Source}"></TextBlock>
```

When the source information is missing from a binding expression, WPF checks the DataContext property of that element. If it's null, WPF searches up the element tree looking for the first data context that isn't null. (Initially, the DataContext property of all elements is null.) If it finds a data context, it uses that for the binding. If it doesn't, the binding expression doesn't apply any value to the target property.

■ **Note** If you create a binding that explicitly specifies a source using the Source property, your element uses that source instead of any data context that might be available.

This example shows how you can create a basic binding to an object that isn't an element. However, to use this technique in a realistic application, you need to pick up a few more skills. In Chapter 19, you'll learn how to display information drawn from a database by building on these data binding techniques.

The Last Word

In this chapter, you took a quick look at data binding fundamentals. You learned how to pull information out of one element and display it in another without a single line of code. And although this technique seems fairly modest right now, it's an essential skill that will allow you to perform much more impressive feats, such as restyling controls with custom control templates (discussed in Chapter 17).

In Chapters 19 and 20, you'll greatly extend your data binding skills. You'll learn how to show entire collections of data objects in a list, handle edits with validation, and turn ordinary text into a richly formatted data display. But for now, you have all the data binding experience you need to tackle the chapters ahead.

CHAPTER 9

Commands

In Chapter 5, you learned about routed events, which you can use to respond to a wide range of mouse and keyboard actions. However, events are a fairly low-level ingredient. In a realistic application, functionality is divided into higher-level *tasks*. These tasks may be triggered by a variety of different actions and through a variety of different user-interface elements, including main menus, context menus, keyboard shortcuts, and toolbars.

WPF allows you to define these tasks—known as *commands*—and connect controls to them so you don't need to write repetitive event handling code. Even more important, the command feature manages the state of your user interface by automatically disabling controls when the linked commands aren't available. It also gives you a central place to store (and localize) the text captions for your commands.

In this chapter, you'll learn how to use the prebuilt command classes in WPF, wire them up to controls, and define your own commands. You'll also consider the limitations of the command model—namely, the lack of a command history and the lack of support for an application-wide Undo feature—and you'll see how you can build your own system for tracking and reversing commands.

Understanding Commands

In a well-designed Windows application, the application logic doesn't sit in the event handlers but is coded in higher-level methods. Each one of these methods represents a single application "task." Each task may rely on other libraries (such as separately compiled components that encapsulate business logic or database access). Figure 9-1 shows this relationship.

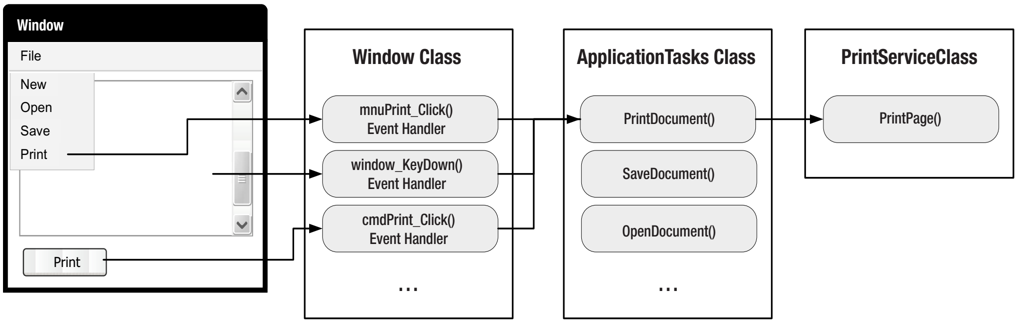

Figure 9-1. *Mapping event handlers to a task*

The most obvious way to use this design is to add event handlers wherever they're needed, and use each event handler to call the appropriate application method. In essence, your window code becomes a stripped-down switchboard that responds to input and forwards requests to the heart of the application.

Although this design is perfectly reasonable, it doesn't save you any work. Many application tasks can be triggered through a variety of routes, so you'll often need to code several event handlers that call the same application method. This in itself isn't much of a problem (because the switchboard code is so simple), but life becomes much more complicated when you need to deal with user interface *state*.

A simple example shows the problem. Imagine you have a program that includes an application method named PrintDocument(). This method can be triggered in four ways: through a main menu (by choosing File ➤ Print), through a context menu (by right-clicking somewhere and choosing Print), through a keyboard shortcut (Ctrl+P), and through a toolbar button. At certain points in your application's lifetime, you need to temporarily disable the PrintDocument() task. That means you need to disable the two menu commands and the toolbar button so they can't be clicked, and you need to ignore the Ctrl+P shortcut. Writing the code that does this (and adding the code that enables these controls later) is messy. Even worse, if it's not done properly, you might wind up with different blocks of state code overlapping incorrectly, causing a control to be switched on even when it shouldn't be available. Writing and debugging this sort of code is one of the least glamorous aspects of Windows development.

Much to the surprise of many experienced Windows developers, the Windows Forms toolkit didn't provide any features that could help you deal with these issues. Developers could build the infrastructure they needed on their own, but most weren't that ambitious.

Fortunately, WPF fills in the gaps with a new command model. It adds two key features:

- It delegates events to the appropriate commands.
- It keeps the enabled state of a control synchronized with the state of the corresponding command.

The WPF command model isn't quite as straightforward as you might expect. To plug into the routed event model, it requires several separate ingredients, which you'll learn about in this chapter. However, the command model is *conceptually* simple. Figure 9-2 shows how a command-based application changes the design shown in Figure 9-1. Now each action that initiates printing (clicking the button, clicking the menu item, or pressing Ctrl+P) is mapped to the same command. A command binding links that command to a single event handler in your code.

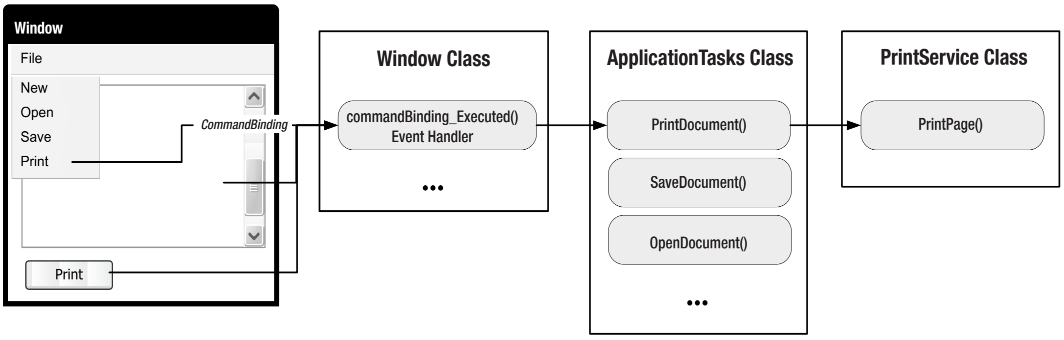

Figure 9-2. *Mapping events to a command*

The WPF command system is a great tool for simplifying application design. However, it still has some fairly significant gaps. Notably, WPF doesn't have any support for the following:

- Command tracking (for example, keeping a history of recent commands)
- Undoable commands
- Commands that have state and can be in different modes (for example, a command that can be toggled on or off)

The WPF Command Model

The WPF command model consists of a surprising number of moving parts. All together, it has four key ingredients:

- **Commands.** A command *represents* an application task and keeps track of whether it can be executed. However, commands don't actually contain the code that *performs* the application task.
- **Command bindings.** Each command binding links a command to the related application logic, for a particular area of your user interface. This factored design is important, because a single command might be used in several places in your application and have a different significance in each place. To handle this, you use the same command with different command bindings.
- **Command sources.** A command source triggers a command. For example, a MenuItem and a Button can both be command sources. Clicking them executes the bound command.
- **Command targets.** A command target is the element on which the command is being performed. For example, a Paste command might insert text into a TextBox, and an OpenFile command might pop a document into a DocumentViewer. The target may or may not be important, depending on the nature of the command.

In the following sections, you'll dig into the first ingredient: the WPF command.

The ICommand Interface

The heart of the WPF command model is the System.Windows.Input.ICommand interface, which defines how commands work. This interface includes two methods and an event:

```
public interface ICommand
{
    void Execute(object parameter);
    bool CanExecute(object parameter);

    event EventHandler CanExecuteChanged;
}
```

In a simple implementation, the Execute() method would contain the application task logic (for example, printing the document). However, as you'll see in the next section, WPF is a bit more elaborate. It uses the Execute() method to fire off a more complicated process that eventually raises an event that's handled elsewhere in your application. This gives you the ability to use ready-made command classes and plug in your own logic. It also gives you the flexibility to use one command (such as Print) in several different places.

The CanExecute() method returns the state of the command: true if it's enabled and false if it's disabled. Both Execute() and CanExecute() accept an additional parameter object that you can use to pass along any extra information you need.

Finally, the CanExecuteChanged event is raised when the state changes. This is a signal to any controls using the command that they should call the CanExecute() method to check the command's state. This is part of the glue that allows command sources (such as a Button or MenuItem) to automatically enable themselves when the command is available and to disable themselves when it's not available.

The RoutedCommand Class

When creating your own commands, you won't implement ICommand directly. Instead, you'll use the System.Windows.Input.RoutedCommand class, which implements this interface for you. The RoutedCommand class is the only class in WPF that implements ICommand. In other words, all WPF commands are instances of RoutedCommand (or a derived class).

One of the key concepts behind the command model in WPF is that the RoutedUICommand class doesn't contain any application logic. It simply *represents* a command. This means one RoutedCommand object has the same capabilities as another.

The RoutedCommand class adds a fair bit of extra infrastructure for event tunneling and bubbling. Whereas the ICommand interface encapsulates the idea of a command—an action that can be triggered and may or may not be enabled—the RoutedCommand modifies the command so that it can bubble through the WPF element hierarchy to get to the correct event handler.

Why WPF Commands Need Event Bubbling

When looking at the WPF command model for the first time, it's tricky to grasp exactly why WPF commands require routed events. After all, shouldn't the command object take care of performing the command, regardless of how it's invoked?

If you were using the ICommand interface directly to create your own command classes, this would be true. The code would be hardwired into the command, so it would work the same way no matter what triggers the command. You wouldn't need event bubbling.

However, WPF uses a number of *prebuilt* commands. These command classes don't contain any real code. They're just conveniently defined objects that represent a common application task (such as printing a document). To act on these commands, you need to use a command binding, which raises an event to your code (as shown in Figure 9-2). To make sure you can handle this event in one place, even if it's fired by different command sources in the same window, you need the power of event bubbling.

This raises an interesting question: why use prebuilt commands at all? Wouldn't it be clearer to have custom command classes do all the work, instead of relying on an event handler? In many ways, this design would be simpler. However, the advantage of prebuilt commands is that they provide much better possibilities for integration. For example, a third-party developer could create a document viewer control that uses the prebuilt Print command. As long as your application uses the same prebuilt command, you won't need to do any extra work to wire up printing in your application. Seen this way, commands are a major piece of WPF's pluggable architecture.

To support routed events, the RoutedCommand class implements the ICommand interface privately and then adds slightly different versions of its methods. The most obvious change you'll notice is that the Execute() and CanExecute() methods take an extra parameter. Here are their new signatures:

```
public void Execute(object parameter, IInputElement target)
{...}

public bool CanExecute(object parameter, IInputElement target)
{...}
```

The *target* is the element where the event handling begins. This event begins at the target element and bubbles up to higher-level containers until your application handles it to perform the appropriate task. (To handle the Executed event, your element needs the help of yet another class—CommandBinding.)

Along with this shift, the RoutedElement also introduces three properties: the command name (Name), the class that this command is a member of (OwnerType), and any keystrokes or mouse actions that can also be used to trigger the command (in the InputGestures collection).

The RoutedUICommand Class

Most of the commands you'll deal with won't be RoutedCommand objects; rather, they will be instances of the RoutedUICommand class, which derives from RoutedCommand. (In fact, all the ready-made commands that WPF provides are RoutedUICommand objects.)

RoutedUICommand is intended for commands with text that is displayed somewhere in the user interface (for example, the text of a menu item or the tooltip for a toolbar button). The RoutedUICommand class adds a single property—Text—which is the display text for that command.

The advantage of defining the command text with the command (rather than directly on the control) is that you can perform your localization in one place. However, if your command text never appears anywhere in the user interface, the RoutedCommand class is equivalent.

■ **Note** You don't need to use the RoutedUICommand text in your user interface. In fact, there may be good reasons to use something else. For example, you might prefer "Print Document" to just "Print," and in some cases you might replace the text altogether with a tiny graphic.

The Command Library

The designers of WPF realized that every application is likely to have a large number of commands and that many commands are common to many different applications. For example, all document-based