CHAPTER 8

Element Binding

At its simplest, data binding is a relationship that tells WPF to extract some information from a *source* object and use it to set a property in a *target* object. The target property is always a dependency property, and it's usually in a WPF element—after all, the ultimate goal of WPF data binding is to display some information in your user interface. However, the source object can be just about anything, ranging from another WPF element to an ADO.NET data object (such as DataTable and DataRow) or a data-only object of your own creation.

In this chapter, you'll begin your exploration of data binding by considering the simplest approach: element-to-element binding. In Chapter 19, you'll revisit the data-binding story, and learn the most efficient way to shuttle data from a database to your data forms.

What's New WPF 4.5 has made several improvements to the data-binding system, most of which you'll consider in Chapter 19. In this chapter, you'll see two of the more minor changes. First, you'll consider the Delay property that allows you to add a pause before a data-binding expression is evaluated (see the "Binding Delays" section). Second, you'll look at WPF's enhanced ability to get binding information programmatically (see the "Retrieving Bindings in Code" section).

Binding Elements Together

The simplest data-binding scenario occurs when your source object is a WPF element and your source property is a dependency property. That's because dependency properties have built-in support for change notification, as explained in Chapter 4. As a result, when you change the value of the dependency property in the source object, the bound property in the target object is updated immediately. This is exactly what you want, and it happens without requiring you to build any additional infrastructure.

Note Although it's nice to know that element-to-element binding is the simplest approach, most developers are more interested in finding out which approach is most common in the real world. Overall, the bulk of your data-binding work will be spent binding elements to data objects. This allows you to display the information that you've extracted from an external source (such as a database or file). However, element-to-element binding is often useful. For example, you can use element-to-element binding to automate the way elements interact so that when a user

modifies a control, another element is updated automatically. This is a valuable shortcut that can save you from writing boilerplate code.

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.

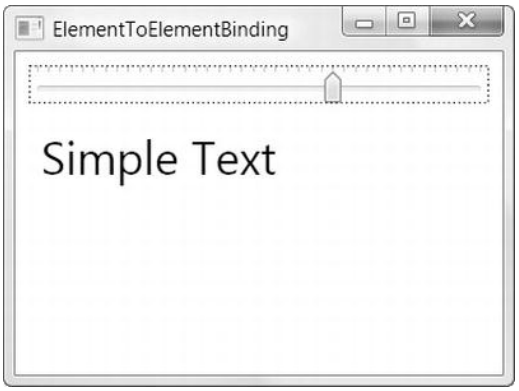

Figure 8-1. *Linked controls through data binding*

Clearly, it wouldn't be difficult to create this behavior by 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 using a literal value to set the FontSize , 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=Text; 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.

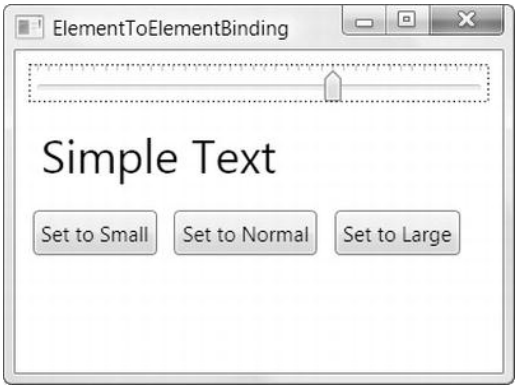

Figure 8-2. *Modifying the data-binding source programmatically*

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 warrant some additional investigation.

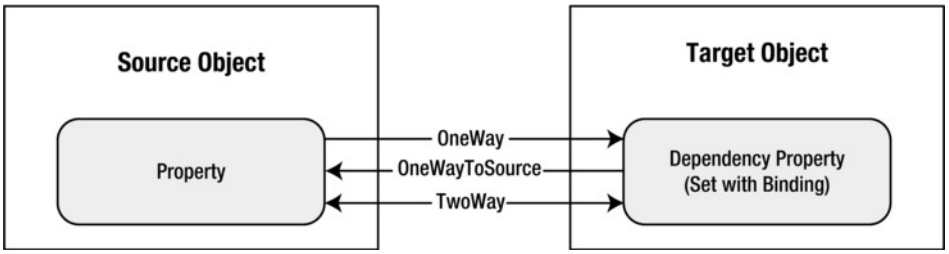

Figure 8-3. *Different ways to bind two properties*

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 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 by using the Binding markup extension. However, you can also create a binding by 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 by 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 by 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.

Retrieving Bindings in Code

You can also use code to retrieve a binding and examine its properties, regardless of whether that binding was originally created with code or markup.

There are two ways to get information about a binding. The first option is to use the static BindingOperations.GetBinding() method to retrieve the corresponding Binding object. You need to supply two arguments: the bound element, and the property that has the binding expression.

For example, if you have a binding like this:

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

You can use code like this to get the binding:

```
Binding binding = BindingOperations.GetBinding(lblSampleText, TextBlock.FontSize);
```

Once you have the Binding object, you can examine its properties. For example, Binding.ElementName provides the name of the bound element that supplies the value for your binding expression (in this case, it's "sliderFontSize"). Binding.Path provides the PropertyPath object that extracts the bound value from the bound object, and Binding.Path.Path gets the name of the bound property (in this case, "Value"). There's also a Binding.Mode property that tells you when the binding updates the target element.

The Binding object may be of some interest if you have to add diagnostic code while testing. But WPF also lets you get a more practical BindingExpression object by calling the BindingOperations.GetBindingExpression() method with the same arguments you used for the GetBinding() method:

```
BindingExpression expression = BindingOperations.GetBindingExpression(lblSampleText,
  TextBlock.FontSize);
```

The BindingExpression object includes a few properties that duplicate the information provided by the Binding object. But by far its most interesting feature is the ResolvedSource property, which allows you to evaluate the binding expression and get its result—the bound data that it's passing along. Here's an example:

```
// Get the source element.
Slider boundObject = (Slider)expression.ResolvedSource;

// Get any data you need from the source element, including its bound property.
string boundData = boundObject.FontSize;
```

This technique becomes more useful when you start binding data objects, as you'll do later in this chapter. Then, you can use the ResolvedSource property to get a reference to the bound data object whenever you need it.

Multiple Bindings

Although the previous example includes just a single binding, you don't need to stop there. If you wanted, you could set up the TextBlock 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.

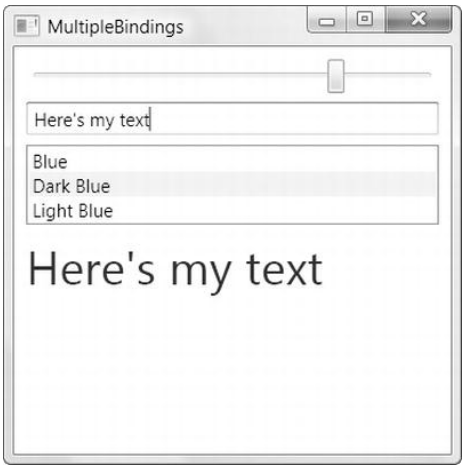

Figure 8-4. *A TextBlock that's bound to three elements*

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.

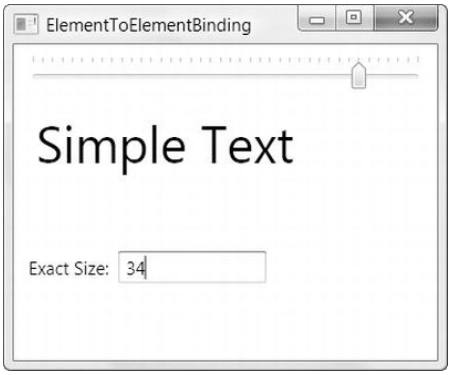

Figure 8-5. *Linking two properties to the font size*

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 by 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 Delays

In some rare cases, you might want to prevent data binding from springing into action and modifying the source object—at least, for a certain interval of time. For example, you might want to pause briefly before copying information from a text box, rather than grabbing it after every keystroke. Or perhaps your source object performs a processor-intensive operation when its data-bound property is changed. In this case, you might want to introduce a small delay so the operation isn't triggered quite as often.

In these specialized scenarios, you can make use of the Delay property of the Binding. It takes a number of milliseconds to wait, after which it commits the change. Here's a revised version of the text box example that waits for the user to stop typing for 500 milliseconds (half a second) before updating the source object:

```
<TextBox Text="{Binding ElementName=txtSampleText, Path=FontSize, Mode=TwoWay,
 UpdateSourceTrigger=PropertyChanged, Delay=500}" Name="txtFontSize"></TextBox>
```

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 directly to the source object—in other words, the object that's supplying the data.

RelativeSource: This is a reference that points to the source object by using a RelativeSource object. This extra layer allows you to base the reference on the current element (the element that holds the binding expression). Although this sounds needlessly complicated, the RelativeSource property is a specialized tool that's handy when writing control templates and data templates.

DataContext: If you don't specify a source by 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—typically, 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 the 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 by 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 by using the Source or ElementName property? However, this isn't always possible, usually because the source and target objects are in distinct sections 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 by 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 it's 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 WPF 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 by 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.