CHAPTER 4

Dependency Properties

Every .NET programmer is familiar with *properties* and *events*, which are core parts of .NET's object abstraction. Few would expect WPF, a user interface technology, to change either of these fundamentals. But surprisingly enough, that's exactly what WPF does.

In this chapter, you'll learn how WPF replaces ordinary .NET properties with a higher-level *dependency property* feature. Dependency properties use more-efficient storage and support additional features such as change notification and property value inheritance (the ability to propagate default values down the element tree). Dependency properties are also the basis for a number of key WPF features, including animation, data binding, and styles. Fortunately, even though the plumbing has changed, you can read and set dependency properties in code in exactly the same way as traditional .NET properties.

In the following pages, you'll take a close look at dependency properties. You'll see how they're defined, registered, and consumed. You'll also learn what features they support and what problems they solve.

Note Understanding dependency properties requires a heavy dose of theory, which you might not want to slog through just yet. If you can't wait to get started building an application, feel free to skip ahead to the following chapters and then return to this one when you need a deeper understanding of how WPF ticks and you want to build dependency properties of your own.

Understanding Dependency Properties

Dependency properties are a new implementation of standard .NET properties—one that has a significant amount of added value. You need dependency properties to plug into core WPF features such as animation, data binding, and styles.Most of the properties that are exposed by WPF elements are dependency properties. In all the examples you've seen up to this point, you've been using dependency properties without realizing it. That's because dependency properties are designed to be consumed in the same way as normal properties.

However, dependency properties are *not* normal properties. It's comforting to think of a dependency property as a normal property (defined in the typical .NET fashion) with a set of WPF features added on. Conceptually, dependency features behave this way, but that's not how they're implemented behind the scenes. The simple reason is performance. If the designers of WPF simply added extra features on top of the .NET property system, they'd need to create a complex, bulky layer for your code to travel through.

Ordinary properties could not support all the features of dependency properties without this extra overhead.

Dependency properties are a WPF-specific creation. However, the dependency properties in the WPF libraries are always wrapped by ordinary .NET property procedures. This makes them usable in the normal way, even with code that has no understanding of the WPF dependency property system. It seems odd to think of an older technology wrapping a newer one, but that's how WPF is able to change a fundamental ingredient such as properties without disrupting the rest of the .NET world.

Defining a Dependency Property

You'll spend much more time using dependency properties than creating them. However, there are still many reasons that you'll need to create your own dependency properties. Obviously, they're a key ingredient if you're designing a custom WPF element. However, they're also required in some cases if you want to add data binding, animation, or another WPF feature to a portion of code that wouldn't otherwise support it. Creating a dependency property isn't difficult, but the syntax takes a little getting used to. It's thoroughly different from creating an ordinary .NET property.

■ **Note** You can add dependency properties only to dependency objects—classes that derive from DependencyObject. Fortunately, most of the key pieces of WPF infrastructure derive indirectly from DependencyObject, with the most obvious example being elements.

The first step is to define an object that *represents* your property. This is an instance of the DependencyProperty class. The information about your property needs to be available all the time, and possibly even shared among classes (as is common with WPF elements). For that reason, your DependencyProperty object must be defined as a static field in the associated class.

For example, the FrameworkElement class defines a Margin property that all elements share. Unsurprisingly, Margin is a dependency property. That means it's defined in the FrameworkElement class like this:

```
public class FrameworkElement: UIElement, ...
{
    public static readonly DependencyProperty MarginProperty;

    ...
}
```

By convention, the field that defines a dependency property has the name of the ordinary property, plus the word *Property* at the end. That way, you can separate the dependency property definition from the name of the actual property. The field is defined with the `readonly` keyword, which means it can be set only in the static constructor for the FrameworkElement, which is the task you'll undertake next.

Registering a Dependency Property

Defining the DependencyProperty object is just the first step. For it to become usable, you need to register your dependency property with WPF. This step needs to be completed before any code uses the property, so it must be performed in a static constructor for the associated class.

WPF ensures that DependencyProperty objects can't be instantiated directly, because the DependencyProperty class has no public constructor. Instead, a DependencyObject instance can be created only by using the static DependencyProperty.Register() method. WPF also ensures that DependencyProperty objects can't be changed after they're created, because all DependencyProperty members are read-only. Instead, their values must be supplied as arguments to the Register() method.

The following code shows an example of how a DependencyProperty must be created. Here, the FrameworkElement class uses a static constructor to initialize the MarginProperty:

```
static FrameworkElement()
{
    FrameworkPropertyMetadata metadata = new FrameworkPropertyMetadata(
      new Thickness(), FrameworkPropertyMetadataOptions.AffectsMeasure);

    MarginProperty = DependencyProperty.Register("Margin",
      typeof(Thickness), typeof(FrameworkElement), metadata,
      new ValidateValueCallback(FrameworkElement.IsMarginValid));
    ...
}
```

There are two steps involved in registering a dependency property. First, you create a FrameworkPropertyMetadata object that indicates what services you want to use with your dependency property (such as support for data binding, animation, and journaling). Next, you register the property by calling the static DependencyProperty.Register() method. At this point, you are responsible for supplying a few key ingredients:

- The property name (Margin in this example)
- The data type used by the property (the Thickness structure in this example)
- The type that owns this property (the FrameworkElement class in this example)
- Optionally, a FrameworkPropertyMetadata object with additional property settings
- Optionally, a callback that performs validation for the property

The first three details are all straightforward. The FrameworkPropertyMetadata object and the validation callback are more interesting.

You use FrameworkPropertyMetadata to configure additional features for your dependency property. Most of the properties of the FrameworkPropertyMetadata class are simple Boolean flags that you set to flip on a feature. (The default value for each Boolean flag is false.) A few are callbacks that point to custom methods that you create to perform a specific task. One—FrameworkPropertyMetadata.DefaultValue—sets the default value that WPF will apply when the property is first initialized. Table 4-1 lists all the FrameworkPropertyMetadata properties.

Table 4-1. *Properties of the FrameworkPropertyMetadata Class*

Name	Description
AffectsArrange, AffectsMeasure, AffectsParentArrange, and AffectsParentMeasure	If true, the dependency property may affect how adjacent elements (or the parent element) are placed during the measure pass and the arrange pass of a layout operation. For example, the Margin dependency property sets AffectsMeasure to true, signaling that if the margin of an element changes, the layout container needs to repeat the measure step to determine the new placement of elements.
AffectsRender	If true, the dependency property may affect something about the way an element is drawn, requiring that the element be repainted.

BindsTwoWayByDefault	If true, this dependency property will use two-way data binding instead of one-way data binding by default. However, you can specify the binding behavior you want explicitly when you create the binding.
Inherits	If true, the dependency property value propagates through the element tree and can be inherited by nested elements. For example, Font is an inheritable dependency property—if you set it on a higher-level element, it's inherited by nested elements, unless they explicitly override it with their own font settings.
IsAnimationProhibited	If true, the dependency property can't be used in an animation.
IsNotDataBindable	If true, the dependency property can't be set with a binding expression.
Journal	If true, this dependency property will be persisted to the journal (the history of visited pages) in a page-based application.
SubPropertiesDoNotAffectRender	If true, WPF will not rerender an object if one of its subproperties (the property of a property) changes.
DefaultUpdateSourceTrigger	This sets the default value for the Binding.UpdateSourceTrigger property when this property is used in a binding expression. The UpdateSourceTrigger determines when a data-bound value applies its changes. You can set the UpdateSourceTrigger property manually when you create the binding.
DefaultValue	This sets the default value for the dependency property.
CoerceValueCallback	This provides a callback that attempts to "correct" a property value before it's validated.
PropertyChangedCallback	This provides a callback that is called when a property value is changed.

In the following sections, you'll take a closer look at the validation callback and some of the metadata options. You'll also see a few more of them at work in examples throughout this book. But first, you need to understand how you can make sure every dependency property can be accessed in the same way as a traditional .NET property.

Adding a Property Wrapper

The final step to creating a dependency property is to wrap it in a traditional .NET property. However, whereas typical property procedures retrieve or set the value of a private field, the property procedures for a WPF property use the GetValue() and SetValue() methods that are defined in the base DependencyObject class. Here's an example:

```
public Thickness Margin
{
    set { SetValue(MarginProperty, value); }
    get { return (Thickness)GetValue(MarginProperty); }
}
```

When you create the property wrapper, you should include nothing more than a call to SetValue() and a call to GetValue(), as in the previous example. You should *not* add any extra code to validate values, raise events, and so on. That's because other features in WPF may bypass the property wrapper and call

SetValue() and GetValue() directly. (One example is when a compiled XAML file is parsed at runtime.) Both SetValue() and GetValue() are public.

■ **Note** The property wrapper isn't the right place to validate data or raise an event. However, WPF does provide a place for this code; the trick is to use dependency property callbacks. Validation should be performed through the DependencyProperty.ValidateValueCallback shown previously, while events can be raised from the FrameworkPropertyMetadata.PropertyChangedCallback shown in the next section.

You now have a fully functioning dependency property, which you can set just like any other .NET property, using the property wrapper:

```
myElement.Margin = new Thickness(5);
```

There's one extra detail. Dependency properties follow strict rules of precedence to determine their current value. Even if you don't set a dependency property directly, it may already have a value—perhaps one that's applied by a binding, style, or animation, or one that's inherited through the element tree. (You'll learn more about these rules of precedence in the next section, "How WPF Uses Dependency Properties.") However, as soon as you set the value directly, it overrides all these other influences.

At some point later, you may want to remove your local value setting and let the property value be determined as though you never set it. Obviously, you can't accomplish this by setting a new value. Instead, you need to use another method that's inherited from DependencyObject: the ClearValue() method. Here's how it works:

```
myElement.ClearValue(FrameworkElement.MarginProperty);
```

How WPF Uses Dependency Properties

As you'll discover throughout this book, dependency properties are required for a range of WPF features. However, all of these features work through two key behaviors that every dependency property supports—change notification and dynamic value resolution.

Contrary to what you might expect, dependency properties do *not* automatically fire events to let you know when a property value changes. Instead, they trigger a protected method named OnPropertyChangedCallback(). This method passes the information along to two WPF services (data binding and triggers). It also calls the PropertyChangedCallback, if one is defined.

In other words, if you want to react when a property changes, you have two choices—you can create a binding that uses the property value (Chapter 8), or you can write a trigger that automatically changes another property or starts an animation (Chapter 11). However, dependency properties don't give you a general-purpose way to fire off some code to respond to a property change.

■ **Note** If you're dealing with a control that you've created, you can use the property callback mechanism to react to property changes and even raise an event. Many common controls use this technique for properties that correspond to user-supplied information. For example, the TextBox provides a TextChanged event, and the ScrollBar provides a ValueChanged event. A control can implement functionality like this by using the PropertyChangedCallback, but this functionality isn't exposed from dependency properties in a general way for performance reasons.

The second feature that's key to the way dependency properties work is dynamic value resolution. This means when you retrieve the value from a dependency property, WPF takes several factors into consideration.

This behavior gives dependency properties their name—in essence, a dependency property *depends* on multiple property providers, each with its own level of precedence. When you retrieve a value from a property value, the WPF property system goes through a series of steps to arrive at the final value. First, it determines the base value for the property by considering the following factors, arranged from lowest to highest precedence:

1. The default value (as set by the FrameworkPropertyMetadata object)
2. The inherited value (if the FrameworkPropertyMetadata.Inherits flag is set and a value has been applied to an element somewhere up the containment hierarchy)
3. The value from a theme style (as discussed in Chapter 18)
4. The value from a project style (as discussed in Chapter 11)
5. The local value (in other words, a value you've set directly on this object by using code or XAML)

As this list shows, you override the entire hierarchy by applying a value directly. If you don't, the value is determined by the next applicable item up on the list.

■ **Note** One of the advantages of this system is that it's very economical. If the value of a property has not been set locally, WPF will retrieve its value from a style, another element, or the default. In this case, no memory is required to store the value. You can quickly see the savings if you add a few buttons to a form. Each button has dozens of properties, which, if they are set through one of these mechanisms, use no memory at all.

WPF follows the previous list to determine the *base value* of a dependency property. However, the base value is not necessarily the final value that you'll retrieve from a property. That's because WPF considers several other providers that can change a property's value.

Here's the four-step process WPF follows to determine a property value:

1. Determine the base value (as described previously).
2. If the property is set by using an expression, evaluate that expression. Currently, WPF supports two types of expressions: data binding (Chapter 8) and resources (Chapter 10).
3. If this property is the target of animation, apply that animation.
4. Run CoerceValueCallback to "correct" the value. (You'll learn how to use this technique later, in the "Property Validation" section.)

Essentially, dependency properties are hardwired into a small set of WPF services. If it weren't for this infrastructure, these features would add unnecessary complexity and significant overhead.

■ **Tip** In future versions of WPF, the dependency property pipeline could be extended to include additional services. When you design custom elements (a topic covered in Chapter 18), you'll probably use dependency properties for most (if not all) of their public properties.

Shared Dependency Properties

Some classes share the same dependency property, even though they have separate class hierarchies. For example, both TextBlock.FontFamily and Control.FontFamily point to the same static dependency property, which is actually defined in the TextElement class and TextElement.FontFamilyProperty. The static constructor of TextElement registers the property, but the static constructors of TextBlock and Control simply reuse it by calling the DependencyProperty.AddOwner() method:

```
TextBlock.FontFamilyProperty =
  TextElement.FontFamilyProperty.AddOwner(typeof(TextBlock));
```

You can use the same technique when you create your own custom classes (assuming the property is not already provided in the class you're inheriting from, in which case you get it for free). You can also use an overload of the AddOwner() method that allows you to supply a validation callback and a new FrameworkPropertyMetadata that will apply only to this new use of the dependency property.

Reusing dependency properties can lead to some strange side effects in WPF, most notably with styles. For example, if you use a style to set the TextBlock.FontFamily property automatically, your style will also affect the Control.FontFamily property, because behind the scenes both classes use the same dependency property. You'll see this phenomenon in action in Chapter 11.

Attached Dependency Properties

Chapter 2 introduced a special type of dependency property called an *attached property*. An attached property is a dependency property, and it's managed by the WPF property system. The difference is that an attached property applies to a class other than the one where it's defined.

The most common example of attached properties is found in the layout containers described in Chapter 3. For example, the Grid class defines the attached properties Row and Column, which you set on the contained elements to indicate where they should be positioned. Similarly, the DockPanel defines the attached property Dock, and the Canvas defines the attached properties Left, Right, Top, and Bottom.

To define an attached property, you use the RegisterAttached() method instead of Register(). Here's an example that registers the Grid.Row property:

```
FrameworkPropertyMetadata metadata = new FrameworkPropertyMetadata(
  0, new PropertyChangedCallback(Grid.OnCellAttachedPropertyChanged));

Grid.RowProperty = DependencyProperty.RegisterAttached("Row", typeof(int),
  typeof(Grid), metadata, new ValidateValueCallback(Grid.IsIntValueNotNegative));
```

As with an ordinary dependency property, you can supply a FrameworkPropertyMetadata object and a ValidateValueCallback.

When creating an attached property, you don't define the .NET property wrapper. That's because attached properties can be set on *any* dependency object. For example, the Grid.Row property may be set on a Grid object (if you have one Grid nested inside another) or on some other element. In fact, the Grid. Row property can be set on an element even if that element isn't in a Grid—and even if there isn't a single Grid object in your element tree.

Instead of using a .NET property wrapper, attached properties require a pair of static methods that can be called to set and get the property value. These methods use the familiar SetValue() and GetValue() methods (inherited from the DependencyObject class). The static methods should be named Set*PropertyName*() and Get*PropertyName*().

Here are the static methods that implement the Grid.Row attached property:

```
public static int GetRow(UIElement element)
{
    if (element == null)
    {
        throw new ArgumentNullException(...);
    }
    return (int)element.GetValue(Grid.RowProperty);
}

public static void SetRow(UIElement element, int value)
{
    if (element == null)
    {
        throw new ArgumentNullException(...);
    }
    element.SetValue(Grid.RowProperty, value);
}
```

Here's an example that uses code to position an element in the first row of a Grid:

```
Grid.SetRow(txtElement, 0);
```

Alternatively, you can call the SetValue() or GetValue() method directly and bypass the static methods:

```
txtElement.SetValue(Grid.RowProperty, 0);
```

The SetValue() method also provides one brain-twisting oddity. Although XAML doesn't allow it, you can use an overloaded version of the SetValue() method in code to attach a value for any dependency property, *even if that property isn't defined as an attached property*. For example, the following code is perfectly legitimate:

```
ComboBox comboBox = new ComboBox();
...
comboBox.SetValue(PasswordBox.PasswordCharProperty, "*");
```

Here, a value for the PasswordBox.PasswordChar property is set for a ComboBox object, even though PasswordBox.PasswordCharProperty is registered as an ordinary dependency property, not an attached property. This action won't change the way the ComboBox works—after all, the code inside the ComboBox won't look for the value of a property that it doesn't know exists—but you could act upon the PasswordChar value in your own code.

Although rarely used, this quirk provides some more insight into the way the WPF property system works, and it demonstrates its remarkable extensibility. It also shows that even though attached properties are registered with a different method than normal dependency properties, in the eyes of WPF there's no real distinction. The only difference is what the XAML parser allows. Unless you register your property as an attached property, you won't be able to set it on other elements in your markup.

Property Validation

When defining any sort of property, you need to face the possibility that it may be set incorrectly. With traditional .NET properties, you might try to catch this sort of problem in the property setter. With dependency properties, this isn't appropriate, because the property may be set directly through the WPF property system by using the SetValue() method.

Instead, WPF provides two ways to prevent invalid values:

ValidateValueCallback: This callback can accept or reject new values. Usually, this callback is used to catch obvious errors that violate the constraints of the property. You can supply it as an argument to the DependencyProperty.Register() method.

CoerceValueCallback: This callback can change new values into something more acceptable. Usually, this callback is used to deal with conflicting dependency property values that are set on the same object. These values might be independently valid but aren't consistent when applied together. To use this callback, supply it as a constructor argument when creating the FrameworkPropertyMetadata object, which you then pass to the DependencyProperty.Register() method.

Here's how all the pieces come into play when an application attempts to set a dependency property:

1. First, the CoerceValueCallback method has the opportunity to modify the supplied value (usually, to make it consistent with other properties) or return DependencyProperty.UnsetValue, which rejects the change altogether.
2. Next, the ValidateValueCallback is fired. This method returns true to accept a value as valid or returns false to reject it. Unlike the CoerceValueCallback, the ValidateValueCallback does not have access to the actual object on which the property is being set, which means you can't examine other property values.
3. Finally, if both these previous stages succeed, the PropertyChangedCallback is triggered. At this point, you can raise a change event if you want to provide notification to other classes.

The Validation Callback

As you saw earlier, the DependencyProperty.Register() method accepts an optional validation callback:

```
MarginProperty = DependencyProperty.Register("Margin",
  typeof(Thickness), typeof(FrameworkElement), metadata,
  new ValidateValueCallback(FrameworkElement.IsMarginValid));
```

You can use this callback to enforce the validation that you'd normally add in the set portion of a property procedure. The callback you supply must point to a method that accepts an object parameter and returns a Boolean value. You return true to accept the object as valid and false to reject it.

The validation of the FrameworkElement.Margin property isn't terribly interesting because it relies on an internal Thickness.IsValid() method. This method makes sure the Thickness object is valid for its current use (representing a margin). For example, it may be possible to construct a perfectly acceptable Thickness object that isn't acceptable for setting the margin. One example is a Thickness object with negative dimensions. If the supplied Thickness object isn't valid for a margin, the IsMarginValid property returns false:

```
private static bool IsMarginValid(object value)
{
    Thickness thickness1 = (Thickness) value;
    return thickness1.IsValid(true, false, true, false);
}
```

There's one limitation with validation callbacks: they are static methods that don't have access to the object that's being validated. All you get is the newly applied value. Although that makes them easier to reuse, it also makes it impossible to create a validation routine that takes other properties into account. The classic example is an element with Maximum and Minimum properties. Clearly, it should not be possible to set the Maximum to a value that's less than the Minimum. However, you can't enforce this logic with a validation callback because you'll have access to only one property at a time.

■ **Note** The preferred approach to solve this problem is to use value coercion. *Coercion* is a step that occurs just before validation, and it allows you to modify a value to make it more acceptable (for example, raising the Maximum so it's at least equal to the Minimum) or disallow the change altogether. The coercion step is handled through another callback, but this one is attached to the FrameworkPropertyMetadata object, which is described in the next section.

The Coercion Callback

You use the CoerceValueCallback through the FrameworkPropertyMetadata object. Here's an example:

```
FrameworkPropertyMetadata metadata = new FrameworkPropertyMetadata();
metadata.CoerceValueCallback = new CoerceValueCallback(CoerceMaximum);

DependencyProperty.Register("Maximum", typeof(double),
  typeof(RangeBase), metadata);
```

The CoerceValueCallback allows you to deal with interrelated properties. For example, the ScrollBar provides Maximum, Minimum, and Value properties, all of which are inherited from the RangeBase class. One way to keep these properties aligned is to use property coercion.

For example, when the Maximum is set, it must be coerced so that it can't be less than the Minimum:

```
private static object CoerceMaximum(DependencyObject d, object value)
{
    RangeBase base1 = (RangeBase)d;
    if (((double) value) < base1.Minimum)
    {
        return base1.Minimum;
    }
    return value;
}
```

In other words, if the value that's applied to the Maximum property is less than the Minimum, the Minimum value is used instead to cap the Maximum. Notice that the CoerceValueCallback passes two parameters—the value that's being applied *and* the object to which it's being applied.

When the Value is set, a similar coercion takes place. The Value property is coerced so that it can't fall outside the range defined by the Minimum and Maximum, using this code:

```
internal static object ConstrainToRange(DependencyObject d, object value)
{
    double newValue = (double)value;
    RangeBase base1 = (RangeBase)d;

    double minimum = base1.Minimum;
    if (newValue < minimum)
    {
        return minimum;
    }
    double maximum = base1.Maximum;
    if (newValue > maximum)
    {
        return maximum;
    }
    return newValue;
}
```

The Minimum property doesn't use value coercion at all. Instead, after it has been changed, it triggers a PropertyChangedCallback which forces the Maximum and Value properties to follow along by manually triggering *their* coercion:

```
private static void OnMinimumChanged(DependencyObject d,
  DependencyPropertyChangedEventArgs e)
{
    RangeBase base1 = (RangeBase)d;
    ...
    base1.CoerceMaximum(RangeBase.MaximumProperty);
    base1.CoerceValue(RangeBase.ValueProperty);
}
```

Similarly, after the Maximum has been set and coerced, it manually coerces the Value property to fit:

```
private static void OnMaximumChanged(DependencyObject d,
  DependencyPropertyChangedEventArgs e)
{
    RangeBase base1 = (RangeBase)d;
    ...
    base1.CoerceValue(RangeBase.ValueProperty);
    base1.OnMaximumChanged((double) e.OldValue, (double)e.NewValue);
}
```

The end result is that if you set conflicting values, the Minimum takes precedence, the Maximum gets its say next (and may possibly be coerced by the Minimum), and then the Value is applied (and may be coerced by both the Maximum and the Minimum).

The goal of this somewhat confusing sequence of steps is to ensure that the ScrollBar properties can be set in various orders without causing an error. This is an important consideration for initialization, such as when a window is being created for a XAML document. All WPF controls guarantee that their properties can be set in any order, without causing any change in behavior.

A careful review of the previous code calls this goal into question. For example, consider this code:

```
ScrollBar bar = new ScrollBar();
bar.Value = 100;
bar.Minimum = 1;
bar.Maximum = 200;
```

When the ScrollBar is first created, Value is 0, Minimum is 0, and Maximum is 1.

After the second line of code, the Value property is coerced to 1 (because initially the Maximum property is set to the default value 1). But something remarkable happens when you reach the fourth line of code. When the Maximum property is changed, it triggers coercion on both the Minimum and Value properties. This coercion acts on the values you specified *originally*. In other words, the local value of 100 is still stored by the WPF dependency property system, and now that it's an acceptable value, it can be applied to the Value property. Thus, after this single line of code executes, two properties have changed. Here's a closer look at what's happening:

```
ScrollBar bar = new ScrollBar();
bar.Value = 100;
// (Right now bar.Value returns 1.)
bar.Minimum = 1;
// (bar.Value still returns 1.)
bar.Maximum = 200;
// (Now now bar.Value returns 100.)
```

This behavior persists no matter when you set the Maximum property. For example, if you set a Value of 100 when the window loads and set the Maximum property later when the user clicks a button, the Value property is still restored to its rightful value of 100 at that point. (The only way to prevent this from taking place is to set a different value or remove the local value that you've applied by using the ClearValue() method that all elements inherit from DependencyObject.)

This behavior is due to WPF's property resolution system, which you learned about earlier. Although WPF stores the exact local value you've set internally, it *evaluates* what the property should be (using coercion and a few other considerations) when you read the property.

The Last Word

In this chapter, you took a deep look at WPF dependency properties. First, you saw how dependency properties are defined and registered. Next, you learned how they plug into other WPF services and support validation and coercion. In the next chapter, you'll explore another WPF feature that extends a core part of the traditional .NET infrastructure: routed events.

■ **Tip** One of the best ways to learn more about the internals of WPF is to browse the code for basic WPF elements, such as Button, UIElement, and FrameworkElement. One of the best tools to perform this browsing is Reflector, which is available at `www.reflector.net`. Using Reflector, you can see the definitions for dependency properties, browse through the static constructor code that initializes them, and even explore how they're used in the class code. You can also get similar low-level information about routed events, which are described in the next chapter.
