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 databound 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.

Name	Description
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 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 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 using an expression, evaluate that expression. Currently, WPF supports two types of expression: data binding (Chapter 8) and resources (Chapter 10).
3. If this property is the target of animation, apply that animation.
4. Run the 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 positions an element in the first row of a Grid using code:

```
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 in 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 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 only to 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 of 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, once it has been changed, it triggers a PropertyChangedCallback that 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, once 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 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.

■ **Note** Long-time Windows Forms programmers may remember the ISupportInitialize interface, which was used to solve similar problems in property initialization by wrapping a series of property changes into a batch process. Although you can use ISupportInitialize with WPF (and the XAML parser respects it), few of the WPF elements use this technique. Instead, it's encouraged to resolve these problems using value coercion. There are a number of reasons that coercion is preferred. For example, coercion solves other problems that can occur when an invalid value is applied through a data binding or animation, unlike the ISupportInitialize interface.

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 `http://www.red-gate.com/products/reflector`. 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.

CHAPTER 5

■ ■ ■

Routed Events

In the previous chapter, you saw how WPF created a new dependency property system, reworking traditional .NET properties to improve performance and integrate new capabilities such as data binding and animation. In this chapter, you'll learn about the second shift: replacing ordinary .NET events with a higher-level *routed event* feature.

Routed events are events with more traveling power—they can tunnel down or bubble up the element tree and be processed by event handlers along the way. Routed events allow an event to be handled on one element (such as a label) even though it originates on another (such as an image inside that label). As with dependency properties, routed events can be consumed in the traditional way—by connecting an event handler with the right signature—but you need to understand how they work to unlock all their features.

In this chapter, you'll explore the WPF event system and learn how to fire and handle routed events. Once you've learned the basics, you'll consider the family of events that WPF elements provide. These include events for dealing with initialization, mouse and keyboard input, and multitouch devices.

■ **What's New** Dependency properties and routed events work the same in WPF 4 as they did in all earlier versions. However, WPF 4 introduces one entirely new feature: the ability to capture input from next-generation multitouch devices (for example, tablet computers with sophisticated touchscreens). Multitouch is covered in the "Multitouch Input" section later this chapter.

Understanding Routed Events

Every .NET developer is familiar with the idea of *events*—messages that are sent by an object (such as a WPF element) to notify your code when something significant occurs. WPF enhances the .NET event model with the concept of *event routing*. Event routing allows an event to originate in one element but be raised by another one. For example, event routing allows a click that begins in a toolbar button to rise up to the toolbar and then to the containing window before it's handled by your code.

Event routing gives you the flexibility to write tight, well-organized code that handles events in the most convenient place. It's also a necessity for working with the WPF content model, which allows you to build simple elements (such as a button) out of dozens of distinct ingredients, each of which has its own independent set of events.

Defining, Registering, and Wrapping a Routed Event

The WPF event model is quite similar to the WPF property model. As with dependency properties, routed events are represented by read-only static fields, registered in a static constructor, and wrapped by a standard .NET event definition.

For example, the WPF Button class provides the familiar Click event, which is inherited from the abstract ButtonBase class. Here's how the event is defined and registered:

```
public abstract class ButtonBase : ContentControl, ...
{
    // The event definition.
    public static readonly RoutedEvent ClickEvent;

    // The event registration.
    static ButtonBase()
    {
        ButtonBase.ClickEvent = EventManager.RegisterRoutedEvent(
          "Click", RoutingStrategy.Bubble,
          typeof(RoutedEventHandler), typeof(ButtonBase));
        ...
    }

    // The traditional event wrapper.
    public event RoutedEventHandler Click
    {
        add
        {
            base.AddHandler(ButtonBase.ClickEvent, value);
        }
        remove
        {
            base.RemoveHandler(ButtonBase.ClickEvent, value);
        }
    }

    ...
}
```

While dependency properties are registered with the DependencyProperty.Register() method, routed events are registered with the EventManager.RegisterRoutedEvent() method. When registering an event, you need to specify the name of the event, the type of routine (more on that later), the delegate that defines the syntax of the event handler (in this example, RoutedEventHandler), and the class that owns the event (in this example, ButtonBase).

Usually, routed events are wrapped by ordinary .NET events to make them accessible to all .NET languages. The event wrapper adds and removes registered callers using the AddHandler() and RemoveHandler() methods, both of which are defined in the base FrameworkElement class and inherited by every WPF element.

Sharing Routed Events

As with dependency properties, the definition of a routed event can be shared between classes. For example, two base classes use the MouseUp event: UIElement (which is the starting point for ordinary WPF elements) and ContentElement (which is the starting point for content elements, which are individual bits of content that can be placed in a flow document). The MouseUp event is defined by the System.Windows.Input.Mouse class. The UIElement and ContentElement classes simply reuse it with the RoutedEvent.AddOwner() method:

```
UIElement.MouseUpEvent = Mouse.MouseUpEvent.AddOwner(typeof(UIElement));
```

Raising a Routed Event

Of course, like any event, the defining class needs to raise it at some point. Exactly where this takes place is an implementation detail. However, the important detail is that your event is *not* raised through the traditional .NET event wrapper. Instead, you use the RaiseEvent() method that every element inherits from the UIElement class. Here's the appropriate code from deep inside the ButtonBase class:

```
RoutedEventArgs e = new RoutedEventArgs(ButtonBase.ClickEvent, this);
base.RaiseEvent(e);
```

The RaiseEvent() method takes care of firing the event to every caller that's been registered with the AddHandler() method. Because AddHandler() is public, callers have a choice—they can register themselves directly by calling AddHandler(), or they can use the event wrapper. (The following section demonstrates both approaches.) Either way, they'll be notified when the RaiseEvent() method is invoked.

All WPF events use the familiar .NET convention for event signatures. That first parameter of every event handler provides a reference to the object that fired the event (the sender). The second parameter is an EventArgs object that bundles together any additional details that might be important. For example, the MouseUp event provides a MouseEventArgs object that indicates what mouse buttons were pressed when the event occurred:

```
private void img_MouseUp(object sender, MouseButtonEventArgs e)
{
}
```

In Windows Forms applications, it was customary for many events to use the base EventArgs class if they didn't need to pass along any extra information. However, the situation is different in WPF applications because of their support for the routed event model.

In WPF, if an event doesn't need to send any additional details, it uses the RoutedEventArgs class, which includes some details about how the event was routed. If the event *does* need to transmit extra information, it uses a more specialized RoutedEventArgs-derived object (such as MouseButtonEventArgs in the previous example). Because every WPF event argument class derives from RoutedEventArgs, every WPF event handler has access to information about event routing.

Handling a Routed Event

As you learned in Chapter 2, here are several ways to attach an event handler..The most common approach is to add an event attribute to your XAML markup. The event attribute is named after the event you want to handle, and its value is the name of the event handler method. Here's an example that uses this syntax to connect the MouseUp event of the Image to an event handler named img_MouseUp:

```
<Image Source="happyface.jpg" Stretch="None"
 Name="img" MouseUp="img_MouseUp" />
```

Although it's not required, it's a common convention to name event handler methods in the form ElementName_EventName. If the element doesn't have a defined name (presumably because you don't need to interact with it in any other place in your code), consider using the name it *would* have:

```
<Button Click="cmdOK_Click">OK</Button>
```

■ **Tip** It may be tempting to attach an event to a high-level method that performs a task, but you'll have more flexibility if you keep an extra layer of event handling code. For example, when you click a button named cmdUpdate, it shouldn't trigger a method named UpdateDatabase() directly. Instead, it should call an event handler such as cmdUpdate_Click(), which can then call the UpdateDatabase() method that does the real work. This pattern gives you the flexibility to change where your database code is located, replace the update button with a different control, and wire several controls to the same process, all without limiting your ability to change the user interface later. If you want a simpler way to deal with actions that can be triggered from several different places in a user interface (toolbar buttons, menu commands, and so on), you'll want to add the WPF command feature that's described in Chapter 9.

You can also connect an event with code. Here's the code equivalent of the XAML markup shown previously:

```
img.MouseUp += new MouseButtonEventHandler(img_MouseUp);
```

This code creates a delegate object that has the right signature for the event (in this case, an instance of the MouseButtonEventHandler delegate) and points that delegate to the img_MouseUp() method. It then adds the delegate to the list of registered event handlers for the img.MouseUp event.

C# also allows a more streamlined syntax that creates the appropriate delegate object implicitly:

```
img.MouseUp += img_MouseUp;
```

The code approach is useful if you need to dynamically create a control and attach an event handler at some point during the lifetime of your window. By comparison, the events you hook up in XAML are always attached when the window object is first instantiated. The code approach also allows you to keep your XAML simpler and more streamlined, which is perfect if you plan to share it with nonprogrammers, such as a design artist. The drawback is a significant amount of boilerplate code that will clutter up your code files.

The previous code approach relies on the event wrapper, which calls the UIElement.AddHandler() method, as shown in the previous section. You can also connect an event directly by calling UIElement.AddHandler() method yourself. Here's an example:

```
img.AddHandler(Image.MouseUpEvent,
  new MouseButtonEventHandler(img_MouseUp));
```

When you use this approach, you always need to create the appropriate delegate type (such as MouseButtonEventHandler). You can't create the delegate object implicitly, as you can when hooking up an event through the property wrapper. That's because the UIElement.AddHandler() method supports all WPF events and it doesn't know the delegate type that you want to use.

Some developers prefer to use the name of the class where the event is defined, rather than the name of the class that is firing the event. Here's the equivalent syntax that makes it clear that the MouseUpEvent is defined in UIElement:

```
img.AddHandler(UIElement.MouseUpEvent,
  new MouseButtonEventHandler(img_MouseUp));
```

■ **Note** Which approach you use is largely a matter of taste. However, the drawback to this second approach is that it doesn't make it obvious that the Image class *provides* a MouseUpEvent. It's possible to confuse this code and assume it's attaching an event handler that's meant to deal with the MouseUpEvent in a nested element. You'll learn more about this technique in the section "Attached Events" later in this chapter.

If you want to detach an event handler, code is your only option. You can use the -= operator, as shown here:

```
img.MouseUp -= img_MouseUp;
```

Or you can use the UIElement.RemoveHandler() method:

```
img.RemoveHandler(Image.MouseUpEvent,
  new MouseButtonEventHandler(img_MouseUp));
```

It is technically possible to connect the same event handler to the same event more than once. This is usually the result of a coding mistake. (In this case, the event handler will be triggered multiple times.) If you attempt to remove an event handler that's been connected twice, the event will still trigger the event handler but just once.

Event Routing

As you learned in the previous chapter, many controls in WPF are content controls, and content controls can hold any type and amount of nested content. For example, you can build a graphical button out of shapes, create a label that mixes text and pictures, or put content in a specialized container to get a scrollable or collapsible display. You can even repeat this nesting process to go as many layers deep as you want.

This ability for arbitrary nesting raises an interesting question. For example, imagine you have a label like this one, which contains a StackPanel that brings together two blocks of text and an image:

```
<Label BorderBrush="Black" BorderThickness="1">
  <StackPanel>
    <TextBlock Margin="3">
     Image and text label</TextBlock>
    <Image Source="happyface.jpg" Stretch="None" />
    <TextBlock Margin="3">
     Courtesy of the StackPanel</TextBlock>
  </StackPanel>
</Label>
```

As you already know, every ingredient you place in a WPF window derives from UIElement at some point, including the Label, StackPanel, TextBlock, and Image. UIElement defines some core events. For example, every class that derives from UIElement provides a MouseDown and MouseUp event.

But consider what happens when you click the image part of the fancy label shown here. Clearly, it makes sense for the Image.MouseDown and Image.MouseUp events to fire. But what if you want to treat all label clicks in the same way? In this case, it shouldn't matter whether the user clicks the image, some of the text, or part of the blank space inside the label border. In every case, you'd like to respond with the same code.

Clearly, you could wire up the same event handler to the MouseDown or MouseUp event of each element, but that would result in a significant amount of clutter and make your markup more difficult to maintain. WPF provides a better solution with its routed event model.

Routed events actually come in the following three flavors:

- **Direct events.** These are like ordinary .NET events. They originate in one element and don't pass to any other. For example, MouseEnter (which fires when the mouse pointer moves over an element) is a direct event.
- **Bubbling events.** These events travel *up* the containment hierarchy. For example, MouseDown is a bubbling event. It's raised first by the element that is clicked. Next, it's raised by that element's parent, then by *that* element's parent, and so on, until WPF reaches the top of the element tree.
- **Tunneling events.** These events travel *down* the containment hierarchy. They give you the chance to preview (and possibly stop) an event before it reaches the appropriate control. For example, PreviewKeyDown allows you to intercept a key press, first at the window level and then in increasingly more specific containers until you reach the element that had focus when the key was pressed.

When you register a routed event using the EventManager.RegisterEvent() method, you pass a value from the RoutingStrategy enumeration that indicates the event behavior you want to use for your event.

Because MouseUp and MouseDown are bubbling events, you can now determine what happens in the fancy label example. When the happy face is clicked, the MouseDown event fires in this order:

1. Image.MouseDown
2. StackPanel.MouseDown
3. Label.MouseDown

After the MouseDown event is raised for the label, it's passed on to the next control (which in this case is the Grid that lays out the containing window) and then to its parent (the window). The window is the top level of the containment hierarchy and the final stop in the event bubbling sequence. It's your last chance to handle a bubbling event such as MouseDown. If the user releases the mouse button, the MouseUp event fires in the same sequence.

■ **Note** In Chapter 24, you'll learn how to create a page-based WPF application. In this situation, the top-level container isn't a window but an instance of the Page class.

You aren't limited to handling a bubbling event in one place. In fact, there's no reason why you can't handle the MouseDown or MouseUp event at every level. But usually you'll choose the most appropriate level !!!event routing for the task at hand.

The RoutedEventArgs Class

When you handle a bubbling event, the sender parameter provides a reference to the last link in the chain. For example, if an event bubbles up from an image to a label before you handle it, the sender parameter references the label object.

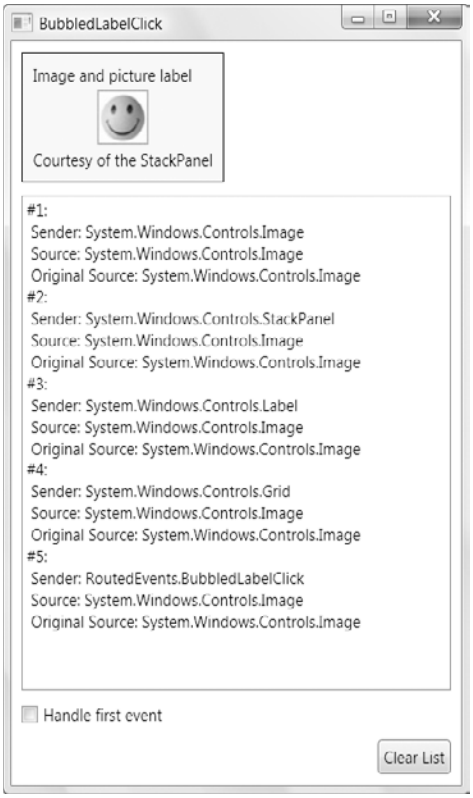
In some cases, you'll want to determine where the event originally took place. You can get that information and other details from the properties of the RoutedEventArgs class (which are listed in Table 5-1). Because all WPF event argument classes inherit from RoutedEventArgs, these properties are available in any event handler.

Table 5-1. *Properties of the RoutedEventArgs Class*

Name	Description
Source	Indicates what object raised the event. In the case of a keyboard event, this is the control that had focus when the event occurred (for example, when the key was pressed). In the case of a mouse event, this is the topmost element under the mouse pointer when the event occurred (for example, when a mouse button was clicked).
OriginalSource	Indicates what object originally raised the event. Usually, the OriginalSource is the same as the source. However, in some cases the OriginalSource goes deeper in the object tree to get a behind-the-scenes element that's part of a higher-level element. For example, if you click close to the border of a window, you'll get a Window object for the event source but a Border object for the original source. That's because a Window is composed out of individual, smaller components. To take a closer look at this composition model (and learn how to change it), head to Chapter 17, which discusses control templates.
RoutedEvent	Provides the RoutedEvent object for the event triggered by your event handler (such as the static UIElement.MouseUpEvent object). This information is useful if you're handling different events with the same event handler.
Handled	Allows you to halt the event bubbling or tunneling process. When a control sets the Handled property to true, the event doesn't travel any further and isn't raised for any other elements. (As you'll see in the section "Handling a Suppressed Event," there is one way around this limitation.)

Bubbling Events

Figure 5-1 shows a simple window that demonstrates event bubbling. When you click a part of the label, the event sequence is shown in a list box. Figure 5-1 shows the appearance of this window immediately after you click the image in the label. The MouseUp event travels through five levels, ending up at the custom BubbledLabelClick form.



```
#1:
 Sender: System.Windows.Controls.Image
 Source: System.Windows.Controls.Image
 Original Source: System.Windows.Controls.Image
#2:
 Sender: System.Windows.Controls.StackPanel
 Source: System.Windows.Controls.Image
 Original Source: System.Windows.Controls.Image
#3:
 Sender: System.Windows.Controls.Label
 Source: System.Windows.Controls.Image
 Original Source: System.Windows.Controls.Image
#4:
 Sender: System.Windows.Controls.Grid
 Source: System.Windows.Controls.Image
 Original Source: System.Windows.Controls.Image
#5:
 Sender: RoutedEvents.BubbledLabelClick
 Source: System.Windows.Controls.Image
 Original Source: System.Windows.Controls.Image
```

Figure 5-1. A bubbled image click

To create this test form, the image and every element above it in the element hierarchy are wired up to the same event handler—a method named SomethingClicked(). Here's the XAML that does it:

```
<Window x:Class="RoutedEvents.BubbledLabelClick"
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 Title="BubbledLabelClick" Height="359" Width="329"
 MouseUp="SomethingClicked">
  <Grid Margin="3" MouseUp="SomethingClicked">
    <Grid.RowDefinitions>
      <RowDefinition Height="Auto"></RowDefinition>
      <RowDefinition Height="*"></RowDefinition>
      <RowDefinition Height="Auto"></RowDefinition>
      <RowDefinition Height="Auto"></RowDefinition>
    </Grid.RowDefinitions>

    <Label Margin="5" Grid.Row="0" HorizontalAlignment="Left"
     Background="AliceBlue" BorderBrush="Black" BorderThickness="1"
     MouseUp="SomethingClicked">
      <StackPanel MouseUp="SomethingClicked">
        <TextBlock Margin="3"
         MouseUp="SomethingClicked">
         Image and text label</TextBlock>
        <Image Source="happyface.jpg" Stretch="None"
         MouseUp="SomethingClicked" />
        <TextBlock Margin="3"
         MouseUp="SomethingClicked">
         Courtesy of the StackPanel</TextBlock>
      </StackPanel>
    </Label>

    <ListBox Grid.Row="1" Margin="5" Name="lstMessages"></ListBox>
    <CheckBox Grid.Row="2"  Margin="5" Name="chkHandle">
     Handle first event</CheckBox>
    <Button Grid.Row="3" Margin="5" Padding="3" HorizontalAlignment="Right"
     Name="cmdClear" Click="cmdClear_Click">Clear List</Button>
  </Grid>
</Window>
```

The SomethingClicked() method simply examines the properties of the RoutedEventArgs object and adds a message to the list box:

```
protected int eventCounter = 0;

private void SomethingClicked(object sender, RoutedEventArgs e)
{
    eventCounter++;
    string message = "#" + eventCounter.ToString() + ":\r\n" +
      " Sender: " + sender.ToString() + "\r\n" +
      " Source: " + e.Source + "\r\n" +
      " Original Source: " + e.OriginalSource;
    lstMessages.Items.Add(message);
    e.Handled = (bool)chkHandle.IsChecked;
}
```

■ **Note** Technically, the MouseUp event provides a MouseButtonEventArgs object with additional information about the mouse state at the time of the event. However, the MouseButtonEventArgs object derives from MouseEventArgs, which in turn derives from RoutedEventArgs. As a result, it’s possible to use it when declaring the event handler (as shown here) if you don’t need additional information about the mouse.
