CHAPTER 3

Important New Concepts in WPF

IN THIS CHAPTER

- Logical and Visual Trees
- Dependency Properties
- Routed Events
- Commands
- A Tour of the Class Hierarchy

To finish Part I of this book, and before getting to the *really* fun topics, it's helpful to examine some of the main concepts that WPF introduces above and beyond what .NET programmers are already familiar with. The topics in this chapter are some of the main culprits responsible for WPF's notoriously steep learning curve. By familiarizing yourself with these concepts now, you'll be able to approach the rest of this book (or any other WPF documentation) with confidence.

Some of this chapter's concepts are brand new (such as logical and visual trees), but others are just extensions of concepts that should be quite familiar (such as properties and events). As you learn about each one, you'll also see how to apply it to a very simple piece of user interface that most programs need—an *About dialog*.

Logical and Visual Trees

XAML is natural for representing a user interface because of its hierarchical nature. In WPF, user interfaces are constructed from a tree of objects known as a *logical tree*.

Listing 3.1 defines the beginnings of a hypothetical About dialog, using a `Window` as the root of the logical tree. The `Window` has a `StackPanel` child element (described in Chapter 6, "Layout with Panels") containing a few simple controls plus another `StackPanel` which contains `Buttons`.

LISTING 3.1 A Simple About Dialog in XAML

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  Title="About WPF Unleashed" SizeToContent="WidthAndHeight"
  Background="OrangeRed">
  <StackPanel>
    <Label FontWeight="Bold" FontSize="20" Foreground="White">
      WPF Unleashed (Version 3.0)
    </Label>
    <Label>© 2006 SAMS Publishing</Label>
    <Label>Installed Chapters:</Label>
    <ListBox>
      <ListBoxItem>Chapter 1</ListBoxItem>
      <ListBoxItem>Chapter 2</ListBoxItem>
    </ListBox>
    <StackPanel Orientation="Horizontal" HorizontalAlignment="Center">
      <Button MinWidth="75" Margin="10">Help</Button>
      <Button MinWidth="75" Margin="10">OK</Button>
    </StackPanel>
    <StatusBar>You have successfully registered this product.</StatusBar>
  </StackPanel>
</Window>
```

Figure 3.1 shows the rendered dialog (which you can easily produce by pasting the content of Listing 3.1 into a tool such as XamlPad), and Figure 3.2 illustrates the logical tree for this dialog.

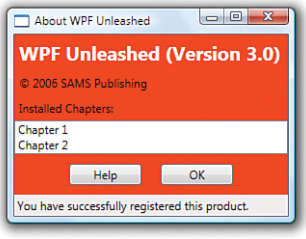

FIGURE 3.1 The rendered dialog from Listing 3.1.

Note that a logical tree exists even for WPF user interfaces that aren't created in XAML. Listing 3.1 could be implemented entirely in procedural code and the logical tree would be identical.

The logical tree concept is straightforward, but why should you care about it? Because just about every aspect of WPF (properties, events, resources, and so on) has behavior tied to the logical tree. For example, property values are sometimes propagated down the tree to child elements automatically, and raised events can travel up or down the tree. Both of these behaviors are discussed later in this chapter.

A similar concept to the logical tree is the *visual tree*. A visual tree is basically an expansion of a logical tree, in which nodes are broken down into their core visual components. Rather than leaving each element as a "black box," a visual tree exposes the visual implementation details. For example, although a `ListBox` is logically a single control, its default visual representation is composed of more primitive WPF elements: a `Border`, two `ScrollBars`, and more.

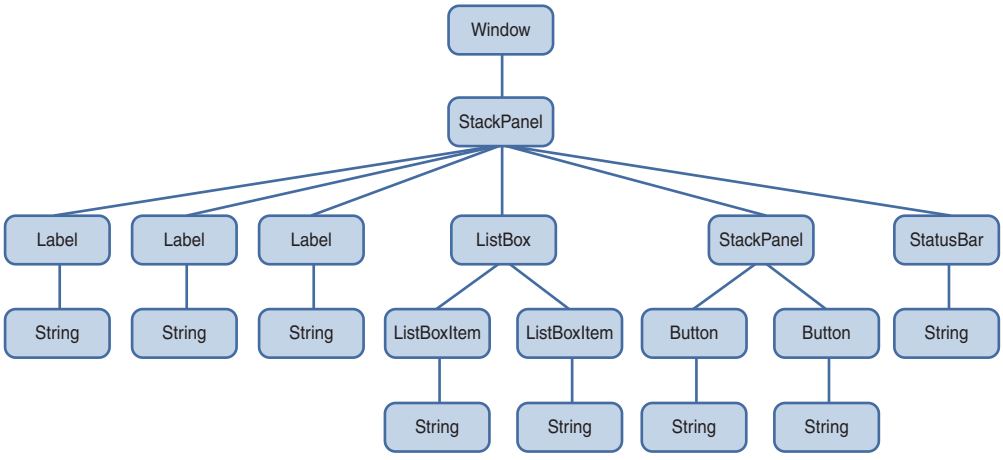

FIGURE 3.2 The logical tree for Listing 3.1.

Not all logical tree nodes appear in the visual tree; only the elements that derive from `System.Windows.Media.Visual` or `System.Windows.Media.Visual3D` are included. Other elements (and simple string content, as in Listing 3.1) are not included because they don't have inherent rendering behavior of their own.

Figure 3.3 illustrates the default visual tree for Listing 3.1 when running on Windows Vista with the Aero theme. This diagram exposes some inner components of the UI that are currently invisible, such as the `ListBox`'s two `ScrollBars` and each `Label`'s `Border`. It also reveals that `Button`, `Label`, and `ListBoxItem` are all comprised of the same elements, except `Button` uses an obscure `ButtonChrome` element rather than a `Border`. (These controls have other visual differences as the result of different default property values. For example, `Button` has a default `Margin` of 10 on all sides whereas `Label` has a default `Margin` of 0.)

TIP

XamlPad contains a button in its toolbar that reveals the visual tree (and property values) for any XAML that it renders. It doesn't work when hosting a `Window` (as in Figure 3.1), but you can change the `Window` element to a `Page` (and remove the `SizeToContent` property) to take advantage of this functionality.

Because they enable you to peer inside the deep composition of WPF elements, visual trees can be surprisingly complex. Fortunately, although visual trees are an essential part of the WPF infrastructure, you often don't need to worry about them unless you're radically restyling controls (covered in Chapter 10, "Styles, Templates, Skins, and Themes") or doing low-level drawing (covered in Chapter 11, "2D Graphics"). Writing code that depends on a specific visual tree for a `Button`, for example, breaks one of WPF's core tenets—the separation of look and logic. When someone restyles a control like `Button` using the techniques described in Chapter 10, its entire visual tree is replaced with something that could be completely different.

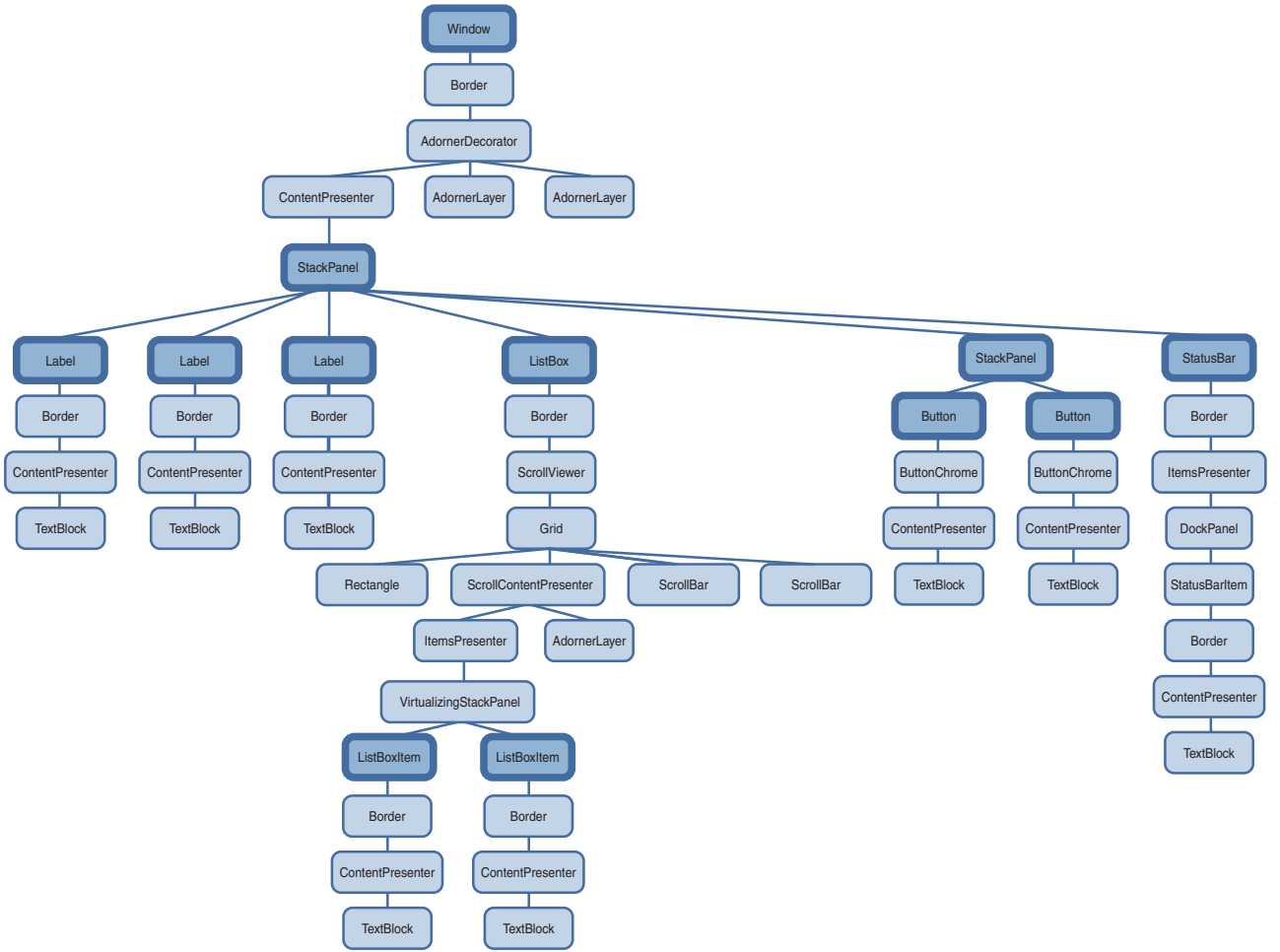

FIGURE 3.3 The visual tree for Listing 3.1, with logical tree nodes emphasized.

That said, you can easily traverse both the logical and visual trees using the somewhat symmetrical `System.Windows.LogicalTreeHelper` and `System.Windows.Media.VisualTreeHelper` classes. Listing 3.2 contains a code-behind file for Listing 3.1 that, when run under a debugger, outputs a simple depth-first representation of both the logical and visual trees for the About dialog. (This requires adding `x:Class="AboutDialog"` and the corresponding `xmlns:x` directive to Listing 3.1 in order to hook it up to this procedural code.)

WARNING

Avoid writing code that depends on a specific visual tree!

Whereas a logical tree is static without programmer intervention (such as dynamically adding/removing elements), a visual tree can change simply by a user switching to a different Windows theme!

LISTING 3.2 Walking and Printing the Logical and Visual Trees

```
using System;
using System.Diagnostics;
using System.Windows;
using System.Windows.Media;

public partial class AboutDialog : Window
{
  public AboutDialog()
  {
    InitializeComponent();
    PrintLogicalTree(0, this);
  }

  protected override void OnContentRendered(EventArgs e)
  {
    base.OnContentRendered(e);
    PrintVisualTree(0, this);
  }

  void PrintLogicalTree(int depth, object obj)
  {
    // Print the object with preceding spaces that represent its depth
    Debug.WriteLine(new string(' ', depth) + obj);

    // Sometimes leaf nodes aren't DependencyObjects (e.g. strings)
    if (!(obj is DependencyObject)) return;

    // Recursive call for each logical child
    foreach (object child in LogicalTreeHelper.GetChildren(
      obj as DependencyObject))
```

LISTING 3.2 Continued

```
      PrintLogicalTree(depth + 1, child);
  }

  void PrintVisualTree(int depth, DependencyObject obj)
  {
    // Print the object with preceding spaces that represent its depth
    Debug.WriteLine(new string(' ', depth) + obj);

    // Recursive call for each visual child
    for (int i = 0; i < VisualTreeHelper.GetChildrenCount(obj); i++)
      PrintVisualTree(depth + 1, VisualTreeHelper.GetChild(obj, i));
  }
}
```

When calling these methods with a depth of 0 and the current `Window` instance, the result is a text-based tree with the exact same nodes shown in Figures 3.2 and 3.3. Although the logical tree can be traversed within the `Window`'s constructor, the visual tree is empty until the `Window` undergoes layout at least once. That is why `PrintVisualTree` is called within `OnContentRendered`, which doesn't get called until after layout occurs.

Navigating either tree can sometimes be done with instance methods on the elements themselves. For example, the `Visual` class contains three protected members (`VisualParent`, `VisualChildrenCount`, and `GetVisualChild`) for examining its visual parent and children. `FrameworkElement`, a common base class for controls such as `Button` and `Label`, defines a public `Parent` property representing the logical parent. Specific subclasses of `FrameworkElement` expose their logical children in different ways. For example, some classes expose a `Children` collection, and other classes (such as `Button` and `Label`) expose a `Content` property, enforcing that the element can only have one logical child.

TIP

Visual trees like the one in represented in Figure 3.3 are often referred to simply as *element trees*, because they encompass both elements in the logical tree and elements specific to the visual tree. The term *visual tree* is then used to describe any subtree that contains visual-only (illogical?) elements. For example, most people would say that `Window`'s default visual tree consists of a `Border`, `AdornerDecorator`, two `AdornerLayers`, a `ContentPresenter`, and nothing more. In Figure 3.3, the top-most `StackPanel` is generally *not* considered to be the visual child of the `ContentPresenter`, despite the fact that `VisualTreeHelper` presents it as one.

Dependency Properties

WPF introduces a new type of property called a *dependency property*, used throughout the platform to enable styling, automatic data binding, animation, and more. You might first meet this concept with skepticism, as it complicates the picture of .NET types having simple fields, properties, methods, and events. But after you understand the problems that dependency properties solve, you will likely accept them as a welcome addition.

A dependency property *depends* on multiple providers for determining its value at any point in time. These providers could be an animation continuously changing its value, a parent element whose property value trickles down to its children, and so on. Arguably the biggest feature of a dependency property is its built-in ability to provide change notification.

The motivation for adding such intelligence to properties is to enable rich functionality directly from declarative markup. The key to WPF's declarative-friendly design is its heavy use of properties. `Button`, for example, has 96 public properties! Properties can be easily set in XAML (directly or by a design tool) without any procedural code. But without the extra plumbing in dependency properties, it would be hard for the simple action of setting properties to get the desired results without writing additional code.

In this section, we'll briefly look at the implementation of a dependency property to make this discussion more concrete, and then we'll dig deeper into some of the ways that dependency properties add value on top of plain .NET properties:

- Change notification
- Property value inheritance
- Support for multiple providers

Understanding most of the nuances of dependency properties is usually only important for custom control authors. However, even casual users of WPF end up needing to be aware of what they are and how they work. For example, you can only style and animate dependency properties. After working with WPF for a while you might find yourself wishing that all properties would be dependency properties!

A Dependency Property Implementation

In practice, dependency properties are just normal .NET properties hooked into some extra WPF infrastructure. This is all accomplished via WPF APIs; no .NET languages (other than XAML) have an intrinsic understanding of a dependency property.

Listing 3.3 demonstrates how `Button` effectively implements one of its dependency properties called `IsDefault`.

LISTING 3.3 A Standard Dependency Property Implementation

```
public class Button : ButtonBase
{
  // The dependency property
  public static readonly DependencyProperty IsDefaultProperty;

  static Button()
  {
    // Register the property
    Button.IsDefaultProperty = DependencyProperty.Register("IsDefault",
      typeof(bool), typeof(Button),
      new FrameworkPropertyMetadata(false,
      new PropertyChangedCallback(OnIsDefaultChanged)));
    …
  }

  // A .NET property wrapper (optional)
  public bool IsDefault
  {
    get { return (bool)GetValue(Button.IsDefaultProperty); }
    set { SetValue(Button.IsDefaultProperty, value); }
  }

  // A property changed callback (optional)
  private static void OnIsDefaultChanged(
    DependencyObject o, DependencyPropertyChangedEventArgs e) { … }
  …
}
```

The static `IsDefaultProperty` field is the actual dependency property, represented by the `System.Windows.DependencyProperty` class. By convention all `DependencyProperty` fields are public, static, and have a `Property` suffix. Dependency properties are usually created by calling the static `DependencyProperty.Register` method, which requires a name (`IsDefault`), a property type (`bool`), and the type of the class claiming to own the property (`Button`). Optionally (via different overloads of `Register`), you can pass metadata that customizes how the property is treated by WPF, as well as callbacks for handling property value changes, coercing values, and validating values. `Button` calls an overload of `Register` in its static constructor to give the dependency property a default value of `false` and to attach a delegate for change notifications.

Finally, the traditional .NET property called `IsDefault` implements its accessors by calling `GetValue` and `SetValue` methods inherited from `System.Windows.DependencyObject`, a low-level base class from which all classes with dependency properties must derive. `GetValue` returns the last value passed to `SetValue` or, if `SetValue` has never been called, the default value registered with the property. The `IsDefault` .NET property (sometimes

called a *property wrapper* in this context) is not strictly necessary; consumers of `Button` could always directly call the `GetValue/SetValue` methods because they are exposed publicly. But the .NET property makes programmatic reading and writing of the property much more natural for consumers, and it enables the property to be set via XAML.

WARNING

.NET property wrappers are bypassed at run-time when setting dependency properties in XAML!

Although the XAML compiler depends on the property wrapper at compile-time, at run-time WPF calls the underlying `GetValue` and `SetValue` methods directly! Therefore, to maintain parity between setting a property in XAML and procedural code, it's crucial that property wrappers do not contain any logic in addition to the `GetValue/SetValue` calls. If you want to add custom logic, that's what the registered callbacks are for. All of WPF's built-in property wrappers abide by this rule, so this warning is for anyone writing a custom class with its own dependency properties.

On the surface, Listing 3.3 looks like an overly verbose way of representing a simple Boolean property. However, because `GetValue` and `SetValue` internally use an efficient sparse storage system and because `IsDefaultProperty` is a static field (rather than an instance field), the dependency property implementation saves per-instance memory compared to a typical .NET property. If all the properties on WPF controls were wrappers around instance fields (as most .NET properties are), they would consume a significant amount of memory because of all the local data attached to each instance. Having 96 fields for each `Button`, 89 fields for each `Label`, and so forth would add up quickly! Instead, 78 out of `Button`'s 96 properties are dependency properties, and 71 out of `Label`'s 89 properties are dependency properties.

The benefits of the dependency property implementation extend to more than just memory usage, however. It centralizes and standardizes a fair amount of code that property implementers would have to write to check thread access, prompt the containing element to be re-rendered, and so on. For example, if a property requires its element to be re-rendered when its value changes (such as `Button`'s `Background` property), it can simply pass the `FrameworkPropertyMetadataOptions.AffectsRender` flag to an overload of `DependencyProperty.Register`. In addition, this implementation enables the three features listed earlier that we'll now examine one-by-one, starting with change notification.

Change Notification

Whenever the value of a dependency property changes, WPF can automatically trigger a number of actions depending on the property's metadata. These actions can be re-rendering the appropriate elements, updating the current layout, refreshing data bindings, and much more. One of the most interesting features enabled by this built-in change notification is *property triggers*, which enable you to perform your own custom actions when a property value changes without writing any procedural code.

For example, imagine that you want the text in each `Button` from the About dialog in Listing 3.1 to turn blue when the mouse pointer hovers over it. Without property triggers, you can attach two event handlers to each `Button`, one for its `MouseEnter` event and one for its `MouseLeave` event:

```
<Button MouseEnter="Button_MouseEnter" MouseLeave="Button_MouseLeave"
        MinWidth="75" Margin="10">Help</Button>
<Button MouseEnter="Button_MouseEnter" MouseLeave="Button_MouseLeave"
        MinWidth="75" Margin="10">OK</Button>
```

These two handlers could be implemented in a C# code-behind file as follows:

```
// Change the foreground to blue when the mouse enters the button
void Button_MouseEnter(object sender, MouseEventArgs e)
{
  Button b = sender as Button;
  if (b != null) b.Foreground = Brushes.Blue;
}

// Restore the foreground to black when the mouse exits the button
void Button_MouseLeave(object sender, MouseEventArgs e)
{
  Button b = sender as Button;
  if (b != null) b.Foreground = Brushes.Black;
}
```

With a property trigger, however, you can accomplish this same behavior purely in XAML. The following concise `Trigger` object is (just about) all you need:

```
<Trigger Property="IsMouseOver" Value="True">
  <Setter Property="Foreground" Value="Blue"/>
</Trigger>
```

This trigger can act upon `Button`'s `IsMouseOver` property, which becomes `true` at the same time the `MouseEnter` event is raised and `false` at the same time the `MouseLeave` event is raised. Note that you don't have to worry about reverting `Foreground` to black when `IsMouseOver` changes to `false`. This is automatically done by WPF!

The only trick is assigning this `Trigger` to each `Button`. Unfortunately, because of an artificial limitation in WPF version 3.0, you can't apply property triggers directly to elements such as `Button`. They can only be applied inside a `Style` object, so an in-depth examination of property triggers is saved for Chapter 10. In the meantime, if you want to experiment with property triggers, you could apply the preceding `Trigger` to a `Button` by wrapping it in a few intermediate XML elements as follows:

```
<Button MinWidth="75" Margin="10">
<Button.Style>
  <Style TargetType="{x:Type Button}">
```

```
  <Style.Triggers>
    <Trigger Property="IsMouseOver" Value="True">
      <Setter Property="Foreground" Value="Blue"/>
    </Trigger>
  </Style.Triggers>
  </Style>
</Button.Style>
  OK
</Button>
```

Property triggers are just one of three types of triggers supported by WPF. A *data trigger* is a form of property trigger that works for all .NET properties (not just dependency properties), also covered in Chapter 10. An *event trigger* enables you to declaratively specify actions to take when a routed event (covered later in the chapter) is raised. Event triggers always involve working with animations or sounds, so they aren't covered until Chapter 13, "Animation."

WARNING

Don't be fooled by an element's `Triggers` collection!

`FrameworkElement`'s `Triggers` property is a read/write collection of `TriggerBase` items (the common base class for all three types of triggers), so it looks like an easy way to attach property triggers to controls such as `Button`. Unfortunately, this collection can only contain event triggers in version 3.0 of WPF simply because the WPF team didn't have time to implement this support. Attempting to add a property trigger (or data trigger) to the collection causes an exception to be thrown at run-time.

Property Value Inheritance

The term *property value inheritance* (or *property inheritance* for short) doesn't refer to traditional object oriented class-based inheritance, but rather the flowing of property values down the element tree. A simple example of this can be seen in Listing 3.4, which updates the `Window` from Listing 3.1 by explicitly setting its `FontSize` and `FontStyle` dependency properties. Figure 3.4 shows the result of this change. (Notice that the `Window` automatically resizes to fit all the content thanks to its slick `SizeToContent` setting!)

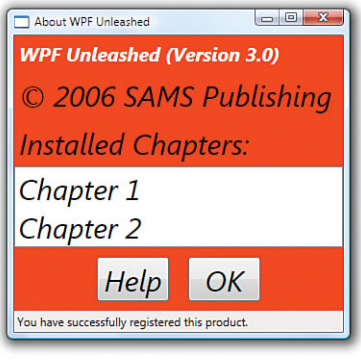

FIGURE 3.4 The About dialog with `FontSize` and `FontStyle` set on the root `Window`.

LISTING 3.4 The About Dialog with Font Properties Set on the Root `Window`

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  Title="About WPF Unleashed" SizeToContent="WidthAndHeight"
  FontSize="30" FontStyle="Italic"
  Background="OrangeRed">
  <StackPanel>
    <Label FontWeight="Bold" FontSize="20" Foreground="White">
      WPF Unleashed (Version 3.0)
    </Label>
    <Label>© 2006 SAMS Publishing</Label>
    <Label>Installed Chapters:</Label>
    <ListBox>
      <ListBoxItem>Chapter 1</ListBoxItem>
      <ListBoxItem>Chapter 2</ListBoxItem>
    </ListBox>
    <StackPanel Orientation="Horizontal" HorizontalAlignment="Center">
      <Button MinWidth="75" Margin="10">Help</Button>
      <Button MinWidth="75" Margin="10">OK</Button>
    </StackPanel>
    <StatusBar>You have successfully registered this product.</StatusBar>
  </StackPanel>
</Window>
```

For the most part, these two settings flow all the way down the tree and are inherited by children. This affects even the `Buttons` and `ListBoxItems`, which are three levels down the logical tree. The first `Label`'s `FontSize` does not change because it is explicitly marked with a `FontSize` of `20`, overriding the inherited value of `30`. The inherited `FontStyle` setting of `Italic` affects all `Labels`, `ListBoxItems`, and `Buttons`, however, because none of them have this set explicitly.

Notice that the text in the `StatusBar` is unaffected by either of these values, despite the fact that it supports these two properties just like the other controls. The behavior of property value inheritance can be subtle in cases like this for two reasons:

- Not every dependency property participates in property value inheritance. (Internally, dependency properties can opt in to inheritance by passing `FrameworkPropertyMetadataOptions.Inherits` to `DependencyProperty.Register`.)
- There may be other higher-priority sources setting the property value, as explained in the next section.

In this case, the latter reason is to blame. A few controls such as `StatusBar`, `Menu`, and `ToolTip` internally set their font properties to match current system settings. This way, users get the familiar experience of controlling their font via Control Panel. The result can be confusing, however, because such controls end up "swallowing" any inheritance from proceeding further down the element tree. For example, if you add a `Button` as a logical

child of the `StatusBar` in Listing 3.4, its `FontSize` and `FontStyle` would be the default values of `12` and `Normal`, respectively, unlike the other `Buttons` outside of the `StatusBar`.

DIGGING DEEPER

Property Value Inheritance in Additional Places

Property value inheritance was originally designed to operate on the element tree, but it has been extended to work in a few other contexts as well. For example, values can be passed down to certain elements that *look like* children in the XML sense (because of XAML's property element syntax) but *are not* children in terms of the logical or visual trees. These pseudochildren can be an element's triggers or the value of *any* property (not just `Content` or `Children`) as long as it is an object deriving from `Freezable`. This may sound arbitrary and isn't well documented, but the intention is that several XAML-based scenarios "just work" as you would expect without having to think about it.

Support for Multiple Providers

WPF contains many powerful mechanisms that independently attempt to set the value of dependency properties. Without a well-defined mechanism for handling these disparate property value providers, the system would be a bit chaotic and property values could be unstable. Of course, as their name indicates, dependency properties were designed to depend on these providers in a consistent and orderly manner.

Figure 3.5 illustrates the five-step process that WPF runs each dependency property through in order to calculate its final value. This process happens automatically thanks to the built-in change notification in dependency properties.

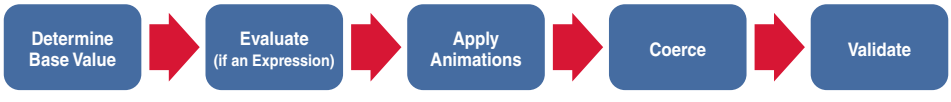

FIGURE 3.5 The pipeline for calculating the value of a dependency property.

Step 1: Determine Base Value

Most of the property value providers factor into the base value calculation. The following list reveals the eight providers that can set the value of most dependency properties, in order from highest to lowest precedence:

1. Local value
2. Style triggers
3. Template triggers
4. Style setters
5. Theme style triggers
6. Theme style setters
7. Property value inheritance
8. Default value

You've already seen some of the property value providers, such as property value inheritance. *Local value* technically means any call to `DependencyObject.SetValue`, but this is typically seen with a simple property assignment in XAML or procedural code (because of the way dependency properties are implemented, as shown previously with `Button.IsDefault`). *Default value* refers to the initial value registered with the dependency property, which naturally has the lowest precedence. The other providers, which all involve styles and templates, are explained further in Chapter 10.

This order of precedence explains why `StatusBar`'s `FontSize` and `FontStyle` were not impacted by property value inheritance in Listing 3.4. The setting of `StatusBar`'s font properties to match system settings is done via theme style setters (#6 in the list). Although this has precedence over property value inheritance (#7 in the list), you can still override these font settings using any mechanism with a higher precedence, such as simply setting local values on the `StatusBar`.

Step 2: Evaluate

If the value from step one is an *expression* (an object deriving from `System.Windows.Expression`), then WPF performs a special evaluation step to convert the expression into a concrete result. In version 3.0 of WPF, expressions only come into play when using dynamic resources (covered in Chapter 8, "Resources") or data binding (the topic of Chapter 9, "Data Binding"). Future versions of WPF may enable additional kinds of expressions.

Step 3: Apply Animations

If one or more animations are running, they have the power to alter the current property value (using the value after step 2 as input) or completely replace it. Therefore, animations (the topic of Chapter 13) can trump all other property value providers—even local values! This is often a stumbling block for people who are new to WPF.

Step 4: Coerce

After all the property value providers have had their say, WPF takes the almost-final property value and passes it to a `CoerceValueCallback` delegate, if one was registered with the dependency property. The callback is responsible for returning a new value, based on custom logic. For example, built-in WPF controls such as `ProgressBar` use this callback to constrain its `Value` dependency property to a value between its `Minimum` and `Maximum` values, returning `Minimum` if the input value is less than `Minimum` or `Maximum` if the input value is greater than `Maximum`.

Step 5: Validate

Finally, the potentially-coerced value is passed to a `ValidateValueCallback` delegate, if one was registered with the dependency property. This callback must return `true` if the input value is valid or `false` otherwise. Returning `false` causes an exception to be thrown, cancelling the entire process.

TIP

If you can't figure out where a given dependency property is getting its current value from, you can use the static `DependencyPropertyHelper.GetValueSource` method as a debugging aid. This returns a `ValueSource` structure that contains a few pieces of data: a `BaseValueSource` enumeration that reveals where the base value came from (step 1 in the process) and Boolean `IsExpression`, `IsAnimated`, and `IsCoerced` properties that reveal information about steps 2-4.

When calling this method on the `StatusBar` instance from Listing 3.1 or 3.4 with the `FontSize` or `FontStyle` property, the returned `BaseValueSource` is `DefaultStyle`, revealing that the value comes from a theme style setter. (Theme styles are sometimes referred to as *default styles*. The enumeration value for a theme style trigger is `DefaultStyleTrigger`.)

Do *not* use this method in production code! Future versions of WPF could break assumptions you've made about the value calculation, plus treating a property value differently depending on its source goes against the way things are supposed to work in WPF applications.

DIGGING DEEPER

Clearing a Local Value

The earlier "Change Notification" section demonstrated the use of procedural code to change a `Button`'s `Foreground` to blue in response to the `MouseEnter` event, and then changing it back to black in response to the `MouseLeave` event. The problem with this approach is that black is set as a local value inside `MouseLeave`, which is much different from the `Button`'s initial state in which its black `Foreground` comes from a setter in its theme style. If the theme is changed and the new theme tries to change the default `Foreground` color (or if other providers with higher precedence try to do the same), it gets trumped by the local setting of black.

What you likely want to do instead is *clear* the local value and let WPF set the value from the relevant provider with the next-highest precedence. Fortunately, `DependencyObject` provides exactly this kind of mechanism with its `ClearValue` method. This can be called on a `Button` `b` as follows in C#:

```
b.ClearValue(Button.ForegroundProperty);
```

(`Button.ForegroundProperty` is the static `DependencyProperty` field.) After calling `ClearValue`, the local value is simply removed from the equation when WPF recalculates the base value.

Note that the trigger on the `IsMouseOver` property from the "Change Notification" section does not have the same problem as the implementation with event handlers. A trigger is either active or inactive, and when inactive it is simply ignored in the property value calculation.

Attached Properties

An attached property is a special form of dependency property that can effectively be *attached* to arbitrary objects. This may sound strange at first, but this mechanism has several applications in WPF.

For the About dialog example, imagine that rather than setting `FontSize` and `FontStyle` for the entire `Window` (as done in Listing 3.4), you would rather set them on the inner `StackPanel` so they are inherited only by the two `Button`s. Moving the property attributes to the inner `StackPanel` element doesn't work, however, because `StackPanel` doesn't have any font-related properties of its own! Instead, you must use the `FontSize` and `FontStyle` attached properties that happen to be defined on a class called `TextElement`. Listing 3.5 demonstrates this, introducing new XAML syntax designed especially for attached properties. This enables the desired property value inheritance, as shown in Figure 3.6.

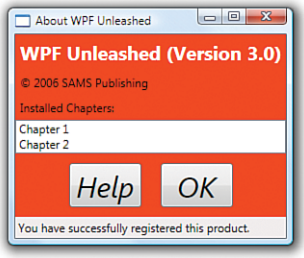

FIGURE 3.6 The About dialog with `FontSize` and `FontStyle` set on both `Button`s via inheritance from the inner `StackPanel`.

LISTING 3.5 The About Dialog with Font Properties Moved to the Inner `StackPanel`

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  Title="About WPF Unleashed" SizeToContent="WidthAndHeight"
  Background="OrangeRed">
  <StackPanel>
    <Label FontWeight="Bold" FontSize="20" Foreground="White">
      WPF Unleashed (Version 3.0)
    </Label>
    <Label>© 2006 SAMS Publishing</Label>
    <Label>Installed Chapters:</Label>
    <ListBox>
      <ListBoxItem>Chapter 1</ListBoxItem>
      <ListBoxItem>Chapter 2</ListBoxItem>
    </ListBox>
    <StackPanel TextElement.FontSize="30" TextElement.FontStyle="Italic"
      Orientation="Horizontal" HorizontalAlignment="Center">
      <Button MinWidth="75" Margin="10">Help</Button>
      <Button MinWidth="75" Margin="10">OK</Button>
    </StackPanel>
    <StatusBar>You have successfully registered this product.</StatusBar>
  </StackPanel>
</Window>
```

`TextElement.FontSize` and `TextElement.FontStyle` (rather than simply `FontSize` and `FontStyle`) must be used in the `StackPanel` element because `StackPanel` does not have these properties. When a XAML parser or compiler encounters this syntax, it requires that `TextElement` (sometimes called the *attached property provider*) has static methods called `SetFontSize` and `SetFontStyle` that can set the value accordingly. Therefore, the `StackPanel` declaration in Listing 3.5 is equivalent to the following C# code:

```
StackPanel panel = new StackPanel();
TextElement.SetFontSize(panel, 30);
TextElement.SetFontStyle(panel, FontStyles.Italic);
panel.Orientation = Orientation.Horizontal;
panel.HorizontalAlignment = HorizontalAlignment.Center;
Button helpButton = new Button();
helpButton.MinWidth = 75;
helpButton.Margin = new Thickness(10);
helpButton.Content = "Help";
Button okButton = new Button();
okButton.MinWidth = 75;
okButton.Margin = new Thickness(10);
okButton.Content = "OK";
panel.Children.Add(helpButton);
panel.Children.Add(okButton);
```

Notice that the enumeration values such as `FontStyles.Italic`, `Orientation.Horizontal`, and `HorizontalAlignment.Center` were previously specified in XAML simply as `Italic`, `Horizontal`, and `Center`, respectively. This is possible thanks to the `EnumConverter` type converter in the .NET Framework, which can convert any case-insensitive string.

Although the XAML in Listing 3.5 nicely represents the logical attachment of `FontSize` and `FontStyle` to `StackPanel`, the C# code reveals that there's no real magic here; just a method call that associates an element with an otherwise-unrelated property. One of the interesting things about the attached property abstraction is that no .NET property is a part of it!

Internally, methods like `SetFontSize` simply call the same `DependencyObject.SetValue` method that a normal dependency property accessor calls, but on the passed-in `DependencyObject` rather than the current instance:

```
public static void SetFontSize(DependencyObject element, double value)
{
  element.SetValue(TextElement.FontSizeProperty, value);
}
```

Similarly, attached properties also define a static `Get`*`XXX`* method (where *XXX* is the name of the property) that calls the familiar `DependencyObject.GetValue` method:

```
public static double GetFontSize(DependencyObject element)
{
  return (double)element.GetValue(TextElement.FontSizeProperty);
}
```

As with property wrappers for normal dependency properties, these `Get`*`XXX`* and `Set`*`XXX`* methods must not do anything other than making a call to `GetValue` and `SetValue`.

DIGGING DEEPER

Understanding the Attached Property Provider

The most confusing part about the `FontSize` and `FontStyle` attached properties used in Listing 3.5 is that they aren't defined by `Button` or even `Control`, the base class that defines the normal `FontSize` and `FontStyle` dependency properties! Instead, they are defined by the seemingly unrelated `TextElement` class (and also by the `TextBlock` class, which could also be used in the preceding examples).

How can this possibly work when `TextElement.FontSizeProperty` is a separate `DependencyProperty` field from `Control.FontSizeProperty` (and `TextElement.FontStyleProperty` is separate from `Control.FontStyleProperty`)? The key is the way these dependency properties are internally registered. If you were to look at the source code for `TextElement`, you would see something like the following:

```
TextElement.FontSizeProperty = DependencyProperty.RegisterAttached(
  "FontSize", typeof(double), typeof(TextElement), new FrameworkPropertyMetadata(
  SystemFonts.MessageFontSize, FrameworkPropertyMetadataOptions.Inherits |
  FrameworkPropertyMetadataOptions.AffectsRender |
  FrameworkPropertyMetadataOptions.AffectsMeasure),
  new ValidateValueCallback(TextElement.IsValidFontSize));
```

This is similar to the earlier example of registering `Button`'s `IsDefault` dependency property, except that the `RegisterAttached` method optimizes the handling of property metadata for attached property scenarios.

`Control`, on the other hand, doesn't register its `FontSize` dependency property! Instead, it calls `AddOwner` on `TextElement`'s already-registered property, getting a reference to the exact same instance:

```
Control.FontSizeProperty = TextElement.FontSizeProperty.AddOwner(
  typeof(Control), new FrameworkPropertyMetadata(SystemFonts.MessageFontSize,
  FrameworkPropertyMetadataOptions.Inherits));
```

Therefore, the `FontSize`, `FontStyle`, and other font-related dependency properties inherited by all controls *are* the same properties exposed by `TextElement`!

Fortunately, in most cases, the class that exposes an attached property (e.g. `Get`*`XXX`* and `Set`*`XXX`* methods) is the same class that defines the normal dependency property, avoiding this confusion.

Although the About dialog example uses attached properties for advanced property value inheritance, attached properties are most commonly used for layout of user interface elements. (In fact, attached properties were originally designed for WPF's layout system.) Various `Panel`-derived classes define attached properties designed to be attached to their children for controlling how they are arranged. This way, each `Panel` can apply its own custom behavior to arbitrary children without requiring all possible child elements to be burdened with their own set of relevant properties. It also enables systems like layout to be easily extensible, because anyone can write a new `Panel` with custom attached properties. Chapter 6, "Layout with Panels," and Chapter 17, "Layout with Custom Panels," have all the details.

DIGGING DEEPER

Attached Properties as an Extensibility Mechanism

Just like previous technologies such as Windows Forms, many classes in WPF define a `Tag` property (of type `System.Object`) intended for storing arbitrary custom data with each instance. But attached properties are a more powerful and flexible mechanism for attaching custom data to any object deriving from `DependencyObject`. It's often overlooked that attached properties enable you to effectively add custom data to instances of sealed classes (something that WPF has plenty of)!

A further twist to the story of attached properties is that although setting them in XAML relies on the presence of the static `Set`*`XXX`* method, you can bypass this method in procedural code and call `DependencyObject.SetValue` directly. This means that you can use *any* dependency property as an attached property in procedural code. For example, the following line of code attaches `ListBox`'s `IsTextSearchEnabled` property to a `Button` and assigns it a value:

```
// Attach an unrelated property to a Button and set its value to true:
okButton.SetValue(ListBox.IsTextSearchEnabledProperty, true);
```

Although this seems nonsensical, and it certainly doesn't magically enable new functionality on this `Button`, you have the freedom to consume this property value in a way that makes sense to your application or component.

There are more interesting ways to extend elements in this manner. For example, `FrameworkElement`'s `Tag` property is a dependency property, so you can attach it to an instance of a `GeometryModel3D` (a class you'll see again in Chapter 12, "3D Graphics," that is sealed and does *not* have a `Tag` property) as follows:

```
GeometryModel3D model = new GeometryModel3D();
model.SetValue(FrameworkElement.TagProperty, "my custom data");
```

This is just one of the ways in which WPF provides extensibility without the need for traditional inheritance.

Routed Events

Just as WPF adds more infrastructure on top of the simple notion of .NET properties, it also adds more infrastructure on top of the simple notion of .NET events. *Routed events* are events that are designed to work well with a tree of elements. When a routed event is raised, it can travel up or down the visual and logical tree, getting raised on each element in a simple and consistent fashion, without the need for any custom code.

Event routing helps most applications remain oblivious to details of the visual tree (which is good for restyling) and is crucial to the success of WPF's element composition. For example, `Button` exposes a `Click` event based on handling lower-level `MouseLeftButtonDown` and `KeyDown` events. When a user presses the left mouse button with the mouse pointer over a standard `Button`, however, they are really interacting with its `ButtonChrome` or `TextBlock` visual child. Because the event travels up the *visual* tree, the `Button` eventually sees the event and can handle it. Similarly, for the VCR-style Stop `Button` in the preceding chapter, a user might press the left mouse button directly over the `Rectangle` logical child. Because the event travels up the *logical* tree, the `Button` still sees the event and can handle it as well. (Yet if you really wish to distinguish between an event on the `Rectangle` versus the outer `Button`, you have the freedom to do so.)

Therefore, you can embed arbitrarily complex content inside an element like `Button` or give it an arbitrarily complex visual tree (using the techniques in Chapter 10), and a mouse left-click on any of the internal elements still results in a `Click` event raised by the parent `Button`. Without routed events, producers of the inner content or consumers of the `Button` would have to write code to patch everything together.

The implementation and behavior of routed events have many parallels to dependency properties. As with the dependency property discussion, we'll first look at how a simple routed event is implemented to make things more concrete. Then we'll examine some of the features of routed events and apply them to the About dialog.

A Routed Event Implementation

In most cases, routed events don't look very different from normal .NET events. As with dependency properties, no .NET languages (other than XAML) have an intrinsic understanding of the *routed* designation. The extra support is based on a handful of WPF APIs.

Listing 3.6 demonstrates how `Button` effectively implements its `Click` routed event. (`Click` is actually implemented by `Button`'s base class, but that's not important for this discussion.)

Just as dependency properties are represented as public static `DependencyProperty` fields with a conventional `Property` suffix, routed events are represented as public static `RoutedEvent` fields with a conventional `Event` suffix. The routed event is registered much like a dependency property in the static constructor, and a normal .NET event—or *event wrapper*—is defined to enable more familiar use from procedural code and adding a handler in XAML with event attribute syntax. As with a property wrapper, an event wrapper must not do anything in its accessors other than call `AddHandler` and `RemoveHandler`.

LISTING 3.6 A Standard Routed Event Implementation

```
public class Button : ButtonBase
{
  // The routed event
  public static readonly RoutedEvent ClickEvent;

  static Button()
  {
    // Register the event
    Button.ClickEvent = EventManager.RegisterRoutedEvent("Click",
      RoutingStrategy.Bubble, typeof(RoutedEventHandler), typeof(Button));
    …
  }

  // A .NET event wrapper (optional)
  public event RoutedEventHandler Click
  {
    add { AddHandler(Button.ClickEvent, value); }
    remove { RemoveHandler(Button.ClickEvent, value); }
  }

  protected override void OnMouseLeftButtonDown(MouseButtonEventArgs e)
  {
    …
    // Raise the event
    RaiseEvent(new RoutedEventArgs(Button.ClickEvent, this));
    …
  }
  …
}
```

These `AddHandler` and `RemoveHandler` methods are not inherited from `DependencyObject`, but rather `System.Windows.UIElement`, a higher-level base class of elements such as `Button`. (This class hierarchy is examined in more depth at the end of this chapter.) These methods attach and remove a delegate to the appropriate routed event. Inside `OnMouseLeftButtonDown`, `RaiseEvent` (also defined on the base `UIElement` class) is called with the appropriate `RoutedEvent` field to raise the `Click` event. The current `Button` instance (`this`) is passed as the source element of the event. It's not shown in this listing, but `Button`'s `Click` event is also raised in response to a `KeyDown` event to support clicking with the spacebar or sometimes the Enter key.

Routing Strategies and Event Handlers

When registered, every routed event chooses one of three *routing strategies*—the way in which the event raising travels through the element tree. These strategies are exposed as values of a `RoutingStrategy` enumeration:

- **`Tunneling`**—The event is first raised on the root, then on each element down the tree until the source element is reached (or until a handler halts the tunneling by marking the event as handled).
- **`Bubbling`**—The event is first raised on the source element, then on each element up the tree until the root is reached (or until a handler halts the bubbling by marking the event as handled).
- **`Direct`**—The event is only raised on the source element. This is the same behavior as a plain .NET event, except that such events can still participate in mechanisms specific to routed events such as event triggers.

Handlers for routed events have a signature matching the pattern for general .NET event handlers: The first parameter is a `System.Object` typically named `sender`, and the second parameter (typically named `e`) is a class that derives from `System.EventArgs`. The `sender` parameter passed to a handler is always the element to which the handler was attached. The `e` parameter is (or derives from) an instance of `RoutedEventArgs`, a subclass of `EventArgs` that exposes four useful properties:

- **`Source`**—The element in the logical tree that originally raised the event.
- **`OriginalSource`**—The element in the visual tree that originally raised the event (for example, the `TextBlock` or `ButtonChrome` child of a standard `Button`).
- **`Handled`**—A Boolean that can be set to `true` to mark the event as handled. This is precisely what halts any tunneling or bubbling.
- **`RoutedEvent`**—The actual routed event object (such as `Button.ClickEvent`), which can be helpful for identifying the raised event when the same handler is used for multiple routed events.

The presence of both `Source` and `OriginalSource` enable you to work with the higher-level logical tree or the lower-level visual tree. This distinction only applies to physical events like mouse events, however. For more abstract events that don't necessarily have a direct relationship with an element in the visual tree (like `Click` due to its keyboard support), the same object is passed for both `Source` and `OriginalSource`.

Routed Events in Action

The `UIElement` class defines many routed events for keyboard, mouse, and stylus input. Most of these are bubbling events, but many of them are paired with a tunneling event. Tunneling events can be easily identified because, by convention, they are named with a `Preview` prefix. These events, also by convention, are raised immediately before their

bubbling counterpart. For example, `PreviewMouseMove` is a tunneling event raised before the `MouseMove` bubbling event.

The idea behind having a pair of events for various activities is to give elements a chance to effectively cancel or otherwise modify an event that's about to occur. By convention, WPF's built-in elements only take action in response to a bubbling event (when a bubbling and tunneling pair is defined), ensuring that the tunneling event lives up to its "preview" name. For example, imagine you want to implement a `TextBox` that restricts its input to a certain pattern or regular expression (such as a phone number or zip code). If you handle `TextBox`'s `KeyDown` event, the best you can do is remove text that has already been displayed inside the `TextBox`. But if you handle `TextBox`'s `PreviewKeyDown` event instead, you can mark it as "handled" to not only stop the tunneling but also stop the bubbling `KeyDown` event from being raised. In this case, the `TextBox` will never receive the `KeyDown` notification and the current character will not get displayed.

DIGGING DEEPER

Using Stylus Events

A *stylus*—the pen-like device used by Tablet PCs—acts like a mouse by default. In other words, its use raises events such as `MouseMove`, `MouseDown`, and `MouseUp`. This behavior is essential for a stylus to be usable with programs that aren't designed specifically for a Tablet PC. However, if you want to provide an experience that is *optimized* for a stylus, you can handle stylus-specific events such as `StylusMove`, `StylusDown`, and `StylusUp`. A stylus can do more "tricks" than a mouse, as evidenced by some of its events that have no mouse counterpart, such as `StylusInAirMove`, `StylusSystemGesture`, `StylusInRange`, and `StylusOutOfRange`. There are other ways to exploit a stylus without handling these events directly, however. The next chapter, "Introducing WPF's Controls," shows how this can be done with a powerful `InkCanvas` element.

To demonstrate the use of a simple bubbling event, Listing 3.7 updates the original About dialog from Listing 3.1 by attaching an event handler to `Window`'s `MouseRightButtonDown` event. Listing 3.8 contains the C# code-behind file with the event handler implementation.

LISTING 3.7 The About Dialog with an Event Handler on the Root `Window`

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  x:Class="AboutDialog" MouseRightButtonDown="AboutDialog_MouseRightButtonDown"
  Title="About WPF Unleashed" SizeToContent="WidthAndHeight"
  Background="OrangeRed">
  <StackPanel>
    <Label FontWeight="Bold" FontSize="20" Foreground="White">
      WPF Unleashed (Version 3.0)
    </Label>
    <Label>© 2006 SAMS Publishing</Label>
    <Label>Installed Chapters:</Label>
    <ListBox>
```

LISTING 3.7 Continued

```
      <ListBoxItem>Chapter 1</ListBoxItem>
      <ListBoxItem>Chapter 2</ListBoxItem>
    </ListBox>
    <StackPanel Orientation="Horizontal" HorizontalAlignment="Center">
      <Button MinWidth="75" Margin="10">Help</Button>
      <Button MinWidth="75" Margin="10">OK</Button>
    </StackPanel>
    <StatusBar>You have successfully registered this product.</StatusBar>
  </StackPanel>
</Window>
```

LISTING 3.8 The Code-Behind File for Listing 3.7

```
using System.Windows;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Controls;

public partial class AboutDialog : Window
{
  public AboutDialog()
  {
    InitializeComponent();
  }

  void AboutDialog_MouseRightButtonDown(object sender, MouseButtonEventArgs e)
  {
    // Display information about this event
    this.Title = "Source = " + e.Source.GetType().Name + ", OriginalSource = " +
      e.OriginalSource.GetType().Name + " @ " + e.Timestamp;

    // In this example, all possible sources derive from Control
    Control source = e.Source as Control;

    // Toggle the border on the source control
    if (source.BorderThickness != new Thickness(5))
    {
      source.BorderThickness = new Thickness(5);
      source.BorderBrush = Brushes.Black;
    }
    else
      source.BorderThickness = new Thickness(0);
  }
}
```

The `AboutDialog_MouseRightButtonDown` handler performs two actions whenever a right-click bubbles up to the `Window`: It prints information about the event to the `Window`'s title bar, and it adds (then subsequently removes) a thick black border around the specific element in the logical tree that was right-clicked. Figure 3.7 shows the result. Notice that right-clicking on the `Label` reveals `Source` set to the `Label` but `OriginalSource` set to its `TextBlock` visual child.

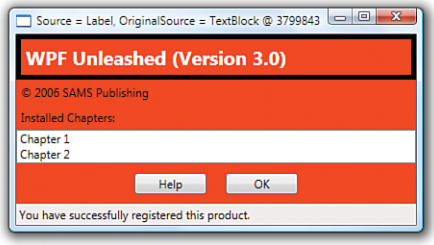

FIGURE 3.7 The modified About dialog, after the first `Label` is right-clicked.

If you run this example and right-click on everything, you'll notice two interesting behaviors:

- `Window` never receives the `MouseRightButtonDown` event when you right-click on either `ListBoxItem`. That's because `ListBoxItem` internally handles this event as well as the `MouseLeftButtonDown` event (halting the bubbling) to implement item selection.
- `Window` receives the `MouseRightButtonDown` event when you right-click on a `Button`, but setting `Button`'s `Border` property has no visual effect. This is due to `Button`'s default visual tree, which was shown back in Figure 3.3. Unlike `Window`, `Label`, `ListBox`, `ListBoxItem`, and `StatusBar`, the visual tree for `Button` has no `Border` element.

FAQ

Where is the event for handling the pressing of a mouse's *middle* button?

If you browse through the various mouse events exposed by `UIElement` or `ContentElement`, you'll find events for `MouseLeftButtonDown`, `MouseLeftButtonUp`, `MouseRightButtonDown`, and `MouseRightButtonUp` (as well as the tunneling `Preview` version of each event). But what about the additional buttons present on some mice?

This information can be retrieved via the more generic `MouseDown` and `MouseUp` events (which also have `Preview` counterparts). The arguments passed to such event handlers include a `MouseButton` enumeration that indicates which button's state just changed: `Left`, `Right`, `Middle`, `XButton1`, or `XButton2`. A corresponding `MouseButtonState` enumeration indicates whether that button is `Pressed` or `Released`.

DIGGING DEEPER

Halting a Routed Event Is an Illusion

Although setting the `RoutedEventArgs` parameter's `Handled` property to `true` in a routed event handler appears to stop the tunneling or bubbling, individual handlers further up or down the tree can opt to receive the events anyway! This can only be done from procedural code, using an overload of `AddHandler` that adds a Boolean `handledEventsToo` parameter.

For example, the event attribute could be removed from Listing 3.7 and replaced with the following `AddHandler` call in `AboutDialog`'s constructor:

```
public AboutDialog()
{
  InitializeComponent();
  this.AddHandler(Window.MouseRightButtonDownEvent,
    new MouseButtonEventHandler(AboutDialog_MouseRightButtonDown), true);
}
```

With `true` passed as a third parameter, `AboutDialog_MouseRightButtonDown` now receives events when you right-click on a `ListBoxItem`, and can add the black border!

You should avoid processing handled events whenever possible, because there is likely a reason the event is handled in the first place. Attaching a handler to the `Preview` version of an event is the preferred alternative.

The bottom line, however, is that the halting of tunneling or bubbling is really just an illusion. It's more correct to say that tunneling and bubbling still continue when a routed event is marked as handled, but that event handlers only see unhandled events by default.

Attached Events

The tunneling and bubbling of a routed event is natural when every element in the tree exposes that event. But WPF supports tunneling and bubbling of routed events through elements that don't even define that event! This is possible thanks to the notion of *attached events*.

Attached events operate much like attached properties (and their use with tunneling or bubbling is very similar to using attached properties with property value inheritance). Listing 3.9 changes the About dialog again by handing the bubbling `SelectionChanged` event raised by its `ListBox` and the bubbling `Click` event raised by both of its `Buttons` directly on the root `Window`. Because `Window` doesn't define its own `SelectionChanged` or `Click` events, the event attribute names must be prefixed with the class name defining these events. Listing 3.10 contains the corresponding code-behind file that implements the two event handlers. Both event handlers simply show a `MessageBox` with information about what just happened.

LISTING 3.9 The About Dialog with Two Attached Event Handlers on the Root Window

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  x:Class="AboutDialog" ListBox.SelectionChanged="ListBox_SelectionChanged"
  Button.Click="Button_Click"
  Title="About WPF Unleashed" SizeToContent="WidthAndHeight"
  Background="OrangeRed">
  <StackPanel>
    <Label FontWeight="Bold" FontSize="20" Foreground="White">
      WPF Unleashed (Version 3.0)
    </Label>
    <Label>© 2006 SAMS Publishing</Label>
    <Label>Installed Chapters:</Label>
    <ListBox>
      <ListBoxItem>Chapter 1</ListBoxItem>
      <ListBoxItem>Chapter 2</ListBoxItem>
    </ListBox>
    <StackPanel Orientation="Horizontal" HorizontalAlignment="Center">
      <Button MinWidth="75" Margin="10">Help</Button>
      <Button MinWidth="75" Margin="10">OK</Button>
    </StackPanel>
    <StatusBar>You have successfully registered this product.</StatusBar>
  </StackPanel>
</Window>
```

LISTING 3.10 The Code-Behind File for Listing 3.9

```
using System.Windows;
using System.Windows.Controls;

public partial class AboutDialog : Window
{
  public AboutDialog()
  {
    InitializeComponent();
  }

  void ListBox_SelectionChanged(object sender, SelectionChangedEventArgs e)
  {
    if (e.AddedItems.Count > 0)
      MessageBox.Show("You just selected " + e.AddedItems[0]);
  }
```

LISTING 3.10 Continued

```
  void Button_Click(object sender, RoutedEventArgs e)
  {
    if (e.AddedItems.Count > 0)
      MessageBox.Show("You just clicked " + e.Source);
  }
}
```

Every routed event can be used as an attached event. The attached event syntax used in Listing 3.9 is valid because the XAML compiler sees the `SelectionChanged` .NET event defined on `ListBox` and the `Click` .NET event defined on `Button`. At run-time, however, `AddHandler` is directly called to attach these two events to the `Window`. Therefore, the two event attributes are equivalent to placing the following code inside the `Window`'s constructor:

```
public AboutDialog()
{
  InitializeComponent();
  this.AddHandler(ListBox.SelectionChangedEvent,
    new SelectionChangedEventHandler(ListBox_SelectionChanged));
  this.AddHandler(Button.ClickEvent, new RoutedEventHandler(Button_Click));
}
```

DIGGING DEEPER

Consolidating Routed Event Handlers

Because of the rich information passed to routed events, you could handle every event that tunnels or bubbles with one top-level "megahandler" if you really wanted to! This handler could examine the `RoutedEvent` object to determine which event got raised, cast the `RoutedEventArgs` parameter to an appropriate subclass (such as `KeyEventArgs`, `MouseButtonEventArgs`, and so on) and go from there.

For example, Listing 3.9 could be changed to assign both `ListBox.SelectionChanged` and `Button.Click` to the same `GenericHandler` method, defined as follows:

```
void GenericHandler(object sender, RoutedEventArgs e)
{
  if (e.RoutedEvent == Button.ClickEvent)
  {
    MessageBox.Show("You just clicked " + e.Source);
  }
  else if (e.RoutedEvent == ListBox.SelectionChangedEvent)
  {
```

Continued

```
    SelectionChangedEventArgs sce = (SelectionChangedEventArgs)e;
    if (sce.AddedItems.Count > 0)
      MessageBox.Show("You just selected " + sce.AddedItems[0]);
  }
}
```

This is also made possible by the *delegate contravariance* feature added in version 2.0 of the .NET Framework, enabling a delegate to be used with a method whose signature uses a base class of an expected parameter (e.g. `RoutedEventArgs` instead of `SelectionChangedEventArgs`). `GenericHandler` simply casts the `RoutedEventArgs` parameter when necessary to get the extra information specific to the `SelectionChanged` event.

Commands

WPF provides built-in support for *commands*, a more abstract and loosely-coupled version of events. Whereas events are tied to details about specific user actions (such as a `Button` being clicked or a `ListBoxItem` being selected), commands represent actions independent from their user interface exposure. Canonical examples of commands are Cut, Copy, and Paste. Applications often expose these actions through many mechanisms simultaneously: `MenuItems` in a `Menu`, `MenuItems` on a `ContextMenu`, `Buttons` on a `ToolBar`, keyboard shortcuts, and so on.

You could handle the multiple exposures of commands such as Cut, Copy, and Paste with events fairly well. For example, you could define a generic event handler for each of the three actions and then attach each handler to the appropriate events on the relevant elements (the `Click` event on a `Button`, the `KeyDown` event on the main `Window`, and so on). In addition, you'd probably want to enable and disable the appropriate controls whenever the corresponding actions are invalid (for example, disabling any user interface for Paste when there is nothing on the clipboard). This two-way communication gets a bit more cumbersome, especially if you don't want to hard-code a list of controls that need updating.

Fortunately, WPF's support for commands is designed to make such scenarios very easy. The support reduces the amount of code you need to write (and in some cases eliminating all procedural code), and it gives you more flexibility to change your user interface without breaking the back-end logic. Commands are not a new invention of WPF; older technologies such as Microsoft Foundation Classes (MFC) have a similar mechanism. Of course, even if you're familiar with MFC, commands in WPF have their own unique traits to learn about.

Much of the power of commands comes from the following three features:

- WPF defines a number of built-in commands.
- Commands have automatic support for input gestures (such as keyboard shortcuts).
- Some of WPF's controls have built-in behavior tied to various commands.

Built-In Commands

A command is any object implementing the `ICommand` interface (from `System.Windows.Input`), which defines three simple members:

- **`Execute`**—The method that executes the command-specific logic
- **`CanExecute`**—A method returning `true` if the command is enabled or `false` if it is disabled
- **`CanExecuteChanged`**—An event that is raised whenever the value of `CanExecute` changes

If you want to create Cut, Copy, and Paste commands, you could define and implement three classes implementing `ICommand`, find a place to store them (perhaps as static fields of your main `Window`), call `Execute` from relevant event handlers (when `CanExecute` returns `true`), and handle the `CanExecuteChanged` event to toggle the `IsEnabled` property on the relevant pieces of user interface. This doesn't sound much better than simply using events, however.

Fortunately, controls such as `Button`, `CheckBox`, and `MenuItem` have logic to interact with any command on your behalf. They expose a simple `Command` property (of type `ICommand`). When set, these controls automatically call the command's `Execute` method (when `CanExecute` returns true) whenever their `Click` event is raised. In addition, they automatically keep their value for `IsEnabled` synchronized with the value of `CanExecute` by leveraging the `CanExecuteChanged` event. By supporting all this via a simple property assignment, all of this logic is available from XAML.

Even more fortunately, WPF defines a bunch of commands already, so you don't have to implement `ICommand` objects for Cut, Copy, and Paste and worry about where to store them. WPF's built-in commands are exposed as static properties of five different classes:

- **`ApplicationCommands`**—`Close`, `Copy`, `Cut`, `Delete`, `Find`, `Help`, `New`, `Open`, `Paste`, `Print`, `PrintPreview`, `Properties`, `Redo`, `Replace`, `Save`, `SaveAs`, `SelectAll`, `Stop`, `Undo`, and more
- **`ComponentCommands`**—`MoveDown`, `MoveLeft`, `MoveRight`, `MoveUp`, `ScrollByLine`, `ScrollPageDown`, `ScrollPageLeft`, `ScrollPageRight`, `ScrollPageUp`, `SelectToEnd`, `SelectToHome`, `SelectToPageDown`, `SelectToPageUp`, and more
- **`MediaCommands`**—`ChannelDown`, `ChannelUp`, `DecreaseVolume`, `FastForward`, `IncreaseVolume`, `MuteVolume`, `NextTrack`, `Pause`, `Play`, `PreviousTrack`, `Record`, `Rewind`, `Select`, `Stop`, and more

- **NavigationCommands**—BrowseBack, BrowseForward, BrowseHome, BrowseStop, Favorites, FirstPage, GoToPage, LastPage, NextPage, PreviousPage, Refresh, Search, Zoom, and more
- **EditingCommands**—AlignCenter, AlignJustify, AlignLeft, AlignRight, CorrectSpellingError, DecreaseFontSize, DecreaseIndentation, EnterLineBreak, EnterParagraphBreak, IgnoreSpellingError, IncreaseFontSize, IncreaseIndentation, MoveDownByLine, MoveDownByPage, MoveDownByParagraph, MoveLeftByCharacter, MoveLeftByWord, MoveRightByCharacter, MoveRightByWord, and more

Each of these properties does not return a unique type implementing `ICommand`. Instead, they are all instances of `RoutedUICommand`, a class that not only implements `ICommand`, but supports bubbling just like a routed event.

The About dialog has a "Help" `Button` that currently does nothing, so let's demonstrate how these built-in commands work by attaching some logic with the `Help` command defined in `ApplicationCommands`. Assuming the `Button` is named `helpButton`, you can associate it with the `Help` command in C# as follows:

```
helpButton.Command = ApplicationCommands.Help;
```

All `RoutedUICommand` objects define a `Text` property containing a name for the command that's appropriate to show in a user interface. (This property is the only difference between `RoutedUICommand` and its base `RoutedCommand` class.) For example, the `Help` command's `Text` property is (unsurprisingly) set to the string `Help`. The hard-coded `Content` on this `Button` could therefore be replaced as follows:

```
helpButton.Content = ApplicationCommands.Help.Text;
```

If you were to run the About dialog with this change, you would see that the `Button` is now permanently disabled. That's because the built-in commands can't possibly know when they should be enabled or disabled, or even what action to take when they are executed. They delegate this logic to consumers of the commands.

To plug in custom logic, you need to add a `CommandBinding` to the element that will execute the command *or any parent element* (thanks to the bubbling behavior of routed commands). All classes deriving from `UIElement` (and

TIP

The `Text` string defined by all `RoutedUICommands` is automatically localized into every language supported by WPF! This means that a `Button` whose `Content` is assigned to `ApplicationCommands.Help.Text` automatically displays "Ayuda" rather than "Help" when the thread's current UI culture represents Spanish rather than English. Even in a context where you want to expose images rather than text (perhaps on a `ToolBar`), you can still leverage this localized string elsewhere, such as in a `ToolTip`.

Of course, you're still responsible for localizing any of your own strings that get displayed in your user interface. Leveraging `Text` on commands can simply cut down on the number of terms you need to translate.

ContentElement) contain a CommandBindings collection that can hold one or more CommandBinding objects. Therefore, you can add a CommandBinding for Help to the About dialog's root Window as follows in its code-behind file:

```
this.CommandBindings.Add(new CommandBinding(ApplicationCommands.Help,
  HelpExecuted, HelpCanExecute));
```

This assumes that methods called HelpExecuted and HelpCanExecute have been defined. These methods will be called back at appropriate times in order to plug in an implementation for the Help command's CanExecute and Execute methods.

Listings 3.11 and 3.12 change the About dialog one last time, binding the Help Button to the Help command entirely in XAML (although the two handlers must be defined in the code-behind file).

LISTING 3.11 The About Dialog Supporting the Help Command

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  x:Class="AboutDialog"
  Title="About WPF Unleashed" SizeToContent="WidthAndHeight"
  Background="OrangeRed">
<Window.CommandBindings>
  <CommandBinding Command="Help"
    CanExecute="HelpCanExecute" Executed="HelpExecuted"/>
</Window.CommandBindings>
  <StackPanel>
    <Label FontWeight="Bold" FontSize="20" Foreground="White">
      WPF Unleashed (Version 3.0)
    </Label>
    <Label>© 2006 SAMS Publishing</Label>
    <Label>Installed Chapters:</Label>
    <ListBox>
      <ListBoxItem>Chapter 1</ListBoxItem>
      <ListBoxItem>Chapter 2</ListBoxItem>
    </ListBox>
    <StackPanel Orientation="Horizontal" HorizontalAlignment="Center">
      <Button MinWidth="75" Margin="10" Command="Help" Content=
        "{Binding RelativeSource={RelativeSource Self}, Path=Command.Text}"/>
      <Button MinWidth="75" Margin="10">OK</Button>
    </StackPanel>
    <StatusBar>You have successfully registered this product.</StatusBar>
  </StackPanel>
</Window>
```

LISTING 3.12 The Code-Behind File for Listing 3.11

```
using System.Windows;
using System.Windows.Input;

public partial class AboutDialog : Window
{
  public AboutDialog()
  {
    InitializeComponent();
  }

  void HelpCanExecute(object sender, CanExecuteRoutedEventArgs e)
  {
    e.CanExecute = true;
  }

  void HelpExecuted(object sender, ExecutedRoutedEventArgs e)
  {
    System.Diagnostics.Process.Start("http://www.adamnathan.net/wpf");
  }
}
```

`Window`'s `CommandBinding` can be set in XAML because it defines a default constructor and enables its data to be set with properties. `Button`'s `Content` can even be set to the chosen command's `Text` property in XAML thanks to a popular data binding technique discussed in Chapter 9. In addition, notice that a type converter simplifies specifying the `Help` command in XAML. A `CommandConverter` class knows about all the built-in commands, so the `Command` property can be set to `Help` in both places rather than the more verbose `{x:Static ApplicationCommands.Help}`. (Custom commands don't get the same special treatment.) In the code-behind file, `HelpCanExecute` keeps the command enabled at all times, and `HelpExecuted` launches a web browser with an appropriate help URL.

Executing Commands with Input Gestures

Using the `Help` command in such a simple dialog may seem like overkill when a simple event handler for `Click` would do, but the command has provided an extra benefit (other than localized text): automatic binding to a keyboard shortcut.

Applications typically invoke their version of help when the user presses the F1 key. Sure enough, if you press F1 while displaying the dialog defined in Listing 3.10, the `Help` command is automatically launched, as if you clicked the Help `Button`! That's because commands such as `Help` define a default *input gesture* that executes the command. You can bind your own input gestures to a command by adding `KeyBinding` and/or `MouseBinding` objects to the relevant element's `InputBindings` collection. For example, to

assign F2 as a keyboard shortcut that executes `Help`, you could add the following statement to `AboutDialog`'s constructor:

```
this.InputBindings.Add(
  new KeyBinding(ApplicationCommands.Help, new KeyGesture(Key.F2)));
```

This would make *both* F1 and F2 execute `Help`, however. You could additionally suppress the default F1 behavior by binding F1 to a special `NotACommand` command as follows:

```
this.InputBindings.Add(
  new KeyBinding(ApplicationCommands.NotACommand, new KeyGesture(Key.F1)));
```

Both of these statements could alternatively be represented in XAML as follows:

```
<Window.InputBindings>
  <KeyBinding Command="Help" Key="F2"/>
  <KeyBinding Command="NotACommand" Key="F1"/>
</Window.InputBindings>
```

Controls with Built-In Command Bindings

It can seem almost magical when you encounter it, but some controls in WPF contain their own command bindings. The simplest example of this is the `TextBox` control, which has its own built-in bindings for the `Cut`, `Copy`, and `Paste` commands that interact with the clipboard, as well as `Undo` and `Redo` commands. This not only means that `TextBox` responds to the standard Ctrl+X, Ctrl+C, Ctrl+V, Ctrl+Z, and Ctrl+Y keyboard shortcuts, but that it's easy for additional elements to participate in these actions.

The following standalone XAML demonstrates the power of these built-in command bindings:

```
<StackPanel xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Orientation="Horizontal" Height="25">
  <Button Command="Cut" CommandTarget="{Binding ElementName=textBox}"
    Content="{Binding RelativeSource={RelativeSource Self}, Path=Command.Text}"/>
  <Button Command="Copy" CommandTarget="{Binding ElementName=textBox}"
    Content="{Binding RelativeSource={RelativeSource Self}, Path=Command.Text}"/>
  <Button Command="Paste" CommandTarget="{Binding ElementName=textBox}"
    Content="{Binding RelativeSource={RelativeSource Self}, Path=Command.Text}"/>
  <Button Command="Undo" CommandTarget="{Binding ElementName=textBox}"
    Content="{Binding RelativeSource={RelativeSource Self}, Path=Command.Text}"/>
  <Button Command="Redo" CommandTarget="{Binding ElementName=textBox}"
    Content="{Binding RelativeSource={RelativeSource Self}, Path=Command.Text}"/>
  <TextBox x:Name="textBox" Width="200"/>
</StackPanel>
```

You can paste this content into XamlPad or save it as a `.xaml` file to view in Internet Explorer, because no procedural code is necessary. Each of the five `Buttons` is associated

with one of the commands and sets its `Content` to the string returned by each command's `Text` property. The only new thing here is the setting of each `Button`'s `CommandTarget` property to the instance of the `TextBox` (again using data binding functionality discussed in Chapter 9). This causes the command to be executed from the `TextBox` rather than the `Button`, which is necessary in order for it to react to the commands.

This XAML produces the result in Figure 3.8. The first two `Buttons` are automatically disabled when no text in the `TextBox` is selected, and automatically enabled when there is a selection. Similarly, the Paste `Button` is automatically enabled whenever there is text content on the clipboard, or disabled otherwise.

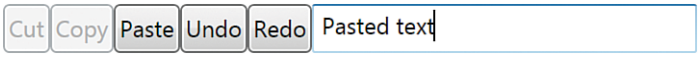

FIGURE 3.8 The five `Buttons` work as expected without any procedural code, thanks to `TextBox`'s built-in bindings.

`Button` and `TextBox` have no direct knowledge of each other, yet though commands they can achieve rich interaction. This is why WPF's long list of built-in commands is so important. The more that third-party controls standardize on WPF's built-in commands, the more seamless (and declarative) interaction can be achieved among controls that have no direct knowledge of each other.

A Tour of the Class Hierarchy

WPF's classes have a very deep inheritance hierarchy, so it can be hard to get your head wrapped around the significance of various classes and their relationships. The inside cover of this book contains a map of these classes to help you put them in perspective as you encounter new ones. It is incomplete due to space constraints, but the major classes are covered.

A handful of classes are fundamental to the inner-workings of WPF, and deserve a quick explanation before we get any further in the book. Some of these have been mentioned in passing already. Figure 3.9 shows these important classes and their relationships without all the extra clutter from the inside cover.

These ten classes have the following significance:

- **`Object`**—The base class for all .NET classes.
- **`DispatcherObject`**—The base class for any object that wishes to be accessed only on the thread that created it. Most WPF classes derive from `DispatcherObject`, and are therefore inherently thread-unsafe. The `Dispatcher` part of the name refers to WPF's version of a Win32-like message loop, discussed further in Chapter 7, "Structuring and Deploying an Application."
- **`DependencyObject`**—The base class for any object that can support dependency properties. `DependencyObject` defines the `GetValue` and `SetValue` methods that are central to the operation of dependency properties.

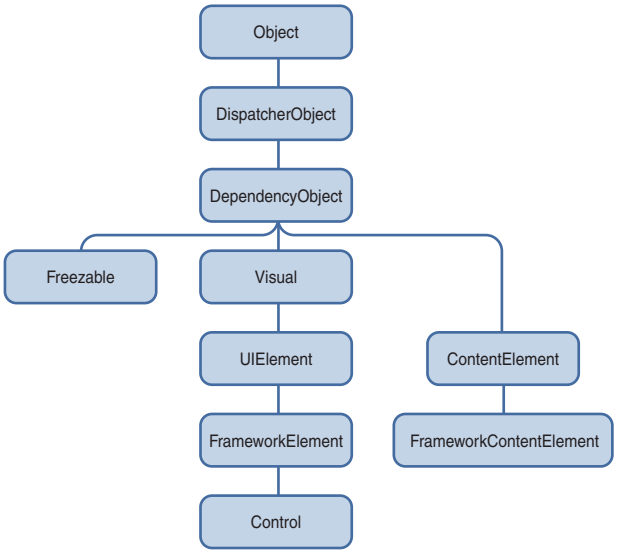

FIGURE 3.9 The core classes in the WPF Presentation Framework.

- **`Freezable`**—The base class for objects that can be "frozen" into a read-only state for performance reasons. `Freezables`, once frozen, can even be safely shared among multiple threads, unlike all other `DispatcherObjects`. Frozen objects can never be unfrozen, but you can clone them to create unfrozen copies.
- **`Visual`**—The base class for all objects that have their own visual representation. `Visuals` are discussed in depth in Chapter 11.
- **`UIElement`**—The base class for all visual objects with support for routed events, command binding, layout, and focus.
- **`ContentElement`**—A base class similar to `UIElement`, but for pieces of content that don't have rendering behavior on their own. Instead, `ContentElements` are hosted in a `Visual`-derived class to be rendered on the screen.
- **`FrameworkElement`**—The base class that adds support for styles, data binding, resources, and a few common mechanisms for Windows-based controls such as tooltips and context menus.
- **`FrameworkContentElement`**—The analog to `FrameworkElement` for content. Chapter 14 examines the `FrameworkContentElements` in WPF.
- **`Control`**—The base class for familiar controls such as `Button`, `ListBox`, and `StatusBar`. `Control` adds many properties to its `FrameworkElement` base class, such as `Foreground`, `Background`, and `FontSize`. `Controls` also support templates that enable you to completely replace their visual tree, discussed in Chapter 10. The next chapter examines WPF's `Controls` in depth.

Throughout the book, the simple term *element* is used to refer to an object that derives from `UIElement` or `FrameworkElement`, and sometimes `ContentElement` or `FrameworkContentElement`. The distinction between `UIElement` versus `FrameworkElement` or `ContentElement` versus `FrameworkContentElement` is not important because WPF doesn't ship any other public subclasses of `UIElement` and `ContentElement`.

Conclusion

In this chapter and the preceding two chapters, you've learned about all the major ways that WPF builds on top of the foundation of the .NET Framework. The WPF team could have exposed its features via typical .NET APIs similar to Windows Forms and still have an interesting technology. Instead, the team added several fundamental concepts that enable a wide range of features to be exposed in a way that can provide great productivity for developers and designers.

Indeed, when you focus on these core concepts (as this chapter has done), you can see that the landscape isn't quite as simple as it used to be: There are multiple types of properties, multiple types of events, multiple trees, and multiple ways of achieving the same results (such as writing declarative versus procedural code)! Hopefully you can now appreciate some of the value of these new mechanisms. Throughout the rest of the book, these concepts generally fade into the background as we focus on accomplishing specific development tasks.

Because of the (primitive) examples used in this chapter, you should now have a feel for some of WPF's controls and how WPF user interfaces are arranged. The next three chapters build on this by formally introducing you to WPF's controls and layout mechanisms.

This page intentionally left blank