CHAPTER 11

# Styles and Behaviors

WPF applications would be a drab bunch if you were limited to the plain, gray look of ordinary buttons and other common controls. Fortunately, WPF has several features that allow you to inject some flair into basic elements and standardize the visual appearance of your application. In this chapter, you'll learn about two of the most important: styles and behaviors.

*Styles* are an essential tool for organizing and reusing for formatting choices. Rather than filling your XAML with repetitive markup to set details such as margins, padding, colors, and fonts, you can create a set of styles that encompass all these details. You can then apply the styles where you need them by setting a single property.

*Behaviors* are a more ambitious tool for reusing user interface code. The basic idea is that a behavior encapsulates a common bit of UI functionality (for example, the code that makes an element draggable). If you have the right behavior, you can attach it to any element with a line or two of XAML markup, saving you the effort of writing and debugging the code yourself.

## Style Basics

In the previous chapter, you learned about the WPF resource system, which lets you define objects in one place and reuse them throughout your markup. Although you can use resources to store a wide variety of objects, one of the most common reasons you'll use them is to hold s*tyles*.

A style is a collection of property values that can be applied to an element. The WPF style system plays a similar role to the Cascading Style Sheets (CSS) standard in HTML markup. Like CSS, WPF styles allow you to define a common set of formatting characteristics and apply them throughout your application to ensure consistency. And as with CSS, WPF styles can work automatically, target specific element types, and cascade through the element tree. However, WPF styles are more powerful because they can set *any* dependency property. That means you can use them to standardize characteristics that have nothing to do with formatting, such as properties that control the behavior of a control. WPF styles also support *triggers*, which allow you to change the style of a control when another property is changed (as you'll see in this chapter), and they can use *templates* to redefine the built-in appearance of a control (as you'll see in Chapter 17). Once you've learned how to use styles, you'll be sure to include them in all your WPF applications.

To understand how styles fit in, it helps to consider a simple example. Imagine you need to standardize the font that's used in a window. The simplest approach is to set the font properties of the containing window. These properties, which are defined in the Control class, include FontFamily, FontSize, FontWeight (for bold), FontStyle (for italics), and FontStretch (for compressed and expanded variants).



www.it-ebooks.info

Thanks to the property value inheritance feature, when you set these properties at the window level, all the elements inside the window will acquire the same values, unless they explicitly override them.

---

■ **Note** Property value inheritance is one of the many optional features that dependency properties can provide. Dependency properties are described in Chapter 4.

---

Now consider a different situation, one in which you want to lock down the font that's used for just a portion of your user interface. If you can isolate these elements in a specific container (for example, if they're all inside one Grid or StackPanel), you can use essentially the same approach and set the font properties of the container. However, life is not usually that easy. For example, you may want to give all buttons a consistent typeface and text size independent from the font settings that are used in other elements. In this case, you need a way to define these details in one place and reuse them wherever they apply.

Resources give you a solution, but it's somewhat awkward. Because there's no Font object in WPF (just a collection of font-related properties), you're stuck defining several related resources, as shown here:

```
<Window.Resources>
  <FontFamily x:Key="ButtonFontFamily">Times New Roman</FontFamily>
  <sys:Double x:Key="ButtonFontSize">18</s:Double>
  <FontWeight x:Key="ButtonFontWeight">Bold</FontWeight>
</Window.Resources>
```

This snippet or markup adds three resources to a window: a FontFamily object with the name of the font you want to use, a double that stores the number 18, and the enumerated value FontWeight.Bold. It assumes you've mapped the .NET namespace System to the XML namespace prefix sys, as shown here:

```
<Window xmlns:sys="clr-namespace:System;assembly=mscorlib" ... >
```

---

■ **Tip** When setting properties using a resource, it's important that the data types match exactly. WPF won't use a type converter in the same way it does when you set an attribute value directly. For example, if you're setting the FontFamily attribute in an element, you can use the string "Times New Roman" because the FontFamilyConverter will create the FontFamily object you need. However, the same magic won't happen if you try to set the FontFamily property using a string resource—in this situation, the XAML parser throws an exception.

---

Once you've defined the resources you need, the next step is to actually use these resources in an element. Because the resources are never changed over the lifetime of the application, it makes sense to use static resources, as shown here:

```
<Button Padding="5" Margin="5" Name="cmd"
 FontFamily="{StaticResource ButtonFontFamily}"
 FontWeight="{StaticResource ButtonFontWeight}"
 FontSize="{StaticResource ButtonFontSize}">
  A Customized Button
</Button>
```

This example works, and it moves the font details (the so-called magic numbers) out of your markup. However, it also presents two new problems:



www.it-ebooks.info

- There's no clear indication that the three resources are related (other than the similar resource names). This complicates the maintainability of the application. It's especially a problem if you need to set more font properties or if you decide to maintain different font settings for different types of elements.
- The markup you need to use your resources is quite verbose. In fact, it's less concise than the approach it replaces (defining the font properties directly in the element).

You could improve on the first issue by defining a custom class (such as FontSettings) that bundles all the font details together. You could then create one FontSettings object as a resource and use its various properties in your markup. However, this still leaves you with verbose markup—and it makes for a fair bit of extra work.

Styles provide the perfect solution. You can define a single style that wraps all the properties you want to set. Here's how:

```
<Window.Resources>
  <Style x:Key="BigFontButtonStyle">
    <Setter Property="Control.FontFamily" Value="Times New Roman" />
    <Setter Property="Control.FontSize" Value="18" />
    <Setter Property="Control.FontWeight" Value="Bold" />
  </Style>
</Window.Resources>
```

This markup creates a single resource: a System.Windows.Style object. This style object holds a Setters collection with three Setter objects, one for each property you want to set. Each Setter object names the property that it acts on and the value that it applies to that property. Like all resources, the style object has a key name so you can pull it out of the collection when needed. In this case, the key name is BigFontButtonStyle. (By convention, the key names for styles usually end with *Style*.)

Every WPF element can use a single style (or no style). The style plugs into an element through the element's Style property (which is defined in the base FrameworkElement class). For example, to configure a button to use the style you created previously, you'd point the button to the style resource like this:

```
<Button Padding="5" Margin="5" Name="cmd"
 Style="{StaticResource BigFontButtonStyle}">
  A Customized Button
</Button>
```

Of course, you could also set a style programmatically. All you need to do is pull the style out of the closest Resources collection using the familiar FindResource() method. Here's the code you'd use for a Button object named cmd:

```
cmd.Style = (Style)cmd.FindResource("BigFontButtonStyle");
```

Figure 11-1 shows a window with two buttons that use the BigFontButtonStyle.



www.it-ebooks.info

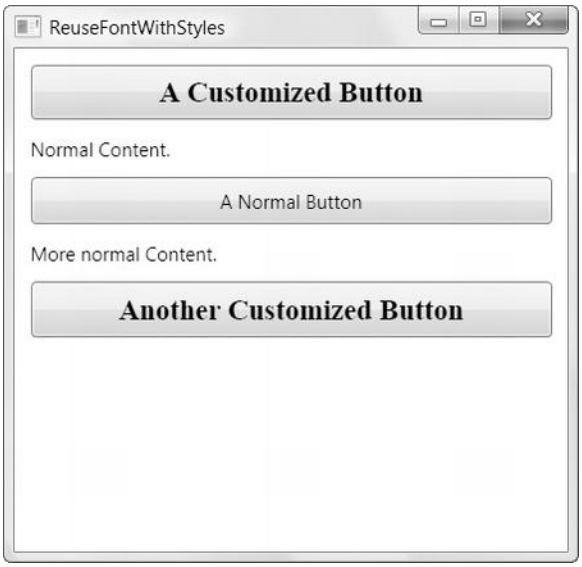


***Figure 11-1.*** *Reusing button settings with a style*

■ **Note** Styles set the initial appearance of an element, but you're free to override the characteristics they set. For example, if you apply the BigFontButtonStyle style and set the FontSize property explicitly, the FontSize setting in the button tag overrides the style. Ideally, you won't rely on this behavior—instead, create more styles so that you can set as many details as possible at the style level. This gives you more flexibility to adjust your user interface in the future with minimum disruption.

The style system adds many benefits. Not only does it allow you to create groups of settings that are clearly related, it also streamlines your markup by making it easier to apply these settings. Best of all, you can apply a style without worrying about what properties it sets. In the previous example, the font settings were organized into a style named BigFontButtonStyle. If you decide later that your big-font buttons also need more padding and margin space, you can add setters for the Padding and Margin properties as well. All the buttons that use the style automatically acquire the new style settings.

The Setters collection is the most important property of the Style class. But there are five key properties altogether, which you'll consider in this chapter. Table 11-1 shows a snapshot.

***Table 11-1.*** *Properties of the Style Class*

| Property | Description |
|---|---|
| Setters | A collection of Setter or EventSetter objects that set property values and attach event handlers automatically. |
| Triggers | A collection of objects that derive from TriggerBase and allow you to change style settings automatically. For example, you can modify a style when another property changes or when an event occurs. |



www.it-ebooks.info

| | |
|---|---|
| Resources | A collection of resources that you want to use with your styles. For example, you might need to use a single object to set more than one property. In that case, it's more efficient to create the object as a resource and then use that resource in your Setter object (rather than create the object as part of each Setter, using nested tags). |
| BasedOn | A property that allows you to create a more specialized style that inherits (and optionally overrides) the settings of another style. |
| TargetType | A property that identifies the element type that this style acts upon. This property allows you to create setters that only affect certain elements, and it allows you to create setters that spring into action automatically for the right element type. |

Now that you've seen a basic example of a style at work, you're ready to look into the style model more deeply.

## Creating a Style Object

In the previous example, the style object is defined at the window level and then reused in two buttons inside that window. Although that's a common design, it's certainly not your only choice.

If you want to create more finely targeted styles, you could define them using the Resources collection of their container, such as a StackPanel or a Grid. If you want to reuse styles across an application, you can define them using the Resources collection of your application. These are also common approaches.

Strictly speaking, you don't need to use styles and resources together. For example, you could define the style of a particular button by filling its Style collection directly, as shown here:

```
<Button Padding="5" Margin="5">
  <Button.Style>
    <Style>
      <Setter Property="Control.FontFamily" Value="Times New Roman" />
      <Setter Property="Control.FontSize" Value="18" />
      <Setter Property="Control.FontWeight" Value="Bold" />
    </Style>
  </Button.Style>
  <Button.Content>A Customized Button</Button.Content>
</Button>
```

This works, but it's obviously a lot less useful. Now there's no way to share this style with other elements.

This approach isn't worth the trouble if you're simply using a style to set some properties (as in this example) because it's easier to set the properties directly. However, this approach is occasionally useful if you're using another feature of styles and you want to apply it to a single element only. For example, you can use this approach to attach triggers to an element. This approach also allows you to modify a part of an element's control template. (In this case, you use the Setter.TargetName property to apply a setter to a specific component inside the element, such as the scroll bar buttons in a list box. You'll learn more about this technique in Chapter 17.)



www.it-ebooks.info

## Setting Properties

As you've seen, every Style object wraps a collection of Setter objects. Each Setter object sets a single property in an element. The only limitation is that a setter can only change a dependency property—other properties can't be modified.

In some cases, you won't be able to set the property value using a simple attribute string. For example, an ImageBrush object can't be created with a simple string. In this situation, you can use the familiar XAML trick of replacing the attribute with a nested element. Here's an example:

```
<Style x:Key="HappyTiledElementStyle">
  <Setter Property="Control.Background">
    <Setter.Value>
      <ImageBrush TileMode="Tile"
        ViewportUnits="Absolute" Viewport="0 0 32 32"
        ImageSource="happyface.jpg" Opacity="0.3">
      </ImageBrush>
    </Setter.Value>
  </Setter>
</Style>
```

---

■ **Tip** If you want to reuse the same image brush in more than one style (or in more than one setter in the same style), you can define it as a resource and then use that resource in your style.

---

To identify the property you want to set, you need to supply both a class and a property name. However, the class name you use doesn't need to be the class where the property is defined. It can also be a derived class that inherits the property. For example, consider the following version of the BigFontButton style, which replaces the references to the Control class with references to the Button class:

```
<Style x:Key="BigFontButtonStyle">
  <Setter Property="Button.FontFamily" Value="Times New Roman" />
  <Setter Property="Button.FontSize" Value="18" />
  <Setter Property="Button.FontWeight" Value="Bold" />
</Style>
```

If you substitute this style in the same example (Figure 11-1), you'll get the same result. So, why the difference? In this case, the distinction is how WPF handles other classes that may include the same FontFamily, FontSize, and FontWeight properties but that don't derive from Button. For example, if you apply this version of the BigFontButton style to a Label control, it has no effect. WPF simply ignores the three properties because they don't apply. But if you use the original style, the font properties will affect the label because the Label class derives from Control.

---

■ **Tip** The fact that WPF ignores properties that don't apply means you can also set properties that won't necessarily be available in the element to which you apply the style. For example, if you set the ButtonBase.IsCancel property, it will have an effect only when you set the style on a button.

---

There are some cases in WPF where the same properties are defined in more than one place in the element hierarchy. For example, the full set of font properties (such as FontFamily) is defined in both the



www.it-ebooks.info

Control class and the TextBlock class. If you're creating a style that applies to TextBlock objects and elements that derive from Control, it might occur to you to create markup like this:

```
<Style x:Key="BigFontStyle">
  <Setter Property="Button.FontFamily" Value="Times New Roman" />
  <Setter Property="Button.FontSize" Value="18" />

  <Setter Property="TextBlock.FontFamily" Value="Arial" />
  <Setter Property="TextBlock.FontSize" Value="10" />
</Style>
```

However, this won't have the desired effect. The problem is that although Button.FontFamily and TextBlock.FontFamily are declared separately in their respective base classes, they are both references to the same dependency property. (In other words, TextBlock.FontSizeProperty and Control.FontSizeProperty are references that point to the same DependencyProperty object. You first learned about this possible issue in Chapter 4.) As a result, when you use this style, WPF sets the FontFamily and FontSize property twice. The last-applied settings (in this case, 10-unit Arial) take precedence and are applied to both Button and TextBlock objects. Although this problem is fairly specific and doesn't occur with many properties, it's important to be on the lookout for it if you often create styles that apply different formatting to different element types.

There's one more trick that you can use to simplify style declarations. If all your properties are intended for the same element type, you can set the TargetType property of the Style object to indicate the class to which your properties apply. For example, if you're creating a button-only style, you could create the style like this:

```
<Style x:Key="BigFontButtonStyle" TargetType="Button">
  <Setter Property="FontFamily" Value="Times New Roman" />
  <Setter Property="FontSize" Value="18" />
  <Setter Property="FontWeight" Value="Bold" />
</Style>
```

This is a relatively minor convenience. As you'll discover later, the TargetType property also doubles as a shortcut that allows you to apply styles automatically if you leave out the style key name.

## Attaching Event Handlers

Property setters are the most common ingredient in any style, but you can also create a collection of EventSetter objects that wire up events to specific event handlers. Here's an example that attaches the event handlers for the MouseEnter and MouseLeave events:

```
<Style x:Key="MouseOverHighlightStyle">
  <EventSetter Event="TextBlock.MouseEnter" Handler="element_MouseEnter" />
  <EventSetter Event="TextBlock.MouseLeave" Handler="element_MouseLeave" />
  <Setter Property="TextBlock.Padding" Value="5"/>
</Style>
```

Here's the event handling code:

```
private void element_MouseEnter(object sender, MouseEventArgs e)
{
    ((TextBlock)sender).Background =
      new SolidColorBrush(Colors.LightGoldenrodYellow);
}
```



www.it-ebooks.info

```
private void element_MouseLeave(object sender, MouseEventArgs e)
{
    ((TextBlock)sender).Background = null;
}
```

MouseEnter and MouseLeave use direct event routing, which means they don't bubble up or tunnel down the element tree. If you want to apply a mouseover effect to a large number of elements (for example, you want to change the background color of an element when the mouse moves overtop of it), you need to add the MouseEnter and MouseLeave event handlers to each element. The style-based event handlers simplify this task. Now you simply need to apply a single style, which can include property setters and event setters:

```
<TextBlock Style="{StaticResource MouseOverHighlightStyle}">
 Hover over me.
</TextBlock>
```

Figure 11-2 shows a simple demonstration of this technique with three elements, two of which use the MouseOverHighlightStyle.


***Figure 11-2.*** *Handling the MouseEnter and MouseLeave events with a style*

Event setters are a rare technique in WPF. If you need the functionality shown here, you're more likely to use event triggers, which define the action you want declaratively (and so require no code). Event triggers are designed to implement animations, which makes them more useful when creating mouseover effects.

Event setters aren't a good choice when handling an event that uses bubbling. In this situation, it's usually easier to handle the event you want on a higher-level element. For example, if you want to link all the buttons in a toolbar to the same event handler for the Click event, the best approach is to attach a



www.it-ebooks.info

single event handler to the Toolbar element that holds all the buttons. In this situation, an event setter is an unnecessary complication.

---

■ **Tip** In many cases it's clearer to explicitly define all your events and avoid event setters altogether. If you need to link several events to the same event handler, do it by hand. You can also use tricks such as attaching an event handler at the container level and centralizing logic with commands (Chapter 9).

---

## The Many Layers of Styles

Although you can define an unlimited number of styles at many different levels, each WPF element can use only a single style object at once. Although this might appear to be a limitation at first, it usually isn't because of property value inheritance and style inheritance.

For example, imagine you want to give a group of controls the same font without applying the same style to each one. In this case, you may be able to place them in a single panel (or another type of container) and set the style of the container. As long as you're setting properties that use the property value inheritance feature, these values will flow down to the children. Properties that use this model include IsEnabled, IsVisible, Foreground, and all the font properties.

In other cases, you might want to create a style that builds upon another style. You can use this sort of style inheritance by setting the BasedOn attribute of a style. For example, consider these two styles:

```
<Window.Resources>
  <Style x:Key="BigFontButtonStyle">
    <Setter Property="Control.FontFamily" Value="Times New Roman" />
    <Setter Property="Control.FontSize" Value="18" />
    <Setter Property="Control.FontWeight" Value="Bold" />
  </Style>

  <Style x:Key="EmphasizedBigFontButtonStyle"
    BasedOn="{StaticResource BigFontButtonStyle}">
    <Setter Property="Control.Foreground" Value="White" />
    <Setter Property="Control.Background" Value="DarkBlue" />
  </Style>
</Window.Resources>
```

The first style (BigFontButtonStyle) defines three font properties. The second style (EmphasizedBigFontButtonStyle) acquires these aspects from BigFontButtonStyle and then supplements them with two more properties that change the foreground and the background brushes. This two-part design gives you the ability to apply just the font settings or the font-and-color combination. This design also allows you to create more styles that incorporate the font or color details you've defined (but not necessarily both).

---

■ **Note** You can use the BasedOn property to create an entire chain of inherited styles. The only rule is that if you set the same property twice, the last property setter (the one in the derived class furthest down the inheritance chain) overrides any earlier definitions.

---

Figure 11-3 shows style inheritance at work in a simple window that uses both styles.



www.it-ebooks.info


***Figure 11-3.*** *Creating a style based on another style*

### STYLE INHERITANCE ADDS COMPLEXITY

Although style inheritance seems like a great convenience at first glance, it's usually not worth the trouble. That's because style inheritance is subject to the same problems as code inheritance: dependencies that make your application more fragile. For example, if you use the markup shown previously, you're forced to keep the same font characteristics for two styles. If you decide to change BigFontButtonStyle, EmphasizedBigFontButtonStyle changes as well—unless you explicitly add more setters that override the inherited values.

This problem is trivial enough in the two-style example, but it becomes a significant issue if you use style inheritance in a more realistic application. Usually, styles are categorized based on different types of content and the role that the content plays. For example, a sales application might include styles such as ProductTitleStyle, ProductTextStyle, HighlightQuoteStyle, NavigationButtonStyle, and so on. If you base ProductTitleStyle on ProductTextStyle (perhaps because they both share the same font), you'll run into trouble if you apply settings to ProductTextStyle later that you don't want to apply to ProductTitleStyle (such as different margins). In this case, you'll be forced to define your settings in ProductTextStyle and explicitly override them in ProductTitleStyle. At the end, you'll be left with a more complicated model and very few style settings that are actually reused.

Unless you have a specific reason to base one style on another (for example, the second style is a special case of the first and changes just a few characteristics out of a large number of inherited settings), don't use style inheritance.

## Automatically Applying Styles by Type

So far, you've seen how to create named styles and refer to them in your markup. However, there's another approach. You can apply a style automatically to elements of a certain type.



www.it-ebooks.info

Doing this is quite easy. You simply need to set the TargetType property to indicate the appropriate type (as described earlier) and leave out the key name altogether. When you do this, WPF actually sets the key name implicitly using the type markup extension, as shown here:

```
x:Key="{x:Type Button}"
```

Now the style is automatically applied to any buttons all the way down the element tree. For example, if you define a style in this way on the window, it applies to every button in that window (unless there's a style further downstream that replaces it).

Here's an example with a window that sets the button styles automatically to get the same effect you saw in Figure 11-1:

```
<Window.Resources>
  <Style TargetType="Button">
    <Setter Property="FontFamily" Value="Times New Roman" />
    <Setter Property="FontSize" Value="18" />
    <Setter Property="FontWeight" Value="Bold" />
  </Style>
</Window.Resources>

<StackPanel Margin="5">
  <Button Padding="5" Margin="5">Customized Button</Button>
  <TextBlock Margin="5">Normal Content.</TextBlock>
  <Button Padding="5" Margin="5" Style="{x:Null}">A Normal Button</Button>
  <TextBlock Margin="5">More normal Content.</TextBlock>
  <Button Padding="5" Margin="5">Another Customized Button</Button>
</StackPanel>
```

In this example, the middle button explicitly replaces the style. But rather than supply a new style of its own, this button sets the Style property to a null value, which effectively removes the style.

Although automatic styles are convenient, they can complicate your design. Here are a few reasons why:

- In a complex window with many styles and multiple layers of styles, it becomes difficult to track down whether a given property is set through property value inheritance or a style (and if it's a style, which one). As a result, if you want to change a simple detail, you may need to wade through the markup of your entire window.
- The formatting in a window often starts out more general and becomes increasingly fine-tuned. If you apply automatic styles to the window early on, you'll probably need to override the styles in many places with explicit styles. This complicates the overall design. It's much more straightforward to create named styles for every combination of formatting characteristics you want and apply them by name.
- For example, if you create an automatic style for the TextBlock element, you'll wind up modifying other controls that use the TextBlock (such as a template-driven ListBox control).

To avoid problems, it's best to apply automatic styles judiciously. If you do decide to give your entire user interface a single, consistent look using automatic styles, try to limit your use of explicit styles to special cases.



www.it-ebooks.info

# Triggers

One of the themes in WPF is extending what you can do *declaratively*. Whether you're using styles, resources, or data binding, you'll find that you can do quite a bit without resorting to code.

Triggers are another example of this trend. Using triggers, you can automate simple style changes that would ordinarily require boilerplate event handling logic. For example, you can react when a property is changed and adjust a style automatically.

Triggers are linked to styles through the Style.Triggers collection. Every style can have an unlimited number of triggers, and each trigger is an instance of a class that derives from System.Windows.TriggerBase. WPF gives you the choices listed in Table 11-2.

You can apply triggers directly to elements, without needing to create a style, by using the FrameworkElement.Triggers collection. However, there's a sizable catch. This Triggers collection supports event triggers only. (There's no technical reason for this limitation; it's simply a feature the WPF team didn't have time to implement and may include in future versions.)

***Table 11-2.*** *Classes That Derive from TriggerBase*

| Name | Description |
|---|---|
| Trigger | This is the simplest form of trigger. It watches for a change in a dependency property and then uses a setter to change the style. |
| MultiTrigger | This is similar to Trigger but combines multiple conditions. All the conditions must be met before the trigger springs into action. |
| DataTrigger | This trigger works with data binding. It's similar to Trigger, except it watches for a change in any bound data. |
| MultiDataTrigger | This combines multiple data triggers. |
| EventTrigger | This is the most sophisticated trigger. It applies an animation when an event occurs. |

## A Simple Trigger

You can attach a simple trigger to any dependency property. For example, you can create mouseover and focus effects by responding to changes in the IsFocused, IsMouseOver, and IsPressed properties of the Control class.

Every simple trigger identifies the property you're watching and the value that you're waiting for. When this value occurs, the setters you've stored in the Trigger.Setters collection are applied. (Unfortunately, it isn't possible to use more sophisticated trigger logic that compares a value to see how it falls in a range, performs a calculation, and so on. In these situations, you're better off using an event handler.)

Here's a trigger that waits for a button to get the keyboard focus, at which point it's given a dark red background:

```
<Style x:Key="BigFontButton">
  <Style.Setters>
    <Setter Property="Control.FontFamily" Value="Times New Roman" />
    <Setter Property="Control.FontSize" Value="18" />
  </Style.Setters>

  <Style.Triggers>
    <Trigger Property="Control.IsFocused" Value="True">
```



www.it-ebooks.info

```
      <Setter Property="Control.Foreground" Value="DarkRed" />
    </Trigger>
  </Style.Triggers>
</Style>
```

The nice thing about triggers is that there's no need to write any logic to reverse them. As soon as the trigger stops applying, your element reverts to its normal appearance. In this example, that means the button gets its ordinary gray background as soon as the user tabs away.

---

■ **Note** To understand how this works, you need to remember the dependency property system that you learned about in Chapter 4. Essentially, a trigger is one of the many property providers that can override the value that's returned by a dependency property. However, the original value (whether it is set locally or by a style) still remains. As soon as the trigger becomes deactivated, the pre-trigger value is available again.

---

It's possible to create multiple triggers that may apply to the same element at once. If these triggers set different properties, there's no ambiguity in this situation. However, if you have more than one trigger that modifies the same property, the last trigger in the list wins.

For example, consider the following triggers, which adjust a control depending on whether it is focused, whether the mouse is hovering over it, and whether it's been clicked:

```
<Style x:Key="BigFontButton">
  <Style.Setters>
    ...
  </Style.Setters>
  <Style.Triggers>
    <Trigger Property="Control.IsFocused" Value="True">
      <Setter Property="Control.Foreground" Value="DarkRed" />
    </Trigger>
    <Trigger Property="Control.IsMouseOver" Value="True">
      <Setter Property="Control.Foreground" Value="LightYellow" />
      <Setter Property="Control.FontWeight" Value="Bold" />
    </Trigger>
    <Trigger Property="Button.IsPressed" Value="True">
      <Setter Property="Control.Foreground" Value="Red" />
    </Trigger>
  </Style.Triggers>
</Style>
```

Obviously, it's possible to hover over a button that currently has the focus. This doesn't pose a problem because these triggers modify different properties. But if you click the button, there are two different triggers attempting to set the foreground. Now the trigger for the Button.IsPressed property wins because it's last in the list. It doesn't matter which trigger *occurs* first—for example, WPF doesn't care that a button gets focus before you click it. The order in which the triggers are listed in your markup is all that matters.

---

■ **Note** In this example, triggers aren't all you need to get a nice-looking button. You're also limited by the button's control template, which locks down certain aspects of its appearance. For best results when customizing elements to this degree, you need to use a control template. However, control templates don't replace triggers—in fact, control



www.it-ebooks.info

templates often use triggers to get the best of both worlds: controls that can be completely customized and react to mouseovers, clicks, and other events to change some aspect of their visual appearance.

---

If you want to create a trigger that switches on only if several criteria are true, you can use a MultiTrigger. It provides a Conditions collection that lets you define a series of property and value combinations. Here's an example that applies formatting only if a button has focus and the mouse is over it:

```
<Style x:Key="BigFontButton">
  <Style.Setters>
    ...
  </Style.Setters>
  <Style.Triggers>
    <MultiTrigger>
      <MultiTrigger.Conditions>
        <Condition Property="Control.IsFocused" Value="True">
        <Condition Property="Control.IsMouseOver" Value="True">
      </MultiTrigger.Conditions>
      <MultiTrigger.Setters>
        <Setter Property="Control.Foreground" Value="DarkRed" />
      </MultiTrigger.Setters>
    </MultiTrigger>
  </Style.Triggers>
</Style>
```

In this case, it doesn't matter what order you declare the conditions in because they must all hold true before the background is changed.

## An Event Trigger

While an ordinary trigger waits for a property change to occur, an event trigger waits for a specific event to be fired. You might assume that at this point you use setters to change the element, but that's not the case. Instead, an event trigger requires that you supply a series of actions that modify the control. These actions are used to apply an animation.

Although you won't consider animations in detail until Chapter 15, you can get the idea with a basic example. The following event trigger waits for the MouseEnter event and then animates the FontSize property of the button, enlarging it to 22 units for 0.2 seconds:

```
<Style x:Key="BigFontButtonStyle">
  <Style.Setters>
    ...
  </Style.Setters>

  <Style.Triggers>
    <EventTrigger RoutedEvent="Mouse.MouseEnter">
      <EventTrigger.Actions>
        <BeginStoryboard>
          <Storyboard>
            <DoubleAnimation
              Duration="0:0:0.2"
```



www.it-ebooks.info

```
            Storyboard.TargetProperty="FontSize"
            To="22"  />
        </Storyboard>
      </BeginStoryboard>
    </EventTrigger.Actions>
  </EventTrigger>
  ...
```

In XAML, every animation must be defined in a storyboard, which provides the timeline for the animation. Inside the storyboard, you define the animation object (or objects) that you want to use. Every animation object performs essentially the same task: it modifies a dependency property over some time period.

In this example, a prebuilt animation class named DoubleAnimation is being used (which is found in the System.Windows.Media.Animation namespace, like all animation classes). DoubleAnimation can gradually change any double value (such as FontSize) to a set target over a given period of time. Because the double value is changed in small fractional units, you'll see the font grow gradually. The actual size of the change depends on the total amount of time and the total change you need to make. In this example, the font changes from its current set value to 22 units, over a time period of 0.2 seconds. (You can fine-tune details such as these and create an animation that accelerates or decelerates by tweaking the properties of the DoubleAnimation class.)

Unlike property triggers, you need to reverse event triggers if you want the element to return to its original state. (That's because the default animation behavior is to remain active once the animation is complete, holding the property at the final value. You'll learn more about how this system works in Chapter 15.)

To reverse the font size in this example, the style uses an event trigger that reacts to the MouseLeave event and shrinks the font back to its original size over a full two seconds. You don't need to indicate the target font size in this case—if you don't, WPF assumes you want the original font size that the button had before the first animation kicked in:

```
  ...
  <EventTrigger RoutedEvent="Mouse.MouseLeave">
    <EventTrigger.Actions>
      <BeginStoryboard>
        <Storyboard>
          <DoubleAnimation
            Duration="0:0:1"
            Storyboard.TargetProperty="FontSize"  />
        </Storyboard>
      </BeginStoryboard>
    </EventTrigger.Actions>
  </EventTrigger>
</Style.Triggers>
</Style>
```

Interestingly, you can also perform an animation when a dependency property hits a specific value. This is useful if you want to perform an animation and there isn't a suitable event to use.

To use this technique you need a property trigger, as described in the previous section. The trick is to not supply any Setter objects for your property trigger. Instead, you set the Trigger.EnterActions and Trigger.ExitActions properties. Both properties take a collection of actions, such as the BeginStoryboard action that starts an animation. The EnterActions are performed when the property reaches the designated value, and ExitActions are performed when the property changes away from the designated value.



www.it-ebooks.info

You'll learn much more about using event triggers and property triggers to launch animations in Chapter 15.

# Behaviors

Styles give you a practical way to reuse groups of property settings. They're a great first step that can help you build consistent, well-organized interfaces—but they're also broadly limited.

The problem is that property settings are only a small part of the user-interface infrastructure in a typical application. Even the most basic program usually needs reams of user-interface code that has nothing to do with the application's functionality. In many programs, the code that's used for UI tasks (such as driving animations, powering slick effects, maintaining user-interface state, and supporting user-interface features such as dragging, zooming, and docking) far outweighs the business code in both size and complexity. Much of this code is generic, meaning you end up writing the same thing in every WPF project you create. Almost all of it is tedious.

In response to this challenge, the creators of Expression Blend have developed a feature called *behaviors*. The idea is simple: you (or another developer) create a behavior that encapsulates a common bit of user-interface functionality. This functionality can be basic (such as starting a storyboard or navigating to a hyperlink). Or, it can be complex (such as handling multitouch interactions or modeling a collision with a real-time physics engine). Once built, you can add this functionality to another control in any application by hooking that control to the right behavior and setting the behavior's properties. In Expression Blend, using a behavior takes little more than a drag-and-drop operation.

---

■ **Note** Custom controls are another technique for reusing user-interface functionality in an application (or among multiple applications). However, a custom control must be developed as a tightly linked package of visuals and code. Although custom controls are extremely powerful, they don't address situations where you need to equip many different controls with similar functionality (for example, adding a mouseover rendering effect to a group of different elements). For that reason, styles, behaviors, and custom controls are all complementary.

---

## Getting Support for Behaviors

There's one catch. The infrastructure for reusing common blocks of user-interface code isn't part of WPF. Instead, it's bundled with Expression Blend. This is because behaviors began as a design-time feature for Expression Blend. In fact, Expression Blend is still the only tool that lets you add behaviors by dragging them onto the controls that need them. But that doesn't mean behaviors are useful only in Expression Blend. You can create and use them in a Visual Studio application with only slightly more effort. You simply need to write the markup by hand rather than using the Toolbox.

To get the assemblies that support for behaviors, you have two options:

- You can install Expression Blend 3, Expression Blend 4, or Expression Blend for Visual Studio 2012 (which is currently available only as a preview, at `http://tinyurl.com/c5u84uc`). All these versions include the assemblies you need for the behavior feature in Visual Studio, but Expression Blend for Visual Studio 2012 is the only version that allows you to create and edit WPF 4.5 applications in the Blend environment.



www.it-ebooks.info

- You can install the Expression Blend 3 SDK (which is available at `http://tinyurl.com/kkp4g8`).

---

■ **Note** Microsoft's naming system for Expression Blend is hugely confusing. Many versions of Visual Studio 2012 include a limited version of Expression Blend, which is called Expression Blend for Visual Studio 2012. This limited version only allows the creation of Metro-style applications, and it only works on Windows 8. However, the full version of Blend has the same name (Expression Blend for Visual Studio 2012), even though it supports Silverlight and WPF. At the time of this writing, the full version is available as a free preview only (http://tinyurl.com/c5u84uc).

---

Whether you use an old version of Expression Blend, the new preview, or the SDK, you'll find the same two important assemblies that you need, in a folder like c:\Program Files (x86)\Microsoft SDKs\Expression\Blend 3\Interactivity\Libraries\WPF. They are

- *System.Windows.Interactivity.dll:* This assembly defines the base classes that support behaviors. It's the cornerstone of the behavior feature.
- *Microsoft.Expression.Interactions.dll***:** This assembly adds some useful extensions, with optional action and trigger classes that are based on the core behavior classes.

## Understanding the Behavior Model

The behavior feature comes in two versions (both of which are included with Expression Blend or the Expression Blend SDK). One version is designed to add behavior support to Silverlight, Microsoft's rich client plug-in for the browser, while the other is designed for WPF. Although both offer identical features, the behavior feature meshes more neatly into the Silverlight world, because it fills a bigger gap. Unlike WPF, Silverlight has no support for triggers, so it makes sense that the assemblies that implement behaviors also implement triggers. However, WPF *does* support triggers, which makes it more than a little confusing to find that the behavior feature includes its own trigger system, which doesn't match the WPF model.

The problem is that these two similarly named features overlap partly but not completely. In WPF, the most important role for triggers is building flexible styles and control templates (as you'll see in Chapter 17). With the help of triggers, your styles and template become more intelligent; for example, you can apply a visual effect when some property changes. However, the trigger system in Expression Blend has a different purpose. It allows you to add simple functionality to an application using a visual design tool. In other words, WPF triggers support more powerful styles and control templates. Expression Blend triggers support quick code-free application design.

So, what does all this mean for the average WPF developer? Here are a few guidelines:

- The behavior model is not a core part of WPF, so it's not as established as styles and templates. In other words, you can program WPF applications without using behaviors, but you won't be able to create much more than a Hello World demonstration without styles and templates.
- The trigger feature in Expression Blend may interest you if you spend a lot of time in Expression Blend or you want to develop components for other Expression Blend users. Even though it shares a name with the trigger system in WPF, there's no overlap, and you can use both.
- If you don't work with Expression Blend, you can skip the trigger feature altogether—but you should still look at Expression Blend's full-fledged behavior



www.it-ebooks.info

classes. That's because behaviors are both more powerful and more common than Expression Blend triggers. Eventually, you're bound to find a third-party component that includes a nice, neat behavior for you to use in your own applications. (For example, in Chapter 5 you explored multitouch and learned about a free behavior that you can use to give your elements automatic behavior support.)

In this chapter, you won't look at the Expression Blend trigger system, but you will consider the full-fledged behavior classes. To learn more about Expression Blend triggers and see additional behavior examples (some of which are intended for Silverlight rather than WPF), you can read through the posts at `http://tinyurl.com/yfvakl3`. The downloadable code for this chapter also includes two custom trigger examples.

## Creating a Behavior

Behaviors aim to encapsulate bits of UI functionality so you can apply them to elements without writing the code yourself. Another way of looking at is that every behavior provides a service to an element. This service usually involves listening to several different events and performing several related operations. For example, an example at `http://tinyurl.com/9kwdnsc` provides a watermark behavior for text boxes. If the text box is empty and doesn't currently have focus, a prompt message (like "[ Enter text here ]") is shown in light lettering. When the text box gets focus, the behavior springs into action and removes the watermark text.

The best way to gain a better understanding of behaviors is to create one of your own. Imagine that you want to give any element the ability to be dragged around a Canvas with the mouse. The basic steps for a single element are easy enough—your code listens for mouse events and changes the attached properties that set the Canvas coordinates appropriately. But with a bit more effort, you can turn that code into a reusable behavior that can give dragging support to any element on any Canvas.

Before you go any further, create a WPF class library assembly. (In this example, it's called CustomBehaviorsLibrary.) In it, add a reference to the System.Windows.Interactivity.dll assembly. Then, create a class that derives from the base Behavior class. Behavior is a generic class that takes a type argument. You can use this type argument to restrict your behavior to specific elements, or you can use UIElement or FrameworkElement to include them all, as shown here:

```
public class DragInCanvasBehavior : Behavior<UIElement>
{ ... }
```

---

■ **Note** Ideally, you won't need to create a behavior yourself. Instead, you'll use a ready-made behavior that someone else has created.

---

The first step in any behavior is to override the OnAttached() and OnDetaching() methods. When OnAttached() is called, you can access the element where the behavior is placed (through the AssociatedObject property) and attach event handlers. When OnDetaching() is called, you remove your event handlers.

Here's the code that the DragInCanvasBehavior uses to monitor the MouseLeftButtonDown, MouseMove, and MouseLeftButtonUp events:

```
protected override void OnAttached()
{
    base.OnAttached();
```



www.it-ebooks.info

```
    // Hook up event handlers.
    this.AssociatedObject.MouseLeftButtonDown +=
      AssociatedObject_MouseLeftButtonDown;
    this.AssociatedObject.MouseMove += AssociatedObject_MouseMove;
    this.AssociatedObject.MouseLeftButtonUp += AssociatedObject_MouseLeftButtonUp;
}

protected override void OnDetaching()
{
    base.OnDetaching();

    // Detach event handlers.
    this.AssociatedObject.MouseLeftButtonDown -=
      AssociatedObject_MouseLeftButtonDown;
    this.AssociatedObject.MouseMove -= AssociatedObject_MouseMove;
    this.AssociatedObject.MouseLeftButtonUp -= AssociatedObject_MouseLeftButtonUp;
}
```

The final step is to run the appropriate code in the event handlers. For example, when the user clicks the left mouse button, DragInCanvasBehavior starts a dragging operation, records the offset between the upper-left corner of the element and the mouse pointer, and captures the mouse:

```
// Keep track of the Canvas where this element is placed.
private Canvas canvas;

// Keep track of when the element is being dragged.
private bool isDragging = false;

// When the element is clicked, record the exact position
// where the click is made.
private Point mouseOffset;

private void AssociatedObject_MouseLeftButtonDown(object sender,
  MouseButtonEventArgs e)
{
    // Find the Canvas.
    if (canvas == null)
      canvas = (Canvas)VisualTreeHelper.GetParent(this.AssociatedObject);

    // Dragging mode begins.
    isDragging = true;

    // Get the position of the click relative to the element
    // (so the top-left corner of the element is (0,0).
    mouseOffset = e.GetPosition(AssociatedObject);

    // Capture the mouse. This way you'll keep receiving
    // the MouseMove event even if the user jerks the mouse
    // off the element.
    AssociatedObject.CaptureMouse();
}
```



www.it-ebooks.info

When the element is in dragging mode and the mouse moves, the element is repositioned:

```
private void AssociatedObject_MouseMove(object sender, MouseEventArgs e)
{
    if (isDragging)
    {
        // Get the position of the element relative to the Canvas.
        Point point = e.GetPosition(canvas);

        // Move the element.
        AssociatedObject.SetValue(Canvas.TopProperty, point.Y - mouseOffset.Y);
        AssociatedObject.SetValue(Canvas.LeftProperty, point.X - mouseOffset.X);
    }
}
```

And when the mouse button is released, dragging ends:

```
private void AssociatedObject_MouseLeftButtonUp(object sender,
  MouseButtonEventArgs e)
{
    if (isDragging)
    {
        AssociatedObject.ReleaseMouseCapture();
        isDragging = false;
    }
}
```

## Using a Behavior

To test your behavior, being by creating a new WPF Application project. Then, add a reference to the class library that defines the DragInCanvasBehavior class (which you created in the previous section) and the System.Windows.Interactivity.dll assembly. Next, map both namespaces in your XML. Assuming the DragInCanvasBehavior class is stored in a class library named CustomBehaviorsLibrary, you'll need markup like this:

```
<Window xmlns:i=
"clr-namespace:System.Windows.Interactivity;assembly=System.Windows.Interactivity"
 xmlns:custom=
"clr-namespace:CustomBehaviorsLibrary;assembly=CustomBehaviorsLibrary" ... >
```

To use this behavior, you simply need to add to any element inside a Canvas using the Interaction.Behaviors attached property. The following markup creates a Canvas with three shapes. The two Ellipse elements use the DragInCanvasBehavior and can be dragged around the Canvas. The Rectangle element does not and so cannot be moved.

```
<Canvas>
  <Rectangle Canvas.Left="10" Canvas.Top="10" Fill="Yellow" Width="40" Height="60">
  </Rectangle>

  <Ellipse Canvas.Left="10" Canvas.Top="70" Fill="Blue" Width="80" Height="60">
    <i:Interaction.Behaviors>
      <custom:DragInCanvasBehavior></custom:DragInCanvasBehavior>
```



www.it-ebooks.info

```
    </i:Interaction.Behaviors>
  </Ellipse>

  <Ellipse Canvas.Left="80" Canvas.Top="70" Fill="OrangeRed" Width="40" Height="70">
    <i:Interaction.Behaviors>
      <custom:DragInCanvasBehavior></custom:DragInCanvasBehavior>
    </i:Interaction.Behaviors>
  </Ellipse>
</Canvas>
```

Figure 11-4 shows this example in action.

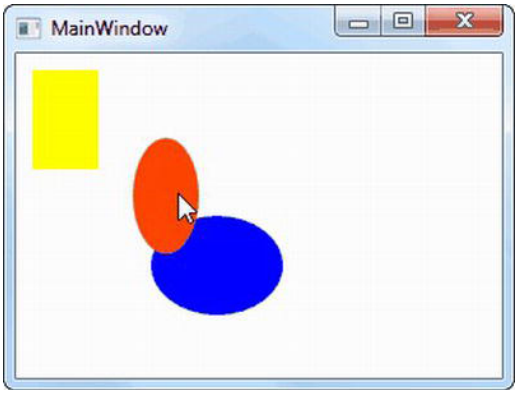


***Figure 11-4.*** *Making elements draggable with a behavior*

But this isn't the whole story. If you're developing in Expression Blend, behaviors give you an even better design experience—one that can save you from writing any markup at all.

## Design-Time Behavior Support in Blend

In Expression Blend, working with behaviors is a drag-and-drop-and-configure operation. First, you need to make sure your application has a reference to the assembly that has the behaviors you want to use. (In this case, that's the class library assembly where DragInCanvasBehavior is defined.) Next, you need to ensure that it also has a reference to the System.Windows.Interactivity.dll assembly.

Expression Blend automatically searches all referenced assemblies for behaviors and displays them in the Asset Library (the same panel you use for choosing elements when designing a Silverlight page). It also adds the behaviors from the Microsoft.Expression.Interactions.dll assembly, even if they aren't yet referenced by your project.

To see the behaviors you have to choose from, start by drawing a button on the design surface of your page, click the Asset Library button, and click the Behaviors tab (see Figure 11-5).



www.it-ebooks.info

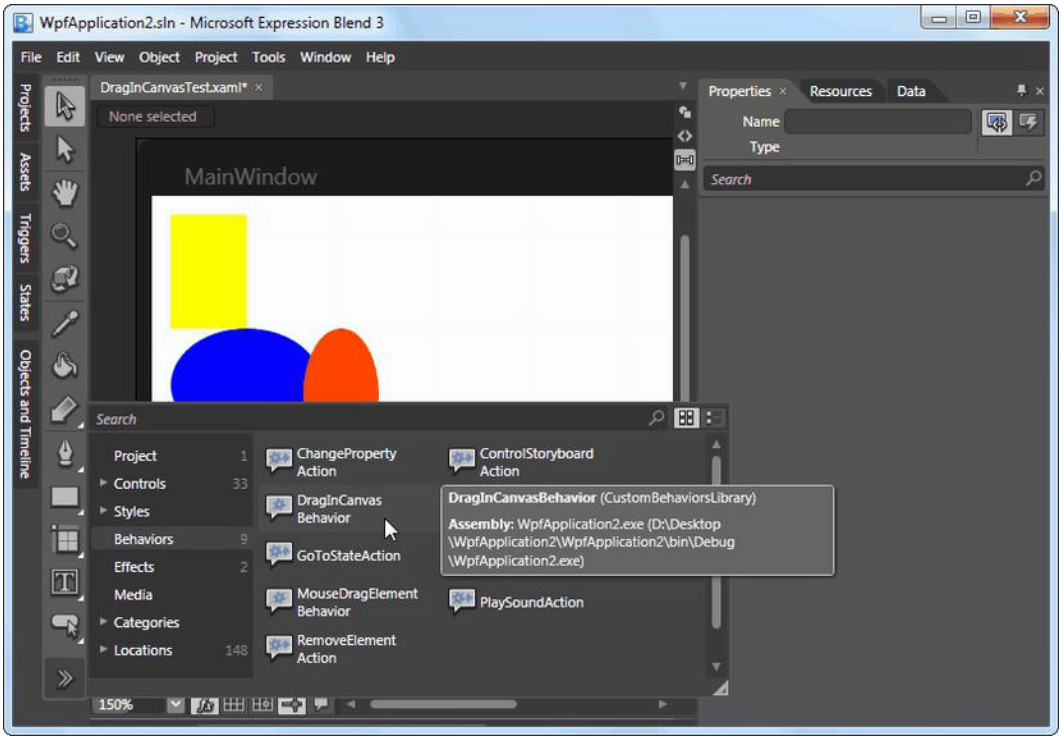


***Figure 11-5.*** *Actions in the Asset Library*

To add an action to a control, drag it from the Asset Library, and drop it onto your control (in this case, one of the shapes in the Canvas). When you take this step, Expression Blend automatically creates the behavior, which you can then configure (if it has any properties).

# The Last Word

In this chapter, you saw how to use styles to reuse formatting settings with elements. You also considered how to use behaviors to develop tidy packages of user interface functionality, which can then be wired up to any element. Both tools give you a way to make more intelligent, maintainable user interfaces—ones that centralize formatting details and complex logic rather than forcing you to distribute it throughout your application and repeat it many times over.



www.it-ebooks.info