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 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 *styles*.

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). 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:

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

```xml
<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:

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