```
{
    this.RemoveHandler(CommandManager.PreviewExecutedEvent,
      new ExecutedRoutedEventHandler(CommandExecuted));
}
```

When the PreviewExecuted event fires, you need to determine whether it's a command that deserves your attention. If so, you can create the CommandHistoryItem and add it to the Undo stack. You also need to watch out for two potential problems. First, when you click a toolbar button to perform a command on the text box, the CommandExecuted event is raised twice: once for the toolbar button and once for the text box. This code avoids duplicate entries in the Undo history by ignoring the command if the sender is ICommandSource. Second, you need to explicitly ignore the commands you don't want to add to the Undo history. One example is the ApplicationUndo command, which allows you to reverse the previous action.

```
private void CommandExecuted(object sender, ExecutedRoutedEventArgs e)
{
    // Ignore menu button source.
    if (e.Source is ICommandSource) return;

    // Ignore the ApplicationUndo command.
    if (e.Command == MonitorCommands.ApplicationUndo) return;

    TextBox txt = e.Source as TextBox;
    if (txt != null)
    {
        RoutedCommand cmd = (RoutedCommand)e.Command;
        CommandHistoryItem historyItem = new CommandHistoryItem(
          cmd.Name, txt, "Text", txt.Text);

        ListBoxItem item = new ListBoxItem();
        item.Content = historyItem;
        lstHistory.Items.Add(historyItem);
    }
}
```

This example stores all CommandHistoryItem objects in a ListBox. The ListBox has DisplayMember set to Name so that it shows the CommandHistoryItem.Name property of each item. This code supports the Undo feature only if the command is being fired for a text box. However, it's generic enough to work with any text box in the window. You could extend this code to support other controls and properties.

The last detail is the code that performs the application-wide Undo. Using a CanExecute handler, you can make sure that this code is executed only when there is at least one item in the Undo history:

```
private void ApplicationUndoCommand_CanExecute(object sender,
  CanExecuteRoutedEventArgs e)
{
    if (lstHistory == null || lstHistory.Items.Count == 0)
      e.CanExecute = false;
    else
      e.CanExecute = true;
}
```

To revert the last change, you simply call the Undo() method of the CommandHistoryItem and then remove it from the list:

```
private void ApplicationUndoCommand_Executed(object sender, RoutedEventArgs e)
{
    CommandHistoryItem historyItem = (CommandHistoryItem)
      lstHistory.Items[lstHistory.Items.Count - 1];

    if (historyItem.CanUndo) historyItem.Undo();
    lstHistory.Items.Remove(historyItem);
}
```

Although this example demonstrates the concept and presents a simple application with multiple controls that fully support the Undo feature, you would need to make many refinements before you would use an approach like this in a real-world application. For example, you would need to spend considerable time refining the event handler for the CommandManager.PreviewExecuted event to ignore commands that clearly shouldn't be tracked. (Currently, events such as selecting text with the keyboard and hitting the spacebar raise commands.) Similarly, you probably would want to add CommandHistoryItem objects for actions that should be reversible but aren't represented by commands, such as typing a bunch of text and then navigating to another control. Finally, you probably would want to limit the Undo history to just the most recent commands.

The Last Word

In this chapter, you explored the WPF command model. You learned how to hook controls to commands, respond when the commands are triggered, and handle commands differently based on where they occur. You also designed your own custom commands and learned how to extend the WPF command system with a basic command history and Undo feature.

Overall, the WPF command model isn't quite as streamlined as other bits of WPF architecture. The way that it plugs into the routed event model requires a fairly complex assortment of classes, and the inner workings aren't extensible. However, the command model is still a great stride forward over Windows Forms, which lacked any sort of command feature.

CHAPTER 10

■ ■ ■

Resources

WPF's resource system is simply a way of keeping around a set of useful objects, such as commonly used brushes, styles, or templates, so you can reuse them more easily.

Although you can create and manipulate resources in code, you'll usually define them in your XAML markup. Once a resource is defined, you can use it throughout the rest of the markup in your window (or, in the case of an application resource, throughout the rest of your application). This technique simplifies your markup, saves repetitive coding, and allows you to store user interface details (such as your application's color scheme) in a central place so they can be modified easily. Object resources are also the basis for reusing WPF styles, as you'll see in the next chapter.

■ **Note** Don't confuse WPF's *object* resources with the *assembly* resources you learned about in Chapter 7. An assembly resource is a chunk of binary data that's embedded in your compiled assembly. You can use an assembly resource to make sure your application has an image or sound file it needs. An object resource, on the other hand, is a .NET object that you want to define in one place and use in several others.

Resource Basics

WPF allows you to define resources in code or in a variety of places in your markup (along with specific controls, in specific windows, or across the entire application).

Resources have a number of important benefits:

- **Efficiency.** Resources let you define an object once and use it in several places in your markup. This streamlines your code and makes it marginally more efficient.
- **Maintainability.** Resources let you take low-level formatting details (such as font sizes) and move them to a central place where they're easy to change. It's the XAML equivalent of creating constants in your code.
- **Adaptability.** Once certain information is separated from the rest of your application and placed in a resource section, it becomes possible to modify it dynamically. For example, you may want to change resource details based on user preferences or the current language.

The Resources Collection

Every element includes a Resources property, which stores a dictionary collection of resources. (It's an instance of the ResourceDictionary class.) The resources collection can hold any type of object, indexed by string.

Although every element includes the Resources property (which is defined as part of the FrameworkElement class), the most common way to define resources is at the window level. That's because every element has access to the resources in its own resource collection and the resources in all of its parents' resource collections.

For example, consider the window with three buttons shown in Figure 10-1. Two of the three buttons use the same brush—an image brush that paints a tile pattern of happy faces.

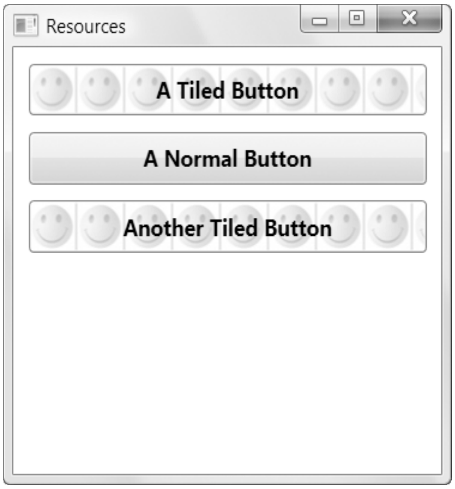

Figure 10-1. *A window that reuses a brush*

In this case, it's clear that you want both the top and bottom buttons to have the same styling. However, you might want to change the characteristics of the image brush later. For that reason, it makes sense to define the image brush in the resources for the window and reuse it as necessary.

Here's how you define the brush:

```
<Window.Resources>
  <ImageBrush x:Key="TileBrush" TileMode="Tile"
    ViewportUnits="Absolute" Viewport="0 0 32 32"
    ImageSource="happyface.jpg" Opacity="0.3">
  </ImageBrush>
</Window.Resources>
```

The details of the image brush aren't terribly important (although Chapter 12 has the specifics). What *is* important is the first attribute, named Key (and preceded by the x: namespace prefix, which puts it in the XAML namespace rather than the WPF namespace). This assigns the name under which the brush will be indexed in the Window.Resources collection. You can use whatever you want, so long as you use the same name when you need to retrieve the resource.

■ **Note** You can instantiate any .NET class in the resources section (including your own custom classes), as long as it's XAML-friendly. That means it needs to have a few basic characteristics, such as a public zero-argument constructor and writeable properties.

To use a resource in your XAML markup, you need a way to refer to it. This is accomplished using a markup extension. In fact, there are two markup extensions that you can use: one for dynamic resources and one for static resources. Static resources are set once, when the window is first created. Dynamic resources are reapplied if the resource is changed. (You'll study the difference more closely a little bit later in this chapter.) In this example, the image brush never changes, so the static resource is fine.

Here's one of the buttons that uses the resource:

```
<Button Background="{StaticResource TileBrush}"
  Margin="5" Padding="5" FontWeight="Bold" FontSize="14">
  A Tiled Button
</Button>
```

In this case, the resource is retrieved and used to assign the Button.Background property. You could perform the same feat (with slightly more overhead) by using a dynamic resource:

```
<Button Background="{DynamicResource TileBrush}"
```

Using a simple .NET object for a resource really is this easy. However, there are a few finer points you need to consider. The following sections will fill you in.

The Hierarchy of Resources

Every element has its own resource collection, and WPF performs a recursive search up your element tree to find the resource you want. In the current example, you could move the image brush from the Resources collection of the window to the Resources collection of the StackPanel that holds all three buttons without changing the way the application works. You could also put the image brush in Button.Resources collection, but then you'd need to define it twice—once for each button.

There's another issue to consider. When using a static resource, you must always define a resource in your markup *before* you refer to it. That means that even though it's perfectly valid (from a markup perspective) to put the Windows.Resources section after the main content of the window (the StackPanel that contains all the buttons), this change will break the current example. When the XAML parser encounters a static reference to a resource it doesn't know, it throws an exception. (You can get around this problem using a dynamic resource, but there's no good reason to incur the extra overhead.)

As a result, if you want to place your resource in the button element, you need to rearrange your markup a little so that the resource is defined before the background is set. Here's one way to do it:

```
<Button Margin="5" Padding="5" FontWeight="Bold" FontSize="14">
  <Button.Resources>
    <ImageBrush x:Key="TileBrush" TileMode="Tile"
      ViewportUnits="Absolute" Viewport="0 0 10 10"
      ImageSource="happyface.jpg" Opacity="0.3"></ImageBrush>
  </Button.Resources>
  <Button.Background>
    <StaticResource ResourceKey="TileBrush"/>
  </Button.Background>

  <Button.Content>Another Tiled Button</Button.Content>
</Button>
```

The syntax for the static resource markup extension looks a bit different in this example because it's set in a nested element (not an attribute). The resource key is specified using the ResourceKey property to point to the right resource.

Interestingly, you can reuse resource names as long as you don't use the same resource name more than once in the same collection. That means you could create a window like this, which defines the image brush in two places:

```
<Window x:Class="Resources.TwoResources"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Resources" Height="300" Width="300" >

  <Window.Resources>
    <ImageBrush x:Key="TileBrush" TileMode="Tile"
                ViewportUnits="Absolute" Viewport="0 0 32 32"
                ImageSource="happyface.jpg" Opacity="0.3"></ImageBrush>
  </Window.Resources>

  <StackPanel Margin="5">
    <Button Background="{StaticResource TileBrush}" Padding="5"
      FontWeight="Bold" FontSize="14" Margin="5" >A Tiled Button</Button>

    <Button Padding="5" Margin="5"
      FontWeight="Bold" FontSize="14">A Normal Button</Button>
    <Button Background="{DynamicResource TileBrush}" Padding="5" Margin="5"
      FontWeight="Bold" FontSize="14">
      <Button.Resources>
        <ImageBrush x:Key="TileBrush" TileMode="Tile"
          ViewportUnits="Absolute" Viewport="0 0 32 32"
          ImageSource="sadface.jpg" Opacity="0.3"></ImageBrush>
      </Button.Resources>
      <Button.Content>Another Tiled Button</Button.Content>
    </Button>

  </StackPanel>
</Window>
```

In this case, the button uses the resource it finds first. Because it begins by searching its own Resources collection, the second button uses the sadface.jpg graphic, while the first button gets the brush from the containing window and uses the happyface.jpg image.

Static and Dynamic Resources

You might assume that because the previous example used a static resource it's immune to any changes you make to your resource (in this case, the image brush). However, that's actually not the case.

For example, imagine you execute this code at some point after the resource has been applied and the window has been displayed:

```
ImageBrush brush = (ImageBrush)this.Resources["TileBrush"];
brush.Viewport = new Rect(0, 0, 5, 5);
```

This code retrieves the brush from the Window.Resources collection and manipulates it. (Technically, the code changes the size of each tile, shrinking the happy face and packing the image pattern more tightly.) When you run this code, you probably don't expect any reaction in your user interface—after all, it's a static resource. However, this change does propagate to the two buttons. In fact, the buttons are updated with the new Viewport property setting, *regardless* of whether they use the brush through a static resource or a dynamic resource.

The reason this works is because the Brush class derives from a class named Freezable. The Freezable class has basic-change tracking features (and it can be "frozen" to a read-only state if it doesn't need to change). What that means is whenever you change a brush in WPF, any controls that use that brush refresh themselves automatically. It doesn't matter whether they get their brushes through a resource.

At this point, you're probably wondering what the difference is between static and dynamic resource. The difference is that a static resource grabs the object from the resources collection once. Depending on the type of object (and the way it's used), any changes you make to that object may be noticed right away. However, the dynamic resource looks the object up in the resources collection every time it's needed. That means you could place an entirely new object under the same key, and the dynamic resource would pick up your change.

To see an example that illustrates the difference, consider the following code, which replaces the current image brush with a completely new (and boring) solid blue brush:

```
this.Resources["TileBrush"] = new SolidColorBrush(Colors.LightBlue);
```

A dynamic resource picks up this change, while a static resource has no idea that its brush has been replaced in the Resources collection by something else. It continues using the original ImageBrush instead.

Figure 10-2 shows this example in a window that includes a dynamic resource (the top button) and a static resource (the bottom button).

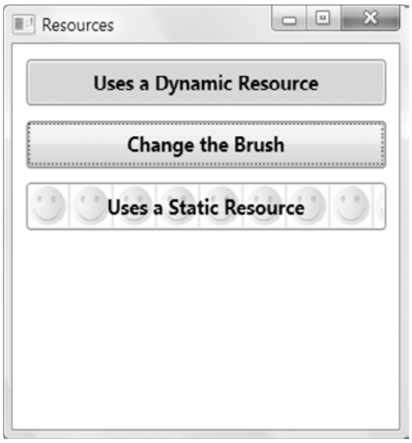

Figure 10-2. Dynamic and static resources

Usually, you don't need the overhead of a dynamic resource, and your application will work perfectly well with a static resource. One notable exception is if you're creating resources that depend on Windows settings (such as system colors). In this situation, you need to use dynamic resources if you want to be able to react to any change in the current color scheme. (Or if you use static resources, you'll keep using the old color scheme until the user restarts the application.) You'll learn more about how this works when you tackle system resources a bit later in this chapter.

As a general guideline, only use dynamic properties when

- Your resource has properties that depend on system settings (such as the current Windows colors or fonts).
- You plan to replace your resource objects programmatically (for example, to implement some sort of dynamic skinning feature, as demonstrated in Chapter 17).

However, you shouldn't get overly ambitious with dynamic resources. The primary issue is that changing a resource doesn't necessarily trigger a refresh in your user interface. (It does in the brush example because of the way brush objects are constructed—namely, they have this notification support built in.) There are a host of occasions where you need to show dynamic content in a control in a way that the control adjusts itself as the content changes, and for that it makes much more sense to use data binding.

■ **Note** On rare occasions, dynamic resources are also used to improve the first-time load performance of a window. That's because static resources are always loaded when the window is created, while dynamic resources are loaded when they're first used. However, you won't see any benefit unless your resource is extremely large and complex (in which case parsing its markup takes a nontrivial amount of time).

Nonshared Resources

Ordinarily, when you use a resource in multiple places, you're using the same object instance. This behavior—called *sharing*—is usually what you want. However, it's also possible to tell the parser to create a separate instance of your object each time it's used.

To turn off sharing, you use the Shared attribute, as shown here:

```
<ImageBrush x:Key="TileBrush" x:Shared="False" ...></ImageBrush>
```

There are few good reasons for using nonshared resources. You might consider nonshared resources if you want to modify your resource instances separately later. For example, you could create a window that has several buttons that use the same brush but turn off sharing so that you can change each brush individually. This approach isn't very common because it's inefficient. In this example, it would be better to let all the buttons use the same brush initially and then create and apply new brush objects as needed. That way you're incurring the overhead of extra brush objects only when you really need to do so.

Another reason you might use nonshared resources is if you want to reuse an object in a way that otherwise wouldn't be allowed. For example, using this technique, you could define an element (such as an Image or a Button) as a resource and then display that element in several different places in a window.

Once again, this usually isn't the best approach. For example, if you want to reuse an Image element, it makes more sense to store the relevant piece of information (such as the BitmapImage object that identifies the image source) and share that between multiple Image elements. And if you simply want to standardize controls so they share the same properties, you're far better off using styles, which are described in the next chapter. Styles give you the ability to create identical or nearly identical copies of any element, but they also allow you to override property values when they don't apply and attach distinct event handlers, two features you'd lose if you simply cloned an element using a nonshared resource.

Accessing Resources in Code

Usually, you'll define and use resources in your markup. However, if the need arises, you can work with the resources collection in code.

As you've already seen, you can pull items out of the resources collection by name. However, to use this approach, you need to use the resource collection of the right element. As you've already seen, this limitation doesn't apply to your markup. A control such as a button can retrieve a resource without specifically knowing where it's defined. When it attempts to assign the brush to its Background property, WPF checks the resources collection of the button for a resource named TileBrush, and then it checks the resources collection of the containing StackPanel and then the containing window. (This process actually continues to look at application and system resources, as you'll see in the next section.)

You can hunt for a resource in the same way using the FrameworkElement.FindResource() method. Here's an example that looks for the resource of a button (or one of its higher-level containers) when a Click event fires:

```
private void cmdChange_Click(object sender, RoutedEventArgs e)
{
    Button cmd = (Button)sender;
    ImageBrush brush = (ImageBrush)sender.FindResource("TileBrush");
    ...
}
```

Instead of FindResource(), you can use the TryFindResource() method that returns a null reference if a resource can't be found, rather than throwing an exception.

Incidentally, you can also add resources programmatically. Pick the element where you want to place the resource, and use the Add() method of the resources collection. However, it's much more common to define resources in markup.

Application Resources

The Window isn't the last stop in the resource search. If you indicate a resource that can't be found in a control or any of its containers (up to the containing window or page), WPF continues to check the set of resources you've defined for your application. In Visual Studio, these are the resources you've defined in the markup for your App.xaml file, as shown here:

```
<Application x:Class="Resources.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    StartupUri="Menu.xaml"
    >
    <Application.Resources>
      <ImageBrush x:Key="TileBrush" TileMode="Tile"
        ViewportUnits="Absolute" Viewport="0 0 32 32"
        ImageSource="happyface.jpg" Opacity="0.3">
      </ImageBrush>
    </Application.Resources>
</Application>
```

As you've probably already guessed, application resources give you a great way to reuse an object across your entire application. In this example, it's a good choice if you plan to use the image brush in more than one window.

■ **Note** Before creating an application resource, consider the trade-off between complexity and reuse. Adding an application resource gives you better reuse, but it adds complexity because it's not immediately clear which windows use a given resource. (It's conceptually the same as an old-style C++ program with too many global variables.) A good guideline is to use application resources if your object is reused widely (for example, in many windows). If it's used in just two or three, consider defining the resource in each window.

It turns out that application resources *still* aren't the final stop when an element searches for a resource. If the resource can't be found in the application resources, the element continues to look at the system resources.

System Resources

As you learned earlier, dynamic resources are primarily intended to help your application respond to changes in system environment settings. However, this raises a question—how do you retrieve the system environment settings and use them in your code in the first place?

The secret is a set of three classes named SystemColors, SystemFonts, and SystemParameters, all of which are in the System.Windows namespace. SystemColors gives you access to color settings; SystemFonts gives you access to fonts settings; and SystemParameters wraps a huge list of settings that describe the standard size of various screen elements, keyboard and mouse settings, and screen size, and whether various graphical effects (such as hot tracking, drop shadows, and showing window contents while dragging) are switched on.

■ **Note** There are two versions of the SystemColors and SystemFonts classes. They're found in the System.Windows namespace and the System.Drawing namespace. Those in the System.Windows namespace are part of WPF. They use the right data types and support the resource system. The ones in the System.Drawing namespace are part of Windows Forms. They aren't useful in a WPF application.

The SystemColors, SystemFonts, and SystemParameters classes expose all their details through static properties. For example, SystemColors.WindowTextColor gets you a Color structure that you can use as you please. Here's an example that uses it to create a brush and fill the foreground of an element:

```
label.Foreground = new SolidBrush(SystemColors.WindowTextColor);
```

Or to be a bit more efficient, you can just use the ready-made brush property:

```
label.Foreground = SystemColors.WindowTextBrush;
```

In WPF, you can access static properties using the static markup extension. For example, here's how you could set the foreground of the same label using XAML:

```
<Label Foreground="{x:Static SystemColors.WindowTextBrush}">
  Ordinary text
</Label>
```

This example doesn't use a resource. It also suffers from a minor failing—when the window is parsed and the label is created, a brush is created based on the current "snapshot" of the window text color. If you change the Windows colors while this application is running (after the window containing the label has been shown), the label won't update itself. Applications that behave this way are considered to be a bit rude.

To solve this problem, you can't set the Foreground property directly to a brush object. Instead, you need to set it to a DynamicResource object that wraps this system resource. Fortunately, all the System*Xxx* classes provide a complementary set of properties that return ResourceKey objects—references that let you pull the resource out of the collection of system resources. These properties have the same name as the ordinary property that returns the object directly, with the word *Key* added to the end. For example, the resource key for the SystemColors.WindowTextBrush is SystemColors.WindowTextBrushKey.

■ **Note** Resource keys aren't simple names—they're references that tell WPF where to look to find a specific resource. The ResourceKey class is opaque, so it doesn't show you the low-level details about how system resources are identified. However, there's no need to worry about your resources conflicting with the system resources because they are in separate assemblies and are treated differently.

Here's how you can use a resource from one of the System*Xxx* classes:

```
<Label Foreground="{DynamicResource {x:Static SystemColors.WindowTextBrushKey}}">
  Ordinary text
</Label>
```

This markup is a bit more complex than the previous example. It begins by defining a dynamic resource. However, the dynamic resource isn't pulled out of the resource collection in your application. Instead, it uses a key that's defined by the SystemColors.WindowTextBrushKey property. Because this property is static, you also need to throw in the static markup extension so that the parser understands what you're trying to do.

Now that you've made this change, you have a label that can update itself seamlessly when system settings change.

Resource Dictionaries

If you want to share resources between multiple projects, you can create a *resource dictionary*. A resource dictionary is simply a XAML document that does nothing but store the resources you want to use.

Creating a Resource Dictionary

Here's an example of a resource dictionary that has one resource:

```
<ResourceDictionary
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">

  <ImageBrush x:Key="TileBrush" TileMode="Tile"
    ViewportUnits="Absolute" Viewport="0 0 32 32"
    ImageSource="happyface.jpg" Opacity="0.3">
  </ImageBrush>
</ResourceDictionary>
```

When you add a resource dictionary to an application, make sure the Build Action is set to Page (as it is for any other XAML file). This ensures that your resource dictionary is compiled to BAML for best performance. However, it's perfectly allowed to have a resource dictionary with a Build Action of Resource, in which case it's embedded in the assembly but not compiled. Parsing it at runtime is then imperceptibly slower.

Using a Resource Dictionary

To use a resource dictionary, you need to merge it into a resource collection somewhere in your application. You could do this in a specific window, but it's more common to merge it into the resources collection for the application, as shown here:

```
<Application x:Class="Resources.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    StartupUri="Menu.xaml" >
  <Application.Resources>
    <ResourceDictionary>
      <ResourceDictionary.MergedDictionaries>
        <ResourceDictionary Source="AppBrushes.xaml"/>
        <ResourceDictionary Source="WizardBrushes.xaml"/>
      </ResourceDictionary.MergedDictionaries>
    </ResourceDictionary>
  </Application.Resources>
</Application>
```

This markup works by explicitly creating a ResourceDictionary object. The resources collection is always a ResourceDictionary object, but this is one case where you need to specify that detail explicitly so that you can also set the ResourceDictionary.MergedDictionaries property. If you don't take this step, the MergedDictionaries property will be null.

The MergedDictionaries collection is a collection of ResourceDictionary objects that you want to use to supplement your resource collection. In this example, there are two: one that's defined in the AppBrushes.xaml resource dictionary and another that's defined in the WizardBrushes.xaml.

If you want to add your own resources *and* merge in resource dictionaries, you simply need to place your resources before or after the MergedProperties section, as shown here:

```
<Application.Resources>
  <ResourceDictionary>
    <ResourceDictionary.MergedDictionaries>
      <ResourceDictionary Source="AppBrushes.xaml"/>
      <ResourceDictionary Source="WizardBrushes.xaml"/>
    </ResourceDictionary.MergedDictionaries>
    <ImageBrush x:Key="GraphicalBrush1" ... ></ImageBrush>
    <ImageBrush x:Key="GraphicalBrush2" ... ></ImageBrush>
  </ResourceDictionary>
</Application.Resources>
```

■ **Note** As you learned earlier, it's perfectly reasonable to have resources with the same name stored in different but overlapping resource collections. However, it's not acceptable to merge resource dictionaries that use the same resource names. If there's a duplicate, you'll receive a XamlParseException when you compile your application.

One reason to use resource dictionaries is to define one or more reusable application “skins” that you can apply to your controls. (You’ll learn how to develop this technique in Chapter 17.) Another reason is to store content that needs to be localized (such as error message strings).

Sharing Resources Between Assemblies

If you want to share a resource dictionary between multiple applications, you could copy and distribute the XAML file that contains the resource dictionary. This is the simplest approach, but it doesn’t give you any version control. A more structured approach is to compile your resource dictionary in a separate class library assembly and distribute that component instead.

When sharing a compiled assembly with one or more resource dictionaries, there’s another challenge to face—namely, you need a way to extract the resource you want and use it in your application. There are two approaches you can take. The most straightforward solution is to use code that creates the appropriate ResourceDictionary object. For example, if you have a resource dictionary in a class library assembly named ReusableDictionary.xaml, you could use the following code to create it manually:

```
ResourceDictionary resourceDictionary = new ResourceDictionary();
resourceDictionary.Source = new Uri(
  "ResourceLibrary;component/ReusableDictionary.xaml", UriKind.Relative);
```

This code snippet uses the pack URI syntax you learned about earlier in this chapter. It constructs a relative URI that points to the compiled XAML resource named ReusableDictionary.xaml in the other assembly. Once you’ve created the ResourceDictionary object, you can manually retrieve the resource you want from the collection:

```
cmd.Background = (Brush)resourceDictionary["TileBrush"];
```

However, you don’t need to assign resources manually. Any DynamicResource references you have in your window will be automatically reevaluated when you load a new resource dictionary. You’ll see an example of this technique in Chapter 17, when you build a dynamic skinning feature.

If you don’t want to write any code, you have another choice. You can use the ComponentResourceKey markup extension, which is designed for just this purpose. You use the ComponentResourceKey to create the key name for your resource. By taking this step, you indicate to WPF that you plan to share your resource between assemblies.

■ **Note** Up until this point, you’ve only seen resources that use strings (such as “TileBrush”) for key names. Using a string is the most common way to name a resource. However, WPF has some clever resource extensibility that kicks in automatically when you use certain types of key names that aren’t strings. For example, in the next chapter you’ll see that you can use a Type object as a key name for a style. This tells WPF to apply the style to the appropriate type of element automatically. Similarly, you can use an instance of ComponentResourceKey as a key name for any resource you want to share between assemblies.

Before you go any further, you need to make sure you’ve given your resource dictionary the right name. For this trick to work, your resource dictionary must be in a file named generic.xaml, and that file must be placed in a Themes subfolder in your application. The resources in the generic.xaml files are considered part of the default theme, and they're always made available. You'll use this trick many more times, particularly when you build custom controls in Chapter 18.

Figure 10-3 shows the proper organization of files. The top project, named ResourceLibrary, includes the generic.xaml file in the correct folder. The bottom project, named Resources, has a reference to ResourceLibrary, so it can use the resources it contains.

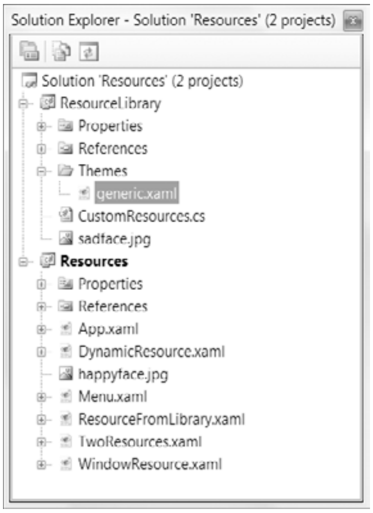

Figure 10-3. *Sharing resources with a class library*

■ **Tip** If you have a lot of resources and you want to organize them in the best way possible, you can create individual resource dictionaries, just as you did before. However, make sure you merge these dictionaries into the generic.xaml file so that they're readily available.

The next step is to create the key name for the resource you want to share, which is stored in the ResourceLibrary assembly. When using a ComponentResourceKey, you need to supply two pieces of information: a reference to a class in your class library assembly and a descriptive resource ID. The class reference is part of the magic that allows WPF to share your resource with other assemblies. When they use the resource, they'll supply the same class reference and the same resource ID.

It doesn't matter what this class actually looks like, and it doesn't need to contain code. The assembly where this type is defined is the same assembly where ComponentResourceKey will find the resource. The example shown in Figure 10-3 uses a class named CustomResources, which has no code:

```
public class CustomResources
{}
```

Now you can create a key name using this class and a resource ID:

```
x:Key="{ComponentResourceKey TypeInTargetAssembly={x:Type local:CustomResources},
 ResourceId=SadTileBrush}"
```

Here's the complete markup for the generic.xaml file, which includes a single resource—an ImageBrush that uses a different graphic:

```
<ResourceDictionary
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 xmlns:local="clr-namespace:ResourceLibrary">

  <ImageBrush
   x:Key="{ComponentResourceKey TypeInTargetAssembly={x:Type local:CustomResources},
ResourceId=SadTileBrush}"
   TileMode="Tile" ViewportUnits="Absolute" Viewport="0 0 32 32"
   ImageSource="ResourceLibrary;component/sadface.jpg" Opacity="0.3">
  </ImageBrush>
</ResourceDictionary>
```

Keen eyes will notice one unexpected detail in this example. The ImageSource property is no longer set with the image name (sadface.jpg). Instead, a more complex relative URI is used that clearly indicates the image is a part of the ResourceLibrary component. This is a required step because this resource will be used in the context of another application. If you simply use the image name, that application will search its own resources to find the image. What you really need is a relative URI that indicates the component where the image is stored.

Now that you've created the resource dictionary, you can use it in another application. First, make sure you've defined a prefix for the class library assembly, as shown here:

```
<Window x:Class="Resources.ResourceFromLibrary"
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 xmlns:res="clr-namespace:ResourceLibrary;assembly=ResourceLibrary"
 ... >
```

You can then use a DynamicResource that contains a ComponentResourceKey. (This makes sense because the ComponentResourceKey *is* the resource name.) The ComponentResourceKey you use in the consumer is the same as the ComponentResourceKey you use in the class library. You supply a reference to the same class and the same resource ID. The only difference is that you may not use the same XML namespace prefix. This example uses res instead of local, so as to emphasize the fact that the CustomResources class is defined in a different assembly:

```
<Button Background="{DynamicResource {ComponentResourceKey
 TypeInTargetAssembly={x:Type res:CustomResources}, ResourceId=SadTileBrush}}"
 Padding="5" Margin="5" FontWeight="Bold" FontSize="14">
  A Resource From ResourceLibrary
</Button>
```

■ **Note** You must use a dynamic resource, not a static resource, when using a ComponentResourceKey.

This completes the example. However, you can take one additional step to make it easier to use your resource. You can define a static property that returns the correct ComponentResourceKey that you need to use. Typically, you'll define this property in a class in your component, as shown here:

```
public class CustomResources
{
    public static ComponentResourceKey SadTileBrushKey
    {
        get
        {
            return new ComponentResourceKey(
              typeof(CustomResources), "SadTileBrush");
        }
    }
}
```

Now you use the Static markup extension to access this property and apply the resource without using the long-winded ComponentResourceKey in your markup:

```
<Button
 Background="{DynamicResource {x:Static res:CustomResources.SadTileBrushKey}}"
 Padding="5" Margin="5" FontWeight="Bold" FontSize="14">
  A Resource From ResourceLibrary
</Button>
```

This handy shortcut is essentially the same technique that's used by the System*Xxx* classes that you saw earlier. For example, when you retrieve SystemColors.WindowTextBrushKey, you are receiving the correct resource key object. The only difference is that it's an instance of the private SystemResourceKey rather than ComponentResourceKey. Both classes derive from the same ancestor: an abstract class named ResourceKey.

The Last Word

In this chapter, you explored how the WPF resource system lets you reuse the same objects in different parts of your application. You saw how to declare resources in code and markup, how to draw on system resources, and how to share resources between applications with class library assemblies.

You're not done looking at resources just yet. One of the most practical uses of object resources is to store *styles*—collections of property settings that you can apply to multiple elements. In the next chapter, you'll learn how to define styles, store them as resources, and reuse them effortlessly.

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.

■ **What's New** Although styles haven't changed in WPF 4, behaviors are an entirely new feature that's been introduced with recent versions of Expression Blend. Behaviors formalize a design pattern (usually called *attached behaviors*) that was already common in WPF applications. However, they also add first-rate design-time support for Expression Blend users.

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 nonformatting characteristics, such as 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:

```
<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:

- 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>
```