There are two sides to a drag-and-drop operation: the source and target. To create a drag-and-drop source, you need to call the DragDrop.DoDragDrop() method at some point to initiate the drag-and-drop operation. At this point you identify the source of the drag-and-drop operation, set aside the content you want to transfer, and indicate what drag-and-drop effects are allowed (copying, moving, and so on).

Usually, the DoDragDrop() method is called in response to the MouseDown or PreviewMouseDown event. Here's an example that initiates a drag-and-drop operation when a label is clicked. The text content from the label is used for the drag-and-drop operation:

```
private void lblSource_MouseDown(object sender, MouseButtonEventArgs e)
{
    Label lbl = (Label)sender;
    DragDrop.DoDragDrop(lbl, lbl.Content, DragDropEffects.Copy);
}
```

The element that receives the data needs to set its AllowDrop property to true. Additionally, it needs to handle the Drop event to deal with the data:

```
<Label Grid.Row="1" AllowDrop="True" Drop="lblTarget_Drop">To Here</Label>
```

When you set AllowDrop to true, you configure an element to allow any type of information. If you want to be pickier, you can handle the DragEnter event. At this point, you can check the type of data that's being dragged and then determine what type of operation to allow. The following example allows only text content—if you drag something that cannot be converted to text, the drag-and-drop operation won't be allowed, and the mouse pointer will change to the forbidding circle-with-a-line cursor:

```
private void lblTarget_DragEnter(object sender, DragEventArgs e)
{
    if (e.Data.GetDataPresent(DataFormats.Text))
      e.Effects = DragDropEffects.Copy;
    else
      e.Effects = DragDropEffects.None;
}
```

Finally, when the operation completes, you can retrieve the data and act on it. The following code takes the dropped text and inserts it into the label:

```
private void lblTarget_Drop(object sender, DragEventArgs e)
{
    ((Label)sender).Content = e.Data.GetData(DataFormats.Text);
}
```

You can exchange any type of object through a drag-and-drop operation. However, although this free-spirited approach is perfect for your applications, it isn't wise if you need to communicate with other applications. If you want to drag and drop into other applications, you should use a basic data type (such as string, int, and so on) or an object that implements ISerializable or IDataObject (which allows .NET to transfer your object into a stream of bytes and reconstruct the object in another application domain). One interesting trick is to convert a WPF element into XAML and reconstitute it somewhere else. All you need is the XamlWriter and XamlReader objects described in Chapter 2.

■ **Note** If you want to transfer data between applications, be sure to check out the System.Windows.Clipboard class, which provides static methods for placing data on the Windows clipboard and retrieving it in a variety of different formats.

Multitouch Input

Multitouch is a way of interacting with an application by touching a screen. What distinguishes multitouch input from more traditional pen-based input is that multitouch allows the user to work with several fingers at once. At its most sophisticated, multitouch recognizes *gestures*—specific ways the user can move more than one finger to perform a common operation. For example, placing two fingers on the touchscreen and moving them together is generally accepted to mean "zoom in," while pivoting one finger around another means "rotate." And because the user makes these gestures directly on the application window, each gesture is naturally connected to a specific object. For example, a simple multitouch-enabled application might show multiple pictures on a virtual desktop and allow the user to drag, resize, and rotate each image to create a new arrangement. This sort of application is a minor showstopper—and with WPF, it's almost easy.

■ **Tip** For a list of standard multitouch gestures that Windows 7 can recognize, see `http://tinyurl.com/yawwhw2`.

Many developers believe that multitouch is the eventual future of application interaction, at least on rich devices such as desktop computers and laptops. But currently, multitouch support is limited to a small set of touchscreen laptops, all-in-one desktops, and LCD monitors. At the time of this writing, you can find a list of current multitouch hardware at `http://tinyurl.com/y8pnsbu`.

This presents a challenge for developers who want to experiment with multitouch applications. By far, the best approach is to invest in a basic multitouch laptop. However, with a bit of work, you can use an *emulator* to simulate multitouch input. All you need to do is connect more than one mouse to your computer and install the drivers from the Multi-Touch Vista open source project (which also works with Windows 7). To get started, surf to `http://multitouchvista.codeplex.com`. But be warned—you'll probably need to follow the tutorial videos to make sure you get the rather convoluted setup procedure done right.

■ **Note** Although some applications may support multitouch on Windows Vista, the support that's built into WPF requires Windows 7, regardless of whether you have supported hardware or use an emulator.

The Levels of Multitouch Support

As you've seen, WPF allows you to work with keyboard and mouse input at a high level (for example, clicks and text changes) or a low level (mouse movements and key presses). This is important, because some applications need a much finer degree of control. The same applies to multitouch input, and WPF provides three separate layers of multitouch support:

- **Raw touch.** This is the lowest level of support, and it gives you access to every touch the user makes. The disadvantage is that it's up to your application to combine separate touch messages together and interpret them. Raw touch makes sense if you don't plan to recognize the standard touch gestures but instead want to create an application that reacts to multitouch input in a unique way. One example is a painting program such as Windows 7 Paint, which lets users "finger paint" on a touchscreen with several fingers at once.
- **Manipulation.** This is a convenient abstraction that translates raw multitouch input into meaningful gestures, much like WPF controls interpret a sequence of MouseDown and MouseUp events as a higher-level MouseDoubleClick. The common gestures that WPF elements support include pan, zoom, rotate, and tap.
- **Built-in element support.** Some elements already react to multitouch events, with no code required. For example, scrollable controls such as the ListBox, ListView, DataGrid, TextBox, and ScrollViewer support touch panning.

The following sections show examples of both raw-touch and manipulation with gestures.

Raw Touch

As with the basic mouse and keyboard events, touch events are built into the low-level UIElement and ContentElement classes. Table 5-7 lists them all.

Table 5-7. *Raw Touch Events for All Elements*

Name	Routing Type	Description
PreviewTouchDown	Tunneling	Occurs when the user touches down on this element
TouchDown	Bubbling	Occurs when the user touches down on this element
PreviewTouchMove	Tunneling	Occurs when the user moves the touched-down finger
TouchMove	Bubbling	Occurs when the user moves the touched-down finger

Name	Routing Type	Description
PreviewTouchUp	Tunneling	Occurs when the user lifts the finger, ending the touch
TouchUp	Bubbling	Occurs when the user lifts the finger, ending the touch
TouchEnter	None	Occurs when a contact point moves from outside this element into this element
TouchLeave	None	Occurs when a contact point moves out of this element

All of these events provide a TouchEventArgs object, which provides two important members. First, the GetTouchPoint() method gives you the screen coordinates where the touch event occurred (along with less commonly used data, such as the size of the contact point). Second, TouchDevice property returns a TouchDevice object. The trick here is that every contact point is treated as a separate device. So if a user presses two fingers down at different positions (either simultaneously or one after the other), WPF treats it as two touch devices and assigns a unique ID to each. As the user moves these fingers and the touch events occur, your code can distinguish between the two contact points by paying attention to the TouchDevice.Id property.

The following example shows how this works with a simple demonstration of raw-touch programming (see Figure 5-9). When the user touches down on the Canvas, the application adds a small ellipse element to show the contact point. Then, as the user moves the finger, the code moves the ellipse so it follows along.

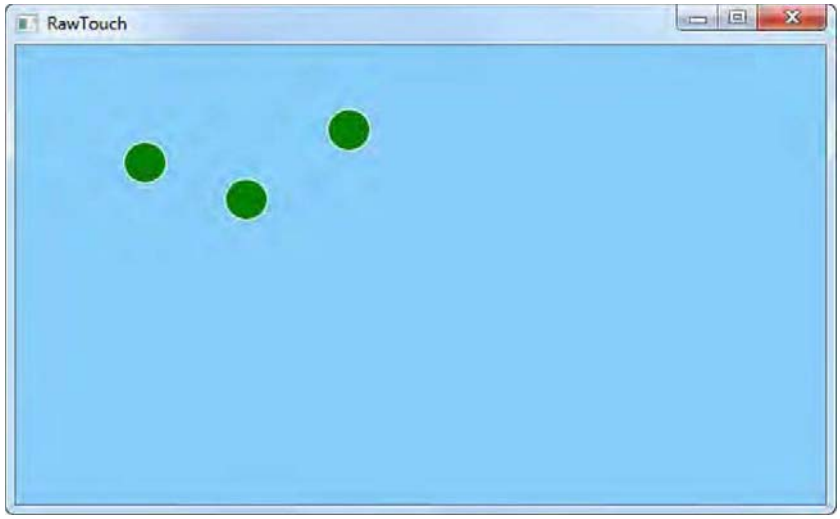

Figure 5-9. *Dragging circles with multitouch*

What distinguishes this example from a similar mouse event test is that the user can touch down with several fingers at once, causing multiple ellipses to appear, each of which can be dragged about independently.

To create this example, you need to handle the TouchDown, TouchUp, and TouchMove events:

```
<Canvas x:Name="canvas" Background="LightSkyBlue"
  TouchDown="canvas_TouchDown" TouchUp="canvas_TouchUp"
  TouchMove="canvas_TouchMove">
</Canvas>
```

To keep track of all the contact points, you need to store a collection as a window member variable. The cleanest approach is to store a collection of UIElement objects (one for each active ellipse), indexed by the touch device ID (which is an integer):

```
private Dictionary<int, UIElement> movingEllipses =
  new Dictionary<int, UIElement>();
```

When the user touches a finger down, the code creates and configures a new Ellipse element (which looks like a small circle). It places the ellipse at the appropriate coordinates using the touch point, adds it to the collection (indexed by the touch device ID), and then shows it in the Canvas:

```
private void canvas_TouchDown(object sender, TouchEventArgs e)
{
    // Create an ellipse to draw at the new contact point.
    Ellipse ellipse = new Ellipse();
    ellipse.Width = 30;
    ellipse.Height = 30;
    ellipse.Stroke = Brushes.White;
    ellipse.Fill = Brushes.Green;

    // Position the ellipse at the contact point.
    TouchPoint touchPoint = e.GetTouchPoint(canvas);
    Canvas.SetTop(ellipse, touchPoint.Bounds.Top);
    Canvas.SetLeft(ellipse, touchPoint.Bounds.Left);

    // Store the ellipse in the active collection.
    movingEllipses[e.TouchDevice.Id] = ellipse;

    // Add the ellipse to the Canvas.
    canvas.Children.Add(ellipse);
}
```

When the user moves a touched-down finger, the TouchMove event fires. At this point, you can determine which point is moving using the touch device ID. All the code needs to do is find the corresponding ellipse and update its coordinates:

```
private void canvas_TouchMove(object sender, TouchEventArgs e)
{
    // Get the ellipse that corresponds to the current contact point.
    UIElement element = movingEllipses[e.TouchDevice.Id];
```

```
    // Move it to the new contact point.
    TouchPoint touchPoint = e.GetTouchPoint(canvas);
    Canvas.SetTop(ellipse, touchPoint.Bounds.Top);
    Canvas.SetLeft(ellipse, touchPoint.Bounds.Left);
}
```

Finally, when the user lifts the finger, the ellipse is removed from the tracking collection. Optionally, you may want to remove it from the Canvas now as well.

```
private void canvas_TouchUp(object sender, TouchEventArgs e)
{
    // Remove the ellipse from the Canvas.
    UIElement element = movingEllipses[e.TouchDevice.Id];
    canvas.Children.Remove(element);

    // Remove the ellipse from the tracking collection.
    movingEllipses.Remove(e.TouchDevice.Id);
}
```

■ **Note** The UIElement also adds the CaptureTouch() and ReleaseTouchCapture() methods, which are analogous to the CaptureMouse() and ReleaseMouseCapture() methods. When touch input is captured by an element, that element receives all the touch events from that touch device, even if the touch events happen in another part of the window. But because there can be multiple touch devices, several different elements can capture touch input at once, as long as each captures the input from a different device.

Manipulation

Raw touch is great for applications that use touch events in a direct, straightforward way, like the dragging circles example or a painting program. But if you want to support the standard touch gestures, raw touch doesn't make it easy. For example, to support a rotation, you'd need to detect two contact points on the same element, keep track of how they move, and use some sort of calculation to determine that one is rotating around the other. Even then, you still need to add the code that actually applies the corresponding rotation effect.

Fortunately, WPF doesn't leave you completely on your own. It includes higher-level support for gestures, called touch *manipulation*. You configure an element to opt in to manipulation by setting its IsManipulationEnabled property to true. You can then react to four manipulation events: ManipulationStarting, ManipulationStarted, ManipulationDelta, and ManipulationCompleted.

Figure 5-10 shows a manipulation example. Here, three images are shown in a Canvas in a basic arrangement. The user can then use panning, rotating, and zooming gestures to move, turn, shrink, or expand them.

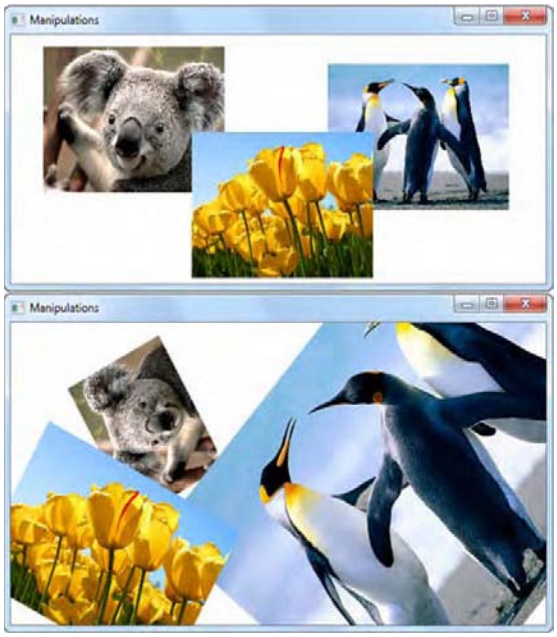

Figure 5-10. *Before and after: three images manipulated with multitouch*

The first step to create this example is to define the Canvas and place the three Image elements. To make life easy, the ManipulationStarting and ManipulationDelta events are handled in the Canvas, after they bubble up from the appropriate Image element inside.

```
<Canvas x:Name="canvas" ManipulationStarting="image_ManipulationStarting"
 ManipulationDelta="image_ManipulationDelta">
  <Image Canvas.Top="10" Canvas.Left="10" Width="200"
   IsManipulationEnabled="True" Source="koala.jpg">
    <Image.RenderTransform>
      <MatrixTransform></MatrixTransform>
    </Image.RenderTransform>
  </Image>
  <Image Canvas.Top="30" Canvas.Left="350" Width="200"
   IsManipulationEnabled="True" Source="penguins.jpg">
    <Image.RenderTransform>
      <MatrixTransform></MatrixTransform>
    </Image.RenderTransform>
  </Image>
  <Image Canvas.Top="100" Canvas.Left="200" Width="200"
   IsManipulationEnabled="True" Source="tulips.jpg">
    <Image.RenderTransform>
      <MatrixTransform></MatrixTransform>
    </Image.RenderTransform>
  </Image>
</Canvas>
```

There's one new detail in this markup. Each image includes a MatrixTransform, which gives the code an easy way to apply a combination of movement, rotation, and zoom manipulations. Currently, the MatrixTransform objects don't do anything, but the code will alter them when the manipulation events occur. (You'll get the full details about how transforms work in Chapter 12.)

When the user touches down on one of the images, the ManipulationStarting event fires. At this point, you need to set the manipulation container, which is the reference point for all manipulation coordinates you'll get later. In this case, the Canvas that contains the images is the natural choice. Optionally, you can choose what types of manipulations should be allowed. If you don't, WPF will watch for any gesture it recognizes: pan, zoom, and rotate.

```
private void image_ManipulationStarting(object sender,
  ManipulationStartingEventArgs e)
{
    // Set the container (used for coordinates.)
    e.ManipulationContainer = canvas;

    // Choose what manipulations to allow.
    e.Mode = ManipulationModes.All;
}
```

The ManipulationDelta event fires when a manipulation is taking place (but not necessarily finished). For example, if the user begins to rotate an image, the ManipulationDelta event will fire continuously, until the user rotation is finished and the user raises up the touched-down fingers.

The current state of the gesture is recorded by a ManipulationDelta object, which is exposed through the ManipulationDeltaEventArgs.DeltaManipulation property. Essentially, the ManipulationDelta object records the amount of zooming, rotating, and panning that should be applied to an object and exposes that information through three straightforward properties: Scale, Rotation, and Translation. The trick is to use this information to adjust the element in your user interface.

In theory, you could deal with the scale and rotation details by changing the element's size and position. But this still doesn't apply a rotation (and the code is somewhat messy). A far better approach is to use *transforms*—objects that allow you to mathematically warp the appearance of any WPF element. The idea is to take the information supplied by the ManipulationDelta object and use it to configure a MatrixTransform. Although this sounds complicated, the code you need to use is essentially the same in every application that uses this feature. It looks like this:

```
private void image_ManipulationDelta(object sender, ManipulationDeltaEventArgs e)
{
    // Get the image that's being manipulated.
    UIElement element = (UIElement)e.Source;

    // Use the matrix of the transform to manipulate the element's appearance.
    Matrix matrix = ((MatrixTransform)element.RenderTransform).Matrix;

    // Get the ManipulationDelta object.
    ManipulationDelta deltaManipulation = e.DeltaManipulation;

    // Find the old center, and apply any previous manipulations.
    Point center = new Point(element.ActualWidth / 2, element.ActualHeight / 2);
    center = matrix.Transform(center);
```

```
    // Apply new zoom manipulation (if it exists).
    matrix.ScaleAt(deltaManipulation.Scale.X, deltaManipulation.Scale.Y,
      center.X, center.Y);

    // Apply new rotation manipulation (if it exists).
    matrix.RotateAt(e.DeltaManipulation.Rotation, center.X, center.Y);

    // Apply new panning manipulation (if it exists).
    matrix.Translate(e.DeltaManipulation.Translation.X,
      e.DeltaManipulation.Translation.Y);

    // Set the final matrix.
    ((MatrixTransform)element.RenderTransform).Matrix = matrix;
}
```

This code allows you to manipulate all the images, as shown in Figure 5-10.

Inertia

WPF has another layer of features that build on its basic manipulation support, called *inertia*. Essentially, inertia allows a more realistic, fluid manipulation of elements.

Right now, if a user drags one of the images in Figure 5-10 using a panning gesture, the image stops moving the moment the fingers are raised from the touchscreen. But if inertia is enabled, the movement would continue for a very short period, decelerating gracefully. This gives manipulation a presence and sense of momentum. And inertia also causes elements to bounce back when they dragged into a boundary they can't cross, allowing them to act like real, physical objects.

To add inertia to the previous example, you simply need to handle the ManipulationInertiaStarting event. Like the other manipulation event, it will begin in one of the images and bubble up to the Canvas. The ManipulationInertiaStarting event fires when the user ends the gesture and releases the element by raising the fingers. At this point, you can use the ManipulationInertiaStartingEventArgs object to determine the current velocity—the speed at which the element was moving when the manipulation ended—and set the deceleration speed you want. Here's an example that adds inertia to translation, zooming, and rotation gestures:

```
private void image_ManipulationInertiaStarting(object sender,
  ManipulationInertiaStartingEventArgs e)
{
    // If the object is moving, decrease its speed by
    // 10 inches per second every second.
    // deceleration = 10 inches * 96 units per inch / (1000 milliseconds)^2
    e.TranslationBehavior = new InertiaTranslationBehavior();
    e.TranslationBehavior.InitialVelocity = e.InitialVelocities.LinearVelocity;
    e.TranslationBehavior.DesiredDeceleration = 10.0 * 96.0 / (1000.0 * 1000.0);

    // Decrease the speed of zooming by 0.1 inches per second every second.
    // deceleration = 0.1 inches * 96 units per inch / (1000 milliseconds)^2
    e.ExpansionBehavior = new InertiaExpansionBehavior();
    e.ExpansionBehavior.InitialVelocity = e.InitialVelocities.ExpansionVelocity;
    e.ExpansionBehavior. DesiredDeceleration = 0.1 * 96 / 1000.0 * 1000.0;
```

```
    // Decrease the rotation rate by 2 rotations per second every second.
    // deceleration = 2 * 360 degrees / (1000 milliseconds)^2
    e.RotationBehavior = new InertiaRotationBehavior();
    e.RotationBehavior.InitialVelocity = e.InitialVelocities.AngularVelocity;
    e.RotationBehavior. DesiredDeceleration = 720 / (1000.0 * 1000.0);
}
```

To make elements bounce back naturally from barriers, you need to check whether they've drifted into the wrong place in the ManipulationDelta event. If a boundary is crossed, it's up to you to report it by calling ManipulationDeltaEventArgs.ReportBoundaryFeedback().

At this point, you might be wondering why you need to write so much of the manipulation code if it's standard boilerplate that all multitouch developers need. One obvious advantage is that it allows you to easily tweak some of the details (such as the amount of deceleration in the inertia settings). However, in many situations, you may be able to get exactly what you need with prebuilt manipulation support, in which case you should check out the WPF Multitouch project at `http://multitouch.codeplex.com`. It includes two convenient ways that you can add manipulation support to a container without writing it yourself—either by using a behavior that applies it automatically (see Chapter 11) or by using a custom control that has the logic hardwired (see Chapter 18). Best of all, it's free to download, and the source code is ready for tweaking.

The Last Word

In this chapter, you took a deep look at routed events. First, you explored routed events and saw how they allow you to deal with events at different levels—either directly at the source or in a containing element. Next, you saw how these routing strategies are implemented in the WPF elements to allow you to deal with keyboard, mouse, and multitouch input.

It may be tempting to begin writing event handlers that respond to common events such as mouse movements to apply simple graphical effects or otherwise update the user interface. But don't start writing this logic just yet. As you'll see later in Chapter 11, you can automate many simple program operations with declarative markup using WPF styles and triggers. But before you branch out to this topic, the next chapter takes a short detour to show you how many of the most fundamental controls (things such as buttons, labels, and text boxes) work in the WPF world.

CHAPTER 6

Controls

Now that you've learned the fundamentals of layout, content, and event handling, you're ready to take a closer look at WPF's family of elements.

In this chapter, you'll consider *controls*—elements that derive from the System.Windows.Control class. You'll begin by examining the base Control class, and learning how it supports brushes and fonts. Then you'll explore the full catalog of WPF controls, including the following:

- **Content controls.** These controls can contain nested elements, giving them nearly unlimited display abilities. They include the Label, Button, and ToolTip classes.
- **Headered content controls.** These are content controls that allow you to add a main section of content and a separate title portion. They are usually intended to wrap larger blocks of user interface. They include the TabItem, GroupBox, and Expander classes.
- **Text controls.** This is the small set of controls that allow users to enter input. The text controls support ordinary text (the TextBox), passwords (the PasswordBox), and formatted text (the RichTextBox, which is discussed in Chapter 28),
- **List controls.** These controls show collections of items in a list. They include the ListBox and ComboBox classes.
- **Range-based controls.** These controls have just one thing in common: a Value property that can be set to any number in a prescribed range. Examples include the Slider and ProgressBar classes.
- **Date controls.** This category includes two controls that allow users to select dates: the Calendar and DatePicker.

There are several types of controls that you won't see in this chapter, including those that create menus, toolbars, and ribbons; those that show grids and trees of bound data; and those that allow rich document viewing and editing. You'll consider these more advanced controls throughout this book, as you explore the related WPF features.

■ **What's New** Although WPF is continually making minor refinements to its control classes, WPF 4 adds just a few significant changes to the controls covered in this chapter. The most important is WPF 4's ability to display clearer text at small sizes (see the section "Text Formatting Mode"). WPF 4 also gives the TextBox control the ability to use a custom spell-check dictionary, and it adds two entirely new date controls: the Calendar and DatePicker.

The Control Class

WPF windows are filled with elements, but only some of these elements are *controls*.

In the world of WPF, a control is generally described as a *user-interactive* element—that is, an element that can receive focus and accept input from the keyboard or mouse. Obvious examples include text boxes and buttons. However, the distinction is sometimes a bit blurry. A tooltip is considered to be a control because it appears and disappears depending on the user's mouse movements. A label is considered to be a control because of its support for *mnemonics* (shortcut keys that transfer the focus to related controls).

All controls derive from the System.Windows.Control class, which adds a bit of basic infrastructure:

- The ability to set the alignment of content inside the control
- The ability to set the tab order
- Support for painting a background, foreground, and border
- Support for formatting the size and font of text content

Background and Foreground Brushes

All controls include the concept of a background and foreground. Usually, the background is the surface of the control (think of the white or gray area inside the borders of a button), and the foreground is the text. In WPF, you set the color of these two areas (but not the content) using the Background and Foreground properties.

It's natural to expect that the Background and Foreground properties would use color objects, as they do in a Windows Forms application. However, these properties actually use something much more versatile: a Brush object. That gives you the flexibility to fill your background and foreground content with a solid color (by using the SolidColorBrush) or something more exotic (for example, by using a LinearGradientBrush or TileBrush). In this chapter, you'll consider only the simple SolidColorBrush, but you'll try fancier brushwork in Chapter 12.

Setting Colors in Code

Imagine you want to set a blue surface area inside a button named cmd. Here's the code that does the trick:

```
cmd.Background = new SolidColorBrush(Colors.AliceBlue);
```

This code creates a new SolidColorBrush using a ready-made color via a static property of the handy Colors class. (The names are based on the color names supported by most web browsers.) It then sets the brush as the background brush for the button, which causes its background to be painted a light shade of blue.

■ **Note** This method of styling a button isn't completely satisfactory. If you try it, you'll find that it configures the background color for a button in its normal (unpressed) state, but it doesn't change the color that appears when you press the button (which is a darker gray). To really customize every aspect of a button's appearance, you need to delve into templates, as discussed in Chapter 17.

You can also grab system colors (which may be based on user preferences) from the System.Windows.SystemColors enumeration. Here's an example:

```
cmd.Background = new SolidColorBrush(SystemColors.ControlColor);
```

Because system brushes are used frequently, the SystemColors class also provides ready-made properties that return SolidColorBrush objects. Here's how to use them:

```
cmd.Background = SystemColors.ControlBrush;
```

As written, both of these examples suffer from a minor problem. If the system color is changed *after* you run this code, your button won't be updated to use the new color. In essence, this code grabs a snapshot of the current color or brush. To make sure your program can update itself in response to configuration changes, you need to use dynamic resources, as described in Chapter 10.

The Colors and SystemColors classes offer handy shortcuts, but they're not the only way to set a color. You can also create a Color object by supplying the R, G, B values (red, green, and blue). Each one of these values is a number from 0 to 255:

```
int red = 0; int green = 255; int blue = 0;
cmd.Foreground = new SolidColorBrush(Color.FromRgb(red, green, blue));
```

You can also make a color partly transparent by supplying an alpha value and calling the Color.FromArgb() method. An alpha value of 255 is completely opaque, while 0 is completely transparent.

RGB and scRGB

The RGB standard is useful because it's used in many other programs. For example, you can get the RGB value of a color in a graphic in a paint program and use the same color in your WPF application. However, it's possible that other devices (such as printers) might support a richer range of colors. For this reason, an alternative scRGB standard has been created. This standard represents each color component (alpha, red, green, and blue) using 64-bit values.

The WPF Color structure supports either approach. It includes a set of standard RGB properties (A, R, G, and B) and a set of properties for scRGB (ScA, ScR, ScG, and ScB). These properties are linked, so that if you set the R property, the ScR property is changed accordingly.

The relationship between the RGB values and the scRGB values is not linear. A 0 value in the RGB system is 0 in scRGB, 255 in RGB becomes 1 in scRGB, and all values in between 0 and 255 in RGB are represented as decimal values in between 0 and 1 in scRGB.

Setting Colors in XAML

When you set the background or foreground in XAML, you can use a helpful shortcut. Rather than define a Brush object, you can supply a color name or color value. The WPF parser will automatically create a SolidColorBrush object using the color you specify, and it will use that brush object for the foreground or background. Here's an example that uses a color name:

```
<Button Background="Red">A Button</Button>
```

It's equivalent to this more verbose syntax:

```
<Button>A Button
  <Button.Background>
    <SolidColorBrush Color="Red" />
  </Button.Background>
</Button>
```

You need to use the longer form if you want to create a different type of brush, such as a LinearGradientBrush, and use that to paint the background.

If you want to use a color code, you need to use a slightly less convenient syntax that puts the R, G, and B values in hexadecimal notation. You can use one of two formats—either #rrggbb or #aarrggbb (the difference being that the latter includes the alpha value). You need only two digits to supply the A, R, G, and B values because they're all in hexadecimal notation. Here's an example that creates the same color as in the previous code snippets using #aarrggbb notation:

```
<Button Background="#FFFF0000">A Button</Button>
```

Here, the alpha value is FF (255), the red value is FF (255), and the green and blue values are 0.

■ **Note** Brushes support automatic change notification. In other words, if you attach a brush to a control and change the brush, the control updates itself accordingly. This works because brushes derive from the System.Windows.Freezable class. The name stems from the fact that all freezable objects have two states—a readable state and a read-only (or "frozen") state.

The Background and Foreground properties are not the only details you can set with a brush. You can also paint a border around controls (and some other elements, such as the Border element) using the BorderBrush and BorderThickness properties. BorderBrush takes a brush of your choosing, and

BorderThickness takes the width of the border in device-independent units. You need to set both properties before you'll see the border.

■ **Note** Some controls don't respect the BorderBrush and BorderThickness properties. The Button object ignores them completely because it defines its background and border using the ButtonChrome decorator. However, you can give a button a new face (with a border of your choosing) using templates, as described in Chapter 17.

Fonts

The Control class defines a small set of font-related properties that determine how text appears in a control. These properties are outlined in Table 6-1.

Table 6-1. *Font-Related Properties of the Control Class*

Name	Description
FontFamily	The name of the font you want to use.
FontSize	The size of the font in device-independent units (each of which is 1/96 inch). This is a bit of a change from tradition that's designed to support WPF's new resolution-independent rendering model. Ordinary Windows applications measure fonts using *points*, which are assumed to be 1/72 inch on a standard PC monitor. If you want to turn a WPF font size into a more familiar point size, you can use a handy trick—just multiply by 3/4. For example, a traditional 38-point font is equivalent to 48 units in WPF.
FontStyle	The angling of the text, as represented as a FontStyle object. You get the FontSyle preset you need from the static properties of the FontStyles class, which includes Normal, Italic, or Oblique lettering. (*Oblique* is an artificial way to create italic text on a computer that doesn't have the required italic font. Letters are taken from the normal font and slanted using a transform. This usually creates a poor result.)
FontWeight	The heaviness of text, as represented as a FontWeight object. You get the FontWeight preset you need from the static properties of the FontWeights class. Bold is the most obvious of these, but some typefaces provide other variations, such as Heavy, Light, ExtraBold, and so on.
FontStretch	The amount that text is stretched or compressed, as represented by a FontStretch object. You get the FontStretch preset you need from the static properties of the FontStretches class. For example, UltraCondensed reduces fonts to 50% of their normal width, while UltraExpanded expands them to 200%. Font stretching is an OpenType feature that is not supported by many typefaces. (To experiment with this property, try using the Rockwell font, which does support it.)

■ **Note** The Control class doesn't define any properties that *use* its font. While many controls include a property such as Text, that isn't defined as part of the base Control class. Obviously, the font properties don't mean anything unless they're used by the derived class.

Font Family

A *font family* is a collection of related typefaces. For example, Arial Regular, Arial Bold, Arial Italic, and Arial Bold Italic are all part of the Arial font family. Although the typographic rules and characters for each variation are defined separately, the operating system realizes they are related. As a result, you can configure an element to use Arial Regular, set the FontWeight property to Bold, and be confident that WPF will switch over to the Arial Bold typeface.

When choosing a font, you must supply the full family name, as shown here:

```
<Button Name="cmd" FontFamily="Times New Roman" FontSize="18">A Button</Button>
```

It's much the same in code:

```
cmd.FontFamily = "Times New Roman";
cmd.FontSize = "18";
```

When identifying a FontFamily, a shortened string is not enough. That means you can't substitute Times or Times New instead of the full name Times New Roman.

Optionally, you can use the full name of a typeface to get italic or bold, as shown here:

```
<Button FontFamily="Times New Roman Bold">A Button</Button>
```

However, it's clearer and more flexible to use just the family name and set other properties (such as FontStyle and FontWeight) to get the variant you want. For example, the following markup sets the FontFamily to Times New Roman and sets the FontWeight to FontWeights.Bold:

```
<Button FontFamily="Times New Roman" FontWeight="Bold">A Button</Button>
```

Text Decorations and Typography

Some elements also support more advanced text manipulation through the TextDecorations and Typography properties. These allow you to add embellishments to text. For example, you can set the TextDecorations property using a static property from the TextDecorations class. It provides just four decorations, each of which allows you to add some sort of line to your text. They include Baseline, OverLine, Strikethrough, and Underline. The Typography property is more advanced—it lets you access specialized typeface variants that only some fonts will provide. Examples include different number alignments, ligatures (connections between adjacent letters), and small caps.

For the most part, the TextDecorations and Typography features are found only in flow document content—which you use to create rich, readable documents. (Chapter 28 describes documents in detail.) However, the frills also turn up on the TextBox class. Additionally, they're supported by the TextBlock, which is a lighter-weight version of the Label that's perfect for showing small amounts of wrappable text content. Although you're unlikely to use text decorations with the TextBox or change its typography, you may want to use underlining in the TextBlock, as shown here:

```
<TextBlock TextDecorations="Underline">Underlined text</TextBlock>
```

If you're planning to place a large amount of text content in a window and you want to format individual portions (for example, underline important words), you should refer to Chapter 28, where you'll learn about many more flow elements. Although flow elements are designed for use with documents, you can nest them directly inside a TextBlock.

Font Inheritance

When you set any of the font properties, the values flow through to nested objects. For example, if you set the FontFamily property for the top-level window, every control in that window gets the same FontFamily value (unless the control explicitly sets a different font). This feature is similar to the Windows Forms concept of *ambient properties*, but the underlying plumbing is different. It works because the font properties are dependency properties, and one of the features that dependency properties can provide is property value inheritance—the magic that passes your font settings down to nested controls.

It's worth noting that property value inheritance can flow through elements that don't even support that property. For example, imagine you create a window that holds a StackPanel, inside of which are three Label controls. You can set the FontSize property of the window because the Window class derives from the Control class. You *can't* set the FontSize property for the StackPanel because it isn't a control. However, if you set the FontSize property of the window, your property value is still able to flow through the StackPanel to get to your labels inside and change their font sizes.

Along with the font settings, several other base properties use property value inheritance. In the Control class, the Foreground property uses inheritance. The Background property does not. (However, the default background is a null reference that's rendered by most controls as a transparent background. That means the parent's background will still show through.) In the UIElement class, AllowDrop, IsEnabled, and IsVisible use property inheritance. In the FrameworkElement, the CultureInfo and FlowDirection properties do.

■ **Note** A dependency property supports inheritance only if the FrameworkPropertyMetadata.Inherits flag is set to true, which is not the default. Chapter 4 discusses the FrameworkPropertyMetadata class and property registration in detail.

Font Substitution

When you're setting fonts, you need to be careful to choose a font that you know will be present on the user's computer. However, WPF does give you a little flexibility with a font fallback system. You can set

FontFamily to a comma-separated list of font options. WPF will then move through the list in order, trying to find one of the fonts you've indicated.

Here's an example that attempts to use Technical Italic font but falls back to Comic Sans MS or Arial if that isn't available:

```
<Button FontFamily="Technical Italic, Comic Sans MS, Arial">A Button</Button>
```

If a font family really does contain a comma in its name, you'll need to escape the comma by including it twice in a row.

Incidentally, you can get a list of all the fonts that are installed on the current computer using the static SystemFontFamilies collection of the System.Windows.Media.Fonts class. Here's an example that uses it to add fonts to a list box:

```
foreach (FontFamily fontFamily in Fonts.SystemFontFamilies)
{
    lstFonts.Items.Add(fontFamily.Source);
}
```

The FontFamily object also allows you to examine other details, such as the line spacing and associated typefaces.

■ **Note** One of the ingredients that WPF doesn't include is a dialog box for choosing a font. The WPF Text team has posted two much more attractive WPF font pickers, including a no-code version that uses data binding (`http://blogs.msdn.com/text/archive/2006/06/20/592777.aspx`) and a more sophisticated version that supports the optional typographic features that are found in some OpenType fonts (`http://blogs.msdn.com/text/archive/2006/11/01/sample-font-chooser.aspx`).

Font Embedding

Another option for dealing with unusual fonts is to embed them in your application. That way, your application never has a problem finding the font you want to use.

The embedding process is simple. First, you add the font file (typically, a file with the extension .ttf) to your application and set the Build Action to Resource. (You can do this in Visual Studio by selecting the font file in the Solution Explorer and changing its Build Action in the Properties window.)

Next, when you use the font, you need to add the character sequence `./#` before the font family name, as shown here:

```
<Label FontFamily="./#Bayern" FontSize="20">This is an embedded font</Label>
```

The `./` characters are interpreted by WPF to mean "the current folder." To understand what this means, you need to know a little more about XAML's packaging system.

As you learned in Chapter 2, you can run stand-alone (known as *loose*) XAML files directly in your browser without compiling them. The only limitation is that your XAML file can't use a code-behind file. In this scenario, the current folder is exactly that, and WPF looks at the font files that are in the same directory as the XAML file and makes them available to your application.

More commonly, you'll compile your WPF application to a .NET assembly before you run it. In this case, the current folder is still the location of the XAML document, only now that document has been compiled and embedded in your assembly. WPF refers to compiled resources using a specialized URI syntax that's discussed in Chapter 7. All application URIs start with `pack://application`. If you create a project named ClassicControls and add a window named EmbeddedFont.xaml, the URI for that window is this:

```
pack://application:,,,/ClassicControls/embeddedfont.xaml
```

This URI is made available in several places, including through the FontFamily.BaseUri property. WPF uses this URI to base its font search. Thus, when you use the `./` syntax in a compiled WPF application, WPF looks for fonts that are embedded as resources alongside your compiled XAML.

After the `./` character sequence, you can supply the file name, but you'll usually just add the number sign (#) and the font's real family name. In the previous example, the embedded font is named Bayern.

■ **Note** Setting up an embedded font can be a bit tricky. You need to make sure you get the font family name exactly right, and you need to make sure you choose the correct build action for the font file. Furthermore, Visual Studio doesn't currently provide design support for embedded fonts (meaning your control text won't appear in the correct font until you run your application). To see an example of the correct setup, refer to the sample code for this chapter.

Embedding fonts raises obvious licensing concerns. Unfortunately, most font vendors allow their fonts to be embedded in documents (such as PDF files) but not applications (such as WPF assemblies), even though an embedded WPF font isn't directly accessible to the end user. WPF doesn't make any attempt to enforce font licensing, but you should make sure you're on solid legal ground before you redistribute a font.

You can check a font's embedding permissions using Microsoft's free font properties extension utility, which is available at `http://www.microsoft.com/typography/TrueTypeProperty21.mspx`. Once you install this utility, right-click any font file, and choose Properties to see more detailed information about it. In particular, check the Embedding tab for information about the allowed embedding for this font. Fonts marked with Installed Embedding Allowed are suitable for WPF applications; fonts with Editable Embedding Allowed may not be. Consult with the font vendor for licensing information about a specific font.

Text Formatting Mode

The text rendering in WPF is significantly different from the rendering in older GDI-based applications. A large part of the difference is due to WPF's device-independent display system, but there are also significant enhancements that allow text to appear clearer and crisper, particularly on LCD monitors.

However, WPF text rendering has one well-known shortcoming. At small text sizes, text can become blurry and show undesirable artifacts (like color fringing around the edges). These problems don't occur with GDI text display, because GDI uses a number of tricks to optimize the clarity of small text. For example, GDI can change the shapes of small letters, adjust their positions, and line up everything on

pixel boundaries. These steps cause the typeface to lose its distinctive character, but they make for a better on-screen reading experience when dealing with very small text.

So how can you fix WPF's small-text display problem? The best solution is to scale up your text (on a 96-dpi monitor, the effect should disappear at a text size of about 15 device-independent units) or use a high-dpi monitor that has enough resolution to show sharp text at any size. But because these options often aren't practical, WPF 4 introduces a new feature: the ability to selectively use GDI-like text rendering.

To use GDI-style text rendering, you add the TextOptions.TextFormattingMode attached property to a text-displaying element like the TextBlock or Label, and set it to Display (rather than the standard value, Ideal). Here's an example:

```
<TextBlock FontSize="12" Margin="5">
This is a Test. Ideal text is blurry at small sizes.
</TextBlock>

<TextBlock FontSize="12" Margin="5" TextOptions.TextFormattingMode="Display">
This is a Test. Display text is crisp at small sizes.
</TextBlock>
```

It's important to remember that the TextFormattingMode property is a solution for small text only. If you use it on larger text (text above 15 points), the text will not be as clear, the spacing will not be as even, and the typeface will not be rendered as accurately. And if you use text in conjunction with a transform (discussed in Chapter 12) that rotates, resizes, or otherwise changes its appearance, you should always use WPF's standard text display mode. That's because the GDI-style optimization for display text is applied before any transforms. Once a transform is applied, the result will no longer be aligned on pixel boundaries, and the text will appear blurry.

Mouse Cursors

A common task in any application is to adjust the mouse cursor to show when the application is busy or to indicate how different controls work. You can set the mouse pointer for any element using the Cursor property, which is inherited from the FrameworkElement class.

Every cursor is represented by a System.Windows.Input.Cursor object. The easiest way to get a Cursor object is to use the static properties of the Cursors class (from the System.Windows.Input namespace). The cursors include all the standard Windows cursors, such as the hourglass, the hand, resizing arrows, and so on. Here's an example that sets the hourglass for the current window:

```
this.Cursor = Cursors.Wait;
```

Now when you move the mouse over the current window, the mouse pointer changes to the familiar hourglass icon (in Windows XP) or the swirl (in Windows Vista and Windows 7).

■ **Note** The properties of the Cursors class draw on the cursors that are defined on the computer. If the user has customized the set of standard cursors, the application you create will use those customized cursors.
