CHAPTER 5

Routed Events

In the previous chapter, you saw how WPF created a new dependency property system, reworking traditional .NET properties to improve performance and integrate new capabilities such as data binding and animation. In this chapter, you'll learn about the second shift: replacing ordinary .NET events with a higher-level routed event feature.

Routed events are events with more traveling power—they can tunnel down or bubble up the element tree and be processed by event handlers along the way. A routed event can be handled on one element (such as a label) even though it originates on another (such as an image inside that label). As with dependency properties, routed events can be consumed in the traditional way—by connecting an event handler with the right signature—but you need to understand how they work to unlock all their features.

In this chapter, you'll explore the WPF event system and learn how to fire and handle routed events. Once you've learned the basics, you'll consider the family of events that WPF elements provide. These include events for dealing with initialization, mouse and keyboard input, and multitouch devices.

Understanding Routed Events

Every .NET developer is familiar with the idea of *events*—messages that are sent by an object (such as a WPF element) to notify your code when something significant occurs. WPF enhances the .NET event model with the concept of *event routing*. Event routing allows an event to originate in one element but be raised by another one. For example, event routing allows a click that begins in a toolbar button to rise up to the toolbar and then to the containing window before it's handled by your code.

Event routing gives you the flexibility to write tight, well-organized code that handles events in the most convenient place. It's also a necessity for working with the WPF content model, which allows you to build simple elements (such as a button) out of dozens of distinct ingredients, each of which has its own independent set of events.

Defining, Registering, and Wrapping a Routed Event

The WPF event model is quite similar to the WPF property model. As with dependency properties, routed events are represented by read-only static fields, registered in a static constructor, and wrapped by a standard .NET event definition.

For example, the WPF Button class provides the familiar Click event, which is inherited from the abstract ButtonBase class. Here's how the event is defined and registered:

```
public abstract class ButtonBase : ContentControl, ...
{
    // The event definition.
    public static readonly RoutedEvent ClickEvent;

    // The event registration.
    static ButtonBase()
    {
        ButtonBase.ClickEvent = EventManager.RegisterRoutedEvent(
          "Click", RoutingStrategy.Bubble,
          typeof(RoutedEventHandler), typeof(ButtonBase));
        ...
    }

    // The traditional event wrapper.
    public event RoutedEventHandler Click
    {
        add
        {
            base.AddHandler(ButtonBase.ClickEvent, value);
        }
        remove
        {
            base.RemoveHandler(ButtonBase.ClickEvent, value);
        }
    }

    ...
}
```

Whereas dependency properties are registered with the DependencyProperty.Register() method, routed events are registered with the EventManager.RegisterRoutedEvent() method. When registering an event, you need to specify the name of the event, the type of routine (more on that later), the delegate that defines the syntax of the event handler (in this example, RoutedEventHandler), and the class that owns the event (in this example, ButtonBase).

Usually, routed events are wrapped by ordinary .NET events to make them accessible to all .NET languages. The event wrapper adds and removes registered callers by using the AddHandler() and RemoveHandler() methods, both of which are defined in the base FrameworkElement class and inherited by every WPF element.

Sharing Routed Events

As with dependency properties, the definition of a routed event can be shared between classes. For example, two base classes use the MouseUp event: UIElement (which is the starting point for ordinary WPF elements) and ContentElement (which is the starting point for content elements, which are individual bits of content that can be placed in a flow document). The MouseUp event is defined by the System.Windows.Input.Mouse class. The UIElement and ContentElement classes simply reuse the event with the RoutedEvent.AddOwner() method:

```
UIElement.MouseUpEvent = Mouse.MouseUpEvent.AddOwner(typeof(UIElement));
```

Raising a Routed Event

Of course, as with any event, the defining class needs to raise the routed event at some point. Exactly where this takes place is an implementation detail. However, the important factor is that your event is *not* raised through the traditional .NET event wrapper. Instead, you use the RaiseEvent() method that every element inherits from the UIElement class. Here's the appropriate code from deep inside the ButtonBase class:

```
RoutedEventArgs e = new RoutedEventArgs(ButtonBase.ClickEvent, this);
base.RaiseEvent(e);
```

The RaiseEvent() method takes care of firing the event to every caller that's been registered with the AddHandler() method. Because AddHandler() is public, callers have a choice—they can register themselves directly by calling AddHandler(), or they can use the event wrapper. (The following section demonstrates both approaches.) Either way, they'll be notified when the RaiseEvent() method is invoked.

All WPF events use the familiar .NET convention for event signatures. That first parameter of every event handler provides a reference to the object that fired the event (the sender). The second parameter is an EventArgs object that bundles together any additional details that might be important. For example, the MouseUp event provides a MouseEventArgs object that indicates what mouse buttons were pressed when the event occurred:

```
private void img_MouseUp(object sender, MouseButtonEventArgs e)
{
}
```

In WPF, if an event doesn't need to send any additional details, it uses the RoutedEventArgs class, which includes some details about how the event was routed. If the event *does* need to transmit extra information, it uses a more specialized RoutedEventArgs-derived object (such as MouseButtonEventArgs in the previous example). Because every WPF event argument class derives from RoutedEventArgs, every WPF event handler has access to information about event routing.

Handling a Routed Event

As you learned in Chapter 2, there are several ways to attach an event handler. The most common approach is to add an event attribute to your XAML markup. The event attribute is named after the event you want to handle, and its value is the name of the event handler method. Here's an example that uses this syntax to connect the MouseUp event of the Image to an event handler named img_MouseUp:

```
<Image Source="happyface.jpg" Stretch="None"
 Name="img" MouseUp="img_MouseUp" />
```

Although it's not required, it's a common convention to name event handler methods in the form *ElementName_EventName*. If the element doesn't have a defined name (presumably because you don't need to interact with it in any other place in your code), consider using the name it *would* have:

```
<Button Click="cmdOK_Click">OK</Button>
```

■ **Tip** It may be tempting to attach an event to a high-level method that performs a task, but you'll have more flexibility if you keep an extra layer of event-handling code. For example, clicking a button named cmdUpdate shouldn't trigger a method named UpdateDatabase() directly. Instead, it should call an event handler such as cmdUpdate_Click(), which can then call the UpdateDatabase() method that does the real work. This pattern gives you the flexibility to change where your database code is located, replace the update button with a different control, and wire several controls to the same process, all without limiting your ability to change the user interface later. If you want a simpler way to deal with actions that can be triggered from various places in a user interface (toolbar buttons, menu commands, and so on), you'll want to add the WPF command feature that's described in Chapter 9.

You can also connect an event with . Here's the code equivalent of the XAML markup shown previously:

```
img.MouseUp += new MouseButtonEventHandler(img_MouseUp);
```

This code creates a delegate object that has the right signature for the event (in this case, an instance of the MouseButtonEventHandler delegate) and points that delegate to the img_MouseUp() method. It then adds the delegate to the list of registered event handlers for the img.MouseUp event.

C# also allows a more streamlined syntax that creates the appropriate delegate object implicitly:

```
img.MouseUp += img_MouseUp;
```

The code approach is useful if you need to dynamically create a control and attach an event handler at some point during the lifetime of your window. By comparison, the events you hook up in XAML are always attached when the window object is first instantiated. The code approach also allows you to keep your XAML simpler and more streamlined, which is perfect if you plan to share it with nonprogrammers, such as a design artist. The drawback is a significant amount of boilerplate code that will clutter up your code files.

The previous code approach relies on the event wrapper, which calls the UIElement.AddHandler() method, as shown in the previous section. You can also connect an event directly by calling the UIElement.AddHandler() method yourself. Here's an example:

```
img.AddHandler(Image.MouseUpEvent,
  new MouseButtonEventHandler(img_MouseUp));
```

When you use this approach, you always need to create the appropriate delegate type (such as MouseButtonEventHandler). You can't create the delegate object implicitly, as you can when hooking up an event through the property wrapper. That's because the UIElement.AddHandler() method supports all WPF events and doesn't know the delegate type that you want to use.

Some developers prefer to use the name of the class where the event is defined, rather than the name of the class that is firing the event. Here's the equivalent syntax that makes it clear that the MouseUp event is defined in UIElement:

```
img.AddHandler(UIElement.MouseUpEvent,
  new MouseButtonEventHandler(img_MouseUp));
```

■ **Note** Which approach you use is largely a matter of taste. However, the drawback to this second approach is that it doesn't make it obvious that the Image class *provides* a MouseUp event. It's possible to confuse this code and assume it's attaching an event handler that's meant to deal with the MouseUp event in a nested element. You'll learn more about this technique in the section "Attached Events" later in this chapter.

If you want to detach an event handler, code is your only option. You can use the -= operator, as shown here:

```
img.MouseUp -= img_MouseUp;
```

Or you can use the UIElement.RemoveHandler() method:

```
img.RemoveHandler(Image.MouseUpEvent,
  new MouseButtonEventHandler(img_MouseUp));
```

It is technically possible to connect an event handler to the same event more than once. This is usually the result of a coding mistake. (In this case, the event handler will be triggered multiple times.) If you attempt to remove an event handler that's been connected twice, the event will still trigger the event handler, but just once.

Event Routing

As you learned in the previous chapter, many controls in WPF are content controls, and content controls can hold any type and amount of nested content. For example, you can build a graphical button out of shapes, create a label that mixes text and pictures, or put content in a specialized container to get a scrollable or collapsible display. You can even repeat this nesting process to go as many layers deep as you want.

This ability for arbitrary nesting raises an interesting question. For example, imagine you have a label like this one, which contains a StackPanel that brings together two blocks of text and an image:

```
<Label BorderBrush="Black" BorderThickness="1">
  <StackPanel>
    <TextBlock Margin="3">
     Image and text label</TextBlock>
    <Image Source="happyface.jpg" Stretch="None" />
    <TextBlock Margin="3">
     Courtesy of the StackPanel</TextBlock>
  </StackPanel>
</Label>
```

As you already know, every ingredient you place in a WPF window derives from UIElement at some point, including Label, StackPanel, TextBlock, and Image. UIElement defines some core events. For example, every class that derives from UIElement provides a MouseDown and MouseUp event.

But consider what happens when you click the image part of the fancy label shown here. Clearly, it makes sense for the Image.MouseDown and Image.MouseUp events to fire. But what if you want to treat all label clicks in the same way? In this case, it shouldn't matter whether the user clicks the image, some of the text, or part of the blank space inside the label border. In every case, you'd like to respond with the same code.

Clearly, you could wire up the same event handler to the MouseDown or MouseUp event of each element, but that would result in a significant amount of clutter and make your markup more difficult to maintain. WPF provides a better solution with its routed event model.

Routed events come in the following three flavors:

> *Direct events*: These are like ordinary .NET events. They originate in one element and don't pass to any other. For example, MouseEnter (which fires when the mouse pointer moves over an element) is a direct event.
>
> *Bubbling events*: These events travel *up* the containment hierarchy. For example, MouseDown is a bubbling event. It's raised first by the element that is clicked. Next, it's raised by that element's parent, then by *that* element's parent, and so on, until WPF reaches the top of the element tree.

Tunneling events These events travel *down* the containment hierarchy. They give you the chance to preview (and possibly stop) an event before it reaches the appropriate control. For example, PreviewKeyDown allows you to intercept a key press, first at the window level and then in increasingly more-specific containers until you reach the element that had focus when the key was pressed.

When you register a routed event by using the EventManager.RegisterEvent() method, you pass a value from the RoutingStrategy enumeration that indicates the event behavior you want to use for your event.

Because MouseUp and MouseDown are bubbling events, you can now determine what happens in the fancy label example. When the happy face is clicked, the MouseDown event fires in this order:

1. Image.MouseDown
2. StackPanel.MouseDown
3. Label.MouseDown

After the MouseDown event is raised for the label, it's passed on to the next control (which in this case is the Grid that lays out the containing window) and then to its parent (the window). The window is the top level of the containment hierarchy and the final stop in the event-bubbling sequence. It's your last chance to handle a bubbling event such as MouseDown. If the user releases the mouse button, the MouseUp event fires in the same sequence.

■ **Note** In Chapter 24, you'll learn how to create a page-based WPF application. In this situation, the top-level container isn't a window but an instance of the Page class.

You aren't limited to handling a bubbling event in one place. In fact, there's no reason you can't handle the MouseDown or MouseUp event at every level. But usually you'll choose the most appropriate level of event routing for the task at hand.

The RoutedEventArgs Class

When you handle a bubbling event, the sender parameter provides a reference to the last link in the chain. For example, if an event bubbles up from an image to a label before you handle it, the sender parameter references the label object.

In some cases, you'll want to determine where the event originally took place. You can get that information and other details from the properties of the RoutedEventArgs class (which are listed in Table 5-1). Because all WPF event argument classes inherit from RoutedEventArgs, these properties are available in any event handler.

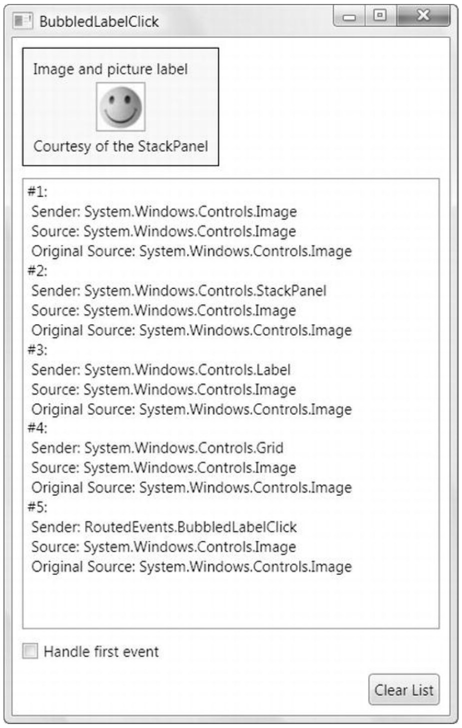
Table 5-1. *Properties of the RoutedEventArgs Class*

Name	Description
Source	Indicates what object raised the event. In the case of a keyboard event, this is the control that had focus when the event occurred (for example, when the key was pressed). In the case of a mouse event, this is the topmost element under the mouse pointer when the event occurred (for example, when a mouse button was clicked).

OriginalSource	Indicates what object originally raised the event. Usually, the OriginalSource is the same as the source. However, in some cases the OriginalSource goes deeper in the object tree to get a behind-the-scenes element that's part of a higher-level element. For example, if you click close to the border of a window, you'll get a Window object for the event source but a Border object for the original source. That's because a Window is composed of individual, smaller components. To take a closer look at this composition model (and learn how to change it), head to Chapter 17, which discusses control templates.
RoutedEvent	Provides the RoutedEvent object for the event triggered by your event handler (such as the static UIElement.MouseUpEvent object). This information is useful if you're handling different events with the same event handler.
Handled	Allows you to halt the event bubbling or tunneling process. When a control sets the Handled property to true, the event doesn't travel any further and isn't raised for any other elements. (As you'll see in the section "Handling a Suppressed Event," there is one way around this limitation.)

Bubbling Events

Figure 5-1 shows a simple window that demonstrates event bubbling. When you click a part of the label, the event sequence is shown in a list box. Figure 5-1 shows the appearance of this window immediately after you click the image in the label. The MouseUp event travels through five levels, ending up at the custom BubbledLabelClick form.



```
#1:
 Sender: System.Windows.Controls.Image
 Source: System.Windows.Controls.Image
 Original Source: System.Windows.Controls.Image
#2:
 Sender: System.Windows.Controls.StackPanel
 Source: System.Windows.Controls.Image
 Original Source: System.Windows.Controls.Image
#3:
 Sender: System.Windows.Controls.Label
 Source: System.Windows.Controls.Image
 Original Source: System.Windows.Controls.Image
#4:
 Sender: System.Windows.Controls.Grid
 Source: System.Windows.Controls.Image
 Original Source: System.Windows.Controls.Image
#5:
 Sender: RoutedEvents.BubbledLabelClick
 Source: System.Windows.Controls.Image
 Original Source: System.Windows.Controls.Image
```

Figure 5-1. *A bubbled image click*

To create this test form, the image and every element above it in the element hierarchy are wired up to the same event handler—a method named SomethingClicked(). Here's the XAML that does it:

```
<Window x:Class="RoutedEvents.BubbledLabelClick"
 xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
 xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 Title="BubbledLabelClick" Height="359" Width="329"
 MouseUp="SomethingClicked">
  <Grid Margin="3" MouseUp="SomethingClicked">
    <Grid.RowDefinitions>
      <RowDefinition Height="Auto"></RowDefinition>
      <RowDefinition Height="*"></RowDefinition>
      <RowDefinition Height="Auto"></RowDefinition>
      <RowDefinition Height="Auto"></RowDefinition>
    </Grid.RowDefinitions>

    <Label Margin="5" Grid.Row="0" HorizontalAlignment="Left"
     Background="AliceBlue" BorderBrush="Black" BorderThickness="1"
     MouseUp="SomethingClicked">
      <StackPanel MouseUp="SomethingClicked">
        <TextBlock Margin="3"
         MouseUp="SomethingClicked">
         Image and text label</TextBlock>
        <Image Source="happyface.jpg" Stretch="None"
         MouseUp="SomethingClicked" />
        <TextBlock Margin="3"
         MouseUp="SomethingClicked">
         Courtesy of the StackPanel</TextBlock>
      </StackPanel>
    </Label>

    <ListBox Grid.Row="1" Margin="5" Name="lstMessages"></ListBox>
    <CheckBox Grid.Row="2"  Margin="5" Name="chkHandle">
     Handle first event</CheckBox>
    <Button Grid.Row="3" Margin="5" Padding="3" HorizontalAlignment="Right"
     Name="cmdClear" Click="cmdClear_Click">Clear List</Button>
  </Grid>
</Window>
```

The SomethingClicked() method simply examines the properties of the RoutedEventArgs object and adds a message to the list box:

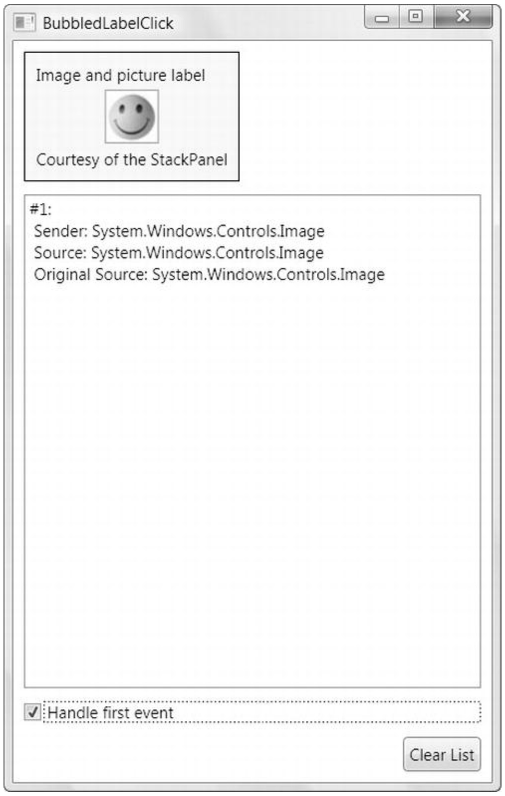
```
protected int eventCounter = 0;

private void SomethingClicked(object sender, RoutedEventArgs e)
{
    eventCounter++;
    string message = "#" + eventCounter.ToString() + ":\r\n" +
      " Sender: " + sender.ToString() + "\r\n" +
      " Source: " + e.Source + "\r\n" +
      " Original Source: " + e.OriginalSource;
    lstMessages.Items.Add(message);
    e.Handled = (bool)chkHandle.IsChecked;
}
```

■ **Note** Technically, the MouseUp event provides a MouseButtonEventArgs object with additional information about the mouse state at the time of the event. However, the MouseButtonEventArgs object derives from MouseEventArgs, which in turn derives from RoutedEventArgs. As a result, it's possible to use RoutedEventArgs when declaring the event handler (as shown here) if you don't need additional information about the mouse.

There's one other detail in this example. If you've selected the chkHandle check box, the SomethingClicked() method sets the RoutedEventArgs.Handled property to true, which stops the event-bubbling sequence the first time an event occurs. As a result, you'll see only the first event appear in the list, as shown in Figure 5-2.

■ **Note** There's an extra cast required here because the CheckBox.IsChecked property is a nullable Boolean value (a *bool?* rather than a *bool*). The null value represents an indeterminate state for the check box, which means it's neither checked or unchecked. This feature isn't used in this example, so a simple cast solves the problem.



```
#1:
 Sender: System.Windows.Controls.Image
 Source: System.Windows.Controls.Image
 Original Source: System.Windows.Controls.Image
```

Figure 5-2. *Marking an event as handled*

Because the SomethingClicked() method handles the MouseUp event that's fired by the Window, you'll be able to intercept clicks on the list box and the blank window surface. However, the MouseUp event doesn't fire when you click the Clear button (which removes all the list box entries). That's because the button includes an interesting bit of code that suppresses the MouseUp event and raises a higher-level Click event. At the same time, the Handled flag is set to true, which prevents the MouseUp event from going any further.

■ **Tip** Most WPF elements don't expose a Click event. Instead, they include the more straightforward MouseDown and MouseUp events. Click is reserved for button-based controls.

Handling a Suppressed Event

Interestingly, there *is* a way to receive events that are marked as handled. Instead of attaching the event handler through XAML, you must use the AddHandler() method described earlier. The AddHandler() method provides an overload that accepts a Boolean value for its third parameter. Set this to true, and you'll receive the event even if the Handled flag has been set:

```
cmdClear.AddHander(UIElement.MouseUpEvent,
  new MouseButtonEventHandler(cmdClear_MouseUp), true);
```

This is rarely a good design decision. The button is designed to suppress the MouseUp event for a reason: to prevent possible confusion. After all, it's a common Windows convention that buttons can be "clicked" with the keyboard in several ways. If you make the mistake of handling the MouseUp event in a Button instead of the Click event, your code will respond only to mouse clicks, not the equivalent keyboard actions.

Attached Events

The fancy label example is a fairly straightforward example of event bubbling because all the elements support the MouseUp event. However, many controls have their own more specialized events. The button is one example—it adds a Click event that isn't defined by any base class.

This introduces an interesting dilemma. Imagine that you wrap a stack of buttons in a StackPanel. You want to handle all the button clicks in one event handler. The crude approach is to attach the Click event of each button to the same event handler. But the Click event supports event bubbling, which gives you a better option. You can handle all the button clicks by handling the Click event at a higher level (such as the containing StackPanel).

Unfortunately, this apparently obvious code doesn't work:

```
<StackPanel Click="DoSomething" Margin="5">
  <Button Name="cmd1">Command 1</Button>
  <Button Name="cmd2">Command 2</Button>
  <Button Name="cmd3">Command 3</Button>
  ...
</StackPanel>
```

The problem is that the StackPanel doesn't include a Click event, so this is interpreted by the XAML parser as an error. The solution is to use a different attached-event syntax in the form *ClassName.EventName*. Here's the corrected example:

```
<StackPanel Button.Click="DoSomething" Margin="5">
  <Button Name="cmd1">Command 1</Button>
  <Button Name="cmd2">Command 2</Button>
  <Button Name="cmd3">Command 3</Button>
  ...
</StackPanel>
```

Now your event handler receives the click for all contained buttons.

■ **Note** The Click event is actually defined in the ButtonBase class and inherited by the Button class. If you attach an event handler to ButtonBase.Click, that event handler will be used when any ButtonBase-derived control is clicked (including the Button, RadioButton, and CheckBox classes). If you attach an event handler to Button.Click, it's used only for Button objects.

You can wire up an attached event in code, but you need to use the UIElement.AddHandler() method rather than the += operator syntax. Here's an example (which assumes that the StackPanel has been given the name pnlButtons):

```
pnlButtons.AddHandler(Button.Click, new RoutedEventHandler(DoSomething));
```

In the DoSomething() event handler, you have several options for determining which button fired the event. You can compare its text (which will cause problems for localization) or its name (which is fragile because you won't catch mistyped names when you build the application). The best approach is to make sure each button has a Name property set in XAML so that you can access the corresponding object through a field in your window class and compare that reference with the event sender. Here's an example:

```
private void DoSomething(object sender, RoutedEventArgs e)
{
    if (e.Source == cmd1)
    { ... }
    else if (e.Source == cmd2)
    { ... }
    else if (e.Source == cmd3)
    { ... }
}
```

Another option is to simply send a piece of information along with the button that you can use in your code. For example, you could set the Tag property of each button, as shown here:

```
<StackPanel Button.Click="DoSomething" Margin="5">
  <Button Name="cmd1" Tag="The first button.">Command 1</Button>
  <Button Name="cmd2" Tag="The second button.">Command 2</Button>
  <Button Name="cmd3" Tag="The third button.">Command 3</Button>
  ...
</StackPanel>
```

You can then access the Tag property in your code:

```
private void DoSomething(object sender, RoutedEventArgs e)
{
    object tag = ((FrameworkElement)sender).Tag;
    MessageBox.Show((string)tag);
}
```

Tunneling Events

Tunneling events work the same as bubbling events but in the opposite direction. For example, if MouseUp was a tunneled event (which it isn't), clicking the image in the fancy label example would cause MouseUp to fire first in the window, then in the Grid, then in the StackPanel, and so on, until it reaches the actual source, which is the image in the label.

Tunneling events are easy to recognize because they begin with the word *Preview*. Furthermore, WPF usually defines bubbling and tunneling events in pairs. That means if you find a bubbling MouseUp event, you can probably also find a tunneling PreviewMouseUp event. The tunneling event always fires before the bubbling event, as shown in Figure 5-3.

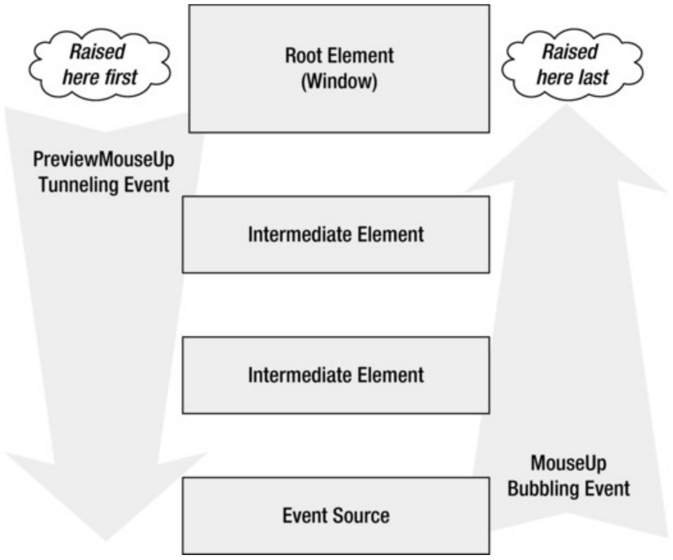
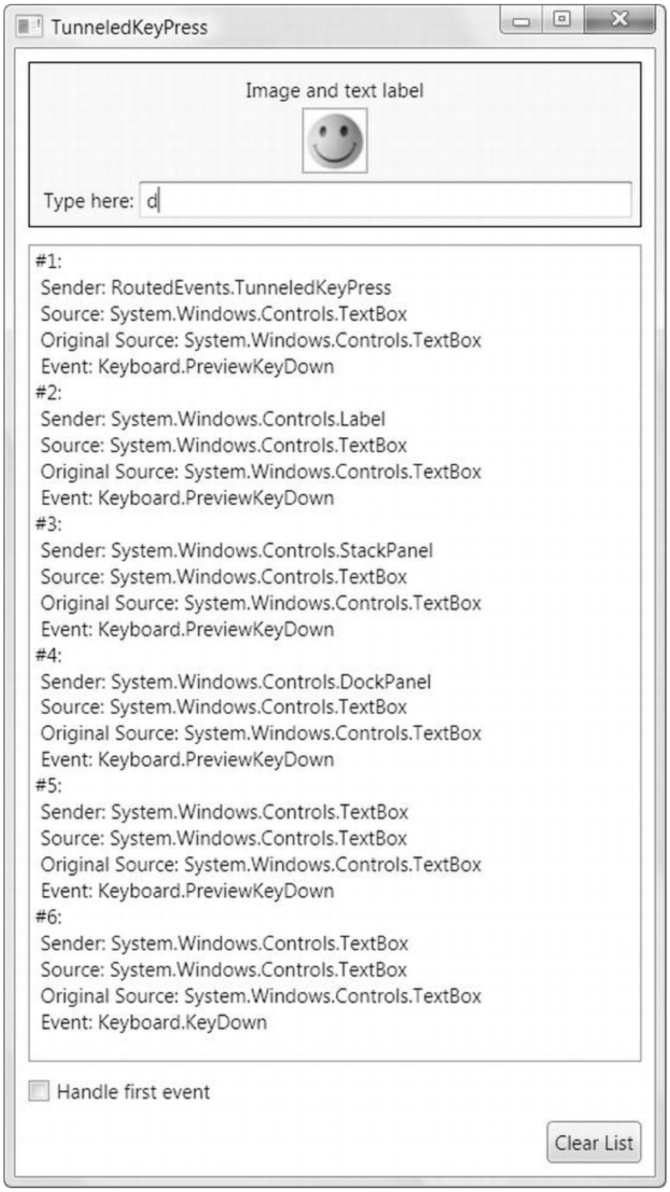


Figure 5-3. *Tunneling and bubbling events*

To make life more interesting, if you mark the tunneling event as handled, the bubbling event won't occur. That's because the two events share the same instance of the RoutedEventArgs class.

Tunneling events are useful if you need to perform some preprocessing that acts on certain keystrokes or filters out certain mouse actions. Figure 5-4 shows an example that tests tunneling with the PreviewKeyDown event When you press a key in the text box, the event is fired first in the window and then down through the hierarchy. And if you mark the PreviewKeyDown event as handled at any point, the bubbling KeyDown event won't occur.



```
#1:
 Sender: RoutedEvents.TunneledKeyPress
 Source: System.Windows.Controls.TextBox
 Original Source: System.Windows.Controls.TextBox
 Event: Keyboard.PreviewKeyDown
#2:
 Sender: System.Windows.Controls.Label
 Source: System.Windows.Controls.TextBox
 Original Source: System.Windows.Controls.TextBox
 Event: Keyboard.PreviewKeyDown
#3:
 Sender: System.Windows.Controls.StackPanel
 Source: System.Windows.Controls.TextBox
 Original Source: System.Windows.Controls.TextBox
 Event: Keyboard.PreviewKeyDown
#4:
 Sender: System.Windows.Controls.DockPanel
 Source: System.Windows.Controls.TextBox
 Original Source: System.Windows.Controls.TextBox
 Event: Keyboard.PreviewKeyDown
#5:
 Sender: System.Windows.Controls.TextBox
 Source: System.Windows.Controls.TextBox
 Original Source: System.Windows.Controls.TextBox
 Event: Keyboard.PreviewKeyDown
#6:
 Sender: System.Windows.Controls.TextBox
 Source: System.Windows.Controls.TextBox
 Original Source: System.Windows.Controls.TextBox
 Event: Keyboard.KeyDown
```

Figure 5-4. *A tunneled key press*

■ **Tip** Be careful about marking a tunneling event as handled. Depending on the way the control is written, this may prevent the control from handling its own event (the related bubbling event) to perform some task or update its state.

IDENTIFYING THE ROUTING STRATEGY OF AN EVENT

Clearly the different routing strategies affect how you'll use an event. But how do you determine what type of routing a given event uses?

Tunneling events are straightforward. By .NET convention, a tunneling event always begins with the word *Preview* (as in PreviewKeyDown). However, there's no similar mechanism to distinguish bubbling events from direct events. For developers exploring WPF, the easiest approach is to find the event in the Visual Studio documentation. You'll see Routed Event Information that indicates the static field for the event, the type of routing, and the event signature.

You can get the same information programmatically by examining the static field for the event. For example, the ButtonBase.ClickEvent.RoutingStrategy property provides an enumerated value that tells you what type of routing the Click event uses.

WPF Events

Now that you've learned how WPF events work, it's time to consider the rich variety of events that you can respond to in your code. Although every element exposes a dizzying array of events, the most important events usually fall into one of five categories:

- *Lifetime events*: These events occur when the element is initialized, loaded, or unloaded.
- *Mouse events*: These events are the result of mouse actions.
- *Keyboard events*: These events are the result of keyboard actions (such as key presses).
- *Stylus events*: These events are the result of using the pen-like stylus, which takes the place of a mouse on a Tablet PC.
- *Multitouch events*: These events are the result of touching down with one or more fingers on a multitouch screen. They're supported only in Windows 7 and Windows 8.

Taken together, mouse, keyboard, stylus, and multitouch events are known as *input events*.

Lifetime Events

All elements raise events when they are first created and when they are released. You can use these events to initialize a window. Table 5-2 lists these events which are defined in the FrameworkElement class.

***Table 5-2.** Lifetime Events for All Elements*

Name	Description
Initialized	Occurs after the element is instantiated and its properties have been set according to the XAML markup. At this point, the element is initialized, but other parts of the window may not be. Also, styles and data binding haven't been applied yet. At this point, the IsInitialized property is true. Initialized is an ordinary .NET event, not a routed event.
Loaded	Occurs after the entire window has been initialized and styles and data binding have been applied. This is the last stop before the element is rendered. At this point, the IsLoaded property is true.
Unloaded	Occurs when the element has been released, either because the containing window has been closed or the specific element has been removed from the window.

To understand how the Initialized and Loaded events relate, it helps to consider the rendering process. The FrameworkElement implements the ISupportInitialize interface, which provides two methods for controlling the initialization process. The first, BeginInit(), is called immediately after the element is instantiated. After BeginInit() is called, the XAML parser sets all the element properties (and adds any content). The second method, EndInit(), is called when initialization is complete, at which point the Initialized event fires.

■ **Note** This is a slight simplification. The XAML parser takes care of calling the BeginInit() and EndInit() methods, as it should. However, if you create an element by hand and add it to a window, it's unlikely that you'll use this interface. In this case, the element raises the Initialized event after you add it to the window, just before the Loaded event.

When you create a window, each branch of elements is initialized in a bottom-up fashion. That means deeply nested elements are initialized before their containers. When the Initialized event fires, you are guaranteed that the tree of elements from the current element down is completely initialized. However, the element that *contains* your element probably isn't initialized, and you can't assume that any other part of the window is initialized.

After each element is initialized, it's also laid out in its container, styled, and bound to a data source, if required. After the Initialized event fires for the window, it's time to go on to the next stage.

After the initialization process is complete, the Loaded event is fired. The Loaded event follows the reverse path of the Initialized event—in other words, the containing window fires the Loaded event first, followed by more deeply nested elements. When the Loaded event has fired for all elements, the window becomes visible and the elements are rendered.

The lifetime events listed in Table 5-2 don't tell the whole story. The containing window also has its own more specialized lifetime events. These events are listed in Table 5-3.

***Table 5-3.** Lifetime Events for the Window Class*

Name	Description
SourceInitialized	Occurs when the HwndSource property of the window is acquired (but before the window is made visible). The HwndSource is a window handle that you may need to use if you're calling legacy functions in the Win32 API.
ContentRendered	Occurs immediately after the window has been rendered for the first time. This isn't a good place to perform any changes that might affect the visual appearance of the window, or you'll force a second render operation. (Use the Loaded event instead.) However, the ContentRendered event does indicate that your window is fully visible and ready for input.
Activated	Occurs when the user switches to this window (for example, from another window in your application or from another application). Activated also fires when the window is loaded for the first time. Conceptually, the Activated event is the window equivalent of a control's GotFocus event.
Deactivated	Occurs when the user switches away from this window (for example, by moving to another window in your application or another application). Deactivated also fires when the window is closed by a user, after the Closing event but before Closed. Conceptually, the Deactivated event is the window equivalent of a control's LostFocus event.
Closing	Occurs when the window is closed, either by a user action or programmatically by using the Window.Close() method or the Application.Shutdown() method. The Closing event gives you the opportunity to cancel the operation and keep the window open by setting the CancelEventArgs.Cancel property to true. However, you won't receive the Closing event if your application is ending because the user is shutting down the computer or logging off. To deal with these possibilities, you need to handle the Application.SessionEnding event described in Chapter 7.
Closed	Occurs after the window has been closed. However, the element objects are still accessible, and the Unloaded event hasn't fired yet. At this point, you can perform cleanup, write settings to a persistent storage place (such as a configuration file or the Windows Registry), and so on.

If you're simply interested in performing first-time initializing for your controls, the best time to take care of this task is when the Loaded event fires. Usually, you can perform all your initialization in one place, which is typically an event handler for the Window.Loaded event.

■ **Tip** You can also use the window constructor to perform your initialization (just add your code immediately after the InitializeComponent() call). However, it's always better to use the Loaded event. That's because if an exception occurs in the constructor of the Window, it's thrown while the XAML parser is parsing the page. As a result, your exception is wrapped in an unhelpful XamlParseException object (with the original exception in the InnerException property).

Input Events

Input events are events that occur when the user interacts with some sort of peripheral hardware, such as a mouse, keyboard, stylus, or multitouch screen. Input events can pass along extra information by using a custom event argument class that derives from InputEventArgs. Figure 5-5 shows the inheritance hierarchy.

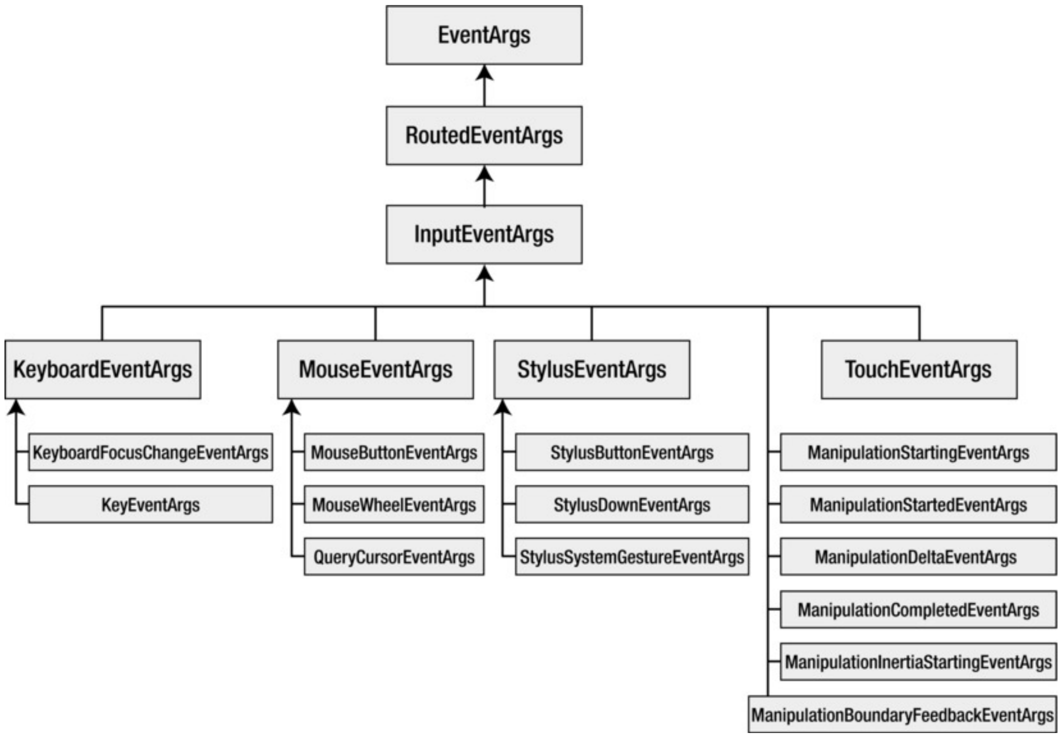

EventArgs
RoutedEventArgs
InputEventArgs
KeyboardEventArgs
MouseEventArgs
StylusEventArgs
TouchEventArgs
KeyboardFocusChangeEventArgs
KeyEventArgs
MouseButtonEventArgs
MouseWheelEventArgs
QueryCursorEventArgs
StylusButtonEventArgs
StylusDownEventArgs
StylusSystemGestureEventArgs
ManipulationStartingEventArgs
ManipulationStartedEventArgs
ManipulationDeltaEventArgs
ManipulationCompletedEventArgs
ManipulationInertiaStartingEventArgs
ManipulationBoundaryFeedbackEventArgs

Figure 5-5. *The EventArgs classes for input events*

The InputEventArgs class adds just two properties: Timestamp and Device. Timestamp provides an integer that indicates when the event occurred as a number of milliseconds. (The actual time that this represents isn't terribly important, but you can compare different timestamp values to determine what event took place first. Larger timestamps signify more-recent events.) The Device returns an object that provides more information about the device that triggered the event, which could be the mouse, the keyboard, or the stylus. Each of these three possibilities is represented by a different class, all of which derive from the abstract System.Windows.Input.InputDevice class.

In the following sections, you'll take a closer look at how you handle mouse, keyboard, and multitouch actions in a WPF application.

Keyboard Input

When the user presses a key, a sequence of events unfolds. Table 5-4 lists these events in the order that they occur.

Table 5-4. *Keyboard Events for All Elements (in the Order They Occur)*

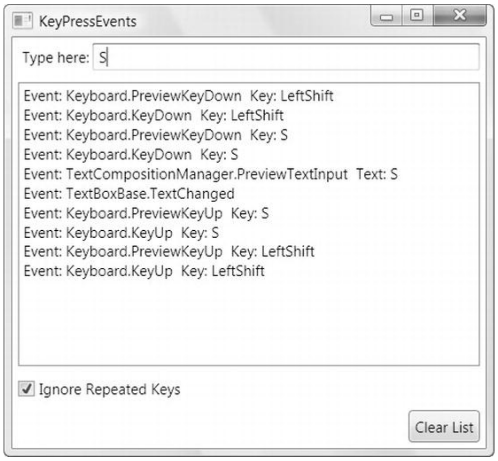
Name	Routing Type	Description
PreviewKeyDown	Tunneling	Occurs when a key is pressed.
KeyDown	Bubbling	Occurs when a key is pressed.
PreviewTextInput	Tunneling	Occurs when a keystroke is complete and the element is receiving the text input. This event isn't fired for keystrokes that don't result in text being "typed" (for example, it doesn't fire when you press Ctrl, Shift, Backspace, the arrow keys, the function keys, and so on).
TextInput	Bubbling	Occurs when a keystroke is complete and the element is receiving the text input. This event isn't fired for keystrokes that don't result in text.
PreviewKeyUp	Tunneling	Occurs when a key is released.
KeyUp	Bubbling	Occurs when a key is released.

Keyboard handling is never quite as straightforward as it seems. Some controls may suppress some of these events so they can perform their own more specialized keyboard handling. The most notorious example is the TextBox control, which suppresses the TextInput event. The TextBox also suppresses the KeyDown event for some keystrokes, such as the arrow keys. In cases like these, you can usually still use the tunneling events (PreviewTextInput and PreviewKeyDown).

The TextBox control also adds one new event, named TextChanged. This event fires immediately after a keystroke causes the text in the text box to change. At this point, the new text is already visible in the text box, so it's too late to prevent a keystroke you don't want.

Handling a Key Press

The best way to understand the key events is to use a sample program such as the one shown in Figure 5-6. It monitors a text box for all the possible key events and reports when they occur. Figure 5-6 shows the result of typing a capital *S* in a text box.



```
Event: Keyboard.PreviewKeyDown  Key: LeftShift
Event: Keyboard.KeyDown  Key: LeftShift
Event: Keyboard.PreviewKeyDown  Key: S
Event: Keyboard.KeyDown  Key: S
Event: TextCompositionManager.PreviewTextInput  Text: S
Event: TextBoxBase.TextChanged
Event: Keyboard.PreviewKeyUp  Key: S
Event: Keyboard.KeyUp  Key: S
Event: Keyboard.PreviewKeyUp  Key: LeftShift
Event: Keyboard.KeyUp  Key: LeftShift
```

Figure 5-6. *Watching the keyboard*

This example illustrates an important point. The PreviewKeyDown and KeyDown events fire every time a key is pressed. However, the TextInput event fires only when a character is “typed” into an element. This action may actually involve multiple key presses. In the example in Figure 5-5, two key presses are needed to create the capital letter *S*. First, the Shift key is pressed, followed by the S key. As a result, you’ll see two KeyDown and KeyUp events but only one TextInput event.

The PreviewKeyDown, KeyDown, PreviewKeyUp, and KeyUp events all provide the same information through the KeyEventArgs object. The most important detail is the Key property, which returns a value from the System.Windows.Input.Key enumeration that identifies the key that was pressed or released. Here’s the event handler that handles key events for the example in Figure 5-6:

```
private void KeyEvent(object sender, KeyEventArgs e)
{
    string message = "Event: " + e.RoutedEvent + " " +
      " Key: " + e.Key;
    lstMessages.Items.Add(message);
}
```

The Key value doesn’t take into account the state of any other keys. For example, it doesn’t matter whether the Shift key is currently pressed when you press the S key; either way you’ll get the same Key value (Key.S).

There’s one more wrinkle. Depending on your Windows keyboard settings, pressing a key causes the keystroke to be repeated after a short delay. For example, holding down the S key obviously puts a stream of *S* characters in the text box. Similarly, pressing the Shift key causes multiple keystrokes and a series of KeyDown events. In a real-world test in which you press Shift+S, your text box will actually fire a series of KeyDown events for the Shift key, followed by a KeyDown event for the S key, a TextInput event (or TextChanged event in the case of a text box), and then a KeyUp event for the Shift and S keys. If you want to ignore these repeated Shift keys, you can check if a keystroke is the result of a key that’s being held down by examining the KeyEventArgs.IsRepeat property, as shown here:

```
if ((bool)chkIgnoreRepeat.IsChecked && e.IsRepeat) return;
```

■ **Tip** The PreviewKeyDown, KeyDown, PreviewKeyUp, and KeyUp events are best for writing low-level keyboard handling (which you’ll rarely need outside a custom control) and handling special keystrokes, such as the function keys.

After the KeyDown event occurs, the PreviewTextInput event follows. (The TextInput event doesn’t occur, because the TextBox suppresses this event.) At this point, the text has not yet appeared in the control.

The TextInput event provides your code with a TextCompositionEventArgs object. This object includes a Text property that gives you the processed text that’s about to be received by the control. Here’s the code that adds this text to the list shown in Figure 5-6:

```
private void TextInput(object sender, TextCompositionEventArgs e)
{
    string message = "Event: " + e.RoutedEvent + " " +
      " Text: " + e.Text;
    lstMessages.Items.Add(message);
}
```

Ideally, you'd use PreviewTextInput to perform validation in a control such as the TextBox. For example, if you're building a numeric-only text box, you could make sure that the current keystroke isn't a letter and set the Handled flag if it is. Unfortunately, the PreviewTextInput event doesn't fire for some keys that you may need to handle. For example, if you press the spacebar in a text box, you'll bypass PreviewTextInput altogether. That means you also need to handle the PreviewKeyDown event.

Unfortunately, it's difficult to write robust validation logic in a PreviewKeyDown event handler because all you have is the Key value, which is a fairly low-level piece of information. For example, the Key enumeration distinguishes between the numeric key pad and the number keys that appear just above the letters on a typical keyboard. That means depending on how you press the number 9, you might get a value of Key.D9 or Key.NumPad9. Checking for all the allowed key values is tedious, to say the least.

One option is to use the KeyConverter to convert the Key value into a more useful string. For example, using KeyConverter.ConvertToString() on both Key.D9 and Key.NumPad9 returns "9" as a string. If you just use the Key.ToString() conversion, you'll get the much less useful enumeration name (either "D9" or "NumPad9"):

```
KeyConverter converter = new KeyConverter();
string key = converter.ConvertToString(e.Key);
```

However, even using the KeyConverter is a bit awkward because you'll end up with longer bits of text (such as "Backspace") for keystrokes that don't result in text input.

The best compromise is to handle both PreviewTextInput (which takes care of most of the validation) and use PreviewKeyDown for keystrokes that don't raise PreviewTextInput in the text box (such as the spacebar). Here's a simple solution that does it:

```
private void pnl_PreviewTextInput(object sender, TextCompositionEventArgs e)
{
    short val;
    if (!Int16.TryParse(e.Text, out val))
    {
        // Disallow non-numeric key presses.
        e.Handled = true;
    }
}

private void pnl_PreviewKeyDown(object sender, KeyEventArgs e)
{
    if (e.Key == Key.Space)
    {
        // Disallow the space key, which doesn't raise a PreviewTextInput event.
        e.Handled = true;
    }
}
```

You can attach these event handlers to a single text box, or you can wire them up to a container (such as a StackPanel that contains several numeric-only text boxes) for greater efficiency.

■ **Note** This key-handling behavior may seem unnecessarily awkward (and it is). One of the reasons that the TextBox doesn't provide better key handling is that WPF focuses on data binding, a feature that lets you wire up controls such as the TextBox to custom objects. When you use this approach, validation is usually provided by the bound object, errors are signaled by an exception, and bad data triggers an error message that appears somewhere in the user interface. Unfortunately, there's no easy way (at present) to combine the useful, high-level data-binding feature with the lower-level keyboard handling that would be necessary to prevent the user from typing invalid characters altogether.

Focus

In the Windows world, a user works with one control at a time. The control that is currently receiving the user's key presses is the control that has *focus*. Sometimes this control is drawn slightly differently. For example, the WPF button uses blue shading to show that it has the focus.

For a control to be able to accept the focus, its Focusable property must be set to true. This is the default for all controls.

Interestingly enough, the Focusable property is defined as part of the UIElement class, which means that other noncontrol elements can also be focusable. Usually, in noncontrol classes, Focusable will be false by default. However, you can set it to true. Try this with a layout container such as the StackPanel—when it receives the focus, a dotted border will appear around the panel's edge.

To move the focus from one element to another, the user can click the mouse or use the Tab and arrow keys. In previous development frameworks, programmers have been forced to take great care to make sure that the Tab key moves focus in a logical manner (generally from left to right and then down the window) and that the correct control has focus when the window first appears. In WPF, this extra work is seldom necessary because WPF uses the hierarchical layout of your elements to implement a tabbing sequence. Essentially, when you press the Tab key, you'll move to the first child in the current element or, if the current element has no children, to the next child at the same level. For example, if you tab through a window with two StackPanel containers, you'll move through all the controls in the first StackPanel and then through all the controls in the second container.

If you want to take control of tab sequence, you can set the TabIndex property for each control to place it in numerical order. The control with a TabIndex of 0 gets the focus first, followed by the next highest TabIndex value (for example, 1, then 2, then 3, and so on). If more than one element has the same TabIndex value, WPF uses the automatic tab sequence, which means it jumps to the nearest subsequent element.

■ **Tip** By default, the TabIndex property for all controls is set to Int32.MaxValue. That means you can designate a specific control as the starting point for a window by setting its TabIndex to 0 but rely on automatic navigation to guide the user through the rest of the window from that starting point, according to the order that your elements are defined.

The TabIndex property is defined in the Control class, along with an IsTabStop property. You can set IsTabStop to false to prevent a control from being included in the tab sequence. The difference between IsTabStop and Focusable is that a control with an IsTabStop value of false can still get the focus in another way—either programmatically (when your code calls its Focus() method) or by a mouse click.

Controls that are invisible or disabled ("grayed out") are generally skipped in the tab order and are not activated regardless of the TabIndex, IsTabStop, and Focusable settings. To hide or disable a control, you set the Visibility and IsEnabled properties, respectively.

Getting Key State

When a key press occurs, you often need to know more than just what key was pressed. It's also important to find out what other keys were held down at the same time. That means you might want to investigate the state of other keys, particularly modifiers such as Shift, Ctrl, and Alt.

The key events (PreviewKeyDown, KeyDown, PreviewKeyUp, and KeyUp) make this information easy to get. First, the KeyEventArgs object includes a KeyStates property that reflects the property of the key that triggered the event. More usefully, the KeyboardDevice property provides the same information for any key on the keyboard.

Not surprisingly, the KeyboardDevice property provides an instance of the KeyboardDevice class. Its properties include information about which element currently has the focus (FocusedElement) and what modifier keys were pressed when the event occurred (Modifiers). The modifier keys include Shift, Ctrl, and Alt, and you can check their status by using bitwise logic like this:

```
if ((e.KeyboardDevice.Modifiers & ModifierKeys.Control) == ModifierKeys.Control)
{
    lblInfo.Text = "You held the Control key.";
}
```

The KeyboardDevice property also provides a few handy methods, listed in Table 5-5. For each of these methods, you pass in a value from the Key enumeration.

Table 5-5. *KeyboardDevice Methods*

Name	Description
IsKeyDown()	Tells you whether this key was pressed down when the event occurred.
IsKeyUp()	Tells you whether this key was up (not pressed) when the event occurred.
IsKeyToggled()	Tells you whether this key was in a "switched on" state when the event occurred. This has a meaning only for keys that can be toggled on or off, such as Caps Lock, Scroll Lock, and Num Lock.
GetKeyStates()	Returns one or more values from the KeyStates enumeration that tell you whether this key is currently up, pressed, or in a toggled state. This method is essentially the same as calling both IsKeyDown() and IsKeyToggled() on the same key.

When you use the KeyEventArgs.KeyboardDevice property, your code gets the *virtual key state*. This means it gets the state of the keyboard at the time the event occurred. This is not necessarily the same as the current keyboard state. For example, consider what happens if the user types faster than your code executes. Each time your KeyPress event fires, you'll have access to the keystroke that fired the event, not the typed-ahead characters. This is almost always the behavior you want.

However, you aren't limited to getting key information in the key events. You can also get the state of the keyboard at any time. The trick is to use the Keyboard class, which is very similar to KeyboardDevice except it's made up of static members. Here's an example that uses the Keyboard class to check the current state of the left Shift key:

```
if (Keyboard.IsKeyDown(Key.LeftShift))
{
    lblInfo.Text = "The left Shift is held down.";
}
```

■ **Note** The Keyboard class also has methods that allow you to attach application-wide keyboard event handlers, such as AddKeyDownHandler() and AddKeyUpHandler(). However, these methods aren't recommended. A better approach to implementing application-wide functionality is to use the WPF command system, as described in Chapter 9.

Mouse Input

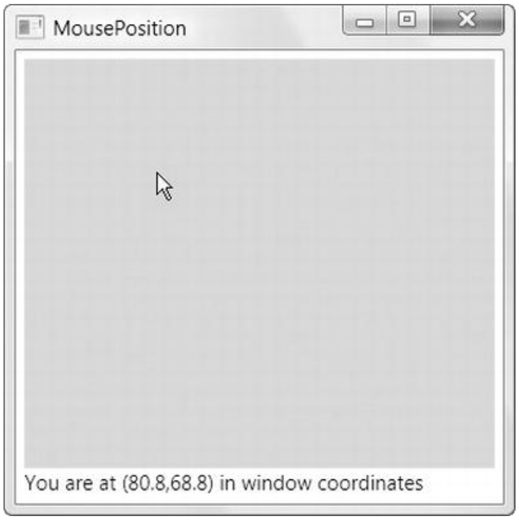
Mouse events perform several related tasks. The most fundamental mouse events allow you to react when the mouse is moved over an element. These events are MouseEnter (which fires when the mouse pointer moves over the element) and MouseLeave (which fires when the mouse pointer moves away). Both are *direct events*, which means they don't use tunneling or bubbling. Instead, they originate in one element and are raised by just that element. This makes sense because of the way controls are nested in a WPF window.

For example, if you have a StackPanel that contains a button and you move the mouse pointer over the button, the MouseEnter event will fire first for the StackPanel (as you enter its borders) and then for the button (as you move directly over it). As you move the mouse away, the MouseLeave event will fire first for the button and then for the StackPanel.

You can also react to two events that fire whenever the mouse moves: PreviewMouseMove (a tunneling event) and MouseMove (a bubbling event). All of these events provide your code with the same information: a MouseEventArgs object. The MouseEventArgs object includes properties that tell you the state that the mouse buttons were in when the event fired, and it includes a GetPosition() method that tells you the coordinates of the mouse in relation to an element of your choosing. Here's an example that displays the position of the mouse pointer in device-independent pixels relative to the form:

```
private void MouseMoved(object sender, MouseEventArgs e)
{
    Point pt = e.GetPosition(this);
    lblInfo.Text =
      String.Format("You are at ({0},{1}) in window coordinates",
      pt.X, pt.Y);
}
```

In this case, the coordinates are measured from the top-left corner of the client area (just below the title bar). Figure 5-7 shows this code in action.



You are at (80.8,68.8) in window coordinates

Figure 5-7. *Watching the mouse*

You'll notice that the mouse coordinates in this example are not whole numbers. That's because this screen capture was taken on a system running at 120 dpi, not the standard 96 dpi. As explained in Chapter 1, WPF automatically scales up its units to compensate, using more physical pixels. Because the size of a screen pixel no longer matches the size of the WPF unit system, the physical mouse position may be translated to a fractional number of WPF units, as shown here.

■ **Tip** The UIElement class also includes two useful properties that can help with mouse hit-testing. Use IsMouseOver to determine whether a mouse is currently over an element or one of its children, and use IsMouseDirectlyOver to find out whether the mouse is over an element but not one of its children. Usually, you won't read and act on these values in code. Instead, you'll use them to build style triggers that automatically change elements as the mouse moves over them. Chapter 11 demonstrates this technique.

Mouse Clicks

Mouse clicks unfold in a similar way to key presses. The difference is that there are distinct events for the left mouse button and the right mouse button. Table 5-6 lists these events in the order they occur. Along with these are two events that react to the mouse wheel: PreviewMouseWheel and MouseWheel.

Table 5-6. *Mouse Click Events for All Elements (in Order)*

Name	Routing Type	Description
PreviewMouseLeftButtonDown and PreviewMouseRightButtonDown	Tunneling	Occurs when a mouse button is pressed
MouseLeftButtonDown and MouseRightButtonDown	Bubbling	Occurs when a mouse button is pressed

PreviewMouseLeftButtonUp and PreviewMouseRightButtonUp	Tunneling	Occurs when a mouse button is released
MouseLeftButtonUp and MouseRightButtonUp	Bubbling	Occurs when a mouse button is released

All mouse button events provide a MouseButtonEventArgs object. The MouseButtonEventArgs class derives from MouseEventArgs (which means it includes the same coordinate and button state information), and it adds a few members. The less important of these are MouseButton (which tells you which button triggered the event) and ButtonState (which tells you whether the button was pressed or unpressed when the event occurred). The more interesting property is ClickCount, which tells you how many times the button was clicked, allowing you to distinguish single clicks (where ClickCount is 1) from double-clicks (where ClickCount is 2).

■ **Tip** Usually, Windows applications react when the mouse key is raised after being clicked (the "up" event rather than the "down" event).

Some elements add higher-level mouse events. For example, the Control class adds PreviewMouseDoubleClick and MouseDoubleClick events that take the place of the MouseLeftButtonUp event. Similarly, the Button class raises a Click event that can be triggered by the mouse or keyboard.

■ **Note** As with key press events, the mouse events provide information about where the mouse was and what buttons were pressed when the mouse event occurred. To get the current mouse position and mouse button state, you can use the static members of the Mouse class, which are similar to those of the MouseButtonEventArgs.

Capturing the Mouse

Ordinarily, every time an element receives a mouse button "down" event, it will receive a corresponding mouse button "up" event shortly thereafter. However, this isn't always the case. For example, if you click an element, hold down the mouse, and then move the mouse pointer off the element, the element won't receive the mouse up event.

In some situations, you may want to have a notification of mouse up events, even if they occur after the mouse has moved off your element. To do so, you need to *capture* the mouse by calling the Mouse.Capture() method and passing in the appropriate element. From that point on, you'll receive mouse down and mouse up events until you call Mouse.Capture() again and pass in a null reference. Other elements won't receive mouse events while the mouse is captured. That means the user won't be able to click buttons elsewhere in the window, click inside text boxes, and so on. Mouse capturing is sometimes used to implement draggable and resizable elements. You'll see an example with the custom-drawn resizable window in Chapter 23.

■ **Tip** When you call Mouse.Capture(), you can pass in an optional CaptureMode value as the second parameter. Ordinarily, when you call Mouse.Capture(), you use CaptureMode.Element, which means your element always receives the mouse events. However, you can use CaptureMode.SubTree to allow mouse events to pass through to the clicked element if that clicked element is a child of the element that's performing the capture. This makes sense if you're already using event bubbling or tunneling to watch mouse events in child elements.

In some cases, you may lose a mouse capture through no fault of your own. For example, Windows may free the mouse if it needs to display a system dialog box. You'll also lose the mouse capture if you don't free the mouse after a mouse up event occurs and the user carries on to click a window in another application. Either way, you can react to losing the mouse capture by handling the LostMouseCapture event for your element.

Although the mouse has been captured by an element, you won't be able to interact with other elements. (For example, you won't be able to click another element on your window.) Mouse capturing is generally used for short-term operations such as drag-and-drop.

■ **Note** Instead of using Mouse.Capture(), you can use two methods that are built into the UIElement class: CaptureMouse() and ReleaseMouseCapture(). Just call these methods on the appropriate element. The only limitation of this approach is that it doesn't allow you to use the CaptureMode.SubTree option.

Drag-and-Drop

Drag-and-drop operations (a technique for pulling information out of one place in a window and depositing it in another) aren't quite as common today as they were a few years ago. Programmers have gradually settled on other methods of copying information that don't require holding down the mouse button (a technique that many users find difficult to master). Programs that do support drag-and-drop often use it as a shortcut for advanced users, rather than a standard way of working.

Essentially, a drag-and-drop operation unfolds in three steps:

1. The user clicks an element (or selects a specific region inside it) and holds the mouse button down. At this point, some information is set aside, and a drag-and-drop operation begins.
2. The user moves the mouse over another element. If this element can accept the type of content that's being dragged (for example, a bitmap or a piece of text), the mouse cursor changes to a drag-and-drop icon. Otherwise, the mouse cursor becomes a circle with a line drawn through it.
3. When the user releases the mouse button, the element receives the information and decides what to do with it. The operation can be canceled by pressing the Esc key (without releasing the mouse button).

You can try the way drag-and-drop is supposed to work by adding two text boxes to a window, because the TextBox control has built-in logic to support drag-and-drop. If you select some text inside a text box, you can drag it to another text box. When you release the mouse button, the text will be moved. The same technique works between applications—for example, you can drag some text from a Word document and drop it into a WPF TextBox object, or vice versa.

Sometimes, you might want to allow drag and drop between elements that don't have the built-in functionality. For example, you might want to allow the user to drag content from a text box and drop it in a label. Or you might want to create the example shown in Figure 5-8, which allows a user to drag text from a Label or TextBox object and drop it into a different label. In this situation, you need to handle the drag-and-drop events.

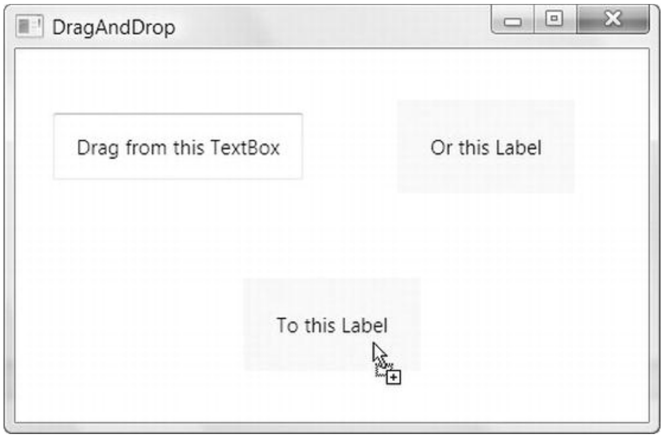


Figure 5-8. *Dragging content from one element to another*

■ **Note** The methods and events that are used for drag-and-drop operations are centralized in the System.Windows.DragDrop class. That way, any element can participate in a drag-and-drop operation by using this class.

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 formats.

Multitouch Input

Multitouch is a way of interacting with an application by touching a screen. What distinguishes multitouch input from old-fashioned pen input is that 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 touch screen 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.

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

Multitouch screens are nearly ubiquitous on smartphones and tablets. However, they're far less common on ordinary computers. Although hardware manufacturers have created touch-screen laptops and touch-screen LCD monitors, conventional laptops and monitors are still far more popular.

This presents a challenge for developers who want to experiment with multitouch applications. Currently, 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. The basic approach is to 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, and it doesn't work on all systems.

■ **Note** Although some applications may support multitouch on Windows Vista, the support that's built into WPF requires Windows 7 or Windows 8, 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 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 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 touch screen 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
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 includes 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, the 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.

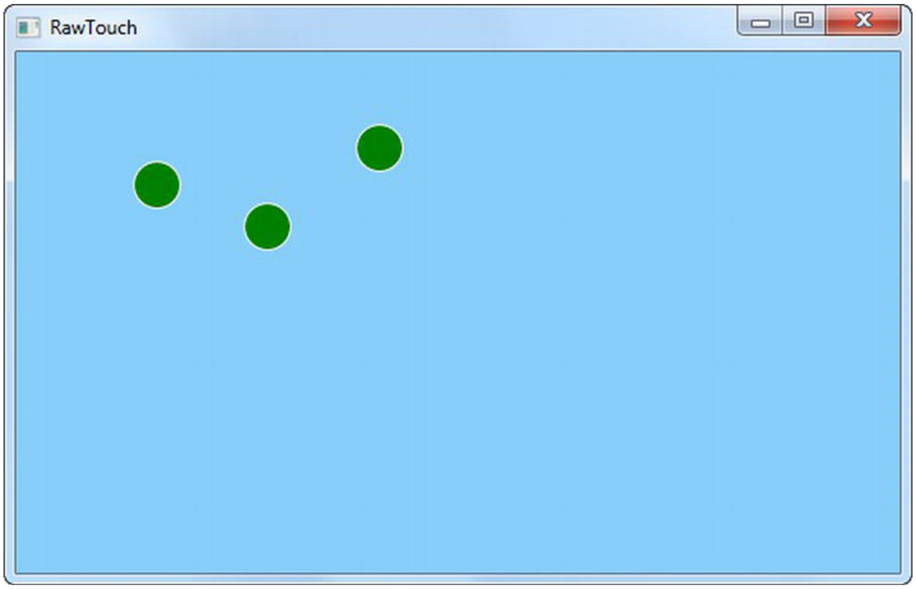


Figure 5-9. *Dragging circles with multitouch*

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 by 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 by 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 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.

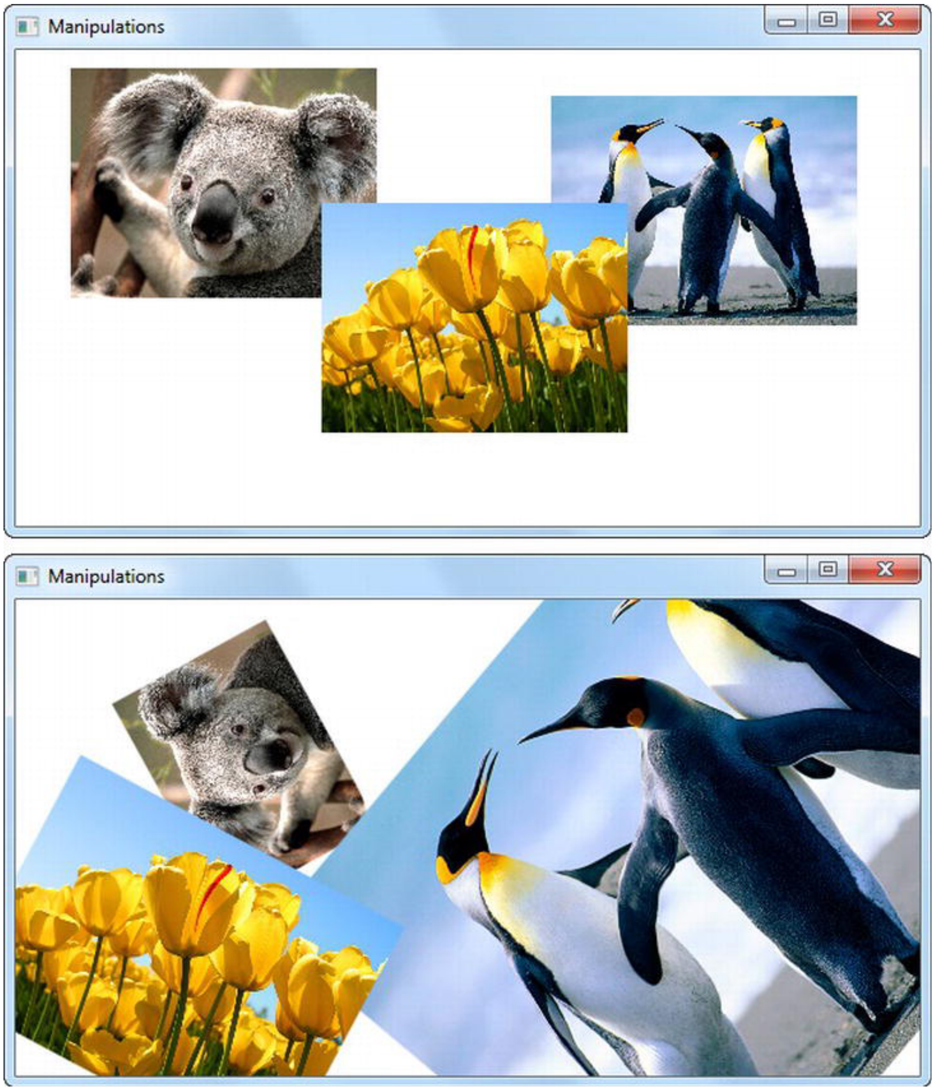


Figure 5-10. Before and after: three images manipulated with multitouch

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 by using a panning gesture, the image stops moving the moment the fingers are raised from the touch screen. 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 are 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 by 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.