There's one other detail in this example. If you've checked 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 nor unchecked. This feature isn't used in this example, so a simple cast solves the problem.

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** Unlike Windows Forms controls, 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 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 only used 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 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.

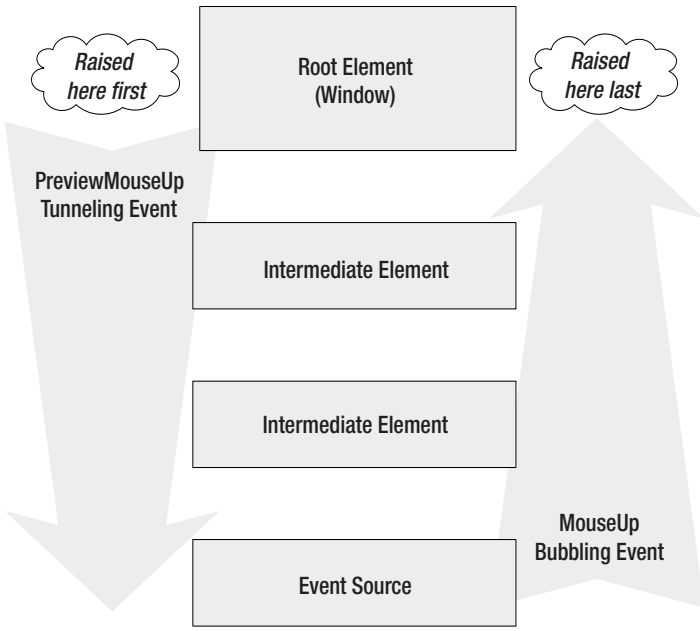

Figure 5-3. *Tunneling and bubbling events*

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.

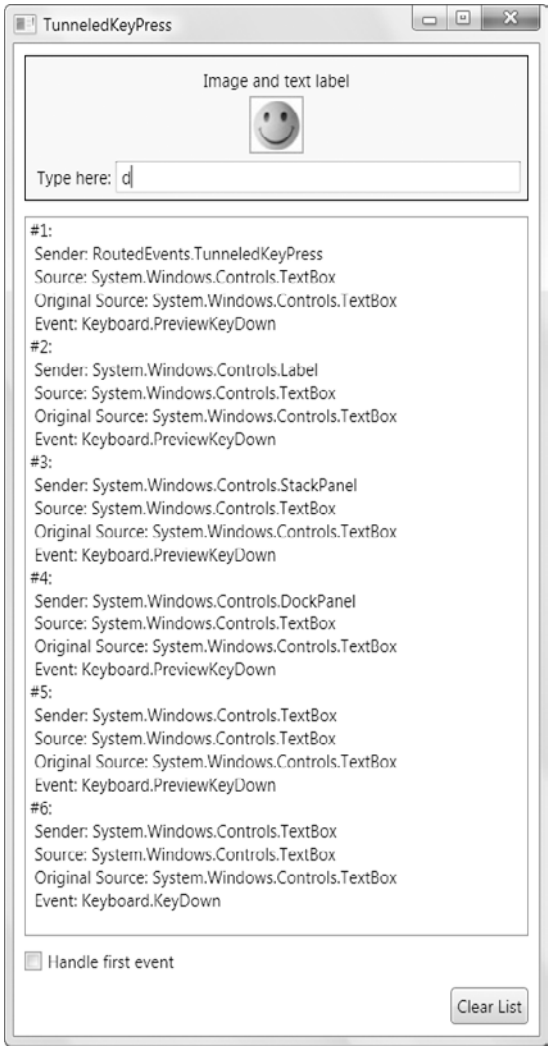

Figure 5-4. *A tunneled key press*

■ **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 *Preview*KeyDown). 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 only supported in Windows 7.

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 once 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.

Once 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 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 using a custom event argument class that derives from InputEventArgs. Figure 5-5 shows the inheritance hierarchy.

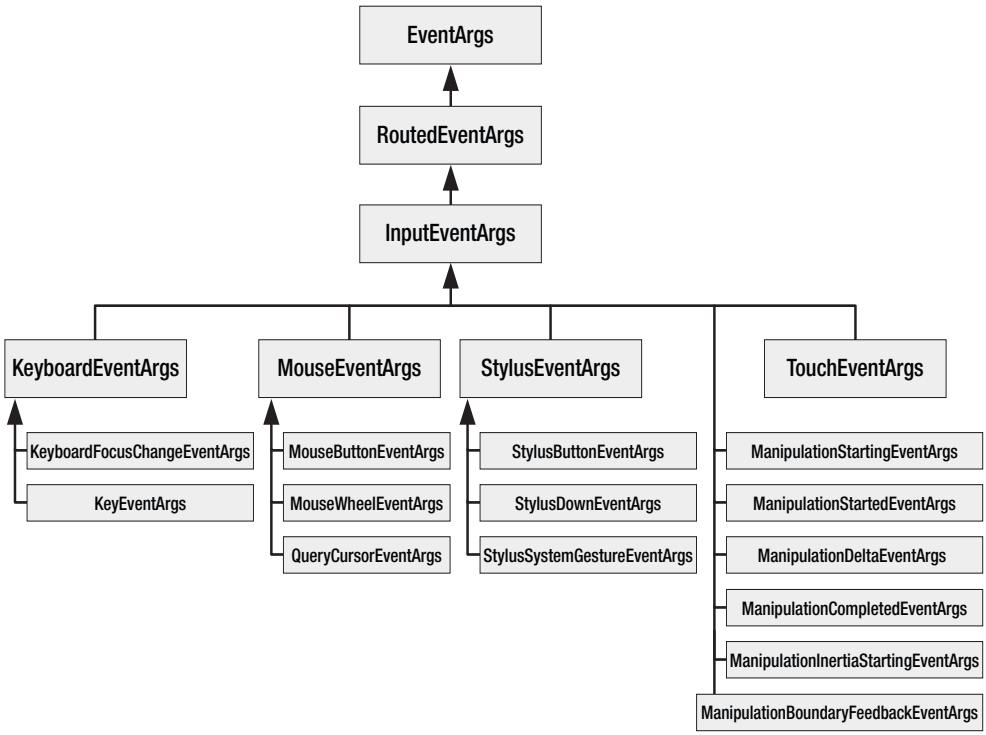

Figure 5-5. *The EventArgs classes for input events*

The InputEventArgs class adds just two properties: Timestamp and Device. The 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 time stamp values to determine what event took place first. Larger time stamps 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.

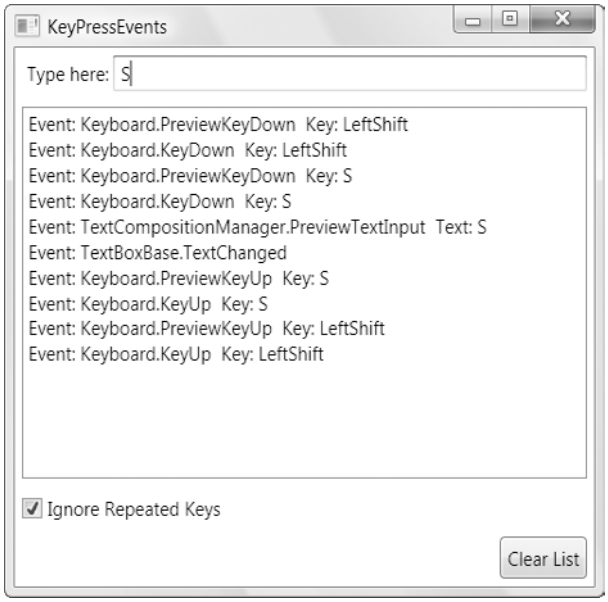

Figure 5-6. *Watching the keyboard*

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 where 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 of 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 the PreviewTextInput to perform validation in a control like 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 space key 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 space key). 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 because 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 right 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 using bitwise logic like this:

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

The KeyboardDevice also provides a few handy methods, as 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 only has a meaning 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 (once you enter its borders) and then for the button (once 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.

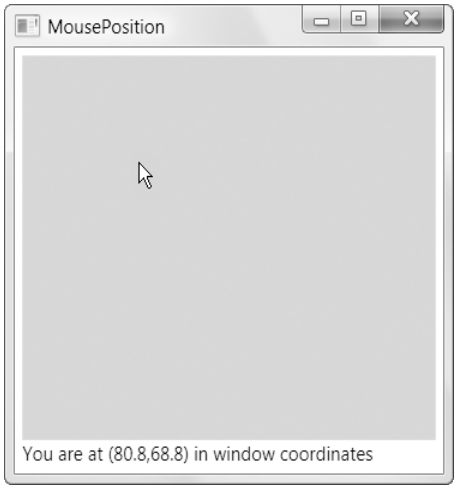

Figure 5-7. *Watching the mouse*

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.

WPF changes very little about drag-and-drop operations. If you've used them in Windows Forms applications, you'll find the programming interface is virtually unchanged in WPF. The key difference is that the methods and events that are used for drag-and-drop operations are centralized in the System.Windows.DragDrop class and then used by other classes (such as UIElement).

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*