MCTS Self-Paced Training Kit (Exam 70-502): Microsoft® .NET Framework 3.5—Windows® Presentation Foundation

Matthew A. Stoecker

To learn more about this book, visit Microsoft Learning at http://www.microsoft.com/MSPress/books12485.aspx

9780735625662

Microsoft® Press

Table of Contents

What do you think of this book? We want to hear from you!

Microsoft is interested in hearing your feedback so we can continually improve our books and learning resources for you. To participate in a brief online survey, please visit:

www.microsoft.com/learning/booksurvey/

Chapter 2

Events, Commands, and Settings

Events and commands form the basis of the architecture for intra-application communication in Windows Presentation Foundation (WPF) applications. Routed events can be raised by multiple controls and allow a fine level of control over user input. Commands are a welcome addition to the Microsoft .NET Framework and provide a central architecture for enabling and disabling high-level tasks. Application settings allow you to persist values between application sessions. In this chapter, you will learn to configure these features.

Exam objectives in this chapter:

- Configure event handling.
- Configure commands.
- Configure application settings.

Lessons in this chapter:

Before You Begin

To complete the lessons in this chapter, you must have

- A computer that meets or exceeds the minimum hardware requirements listed in the "About This Book" section at the beginning of the book
- Microsoft Visual Studio 2005 Professional Edition installed on your computer
- An understanding of Microsoft Visual Basic or C# syntax and familiarity with the .NET Framework

Real World

Matthew Stoecker

By using WPF routed events and commands, I find I have a much finer control over how my user interfaces respond compared to in a Windows Forms application. The Routed Event architecture allows me to implement complex event handling strategies, and the Command architecture provides a way to approach programming common tasks in my user interfaces.

Lesson 1: Configuring Events and Event Handling

Events in WPF programming are considerably different from those in traditional Windows Forms programming. WPF introduces routed events, which can be raised by multiple controls and handled by multiple handlers. *Routed events* allow you to add multiple levels of complexity and sophistication to your user interface and the way it responds to user input. In this lesson, you will learn about routed events, including how to handle a routed event, define and register a new routed event, handle an application lifetime event, and use the *EventManager* class.

After this lesson, you will be able to:

- Explain the difference between a direct event, a bubbling event, and a tunneling event
- Define and register a new routed event
- Define static class event handlers
- Handle an event in a WPF application
- Handle an attached event in a WPF application
- Handle application lifetime events
- Use the *EventManager* class

Estimated lesson time: 30 minutes

Events have been a familiar part of Microsoft Windows programming for years. An *event* is a message sent by an object, such as a control or other part of the user interface, that the program responds to (or handles) by executing code. While the traditional .NET event architecture is still present in WPF programming, WPF builds upon the event concept by introducing routed events.

A key concept to remember in event routing is the control containment hierarchy. In WPF user interfaces, controls frequently contain other controls. For example, a typical user interface might consist of a top-level *Window* object, which contains a *Grid* object, which itself might contain several controls, one of which could be a *ToolBar* control, which in turn contains several *Button* controls. The routed event architecture allows for an event that originates in one control to be raised by another control in the containment hierarchy. Thus, if the user clicks one of the *Button* controls on the toolbar, that event can be raised by the *Button,* the *ToolBar,* the *Grid,* or the *Window.*

Why is it useful to route events? Suppose, for example, that you are designing a user interface for a calculator program. As part of this application, you might have several

Button controls enclosed within a *Grid* control. Suppose that you wanted all button clicks in this grid to be handled by a single event handler? WPF raises the click event from the *Button,* the *Grid,* and any other control in the control containment hierarchy. As the developer, you can decide where and how the event is handled. Thus, you can provide a single event handler for all *Button Click* events originating from *Button* controls in the grid, simplifying code-writing tasks and ensuring consistency in event handling.

Types of Routed Events There are three different types of routed events: direct, bubbling, and tunneling.

Direct Events

Direct events are most similar to standard .NET events. Like a standard .NET event, a direct event is raised only by the control in which it originates. Because other controls in the control containment hierarchy do not raise these events, there is no opportunity for any other control to provide handlers for these events. An example of a direct event is the *MouseLeave* event.

Bubbling Events

Bubbling events are events that are raised first in the control where they originate and then are raised by each control in that control's control containment hierarchy, also known as a visual tree. The *MouseDown* event is an example of a bubbling event. Suppose that you have a *Label* contained inside a *FlowPanel* contained inside a *Window.* When the mouse button is pressed over the *Label,* the first control to raise the *MouseDown* event would be the *Label.* Then the *FlowPanel* would raise the *MouseDown* event, and then finally the *Window* itself. You could provide an event handler at any or all stages of the event process.

Tunneling Events

Tunneling events are the opposite of bubbling events. A *tunneling event* is raised first by the topmost container in the visual tree and then down through each successive container until it is finally raised by the element in which it originated. An example of a tunneling event is the *PreviewMouseDown* event. In the previous example, although the event originates with the *Label* control, the first control to raise the *PreviewMouseDown* event is the *Window,* then the *FlowPanel,* and then finally the *Label.* Tunneling events allow you the opportunity to intercept and handle events in the window or container before the event is raised by the specific control. This allows you to filter input, such as keystrokes, at varying levels.

In the .NET Framework, all tunneling events begin with the word "Preview," such as *PreviewKeyDown*, *PreviewMouseDown*, etc., and are typically defined in pairs with a complementary bubbling event. For example, the tunneling event *PreviewKeyDown* is paired with the bubbling event *KeyDown*. The tunneling event always is raised before its corresponding bubbling event, thus allowing an opportunity for higher-level controls in the visual tree to handle the event. Each tunneling event shares its instance of event arguments with its paired bubbling event. This fact is important to remember when handling events, and it will be discussed in greater detail later in this chapter.

RoutedEventArgs

All routed events include an instance of *RoutedEventArgs* (or a class that inherits *RoutedEventArgs*) in their signatures. The *RoutedEventArgs* class contains a wealth of information about the event and its source control. Table 2-1 describes the properties of the *RoutedEventArgs* class.

Table 2-1 *RoutedEventArgs* Properties

Property	Description
Handled	Indicates whether or not this event has been handled. By setting this property to *True*, you can halt further event bubbling or tunneling.
OriginalSource	Gets the object that originally raised the event. For most WPF controls, this will be the same as the object returned by the *Source* property. However, for some controls, such as composite controls, this property will return a different object.
RoutedEvent	Returns the *RoutedEvent* object for the event that was raised. When handling more than one event with the same event handler, you might need to refer to this property to distinguish which event has been raised.
Source	Returns the object that raised the event.

All *EventArgs* for routed events inherit the *RoutedEventArgs* class, but many of them provide additional information. For example, *KeyboardEventArgs* is used in keyboard events and provides information about keystrokes. Likewise, *MouseEventArgs*, used in mouse events, provides information about the state of the mouse when the event took place.

Quick Check

- What are the three kinds of routed events in WPF and how do they differ?

Quick Check Answer

- Routed events in WPF come in three different types: direct, tunneling, and bubbling. A direct event can be raised only by the element in which it originated. A bubbling event is raised first by the element in which it originates and then is raised by each successive container in the visual tree. A tunneling event is raised first by the topmost container in the visual tree and then down through each successive container until it is finally raised by the element in which it originated. Tunneling and bubbling events allow elements of the user interface to respond to events raised by their contained elements.

Attaching an Event Handler

The preferred way to attach an event handler is directly in the Extensible Application Markup Language (XAML) code. You set the event to the name of a method with the appropriate signature for that event. The following example demonstrates setting the event handler for a *Button* control's *Click* event, as shown in bold:

```
<Button Height="23" Margin="132,80,70,0" Name="button1"
   VerticalAlignment="Top" Click="button1_Click">Button</Button>
```

Just like setting a property, you must supply a string value that indicates the name of the method.

Attached Events

It is possible for a control to define a handler for an event that the control cannot itself raise. These incidents are called *attached events*. For example, consider *Button* controls in a *Grid*. The *Button* class defines a *Click* event, but the *Grid* class does not. However,

you still can define a handler for buttons in the grid by attaching the *Click* event of the *Button* control in the XAML code. The following example demonstrates attaching an event handler for a *Button* contained in a *Grid:*

```
<Grid Button.Click="button_Click">
   <Button Height="23" Margin="132,80,70,0" Name="button1"
   VerticalAlignment="Top" >Button</Button>
</Grid>
```

Now every time a button contained in the *Grid* shown here is clicked, the *button_Click* event handler will handle that event.

Handling a Tunneling or Bubbling Event

At times, you might want to halt the further handling of tunneling or bubbling events. For example, you might want to suppress keystroke handling at a particular level in the control hierarchy. You can handle an event and halt any further tunneling or bubbling by setting the *Handled* property of the *RoutedEventArgs* instance to *True*, as shown here:

```
' VB
Private Sub TextBox1_KeyDown(ByVal sender As System.Object, _
   ByVal e As System.Windows.Input.KeyEventArgs)
   e.Handled = True
End Sub

// C#
private void textBox1_KeyDown(object sender, KeyEventArgs e)
{
   e.Handled = true;
}
```

Note that tunneling events and their paired bubbling events (such as *PreviewKeyDown* and *KeyDown*) share the same instance of *RoutedEventArgs*. Thus, if you set the *Handled* property to *True* on a tunneling event, its corresponding bubbling event also is considered handled and is suppressed.

The *EventManager* Class

EventManager is a static class that manages the registration of all WPF routed events. Table 2-2 describes the methods of the *EventManager* class.

Table 2-2 *EventManager* Methods

Method	Description
GetRoutedEvents	Returns an array that contains all the routed events that have been registered in this application.
GetRoutedEventsForOwner	Returns an array of all the routed events that have been registered for a specified element in this application.
RegisterClassHandler	Registers a class-level event handler, as discussed in the section "Creating a Class-Level Event Handler," later in this chapter.
RegisterRoutedEvent	Registers an instance-level event handler, as discussed in the next section.

Defining a New Routed Event

You can use the *EventManager* class to define a new routed event for your WPF controls. The following procedure describes how to define a new routed event.

▶ **To define a new routed event**

1. Create a static, read-only definition for the event, as shown in this example:

```
' VB
Public Shared ReadOnly SuperClickEvent As RoutedEvent

// C#
public static readonly RoutedEvent SuperClickEvent;
```

2. Create a wrapper for the routed event that exposes it as a traditional .NET Framework event, as shown in this example:

```
' VB
Public Custom Event SuperClick As RoutedEventHandler
   AddHandler(ByVal value As RoutedEventHandler)
      Me.AddHandler(SuperClickEvent, value)
   End AddHandler

   RemoveHandler(ByVal value As RoutedEventHandler)
      Me.RemoveHandler(SuperClickEvent, value)
   End RemoveHandler
```

```
    RaiseEvent(ByVal sender As Object, _
       ByVal e As System.Windows.RoutedEventArgs)
       Me.RaiseEvent(e)
    End RaiseEvent
End Event

// C#
public event RoutedEventHandler SuperClick
{
   add
   {
      this.AddHandler(SuperClickEvent, value);
   }
   remove
   {
      this.RemoveHandler(SuperClickEvent, value);
   }
}
```

Note that you need to use a different *EventArgs* class than *RoutedEventArgs*. You need to derive a new class from *RoutedEventArgs* and create a new delegate that uses those event arguments.

3. Use *EventManager* to register the new event in the constructor of the class that owns this event. You must provide the name of the event, the routing strategy (direct, tunneling, or bubbling), the type of delegate that handles the event, and the type of the class that owns it. An example is shown here:

```
' VB
EventManager.RegisterRoutedEvent("SuperClick", _
   RoutingStrategy.Bubble, GetType(RoutedEventArgs), GetType(Window1))

// C#
EventManager.RegisterRoutedEvent("SuperClick",
   RoutingStrategy.Bubble, typeof(RoutedEventHandler), typeof(Window1));
```

Raising an Event

Once an event is defined, you can raise it in code by creating a new instance of *RoutedEventArgs* and using the *RaiseEvent* method, as shown here:

```
' VB
Dim myEventArgs As New RoutedEventArgs(myControl.myNewEvent)
MyBase.RaiseEvent(myEventArgs)

// C#
RoutedEventArgs myEventArgs = new RoutedEventArgs(myControl.myNewEvent);
RaiseEvent(myEventArgs);
```

Creating a Class-Level Event Handler

You can use the *EventManager* class to register a class-level event handler. A class-level event handler handles a particular event for all instances of a class, and is always invoked before instance handlers. Thus, you can screen and suppress events before they reach instance handlers. The following procedure describes how to implement a class-level event handler.

▶ To create a class-level event handler

1. Create a static method to handle the event. This method must have the same signature as the event. An example is shown here:

```
' VB
Private Shared Sub SuperClickHandlerMethod(ByVal sender As Object, _
   ByVal e As RoutedEventArgs)
   ' Handle the event here
End Sub

// C#
private static void SuperClickHandlerMethod(object sender, RoutedEventArgs e)
{
   // Handle the event here
}
```

2. In the static constructor for the class for which you are creating the class-level event handler, create a delegate to this method, as shown here:

```
' VB
Dim SuperClickHandler As New RoutedEventHandler( _
    AddressOf SuperClickHandlerMethod)

// C#
RoutedEventHandler SuperClickHandler = new
   RoutedEventHandler(SuperClickHandlerMethod);
```

3. Also in the static constructor, call *EventManager.RegisterClassHandler* to register the class-level event handler, as shown here:

```
' VB
EventManager.RegisterClassHandler(GetType(Window1), _
   SuperClickEvent, SuperClickHandler)

// C#
EventManager.RegisterClassHandler(typeof(Window1),
   SuperClickEvent,SuperClickHandler);
```

Application-Level Events

Every WPF application is wrapped by an *Application* object. The *Application* object provides a set of events that relate to the application's lifetime. You can handle these events

to execute code in response to application startup or closure. The *Application* object also provides a set of events related to navigation in page-based applications. These events were discussed in Chapter 1, "WPF Application Fundamentals." Table 2-3 describes the available application-level events, excluding the navigation events.

Table 2-3 Selected Application-Level Events

Event	Description
Activated	Occurs when you switch from another application to your program. It also is raised the first time you show a window.
Deactivated	Occurs when you switch to another program.
DispatcherUnhandledException	Raised when an unhandled exception occurs in your application. You can handle an unhandled exception in the event handler for this event by setting the *DispatcherUnhandledException-EventArgs.Handled* property to *True*.
Exit	Occurs when the application is shut down for any reason.
SessionEnding	Occurs when the Windows session is ending, such as when the user shuts down the computer or logs off.
Startup	Occurs as the application is started.

Application events are standard .NET events (rather than routed events), and you can create handlers for these events in the standard .NET way. The following procedure explains how to create an event handler for an application-level event.

▶ To create an application-level event handler

1. In Visual Studio, in the Solution Explorer, right-click Application.xaml (in Visual Basic) or App.xaml (in C#) and choose View Code to open the code file for the *Application* object.
2. Create a method to handle the event, as shown here:

```
' VB
Private Sub App_Startup(ByVal sender As Object, _
   ByVal e As StartupEventArgs)
   ' Handle event here
End Sub
```

```
// C#
void App_Startup(object sender, StartupEventArgs e)
{
   // Handle the event here
}
```

3. In XAML view for the *Application* object, add the event handler to the Application declaration, as shown in bold here:

```
<Application x:Class="Application"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   StartupUri="Window1.xaml" Startup="App_Startup">
```

Lab: Practice with Routed Events

In this lab, you practice using routed events. You create event handlers for the *TextBox.TextChanged* event in three different controls in the visual tree and observe how the event is raised and handled by each one.

Exercise: Creating an Event Handler

1. In Visual Studio, create a new WPF application.
2. From the Toolbox, drag a *TextBox* and three *RadioButton* controls onto the design surface. Note that at this point, these controls are contained by a *Grid* control that is in itself contained in the top-level *Window* control. Thus any bubbling events raised by the *TextBox* will bubble up first to the *Grid* and then to the *Window*.
3. In XAML view, set the display contents of the *RadioButton* controls as follows:

RadioButton	Content
RadioButton1	Handle Textbox.TextChanged in TextBox
RadioButton2	Handle Textbox.TextChanged in Grid
RadioButton3	Handle Textbox.TextChanged in Window

4. In the XAML for the *TextBox*, just before the />, type **TextChanged** and then press the Tab key twice. An entry for an event handler is created and a corresponding method is created in the code. The event-handler entry should look like the following:

```
TextChanged="TextBox1_TextChanged"
```

5. In the XAML for the *Grid*, type **TextBoxBase.TextChanged** and then press the Tab key twice to generate an event handler. The added XAML should look like this:

```
TextBoxBase.TextChanged="Grid_TextChanged"
```

6. In the XAML for the *Window*, type **TextBoxBase.TextChanged** and then press the Tab key twice to generate an event handler. The added XAML should look like this:

```
TextBoxBase.TextChanged="Window_TextChanged"
```

7. In Code view, add the following code to the *Textbox1_TextChanged* method:

```
' VB
MessageBox.Show("Event raised by Textbox")
e.Handled = RadioButton1.IsChecked

// C#
MessageBox.Show("Event raised by Textbox");
e.Handled = (bool)radioButton1.IsChecked;
```

8. Add the following code to the *Grid_TextChanged* method:

```
' VB
MessageBox.Show("Event raised by Grid")
e.Handled = RadioButton2.IsChecked

// C#
MessageBox.Show("Event raised by Grid");
e.Handled = (bool)radioButton2.IsChecked;
```

9. Add the following code to the *Window_TextChanged* method:

```
' VB
MessageBox.Show("Event raised by Window")
e.Handled = RadioButton3.IsChecked

// C#
MessageBox.Show("Event raised by Window");
e.Handled = (bool)radioButton3.IsChecked;
```

10. Press F5 to build and run your application. Type a letter in the *TextBox*. Three message boxes are displayed, each one indicating the control that raised the event. You can handle the event by choosing one of the radio buttons to halt event bubbling in the event handlers.

Lesson Summary

- WPF applications introduce a new kind of event called routed events. Routed events are raised by WPF controls.
- There are three kinds of routed events: direct, bubbling, and tunneling. Direct events are raised only by the control in which they originate. Bubbling and

tunneling events are raised by the control in which they originate and all controls that are higher in the visual tree.

- A tunneling event is raised first by the top-level control in the visual tree and tunnels down through the tree until it is finally raised by the control in which it originates. A bubbling event is raised first by the control in which the event originates and then bubbles up through the visual tree until it is finally raised by the top-level control in the visual tree.
- You can attach events that exist in contained controls to controls that are higher in the visual tree.
- The *EventManager* class exposes methods that allow you to manage events in your application. You can register a new routed event by using the *EventManager.RegisterRoutedEvent* class. You can create a class-level event handler by using *EventManager.RegisterClassHandler*.
- The *Application* object raises several events that can be handled to execute code at various points in the application's lifetime. You can handle application-level events in the code for the *Application* object.

Lesson Review

You can use the following questions to test your knowledge of the information in Lesson 1, "Configuring Events and Event Handling." The questions are also available on the companion CD of this book if you prefer to review them in electronic form.

NOTE Answers

Answers to these questions and explanations of why each answer choice is correct or incorrect are located in the "Answers" section at the end of the book.

---

1. Suppose you have the following XAML code:

```
<Window x:Class="WpfApplication1.Window1"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Title="Window1" Height="300" Width="300"
   ButtonBase.Click="Window_Click">
   <Grid ButtonBase.Click="Grid_Click">
      <StackPanel Margin="47,54,31,108" Name="stackPanel1"
         ButtonBase.Click="stackPanel1_Click">
         <Button Height="23" Name="button1" Width="75">Button</Button>
      </StackPanel>
   </Grid>
</Window>
```

Which method will be executed first when *button1* is clicked?

A. *Button1_Click*

B. *stackPanel1_Click*

C. *Grid_Click*

D. *Window_Click*

2. Suppose you have the following XAML code:

```
<Window x:Class="WpfApplication1.Window1"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Title="Window1" Height="300" Width="300" MouseDown="Window_MouseDown">
   <Grid PreviewMouseDown="Grid_PreviewMouseDown">
      <StackPanel Margin="47,54,31,108" Name="stackPanel1"
         PreviewMouseDown="stackPanel1_PreviewMouseDown">
         <Button Click="button1_Click" Height="23" Name="button1"
             Width="75">Button</Button>
      </StackPanel>
   </Grid>
</Window>
```

Which method will be executed first when *button1* is clicked?

A. *Window_MouseDown*

B. *Grid_PreviewMouseDown*

C. *stackPanel1_PreviewMouseDown*

D. *button1_Click*

3. You are writing an application that consists of a single WPF window. You have code that you want to execute when the window first appears and every time the window is activated. What application event or events should you handle to accomplish this goal?

A. *Activated*

B. *Startup*

C. *Activated* and *Startup*

D. *Deactivated* and *Startup*

Lesson 2: Configuring Commands

WPF introduces new objects called *commands*. Commands represent high-level tasks that are performed in the application. For example, *Paste* is an example of a command—it represents the task of copying an object from the clipboard into a container. WPF provides a cohesive architecture for creating commands, associating them with application tasks, and hooking those commands up to user interface (UI) elements. In this lesson, you will learn to use the built-in command library, associate these commands with UI elements, define command handlers, add a gesture to a command, and define custom commands.

After this lesson, you will be able to:

- Explain the different parts of a command
- Associate a command with a UI element
- Add a gesture to a command
- Execute a command
- Associate a command with a command handler
- Disable a command
- Create a custom command

Estimated lesson time: 30 minutes

Commands, such as Cut, Copy, and Paste, represent tasks. In past versions of the .NET Framework, there was no complete architecture for associating code with tasks. For example, suppose you wanted to implement a *Paste* task in your application. You would create the code to execute the task, and then associate your UI element with that code via events. For example, you might have a *MenuItem* element that triggers the code when selected. You also might have context menu items and perhaps even a *Button* control. In past versions of the .NET Framework, you would have had to create event handlers for each control with which you want to associate the task. In addition, you would have had to implement code to inactivate each of these controls if the task was unavailable. While not an impossible task, doing this requires tedious coding that can be fraught with errors.

Commands allow you to use a centralized architecture for tasks. You can associate any number of UI controls or input gestures to a command and bind that command to a handler that is executed when controls are activated or gestures are performed. Commands also keep track of whether or not they are available. If a command is disabled, UI elements associated with that command are disabled, too.

Command architecture consists of four principal parts. There is the *Command* object itself, which represents the task. Then there are command sources. A *command source* is a control or gesture that triggers the command when invoked. The *command handler* is a method that is executed when the command is invoked, and *CommandBinding* is an object that is used by the .NET Framework to track what commands are associated with which sources and handlers.

The .NET Framework provides several predefined commands that are available for use by developers. These built-in commands are static objects that are properties of five static classes, which are the following:

- *ApplicationCommands*
- *ComponentCommands*
- *EditingCommands*
- *MediaCommands*
- *NavigationCommands*

Each of these classes exposes a variety of static command objects that you can use in your applications. While some of these commands have default input bindings (for example, the *ApplicationCommands.Open* command has a default binding to the key combination Ctrl+O), none of these commands has any inherent functionality—you must create bindings and handlers for these commands to use them in your application.

A High-Level Procedure for Implementing a Command

The following section describes a high-level procedure for implementing command functionality. The steps of this procedure are discussed in greater detail in the subsequent sections.

▶ **To implement a command**

1. Decide on the command to use, whether it is one of the static commands exposed by the .NET Framework or a custom command.
2. Associate the command with any controls in the user interface and add any desired input gestures to the command.
3. Create a method to handle the command.
4. Create a *CommandBinding* that binds the *Command* object to the command handler and optionally to a method that handles *Command.CanExecute*.

5. Add the command binding to the *Commands* collection of the control or *Window* where the command is invoked.

Invoking Commands

Once a command has been implemented, you can invoke it by associating it with a control, using a gesture, or invoking it directly from code.

Associating Commands with Controls

Many WPF controls implement the *ICommandSource* interface, which allows them to have a command associated with them that is fired automatically when that control is invoked. For example, *Button* and *MenuItem* controls implement *ICommandSource* and thus expose a *Command* property. When this property is set to a command, that command is executed automatically when the control is clicked. You can set a command for a control in XAML, as shown here:

```
<Button Command="ApplicationCommands.Find" Height="23"
HorizontalAlignment="Right" Margin="0,0,38,80" Name="Button3"
VerticalAlignment="Bottom" Width="75">Button</Button>
```

Invoking Commands with Gestures

You also can register mouse and keyboard gestures with *Command* objects that invoke the command when those gestures occur. The following example code shows how to add a mouse gesture and a keyboard gesture to the *InputGestures* collection of the *Application.Find* command:

```
' VB
ApplicationCommands.Find.InputGestures.Add(New _
   MouseGesture(MouseAction.LeftClick, ModifierKeys.Control))
ApplicationCommands.Find.InputGestures.Add(New _
   KeyGesture(Key.Q, ModifierKeys.Control))

// C#
ApplicationCommands.Find.InputGestures.Add(new
   MouseGesture(MouseAction.LeftClick, ModifierKeys.Control));
ApplicationCommands.Find.InputGestures.Add(new
   KeyGesture(Key.Q, ModifierKeys.Control));
```

Once the code in the previous example is executed, the *Find* command executes either when the Ctrl key is held down and the left mouse button is clicked, or when the Ctrl key and the Q key are held down together (Ctrl+Q).

Invoking Commands from Code

You might want to invoke a command directly from code, such as in response to an event in a control that does not expose a *Command* property. To invoke a command directly, simply call the *Command.Execute* method, as shown here:

```
' VB
ApplicationCommands.Find.Execute(aParameter, TargetControl)

// C#
ApplicationCommands.Find.Execute(aParameter, TargetControl);
```

In this example, *aParameter* represents an object that contains any required parameter data for the command. If no parameter is needed, you can use *null* (*Nothing* in Visual Basic). *TargetControl* is a control where the command originates. The run time will start looking for *CommandBindings* in this control and then bubble up through the visual tree until an appropriate *CommandBinding* is found.

Command Handlers and Command Bindings

As stated before, just invoking a command doesn't actually do anything. Commands represent tasks, but they do not contain any of the code for the tasks they represent. To execute code when a command is invoked, you must create a *CommandBinding* that binds the command to a command handler.

Command Handlers

Any method with the correct signature can be a command handler. Command handlers have the following signature:

```
' VB
Private Sub myCommandHandler(ByVal sender As Object, _
   ByVal e As ExecutedRoutedEventArgs)
   ' Handle the command here
End Sub

// C#
private void myCommandHandler(object sender, ExecutedRoutedEventArgs e)
{
   // Handle the command here
}
```

ExecutedRoutedEventArgs is derived from *RoutedEventArgs* and thus exposes all the members that *RoutedEventArgs* does. In addition, it exposes a *Command* property that returns the *Command* object that is being handled.

Command Bindings

The *CommandBinding* object provides the glue that holds the whole command architecture together. A *CommandBinding* associates a command with a command handler. Adding a *CommandBinding* to the *CommandBindings* collection of the *Window* or a control registers the *CommandBinding* and allows the command handler to be called when the command is invoked. The following code demonstrates how to create and register a *CommandBinding:*

```
' VB
Dim abinding As New CommandBinding()
abinding.Command = ApplicationCommands.Find
AddHandler abinding.Executed, AddressOf myCommandHandler
Me.CommandBindings.Add(abinding)

// C#
CommandBinding abinding = new CommandBinding();
abinding.Command = ApplicationCommands.Find;
abinding.Executed += new ExecutedRoutedEventHandler(myCommandHandler);
this.CommandBindings.Add(abinding);
```

In the preceding example, you first create a new *CommandBinding* object. You then associate that *CommandBinding* object with a *Command* object. Next, you specify the command handler that will be executed when the command is invoked, and finally, you add the *CommandBinding* object to the *CommandBindings* collection of the *Window.* Thus, if an object in the window invokes the command, the corresponding command handler will be executed.

You also can define *CommandBindings* directly in the XAML. You can create a new binding and declaratively set the command it is associated with and the associated handlers. The following example demonstrates a new *CommandBinding* in the *CommandBinding* collection of the window that associates the *Application.Find* command with a handler:

```
<Window.CommandBindings>
   <CommandBinding Command="ApplicationCommands.Find"
      Executed="myCommandHandler" />
</Window.CommandBindings>
```

Command Bubbling

Note that all controls have their own *CommandBindings* collection in addition to the window's *CommandBindings* collection. This is because commands, like routed events, bubble up through the visual tree when they are invoked. Commands look for a binding first in the *CommandBindings* collection of the control in which they originate, and

then in the *CommandBindings* collections of controls higher on the visual tree. Like a *routedEvent,* you can stop further processing of the command by setting the *Handled* property of the *ExecutedRoutedEventArgs* parameter to *True*, as shown here:

```
' VB
Private Sub myCommandHandler(ByVal sender As Object, _
   ByVal e As ExecutedRoutedEventArgs)
   ' Stops further Command bubbling
   e.Handled = True
End Sub

// C#
private void myCommandHandler(object sender, ExecutedRoutedEventArgs e)
{
   // Handle the command here
   e.Handled = true;
}
```

Exam Tip Bubbling and tunneling are concepts that are new to WPF and that play important roles both in commands and how WPF handles routed events. Be certain that you understand the concepts of bubbling and tunneling events and bubbling commands for the exam. Remember that a command or event doesn't need to be handled by the same element in which it originates.

Disabling Commands

Any command that is not associated with a *CommandBinding* is automatically disabled. No action is taken when that command is invoked, and any control that has its *Command* property set to that command appears as disabled. However, there might be times that you want to disable a command that is in place and associated with controls and *CommandBindings.* For example, you might want the Print command to be disabled until the focus is on a document. The command architecture allows you to designate a method to handle the *Command.CanExecute* event. The *CanExecute* event is raised at various points in the course of application execution to determine whether a command is in a state that will allow execution.

Methods that handle the *CanExecute* event include an instance of *CanExecuteRoutedEventArgs* as a parameter. This class exposes a property called *CanExecute* that is a boolean value. If *CanExecute* is true, the command can be invoked. If it is false, the command is disabled. You can create a method that handles the *CanExecute* event, determines whether or not the application is in an appropriate state to allow command execution, and sets *e.CanExecute* to the appropriate value.

▶ To handle the *CanExecute* event

1. Create a method to handle the *CanExecute* event. This method should query the application to determine whether the application's state is appropriate to allow the command to be enabled. An example is shown here:

```
' VB
Private canExecute As Boolean
Private Sub abinding_CanExecute(ByVal sender As Object, _
   ByVal e As CanExecuteRoutedEventArgs)
   e.CanExecute = canExecute
End Sub

// C#
bool canExecute;
void abinding_CanExecute(object sender, CanExecuteRoutedEventArgs e)
{
   e.CanExecute = canExecute;
}
```

   In this example, the method returns the value represented by a private variable called *canExecute*. Presumably, the application sets this to False whenever it requires the command to be disabled.

2. Set the *CanExecute* handler on the *CommandBinding* to point to this method, as shown here:

```
' VB
' Assumes that you have already created a CommandBinding called abinding
AddHandler abinding.CanExecute, AddressOf abinding_CanExecute

// C#
// Assumes that you have already created a CommandBinding called abinding
abinding.CanExecute += new CanExecuteRoutedEventHandler(abinding_CanExecute);
```

   Alternatively, create a new binding in XAML and specify the handler there, as shown here in bold:

```
<Window.CommandBindings>
   <CommandBinding Command="ApplicationCommands.Find"
      Executed="CommandBinding_Executed"
      CanExecute="abinding_CanExecute" />
</Window.CommandBindings>
```

Creating Custom Commands

Although a wide variety of pre-existing commands is at your disposal, you might want to create your own custom commands. The best practice for custom commands is to follow the example set in the .NET Framework and create static classes (in C#) or

modules (in Visual Basic) that expose static instances of the custom command. This keeps multiple instances of the command from being created. You also can provide any custom configuration for the command in the static constructor of the class—for example, if you want to map any input gestures to the command. The following example shows how to create a static class that exposes a custom command called *Launch:*

```
' VB
Public Module MyCommands
   Private launch_command As RoutedUICommand
   Sub New()
      Dim myInputGestures As New InputGestureCollection
      myInputGestures.Add(New KeyGesture(Key.L, ModifierKeys.Control))
      launch_command = New RoutedUICommand("Launch", "Launch", _
        GetType(MyCommands), myInputGestures)
   End Sub
   Public ReadOnly Property Launch() As RoutedUICommand
      Get
         Return launch_command
      End Get
   End Property
End Module

// C#
public class MyCommands
{
   private static RoutedUICommand launch_command;
   static MyCommands()
   {
      InputGestureCollection myInputGestures = new
         InputGestureCollection();
      myInputGestures.Add(new KeyGesture(Key.L, ModifierKeys.Control));
      launch_command = new RoutedUICommand("Launch", "Launch",
         typeof(MyCommands), myInputGestures);
   }
   public RoutedUICommand Launch
   {
      get
      {
         return launch_command;
      }
   }
}
```

In this example, a static class or module is created to contain the custom command, which is exposed through a read-only property. In the static constructor, a new *InputGesturesCollection* is created and a key gesture is added to the collection. This collection is then used to initialize the instance of *RoutedUICommand* that is returned through the read-only property.

Using Custom Commands in XAML

Once you have created a custom command, you are ready to use it in code. If you want to use it in XAML, however, you also must map the namespace that contains the custom command to a XAML namespace. The following procedure describes how to use a custom command in XAML.

▶ **To use a custom command in XAML**

1. Create your custom command, as described previously.
2. Add a namespace mapping to your *Window* XAML. The following example demonstrates how to map a namespace called *WpfApplication13.CustomCommands*. Note that in this example, that would mean that your custom commands are kept in a separate namespace:

```
<Window x:Class="Window1"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   xmlns:CustomCommands="clr-namespace:WpfApplication13.CustomCommands"
   Title="Window1" Height="300" Width="300">


   <!-The rest of the XAML is omitted-->
</Window>
```

3. Use the newly mapped XAML namespace in your XAML code, as shown here:

```
<Button Command="CustomCommands:MyCommands.Launch" Height="23"
HorizontalAlignment="Left" Margin="60,91,0,0" Name="Button1" VerticalAlignment="Top"
Width="75">Button</Button>
```

Lab: Creating a Custom Command

In this lab, you create a custom command and then connect your command to UI elements by using a *CommandBinding*.

Exercise 1: Creating a Custom Command

1. From the CD, open the partial solution for this exercise.
2. From the Project menu, choose Add Class (in C#) or Add Module (in Visual Basic). Name the new item *CustomCommands* and click OK. Set the access modifier of this class or module to *public*.
3. If you are working in C#, add the following *using* statement to your class:

```
using System.Windows.Input;
```

   Otherwise, go on to Step 4.

4. Add a read-only property named *Launch* and a corresponding member variable that returns an instance of a *RoutedUICommand,* as shown here. (Note that these should be static members in C#.)

```
' VB
Private launch_command As RoutedUICommand
Public ReadOnly Property Launch() As RoutedUICommand
   Get
      Return launch_command
   End Get
End Property

// C#
private static RoutedUICommand launch_command;
public static RoutedUICommand Launch
{
   get
   {
      return launch_command;
   }
}
```

5. Add a constructor to your module (in Visual Basic) or a static constructor to your class (in C#) that creates a new *InputGestureCollection,* adds an appropriate input gesture to be associated with this new command, and then initializes the member variable that returns the custom command, as shown here:

```
' VB
Sub New()
   Dim myInputGestures As New InputGestureCollection
   myInputGestures.Add(New KeyGesture(Key.L, ModifierKeys.Control))
   launch_command = New RoutedUICommand("Launch", "Launch", _
      GetType(CustomCommands), myInputGestures)
End Sub

// C#
static CustomCommands()
{
   InputGestureCollection myInputGestures = new
      InputGestureCollection();
   myInputGestures.Add(new KeyGesture(Key.L, ModifierKeys.Control));
   launch_command = new RoutedUICommand("Launch", "Launch",
      typeof(CustomCommands), myInputGestures);
}
```

6. From the Build menu, choose Build Solution to build your solution.

Exercise 2: Using Your Custom Command

1. In XAML view, add the following code to your *Window* markup to create a reference to the class that contains your custom command:

   ```
   xmlns:Local="clr-namespace:YourProjectNamespaceGoesHere"
   ```

   The previous code in bold should be replaced with the namespace name of your project.

2. In XAML view, add the following attribute to both your *Button* control and your *Launch* MenuItem:

   ```
   Command="Local:CustomCommands.Launch"
   ```

3. In the *Window1* code view, add the following method:

   ```
   ' VB
   Private Sub Launch_Handler(ByVal sender As Object, _
      ByVal e As ExecutedRoutedEventArgs)
      MessageBox.Show("Launch invoked")
   End Sub

   // C#
   private void Launch_Handler(object sender, ExecutedRoutedEventArgs e)
   {
      MessageBox.Show("Launch invoked");
   }
   ```

4. From the Toolbox, drag a *CheckBox* control onto the form. Set the content of the control to "Enable Launch Command".

5. In the code view for *Window1*, add the following method:

   ```
   ' VB
   Private Sub LaunchEnabled_Handler(ByVal sender As Object, _
      ByVal e As CanExecuteRoutedEventArgs)
      e.CanExecute = CheckBox1.IsChecked
   End Sub

   // C#
   private void LaunchEnabled_Handler(object sender,
      CanExecuteRoutedEventArgs e)
   {
      e.CanExecute = (bool)checkBox1.IsChecked;
   }
   ```

6. Create or replace the constructor for *Window1* that creates and registers a *CommandBinding* for the *Launch* command. This *CommandBinding* should bind the *Launch.Executed* event to the *Launch_Handler* method and bind the *Launch.CanExecute* event to the *LaunchEnabled_Handler* method. An example is shown here:

```
' VB
Public Sub New()
   InitializeComponent()
   Dim abinding As New CommandBinding()
   abinding.Command = CustomCommands.Launch
   AddHandler abinding.Executed, AddressOf Launch_Handler
   AddHandler abinding.CanExecute, AddressOf LaunchEnabled_Handler
   Me.CommandBindings.Add(abinding)
End Sub

// C#
public Window1()
{
   InitializeComponent();
   CommandBinding abinding = new CommandBinding();
   abinding.Command = CustomCommands.Launch;
   abinding.Executed += new ExecutedRoutedEventHandler(Launch_Handler);
   abinding.CanExecute += new
      CanExecuteRoutedEventHandler(LaunchEnabled_Handler);
   this.CommandBindings.Add(abinding);
}
```

7. Press F5 to build and run your application. Note that when the application starts, the *Button* and *Launch* menu item are disabled. Select the check box to enable the command. Now you can invoke the command from the button, from the menu, or by using the Ctrl+L input gesture.

Lesson Summary

- Commands provide a central architecture for managing high-level tasks. The .NET Framework provides a library of built-in commands that map to common tasks that can be used in your applications.
- Commands can be invoked directly, by an input gesture such as a *MouseGesture* or a *KeyGesture*, or by activating a custom control. A single command can be associated with any number of gestures or controls.
- *CommandBindings* associate commands with command handlers. You can specify a method to handle the *Executed* event of a command and another method to handle the *CanExecute* event of a command.
- Methods handling the *CanExecute* event of a command should set the *CanExecute* property of the *CanExecuteRoutedEventArgs* to False when the command should be disabled.
- Commands can be bound by any number of *CommandBindings*. Commands exhibit bubbling behavior. When invoked, commands first look for a binding in

the collection of the element that the command was invoked in, and then look in each higher element in the visual tree.

- You can create custom commands. When you have created a custom command, you must map the namespace in which it exists to a XAML namespace in your XAML view.

Lesson Review

You can use the following questions to test your knowledge of the information in Lesson 2, "Configuring Commands." The questions are also available on the companion CD if you prefer to review them in electronic form.

NOTE Answers

Answers to these questions and explanations of why each answer choice is correct or incorrect are located in the "Answers" section at the end of the book.

---

1. Which of the following is required to bind a command to a command handler? (Choose all that apply.)
   - **A.** Instantiate a new instance of *CommandBinding*
   - **B.** Set the *CommandBinding.Command* property to a command
   - **C.** Add one or more input gestures to your command
   - **D.** Add a handler for the *CommandBinding.Executed* event
   - **E.** Add a handler for the *CommandBinding.CanExecute* event
   - **F.** Add *CommandBinding* to the *CommandBindings* collection of the *Window* or other control associated with the command
2. You are working with an application that exposes a command named *Launch*. This command is registered in the *CommandBindings* collection of a control called *Window11* and requires a *String* parameter. Which of the following code snippets invokes the command from code correctly?
   - **A.**

```
' VB
Launch.CanExecute = True
Launch.Execute("Boom", Window11)

// C#
Launch.CanExecute = true;
Launch.Execute("Boom", Window11);
```

B.

```
' VB
Launch.Execute("Boom")

// C#
Launch.Execute("Boom");
```

C.

```
' VB
Launch.Execute("Boom", Window11)

// C#
Launch.Execute("Boom", Window11);
```

D.

```
' VB
Window11.CanExecute(Launch, True)
Launch.Execute("Boom", Window11)

// C#
Window11.CanExecute(Launch, true);
Launch.Execute("Boom", Window11);
```

Lesson 3: Configuring Application Settings

The .NET Framework allows you to create and access values that persist from application session to application session. The values are called *settings*. Settings can represent any kind of information that an application might need from session to session, such as user preferences, the address of a Web server, or any other kind of necessary information. In this lesson, you will learn to create and access settings. You will learn the difference between a user setting and an application setting, and you will learn to load and save settings at run time.

After this lesson, you will be able to:

- Explain the difference between a user setting and an application setting
- Create a new setting at design time
- Load settings at run time
- Save user settings at run time

Estimated lesson time: 15 minutes

Settings can be used to store information that is valuable to the application but might change from time to time. For example, you can use settings to store user preferences, such as the color scheme of an application, or the address of a Web server used by the application.

Settings have four properties:

- *Name,* which indicates the name of the setting. This is used to access the setting at run time.
- *Type,* which represents the data type of the setting.
- *Value,* which is the value returned by the setting.
- *Scope,* which can be either *User* or *Application*.

The *Name, Type,* and *Value* properties should be fairly self-explanatory. The *Scope* property, however, bears a little closer examination. The *Scope* property can be set to either *Application* or *User*. A setting with *Application* scope represents a value that is used by the entire application regardless of the user, whereas an application with *User* scope is more likely to be user-specific and less crucial to the application.

An important distinction between user settings and application settings is that user settings are read/write. They can be read and written to at run time, and newly written

values can be saved by the application. In contrast, *Application settings* are read-only and the values can be changed only at design time or by editing the Settings file between application sessions.

Creating Settings at Design Time

Visual Studio provides an editor to create settings for your application at design time. This editor is shown in Figure 2-1.

Figure 2-1 The Settings Editor

The Settings Editor allows you to create new settings and set each of their four properties. The *Name* property, the name that you use to retrieve the setting value, must be unique in the application. The *Type* property represents the type of the setting. The *Scope* property is either *Application*, which represents a read-only property, or *User*, which represents a read-write setting. Finally, the *Value* property represents the value returned by the setting. The *Value* property must be of the type specified by the Type property.

▶ To create a setting at design time

1. If you are working in C#, in Solution Explorer, under Properties, locate and double-click Settings.settings to open the Settings Editor. If you are working in Visual Basic, in Solution Explorer, double-click MyProject and select the Settings tab.
2. Set the Name, Type, Scope, and Value for the new setting.
3. If your application has not yet been saved, choose Save All from the File menu to save your application.

Loading Settings at Run Time

At run time, you can access the values contained by the settings. In Visual Basic, settings are exposed through the *My* object, whereas in C#, you access settings through the *Properties.Settings.Default* object. At design time, individual settings appear in IntelliSense as properties of the *Settings* object and can be treated in code as such. Settings are strongly typed and are retrieved as the same type as specified when they were created. The following example code demonstrates how to copy the value from a setting to a variable:

```
' VB
Dim aString As String
aString = My.Settings.MyStringSetting

// C#
String aString;
aString = Properties.Settings.Default.MyStringSetting;
```

Saving User Settings at Run Time

You can save the value of user settings at run time. To change the value of a user setting, simply assign it a new value, just as you would any property or field. Then you must call the *Save* method to save the new value. An example is shown here:

```
' VB
My.Settings.Headline = "This is tomorrow's headline"
My.Settings.Save

// C#
Properties.Settings.Default.Headline = "This is tomorrow's headline";
Properties.Settings.Default.Save();
```

Quick Check

- What is the difference between a user setting and an application setting?

Quick Check Answer

- A user setting is designed to be user-specific, such as a background color. User settings can be written at run time and can vary from user to user. An application setting is designed to be constant for all users of an application, such as a database connection string. Application settings are read-only at run time.

Lab: Practice with Settings

In this lab you create an application that uses settings. You define settings while building the application, read the settings, apply them in your application, and enable the user to change one of the settings.

Exercise: Using Settings

1. In Visual Studio, create a new WPF application.
2. In Solution Explorer, expand Properties and double-click Settings.settings (in C#) or double-click My Project and choose the Settings tab (in Visual Basic) to open the Settings Editor.
3. Add two settings, as described in this table:

Name	Type	Scope	Value
ApplicationName	*String*	Application	Settings App
BackgroundColor	*System.Windows.Media.Color*	User	#ff0000ff

   Note that you will have to browse to find the *System.Windows.Media* type, then expand the node to find the *System.Windows.Media.Color* type.

4. In XAML view, add the following XAML to the *Grid* element to add a *ListBox* with four items and a *Button* to your user interface:

```
<ListBox Margin="15,15,0,0" Name="listBox1" Height="78"
   HorizontalAlignment="Left" VerticalAlignment="Top" Width="107">
   <ListBoxItem>Red</ListBoxItem>
   <ListBoxItem>Blue</ListBoxItem>
   <ListBoxItem>Green</ListBoxItem>
   <ListBoxItem>Tomato</ListBoxItem>
</ListBox>
<Button Margin="15,106,110,130" Name="button1">Change Background
   Color</Button>
```

5. In the designer, double-click *button1* to open the code view to the default handler for the *Click* event. Add the following code:

```
' VB
If Not listBox1.SelectedItem Is Nothing Then
   Dim astring As String = CType(listBox1.SelectedItem, _
      ListBoxItem).Content.ToString
   Select Case astring
      Case "Red"
         My.Settings.BackgroundColor = Colors.Red
```

```
      Case "Blue"
         My.Settings.BackgroundColor = Colors.Blue
      Case "Green"
         My.Settings.BackgroundColor = Colors.Green
      Case "Tomato"
         My.Settings.BackgroundColor = Colors.Tomato
   End Select
   Me.Background = New _
      System.Windows.Media.SolidColorBrush(My.Settings.BackgroundColor)
   My.Settings.Save()
End If

// C#
if (!(listBox1.SelectedItem == null))
{
   String astring =
      ((ListBoxItem)listBox1.SelectedItem).Content.ToString();
   switch (astring)
   {
      case "Red":
         Properties.Settings.Default.BackgroundColor = Colors.Red;
         break;
      case "Blue":
         Properties.Settings.Default.BackgroundColor = Colors.Blue;
         break;
      case "Green":
         Properties.Settings.Default.BackgroundColor = Colors.Green;
         break;
      case "Tomato":
         Properties.Settings.Default.BackgroundColor = Colors.Tomato;
         break;
   }
   this.Background = new
      System.Windows.Media.SolidColorBrush(
         Properties.Settings.Default.BackgroundColor);
   Properties.Settings.Default.Save();
}
```

6. Create or replace the constructor for this class with the following code to read and apply the settings:

```
' VB
Public Sub New()
   InitializeComponent()
   Me.Title = My.Settings.ApplicationName
   Me.Background = New _
      System.Windows.Media.SolidColorBrush(My.Settings.BackgroundColor)
End Sub

// C#
public Window1()
{
   InitializeComponent();
```

```
    this.Title = Properties.Settings.Default.ApplicationName;
    this.Background = new
        System.Windows.Media.SolidColorBrush(
            Properties.Settings.Default.BackgroundColor);
}
```

7. Press F5 to build and run your application. Note that the title of the window is the value of your *ApplicationName* setting and the background color of your window is the value indicated by the *BackgroundColor* setting. You can change the background color by selecting an item in the *ListBox* and clicking the button. After changing the background color, close the application and restart it. Note that the background color of the application at startup is the same as it was when the previous application session ended.

Lesson Summary

- Settings allow you to persist values between application sessions. You can add new settings at design time by using the Settings Editor.
- Settings can be one of two different scopes. Settings with Application scope are read-only at run time and can be changed only by altering the Settings file between application sessions. Settings with User scope are read-write at run time.
- You can access settings in code through *My.Settings* in Visual Basic, or *Properties.Settings.Default* in C#.

Lesson Review

You can use the following questions to test your knowledge of the information in Lesson 3, "Configuring Application Settings." The questions are also available on the companion CD if you prefer to review them in electronic form.

NOTE Answers

Answers to these questions and explanations of why each answer choice is correct or incorrect are located in the "Answers" section at the end of the book.

---

1. Which of the following code snippets correctly sets the value of a setting called *Title* and persists it?

   A.

```
' VB
My.Settings("Title") = "New Title"
My.Settings.Save
```

```
// C#
Properties.Settings.Default["Title"] = "New Title";
Properties.Settings.Default.Save();
```

B.

```
' VB
My.Settings("Title") = "New Title"

// C#
Properties.Settings.Default["Title"] = "New Title";
```

C.

```
' VB
My.Settings.Title = "New Title"
My.Settings.Save()

// C#
Properties.Settings.Default.Title = "New Title";
Properties.Settings.Default.Save();
```

D.

```
' VB
My.Settings.Title = "New Title"

// C#
Properties.Settings.Default.Title = "New Title";
```

2. Which of the following code snippets reads a setting of type *System.Windows.Media.Color* named *MyColor* correctly?

A.

```
' VB
Dim aColor As System.Windows.Media.Color
aColor = CType(My.Settings.MyColor, System.Windows.Media.Color)

// C#
System.Windows.Media.Color aColor;
aColor = (System.Windows.Media.Color)Properties.Settings.Default.MyColor;
```

B.

```
' VB
Dim aColor As System.Windows.Media.Color
aColor = My.Settings.MyColor.ToColor()

// C#
System.Windows.Media.Color aColor;
aColor = Properties.Settings.Default.MyColor.ToColor();
```

C.

```
' VB
Dim aColor As Object
aColor = My.Settings.MyColor

// C#
Object aColor;
aColor = Properties.Settings.Default.MyColor;
```

D.

```
' VB
Dim aColor As System.Windows.Media.Color
aColor = My.Settings.MyColor

// C#
System.Windows.Media.Color aColor;
aColor = Properties.Settings.Default.MyColor;
```

Chapter Review

To practice and reinforce the skills you learned in this chapter further, you can do any or all of the following:

- Review the chapter summary.
- Review the list of key terms introduced in this chapter.
- Complete the case scenarios. These scenarios set up real-world situations involving the topics of this chapter and ask you to create a solution.
- Complete the suggested practices.
- Take a practice test.

Chapter Summary

- Routed events can be raised by multiple UI elements in the visual tree. Bubbling events are raised first by the element in which they originate and then bubble up through the visual tree. Tunneling events are raised first by the topmost element in the visual tree and tunnel down to the element in which the event originates. Direct events are raised only by the element in which they originate.
- Elements in the visual tree can handle events that they do not themselves define. These are called *attached events*. You can define a handler for an attached event in the XAML that defines the element.
- You can use the *EventManager* class to register a new routed event and to register a class event handler.
- Commands provide an architecture that allows you to define high-level tasks, connect those tasks to a variety of inputs, define handlers that execute code when commands are invoked, and determine when a command is unavailable.
- You can use the built-in library of commands or create custom commands for your application. Commands can be triggered by controls, input gestures, or direct invocation.
- The *CommandBinding* object binds commands to command handlers.
- Settings allow you to create applications that persist between application sessions. Application scope settings are read-only at run time, and user scope settings are read-write at run time.
- Settings are exposed as strongly typed properties on the *My.Settings* object (in Visual Basic) and the *Properties.Settings.Default* object (in C#).

Key Terms

Do you know what these key terms mean? You can check your answers by looking up the terms in the glossary at the end of the book.

- Application Setting
- Bubbling Event
- Command
- Command Handler
- Direct Event
- Event Handler
- Gesture
- Routed Event
- Setting
- Tunneling Event
- User Setting

Case Scenarios

In the following case scenarios, you will apply what you've learned about how to use commands, events, and settings to design user interfaces. You can find answers to these questions in the "Answers" section at the end of this book.

Case Scenario 1: Validating User Input

You're creating a form that will be used by Humongous Insurance data entry personnel to input data. The form consists of several *TextBox* controls that receive input. Data entry is expected to proceed quickly and without errors, but to help ensure this you will be designing validation for this form. This validation is somewhat complex—there is a set of characters that is not allowed in any text box on the form, and each text box has additional limitations that differ from control to control. You would like to implement this validation scheme with a minimum of code in order to make troubleshooting and maintenance simple.

Question

Answer the following question for your manager:

- What strategies can we use to implement these requirements?

Case Scenario 2: Humongous Insurance User Interface

The front end for this database is just as complex as the validation requirements. You are faced with a front end that exposes many menu options. Furthermore, for expert users, some of the more commonly used menu items can be triggered by holding down the Ctrl key while performing various gestures with the mouse. Functionality invoked by the menu items sometimes will be unavailable. Finally, you need to allow the operator to edit data in this window quickly and easily.

Technical Requirements

- All main menu items must have access keys, and some have mouse shortcuts.
- Availability of menu items must be communicated to the user in a way that is easy to understand but does not disrupt program flow.
- You must ensure that when a menu item is unavailable, corresponding shortcut keys and mouse gestures are also inactivated.
- Certain *TextBox* controls on the form must fill in automatically when appropriate keystrokes are entered.

Question

- How can this functionality be implemented?

Suggested Practices

- Create a rudimentary text editor with buttons that implement the Cut, Copy, and Paste commands.
- Create an application that stores a color scheme for each user and automatically loads the correct color scheme when the user opens the application.
- Build an application that consists of a window with a single button that the user can chase around the window with the mouse but can never actually click.

Take a Practice Test

The practice tests on this book's companion CD offer many options. For example, you can test yourself on just the content covered in this chapter, or you can test yourself on all the 70-502 certification exam content. You can set up the test so that it closely simulates the experience of taking a certification exam, or you can set it up in study mode so that you can look at the correct answers and explanations after you answer each question.

MORE INFO Practice tests

For details about all the practice test options available, see the section "How to Use the Practice Tests," in this book's Introduction.
