13

Exception Handling

13.7 Exception Properties

- Class `Exception`'s properties are used to formulate error messages indicating a caught exception.
  - Property `Message` stores the error message associated with an `Exception` object.
  - Property `StackTrace` contains a `string` that represents the **method-call stack**.

13.7 Exception Properties (Cont.)

- When an exception occurs, a programmer might use a different error message or indicate a new exception type.
- The original exception object is stored in the `InnerException` property.
- Class `Exception` provides other properties:
  - `HelpLink` specifies the location of a help file that describes the problem.
  - `Source` specifies the name of the application or object that caused the exception.
  - `TargetSite` specifies the method where the exception originated.

13.8 User-Defined Exception Classes

- In some cases, you might create exception classes specific to the problems that occur in your programs.
- **User-defined exception classes** should derive directly or indirectly from class `Exception` of namespace `System`.
- Exceptions should be documented so that other developers will know how to handle them.

13.8 User-Defined Exception Classes (Cont.)

- User-defined exceptions should define three constructors:
  - a parameterless constructor
  - a constructor that receives a `string` argument (the error message)
  - a constructor that receives a `string` argument and an `Exception` argument (the error message and the inner exception object)

NegativeNumber Exception.cs

(1 of 2)

- Class `NegativeNumberException` (Fig. 13.6) represents exceptions that occur when a program performs an illegal operation on a negative number.

```
// Fig. 13.6: NegativeNumberException.cs
// NegativeNumberException represents exceptions caused by
// illegal operations performed on negative numbers.
using System;

namespace SquareRootTest
{
  class NegativeNumberException : Exception
  {
    // default constructor
    public NegativeNumberException()
      : base( "Illegal operation for a negative number" )
    {
      // empty body
    } // end default constructor

```

Inheriting from class `Exception`.

Parameterless constructor.

Fig. 13.6 | `NegativeNumberException` represents exceptions caused by illegal operations performed on negative numbers. (Part 1 of 2.)

NegativeNumberException.cs

(2 of 2)

```
      // constructor for customizing error message
      public NegativeNumberException( string messageValue )
         : base( messageValue )
      {
         // empty body
      } // end one-argument constructor

      // constructor for customizing the exception's error
      // message and specifying the InnerException object
      public NegativeNumberException( string messageValue,
         Exception inner )
         : base( messageValue, inner )
      {
         // empty body
      } // end two-argument constructor
   } // end class NegativeNumberException
} // end namespace SquareRootTest
```

Constructor with a single argument (the `Message`).

Constructor with two arguments (the `Message` and `InnerException`).

Fig. 13.6 | `NegativeNumberException` represents exceptions caused by illegal operations performed on negative numbers. (Part 2 of 2.)

- Class `SquareRootForm` (Fig. 13.7) demonstrates our user-defined exception class.

SquareRootTest.cs

(1 of 4)

```
// Fig. 13.7: SquareRootTest.cs
// Demonstrating a user-defined exception class.
using System;
using System.Windows.Forms;

namespace SquareRootTest
{
  public partial class SquareRootForm : Form
  {
    public SquareRootForm()
    {
      InitializeComponent();
    } // end constructor

```

Fig. 13.7 | Demonstrating a user-defined exception class. (Part 1 of 4.)

SquareRootTest.cs

(2 of 4)

```
    // computes square root of parameter; throws
    // NegativeNumberException if parameter is negative
    public double SquareRoot( double value )
    {
       // if negative operand, throw NegativeNumberException
       if ( value < 0 )
          throw new NegativeNumberException(
             "Square root of negative number not perm );ted"
       else
          return Math.Sqrt( value ); // compute square root
    } // end method SquareRoot

    // obtain user input, convert to double, calculate square root
    private void squareRootButton_Click( object sender, EventArgs e )
    {
       outputLabel.Text = ""; // clear OutputLabel

       // catch any NegativeNumberException thrown
       try
       {
          double result =
             SquareRoot( Convert.ToDouble( inputTextBox.Text ) );

```

If the numeric value that the user enters is negative, SquareRoot throws a NegativeNumber-Exception.

SquareRoot invokes Math’s Sqrt method.

Fig. 13.7 | Demonstrating a user-defined exception class. (Part 2 of 4.)

SquareRootTest.cs

(3 of 4)

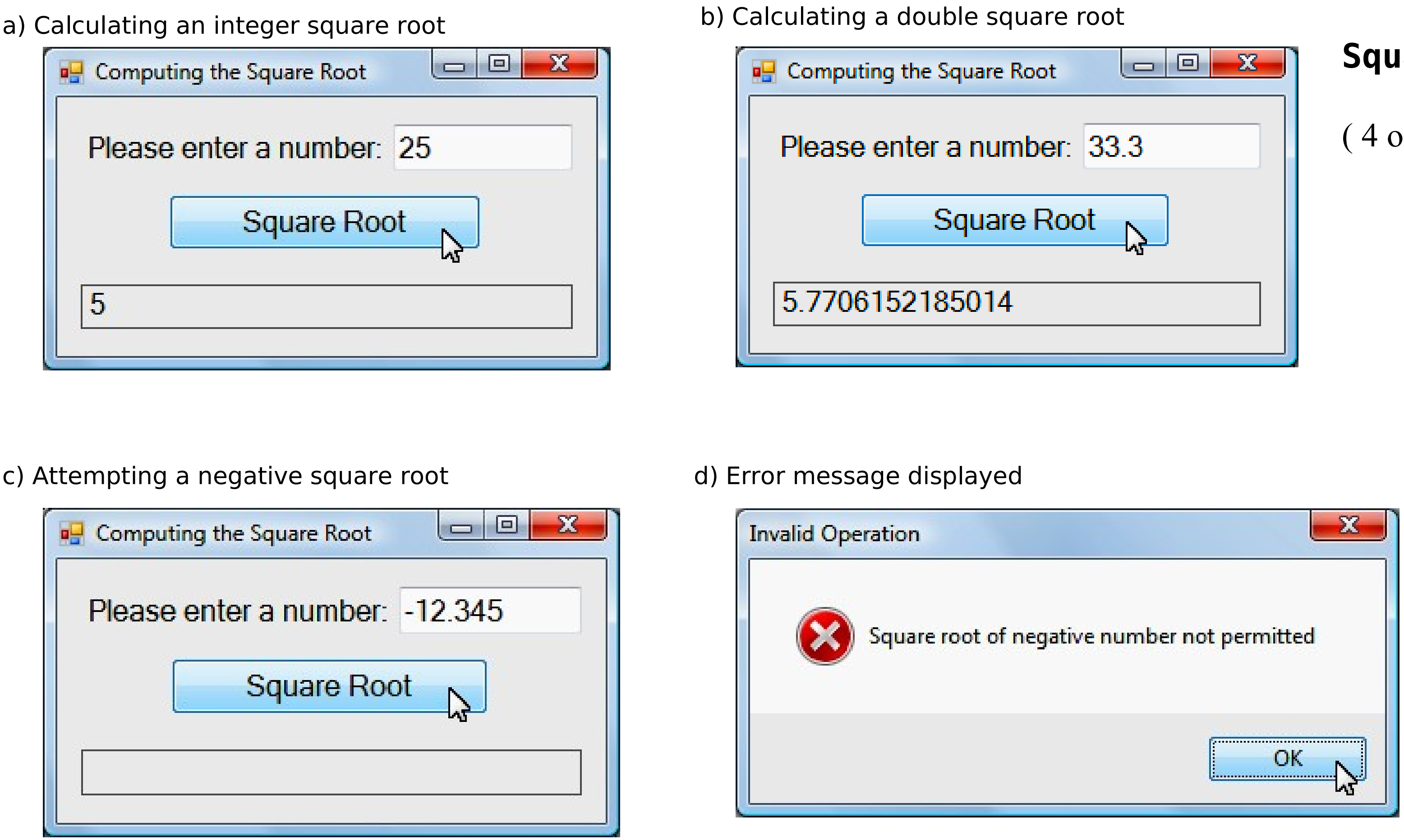
```
               outputLabel.Text = result.ToString();
         } // end try
        catch ( FormatException formatExceptionParameter )
        {
          MessageBox.Show( formatExceptionParameter.Message,
            "Invalid Number Format", MessageBoxButtons.OK,
            MessageBoxIcon.Error );
        } // end catch
        catch ( NegativeNumberException
          negativeNumberExceptionParameter )
        {
          MessageBox.Show( negativeNumberExceptionParameter.Message,
            "Invalid Operation", MessageBoxButtons.OK,
            MessageBoxIcon.Error );
        } // end catch
      } // end method squareRootButton_Click
    } // end class SquareRootForm
} // end namespace SquareRootTest
```

Catching and handling a `NegativeNumber-Exception`.

Fig. 13.7 | Demonstrating a user-defined exception class. (Part 3 of 4.)

SquareRootTest.cs

(4 of 4)



a) Calculating an integer square root

Computing the Square Root

Please enter a number: 25

5

b) Calculating a double square root

Computing the Square Root

Please enter a number: 33.3

5.7706152185014

c) Attempting a negative square root

Computing the Square Root

Please enter a number: -12.345

d) Error message displayed

Invalid Operation

Square root of negative number not permitted

Fig. 13.7 | Demonstrating a user-defined exception class. (Part 4 of 4.)

14

Graphical User Interfaces with Windows Forms: Part 1

… the wisest prophets make sure of the event first.

– Horace Walpole

…The user should feel in control of the computer; not the other way around. This is achieved in applications that embody three qualities: responsiveness, permissiveness, and consistency.

– Inside Macintosh, Volume 1

Apple Computer, Inc. 1985

14.1 Introduction (Cont.)

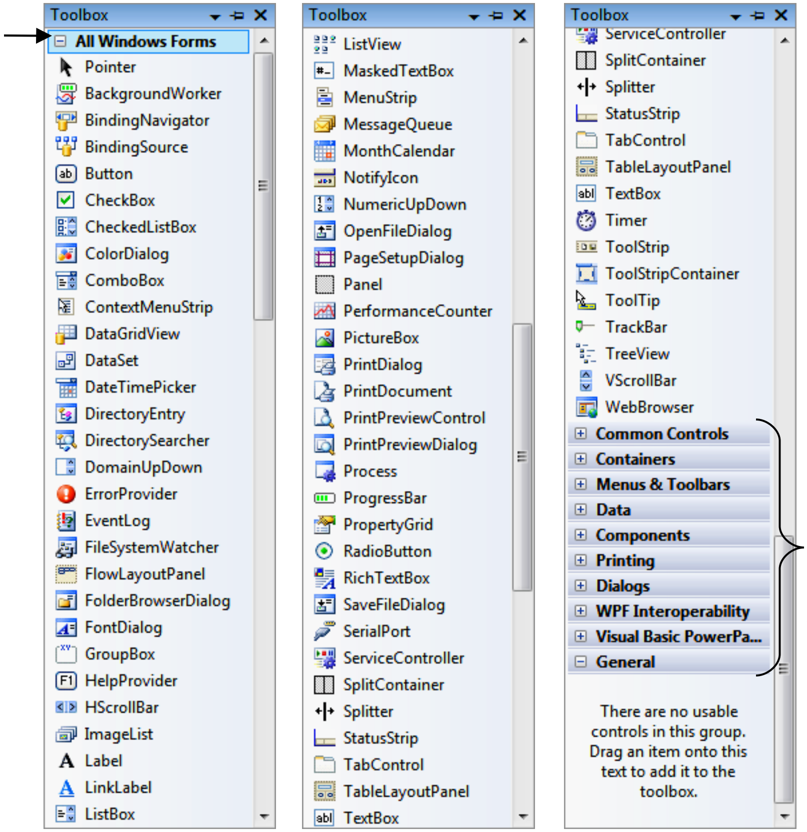
- GUI controls are objects that can display information on the screen or enable users to interact with an application.

Control	Description
Label	Displays images or uneditable text.
TextBox	Enables the user to enter data via the keyboard.
Button	Triggers an event when clicked with the mouse.
CheckBox	Specifies an option that can be checked or not checked.
ComboBox	Provides a drop-down list of items from which the user can make a selection either by clicking an item in the list or by typing in a box.
ListBox	Provides a list of items from which the user can make a selection by clicking an item in the list.
Panel	A container in which controls can be placed and organized.
NumericUpDown	Enables the user to select from a range of numeric input values.

Fig. 14.2 | Some basic GUI controls.

14.2 Windows Forms (Cont.)

Display all controls and components



There are no usable controls in this group. Drag an item onto this text to add it to the toolbox.

Categories that Organize Controls and components by functionality

Fig. 14.3 | Components and controls for Windows Forms.

14.2 Windows Forms (Cont.)

- The **active window** is the frontmost and has a highlighted title bar.
- A window becomes the active window when the user clicks somewhere inside it.
- A `Form` is a **container** for controls and components.
- When you drag a control or component from the **`Toolbox`** on the `Form`, Visual Studio generates code that instantiates the object and sets its basic properties.
- The generated code is placed by the IDE in a separate file using partial classes.

14.2 Windows Forms (Cont.)

- Figure 14.4 lists common `Form` properties, methods and events.

`Form` properties, methods and event	Description
Common Properties	
`AcceptButton`	`Button` that is clicked when *Enter* is pressed.
`AutoScroll`	`Boolean` value that allows or disallows scrollbars when needed.
`CancelButton`	`Button` that is clicked when the *Escape* key is pressed.
`FormBorderStyle`	Border style for the `Form`.
`Font`	Font of text displayed on the `Form`.
`Text`	Text in the `Form`'s title bar.

Fig. 14.4 | Common `Form` properties, methods and an event. (Part 1 of 2.)

14.2 Windows Forms (Cont.)

`Form` properties, methods and an event	Description
Common Methods	
`Close`	Closes a `Form` and releases all resources.
`Hide`	Hides a `Form`, but does not destroy it or release its resources.
`Show`	Displays a hidden `Form`.
Common Event	
`Load`	Occurs before a `Form` is displayed to the user.

Fig. 14.4 | Common `Form` properties, methods and an event. (Part 2 of 2.)

14.3 Event Handling

- GUIs are **event driven**.
- When the user interacts with a GUI component, the **event** drives the program to perform a task.
- A method that performs a task in response to an event is called an **event handler**.

- The application of Fig. 14.5 contains a `Button` that a user clicks to display a `MessageBox`.

SimpleEventExampleForm.cs

(1 of 2)

```
// Fig. 14.5: SimpleEventExampleForm.cs
// Using Visual Studio to create event handlers.
using System;
using System.Windows.Forms;

namespace SimpleEventExample
{
   // Form that shows a simple event handler
   public partial class SimpleEventExampleForm : Form
   {
      // default constructor
      public SimpleEventExampleForm()
      {
         InitializeComponent();
      } // end constructor
```

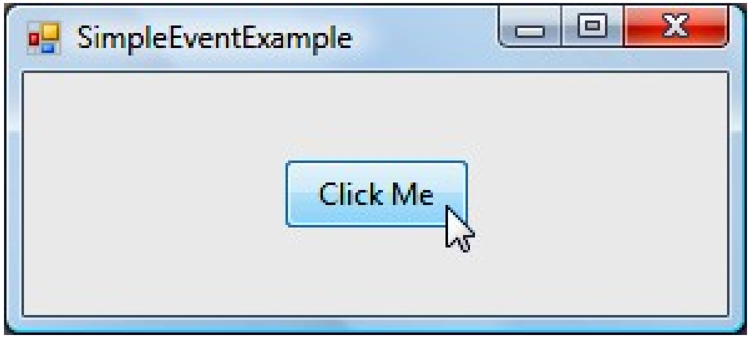
Fig. 14.5 | Simple event-handling example using visual programming. (Part 2 of 2.)

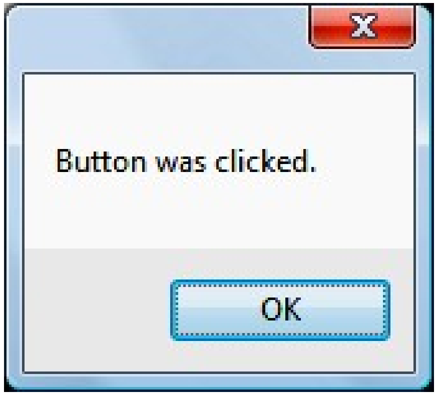
SimpleEventExampleForm.cs

```

      // handles click event of Button clickButton
      private void clickButton_Click( object sender, EventArgs e )
      {
         MessageBox.Show( "Button was clicked." );
      } // end method clickButton_Click
   } // end class SimpleEventExampleForm
} // end namespace SimpleEventExample
```

This event handler is called when the user clicks the button.





Button was clicked.

Fig. 14.5 | Simple event-handling example using visual programming. (Part 2 of 2.)

14.3 Event Handling

- To create the application's event handler, double click the Button on the Form.
- The following empty event handler is declared:

```
private void clickButton_Click ( object
  sender, EventArgs e )
{

} // end method clickButton_Click
```

14.3 Event Handling (Cont.)

- By convention, C# names the event-handler method as *objectName_eventName* (e.g., `clickButton_Click`).
- Each event handler receives two parameters when it is called:
  - An `object` reference named `sender`—a reference to the object that generated the event.
  - A reference to an object of type `EventArgs`, which contains additional information about the event.

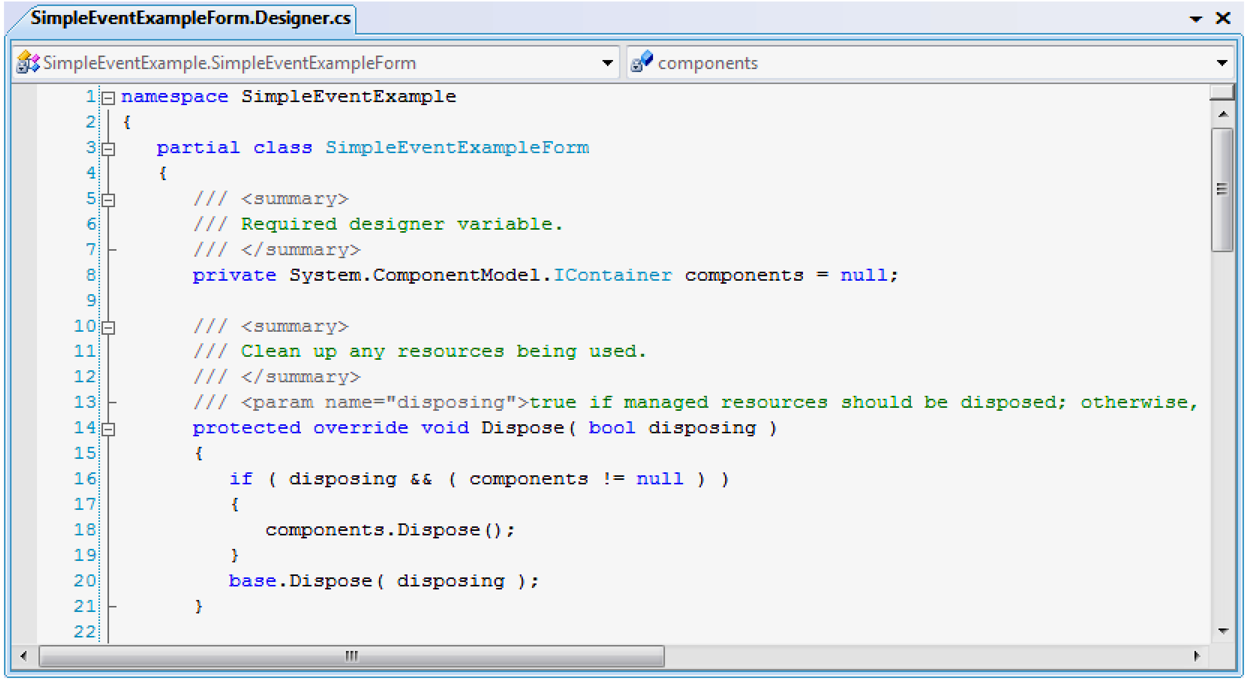
14.3 Event Handling (Cont.)

14.3.2 Another Look at the Visual Studio Generated Code

- Visual Studio generates the code that creates and initializes the GUI.
- This autogenerated code is placed in the `Designer.cs` file of the `Form`.
- Open this file by expanding the node for `SimpleEventExampleForm.cs` and double clicking the file name that ends with `Designer.cs`.

14.3 Event Handling (Cont.)

- Since this code (Figs. 14.6 and 14.7) is created and maintained by Visual Studio, you generally don’t need to look at it.


SimpleEventExampleForm.Designer.cs
SimpleEventExample.SimpleEventExampleForm
components

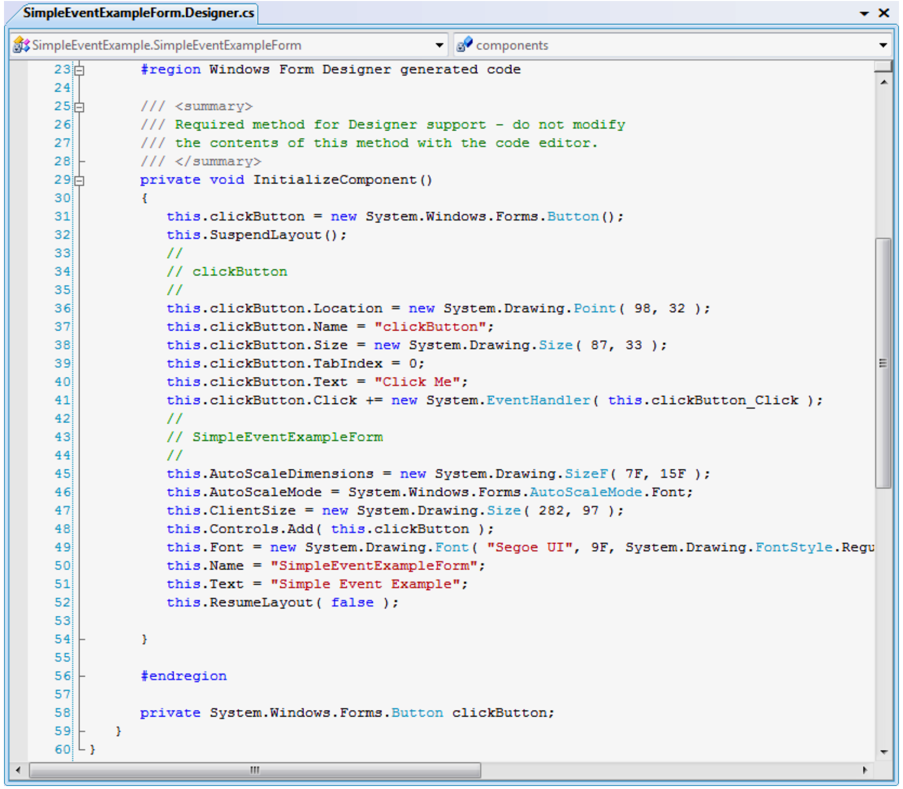
```
namespace SimpleEventExample
{
   partial class SimpleEventExampleForm
   {
      /// <summary>
      /// Required designer variable.
      /// </summary>
      private System.ComponentModel.IContainer components = null;

      /// <summary>
      /// Clean up any resources being used.
      /// </summary>
      /// <param name="disposing">true if managed resources should be disposed; otherwise,
      protected override void Dispose( bool disposing )
      {
         if ( disposing && ( components != null ) )
         {
            components.Dispose();
         }
         base.Dispose( disposing );
      }

```

Fig. 14.6 | First half of the Visual Studio generated code file.

14.3 Event Handling (Cont.)


SimpleEventExampleForm.Designer.cs
SimpleEventExample.SimpleEventExampleForm
components

```
      #region Windows Form Designer generated code

      /// <summary>
      /// Required method for Designer support - do not modify
      /// the contents of this method with the code editor.
      /// </summary>
      private void InitializeComponent()
      {
         this.clickButton = new System.Windows.Forms.Button();
         this.SuspendLayout();
         //
         // clickButton
         //
         this.clickButton.Location = new System.Drawing.Point( 98, 32 );
         this.clickButton.Name = "clickButton";
         this.clickButton.Size = new System.Drawing.Size( 87, 33 );
         this.clickButton.TabIndex = 0;
         this.clickButton.Text = "Click Me";
         this.clickButton.Click += new System.EventHandler( this.clickButton_Click );
         //
         // SimpleEventExampleForm
         //
         this.AutoScaleDimensions = new System.Drawing.SizeF( 7F, 15F );
         this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
         this.ClientSize = new System.Drawing.Size( 282, 97 );
         this.Controls.Add( this.clickButton );
         this.Font = new System.Drawing.Font( "Segoe UI", 9F, System.Drawing.FontStyle.Regu
         this.Name = "SimpleEventExampleForm";
         this.Text = "Simple Event Example";
         this.ResumeLayout( false );

      }

      #endregion

      private System.Windows.Forms.Button clickButton;
   }
}
```

Fig. 14.7 | Second half of the Visual Studio generated code file.

14.3 Event Handling (Cont.)

- The `partial` modifier allows the `Form`'s class to be split among multiple files.
- Note that `clickButton` is declared as a `private` instance variable.
- The property values correspond to the values set in the **`Properties`** window for each control.
- Method `InitializeComponent` is called when the `Form` is created.

Outline

- The application of Fig. 14.5 contains a `Button` that a user clicks to display a `MessageBox`.

SimpleEventExampleForm.cs

(1 of 2)

```
// Fig. 14.5: SimpleEventExampleForm.cs
// Using Visual Studio to create event handlers.
using System;
using System.Windows.Forms;

namespace SimpleEventExample
{
   // Form that shows a simple event handler
   public partial class SimpleEventExampleForm : Form
   {
      // default constructor
      public SimpleEventExampleForm()
      {
         InitializeComponent();
      } // end constructor
```

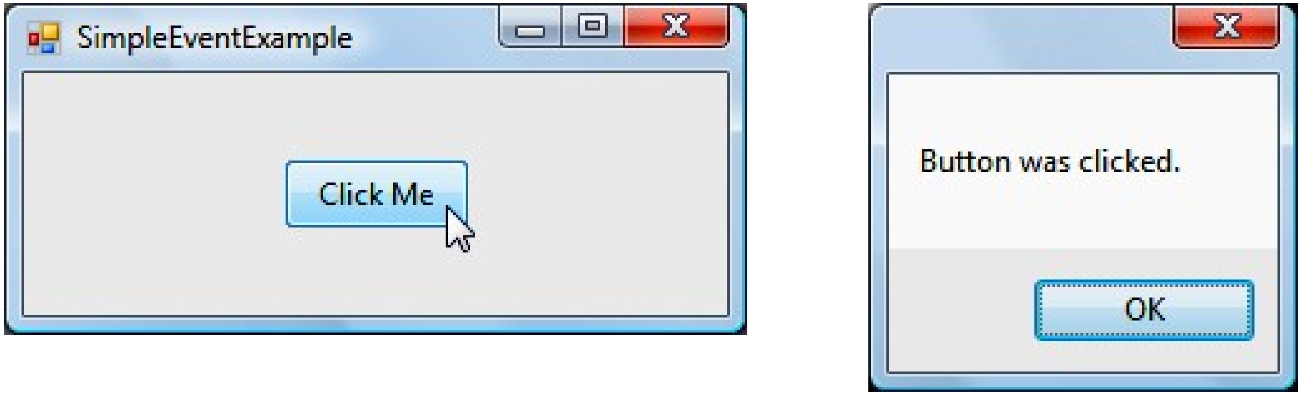
Fig. 14.5 | Simple event-handling example using visual programming. (Part 2 of 2.)

SimpleEventExampleForm.cs

```

      // handles click event of Button clickButton
      private void clickButton Click( object sender, EventArgs e )
      {
         MessageBox.Show( "Button was clicked." );
      } // end method clickButton Click
   } // end class SimpleEventExampleForm
} // end namespace SimpleEventExample
```

This event handler is called when the user clicks the button.



Button was clicked.

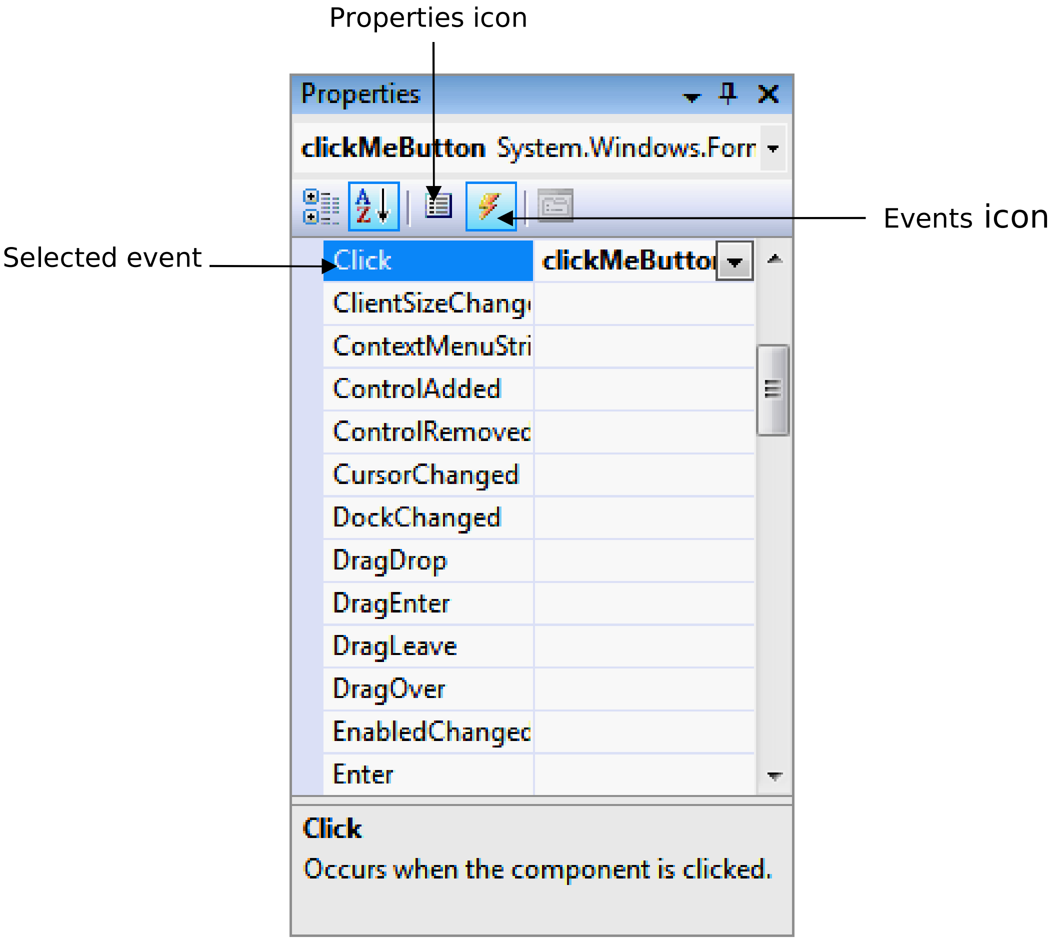
Fig. 14.5 | Simple event-handling example using visual programming. (Part 2 of 2.)

14.3 Event Handling (Cont.)

14.3.4 Other Ways to Create Event Handlers

- Typically, controls can generate many different types of events.
- Clicking the **Events** icon (the lightning-bolt icon) in the **Properties** window (Fig. 14.8), displays all the events for the selected control.

14.3 Event Handling (Cont.)



Click	clickMeButto
ClientSizeChang	
ContextMenuStri	
ControlAdded	
ControlRemovec	
CursorChanged	
DockChanged	
DragDrop	
DragEnter	
DragLeave	
DragOver	
EnabledChangec	
Enter	

Click
Occurs when the component is clicked.

Fig. 14.8 | Viewing events for a Button control in the Properties window.

14.3 Event Handling (Cont.)

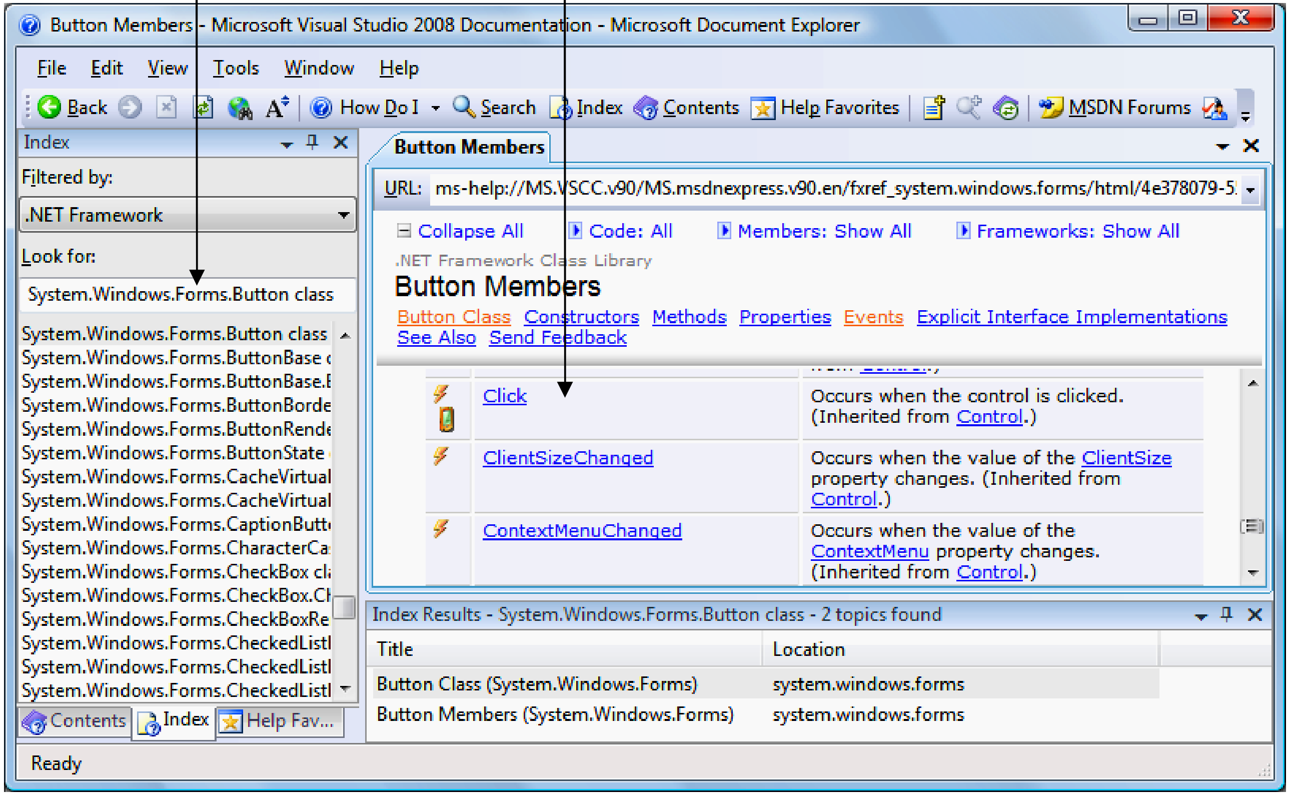
14.3.5 Locating Event Information

- To learn about the events raised by a control, select **Help > Index**.
- In the window, select **.NET Framework** in the **Filtered by** drop-down list and enter the name of the control's class in the **Index** window.
- To display a list of all the class's members (Fig. 14.9), click the **Members** link.

14.3 Event Handling (Cont.)

Class name

List of events



Filtered by:

.NET Framework

Look for:

System.Windows.Forms.Button class

System.Windows.Forms.Button class
System.Windows.Forms.ButtonBase c
System.Windows.Forms.ButtonBase.E
System.Windows.Forms.ButtonBorde
System.Windows.Forms.ButtonRende
System.Windows.Forms.ButtonState
System.Windows.Forms.CacheVirtual
System.Windows.Forms.CacheVirtual
System.Windows.Forms.CaptionButt
System.Windows.Forms.CharacterCa
System.Windows.Forms.CheckBox cla
System.Windows.Forms.CheckBox.Ch
System.Windows.Forms.CheckBoxRe
System.Windows.Forms.CheckedListI
System.Windows.Forms.CheckedListI
System.Windows.Forms.CheckedListI

URL: ms-help://MS.VSCC.v90/MS.msdnexpress.v90.en/fxref_system.windows.forms/html/4e378079-5

⊟ Collapse All   Code: All   Members: Show All   Frameworks: Show All

.NET Framework Class Library

Button Members

Button Class Constructors Methods Properties Events Explicit Interface Implementations
See Also Send Feedback

⚡	Click	Occurs when the control is clicked. (Inherited from Control.)
⚡	ClientSizeChanged	Occurs when the value of the ClientSize property changes. (Inherited from Control.)
⚡	ContextMenuChanged	Occurs when the value of the ContextMenu property changes. (Inherited from Control.)

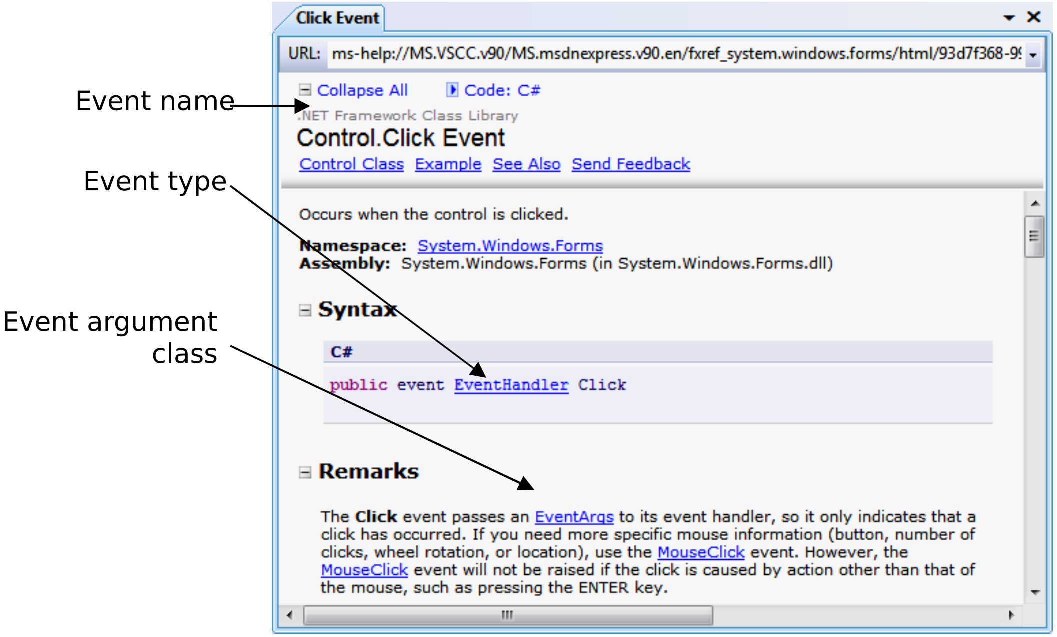
Index Results - System.Windows.Forms.Button class - 2 topics found

Title	Location
Button Class (System.Windows.Forms)	system.windows.forms
Button Members (System.Windows.Forms)	system.windows.forms

Fig. 14.9 | List of Button events.

14.3 Event Handling (Cont.)

- Click the name of an event to view its description and examples of its use (Fig. 14.10).



Event name
Event type
Event argument class
Click Event
URL: ms-help://MS.VSCC.v90/MS.msdnexpress.v90.en/fxref_system.windows.forms/html/93d7f368-9
Collapse All
Code: C#
.NET Framework Class Library

Control.Click Event

Control Class Example See Also Send Feedback

Occurs when the control is clicked.

Namespace: System.Windows.Forms
Assembly: System.Windows.Forms (in System.Windows.Forms.dll)

Syntax

C#

```
public event EventHandler Click
```

Remarks

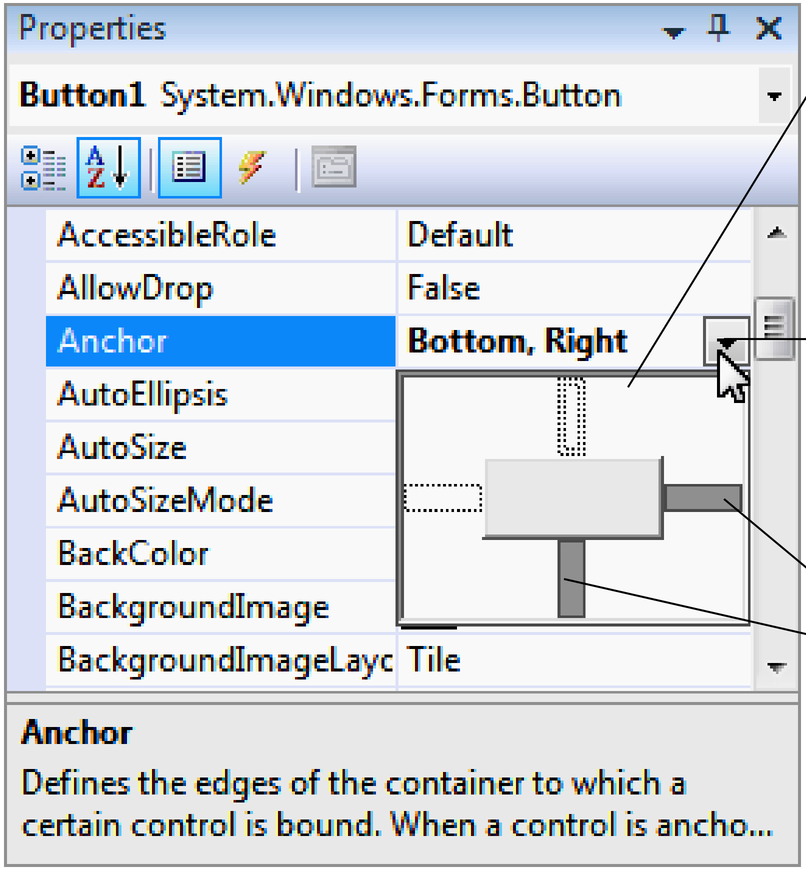
The **Click** event passes an EventArgs to its event handler, so it only indicates that a click has occurred. If you need more specific mouse information (button, number of clicks, wheel rotation, or location), use the MouseClick event. However, the MouseClick event will not be raised if the click is caused by action other than that of the mouse, such as pressing the ENTER key.

Fig. 14.10 | `Click` event details.

14.4 Control Properties and Layout (Cont.)

- **Anchoring** causes controls to remain at a fixed distance from the sides of the container.
- Anchor a control to the right and bottom sides by setting the **Anchor** property (Fig. 14.12).

14.4 Control Properties and Layout (Cont.)



Properties

Button1 System.Windows.Forms.Button

AccessibleRole	Default
AllowDrop	False
Anchor	**Bottom, Right**
AutoEllipsis	
AutoSize	
AutoSizeMode	
BackColor	
BackgroundImage	
BackgroundImageLayc	Tile

Anchor

Defines the edges of the container to which a certain control is bound. When a control is ancho...

Anchoring window

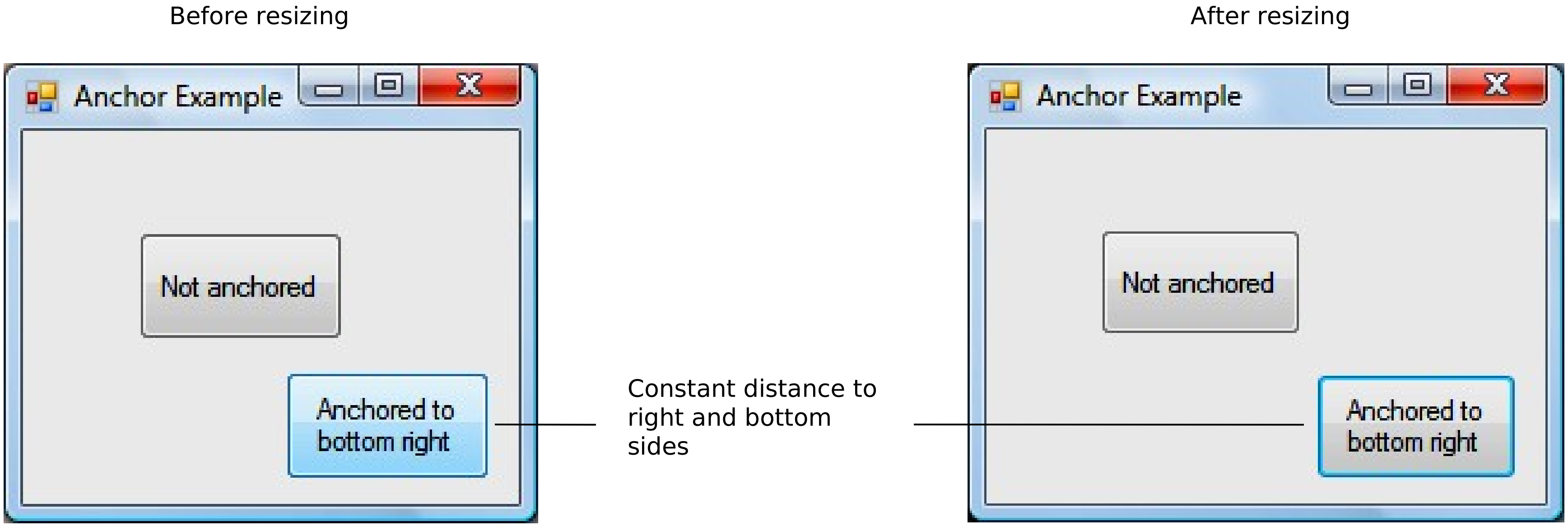
Click the down-arrow in the Anchor property to display the anchoring window

Darkened bars indicate the container’s side(s) to which the control is anchored; use mouse clicks to select or deselect a bar

Fig. 14.12 | Manipulating the Anchor property of a control.

14.4 Control Properties and Layout (Cont.)

- Execute the application and enlarge the Form (Fig. 14.13).



Before resizing

After resizing

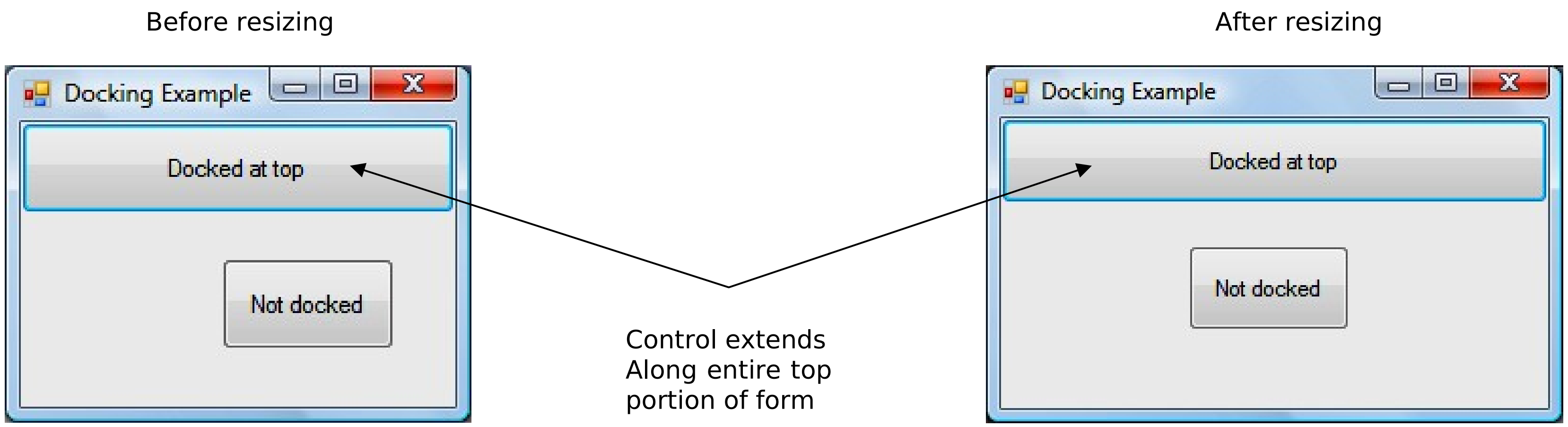
Constant distance to right and bottom sides

Fig. 14.13 | Anchoring demonstration.

14.4 Control Properties and Layout (Cont.)

- Docking allows a control to span an entire side of its parent container or to fill the entire container (Fig. 14.14).
- The `Form`'s **`Padding`** property specifies the distance between the docked controls and the edges.

14.4 Control Properties and Layout (Cont.)



Before resizing

After resizing

Control extends
Along entire top
portion of form

Fig. 14.14 | Docking a `Button` to the top of a `Form`.

14.4 Control Properties and Layout (Cont.)

Control layout properties	Description
Anchor	Causes a control to remain at a fixed distance from the side(s) of the container.
Dock	Allows a control to span one side of its container or to fill the remaining space in the container.
Padding	Sets the space between a container's edges and docked controls.
Location	Specifies the location of the upper-left corner of the control, in relation to its container.
Size	Specifies the size of the control in pixels as a Size object, which has properties Width and Height.
MinimumSize, MaximumSize	Indicates the minimum and maximum size of a Control.

Fig. 14.15 | Control layout properties.

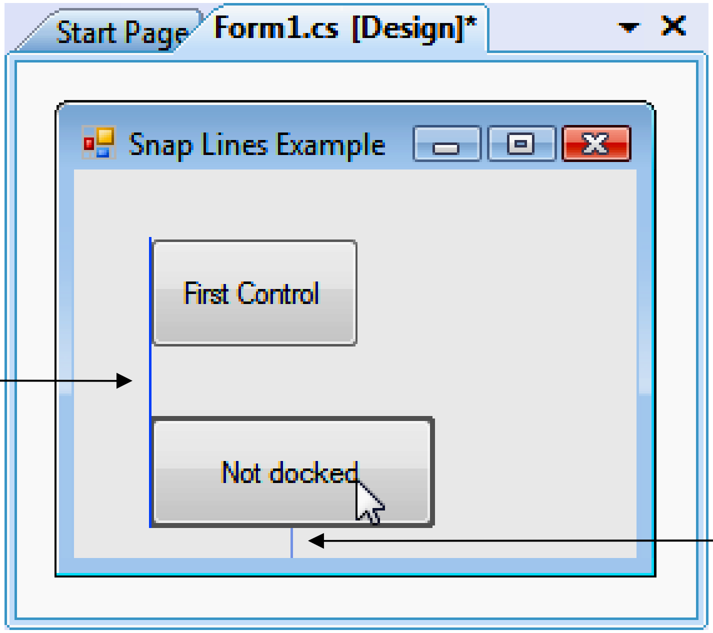
14.4 Control Properties and Layout (Cont.)

Look-and-Feel Observation 14.2

For resizable `Forms`, ensure that the GUI layout appears consistent across various `Form` sizes.

- Visual Studio provides tools that help you with GUI layout.
- When dragging a control across a `Form`, blue lines (known as **snap lines**) help you position the control (Fig. 14.16).
- Visual Studio also provides the **Format** menu, which contains several options for modifying your GUI's layout.

14.4 Control Properties and Layout (Cont.)



Snap line to help align controls on their left sides

Snap line that indicates when a control reaches the minimum recommended distance from the edge of a `Form`.

Fig. 14.16 | Snap lines in Visual Studio 2008.

14.5 Labels, TextBoxes and Buttons

- Label displays text that the user cannot directly modify.

Common Label properties	Description
Font	The font of the text on the Label.
Text	The text on the Label.
TextAlign	The alignment of the Label's text on the control.

Fig. 14.17 | Common Label properties.

14.5 Labels, TextBoxes and Buttons (Cont.)

- A `TextBox` (Fig. 14.18) is an area in which either text can be displayed by a program or the user can type text via the keyboard.
- If you set the property **`UseSystemPasswordChar`** to `True`, the `TextBox` becomes a **password `TextBox`**.

14.5 Labels, TextBoxes and Buttons (Cont.)

TextBox properties and events	Description
Common Properties	
AcceptsReturn	If true in a multiline TextBox, pressing *Enter* in the TextBox creates a new line.
Multiline	If true, the TextBox can span multiple lines. The default value is false.
ReadOnly	If true, the TextBox has a gray background, and its text cannot be edited. The default value is false.
ScrollBars	For multiline textboxes, this property indicates which scrollbars appear.
Text	The TextBox's text content.
UseSystemPasswordChar	When this property is set to True, the TextBox becomes a password TextBox.
Common Event	
TextChanged	Generated when the text changes in a TextBox.

Fig. 14.18 | TextBox properties and events.

14.5 Labels, TextBoxes and Buttons (Cont.)

- Figure 14.19 lists common properties and a common event of class Button.

Button properties and event	Description
Common Properties	
Text	Specifies the text displayed on the Button face.
FlatStyle	Modifies a Button's appearance.
Common Event	
Click	Generated when the user clicks the Button.

Fig. 14.19 | Button properties and event.

Outline

- Figure 14.20 uses a TextBox, a Button and a Label.

LabelTextBoxButtonTestForm.cs

(1 of 2)

```
// Fig. 14.20: LabelTextBoxButtonTestForm.cs
// Using a TextBox, Label and Button to display
// the hidden text in a password TextBox.
using System;
using System.Windows.Forms;

namespace LabelTextBoxButtonTest
{
   // Form that creates a password TextBox and
   // a Label to display TextBox contents
   public partial class LabelTextBoxButtonTestForm : Form
   {
      // default constructor
      public LabelTextBoxButtonTestForm()
      {
         InitializeComponent();
      } // end constructor

```

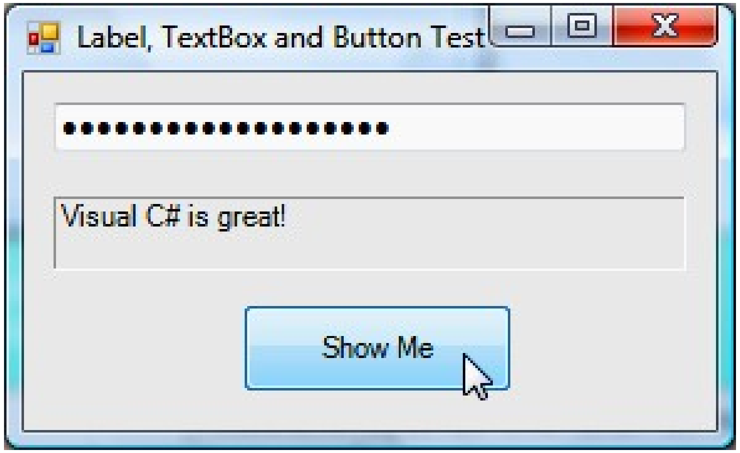
Fig. 14.20 | Program to display hidden text in a password box. (Part 1 of 2.)

```
      // display user input in Label
      private void displayPasswordButton Click(
         object sender, EventArgs e )
      {
         // display the text that the user typed
         displayPasswordLabel.Text = inputPasswordTextBox.Text;
      } // end method displayPasswordButton Click
   } // end class LabelTextBoxButtonTestForm
} // end namespace LabelTextBoxButtonTest
```

LabelTextBoxButtonTestForm.cs

(2 of 2)

The event handler obtains the hidden text entered by the user and displays it in a Label.



••••••••••••••••••••

Visual C# is great!

Fig. 14.20 | Program to display hidden text in a password box. (Part 2 of 2.)

14.6 GroupBoxes and Panels

- **GroupBoxes** and **Panels** arrange related controls on a GUI.
- All of the controls in a GroupBox or Panel move together when the GroupBox or Panel is moved.
- The primary difference is that GroupBoxes can display a caption and do not include scrollbars, whereas Panels can include scrollbars and do not include a caption.

14.6 GroupBoxes and Panels (Cont.)

Look-and-Feel Observation 14.4

`Panels` and `GroupBoxes` can contain other `Panels` and `GroupBox`es for more complex layouts.

Look-and-Feel Observation 14.5

You can organize a GUI by anchoring and docking controls inside a `GroupBox` or `Panel`. The `GroupBox` or `Panel` then can be anchored or docked inside a `Form`. This divides controls into functional "groups" that can be arranged easily.

14.6 GroupBoxes and Panels (Cont.)

GroupBox properties	Description
Controls	The set of controls that the GroupBox contains.
Text	Specifies the caption text displayed at the top of the GroupBox.

Fig. 14.21 | GroupBox properties.

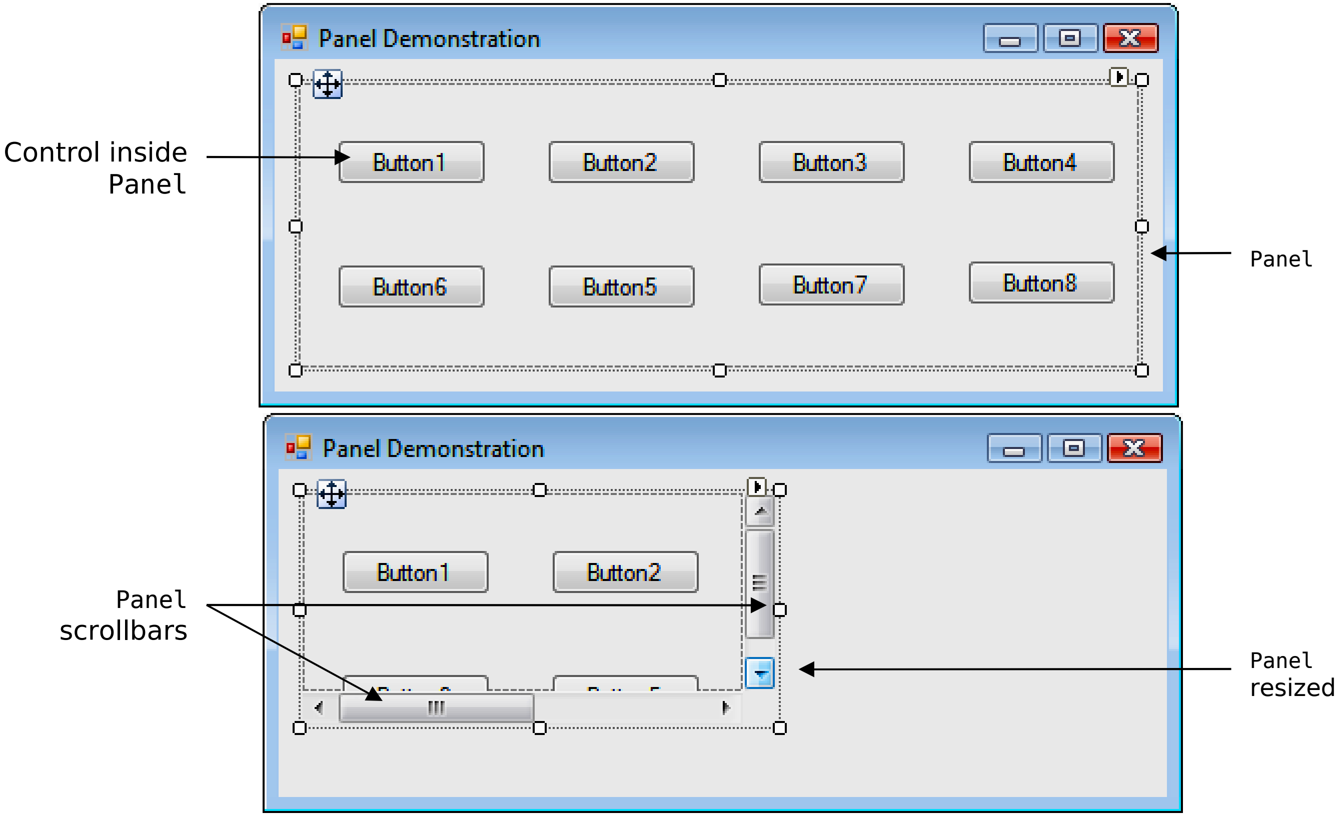
Panel properties	Description
AutoScroll	Indicates whether scrollbars appear when the Panel is too small to display all of its controls.
BorderStyle	Sets the border of the Panel.
Controls	The set of controls that the Panel contains.

Fig. 14.22 | Panel properties.

14.6 GroupBoxes and Panels (Cont.)

- To create a `GroupBox` or `Panel`, drag its icon from the **Toolbox** onto a `Form`.
- Then, drag new controls from the **Toolbox** directly into the `GroupBox` or `Panel`.
- To enable the scrollbars, set the `Panel`'s `AutoScroll` property to `true`.
- If the `Panel` cannot display all of its controls, scrollbars appear (Fig. 14.23).

14.6 GroupBoxes and Panels (Cont.)



Control inside Panel

Panel

Panel scrollbars

Panel resized

Fig. 14.23 | Creating a Panel with scrollbars.

- The program in Fig. 14.24 uses a `GroupBox` and a `Panel` to arrange `Buttons`.

GroupboxPanel ExampleForm.cs

(1 of 3)

```
// Fig. 14.24: GroupboxPanelExampleForm.cs
// Using GroupBoxes and Panels to hold Buttons.
using System;
using System.Windows.Forms;

namespace GroupBoxPanelExample
{
   // Form that displays a GroupBox and a Panel
   public partial class GroupBoxPanelExampleForm : Form
   {
      // default constructor
      public GroupBoxPanelExampleForm()
      {
         InitializeComponent();
      } // end constructor

      // event handler for Hi Button
      private void hiButton_Click( object sender, EventArgs e )
      {
         messageLabel.Text = "Hi pressed"; // change text in Label
      } // end method hiButton_Click
```

The event handler for `hiButton` changes the `Label`'s `Text` property.

Fig. 14.24 | Program to display hidden text in a password box. (Part 1 of 3.)

Outline

GroupboxPanel ExampleForm.cs

(2 of 3)

```
      // event handler for Bye Button
      private void byeButton Click( object sender, EventArgs e )
      {
         messageLabel.Text = "Bye pressed"; // change text in Label
      } // end method byeButton Click

      // event handler for Far Left Button
      private void leftButton Click( object sender, EventArgs e )
      {
         messageLabel.Text = "Far left pressed"; // change text in Label
      } // end method leftButton Click

      // event handler for Far Right Button
      private void rightButton Click( object sender, EventArgs e )
      {
         messageLabel.Text = "Far right pressed"; // change text in Label
      } // end method rightButton Click
   } // end class GroupBoxPanelExampleForm
} // end namespace GroupBoxPanelExample
```

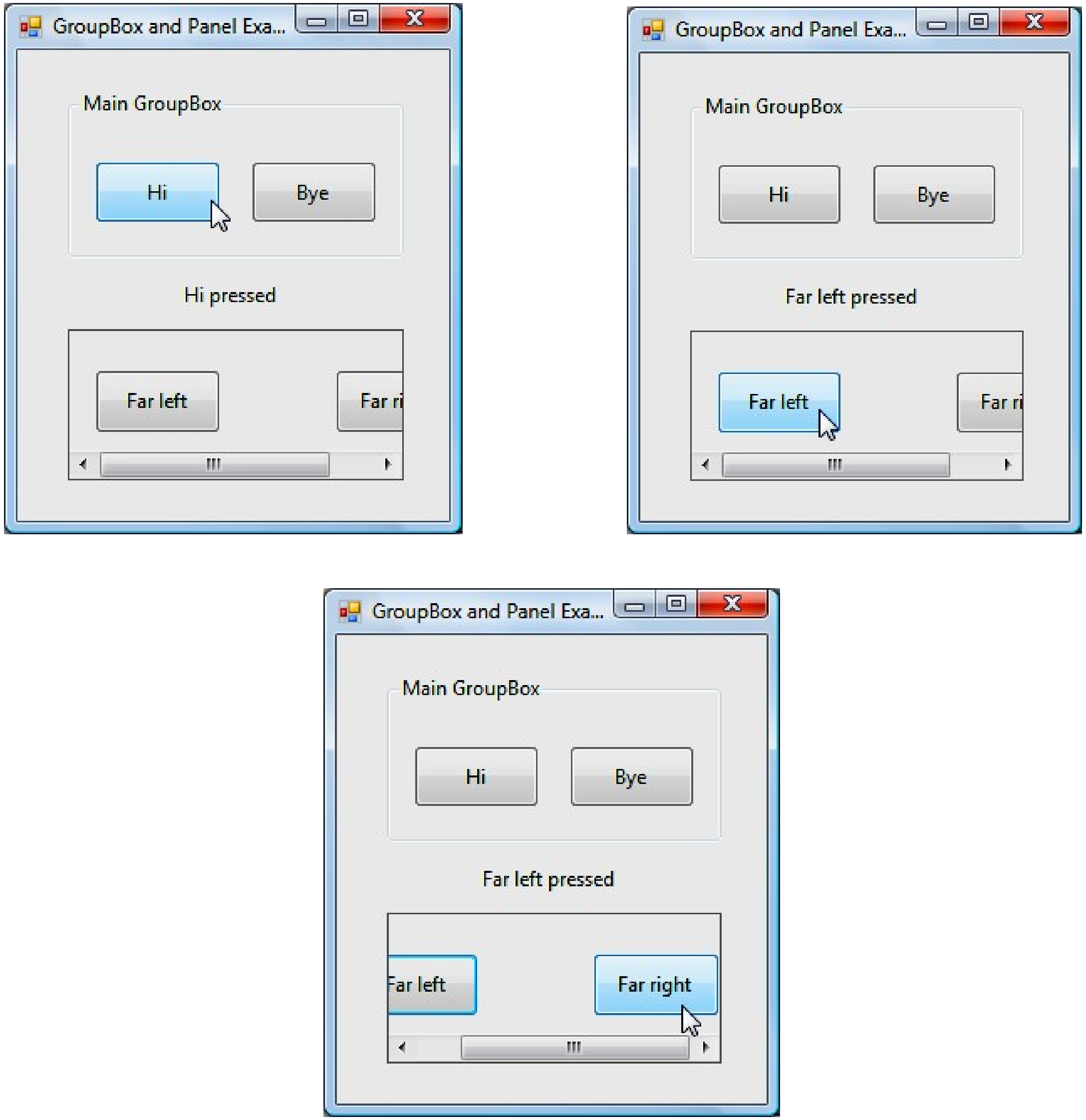
Each Button's Click event changes the Label's Text.

Fig. 14.24 | Program to display hidden text in a password box. (Part 2 of 3.)

Outline

`GroupboxPanel ExampleForm.cs`

(3 of 3)



Hi pressed

Far left pressed

Far left pressed

Fig. 14.24 | Program to display hidden text in a password box. (Part 3 of 3.)

14.7 CheckBoxes and RadioButtons

- A `CheckBox` is a small square that either is blank or contains a check mark.
- Any number of `CheckBox`es can be selected at a time.

14.7 CheckBoxes and RadioButtons (Cont.)

CheckBox properties and events	Description
Common Properties	
Appearance	By default, this property is set to Normal. If it is set to Button, the CheckBox displays as a Button that looks pressed when the CheckBox is checked.
Checked	Indicates whether the CheckBox is checked with a Boolean value.
CheckState	Indicates whether the CheckBox is checked or unchecked with a value from the CheckState enumeration.
Text	Specifies the text displayed to the right of the CheckBox.
ThreeState	When this property is True, the CheckBox has three states—checked, unchecked, and indeterminate.
Common Events	
CheckedChanged	Generated when the Checked property changes.
CheckStateChanged	Generated when the CheckState property changes.

Fig. 14.25 | CheckBox properties and events.

Outline

- The program in Fig. 14.26 allows the user to select CheckBoxes to change a Label's font style.

CheckBoxTestForm.cs

(1 of 3)

```
// Fig. 14.26: CheckBoxTestForm.cs
// Using CheckBoxes to toggle italic and bold styles.
using System;
using System.Drawing;
using System.Windows.Forms;

namespace CheckBoxTest
{
   // Form contains CheckBoxes to allow the user to modify sample text
   public partial class CheckBoxTestForm : Form
   {
      // default constructor
      public CheckBoxTestForm()
      {
         InitializeComponent();
      } // end constructor
```

Fig. 14.26 | Using CheckBoxes to change font styles. (Part 1 of 3.)

CheckBoxTestForm .cs

(2 of 3)

```

      // toggle the font style between bold and
      // not bold based on the current setting
      private void boldCheckBox_CheckedChanged(
         object sender, EventArgs e )
      {
         outputLabel.Font = new Font( outputLabel.Font,
            outputLabel.Font.Style ^ FontStyle.Bold );
      } // end method boldCheckBox_CheckedChanged

      // toggle the font style between italic and
      // not italic based on the current setting
      private void italicCheckBox_CheckedChanged(
         object sender, EventArgs e )
      {
         outputLabel.Font = new Font( outputLabel.Font,
            outputLabel.Font.Style ^ FontStyle.Italic );
      } // end method italicCheckBox_CheckedChanged
   } // end class CheckBoxTestForm
} // end namespace CheckBoxTest
```

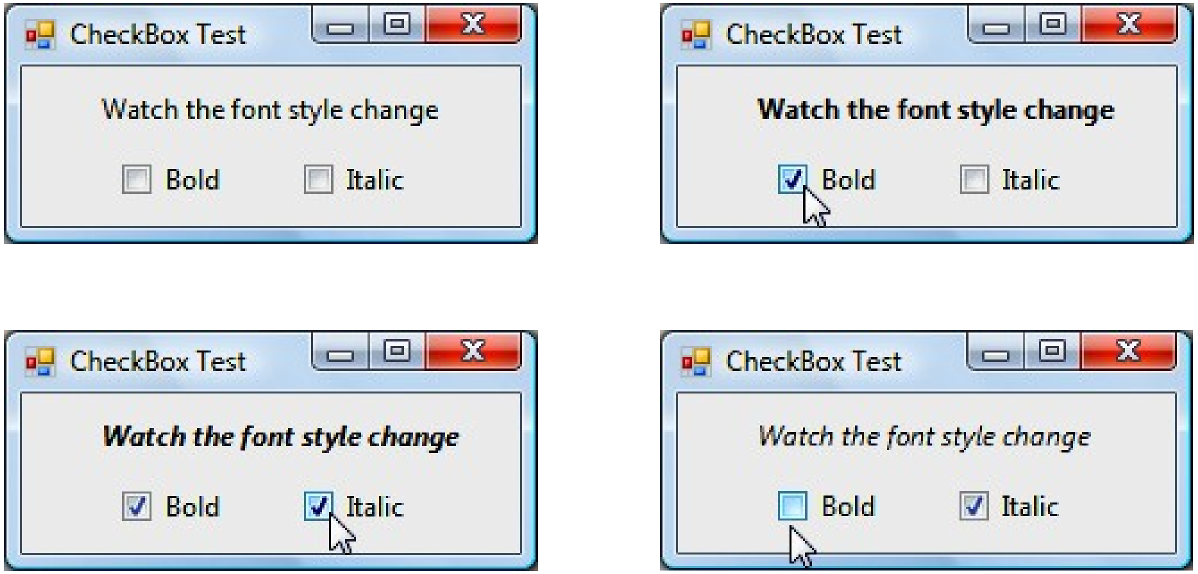
The boldCheckBox sets the Label’s Text property to Bold.

The italicCheckBox sets the Label’s Text property to Italic.

Fig. 14.26 | Using CheckBoxes to change font styles. (Part 2 of 3.)

CheckBoxTestForm.cs

(3 of 3)



CheckBox Test

Watch the font style change

□ Bold □ Italic

CheckBox Test

Watch the font style change

☑ Bold □ Italic

CheckBox Test

Watch the font style change

☑ Bold ☑ Italic

CheckBox Test

Watch the font style change

□ Bold ☑ Italic

Fig. 14.26 | Using CheckBoxes to change font styles. (Part 3 of 3.)

14.7 CheckBoxes and RadioButtons (Cont.)

- To change the font style on a `Label`, you must set its `Font` property to a new **`Font` object**.
- The `Font` constructor we used takes the current font and the new style as arguments.
- Styles can be combined via **bitwise operators**—operators that perform manipulation on bits of information.
- We needed to set the `FontStyle` so that the text appears in bold if it was not bold originally, and vice versa
  - The logical exclusive OR operator makes toggling the text style simple.

14.7 CheckBoxes and RadioButtons (Cont.)

- Radio buttons are similar to `CheckBox`es in that they also have two states—**selected** and **not selected**.
- `RadioButton`s normally appear as a **group**, in which only one `RadioButton` can be selected at a time.
- All `RadioButton`s added to a container become part of the same group.

14.7 CheckBoxes and RadioButtons (Cont.)

Look-and-Feel Observation 14.7

Use `RadioButton`s when the user should choose only one option in a group.

Look-and-Feel Observation 14.8

Use `CheckBox`es when the user should be able to choose multiple options in a group.

14.7 CheckBoxes and RadioButtons (Cont.)

RadioButton properties and event	Description
Common Properties	
Checked	Indicates whether the RadioButton is checked.
Text	Specifies the RadioButton's text.
Common Event	
CheckedChanged	Generated every time the RadioButton is checked or unchecked.

Fig. 14.27 | RadioButton properties and events.

Software Engineering Observation 14.2

Forms, GroupBoxes, and Panels can act as logical groups for RadioButtons. The RadioButtons within each group are mutually exclusive to each other, but not to RadioButtons in different logical groups.

Outline

- The program in Fig. 14.28 uses `RadioButtons` to enable users to select options for a `MessageBox`.

RadioButtonsTestForm.cs

(1 of 8)

```
// Fig. 14.28: RadioButtonsTestForm.cs
// Using RadioButtons to set message-window options.
using System;
using System.Windows.Forms;

namespace RadioButtonsTest
{
   // Form contains several RadioButtons--user chooses one
   // from each group to create a custom MessageBox
   public partial class RadioButtonsTestForm : Form
   {
      // create variables that store the user's choice of options
      private MessageBoxIcon iconType;
      private MessageBoxButtons buttonType;

      // default constructor
      public RadioButtonsTestForm()
      {
         InitializeComponent();
      } // end constructor
```

Initializing variables for the `MessageBoxIcon` and `MessageBoxButtons` selections.

Fig. 14.28 | Using `RadioButtons` to set message-window options. (Part 1 of 8.)

Outline

RadioButtonsTestForm.cs

(5 of 8)

```

         // display stop Icon
         else if ( sender == stopRAdioButton )
            iconType = MessageBoxIcon.Stop;

         // only one option left--display warning Icon
         else
            iconType = MessageBoxIcon.Warning;
      } // end method iconType_CheckChanged

      // display MessageBox and Button user pressed
      private void displayButton_Click( object sender, EventArgs e )
      {
         // display MessageBox and store
         // the value of the Button that was pressed
         DialogResult result = MessageBox.Show(
            "This is your Custom MessageBox.", "Custon MessageBox",
            buttonType, iconType, 0, 0 );
```

Displaying a `MessageBox` with specified icon and button options.

Fig. 14.28 | Using `RadioButtons` to set message-window options. (Part 5 of 8.)

RadioButtonsTestForm.cs

(6 of 8)

```

// check to see which Button was pressed in the MessageBox
// change text displayed accordingly
switch (result)
{
   case DialogResult.OK:
      displayLabel.Text = "OK was pressed.";
      break;
   case DialogResult.Cancel:
      displayLabel.Text = "Cancel was pressed.";
      break;
   case DialogResult.Abort:
      displayLabel.Text = "Abort was pressed.";
      break;
   case DialogResult.Retry:
      displayLabel.Text = "Retry was pressed.";
      break;
   case DialogResult.Ignore:
      displayLabel.Text = "Ignore was pressed.";
      break;
```

Testing for the dialog result and displaying appropriate text.

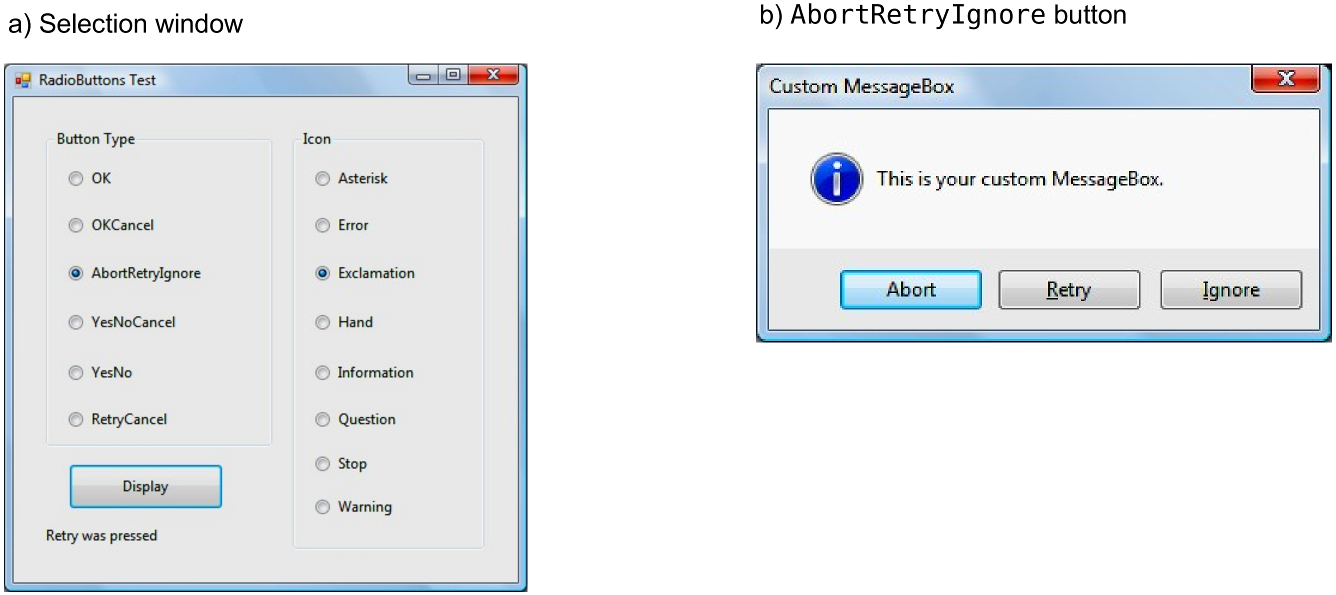
Fig. 14.28 | Using `RadioButtons` to set message-window options. (Part 6 of 8.)

```
            case DialogResult.Yes:
               displayLabel.Text = "Yes was pressed.";
               break;
            case DialogResult.No:
               displayLabel.Text = "No was pressed.";
               break;
         } // end switch
      } // end method displayButton Click
   } // end class RadioButtonsTestForm
} // end namespace RadioButtonsTest
```

RadioButtonsTestForm.cs

(7 of 8)

Testing for the dialog result and displaying appropriate text.



a) Selection window

RadioButtons Test

Button Type
- ○ OK
- ○ OKCancel
- ● AbortRetryIgnore
- ○ YesNoCancel
- ○ YesNo
- ○ RetryCancel

Icon
- ○ Asterisk
- ○ Error
- ● Exclamation
- ○ Hand
- ○ Information
- ○ Question
- ○ Stop
- ○ Warning

Retry was pressed

b) `AbortRetryIgnore` button

Custom MessageBox

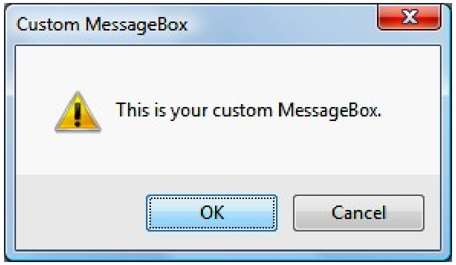
This is your custom MessageBox.

Fig. 14.28 | Using `RadioButtons` to set message-window options. (Part 7 of 8.)

RadioButtonsTestForm.cs

(8 of 8)

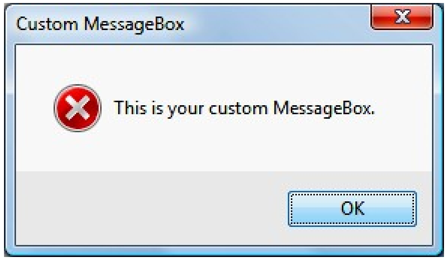
c) `OKCancel` button type



Custom MessageBox

This is your custom MessageBox.

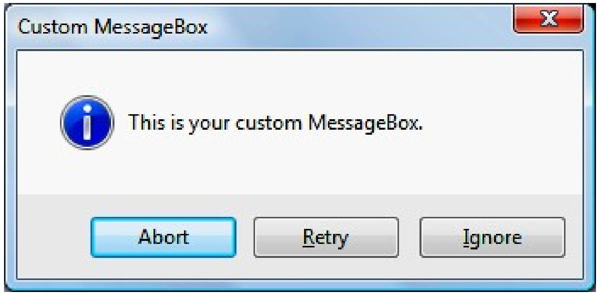
d) `OK` button type



Custom MessageBox

This is your custom MessageBox.

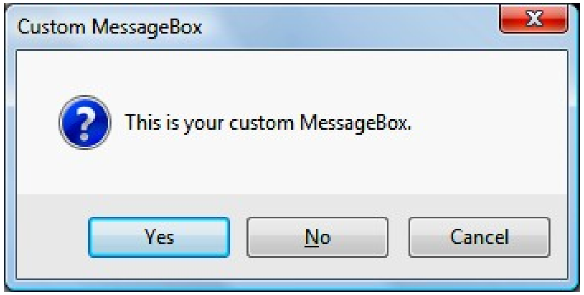
e) `AbortRetryIgnore` button type



Custom MessageBox

This is your custom MessageBox.

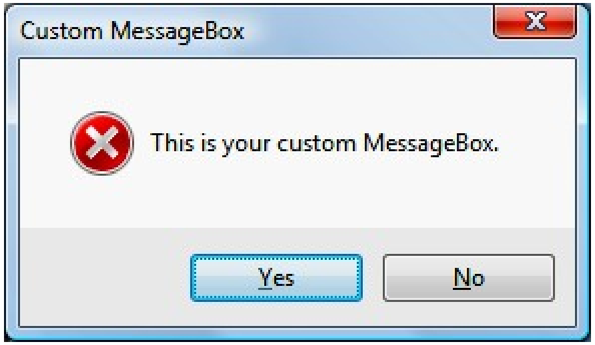
f) `YesNoCancel` button type



Custom MessageBox

This is your custom MessageBox.

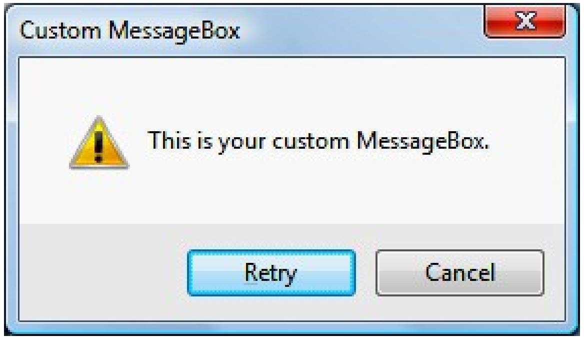
g) `YesNo` button type



Custom MessageBox

This is your custom MessageBox.

h) `RetryCancel` button type



Custom MessageBox

This is your custom MessageBox.

Fig. 14.28 | Using `RadioButtons` to set message-window options. (Part 8 of 8.)

14.8 PictureBoxes

- A PictureBox displays an image.

PictureBox properties and event	Description
Common Properties	
Image	Sets the image to display in the PictureBox.
SizeMode	Enumeration that controls image sizing and positioning.
Common Event	
Click	Occurs when the user clicks the control.

Fig. 14.29 | PictureBox properties and events.

Outline

- Figure 14.30 uses a PictureBox to display bitmap images.

PictureBoxTestForm.cs

(1 of 3)

```
// Fig. 14.30: PictureBoxTestForm.cs
// Using a PictureBox to display images.
using System;
using System.Drawing;
using System.Windows.Forms;

namespace PictureBoxTest
{
   // Form to display different images when PictureBox is clicked
   public partial class PictureBoxTestForm : Form
   {
      private int imageNum = -1; // determines which image is displayed

      // default constructor
      public PictureBoxTestForm()
      {
         InitializeComponent();
      } // end constructor
```

Fig. 14.30 | Using a PictureBox to display images. (Part 1 of 3.)

PictureBoxTestForm.cs

(2 of 3)

```

      // change image whenever Next Button is clicked
      private void nextButton_Click( object sender, EventArgs e )
      {
         imageNum = ( imageNum + 1 ) % 3; // imageNum cycles from 0 to 2

         // retrieve image from resources and load into PictureBox
         imagePictureBox.Image = ( Image )
            ( Properties.Resources.ResourceManager.GetObject(
            string.Format( “image{0}”, imageNum ) ) );
      } // end method nextButton_Click
   } // end class PictureBoxTestForm
} // end namespace PictureBoxTest
```

Displaying an `Image` from the project’s resources.

Fig. 14.30 | Using a `PictureBox` to display images. (Part 2 of 3.)

PictureBoxTestForm.cs

(3 of 3)

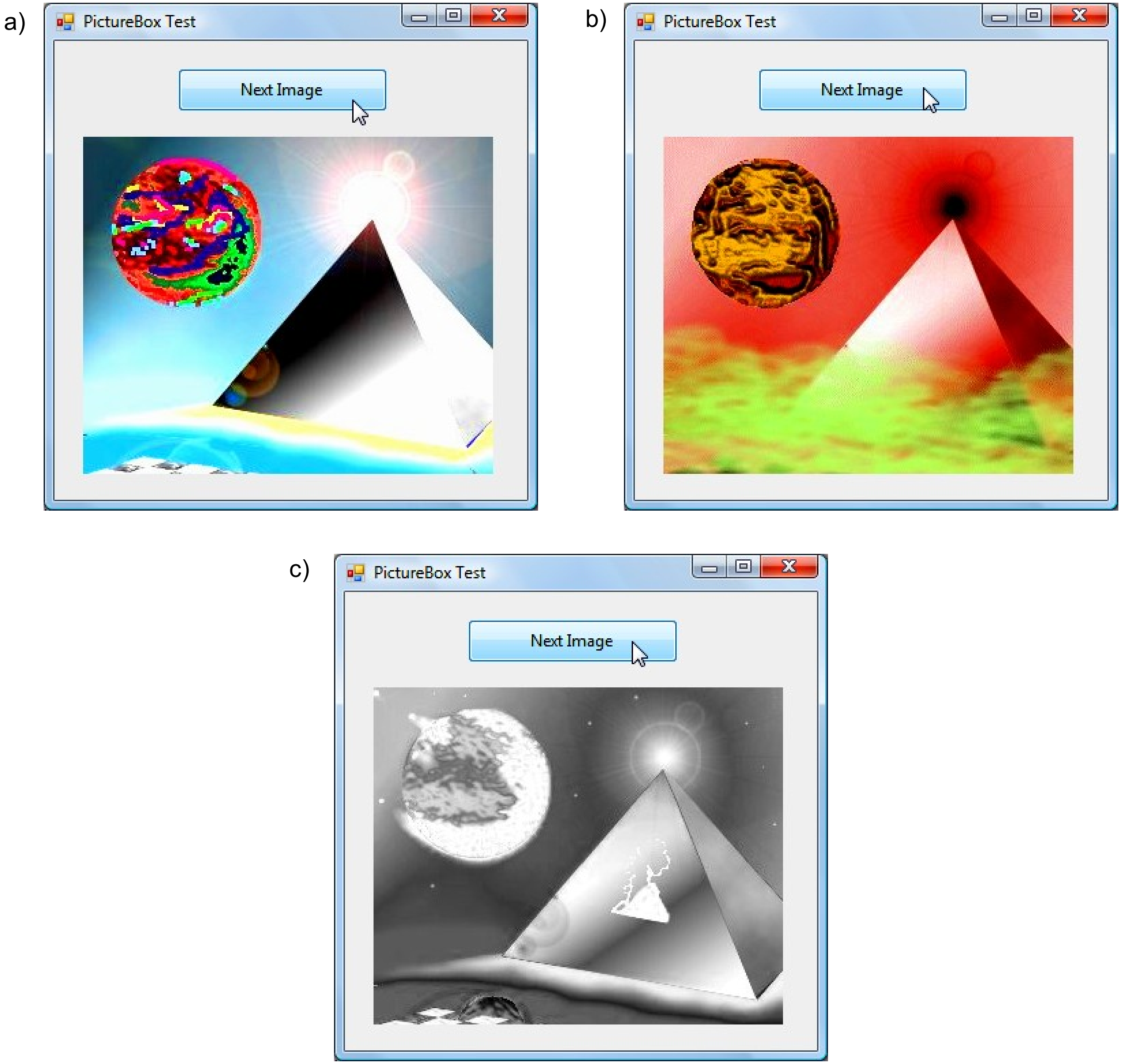


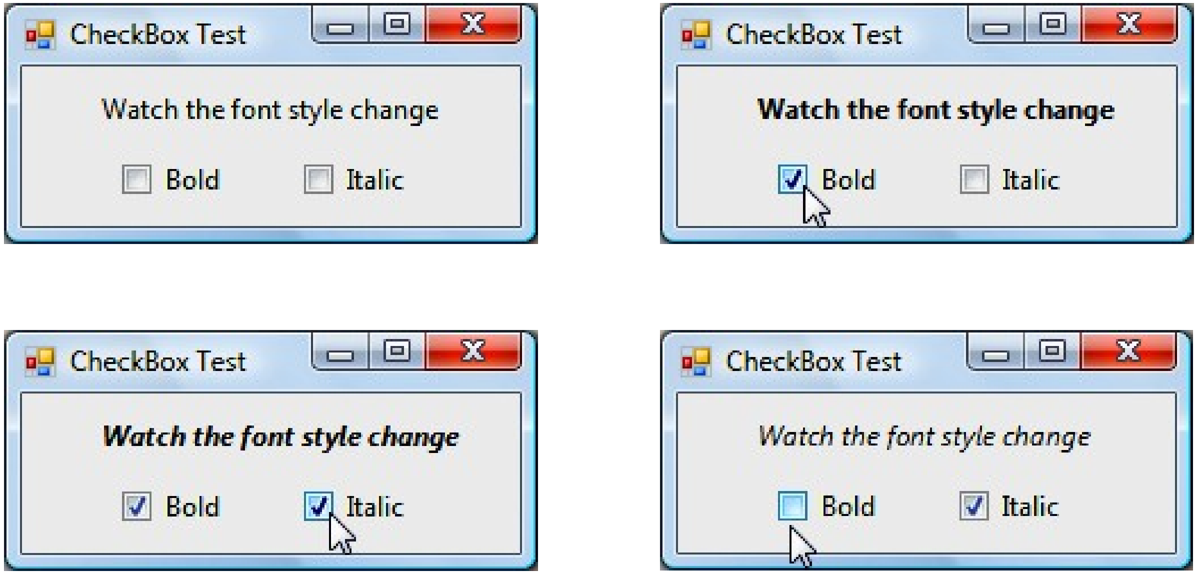
Fig. 14.30 | Using a `PictureBox` to display images. (Part 3 of 3.)

14.8 PictureBoxes (Cont.)

- Embedding the images in the application prevents problems of using several separate files.
- To add a resource:
  - Double click the project’s **Properties** node in the **Solution Explorer**.
  - Click the **Resources** tab.
  - At the top of the **Resources** tab click the down arrow next to the **Add Resource** button and select **Add Existing File…**
  - Locate the files you wish to add and click the **Open** button.
  - Save your project.

CheckBoxTestForm.cs

(3 of 3)



CheckBox Test

Watch the font style change

□ Bold □ Italic

CheckBox Test

Watch the font style change

☑ Bold □ Italic

CheckBox Test

Watch the font style change

☑ Bold ☑ Italic

CheckBox Test

Watch the font style change

□ Bold ☑ Italic

Fig. 14.26 | Using CheckBoxes to change font styles. (Part 3 of 3.)

14.9 ToolTips

- Recall that tool tips are the helpful text that appears when the mouse hovers over an item in a GUI.

`ToolTip` properties and events	Description
Common Properties	
`AutoPopDelay`	The amount of time (in milliseconds) that the tool tip appears.
`InitialDelay`	The amount of time that a mouse must hover before a tool tip appears.
`ReshowDelay`	The amount of time between which two different tool tips can appear.
Common Event	
`Draw`	Raised when the tool tip is displayed.

Fig. 14.31 | `ToolTip` properties and events.

14.9 ToolTips (Cont.)

ToolTipDemonstrationForm.cs

(1of 3)

- A ToolTip component appears in the **component tray**—the gray region below the Form in **Design** mode.
- A **ToolTip on** property for each ToolTip component appears in the **Properties** window for the Form's other controls.

- Figure 14.32 demonstrates the `ToolTip` component.

ToolTipDemonstrationForm.cs

(2of 3)

```
// Fig. 14.32: ToolTipDemonstrationForm.cs
// Demonstrating the ToolTip component.
using System;
using System.Windows.Forms;

namespace ToolTipDemonstration
{
   public partial class ToolTipDemonstrationForm : Form
   {
      // default constructor
      public ToolTipDemonstrationForm()
      {
         InitializeComponent();
      } // end constructor

      // no event handlers needed for this example

   } // end class ToolTipDemonstrationForm
} // end namespace ToolTipDemonstration
```

Fig. 14.32 | Demonstrating the `ToolTip` component. (Part 1 of 2.)

Outline

`ToolTipDemonstrationForm.cs`

(3of 3)

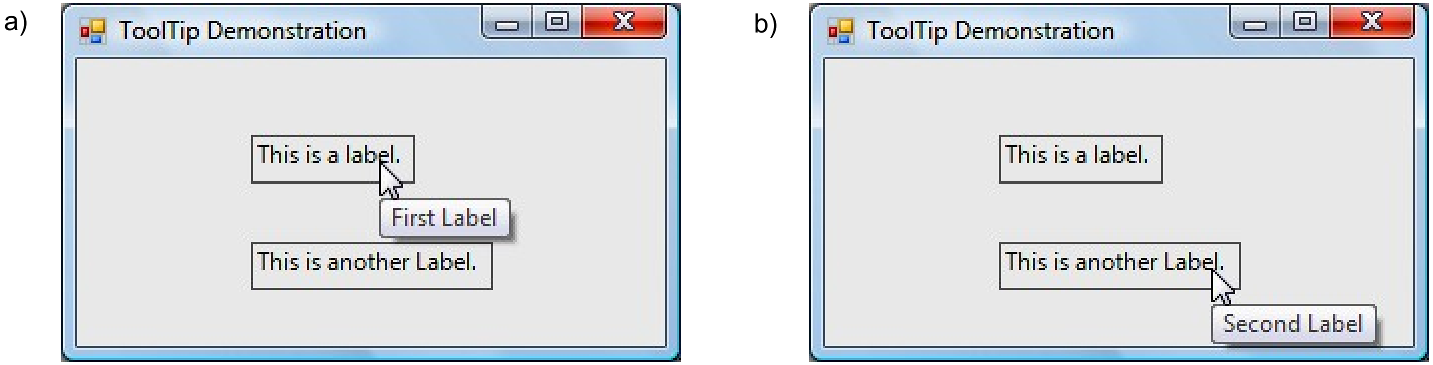
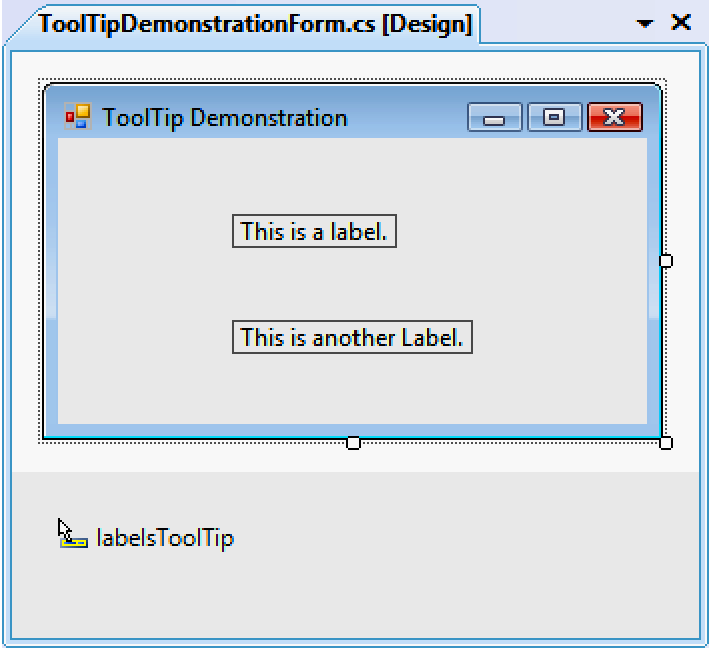


Fig. 14.32 | Demonstrating the `ToolTip` component. (Part 1 of 3.)

14.9 ToolTips (Cont.)

- Set the tool-tip text for the `Labels` to “`First Label`” and “`Second Label`”

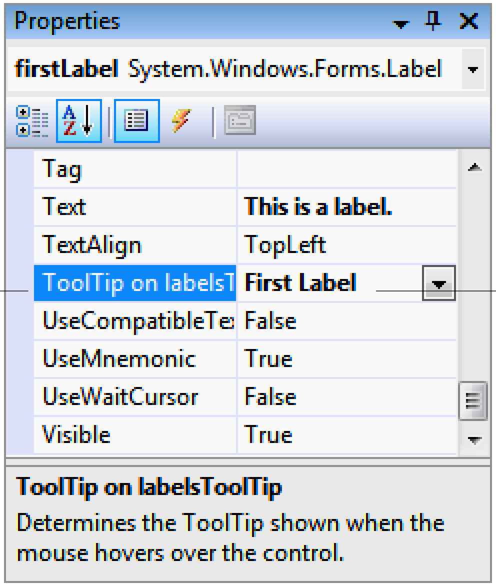


ToolTip in component tray

Fig. 14.33 | Demonstrating the component tray.

14.9 ToolTips (Cont.)

- Figure 14.34 demonstrates setting the tool-tip text for the first `Label`.



firstLabel System.Windows.Forms.Label

Tag	
Text	**This is a label.**
TextAlign	TopLeft
ToolTip on labelsT	**First Label**
UseCompatibleTe:	False
UseMnemonic	True
UseWaitCursor	False
Visible	True

ToolTip on labelsToolTip
Determines the ToolTip shown when the mouse hovers over the control.

Property to set tool-tip text

Tool-tip text

Fig. 14.34 | Setting a control’s tool-tip text.

14.10 NumericUpDown Control

- Restricting a user's input choices to a specific range of numeric values can be done with a **`NumericUpDown control`**.
- A user can type numeric values into this control or click up and down arrows.

14.10 NumericUpDown Control (Cont.)

NumericUpDown properties and events	Description
Common Properties	
DecimalPlaces	Specifies how many decimal places to display in the control.
Increment	Specifies by how much the number in the control changes when the user clicks the up and down arrows.
Maximum	Largest value in the control's range.
Minimum	Smallest value in the control's range.
UpDownAlign	Modifies the alignment of the up and down Buttons on the NumericUpDown control.
Value	The numeric value currently displayed in the control.
Common Event	
ValueChanged	This event is raised when the value in the control is changed.

Fig. 14.35 | NumericUpDown properties and events.

14.10 NumericUpDown Control (Cont.)

- Figure 14.36 demonstrates a GUI application that calculates interest rate.
- For the NumericUpDown control, we set the `Minimum` to `1` and the `Maximum` to `10`.
- We set the `NumericUpDown`'s **`ReadOnly` property** to true to indicate that the user cannot type a number into the control.

Outline

interestCalculatorForm.cs

(1of 3)

```
// Fig. 14.36: interestCalculatorForm.cs
// Demonstrating the NumericUpDown control.
using System;
using System.Windows.Forms;

namespace NumericUpDownTest
{
   public partial class interestCalculatorForm : Form
   {
      // default constructor
      public interestCalculatorForm()
      {
         InitializeComponent();
      } // end constructor

      private void calculateButton_Click(
         object sender, EventArgs e )
      {
         // declare variables to store user input
         decimal principal; // store principal
         double rate; // store interest rate
         int year; // store number of years
         decimal amount; // store amount
         string output; // store output
```

Fig. 14.36 | Demonstrating the NumericUpDown control. (Part 1 of 3.)

interestCalculatorForm.cs

(2 of 3)

```

         // retrieve user input
         principal = Convert.ToDecimal( principalTextBox.Text );
         rate = Convert.ToDouble( interestTextBox.Text );
         year = Convert.ToInt32( yearUpDown.Value );

         // set output header
         output = "Year\tAmount on Deposit\r\n";

         // calculate amount after each year and append to output
         for ( int yearCounter = 1; yearCounter <= year;  yearCounter++ )
         {
            amount =  principal * ( ( decimal )
               Math.Pow( ( 1 + rate / 100 ), yearCounter ) );
            output += ( yearCounter + "\t" +
               string.Format( "{0:C}", amount ) + "\r\n" );
         } // end for

         displayTextBox.Text = output; // display result
      } // end method calculateButton Click
   } // end class interestCalculatorForm
} // end namespace NumericUpDownTest
```

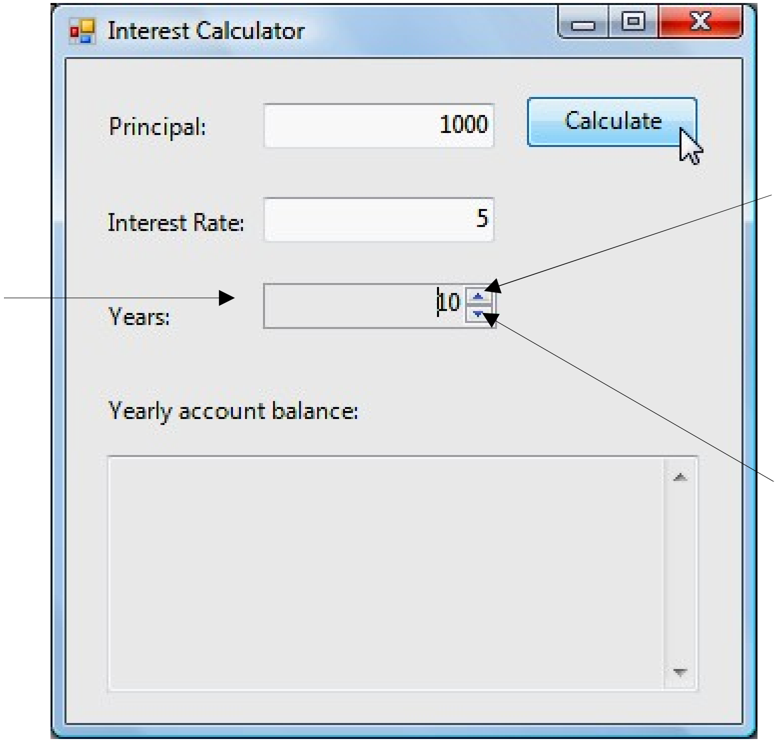
Retrieving the value of the NumericUpDown control.

Performing the interest calculation.

Fig. 14.36 | Demonstrating the NumericUpDown control. (Part 2 of 3.)

`interestCalculatorForm.cs`

(3 of 3)



Principal: 1000

Interest Rate: 5

Years: 10

Yearly account balance:

`NumericUpDown` control

Click to increase number of years

Click to decrease number of years

Fig. 14.36 | Demonstrating the `NumericUpDown` control. (Part 3 of 3.)

14.11 Mouse-Event Handling

- **Mouse events** are generated when the user interacts with a control via the mouse.
- Information about the event is passed through a **`MouseEvent-Args`** object, and the delegate type is **`Mouse-EventHandler`**.
- `MouseEventArgs` *x*- and *y*-coordinates are relative to the control that generated the event.

14.11 Mouse-Event Handling

Mouse events and event arguments	
Mouse Events with Event Argument of Type `EventArgs`	
`MouseEnter`	Occurs when the mouse cursor enters the control's boundaries.
`MouseLeave`	Occurs when the mouse cursor leaves the control's boundaries.
Mouse Events with Event Argument of Type `MouseEventArgs`	
`MouseDown`	Occurs when a mouse button is pressed.

Fig. 14.37 | Mouse events and event arguments. (Part 1 of 2.)

Outline

PictureBoxTestForm.cs

(2 of 3)

```

      // change image whenever Next Button is clicked
      private void nextButton_Click( object sender, EventArgs e )
      {
         imageNum = ( imageNum + 1 ) % 3; // imageNum cycles from 0 to 2

         // retrieve image from resources and load into PictureBox
         imagePictureBox.Image = ( Image )
            ( Properties.Resources.ResourceManager.GetObject(
            string.Format( “image{0}”, imageNum ) ) );
      } // end method nextButton_Click
   } // end class PictureBoxTestForm
} // end namespace PictureBoxTest
```

Displaying an `Image` from the project’s resources.

Fig. 14.30 | Using a `PictureBox` to display images. (Part 2 of 3.)

PainterForm.cs

(1of 2)

```
// Fig. 14.38: PainterForm.cs
// Using the mouse to draw on a Form.
using System;
using System.Drawing;
using System.Windows.Forms;

namespace Painter
{
   // creates a Form that is a drawing surface
   public partial class PainterForm : Form
   {
      bool shouldPaint = false; // determines whether to paint

      // default constructor
      public PainterForm()
      {
         InitializeComponent();
      } // end constructor

      // should paint when mouse button is pressed down
      private void PainterForm_MouseDown(
         object sender, MouseEventArgs e )
```

`shouldPaint` determines whether to draw on the `Form` (`true` while the mouse button is pressed).

Pressing a mouse button sets `shouldPaint` to

Outline

PainterForm.cs

(2of 2)

```
   {
      // indicate that user is dragging the mouse
      shouldPaint = true;
   } // end method PainterForm_MouseDown

   // stop painting when mouse button is released
   private void PainterForm_MouseUp( object sender, MouseEventArgs e )
   {
      // indicate that user released the mouse button
      shouldPaint = false;
   } // end method PainterForm_MouseUp

   // draw circle whenever mouse moves with its button held down
   private void PainterForm_MouseMove(
      object sender, MouseEventArgs e )
   {
      if ( shouldPaint ) // check if mouse button is being pressed
      {
         // draw a circle where the mouse pointer is present
         Graphics graphics = CreateGraphics();
         graphics.FillEllipse(
            new SolidBrush( Color.BlueViolet ), e.X, e.Y, 4, 4 );
         graphics.Dispose();
      } // end if
   } // end method PainterForm_MouseMove
} // end class PainterForm
```

Pressing a mouse button sets `shouldPaint` to

Releasing the mouse button sets `shouldPaint`

The `MouseMove` event continually draws `Ellipse`s using the `Graphics` object.

```
    } // end namespace Painter
```

interestCalculatorForm.cs

(4of 4)

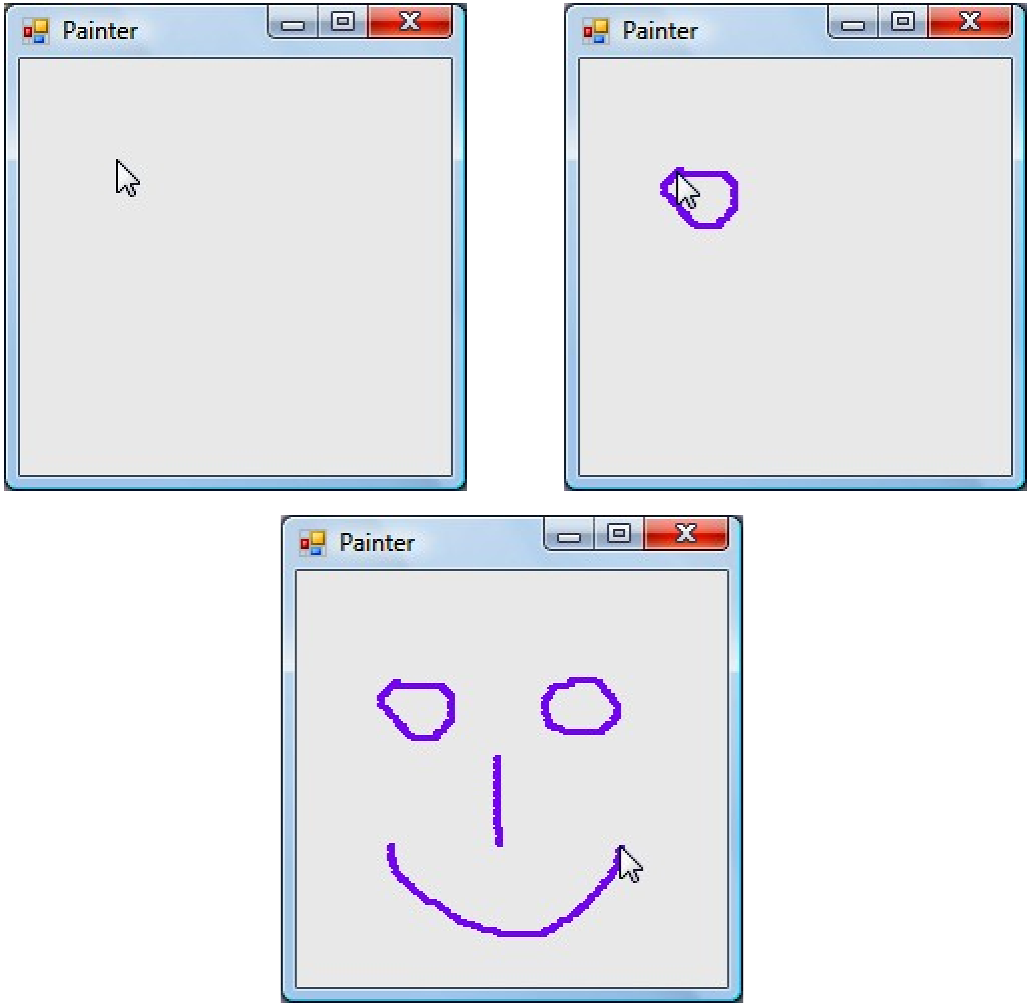
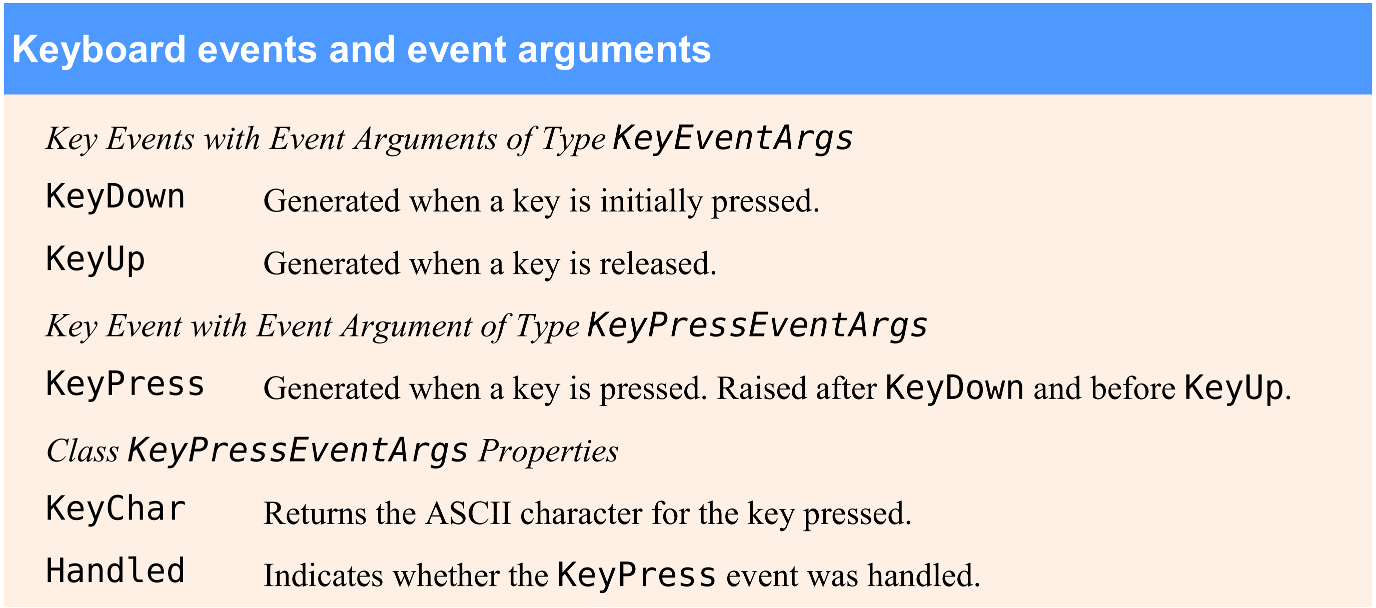


Fig. 14.38 | Using the mouse to draw on a `form.` (Part 4 of 4.)

14.12 Keyboard-Event Handling

- There are three key events:
  - The **`KeyPress`** event occurs when the user presses a key that represents an ASCII character.
  - The `KeyPress` event does not indicate whether **modifier keys** (e.g., *Shift*, *Alt* and *Ctrl*) were pressed; if this information is important, the **`KeyUp`** or **`KeyDown`** events can be used.

14.12 Keyboard-Event Handling (Cont.)



Keyboard events and event arguments	
Key Events with Event Arguments of Type `KeyEventArgs`	
`KeyDown`	Generated when a key is initially pressed.
`KeyUp`	Generated when a key is released.
Key Event with Event Argument of Type `KeyPressEventArgs`	
`KeyPress`	Generated when a key is pressed. Raised after `KeyDown` and before `KeyUp`.
Class `KeyPressEventArgs` *Properties*	
`KeyChar`	Returns the ASCII character for the key pressed.
`Handled`	Indicates whether the `KeyPress` event was handled.

Fig. 14.39 | Keyboard events and event arguments. (Part 1 of 2.)

- Figure 14.32 demonstrates the `ToolTip` component.

ToolTipDemonstrationForm.cs

(2of 3)

```
// Fig. ١٤.٣٢: ToolTipDemonstrationForm.cs
// Demonstrating the ToolTip component.
using System;
using System.Windows.Forms;

namespace ToolTipDemonstration
{
   public partial class ToolTipDemonstrationForm : Form
   {
      // default constructor
      public ToolTipDemonstrationForm()
      {
         InitializeComponent();
      } // end constructor

      // no event handlers needed for this example

   } // end class ToolTipDemonstrationForm
} // end namespace ToolTipDemonstration
```

Fig. 14.32 | Demonstrating the `ToolTip` component. (Part 1 of 2.)

KeyDemoForm.cs

(1of 3)

```
// Fig. 14.40: KeyDemoForm.cs
// Displaying information about the key the user pressed.
using System;
using System.Windows.Forms;

namespace KeyDemo
{
   // Form to display key information when key is pressed
   public partial class KeyDemoForm : Form
   {
      // default constructor
      public KeyDemoForm()
      {
         InitializeComponent();
      } // end constructor

      // display the character pressed using KeyChar
      private void KeyDemoForm_KeyPress(
         object sender, KeyPressEventArgs e )
      {
         charLabel.Text = "Key pressed: " + e.KeyChar;
      } // end method KeyDemoForm_KeyPress
```

`KeyPress` displays the Pressed key as a `char` in `charLabel`.

Fig. 14.40 | Demonstrating keyboard events. (Part 1 of 3.)

KeyDemoForm.cs

(1of 3)

```
      // display modifier keys, key code, key data and key value
      private void KeyDemoForm_KeyDown( object sender, KeyEventArgs e )
      {
         keyInfoLabel.Text =
            "Alt: " + ( e.Alt ? "Yes" : "No" ) + '\n' +
            "Shift: " + ( e.Shift ? "Yes" : "No" ) + '\n' +
            "Ctrl: " + ( e.Control ? "Yes" : "No" ) + '\n' +
            "KeyCode: " + e.KeyCode + '\n' +
            "KeyData: " + e.KeyData + '\n' +
            "KeyValue: " + e.KeyValue;
      } // end method KeyDemoForm_KeyDown

      // clear Labels when key released
      private void KeyDemoForm_KeyUp( object sender, KeyEventArgs e )
      {
         charLabel.Text = "";
         keyInfoLabel.Text = "";
      } // end method KeyDemoForm_KeyUp
   } // end class KeyDemoForm
} // end namespace KeyDemo
```

KeyDown tests for the *Alt*, *Shift* and *Crtl* keys.

The KeyData property includes data about ASCII and modifier keys

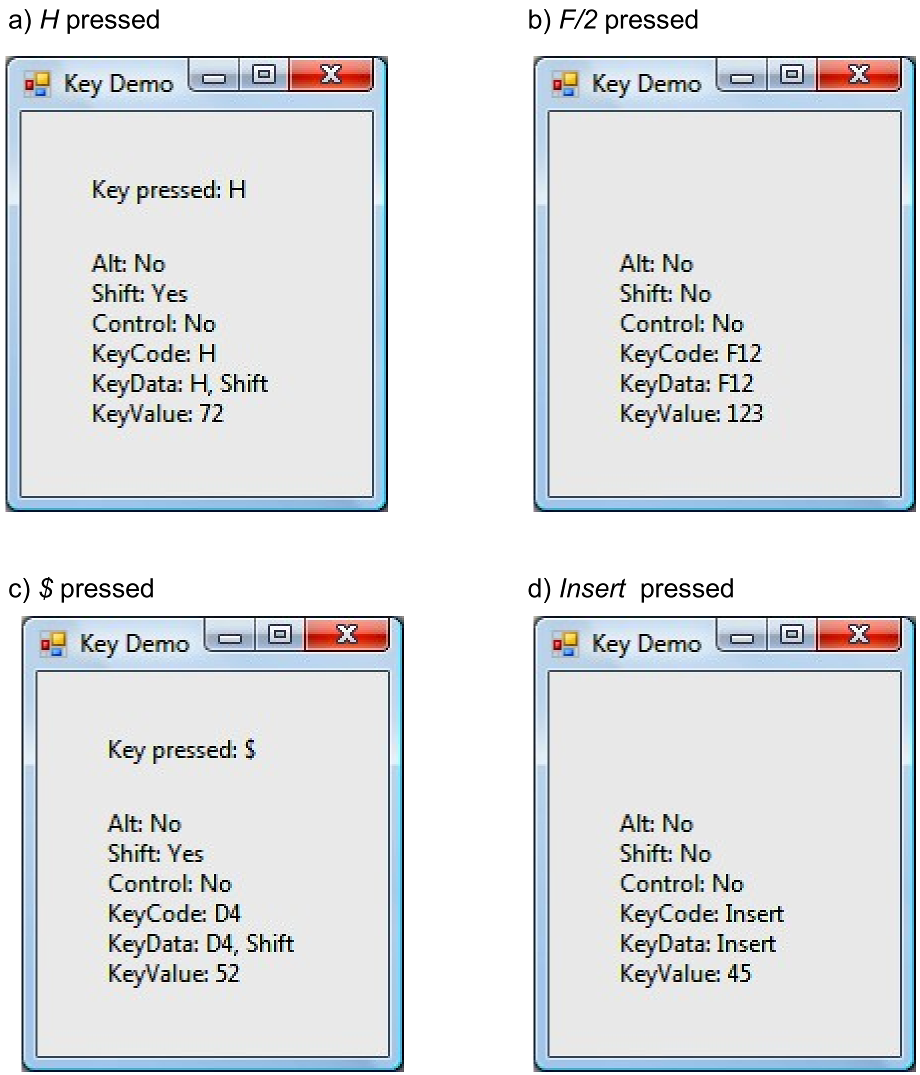
KeyUp clears both Labels when the key is released.

Fig. 14.40 | Demonstrating keyboard events. (Part 2 of 3.)

Outline

`KeyDemoForm.cs`

(3of 3)



a) *H* pressed

```
Key pressed: H

Alt: No
Shift: Yes
Control: No
KeyCode: H
KeyData: H, Shift
KeyValue: 72
```

b) *F/2* pressed

```
Alt: No
Shift: No
Control: No
KeyCode: F12
KeyData: F12
KeyValue: 123
```

c) *$* pressed

```
Key pressed: $

Alt: No
Shift: Yes
Control: No
KeyCode: D4
KeyData: D4, Shift
KeyValue: 52
```

d) *Insert* pressed

```
Alt: No
Shift: No
Control: No
KeyCode: Insert
KeyData: Insert
KeyValue: 45
```

Fig. 14.40 | Demonstrating keyboard events. (Part 3 of 3.)