Graphical User Interfaces with Windows Forms: Part 1

Chapter 14 of Visual C# How to Program, 6/e

OBJECTIVES

In this chapter you'll learn:

- Design principles of graphical user interfaces (GUIs).
- How to create graphical user interfaces.
- How to process events in response to user interactions with GUI controls.
- The namespaces that contain the classes for GUI controls and event handling.
- How to create and manipulate various controls.
- How to add descriptive `ToolTip`s to GUI controls.
- How to process mouse and keyboard events.

14.1 Introduction

- A graphical user interface (GUI) allows a user to interact visually with a program.
- Figure 14.1 shows a Visual Studio window containing various GUI controls.

Look-and-Feel Observation 14.1

Consistent user interfaces enable a user to learn new apps more quickly because the apps have the same "look" and "feel."

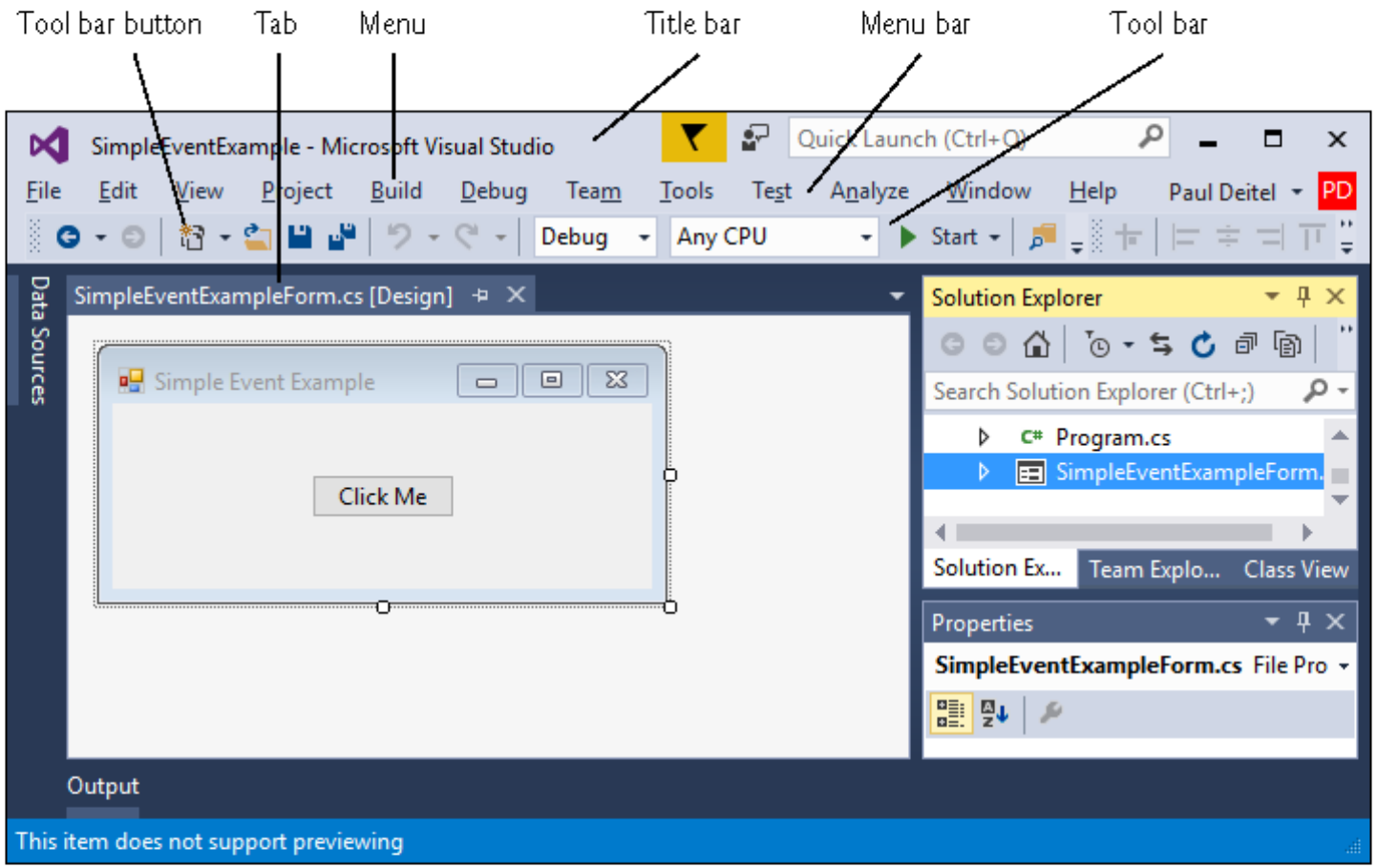


Fig. 14.1 | GUI controls in Visual Studio.

14.1 Introduction (Cont.)

- GUI controls are objects that can display information on the screen or enable users to interact with an app.
- Several common GUI controls are listed in Fig. 14.2.

Control	Description
`Label`	Displays *images* or *uneditable text*.
`TextBox`	Enables the user to *enter data via the keyboard*. It also can be used to *display editable or uneditable text*.
`Button`	Triggers an *event* when clicked with the mouse.
`CheckBox`	Specifies an option that can be *selected* (checked) or *unselected* (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 one or more items.
`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

- A `Form` is a graphical element that appears on your computer's desktop; it can be a dialog, a window or an MDI window.
- A component is an instance of a class that implements the `IComponent` interface, which defines the behaviors that components must implement, such as how the component is loaded.
- A control, such as a `Button` or `Label`, has a graphical representation at runtime.

14.2 Windows Forms (Cont.)

- Figure 14.3 displays the Windows Forms controls and components from the C# Toolbox.
- To add a control or component, select it from the **Toolbox** and drag it onto the `Form`.

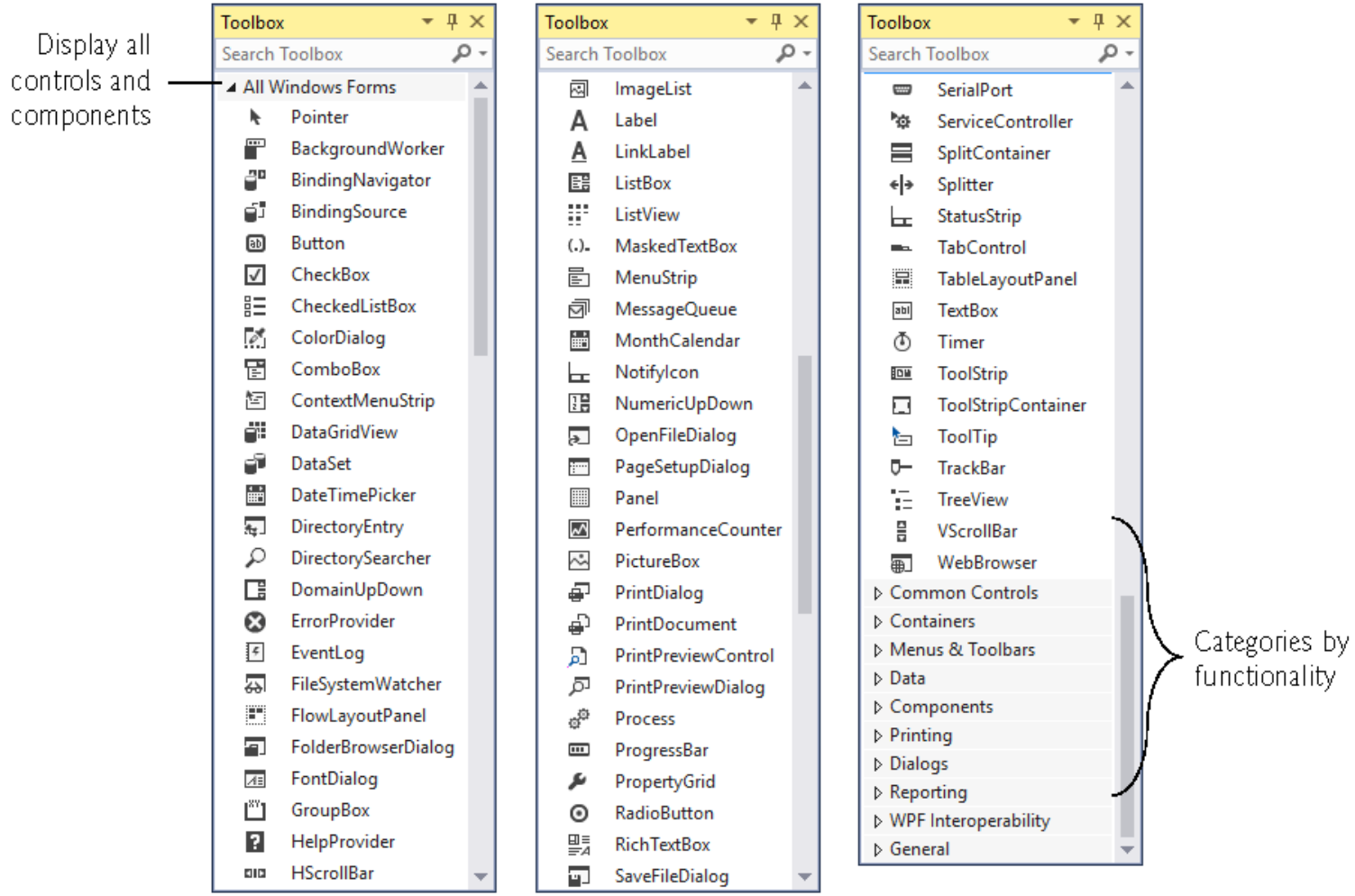


Fig. 14.3 | Components and controls for Windows Forms.

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 items from the **Toolbox** onto the `Form`, Visual Studio generates code that creates the object and sets its basic properties.
- The IDE maintains the generated code in a separate file using partial classes.

14.2 Windows Forms (Cont.)

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

`Form` properties, methods and an event	Description
Common Properties	
`AcceptButton`	Default `Button` that's clicked when you press *Enter*.
`AutoScroll`	`bool` value (`false` by default) that allows or disallows *scrollbars* when needed.
`CancelButton`	`Button` that's clicked when the *Escape* key is pressed.
`FormBorderStyle`	*Border style* for the `Form` (`Sizable` by default).
`Font`	Font of text displayed on the `Form`, and the *default font* for controls added to the `Form`.
`Text`	Text in the `Form`'s title bar.

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

`Form` properties, methods and an event	Description
Common Methods	
`Close`	Closes a `Form` and *releases all resources*, such as the memory used for the `Form`'s contents. A closed `Form` cannot be reopened.
`Hide`	Hides a `Form`, but does *not* destroy the `Form` or release its resources.
`Show`	Displays a *hidden* `Form`.
Common Event	
`Load`	Occurs before a `Form` is displayed to the user. You'll learn about events and event-handling in the next section.

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.

14.3 Event Handling

- 14.3.1 A Simple Event-Driven GUI
- The `Form` in the app of Fig. 14.5 contains a `Button` that a user can click to display a `MessageBox`.

```
1  // Fig. 14.5: SimpleEventExampleForm.cs
2  // Simple event handling example.
3  using System;
4  using System.Windows.Forms;
5
6  namespace SimpleEventExample
7  {
8     // Form that shows a simple event handler
9     public partial class SimpleEventExampleForm : Form
10    {
11       // default constructor
12       public SimpleEventExampleForm()
13       {
14          InitializeComponent();
15       }
16
17       // handles click event of Button clickButton
18       private void clickButton_Click(object sender, EventArgs e)
19       {
20          MessageBox.Show("Button was clicked.");
21       }
22    }
23 }
```

Fig. 14.5 | Simple event-handling example. (Part 1 of 2.)

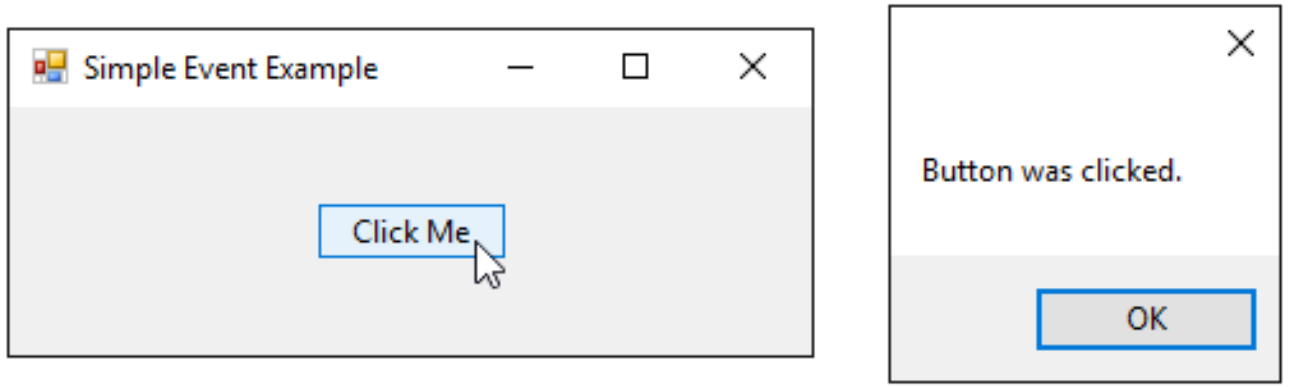


Fig. 14.5 | Simple event-handling example. (Part 2 of 2.)

14.3 Event Handling

- ***Adding an Event Handler for the Button's Click Event***
- To create the app'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)
    {
    }
    ```
- By convention, the IDE names the event-handler method as `objectName_eventName` (e.g., `clickButton_Click`).
    ```

14.3 Event Handling (Cont.)

- ***Event Handler Parameters***
- Each event handler receives two parameters when it's called:
  - This first—an object reference typically named sender—is a reference to the object that generated the event.
  - The second is a reference to an `EventArgs` object (or an object of an `EventArgs` derived class) which contains additional information about the event.

14.3 Event Handling (Cont.)

- 14.3.2 Auto-Generated GUI Code
- Visual Studio places the auto-generated code in the `Designer.cs` file of the `Form`.
- Open this file by expanding the node in the **Solution Explorer** window for the file you're currently working in (`SimpleEventExampleForm.cs`) and double clicking the file name that ends with `Designer.cs`.
- 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 | SimpleEventExample.SimpleEventExampleForm | components

```
1   namespace SimpleEventExample
2   {
3      partial class SimpleEventExampleForm
4      {
5         /// <summary>
6         /// Required designer variable.
7         /// </summary>
8         private System.ComponentModel.IContainer components = null;
9
10        /// <summary>
11        /// Clean up any resources being used.
12        /// </summary>
13        /// <param name="disposing">true if managed resources should be disposed; othe
14        protected override void Dispose(bool disposing)
15        {
16           if (disposing && (components != null))
17           {
18              components.Dispose();
19           }
20           base.Dispose(disposing);
21        }
22
```

123 %

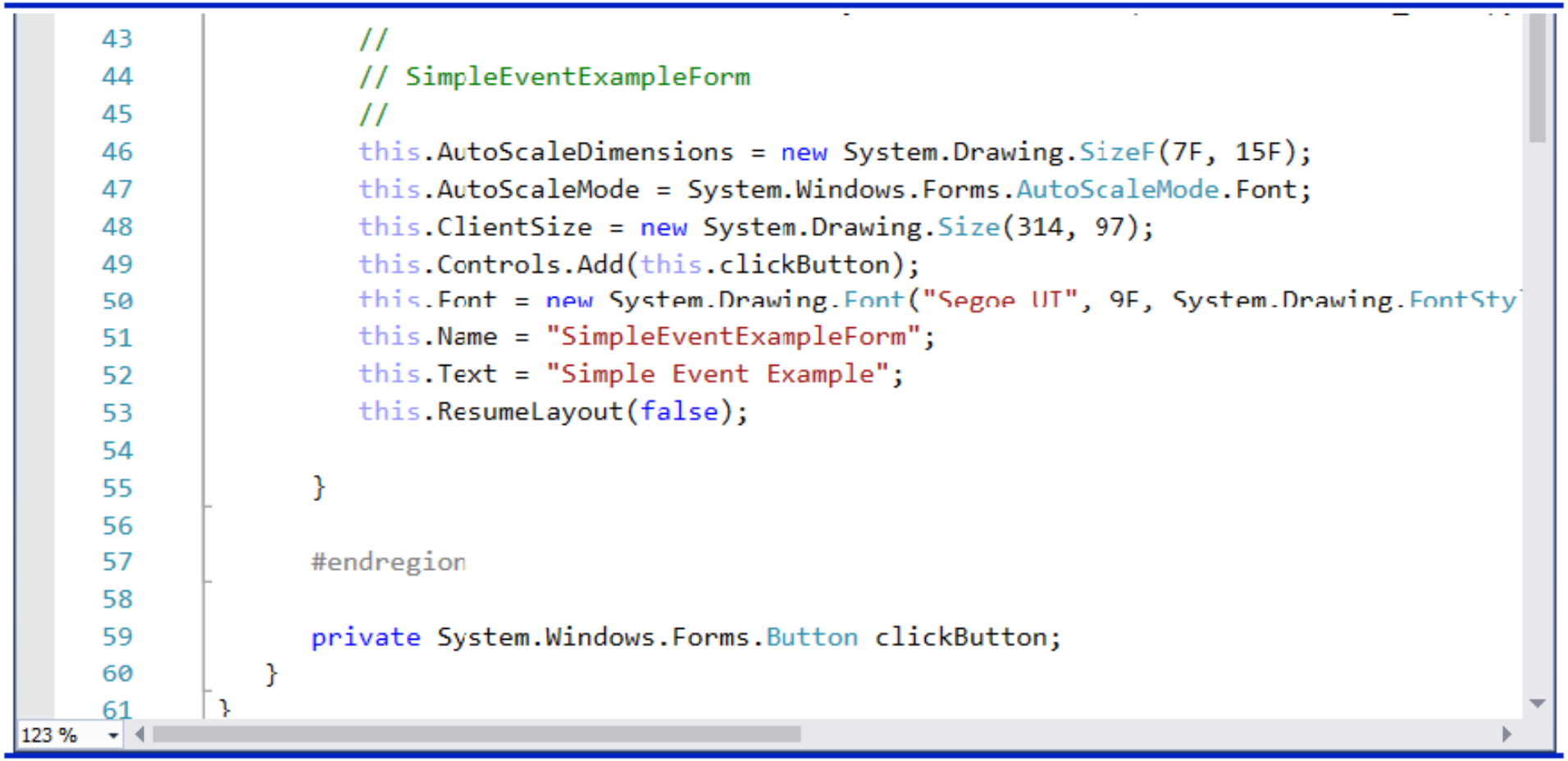
Fig. 14.6 | First half of the Visual Studio generated code file.

SimpleEventExampleForm.Designer.cs

SimpleEventExample | SimpleEventExample.SimpleEventExampleForm | components

```
23      #region Windows Form Designer generated code
24
25      /// <summary>
26      /// Required method for Designer support - do not modify
27      /// the contents of this method with the code editor.
28      /// </summary>
29      private void InitializeComponent()
30      {
31          this.clickButton = new System.Windows.Forms.Button();
32          this.SuspendLayout();
33          //
34          // clickButton
35          //
36          this.clickButton.Location = new System.Drawing.Point(120, 37);
37          this.clickButton.Name = "clickButton";
38          this.clickButton.Size = new System.Drawing.Size(75, 23);
39          this.clickButton.TabIndex = 0;
40          this.clickButton.Text = "Click Me";
41          this.clickButton.UseVisualStyleBackColor = true;
42          this.clickButton.Click += new System.EventHandler(this.clickButton_Click);
```

Fig. 14.7 | Second half of the Visual Studio generated code file. (Part 1 of 2.)



```
43          //
44          // SimpleEventExampleForm
45          //
46          this.AutoScaleDimensions = new System.Drawing.SizeF(7F, 15F);
47          this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
48          this.ClientSize = new System.Drawing.Size(314, 97);
49          this.Controls.Add(this.clickButton);
50          this.Font = new System.Drawing.Font("Segoe UI", 9F, System.Drawing.FontSty
51          this.Name = "SimpleEventExampleForm";
52          this.Text = "Simple Event Example";
53          this.ResumeLayout(false);
54
55      }
56
57      #endregion
58
59      private System.Windows.Forms.Button clickButton;
60  }
61  }
```

123 %

Fig. 14.7 | Second half of the Visual Studio generated code file. (Part 2 of 2.)

14.3 Event Handling (Cont.)

- The partial modifier allows the `Form`'s class to be split among multiple files.
- Fig. 14.7 declares the `clickButton` that we created in `Design` mode. It's declared as an instance variable of class `SimpleEventExampleForm`.
- The property values correspond to the values set in the **Properties** window for each control.
- Method `InitializeComponent` is called when the `Form` is created.

Error-Prevention Tip 14.1

The code in the `Designer.cs` *file that's generated by building a GUI in* Design *mode is not meant to be modified directly, which is why this code is placed in a separate file. Modifying this code can prevent the GUI from being displayed correctly in* Design *mode and might cause an app to function incorrectly. In* Design *mode, it's recommended that you modify control properties only in the* Properties *window, not in the* `Designer.cs` *file.*

14.3 Event Handling (Cont.)

- 14.3.3 Delegates and the Event-Handling Mechanism
- Event handlers are connected to a control's events via special objects called delegates.
- A `delegate` type declaration specifies the signature of a method—in event handling, the signature specifies the return type and arguments for an event handler.
- A `delegate` of type `EventHandler` can hold references to methods that return void and receive two parameters—one of type object and one of type `EventArgs`:
  - `public delegate void EventHandler(object sender, EventArgs e);`

14.3 Event Handling (Cont.)

- ***Indicating the Method that a Delegate Should Call***
- A `Button` calls its `EventHandler` `delegate` in response to a click.
- The `delegate`'s job is to invoke the appropriate method.
- This code is added by Visual Studio when you double click the `Button` control in **Design** mode:
  - `new System.EventHandler(this.clickButton_Click);`
- Adding more `delegates` to an event is called multicast delegates.

14.3 Event Handling (Cont.)

- 14.3.4 Another Way to Create Event Handlers
- ***Using the Properties Window to Create Event Handlers***
- Controls can generate many different events.
- Clicking the **Events** icon (the lightning-bolt icon) in the **Properties** window (Fig. 14.8), displays all the events for the selected control.

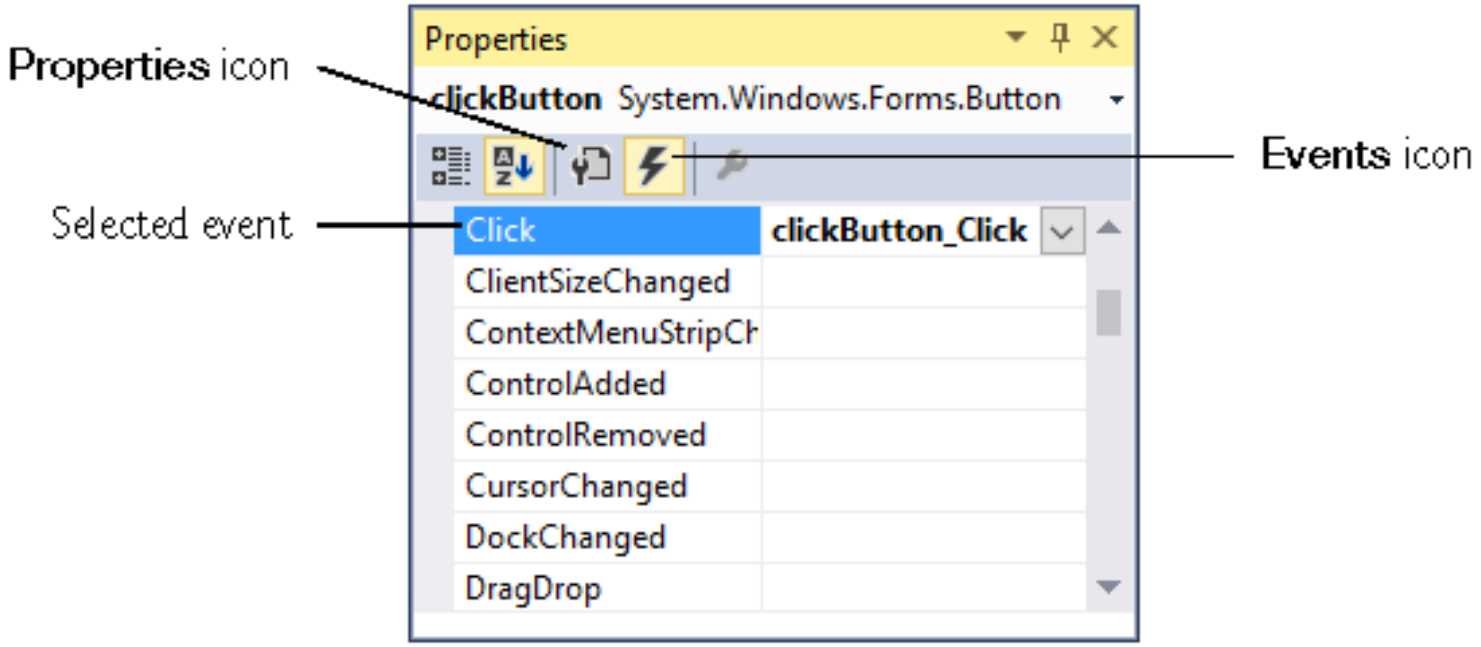


Fig. 14.8 | Viewing events for a **Button** control in the **Properties** window.

14.3 Event Handling (Cont.)

- 14.3.5 Locating Event Information
- Read the Visual Studio documentation to learn about the different events raised by each control.
- To do this, select a control in the IDE and press the *F1* key to display that control's online help (Fig. 14.9).
- The web page that's displayed contains basic information about the control's class.

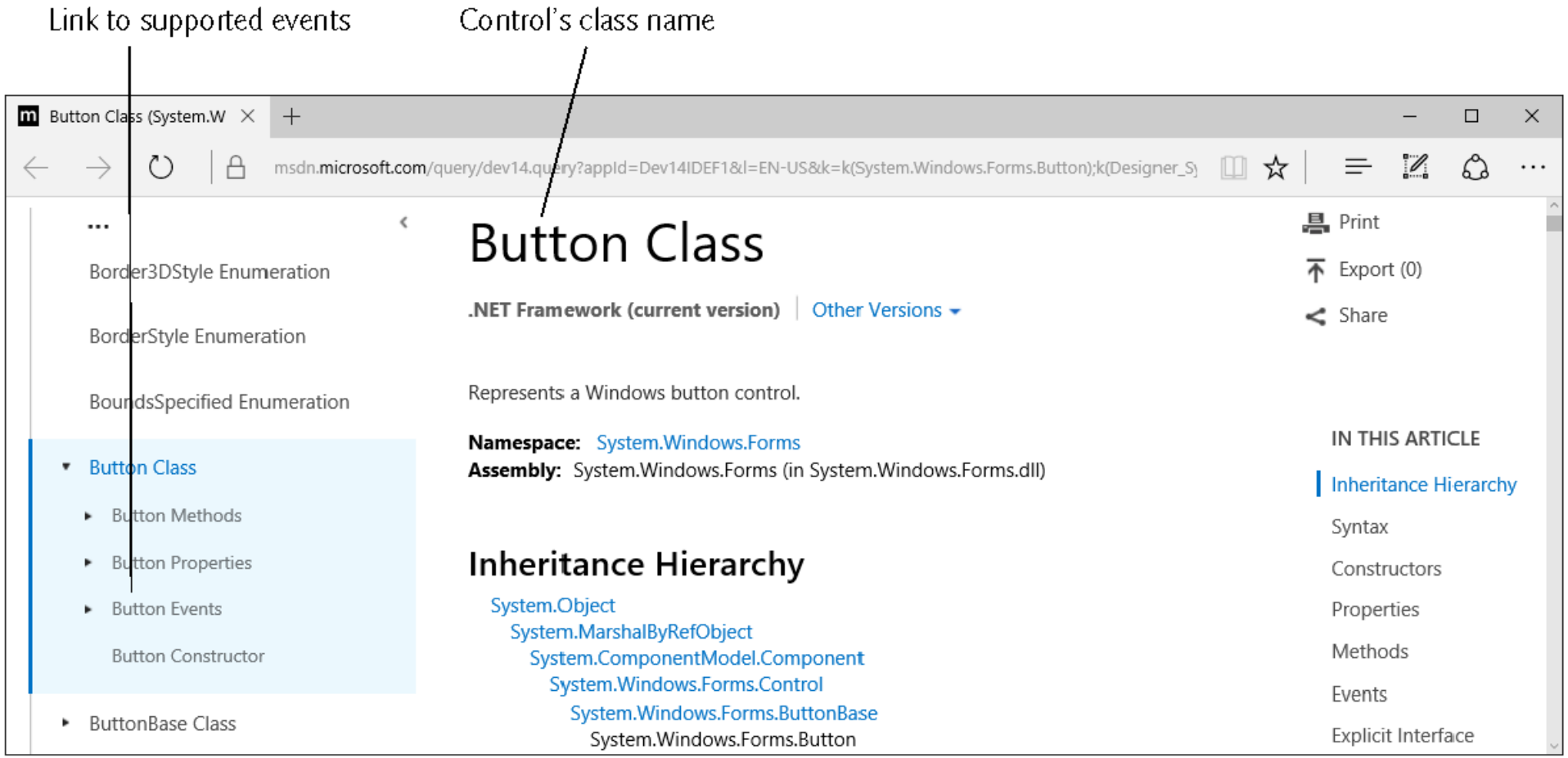


Fig. 14.9 | Link to list of `Button` events.

14.3 Event Handling (Cont.)

- Click the name of an event to view its description and examples of its use.
- We selected the `Click` event to display the information in Fig. 14.10.

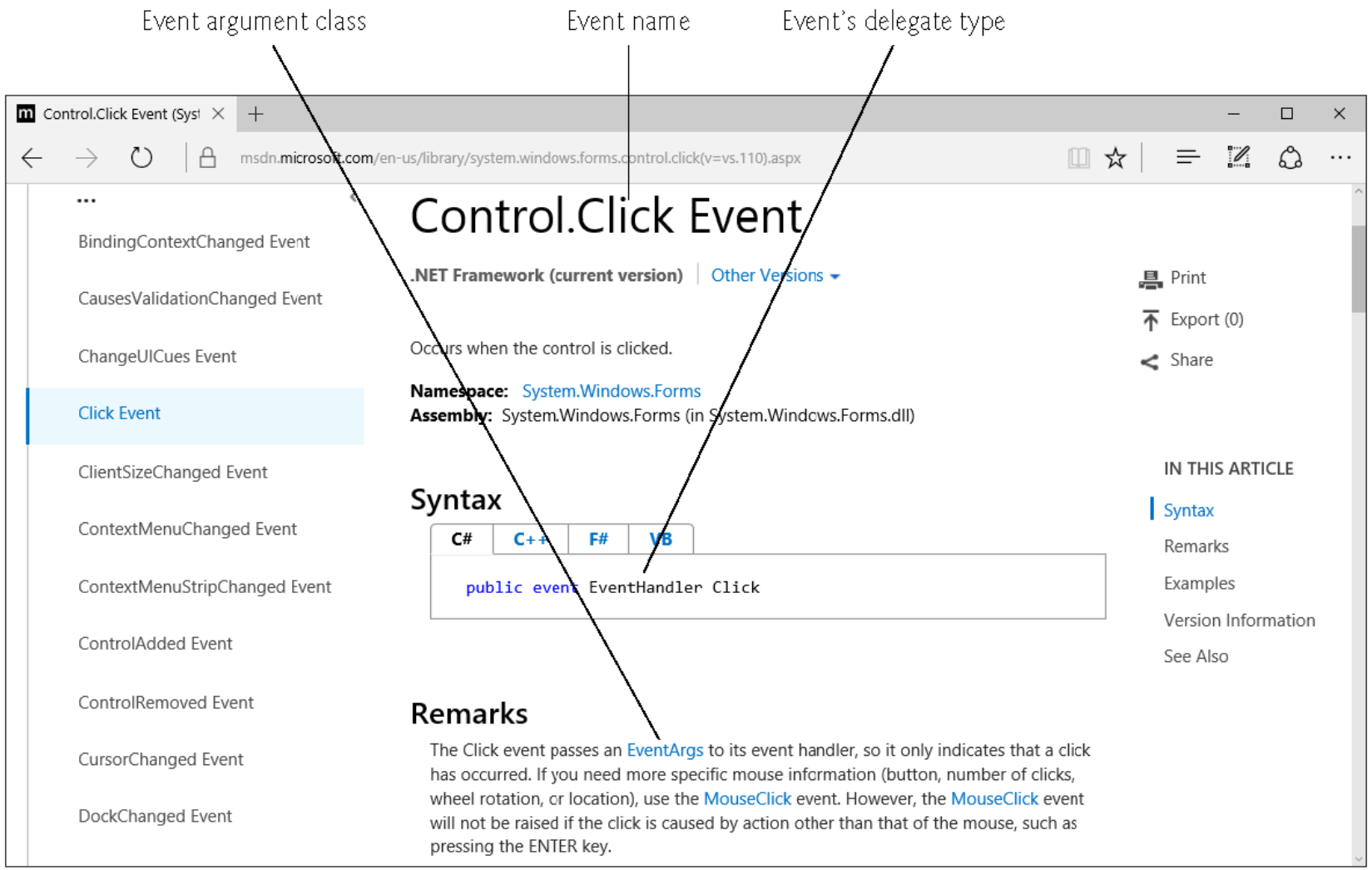


Fig. 14.10 | `Click` event details.

14.4 Control Properties and Layout

- Controls derive from class `Control` (namespace `System.Windows.Forms`).
- Figure 14.11 lists some of class `Control`'s properties and methods.

Class `Control` properties and methods	Description
Common Properties	
`BackColor`	The control's background color.
`BackgroundImage`	The control's background image.
`Enabled`	Specifies whether the control is *enabled* (i.e., if the user can interact with it). Typically, portions of a *disabled* control appear "grayed out" as a visual indication to the user that the control is disabled.
`Focused`	Indicates whether the control *has the focus* (only available at runtime).
`Font`	The `Font` used to display the control's text.
`ForeColor`	The control's foreground color. This usually determines the color of the text in the `Text` property.

Fig. 14.11 | Class `Control` properties and methods. (Part 1 of 3.)

Class `Control` properties and methods	Description
`TabIndex`	The *tab order* of the control. When the *Tab* key is pressed, the focus transfers between controls based on the tab order. You can set this order.
`TabStop`	If `true`, then a user can give focus to this control via the *Tab* key.
`Text`	The text associated with the control. The location and appearance of the text vary depending on the type of control.
`Visible`	Indicates whether the control is visible.

Fig. 14.11 | Class `Control` properties and methods. (Part 2 of 3.)

Class `Control` properties and methods	Description
Common Methods	
`Hide`	Hides the control (sets the `Visible` property to `false`).
`Select`	*Acquires the focus.*
`Show`	Shows the control (sets the `Visible` property to `true`).

Fig. 14.11 | Class `Control` properties and methods. (Part 3 of 3.)

14.4 Control Properties and Layout (Cont.)

- ***Anchoring and Docking***
- 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).

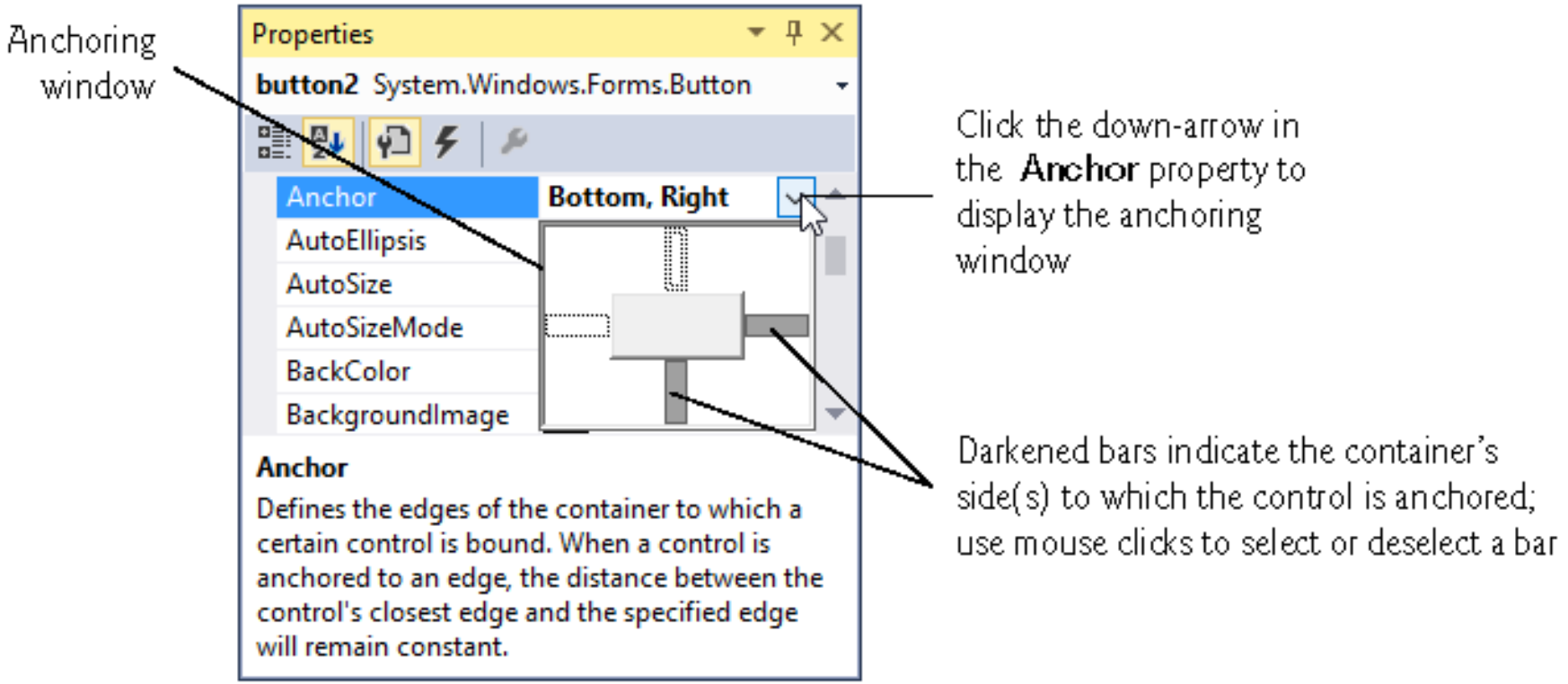


Fig. 14.12 | Manipulating the **Anchor** property of a control.

14.4 Control Properties and Layout (Cont.)

- Execute the app and enlarge the `Form` (Fig. 14.13).

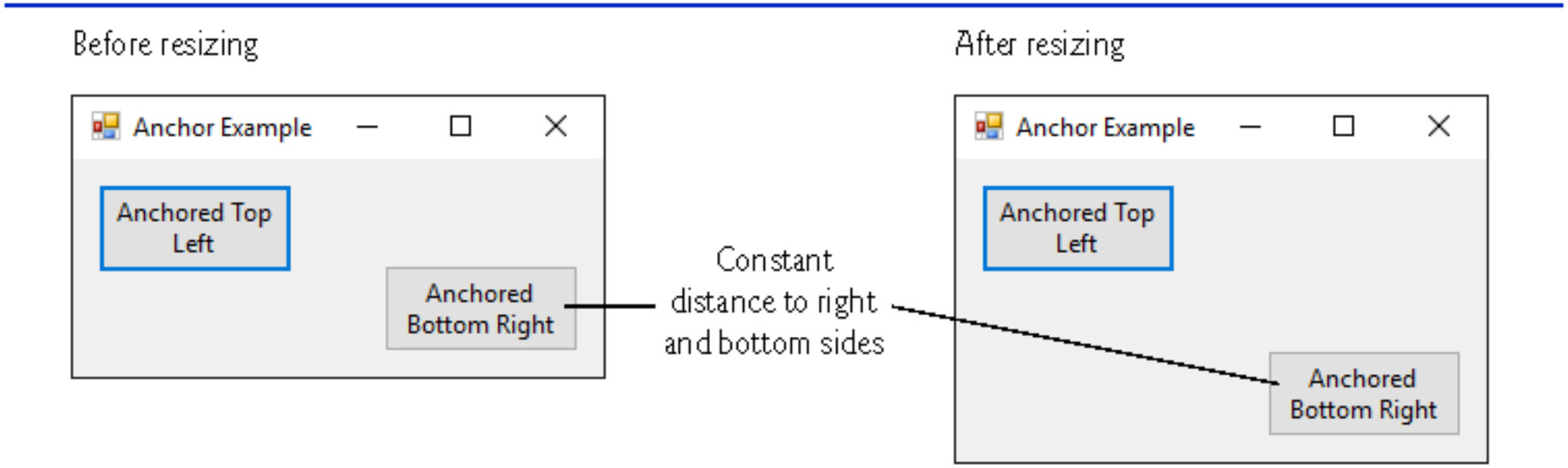


Fig. 14.13 | Anchoring demonstration.

14.4 Control Properties and Layout (Cont.)

- Docking attaches a control to a container such that the control stretches across an entire side or fills an entire area.
- 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.

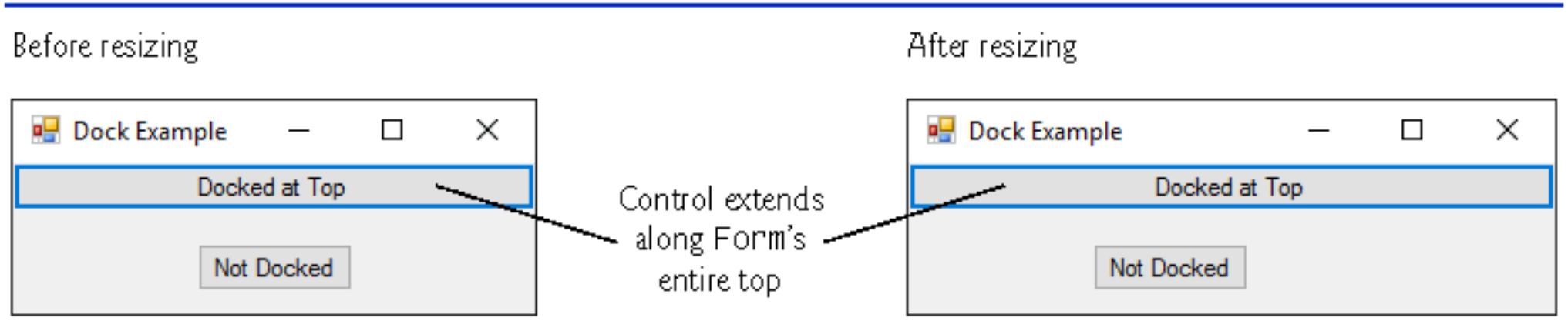


Fig. 14.14 | Docking a **Button** to the top of a **Form**.

`Control` layout properties	Description
`Anchor`	Causes a control to remain at a fixed distance from the side(s) of the container even when the container is resized.
`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. The default is `0`, causing the control to appear flush with the container's sides.
`Location`	Specifies the location (as a set of coordinates) of the upper-left corner of the control, in relation to its container's upper-left corner.
`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`, respectively.

Fig. 14.15 | `Control` layout properties.

14.4 Control Properties and Layout (Cont.)

- ***Using Visual Studio To Edit a GUI's Layout***
- 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.

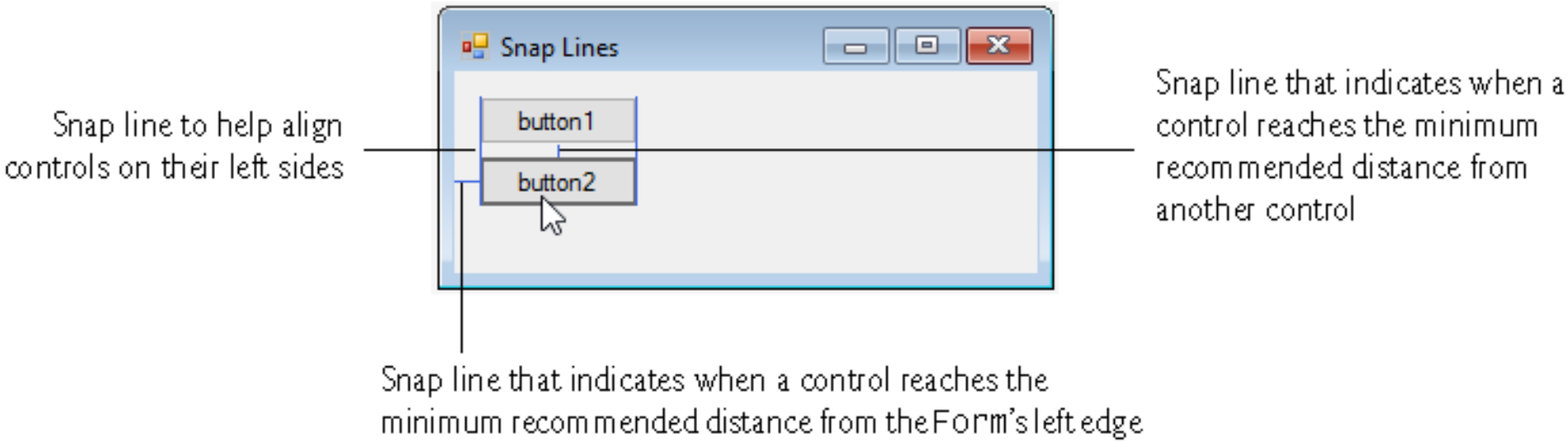


Fig. 14.16 | Snap lines for aligning controls.

14.5 Labels, TextBoxes and Buttons

- `Labels` display text that the user cannot directly modify.
- Figure 14.7 lists common `Label` properties.

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—horizontally (left, center or right) and vertically (top, middle or bottom). The default is top, left.

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`.

TextBox properties and an event	Description
Common Properties	
`AcceptsReturn`	If `true` in a multiline `TextBox`, pressing *Enter* in the `TextBox` creates a new line. If `false` (the default), pressing *Enter* is the same as pressing the default `Button` on the `Form`. The default `Button` is the one assigned to a `Form`'s `AcceptButton` property.
`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 (`None`—the default, `Horizontal`, `Vertical` or `Both`).

Fig. 14.18 | `TextBox` properties and an event. (Part 1 of 2.)

TextBox properties and an event	Description
`Text`	The `TextBox`'s text content.
`UseSystemPasswordChar`	When `true`, the `TextBox` becomes a password `TextBox`, and the system-specified character masks each character the user types.
Common Event	
`TextChanged`	Generated when the text changes in a `TextBox` (i.e., when the user adds or deletes characters). When you double click the `TextBox` control in **Design** mode, an empty event handler for this event is generated.

Fig. 14.18 | `TextBox` properties and an event. (Part 2 of 2.)

14.5 Labels, TextBoxes and Buttons (Cont.)

- A button is a control that the user clicks to trigger a specific action or to select an option in a program.
- A program can use several types of buttons, but all the button classes derive from class `ButtonBase`.

14.5 Labels, TextBoxes and Buttons (Cont.)

- Figure 14.19 lists common properties and a common eventof class `Button`.

`Button` properties and an event	Description
Common Properties	
`Text`	Specifies the text displayed on the `Button` face.
`FlatStyle`	Modifies a `Button`'s appearance—`Flat` (for the `Button` to display without a three-dimensional appearance), `Popup` (for the `Button` to appear flat until the user moves the mouse pointer over the `Button`), `Standard` (three-dimensional) and `System`, where the `Button`'s appearance is controlled by the operating system. The default value is `Standard`.
Common Event	
`Click`	Generated when the user clicks the `Button`. When you double click a `Button` in design view, an empty event handler for this event is created.

Fig. 14.19 | `Button` properties and an event.

14.5 Labels, TextBoxes and Buttons (Cont.)

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

```
1  // Fig. 14.20: LabelTextBoxButtonTestForm.cs
2  // Using a TextBox, Label and Button to display
3  // the hidden text in a password TextBox.
4  using System;
5  using System.Windows.Forms;
6
7  namespace LabelTextBoxButtonTest
8  {
9     // Form that creates a password TextBox and
10    // a Label to display TextBox contents
11    public partial class LabelTextBoxButtonTestForm : Form
12    {
13       // default constructor
14       public LabelTextBoxButtonTestForm()
15       {
16          InitializeComponent();
17       }
```

Fig. 14.20 | Program to display hidden text in a password box.

```
18
19          // display user input in Label
20          private void displayPasswordButton_Click(object sender, EventArgs e)
21          {
22             // display the text that the user typed
23             displayPasswordLabel.Text = inputPasswordTextBox.Text;
24          }
25       }
26    }
```

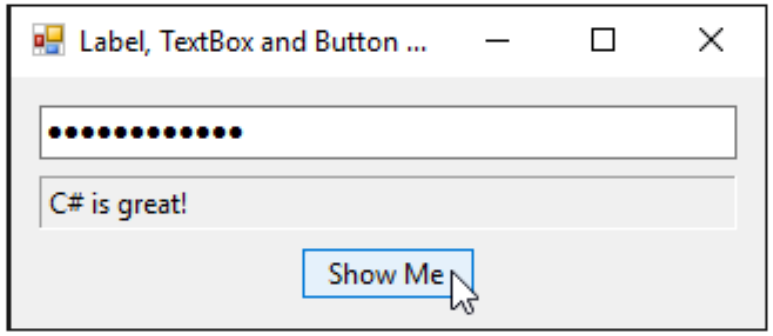


Fig. 14.20 | Program to display hidden text in a password box.

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.

Look-and-Feel Observation 14.2

Panels and GroupBoxes can contain other Panels and GroupBoxes for more complex layouts.

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. The default value is `false`.
`BorderStyle`	Sets the border of the `Panel`. The default value is `None`; other options are `Fixed3D` and `FixedSingle`.
`Controls`	The set of controls that the `Panel` contains.

Fig. 14.22 | `Panel` properties.

Look-and-Feel Observation 14.3

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.)

- 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).

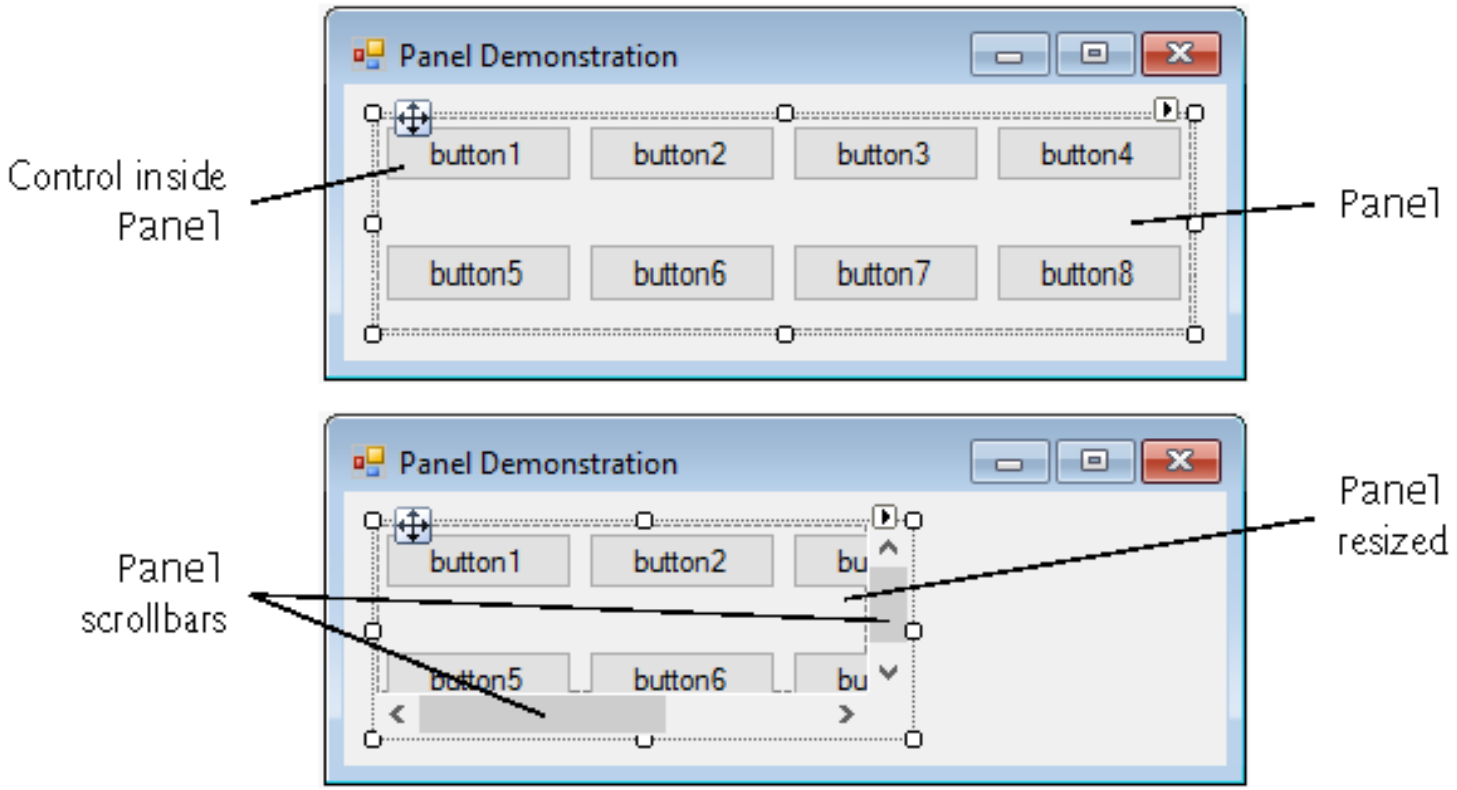


Fig. 14.23 | Creating a **Panel** with scrollbars.

14.6 GroupBoxes and Panels (Cont.)

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

```
1  // Fig. 14.24: GroupBoxPanelExampleForm.cs
2  // Using GroupBoxes and Panels to arrange Buttons.
3  using System;
4  using System.Windows.Forms;
5
6  namespace GroupBoxPanelExample
7  {
8     // Form that displays a GroupBox and a Panel
9     public partial class GroupBoxPanelExampleForm : Form
10    {
11       // default constructor
12       public GroupBoxPanelExampleForm()
13       {
14          InitializeComponent();
15       }
16
17       // event handler for Hi Button
18       private void hiButton_Click(object sender, EventArgs e)
19       {
20          messageLabel.Text = "Hi pressed"; // change text in Label
21       }
```

Fig. 14.24 | Using GroupBoxes and Panels to arrange Buttons. (Part 1 of 3.)

```
22
23          // event handler for Bye Button
24          private void byeButton_Click(object sender, EventArgs e)
25          {
26             messageLabel.Text = "Bye pressed"; // change text in Label
27          }
28
29          // event handler for Far Left Button
30          private void leftButton_Click(object sender, EventArgs e)
31          {
32             messageLabel.Text = "Far Left pressed"; // change text in Label
33          }
34
35          // event handler for Far Right Button
36          private void rightButton_Click(object sender, EventArgs e)
37          {
38             messageLabel.Text = "Far Right pressed"; // change text in Label
39          }
40       }
41    }
```

Fig. 14.24 | Using GroupBoxes and Panels to arrange Buttons. (Part 2 of 3.)

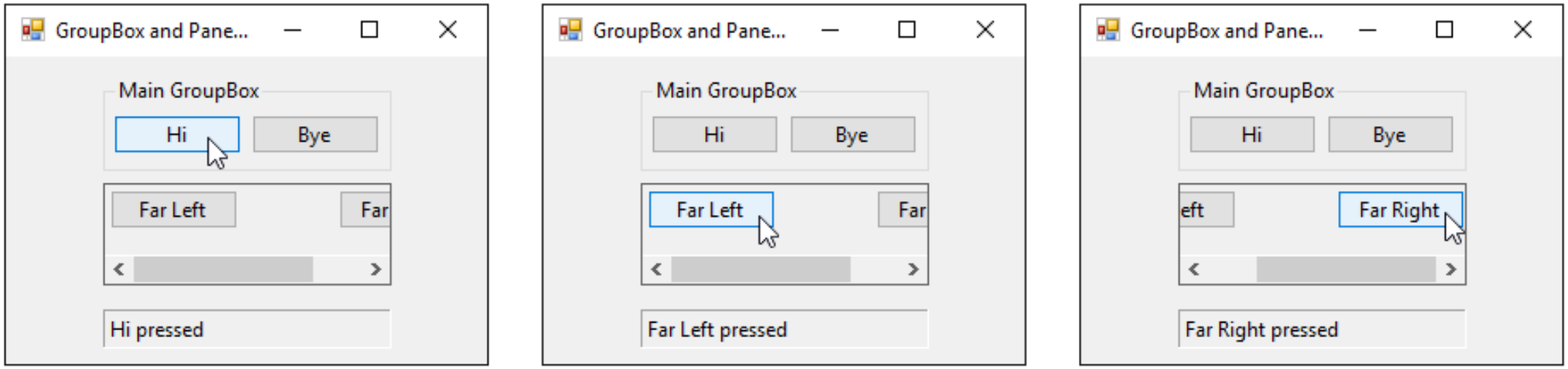


Fig. 14.24 | Using `GroupBox`es and `Panel`s to arrange `Button`s. (Part 3 of 3.)

14.7 CheckBoxes and RadioButtons

- ***CheckBoxes***
- A `CheckBox` is a small square that either is blank or contains a check mark.
- You can also configure a `CheckBox` to toggle between three states by setting its `ThreeState` property to `True`.
- Any number of `CheckBoxes` can be selected at a time.

CheckBox properties and events	Description
Common Properties	
`Appearance`	By default, this property is set to `Normal`, and the `CheckBox` displays as a traditional checkbox. If it's set to `Button`, the `CheckBox` displays as a `Button` that looks pressed when the `CheckBox` is checked.
`Checked`	Indicates whether the `CheckBox` is *checked* (contains a check mark) or unchecked (blank). This property returns a `bool` value. The default is `false` (*unchecked*).
`CheckState`	Indicates whether the `CheckBox` is *checked* or *unchecked* with a value from the `CheckState` enumeration (`Checked`, `Unchecked` or `Indeterminate`). `Indeterminate` is used when it's unclear whether the state should be `Checked` or `Unchecked`. When `CheckState` is set to `Indeterminate`, the `CheckBox` is usually shaded.

Fig. 14.25 | CheckBox properties and an event. (Part 1 of 2.)

CheckBox properties and events	Description
`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*. By default, this property is `false` and the `CheckBox` has only two states—*checked* and *unchecked*. When `true`, `Checked` returns `true` for both the *checked* and *indeterminate* states.
Common Event	
`CheckedChanged`	Generated *any* time the `Checked` or `CheckState` property changes. This is a `CheckBox`'s default event. When a user double clicks the `CheckBox` control in design view, an empty event handler for this event is generated.

Fig. 14.25 | `CheckBox` properties and an event. (Part 2 of 2.)

14.7 CheckBoxes and RadioButtons (cont.)

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

```
1  // Fig. 14.26: CheckBoxTestForm.cs
2  // Using CheckBoxes to toggle italic and bold styles.
3  using System;
4  using System.Drawing;
5  using System.Windows.Forms;
6
7  namespace CheckBoxTest
8  {
9     // Form contains CheckBoxes to allow the user to modify sample text
10    public partial class CheckBoxTestForm : Form
11    {
12       // default constructor
13       public CheckBoxTestForm()
14       {
15          InitializeComponent();
16       }
17
```

Fig. 14.26 | Using CheckBoxes to toggle italic and bold styles. (Part 1 of 3.)

```
18          // toggle the font style between bold and
19          // not bold based on the current setting
20          private void boldCheckBox_CheckedChanged(object sender, EventArgs e)
21          {
22             outputLabel.Font = new Font(outputLabel.Font,
23                outputLabel.Font.Style ^ FontStyle.Bold);
24          }
25
26          // toggle the font style between italic and
27          // not italic based on the current setting
28          private void italicCheckBox_CheckedChanged(
29             object sender, EventArgs e)
30          {
31             outputLabel.Font = new Font(outputLabel.Font,
32                outputLabel.Font.Style ^ FontStyle.Italic);
33          }
34       }
35    }
```

Fig. 14.26 | Using CheckBoxes to toggle italic and bold styles. (Part 2 of 3.)

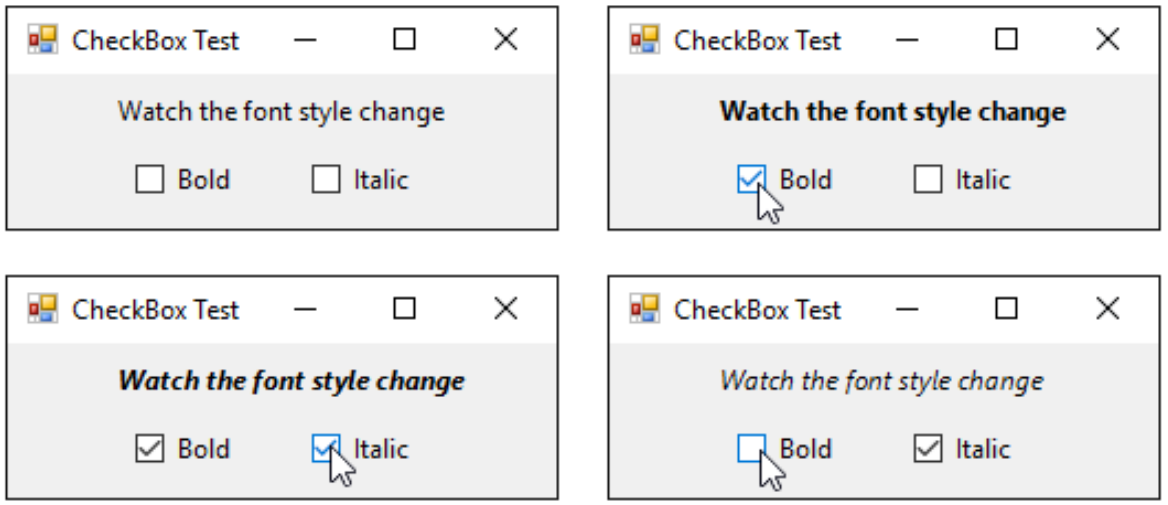


Fig. 14.26 | Using **CheckBox**es to toggle italic and bold styles. (Part 3 of 3.)

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.
- ***Combining Font Styles with Bitwise Operators***
- 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.)

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

Look-and-Feel Observation 14.4

Use `RadioButton`*s when the user should choose only one option in a group. Use* `CheckBox`*es when the user should be able to choose multiple options in a group.*

RadioButton properties and an 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. When you double click a `RadioButton` control in design view, an empty event handler for this event is generated.

Fig. 14.27 | `RadioButton` properties and an event.

Software Engineering Observation 14.1

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.

14.7 CheckBoxes and RadioButtons (Cont.)

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

```
1  // Fig. 14.28: RadioButtonsTestForm.cs
2  // Using RadioButtons to set message window options.
3  using System;
4  using System.Windows.Forms;
5
6  namespace RadioButtonsTest
7  {
8     // Form contains several RadioButtons--user chooses one
9     // from each group to create a custom MessageBox
10    public partial class RadioButtonsTestForm : Form
11    {
12       // create variables that store the user's choice of options
13       private MessageBoxIcon IconType { get; set; }
14       private MessageBoxButtons ButtonType { get; set; }
15
16       // default constructor
17       public RadioButtonsTestForm()
18       {
19          InitializeComponent();
20       }
```

Fig. 14.28 | Using RadioButtons to set message-window options. (Part 1 of 10.)

```
21
22          // change Buttons based on option chosen by sender
23          private void buttonType_CheckedChanged(object sender, EventArgs e)
24          {
25             if (sender == okRadioButton) // display OK Button
26             {
27                ButtonType = MessageBoxButtons.OK;
28             }
29             // display OK and Cancel Buttons
30             else if (sender == okCancelRadioButton)
31             {
32                ButtonType = MessageBoxButtons.OKCancel;
33             }
34             // display Abort, Retry and Ignore Buttons
35             else if (sender == abortRetryIgnoreRadioButton)
36             {
37                ButtonType = MessageBoxButtons.AbortRetryIgnore;
38             }
```

Fig. 14.28 | Using `RadioButtons` to set message-window options. (Part 2 of 10.)

```
39              // display Yes, No and Cancel Buttons
40              else if (sender == yesNoCancelRadioButton)
41              {
42                 ButtonType = MessageBoxButtons.YesNoCancel;
43              }
44              // display Yes and No Buttons
45              else if (sender == yesNoRadioButton)
46              {
47                 ButtonType = MessageBoxButtons.YesNo;
48              }
49              // only one option left--display Retry and Cancel Buttons
50              else
51              {
52                 ButtonType = MessageBoxButtons.RetryCancel;
53              }
54           }
55
```

Fig. 14.28 | Using `RadioButtons` to set message-window options. (Part 3 of 10.)

```
56          // change Icon based on option chosen by sender
57          private void iconType_CheckedChanged(object sender, EventArgs e)
58          {
59             if (sender == asteriskRadioButton) // display asterisk Icon
60             {
61                IconType = MessageBoxIcon.Asterisk;
62             }
63             // display error Icon
64             else if (sender == errorRadioButton)
65             {
66                IconType = MessageBoxIcon.Error;
67             }
68             // display exclamation point Icon
69             else if (sender == exclamationRadioButton)
70             {
71                IconType = MessageBoxIcon.Exclamation;
72             }
73             // display hand Icon
74             else if (sender == handRadioButton)
75             {
76                IconType = MessageBoxIcon.Hand;
77             }
```

Fig. 14.28 | Using RadioButtons to set message-window options. (Part 4 of 10.)

```
78              // display information Icon
79              else if (sender == informationRadioButton)
80              {
81                 IconType = MessageBoxIcon.Information;
82              }
83              // display question mark Icon
84              else if (sender == questionRadioButton)
85              {
86                 IconType = MessageBoxIcon.Question;
87              }
88              // display stop Icon
89              else if (sender == stopRadioButton)
90              {
91                 IconType = MessageBoxIcon.Stop;
92              }
93              // only one option left--display warning Icon
94              else
95              {
96                 IconType = MessageBoxIcon.Warning;
97              }
98           }
```

Fig. 14.28 | Using `RadioButtons` to set message-window options. (Part 5 of 10.)

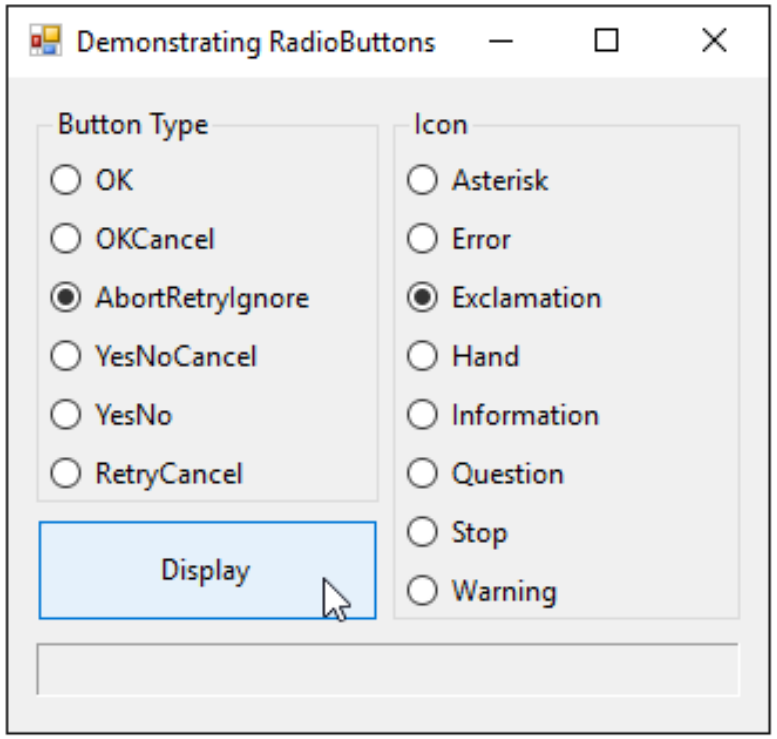
```
99
100         // display MessageBox and Button user pressed
101         private void displayButton_Click(object sender, EventArgs e)
102         {
103            // display MessageBox and store
104            // the value of the Button that was pressed
105            DialogResult result = MessageBox.Show(
106               "This is your Custom MessageBox.", "Custon MessageBox",
107               ButtonType, IconType);
108
109            // check to see which Button was pressed in the MessageBox
110            // change text displayed accordingly
111            switch (result)
112            {
113               case DialogResult.OK:
114                  displayLabel.Text = "OK was pressed.";
115                  break;
116               case DialogResult.Cancel:
117                  displayLabel.Text = "Cancel was pressed.";
118                  break;
119               case DialogResult.Abort:
120                  displayLabel.Text = "Abort was pressed.";
121                  break;
```

Fig. 14.28 | Using `RadioButtons` to set message-window options. (Part 6 of 10.)

```
122                 case DialogResult.Retry:
123                     displayLabel.Text = "Retry was pressed.";
124                     break;
125                 case DialogResult.Ignore:
126                     displayLabel.Text = "Ignore was pressed.";
127                     break;
128                 case DialogResult.Yes:
129                     displayLabel.Text = "Yes was pressed.";
130                     break;
131                 case DialogResult.No:
132                     displayLabel.Text = "No was pressed.";
133                     break;
134             }
135         }
136     }
137 }
```

Fig. 14.28 | Using `RadioButtons` to set message-window options. (Part 7 of 10.)

a) GUI for testing `RadioButtons`



b) `AbortRetryIgnore` button type

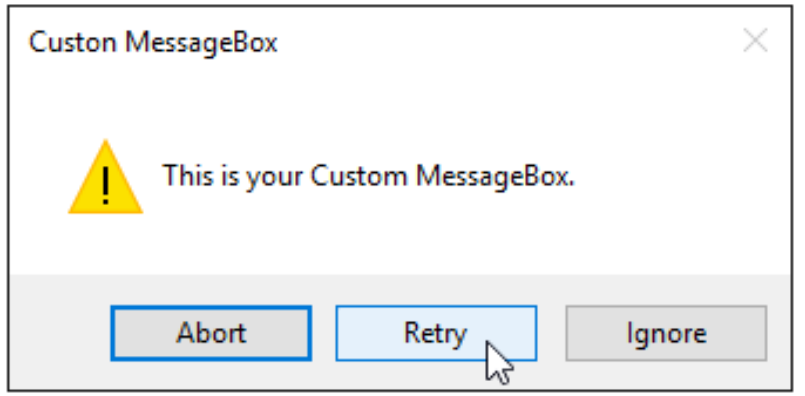
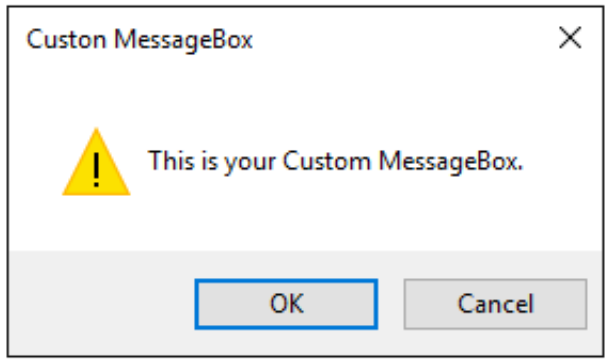
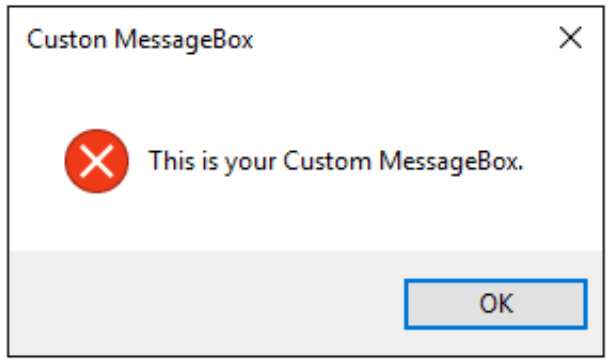


Fig. 14.28 | Using **`RadioButtons`** to set message-window options. (Part 8 of 10.)

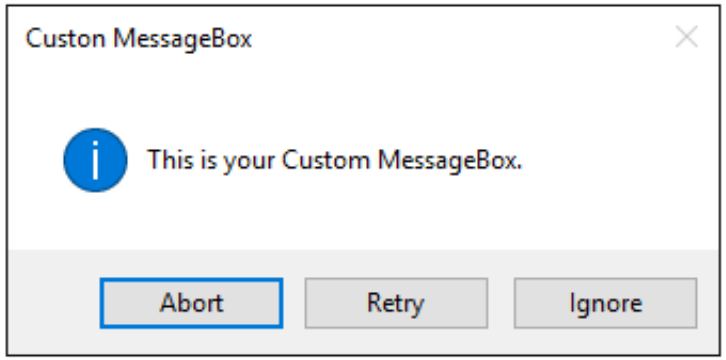
c) `OKCancel` button type



d) `OK` button type



e) `AbortRetryIgnore` button type



f) `YesNoCancel` button type

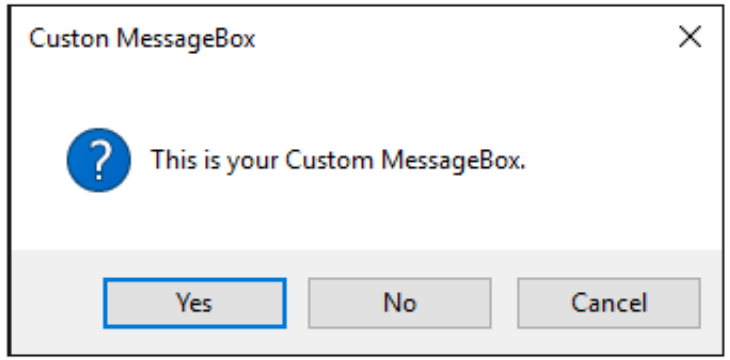
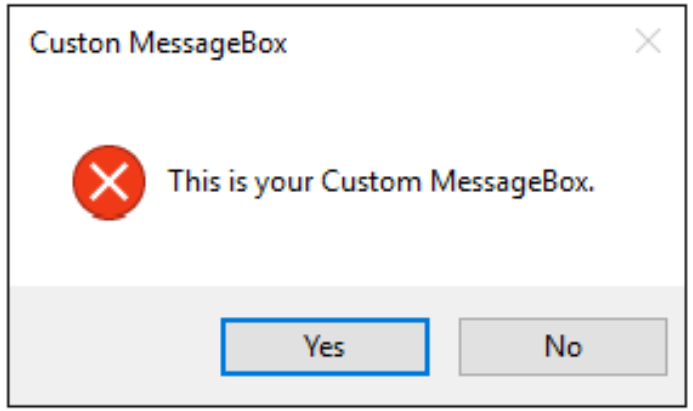


Fig. 14.28 | Using **`RadioButton`**s to set message-window options. (Part 9 of 10.)

g) `YesNo` button type



h) `RetryCancel` button type

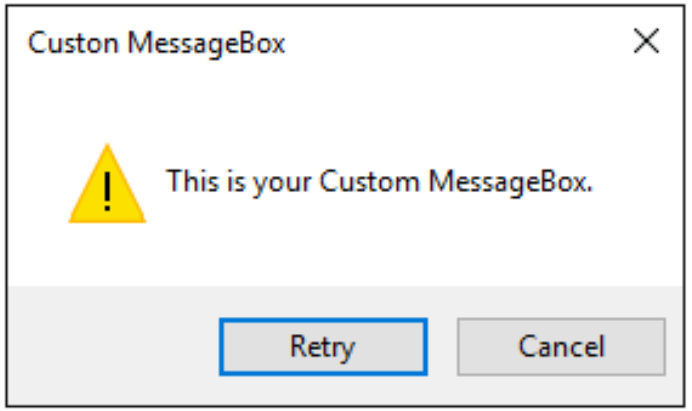


Fig. 14.28 | Using **`RadioButton`**s to set message-window options. (Part 10 of 10.)

14.8 PictureBoxes

- A `PictureBox` displays an image.
- Figure 14.29 describes common `PictureBox` properties and a common event.

`PictureBox` properties and an event	Description
Common Properties	
`Image`	Sets the image to display in the `PictureBox`.
`SizeMode`	Controls image sizing and positioning. Values are `Normal` (default), `StretchImage`, `AutoSize`, `CenterImage` and `Zoom`. `Normal` places the image in the `PictureBox`'s top-left corner, and `CenterImage` puts the image in the middle. Both truncate the image if it's too large. `StretchImage` fits the image in the `PictureBox`. `AutoSize` resizes the `PictureBox` to fit the image. `Zoom` resizes the image to to fit the `PictureBox` but maintains the original aspect ratio.

Fig. 14.29 | `PictureBox` properties and an event. (Part 1 of 2.)

`PictureBox` properties and an event	Description
Common Event	
`Click`	Occurs when the user clicks a control. When you double click this control in the designer, an event handler is generated for this event.

Fig. 14.29 | `PictureBox` properties and an event. (Part 2 of 2.)

14.8 PictureBoxes

- Figure 14.30 uses a `PictureBox` to display bitmap images.

```
1  // Fig. 14.30: PictureBoxTestForm.cs
2  // Using a PictureBox to display images.
3  using System;
4  using System.Drawing;
5  using System.Windows.Forms;
6
7  namespace PictureBoxTest
8  {
9     // Form to display different images when Button is clicked
10    public partial class PictureBoxTestForm : Form
11    {
12       private int ImageNumber { get; set; } = -1; // image to display
13
14       // default constructor
15       public PictureBoxTestForm()
16       {
17          InitializeComponent();
18       }
```

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

```
19
20          // change image whenever Next Button is clicked
21          private void nextButton_Click(object sender, EventArgs e)
22          {
23             ImageNumber = (ImageNumber + 1) % 3; // cycles from 0 to 2
24
25             // retrieve image from resources and load into PictureBox
26             imagePictureBox.Image =
27                (Image) (Properties.Resources.ResourceManager.GetObject(
28                $"image{ImageNumber}"));
29          }
30       }
31    }
```

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

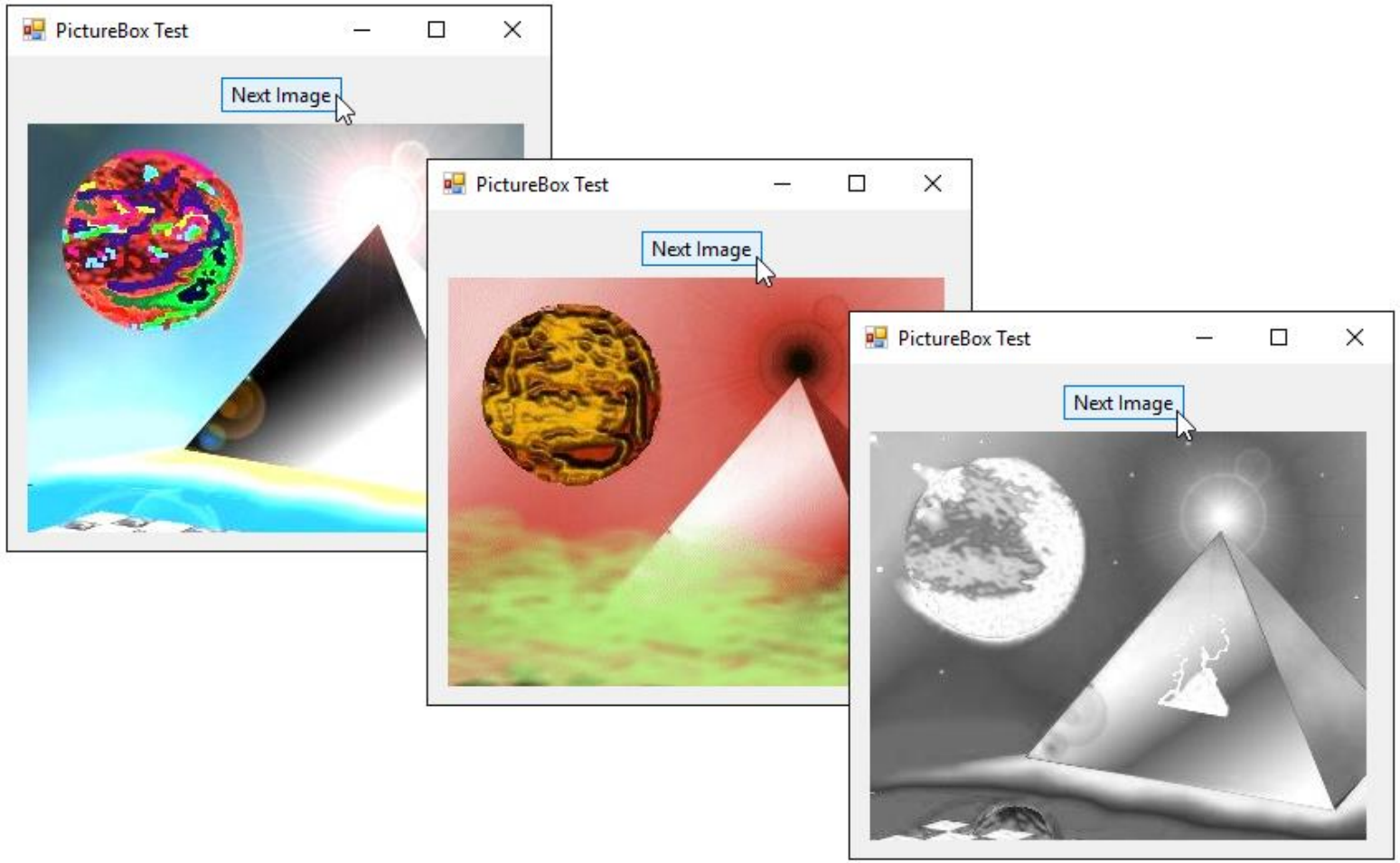


Fig. 14.30 | Using a `PictureBox` to display images. (Part 3 of 3.)

14.8 PictureBoxes (Cont.)

- ***Using Resources Programmatically***
- Embedding the images in the app 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.

14.8 PictureBoxes (Cont.)

- A project's resources are stored in its `Resources` class.
- Its `ResourceManager` object allows interacting with the resources programmatically.
- To access an image, use the method `GetObject`, which takes as an argument the resource name as it appears in the **Resources** tab.
- The `Resources` class also provides direct access with expressions of the form `Resources.resourceName`.

14.9 ToolTips

- Recall that tool tips are the helpful text that appears when the mouse hovers over an item in a GUI.
- Figure 14.31 describes common properties and a common event of class `ToolTip`.

`ToolTip` properties and an event	Description
Common Properties	
`AutoPopDelay`	The amount of time (in milliseconds) that the tool tip appears while the mouse is over a control.
`InitialDelay`	The amount of time (in milliseconds) that a mouse must hover over a control before a tool tip appears.
`ReshowDelay`	The amount of time (in milliseconds) between which two different tool tips appear (when the mouse is moved from one control to another).
Common Event	
`Draw`	Raised when the tool tip is about to be displayed. This event allows programmers to modify the appearance of the tool tip.

Fig. 14.31 | `ToolTip` properties and an event.

14.9 ToolTips

- A `ToolTip` component appears in the component tray—the 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.

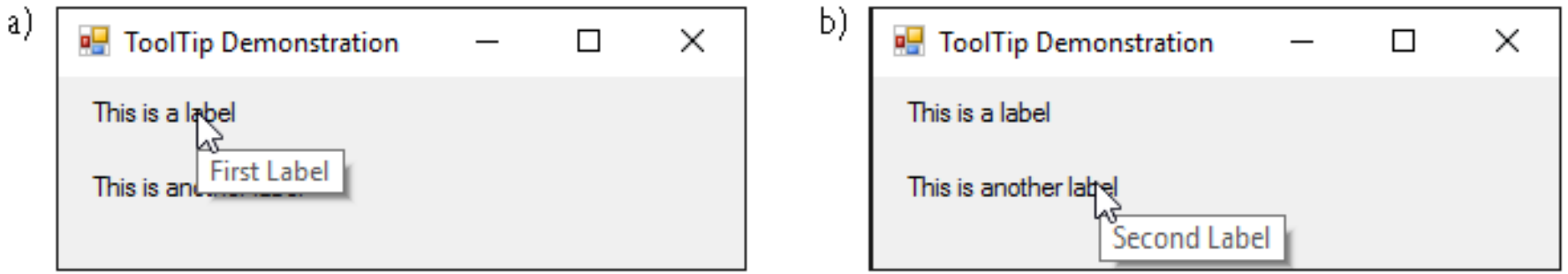


Fig. 14.32 | Demonstrating the `ToolTip` component.

14.9 ToolTips (Cont.)

- Figure 14.33 shows the `ToolTip` in the component tray.
- We set the tool tip text for the first `Label` to `"First Label"` and the tool tip text for the second `Label` to `"Second Label"`.

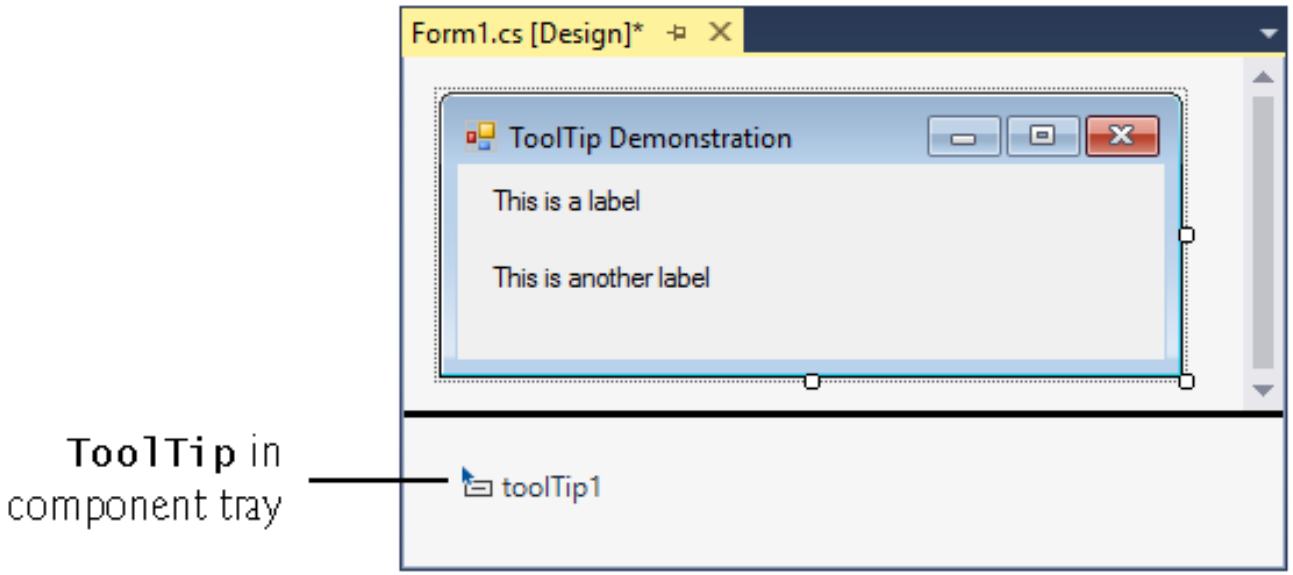


Fig. 14.33 | Demonstrating the component tray.

14.9 ToolTips (Cont.)

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

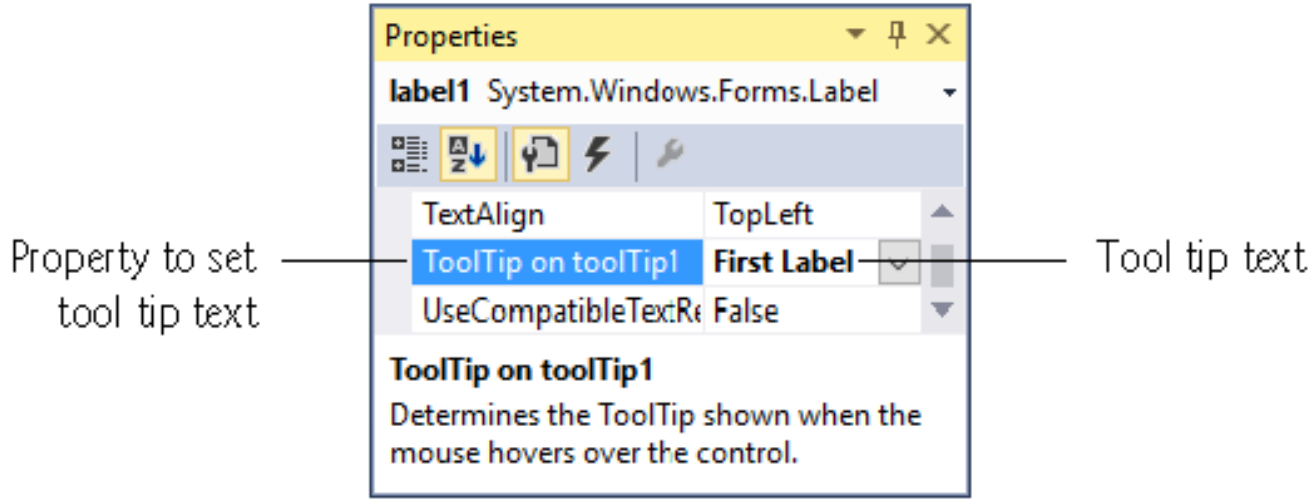


Fig. 14.34 | Setting a control's tool tip text.

14.10 NumericUpDown Control (cont.)

- 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.
- Figure 14.35 describes common `NumericUpDown` properties and an event.

NumericUpDown properties and an event	Description
Common Properties	
`DecimalPlaces`	Specifies how many decimal places to display in the control.
`Increment`	Specifies by how much the current number in the control changes when the user clicks the control's 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. This property can be used to display these `Buttons` either to the left or to the right of the control.
`Value`	The numeric value currently displayed in the control.

Fig. 14.35 | `NumericUpDown` properties and an event. (Part 1 of 2.)

`NumericUpDown` properties and an event	Description
Common Event	
`ValueChanged`	This event is raised when the value in the control is changed. This is the default event for the `NumericUpDown` control.

Fig. 14.35 | `NumericUpDown` properties and an event. (Part 2 of 2.)

14.10 NumericUpDown Control

- Figure 14.36 demonstrates a `NumericUpDown` control in a GUI app 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.

```
1  // Fig. 14.36: InterestCalculatorForm.cs
2  // Demonstrating the NumericUpDown control.
3  using System;
4  using System.Windows.Forms;
5
6  namespace NumericUpDownTest
7  {
8     public partial class InterestCalculatorForm : Form
9     {
10       // default constructor
11       public InterestCalculatorForm()
12       {
13          InitializeComponent();
14       }
15
```

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

```
16          private void calculateButton_Click(object sender, EventArgs e)
17          {
18             // retrieve user input
19             decimal principal = decimal.Parse(principalTextBox.Text);
20             double rate = double.Parse(interestTextBox.Text);
21             int year = (int) yearUpDown.Value;
22
23             // set output header
24             string output = "Year\tAmount on Deposit\r\n";
25
26             // calculate amount after each year and append to output
27             for (int yearCounter = 1; yearCounter <= year; ++yearCounter)
28             {
29                decimal amount =  principal *
30                   ((decimal) Math.Pow((1 + rate / 100), yearCounter));
31                output += $"{yearCounter}\t{amount:C}\r\n";
32             }
33
34             displayTextBox.Text = output; // display result
35          }
36       }
37    }
```

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

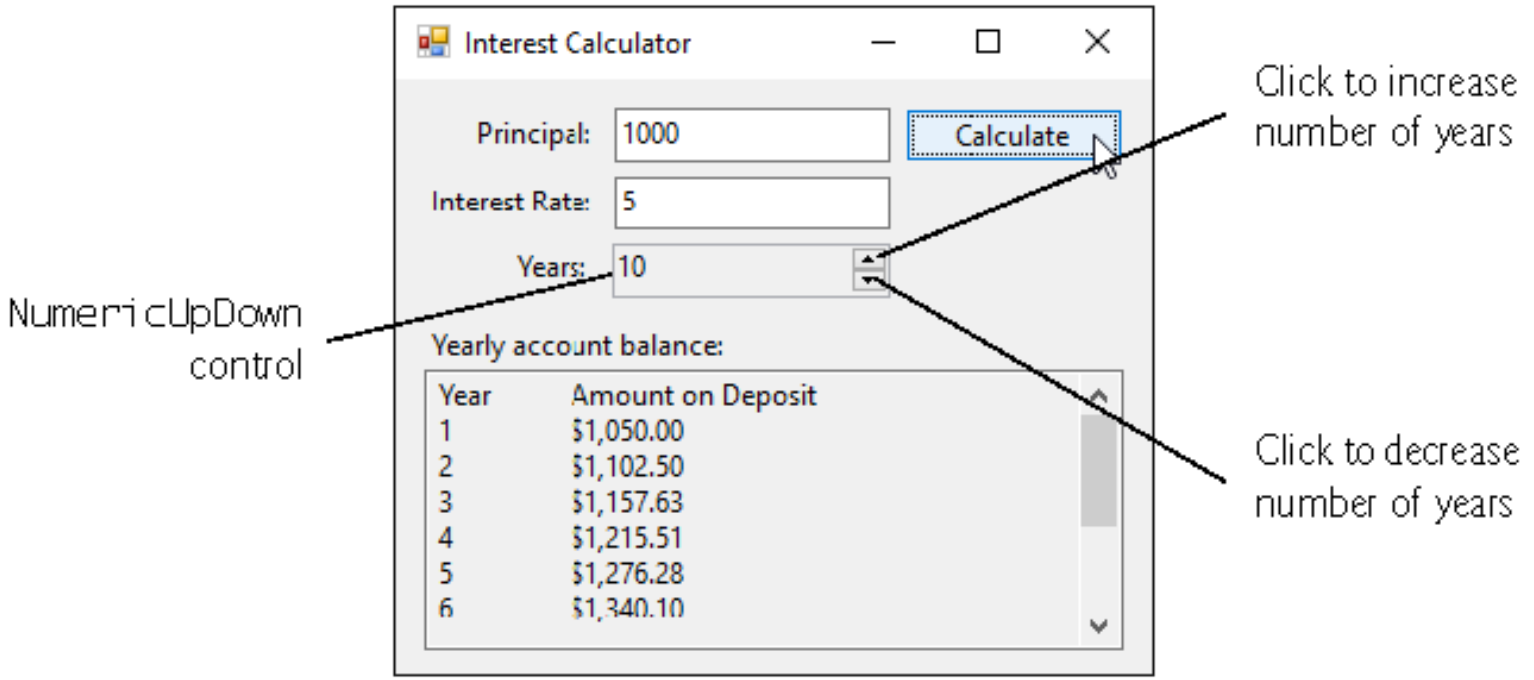


Fig. 14.36 | Demonstrating the `NumericUpDown` control. (Part 3 of 3.)

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 `MouseEventArgs` object, and the delegate type is `MouseEventHandler`.
- `MouseEventArgs` *x*- and *y*-coordinates are relative to the control that generated the event.
- Several common mouse events and event arguments are described in Figure 14.37.

Mouse events and event arguments

Mouse Events with Event Argument of Type **`EventArgs`**	
`MouseEnter`	Mouse cursor enters the control's boundaries.
`MouseHover`	Mouse cursor hovers within the control's boundaries.
`MouseLeave`	Mouse cursor leaves the control's boundaries.
Mouse Events with Event Argument of Type **`MouseEventArgs`**	
`MouseDown`	Mouse button is pressed while the mouse cursor is within a control's boundaries.
`MouseMove`	Mouse cursor is moved while in the control's boundaries.
`MouseUp`	Mouse button is released when the cursor is over the control's boundaries.
`MouseWheel`	Mouse wheel is moved while the control has the focus.

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

Mouse events and event arguments

Class **`MouseEventArgs`** *Properties*

`Button`	Specifies which mouse button was pressed (`Left`, `Right`, `Middle` or `None`).
`Clicks`	The number of times that the mouse button was clicked.
`X`	The *x*-coordinate within the control where the event occurred.
`Y`	The *y*-coordinate within the control where the event occurred.

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

14.11 Mouse-Event Handling

- Figure 14.38 uses mouse events to draw on a `Form`.

```
1  // Fig. 14.38: PainterForm.cs
2  // Using the mouse to draw on a Form.
3  using System;
4  using System.Drawing;
5  using System.Windows.Forms;
6
7  namespace Painter
8  {
9     // creates a Form that's a drawing surface
10    public partial class PainterForm : Form
11    {
12       bool ShouldPaint { get; set; } = false; // whether to paint
13
14       // default constructor
15       public PainterForm()
16       {
17          InitializeComponent();
18       }
```

Fig. 14.38 | Using the mouse to draw on a Form. (Part 1 of 4.)

```
19
20          // should paint when mouse button is pressed down
21          private void PainterForm_MouseDown(object sender, MouseEventArgs e)
22          {
23             // indicate that user is dragging the mouse
24             ShouldPaint = true;
25          }
26
27          // stop painting when mouse button is released
28          private void PainterForm_MouseUp(object sender, MouseEventArgs e)
29          {
30             // indicate that user released the mouse button
31             ShouldPaint = false;
32          }
33
```

Fig. 14.38 | Using the mouse to draw on a Form. (Part 2 of 4.)

```
34          // draw circle whenever mouse moves with its button held down
35          private void PainterForm_MouseMove(object sender, MouseEventArgs e)
36          {
37             if (ShouldPaint) // check if mouse button is being pressed
38             {
39                // draw a circle where the mouse pointer is present
40                using (Graphics graphics = CreateGraphics())
41                {
42                   graphics.FillEllipse(
43                      new SolidBrush(Color.BlueViolet), e.X, e.Y, 4, 4);
44                }
45             }
46          }
47       }
48    }
```

Fig. 14.38 | Using the mouse to draw on a `Form`. (Part 3 of 4.)

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

14.11 Mouse-Event Handling (Cont.)

- Recall from Chapter 13 that the using statement automatically calls `Dispose` on the object that was created in the parentheses following keyword using.
- This is important because `Graphics` objects are a limited resource.
- Calling `Dispose` on a `Graphics` object ensures that its resources are returned to the system for reuse.

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.

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.
Class **`KeyEventArgs`** ***Properties***	
`Alt`	Indicates whether the *Alt* key was pressed.
`Control`	Indicates whether the *Ctrl* key was pressed.
`Shift`	Indicates whether the *Shift* key was pressed.

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

Keyboard events and event arguments	
`KeyCode`	Returns the key code for the key as a value from the `Keys` enumeration. This does not include modifier-key information. It's used to test for a specific key.
`KeyData`	Returns the key code for a key combined with modifier information as a `Keys` value. This property contains all information about the pressed key.
`KeyValue`	Returns the key code as an `int`, rather than as a value from the `Keys` enumeration. This property is used to obtain a numeric representation of the pressed key. The `int` value is known as a Windows virtual key code.
`Modifiers`	Returns a `Keys` value indicating any pressed modifier keys (*Alt*, *Ctrl* and *Shift*). This property is used to determine modifier-key information only.

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

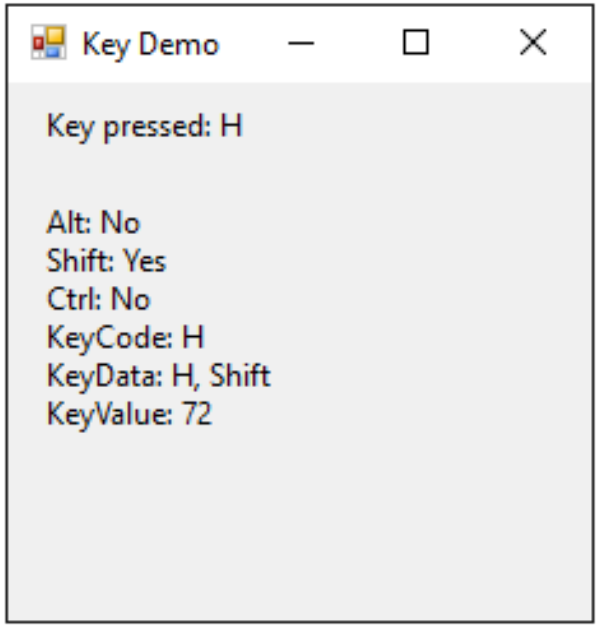
```
1  // Fig. 14.40: KeyDemo.cs
2  // Displaying information about the key the user pressed.
3  using System;
4  using System.Windows.Forms;
5
6  namespace KeyDemo
7  {
8     // Form to display key information when key is pressed
9     public partial class KeyDemo : Form
10    {
11       // default constructor
12       public KeyDemo()
13       {
14          InitializeComponent();
15       }
16
17       // display the character pressed using KeyChar
18       private void KeyDemo_KeyPress(object sender, KeyPressEventArgs e)
19       {
20          charLabel.Text = $"Key pressed: {e.KeyChar}";
21       }
```

Fig. 14.40 | Displaying information about the key the user pressed. (Part 1 of 4.)

```
22
23          // display modifier keys, key code, key data and key value
24          private void KeyDemo_KeyDown(object sender, KeyEventArgs e)
25          {
26             keyInfoLabel.Text =
27                $"Alt: {(e.Alt ? "Yes" : "No")}\n" +
28                $"Shift: {(e.Shift ? "Yes" : "No")}\n"  +
29                $"Ctrl: {(e.Control ? "Yes" : "No")}\n" +
30                $"KeyCode: {e.KeyCode}\n" +
31                $"KeyData: {e.KeyData}\n" +
32                $"KeyValue: {e.KeyValue}";
33          }
34
35          // clear Labels when key released
36          private void KeyDemo_KeyUp(object sender, KeyEventArgs e)
37          {
38             charLabel.Text = "";
39             keyInfoLabel.Text = "";
40          }
41       }
42    }
```

Fig. 14.40 | Displaying information about the key the user pressed. (Part 2 of 4.)

a) *H* pressed



b) *F7* pressed

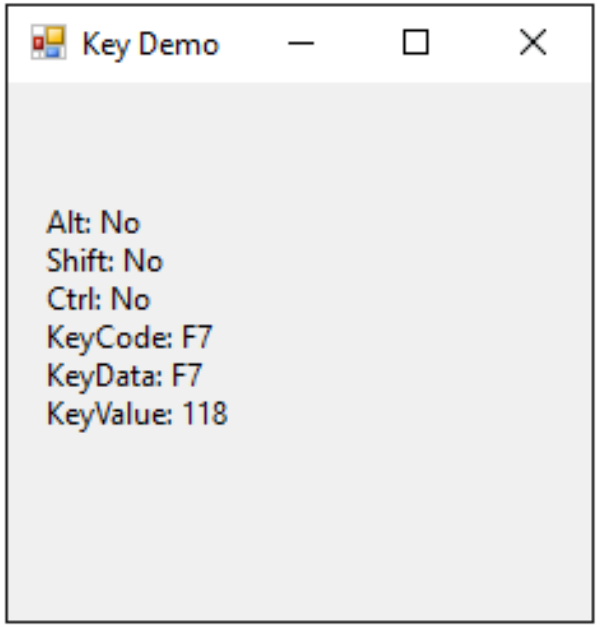
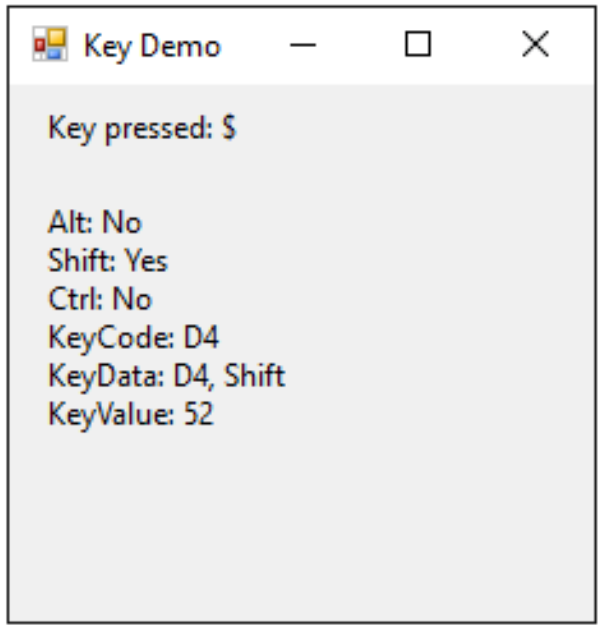


Fig. 14.40 | Displaying information about the key the user pressed. (Part 3 of 4.)

c) $ pressed



d) *Tab* pressed

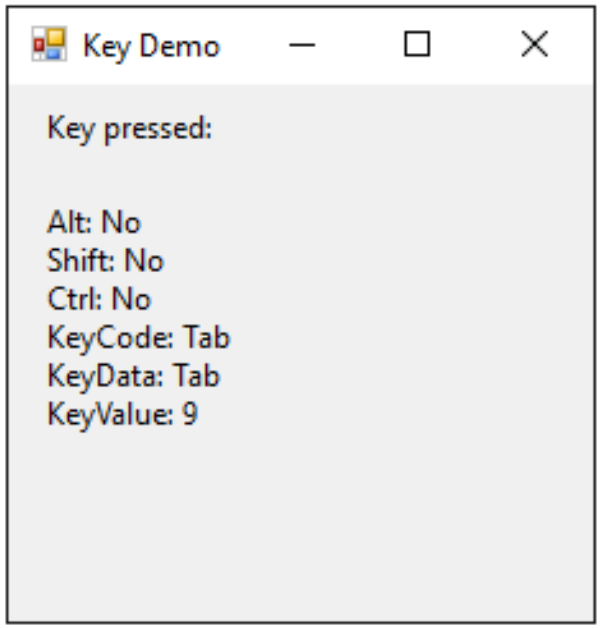


Fig. 14.40 | Displaying information about the key the user pressed. (Part 4 of 4.)

Software Engineering Observation 14.2

To cause a control to react when a particular key is pressed (such as Enter*), handle a key event and test for the pressed key. To cause a* `Button` *to be clicked when the* Enter *key is pressed on a* `Form`*, set the* `Form`*'s* `AcceptButton` *property.*