CHAPTER 12

Using Controls

In this chapter you will:

- Learn about `Controls`
- Examine the IDE's automatically generated code
- Set a `Control`'s `Font`
- Create a `Form` that contains `LinkLabels`
- Add color to a `Form`
- Add `CheckBox` and `RadioButton` objects to a `Form`
- Add a `PictureBox` to a `Form`
- Add `ListBox`, `ComboBox`, and `CheckedListBox` items to a `Form`
- Add a `MonthCalendar` and `DateTimePicker` to a `Form`
- Work with a `Form`'s layout
- Add a `MenuStrip` to a `Form`
- Learn to use other controls

Throughout this book, you have created both console and GUI applications. Your GUI applications have used only a few `Control`s—`Form`s, `Label`s, `TextBox`es, and `Button`s. **Graphical control elements** are the components through which a user interacts with a GUI program. (Since Chapter 3, you have used the simpler name *controls* for these objects.)

If you have been creating mostly console applications while learning the concepts in this book, you might want to review the chapter "Using GUI Objects and the Visual Studio IDE." That chapter provides instruction in the basic procedures used to create GUI applications and describes Visual Studio Help.

When using programs or visiting Internet sites, you have encountered and used many other interactive controls such as scroll bars, check boxes, and radio buttons. C# has many classes that represent these GUI objects, and the Visual Studio IDE makes it easy to add them to your programs. (Controls are also often called **widgets**.) In this chapter, you will learn to incorporate some of the most common and useful widgets into your programs. Additionally, you will see how these components work in general so you can use other controls that are not covered in this book or that become available to programmers in future releases of C#.

GUI components are referred to as *widgets*, which some sources claim is a combination of the terms *window* and *gadgets*. Originally, *widget* comes from the 1924 play "Beggar on Horseback," by George Kaufman and Marc Connelly. In the play, a young composer gets engaged to the daughter of a rich businessman and foresees spending his life doing pointless work in a bureaucratic big business that manufactures widgets, which represent a useless item whose purpose is never explained.

Understanding `Control`s

When you design a `Form`, you can place `Button`s and other controls on the `Form` surface. In C#, the `Control` class provides the definitions for these GUI objects. `Control` objects such as `Form`s and `Button`s, like all other objects in C#, ultimately derive from the `Object` class. Figure 12-1 shows where the `Control` class fits into the inheritance hierarchy.

```
System.Object
   System.MarshalByRefObject
      System.ComponentModel.Component
         System.Windows.Forms.Control
            26 Derived classes
```

Figure 12-1 The `Control` class inheritance hierarchy

Figure 12-1 shows that all `Control`s are `Object`s, of course. They are also all `MarshalByRefObject`s. (A `MarshalByRefObject` is one you can instantiate on a remote computer so that you can manipulate a reference to the object rather than a local copy of the object.) `Control`s also descend from `Component`. (The `Component` class provides

containment and cleanup for other objects—inheriting from Component allows Controls to be contained in objects such as Forms and provides for disposal of Controls when they are destroyed. The Control class adds visual representation to Components.) The Control class implements very basic functionality required by classes that define the GUI objects the user sees on the screen. This class handles user input through the keyboard, pointing devices, and touch screens as well as message routing and security. It defines the bounds of a Control by determining its position and size.

Table 12-1 shows the 26 direct descendents of Control and some commonly used descendents of those classes. It does not show all the descendents that exist; rather, it shows only the descendents covered previously or in this chapter. For example, the ButtonBase class is the parent of Button, a class you have used throughout this book. In this chapter, you will use two other ButtonBase children—CheckBox and RadioButton. This chapter cannot cover every Control that has been invented; however, after you learn to use some Controls, you will find that others work in much the same way. You also can read more about them in the Visual Studio Help documentation.

Class	Commonly used descendents
Microsoft.WindowsCE.Forms.DocumentList	
System.Windows.Forms.AxHost	
System.Windows.Forms.ButtonBase	Button, CheckBox, RadioButton
System.Windows.Forms.DataGrid	
System.Windows.Forms.DataGridView	
System.Windows.Forms.DateTimePicker	
System.Windows.Forms.GroupBox	
System.Windows.Forms.Integration.ElementHost	
System.Windows.Forms.Label	LinkLabel
System.Windows.Forms.ListControl	ListBox, ComboBox, CheckedListBox
System.Windows.Forms.ListView	
System.Windows.Forms.MdiClient	
System.Windows.Forms.MonthCalendar	
System.Windows.Forms.PictureBox	

Table 12-1 Classes derived from System.Windows.Forms.Control *(continues)*

(continued)

Class	Commonly used descendents
System.Windows.Forms.PrintPreviewControl	
System.Windows.Forms.ProgressBar	
System.Windows.Forms.ScrollableControl	
System.Windows.Forms.ScrollBar	
System.Windows.Forms.Splitter	
System.Windows.Forms.StatusBar	
System.Windows.Forms.TabControl	
System.Windows.Forms.TextBoxBase	TextBox
System.Windows.Forms.ToolBar	
System.Windows.Forms.TrackBar	
System.Windows.Forms.TreeView	
System.Windows.Forms.WebBrowserBase	

Table 12-1 Classes derived from `System.Windows.Forms.Control`

Because `Control`s are all relatives, they share many of the same attributes. Each `Control` has more than 80 `public` properties and 20 `protected` properties. For example, each `Control` has a `Font` and a `ForeColor` that dictate how its text is displayed, and each `Control` has a `Width` and `Height`. Table 12-2 shows just some of the `public` properties associated with `Control`s in general; reading through them will give you an idea of the `Control` attributes that you can change.

Property	Description
`AllowDrop`	Gets or sets a value indicating whether the control can accept data that the user drags onto it
`Anchor`	Gets or sets the edges of the container to which a control is bound and determines how a control is resized with its parent
`BackColor`	Gets or sets the background color for the control
`BackgroundImage`	Gets or sets the background image displayed in the control

Table 12-2 Selected `public Control` properties *(continues)*

(continued)

Property	Description
Bottom	Gets the distance, in pixels, between the bottom edge of the control and the top edge of its container's client area
Bounds	Gets or sets the size and location of the control, including its nonclient elements, in pixels, relative to the parent control
CanFocus	Gets a value indicating whether the control can receive focus
CanSelect	Gets a value indicating whether the control can be selected
Capture	Gets or sets a value indicating whether the control has captured the mouse
Container	Gets the IContainer that contains the Component (inherited from Component)
ContainsFocus	Gets a value indicating whether the control or one of its child controls currently has the input focus
Cursor	Gets or sets the cursor that is displayed when the mouse pointer is over the control
Disposing	Gets a value indicating whether the base Control class is in the process of disposing
Dock	Gets or sets which control borders are docked to their parent control and determines how a control is resized with its parent
Enabled	Gets or sets a value indicating whether the control can respond to user interaction
Focused	Gets a value indicating whether the control has input focus
Font	Gets or sets the font of the text displayed by the control
ForeColor	Gets or sets the foreground color of the control
HasChildren	Gets a value indicating whether the control contains one or more child controls
Height	Gets or sets the height of the control
IsDisposed	Gets a value indicating whether the control has been disposed of
Left	Gets or sets the distance, in pixels, between the left edge of the control and the left edge of its container's client area
Location	Gets or sets the coordinates of the upper-left corner of the control relative to the upper-left corner of its container
Margin	Gets or sets the space between controls

Table 12-2 Selected public Control properties *(continues)*

(continued)

Property	Description
`ModifierKeys`	Gets a value indicating which of the modifier keys (Shift, Ctrl, and Alt) is in a pressed state
`MouseButtons`	Gets a value indicating which of the mouse buttons is in a pressed state
`MousePosition`	Gets the position of the mouse cursor in screen coordinates
`Name`	Gets or sets the name of the control
`Parent`	Gets or sets the parent container of the control
`Right`	Gets the distance, in pixels, between the right edge of the control and the left edge of its container's client area
`Size`	Gets or sets the height and width of the control
`TabIndex`	Gets or sets the tab order of the control within its container
`TabStop`	Gets or sets a value indicating whether the user can give focus to the control using the Tab key
`Text`	Gets or sets the text associated with this control
`Top`	Gets or sets the distance, in pixels, between the top edge of the control and the top edge of its container's client area
`TopLevelControl`	Gets the parent control that is not parented by another Windows `Forms` control; typically, this is the outermost `Form` in which the control is contained
`Visible`	Gets or sets a value indicating whether the control and all its parent controls are displayed
`Width`	Gets or sets the width of the control

Table 12-2 Selected `public Control` properties

The description of each property in Table 12-2 indicates whether the property is read-only; such properties only get values and do not set them.

You have altered `Label`, `TextBox`, and `Button` properties such as `Text` and `Visible` using the Properties window in Visual Studio. All the other `Controls` you learn about in this chapter can be manipulated in the same way.

A project can contain multiple `Forms`, each with its own `Controls`. You will learn how to add more `Forms` to a project in the "You Do It" exercises later in this chapter.

TWO TRUTHS & A LIE

Understanding Controls

1. The `Control` class implements basic functionality required by GUI objects that a user sees on the screen.
2. Most `Controls` have `Font` and `ForeColor` properties.
3. Every `Control` has `Width` and `Height` properties.

The false statement is #2. Every `Control` has `Font` and `ForeColor` properties.

Examining the IDE's Automatically Generated Code

Figure 12-2 shows a `Form` created in the IDE. The following actions have been performed:

1. A new Windows Forms project has been started and given the name `FormWithALabelAndAButton`.
2. A `Label` has been dragged onto `Form1`. Using the Properties window in the IDE, the `Label`'s `Text` property has been changed to *Click the button*, and its `Font` has been changed to Georgia, Bold, and size 16. Its `Name` has not been changed from the default name label1.
3. A `Button` has been dragged onto `Form1`. The `Button`'s `Text` property has been changed to *OK*, and its `Name` has been changed from the default `Name button1` to `okButton`.

Figure 12-2 A `Form` generated by the `FormWithALabelAndAButton` program

As you drag controls in the Form Designer or change properties in the Properties Window, Visual Studio automatically generates code in the file named Form1.Designer.cs. When you open the Form1.Designer.cs file in Visual Studio, you can look for the method named `InitializeComponent()`, and you can see two lines of code generated within it near the bottom as follows:

```
private System.Windows.Forms.Label
label1;
private System.Windows.Forms.Button
okButton;
```

If you were to continue to drag additional components onto the `Form`, more declarations would be generated.

The following lines appear just before the method header for the `InitializeComponent()` method:

```
#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>
```

Within the `InitializeComponent()` method, you can see automatically-generated statements that set the properties of the label, the button, and the form itself as partially shown in Figure 12-3. In the code in the figure, every instance of `this` means "this `Form`".

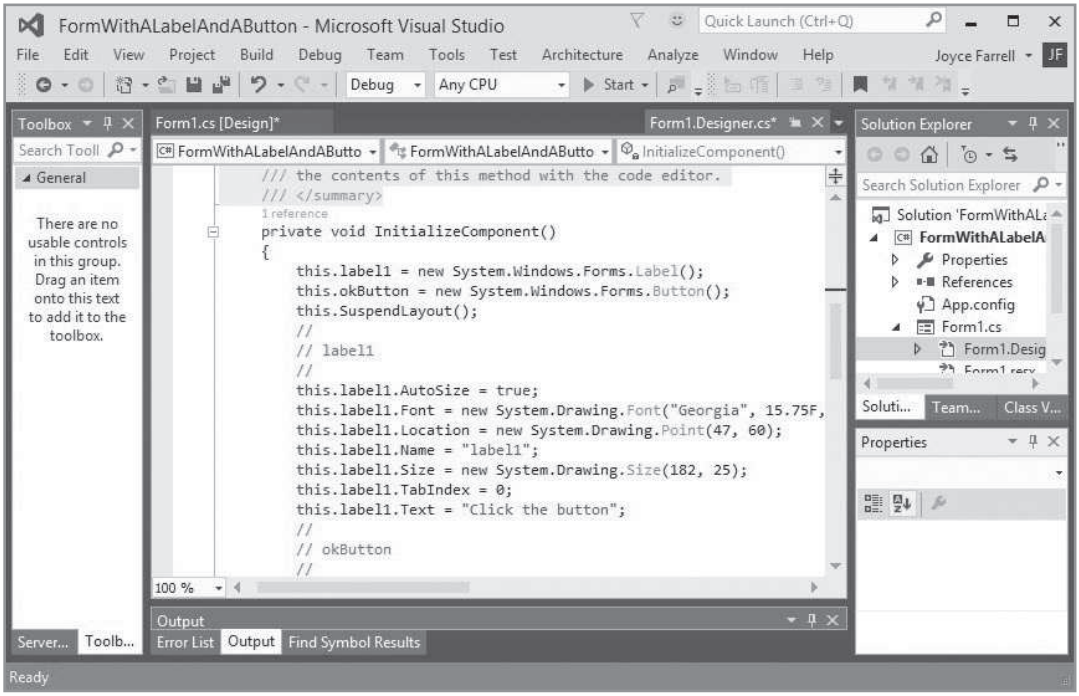

Figure 12-3 Some of the code in the `InitializeComponent()` method for `FormWithALabelAndAButton`

So much code is automatically generated by Visual Studio that it can be hard to find what you want. To locate a line of code, click Edit on the menu bar in the IDE, click Find and Replace, click Quick Find, type a key phrase to search for. Make sure that the drop down box correctly identifies the area in which you want to search—for example, the current selection or the current document.

Do not be intimidated by the amount of code automatically generated by the IDE. Based on what you have learned so far in this book, you can easily make sense of most of it.

- After the `InitializeComponent()` method header and opening brace, the next two statements call the constructor for each control that the programmer dragged onto the `Form`:

```
this.label1 = new System.Windows.Forms.Label();
this.okButton = new System.Windows.Forms.Button();
```

  These statements create the actual objects.

- The next statement is a method call as follows:

```
this.SuspendLayout();
```

  `SuspendLayout()` is a method that prevents conflicts when you are placing `Control`s on a form. Its counterparts, `ResumeLayout()` and `PerformLayout()`, appear at the bottom of the method. If you remove these method calls from small applications, you won't notice the difference. However, in large applications, suspending the layout logic while you adjust the appearance of components improves performance.

- Comments that start with forward slashes serve to separate the `label1` code from other code in the method. Following the `label1` comment lines, seven statements set properties of the `Label` as follows:

```
this.label1.AutoSize = true;
this.label1.Font = new System.Drawing.Font("Georgia",
   15.75F, System.Drawing.FontStyle.Bold,
   System.Drawing.GraphicsUnit.Point, ((byte)(0)));
this.label1.Location = new System.Drawing.Point(47, 60);
this.label1.Name = "label1";
this.label1.Size = new System.Drawing.Size(182, 25);
this.label1.TabIndex = 0;
this.label1.Text = "Click the button";
```

  You can see that the `Font`, `Location`, `Name`, `Size`, and `Text` have been assigned values based on the programmer's choices in the IDE. The `TabIndex` for `label1` is 0 by default; `TabIndex` values determine the order in which `Control`s receive focus when the user presses the Tab key. This property is typically more useful for selectable items like `Button`s.

- If you return to the visual designer and make changes to the form or its components, for example by relocating the label, when you next view the code, it will have been updated accordingly.

- Although not completely visible in Figure 12-3, the next set of statements defines the properties of the `Button` on the `Form`. The `TabIndex` for the `Button` is set to 1 because it was dragged onto the `Form` after the `Label`. Additional `Control`s would receive consecutive `TabIndex` values.

```
this.okButton.Location = new System.Drawing.Point(98, 117);
this.okButton.Name = "okButton";
this.okButton.Size = new System.Drawing.Size(75, 23);
this.okButton.TabIndex = 1;
this.okButton.Text = "OK";
this.okButton.UseVisualStyleBackColor = true;
```

- The `InitializeComponent()` method ends with statements that set the properties of the `Form`, such as its drawing size and text, and that add the `Label` and `Button` to the `Form`:

```
// Form1
//
this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);
this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
this.ClientSize = new System.Drawing.Size(284, 262);
this.Controls.Add(this.okButton);
this.Controls.Add(this.label1);
this.Name = "Form1";
this.Text = "Form1";
this.ResumeLayout(false);
this.PerformLayout();
```

Although the property settings for the `Label` and `Button` include identifiers for the `Controls` (for example, `this.label1.Name` or `this.okButton.Text`), the property settings for the `Form` itself use only the reference `this`. That's because the statements are part of the `Form1` class.

In this chapter, you will learn about several additional `Controls`. When designing a `Form`, you should use the drag-and-drop design features in the IDE to place components and use the Properties window in the IDE to set properties instead of typing statements in the code editor. However, this chapter also teaches you about the code behind these actions so you can troubleshoot problems in projects and write usable statements when necessary.

Watch the video *Examining the IDE Code*.

TWO TRUTHS & A LIE

Examining the IDE's Automatically Generated Code

1. By using the Form Designer and the Properties window, you save time and eliminate many chances for error.
2. When you use the Form Designer to drag a `Label` onto a `Form`, no constructor call is needed for the `Label`.
3. You can use the Properties Window to set properties for a `Label` such as `Font`, `Location`, `Size`, and `Text`.

The false statement is #2. When you use the Form Designer to drag a `Label` onto a `Form`, you do not have to write a constructor call, but one is generated for you.

Setting a Control's Font

You use the Font class to change the appearance of printed text on your Forms. When designing a Label, Button, or other Control on a Form, it is easiest to select a Font from the Properties list. After you place a Control on a Form in the IDE, you can select the ellipsis (three dots) that follows the current Font property name in the Properties list. (See Figure 12-4.) This selection displays a Font window in which you can choose a Font name, size, style, and other effects. (See Figure 12-5.)

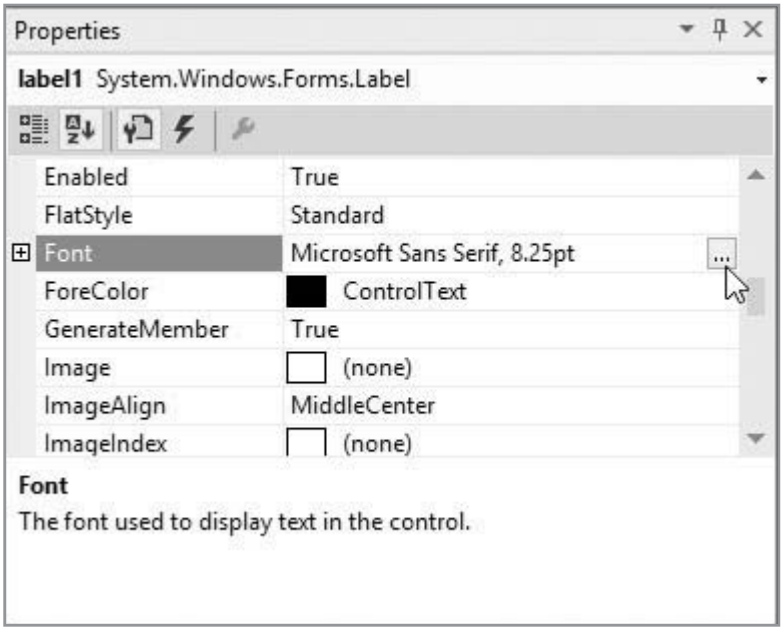

Figure 12-4 Clicking the ellipsis following the Font property

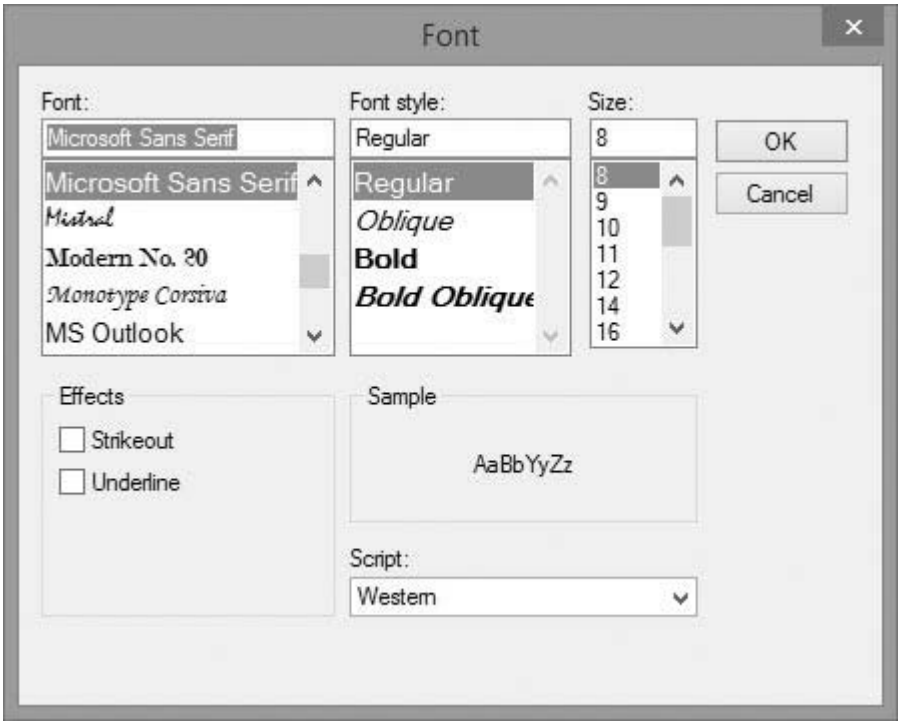

Figure 12-5 The Font window

However, if you wanted to change a `Font` later in a program—for example, after a user clicks a button—you might want to create your own instance of the `Font` class. As another example, suppose you want to create multiple controls that use the same `Font`. In that case, it makes sense to declare a named instance of the `Font` class. For example, you can declare the following `Font`:

```
System.Drawing.Font bigFont = new
   System.Drawing.Font("Courier New", 16.5f);
```

This version of the `Font` constructor requires two arguments—a `string` and a `float`. The `string` you pass to the `Font` constructor is the name of the font. If you use a font name that does not exist in your system, the font defaults to Microsoft Sans Serif. The second value is a `float` that represents the font size. Notice that you must use an *F* (or an *f*) following the `Font` size value constant when it contains a decimal point to ensure that the constant will be recognized as a `float` and not a `double`. (If you use an `int` as the font size, you do not need the *f* because the `int` will automatically be cast to a `float`.) An alternative would be to instantiate a `float` constant or variable and use its name as an argument to the `Font` constructor.

In Chapter 2, you learned to use an *f* following a floating-point constant to indicate the `float` type. Recall that a numeric constant with a decimal point is a `double` by default.

After a `Font` object named `bigFont` is instantiated, you can code statements similar to the following:

```
this.label1.Font = bigFont;
this.okButton.Font = bigFont;
```

If you want to change the properties of several objects at once in the IDE, you can drag your mouse around them to create a temporary group and then change the property for all of them with one entry in the Properties list.

The `Font` class includes a number of overloaded constructors. For example, you also can create a `Font` using three arguments, adding a `FontStyle`, as in the following declaration:

```
Font aFancyFont = new Font("Arial", 24, FontStyle.Italic);
```

Table 12-3 lists the available `FontStyle`s. You can combine multiple styles using the pipe (|), which is also called the *logical OR operator* or the *bitwise OR operator*. (You first learned about this operator in Chapter 4.) The word *or* indicates that bits are turned on in the result when either of the operands contains an on-bit in a given position. For example, the following code creates a `Font` that is bold and underlined because the bits that indicate bold and underlined are both turned on in the result:

```
Font boldAndUnderlined = new Font("Helvetica",
   10, FontStyle.Bold | FontStyle.Underline);
```

Member Name	Description
`Bold`	Bold text
`Italic`	Italic text
`Regular`	Normal text
`Strikeout`	Text with a line through the middle
`Underline`	Underlined text

Table 12-3 `FontStyle` enumeration

Instead of instantiating a named `Font` object, you can create and assign an anonymous `Font` in one step. In other words, an identifier is not provided for the `Font`, as in this example:

```
this.label1.Font = new
   System.Drawing.Font("Courier New", 12.5F);
```


If you don't provide an identifier for a `Font`, you can't reuse it. You will have to create it again to use it with additional `Controls`.

TWO TRUTHS & A LIE

Setting a `Control`'s `Font`

1. You use the `Font` class to change the appearance of printed text on `Controls` in your `Forms`.
2. When designing a `Control` on a `Form`, you must select a `Font` from the Properties list in the IDE.
3. The `Font` class includes several overloaded constructors.

The false statement is #2. When designing a `Control` on a `Form`, a default font is selected. If you want to change the font, it is easiest to select a font from the Properties list, but you also can create your own instance of the `Font` class.

Using a `LinkLabel`

A **link label** is a control with a label that provides the user a way to link to other sources, such as Web pages or files. The C# class that creates a link label is `LinkLabel`; it is a child of `Label`. Table 12-4 summarizes the properties and lists the default event method for a `LinkLabel`. The **default event** for a `Control` is:

- The method whose shell is automatically created when you double-click the `Control` while designing a project in the IDE
- The method that you are most likely to alter when you use the `Control`
- The event that users most likely expect to generate when they encounter the `Control` in a working application

With many `Controls`, including a `LinkLabel`, a mouse click by the user triggers the default event. When designing a program, you can double-click a `Control` in the IDE to generate the default method shell, and then write any necessary statements within the shell.

Property or Method	Description
ActiveLinkColor	The color of the link when it is clicked
LinkColor	The original color of links before they have been visited; usually blue by default
LinkVisited	If true, the link's color is changed to the VisitedLinkColor
VisitedLinkColor	The color of a link after it has been visited; usually purple by default
LinkClicked()	Default event that is generated when the link is clicked by the user

Table 12-4 Commonly used LinkLabel properties and default event

The default event for many Controls, such as Buttons and LinkLabels, occurs when the user clicks the Control. However, the default event for a Form is the Load() method. In other words, if you double-click a Form in the IDE, you generate this method. In it, you can place statements that execute as soon as a Form is loaded.

When you create a LinkLabel, it appears as underlined text. The text is blue by default, but you can change the color in the LinkLabel Properties list in the IDE. When you pass the mouse pointer over a LinkLabel, the pointer changes to a hand; you have seen similar behavior while using hyperlinks in Web pages. When a user clicks a LinkLabel, it generates a click event, just as clicking a Button does. When a click event is fired from a LinkLabel, a LinkClicked() method is executed, similar to how clicking a Button can execute a Click() method.

You can create a program so that a user generates an event by clicking many types of objects. For example, for a Label named label1, you could write statements in a label1_Click() method. However, users do not usually expect to click Labels; they do expect to click LinkLabels.

When you double-click a Label (as well as most other controls), the automatically generated method name ends with Click(), but when you double-click a LinkLabel, the corresponding method ends with Clicked().

Figure 12-6 shows a Form onto which two LinkLabels have been dragged from the Toolbox in the IDE. The default Frame size has been reduced, and the Text properties of the LinkLabels have been changed to *Course Technology Website* and *Read Our Policy*.

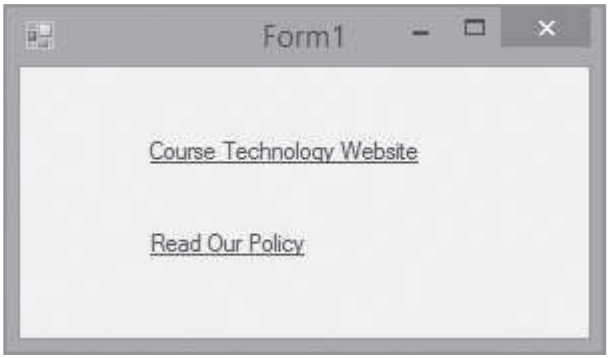

Figure 12-6 A `Form` with two `LinkLabels`

If you double-click a `LinkLabel` in the IDE, a method shell is created for you in the format `xxx_LinkClicked()`, where `xxx` is the value of the `Name` property assigned to the `LinkLabel`. (This corresponds to what happens when you double-click a `Button` in the IDE.) For example, Figure 12-7 shows the two generated methods for the `Form` in Figure 12-6 when the default `LinkLabel` identifiers `linkLabel1` and `linkLabel2` are used. In Figure 12-7, all the code was automatically generated except for the two shaded statements. The programmer added those lines to indicate which actions should occur when a user clicks the corresponding `LinkLabel` in a running application.

```
public partial class Form1 : Form
{
   public Form1()
   {
      InitializeComponent();
   }
   private void linkLabel1_LinkClicked(object sender,
      LinkLabelLinkClickedEventArgs e)
   {
      System.Diagnostics.Process.Start("IExplore",
         "http://www.course.com");
   }
   private void linkLabel2_LinkClicked(object sender,
      LinkLabelLinkClickedEventArgs e)
   {
      System.Diagnostics.Process.Start
         (@"C:\C#\Chapter.12\Policy.txt");
   }
}
```

Figure 12-7 Two `LinkClicked()` methods

If you add `using System.Diagnostics;` at the top of your file, you can eliminate the references in Figure 12-7 and refer to the process simply as `Process.Start`.

In each of the LinkClicked() methods in Figure 12-7, the programmer has added a call to System.Diagnostics.Process.Start(). This method allows you to run other programs within an application. The Start() method has two overloaded versions:

- When you use one string argument, you provide the name of the file to be opened.
- When you use two arguments, you provide the name of an application and its needed arguments.

In the linkLabel1_LinkClicked() method, the two arguments open Internet Explorer ("IExplore") and pass it the address of the Course Technology Web site. If an Internet connection is active, control transfers to the Web site.

In the linkLabel2_LinkClicked() method, only one argument is provided. It opens a file stored on the local disk. The default application opens based on the file's application type, which is determined by its file extension. For example, Notepad is the default application for a file with a .txt extension. Alternatively, you could code the following, which explicitly names Notepad as the application:

```
System.Diagnostics.Process.Start("Notepad",
   @"C:\C#\Chapter.12\Policy.txt");
```

In the linkLabel2_LinkClicked() method, an at sign (@) appears in front of the filename to be opened. This symbol indicates that all characters in the string should be interpreted literally. Therefore, the backslashes in the path are not interpreted as escape sequence characters.

The LinkVisited property can be set to true when you determine that a user has clicked a link, as shown in the shaded statement in Figure 12-8. This setting indicates that the link should be displayed in a different color so the user can see the link has been visited. By default, the visited link color is purple, but you can change this setting in the Properties list for the LinkLabel.

```
private void linkLabel1_LinkClicked(object sender,
   LinkLabelLinkClickedEventArgs e)
{
   System.Diagnostics.Process.Start("IExplore",
      "http://www.course.com");
   linkLabel1.LinkVisited = true;
}
```

Figure 12-8 Setting the LinkVisited property

TWO TRUTHS & A LIE

Using a LinkLabel

1. A `LinkLabel` is a child class of `Label`, and it provides the additional capability to link the user to other sources.
2. The default event for a `Control` is the method whose shell is automatically created when you double-click the `Control` while designing a project in the IDE. Users most likely expect to generate this event when they encounter the `Control` in a working application.
3. When you create a `LinkLabel`, it appears as italicized underlined text, and when you pass the mouse pointer over a `LinkLabel`, the pointer changes to an hourglass.

The false statement is #3. When you create a `LinkLabel`, it appears as underlined text, and when you pass the mouse pointer over a `LinkLabel`, the pointer changes to a hand.

Adding Color to a Form

The `Color` class contains a wide variety of predefined `Color`s that you can use with your `Control`s (see Table 12-5).

C# also allows you to create custom colors. If no color in Table 12-5 suits your needs, search for *custom color* in Visual Studio Help to obtain more information.

AliceBlue	DeepPink	Lime	RosyBrown
AntiqueWhite	DeepSkyBlue	LimeGreen	RoyalBlue
Aqua	DimGray	Linen	SaddleBrown
Aquamarine	DodgerBlue	Magenta	Salmon
Azure	Firebrick	Maroon	SandyBrown
Beige	FloralWhite	MediumAquamarine	SeaGreen
Bisque	ForestGreen	MediumBlue	SeaShell
Black	Fuchsia	MediumOrchid	Sienna

Table 12-5 `Color` properties *(continues)*

(continued)

BlanchedAlmond	Gainsboro	MediumPurple	Silver
Blue	GhostWhite	MediumSeaGreen	SkyBlue
BlueViolet	Gold	MediumSlateBlue	SlateBlue
Brown	Goldenrod	MediumSpringGreen	SlateGray
BurlyWood	Gray	MediumTurquoise	Snow
CadetBlue	Green	MediumVioletRed	SpringGreen
Chartreuse	GreenYellow	MidnightBlue	SteelBlue
Chocolate	Honeydew	MintCream	Tan
Coral	HotPink	MistyRose	Teal
CornflowerBlue	IndianRed	Moccasin	Thistle
Cornsilk	Indigo	NavajoWhite	Tomato
Crimson	Ivory	Navy	Transparent
Cyan	Khaki	OldLace	Turquoise
DarkBlue	Lavender	Olive	Violet
DarkCyan	LavenderBlush	OliveDrab	Wheat
DarkGoldenrod	LawnGreen	Orange	White
DarkGray	LemonChiffon	OrangeRed	WhiteSmoke
DarkGreen	LightBlue	Orchid	Yellow
DarkKhaki	LightCoral	PaleGoldenrod	YellowGreen
DarkMagenta	LightCyan	PaleGreen	
DarkOliveGreen	LightGoldenrodYellow	PaleTurquoise	
DarkOrange	LightGray	PaleVioletRed	
DarkOrchid	LightGreen	PapayaWhip	
DarkRed	LightPink	PeachPuff	
DarkSalmon	LightSalmon	Peru	
DarkSeaGreen	LightSeaGreen	Pink	
DarkSlateBlue	LightSkyBlue	Plum	
DarkSlateGray	LightSlateGray	PowderBlue	
DarkTurquoise	LightSteelBlue	Purple	
DarkViolet	LightYellow	Red	

Table 12-5 `Color` properties

When you are designing a `Form`, you can choose colors from a list next to the `BackColor` and `ForeColor` properties in the IDE's Properties list. The statements created will be similar to the following:

```
this.label1.BackColor = System.Drawing.Color.Blue;
this.label1.ForeColor = System.Drawing.Color.Gold;
```


If you add `using System.Drawing;` at the top of your file, you can eliminate the references in the preceding lines and refer to the colors simply as `Color.Blue` and `Color.Gold`.

For professional-looking results when you prepare a resume or most other business documents, experts recommend that you use only one or two fonts and colors, even though your word-processing program allows many such selections. The same is true when you design GUI applications. Although many fonts and colors are available, you probably should stick with just a few choices in a single project.

TWO TRUTHS & A LIE

Adding Color to a Form

1. Because the choice of colors in C# is limited, you are required to create custom colors for many GUI applications.
2. When you are designing a `Form`, color choices appear in a list next to the `BackColor` and `ForeColor` properties in the IDE's Properties list.
3. The complete name of the color pink in C# is `System.Drawing.Color.Pink`.

The false statement is #1. The `Color` class contains a wide variety of predefined `Color`s that you can use with your `Control`s.

You Do It

Adding `Label`s to a `Form` and Changing Their Properties

In the next steps, you begin to create an application for Bailey's Bed and Breakfast. The main `Form` allows the user to select one of two suites and discover the amenities and price associated with each choice. You will start by placing two `Label`s on a `Form` and setting several of their properties.

(continues)

(continued)

The screen images in the next steps represent a typical Visual Studio environment. Based on the version of Visual Studio you are using and the options selected during your installation, your screen might look different.

1. Open Microsoft Visual Studio. Select **New Project** and **Windows Forms Application**. Near the bottom of the New Project window, click in the **Name** text box, and replace the default name with **BedAndBreakfast**. Make sure that the Location field contains the folder where you want to store the project. See Figure 12-9.

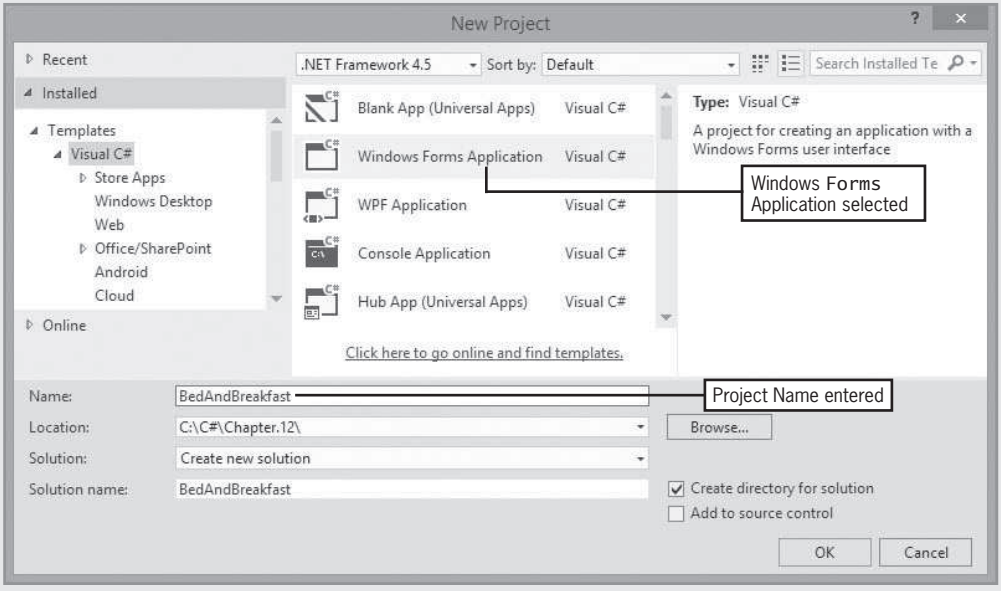

Figure 12-9 The New Project window for the `BedAndBreakfast` application

2. Click **OK**. The design screen opens. The blank `Form` in the center of the screen has an empty title bar. Click the `Form`. The lower-right corner of the screen contains a Properties window that lists the `Form`'s properties. (If you do not see the Properties window, you can click **View** on the menu bar, and then click **Properties Window**.) In the Properties list, click the **`Name`** property and change the `Name` of the `Form` to **BaileysForm**. Click the **`Text`** property and change it to **Bailey's Bed and Breakfast**.

(continues)

(continued)

3. From the Toolbox at the left of the screen, drag a `Label` onto the `Form`. Change the `Name` of `label1` to **`welcomeLabel`** and change the `Text` property to **Welcome to Bailey's**. Drag and resize the `Label` so it is close to the position of the `Label` in Figure 12-10. (If you prefer to set the `Label`'s `Location` property manually in the Properties list, the `Location` should be **60, 30**.)

If you do not see the Toolbox, click the **Toolbox tab** at the left side of the screen and pin it to the screen by clicking the pushpin. Alternatively, you can select **View** from the menu bar, and then click **Toolbox**.

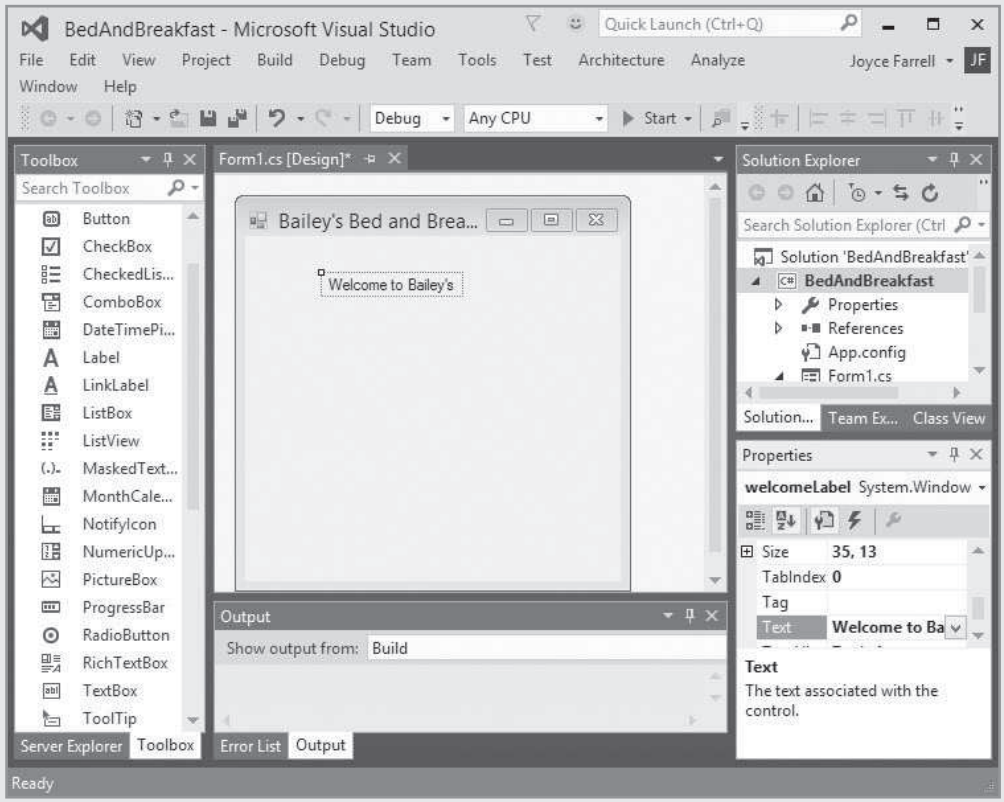

Figure 12-10 A `Label` placed on the `Form` in the `BedAndBreakfast` project

(continues)

(continued)

4. Locate the `Font` property in the Properties list. Currently, it lists the default font: Microsoft Sans Serif, 8.25 pt. Notice the ellipsis (three dots) at the right of the `Font` property name. (You might have to click in the Property to see the button.) Click the ellipsis to display the `Font` dialog box. Make selections to change the font to **Microsoft Sans Serif**, **18 point**, and **Bold**. Click **OK**. When you enlarge the `Font` for the `Label`, it is too close to the right edge of the `Form`. Drag the `Label` to change its `Location` property to approximately **20, 30**, or type the new `Location` value in the Properties window.

5. Drag a second `Label` onto the `Form` beneath the first one, and then set its `Name` property to `rateLabel` and its `Text` property to **Check our rates**. Change its `Location` to approximately **80, 80** and its `Font` to **Microsoft Sans Serif**, **12 point**, **Regular**.

6. Save the project, click **Debug** on the menu bar, and click **Start Without Debugging**, or use the shortcut keys listed in the menu. The `Form` appears, as shown in Figure 12-11.

Figure 12-11 The `BedAndBreakfast Form` with two `Labels`

7. Dismiss the `Form` by clicking the **Close** button in its upper-right corner.

(continues)

(continued)

Examining the Code Generated by the IDE

In the next steps, you examine the code generated by the IDE for the following reasons:

- To gain an understanding of the types of statements created by the IDE.
- To lose any intimidation you might have about the code that is generated. You will recognize many of the C# statements from what you have already learned in this book.

1. In the Solution Explorer at the right side of the screen, double-click **Form1.Designer.cs**. Scroll down until you can view the statements similar to those shown in Figure 12-12. (You can drag the bottom and side borders of the code window to expose more of the code, or you can scroll to see all of it.) You should be able to view statements that assign values to the properties of the components that you dragged into the `Form`.

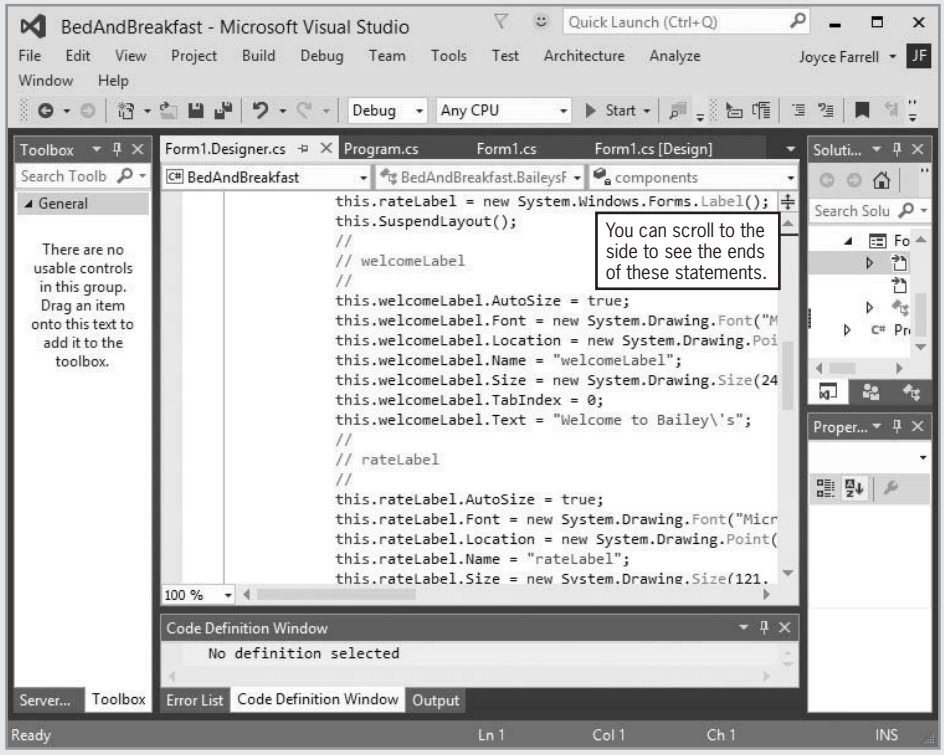

Figure 12-12 Part of the Form1.Designer.cs code

(continues)

(continued)

2. Next, change the `BackColor` property of the Bailey's Bed and Breakfast `Form`. Click the **Form1.cs[Design]** tab and click the **`Form`**, or click the list box of components at the top of the Properties window and select **BaileysForm**. In the Properties list, click the **`BackColor`** property and click its down arrow to see its list of choices. Choose the **Custom** tab and select **Yellow** in the third row of available colors. Click the **`Form`**, notice the color change, and then view the code in the Form1.Designer.cs file. Locate the statement that changes the `BackColor` of the `Form` to `Yellow`. As you continue to design `Form`s, periodically check the code to confirm your changes and better learn C#.
3. Save the project.
4. If you want to take a break at this point, close Visual Studio. You return to this project in the "You Do It" section at the end of this chapter.

Using CheckBox and RadioButton Objects

A **checkbox** is a control that is a small rectangle that indicates whether a user has chosen an option. The C# class that allows you to create a checkbox is `CheckBox`. When a form contains multiple checkboxes, the user can select any number of them. When options are grouped so that only one can be checked at a time and selecting one deselects the others, they are called **radio buttons**. (You might also hear the term *option buttons*.) The C# class that creates a radio button is `RadioButton`. Like `Button`, both `CheckBox` and `RadioButton` descend from `ButtonBase`.

Table 12-6 contains commonly used `CheckBox` and `RadioButton` properties and the default event for which a method shell is generated when you double-click a `CheckBox` or `RadioButton` in the IDE.

Property or Method	Description
`Checked`	Indicates whether the `CheckBox` or `RadioButton` is checked
`Text`	The text displayed to the right of the `CheckBox` or `RadioButton`
`CheckedChanged()`	Default event that is generated when the `Checked` property changes

Table 12-6 Commonly used `CheckBox` and `RadioButton` properties and default event

If you precede a letter with an ampersand (&) in the `Text` property value of a `ButtonBase` object, that letter acts as an access key. An **access key** provides a shortcut way to make a selection using the keyboard. For example, if a `Button`'s text is defined as `&Press`, then typing Alt+P has the same effect as clicking the `Button`. Access keys are also called *hot keys*.

You can place multiple groups of `RadioButton`s on a `Form` by using a `GroupBox` or `Panel`. You will learn more about `GroupBox`es and `Panel`s later in this chapter.

Figure 12-13 shows an example of a `Form` with which a user can select pizza otions. It contains several `Label`s, four `CheckBox` objects, and three `RadioButton` objects. It makes sense for the pizza topping choices to be displayed using `CheckBox`es because a user might select multiple toppings. However, options for delivery, pick-up, and dining in the restaurant are mutually exclusive, so they are presented using `RadioButton` objects.

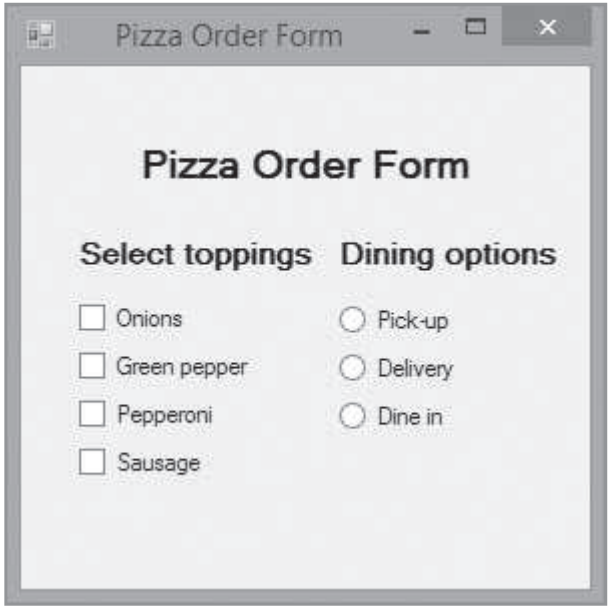

Figure 12-13 A `Form` with `Label`s, `CheckBox`es, and `RadioButton`s

When you add `CheckBox` and `RadioButton` objects to a form, they automatically are named using the same conventions you have seen with `Button`s and `Label`s. That is, the first `CheckBox` without an explicitly assigned name is `checkBox1` by default, the second is named `checkBox2`, and so on. Using the Properties list, you can assign more meaningful names such as `sausageCheckBox` and `pepperoniCheckBox`. Naming objects appropriately makes your code more understandable to others, and makes your programming job easier.

Both `CheckBox` and `RadioButton` objects have a `Checked` property whose value is `true` or `false`. For example, if you create a `CheckBox` named `sausageCheckBox` and you want to add $1.00 to a `pizzaPrice` value when the user checks the box, you can write the following:

```
if(sausageCheckBox.Checked)
   pizzaPrice = pizzaPrice + 1.00;
```

The Checked property is a read/write property. That is, you can assign a value to it as well as access its value.

The default method that executes when a user clicks either a CheckBox or RadioButton is xxx_CheckedChanged(), where xxx represents the name of the invoking object. For example, suppose that the total price of a pizza should be altered based on a user's CheckBox selections. In this example, the base price for a pizza is $12.00, and $1.25 is added for each selected topping. You can declare constants for the BASE_PRICE and TOPPING_PRICE of a pizza and declare a variable that is initialized to the pizza base price as follows:

```
private const double BASE_PRICE = 12.00;
private const double TOPPING_PRICE = 1.25;
private double price = BASE_PRICE;
```

These declarations typically are placed in a Form's .cs file in the Form1 class, above both the constructor and other methods. Figure 12-14 shows the code you would add to the Form.cs file for the application. The sausageCheckBox_CheckedChanged() method changes the pizza price. The method shell was created by double-clicking the sausageCheckBox on the form; the statements within the method were written by a programmer. If a change occurs because the sausageCheckBox was checked, then the TOPPING_PRICE is added to the price. If the change to the checkBox was to uncheck it, the TOPPING_PRICE is subtracted from the price. Either way, the Text property of a Label named outputLabel is changed to reflect the new price.

```
private void sausageCheckBox_CheckedChanged(object sender, EventArgs e)
{
   if (sausageCheckBox.Checked)
      price += TOPPING_PRICE;
   else
      price -= TOPPING_PRICE;
   outputLabel.Text = "Total is " + price.ToString("C");
}
```

Figure 12-14 The sausageCheckBox_CheckedChanged() method

In a similar fashion, you can add appropriate code for RadioButton objects. For example, assume that a $2.00 delivery charge is in effect, but there is no extra charge for customers who pick up a pizza or dine in. The code for deliverRadioButton_CheckedChanged() appears in Figure 12-15. When the user selects the deliverRadioButton, $2.00 is added to the total. When the user selects either of the other RadioButtons, the deliverRadioButton becomes unchecked and the $2.00 charge is removed from the total. Figure 12-16 shows a typical execution of the PizzaOrder program after it is complete.

```
private void deliverRadioButton_CheckedChanged(object sender, EventArgs e)
{
   const double DELIVERY_CHARGE = 2.00;
   if (deliverRadioButton.Checked)
      price += DELIVERY_CHARGE;
   else
      price -= DELIVERY_CHARGE;
   outputLabel.Text = "Total is " + price.ToString("C");
}
```

Figure 12-15 The `deliverRadioButton_CheckedChanged()` method

The entire pizza-order application can be found in the downloadable student files.

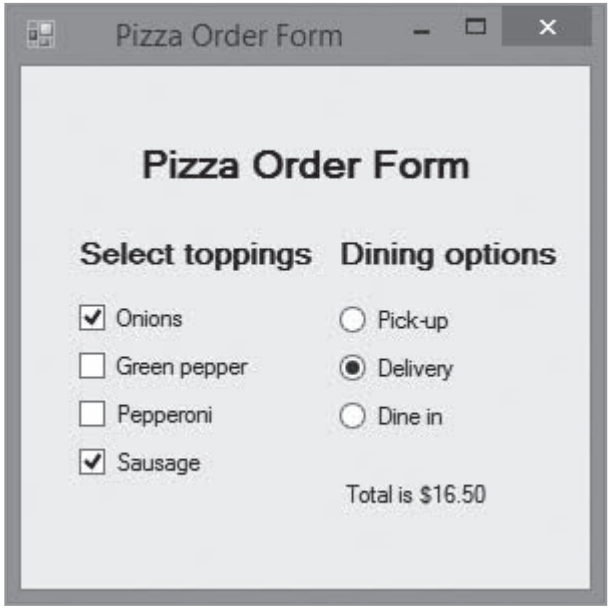

Figure 12-16 Typical execution of the `PizzaOrder` program

When an application starts, sometimes you want a specific `CheckBox` or `RadioButton` to be selected by default. If so, you can set the `Control`'s `Checked` property to `true` in the Properties list in the IDE.

Watch the video *CheckBoxes and RadioButtons*.

TWO TRUTHS & A LIE

Using CheckBox and RadioButton Objects

1. CheckBox objects are GUI widgets that the user can click to select or deselect an option; when a Form contains multiple CheckBoxes, any number of them can be checked or unchecked at the same time.
2. RadioButtons are similar to CheckBoxes, except that when they are placed on a Form, only one RadioButton can be selected at a time—selecting any RadioButton automatically deselects the others.
3. The default event for a CheckBox is CheckBoxChanged(), and the default event for a RadioButton is RadioButtonChanged().

The false statement is #3. The default event for both CheckBox and RadioButton objects is CheckedChanged().

Adding a PictureBox to a Form

A **picture box** is a GUI element you use to display graphics. In C#, a PictureBox object can display graphics from a bitmap, icon, JPEG, GIF, or other image file type. Just as with a Button or a Label, you can easily drag a PictureBox Control onto a Form in the Visual Studio IDE. Table 12-7 shows the common properties and default event for a PictureBox.

Property or Method	Description
Image	Sets the image that appears in the PictureBox
SizeMode	Controls the size and position of the image in the PictureBox; values are Normal, StretchImage (which resizes the image to fit the PictureBox), AutoSize (which resizes the PictureBox to fit the image), and CenterImage (which centers the image in the PictureBox)
Click()	Default event that is generated when the user clicks the PictureBox

Table 12-7 Commonly used PictureBox properties and default event

Figure 12-17 shows a new project in the IDE. The following tasks have been completed:

- A project was started.
- The Form Text property was changed to *Save Money*.

- The `Form BackColor` property was changed to White.
- A `Label` was dragged onto the `Form`, and its `Text` and `Font` were changed.
- A `PictureBox` was dragged onto the `Form`.

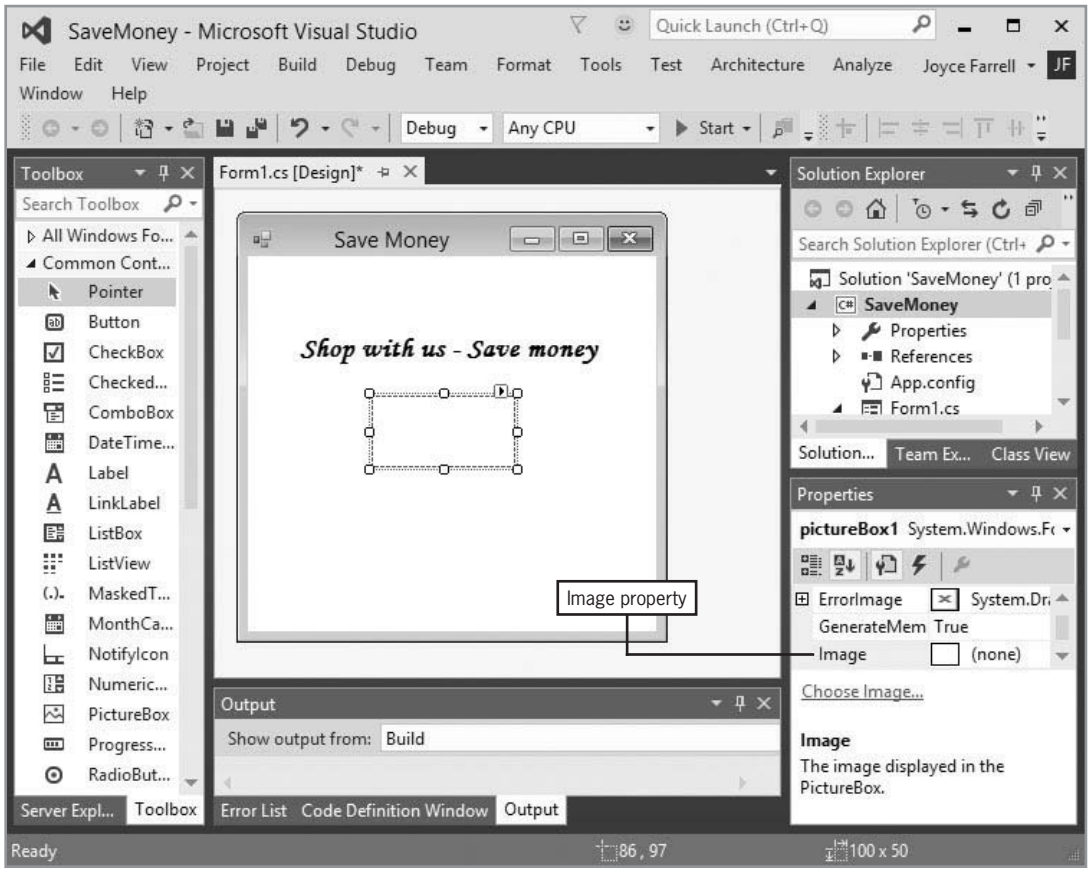

Figure 12-17 The IDE with a `Form` that contains a `PictureBox`

In Figure 12-17, in the Properties list at the right of the screen, the `Image` property is set to (*none)*. If you click the value, a button with an ellipsis appears. If you click it, a Select Resource window appears, as shown on the left in Figure 12-18. When you click the Import button, you can browse for stored images. When you select one, you see a preview in the Select Resource window, as shown on the right in Figure 12-18.

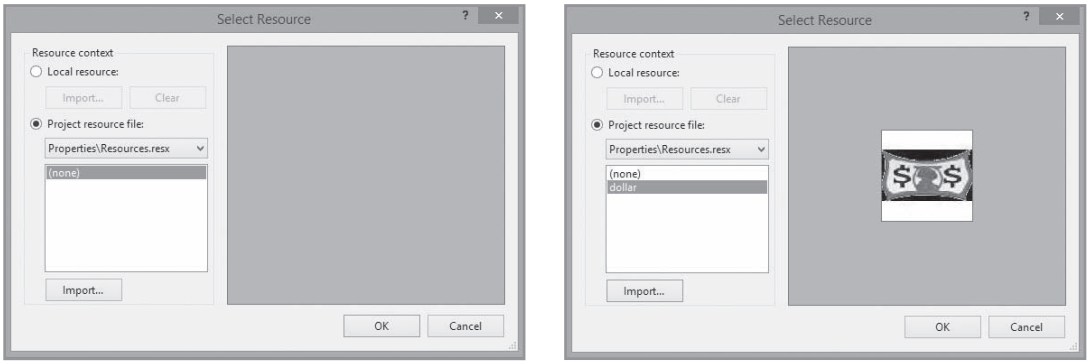

Figure 12-18 The Select Resource window before and after an image is selected

After you click OK, the image appears in the `PictureBox`, as in Figure 12-19. (You can resize the `PictureBox` in the IDE so the image displays completely.)

If you examine the generated code, you can find the statements that instantiate a `PictureBox` (named `pictureBox1` by default) and statements that set its properties, such as `Size` and `Location`.

Figure 12-19 The `SaveMoney Frame` with an inserted image

TWO TRUTHS & A LIE

Adding a PictureBox to a Form

1. A PictureBox is a Control in which you can display graphics from a bitmap, icon, JPEG, GIF, or other image file type.
2. The default event for a PictureBox is LoadImage().
3. The Image property of a PictureBox holds the name of a file where a picture is stored.

The false statement is #2. The default event for a PictureBox is Click().

Adding ListBox, CheckedListBox, and ComboBox Controls to a Form

ListBox, CheckedListBox, and ComboBox objects all allow users to select choices from a list. The three classes descend from ListControl. Of course, they are also Controls and so inherit properties such as Text and BackColor from the Control class. Other properties are more specific to list-type objects. Table 12-8 describes some commonly used ListBox properties.

Property or Method	Description
Items	The collection of items in the ListBox; frequently, these are strings, but they can also be other types of objects
MultiColumn	Indicates whether display can be in multiple columns
SelectedIndex	Returns the index of the selected item. If no item has been selected, the value is –1. Otherwise, it is a value from 0 through *n* – 1, where *n* is the number of items in the ListBox.
SelectedIndices	Returns a collection of all the selected indices (when SelectionMode is more than One)
SelectedItem	Returns a reference to the selected item
SelectedItems	Returns a collection of the selected items (when SelectionMode is more than One)
SelectionMode	Determines how many items can be selected (see Table 12-9)
Sorted	Sorts the items when set to true
SelectedIndexChanged()	Default event that is generated when the selected index changes

Table 12-8 Commonly used ListBox properties and default event

A **list box** displays a list of items from which the user can select by clicking. Figure 12-20 shows a typical `ListBox` on a `Form`. After you drag a `ListBox` onto a `Form`, you can select its `Items` property and type a list into a String Collection Editor, as shown on the left in Figure 12-20.

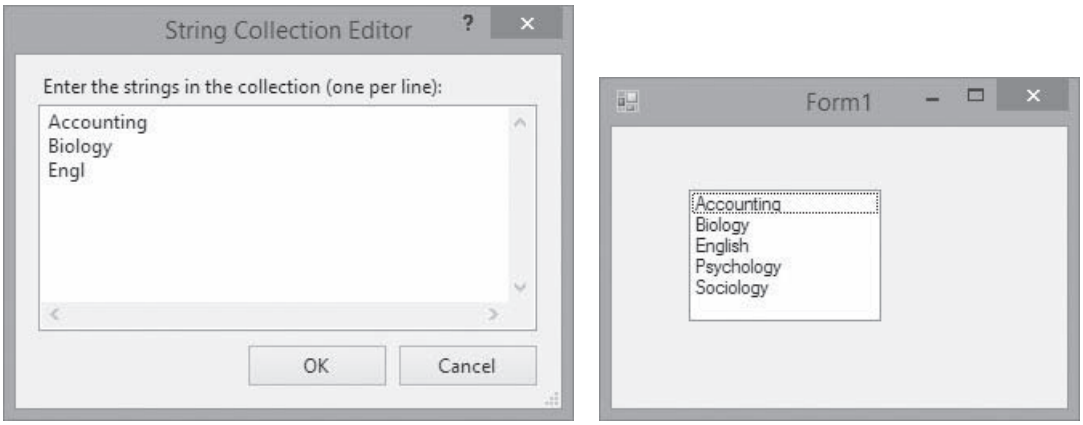

Figure 12-20 The String Collection Editor while filling a `ListBox` and the completed `ListBox` on a `Form`

Assuming the `Name` property of the `ListBox` is `majorListBox`, the following code is generated in the `InitializeComponent()` method when you fill the String Collection Editor with the strings in Figure 12-20:

```
this.majorListBox.Items.AddRange(new object[] {
    "Accounting",
    "Biology",
    "English",
    "Psychology",
    "Sociology" });
```

Objects added to a `ListBox` are not required to be strings. For example, you could add a collection of `Employee` or `Student` objects. The value returned by each added object's `ToString()` method is displayed in the `ListBox`. After the user selects an object, you can cast the `ListBox`'s `SelectedItem` to the appropriate type and access the object's other properties. The Chapter.12 folder of your student files contains a project named `AddRangeObjectsDemo` that illustrates this technique.

With a ListBox, you allow the user to make a single selection or multiple selections by setting the SelectionMode property appropriately. For example, when the SelectionMode property is set to One, the user can make only a single selection from the ListBox. When the SelectionMode is set to MultiExtended, pressing Shift and clicking the mouse or pressing Shift and one of the arrow keys (up, down, left, or right) extends the selection to span from the previously selected item to the current item. Pressing Ctrl and clicking the mouse selects or deselects an item in the list. Table 12-9 lists the possible SelectionMode values.

When the SelectionMode property is set to SelectionMode.MultiSimple, click the mouse or press the spacebar to select or deselect an item in the list.

Member Name	Description
MultiExtended	Multiple items can be selected, and the user can press the Shift, Ctrl, and arrow keys to make selections.
MultiSimple	Multiple items can be selected.
None	No items can be selected.
One	Only one item can be selected.

Table 12-9 SelectionMode enumeration list

For example, within a Form's Load() method (the one that executes when a Form is first loaded), you could add the following:

```
this.majorListBox.SelectionMode =
   System.Windows.Forms.SelectionMode.MultiExtended;
```

As the example in Figure 12-21 shows, when you size a ListBox so that all the items cannot be displayed at the same time, a scroll bar is provided automatically on the side. The ListBox also provides the Boolean MultiColumn property, which you can set to display items in columns instead of a straight vertical list. This approach allows the control to display more items and can eliminate the need for the user to scroll down to an item. See Figure 12-22 which shows a multi-column list box to which eight strings have been added.

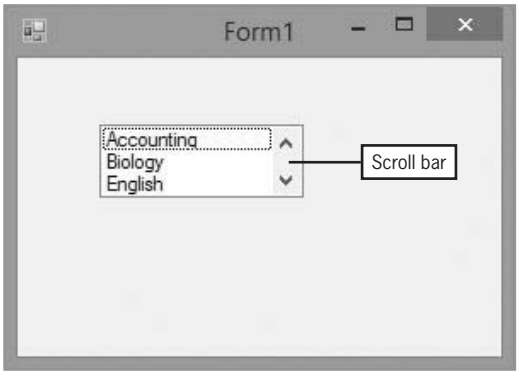

Figure 12-21 A `ListBox` with a scroll bar

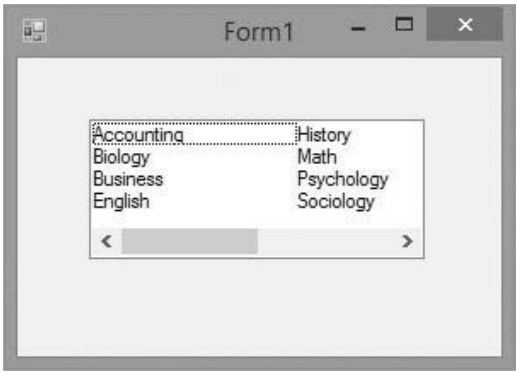

Figure 12-22 A multicolumn `ListBox`

The `SelectedItem` property of a `ListBox` contains a reference to the item a user has selected. For example, you can modify a `Label`'s `Text` property in the `majorListBox_SelectedIndexChanged()` method with a statement such as the following, which appends the `SelectedItem` value to a label. Figure 12-23 shows a tyical result.

```
private void majorListBox_SelectedIndexChanged
   (object sender, EventArgs e)
{
   majorLabel.Text = "You selected " + majorListBox.SelectedItem;
}
```

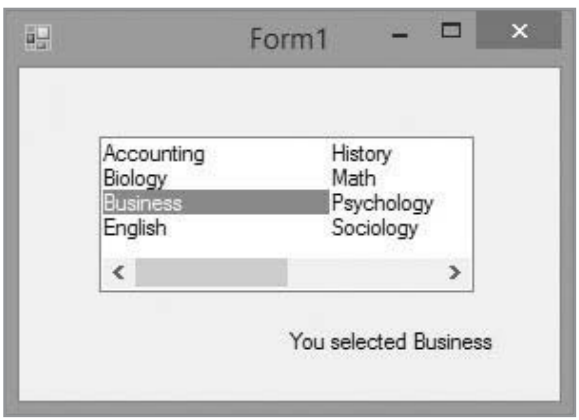

Figure 12-23 The `ListBoxDemo` application after a user has chosen *Business*

The `Items.Count` property of a `ListBox` object holds the number of items in the `ListBox`. The `GetSelected()` method accepts an integer argument representing the position of an item in the list. The method returns `true` if an item is selected and `false` if it is not. Therefore, code like the following could be used to count the number of selections a user makes from `majorListBox`:

```
int count = 0;
for(int x = 0; x < majorListBox.Items.Count; ++x)
   if(majorListBox.GetSelected(x))
      ++count;
```

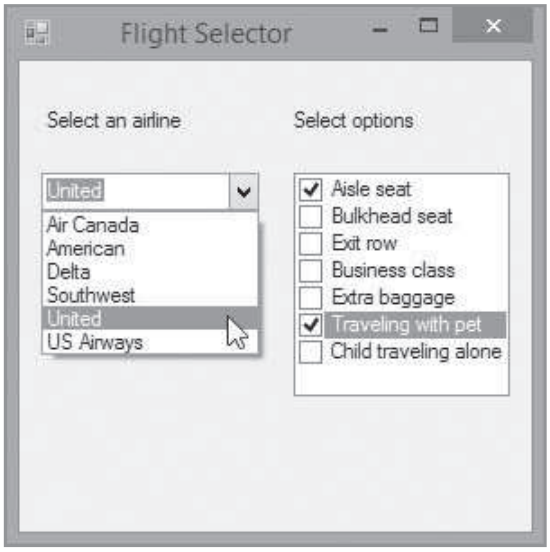

Figure 12-24 The `FlightSelector` application after the user has made some selections

Recall that the first position in an array is position 0. The same is true in a `ListBox`.

Alternatively, you can use a `ListBox`'s `SelectedItems` property; it contains the items selected in a list. The following code assigns the number of selections a user makes from the `majorListBox` to count:

```
count = majorListBox.SelectedItems.
Count;
```

The `SetSelected()` method can be used to set a `ListBox` item to be automatically selected. For example, the following statement causes the first item in `majorListBox` to be selected:

```
majorListBox.SetSelected(0, true);
```

A **combo box** is a combination of a list box and and an editing control that allows a user to select from the list or to enter new text. The C# class is `ComboBox`, and the default `ComboBox` displays an editing area with a hidden list box. The application in Figure 12-24 contains a `ComboBox` for selecting an airline. A `CheckedListBox` is also similar to a `ListBox`, with check boxes appearing to the left of each desired item. The application in Figure 12-24 uses a `CheckedListBox` for flight options.

TWO TRUTHS & A LIE

Adding `ListBox`, `CheckedListBox`, and `ComboBox` Controls to a `Form`

1. With a `ListBox Control`, the user can select only one option at a time.
2. A `ComboBox` is similar to a `ListBox`, except that it displays an additional editing control that allows users to select from the list or to enter new text.
3. A `CheckedListBox` is similar to a `ListBox`, with check boxes appearing to the left of each desired item.

The false statement is #1. With a `ListBox`, you allow the user to make a single selection or multiple selections by setting the `SelectionMode` property appropriately.

Adding MonthCalendar and DateTimePicker Controls to a Form

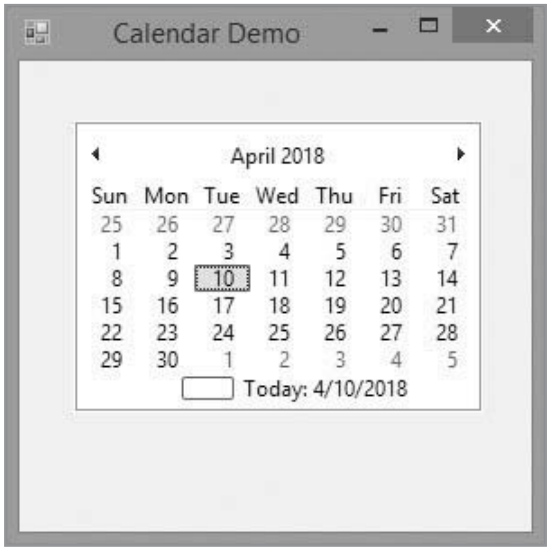

Figure 12-25 Typical execution of the MonthCalendarDemo application at startup

The MonthCalendar and DateTimePicker Controls allow you to retrieve date and time information. Figure 12-25 shows a MonthCalendar that has been placed on a Form. The current date is contained in a rectangle by default. When the user clicks a different date, it is shaded. Controls at the top of the calendar allow the user to go forward or back one month at a time, or the user can also move to a specific month or year by clicking the month and year title at the top of the calendar and then making a new month and year selection. Table 12-10 describes common MonthCalendar properties and the default event.

Property or Method	Description
MaxDate	Sets the last day that can be selected (the default is 12/31/9998)
MaxSelectionCount	Sets the maximum number of dates that can be selected at once (the default is 7)
MinDate	Sets the first day that can be selected (the default is 1/1/1753)
MonthlyBoldedDates	An array of dates that appear in boldface in the calendar (for example, holidays)
SelectionEnd	The last of a range of dates selected by the user
SelectionRange	The dates selected by the user
SelectionStart	The first of a range of dates selected by the user
ShowToday	If true, the date is displayed in text at the bottom of the calendar
ShowTodayCircle	If true, today's date is circled (the "circle" appears as a square)
DateChanged()	Default event that is generated when the user selects a date

Table 12-10 Commonly used MonthCalendar properties and default event

Several useful methods can be applied to the `SelectionStart` and `SelectionEnd` properties of `MonthCalendar`, including the following:

- `ToShortDateString()`, which displays the date in the format of 2/16/2016
- `ToLongDateString()`, which displays the date in the format of Sunday, February 16, 2016
- `AddDays()`, which takes a `double` argument and adds a specified number of days to the date
- `AddMonths()`, which takes an `int` argument and adds a specified number of months to the date
- `AddYears()`, which takes an `int` argument and adds a specified number of years to the date

The format in which dates are displayed depends on the operating system's regional settings. For example, using United Kingdom settings, the short string format would use the day first, followed by the month, as in 16/02/2016. The examples in the list above assume United States settings. To change your regional setting in Windows, you can go to Control Panel, click Region, and choose a region from the drop down list.

The `AddDays()` method accepts a `double` argument because you can add fractional days to `SelectionStart` and `SelectionEnd`.

`SelectionStart` and `SelectionEnd` are structures of the `DateTime` type. The chapter "Files and Streams" contains additional information about using `DateTime` objects to determine when files were created, modified, or accessed.

Many business and financial applications use `AddDays()`, `AddMonths()`, and `AddYears()` to calculate dates for events, such as payment for a bill (perhaps due in 10 days from an order) or scheduling a salesperson's callback to a customer (perhaps two months after initial contact). The default event for `MonthCalendar` is `DateChanged()`. For example, Figure 12-26 shows a method that executes when the user clicks a `MonthCalendar` named `calendar`. Ten days are added to a selected date and the result is displayed on a `Label` that has been named `messageLabel`. Figure 12-27 shows the output when the user selects May 29, 2018. The date that is 10 days in the future is correctly calculated as June 8.

```
private void calendar_DateChanged(object sender, DateRangeEventArgs e)
{
   const int DAYS_TO_ADD = 10;
   messageLabel.Text = "Date " + DAYS_TO_ADD +
      " days after selection is " +
      calendar.SelectionStart.AddDays(DAYS_TO_ADD).ToShortDateString();
}
```

Figure 12-26 The `calendar_DateChanged()` method

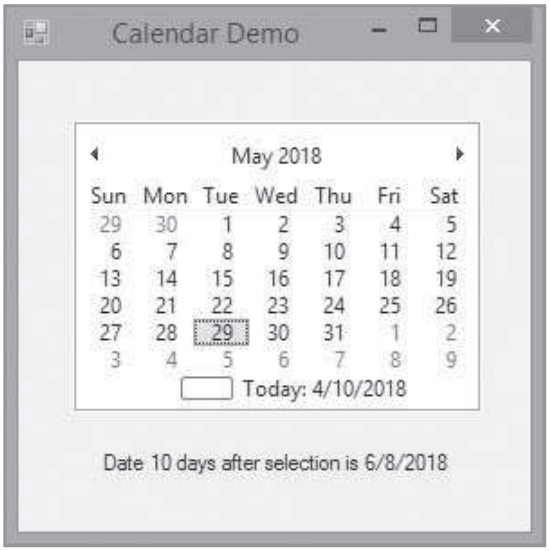

Figure 12-27 Typical execution of the MonthCalendarDemo program

In Chapter 14, you will learn to use `DateTime` objects to hold data about dates and times.

If you set the `MinDate` value to the `MonthCalendar`'s `TodayDate` property, the user cannot select a date before the date you select. For example, you cannot make appointments or schedule deliveries in the past, so you might code the following in a form's `Load()` method:

```
calendar1.MinDate =
calendar1.TodayDate;
```

Conversely, you might want to prevent users from selecting a date in the future—for example, if the user is entering his birth date, it cannot be in the future. In that case, you could code a statement similar to the following:

```
calendar1.MaxDate = calendar1.TodayDate;
```

The `DateTimePicker Control` displays a month calendar when the down arrow is selected. This feature can be especially useful if you do not have much space available on a `Form`. Figure 12-28 shows a `DateTimePicker` before and after the user clicks the down arrow.

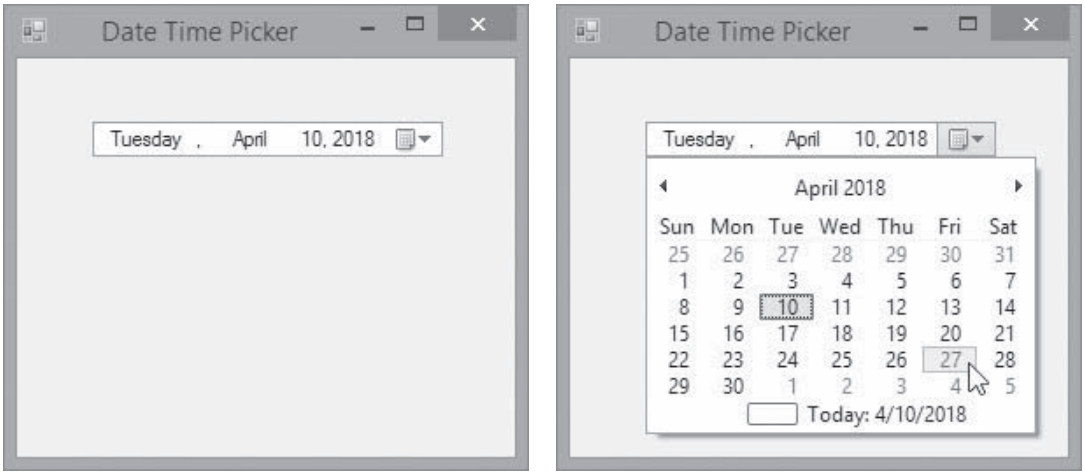

Figure 12-28 The `DateTimePicker Control`

When you use the CustomFormat property, the date displayed in a DateTimePicker Control is more customizable than the one in a MonthCalendar. Table 12-11 describes some commonly used DateTimePicker properties and the default event.

Property or Method	Description
CalendarForeColor	Sets the calendar text color
CalendarMonthBackground	Sets the calendar background color
CustomFormat	A string value that uses codes to set a custom date and time format. For example, to display the date and time as 02/16/2011 12:00 PM - Wednesday, set this property to "MM'/'dd'/'yyyy hh':'mm tt - dddd". See the C# documentation for a complete set of format string characters.
Format	Sets the format for the date or time. Options are Long (for example, Wednesday, February 16, 2011), Short (2/16/2011), and Time (for example, 3:15:01 PM). You can also create a CustomFormat.
Value	The data selected by the user
ValueChanged()	Default event that is generated when the Value property changes

Table 12-11 Commonly used DateTimePicker properties and default event

TWO TRUTHS & A LIE

Adding MonthCalendar and DateTimePicker Controls to a Form

1. The MonthCalendar and DateTimePicker Controls allow you to retrieve date and time information.
2. The default event for MonthCalendar is DateChanged().
3. The DateTimePicker Control displays a small clock when you click it.

The false statement is #3. The DateTimePicker Control displays a month calendar when the down arrow is selected.

Working with a Form's Layout

When you place Controls on a Form in the IDE, you can drag them to any location to achieve the effect you want.

When you drag multiple Controls onto a Form, blue **snap lines** appear and help you align new Controls with others already in place. Figure 12-29 shows two snap lines that you can use to align a second label below the first one. Snap lines also appear when you place a control closer to the edge of a container than is recommended.

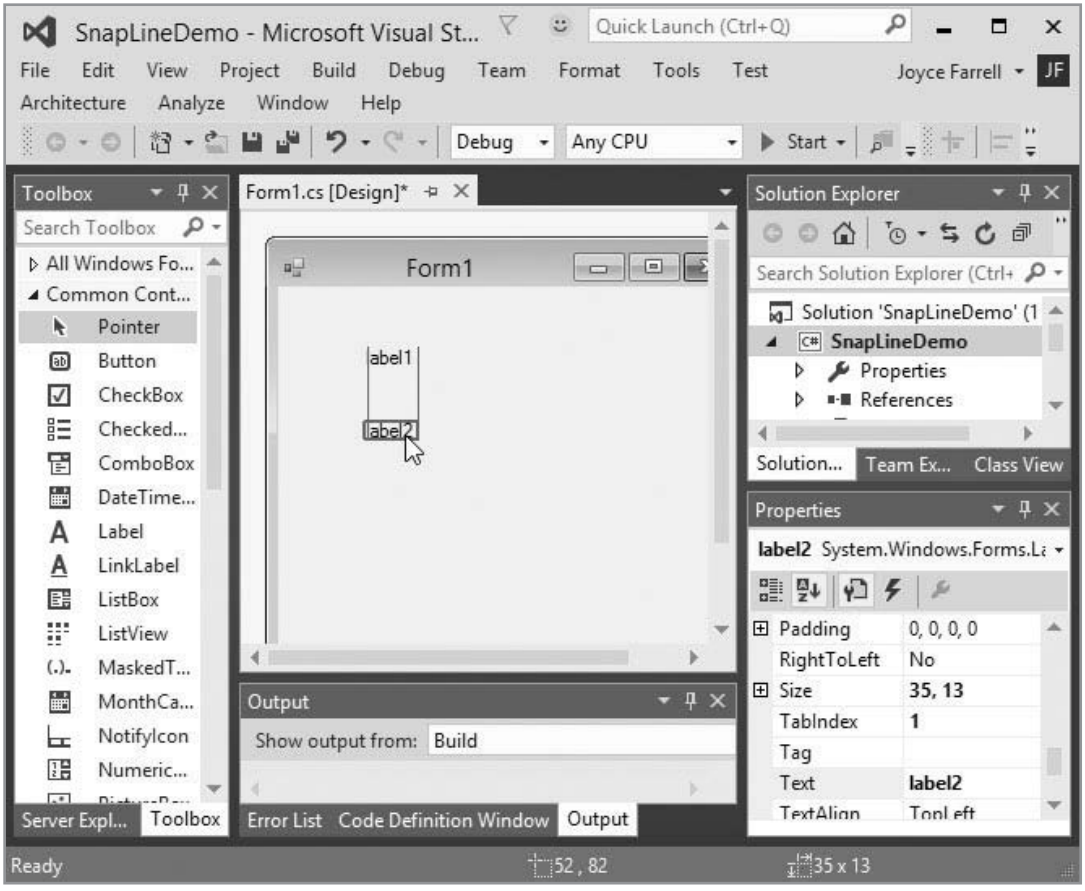

Figure 12-29 Snap lines in the Visual Studio Designer

You also can use the Location property in a Control's Properties list to specify a location. With either technique, code like the following is generated:

```
this.label1.Location = new System.Drawing.Point(23, 19);
```

Several other properties can help you to manage the appearance of a `Form` (or other `ContainerControl`). For example, setting the `Anchor` property causes a `Control` to remain at a fixed distance from the side of a container when the user resizes it. Figure 12-30 shows the Properties window for a `Label` that has been placed on a `Form`. The `Anchor` property has a drop-down window that lets you select or deselect the sides to which the label should be anchored. For most `Controls`, the default setting for `Anchor` is `Top, Left`.

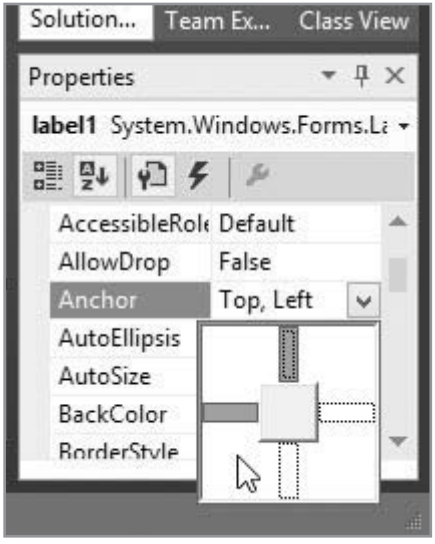

Figure 12-30 Selecting an `Anchor` property

Figure 12-31 shows a `Form` with two `Labels`. On the `Form`, `label1` has been anchored to the top left, and `label2` has been anchored to the bottom right. The left side of the figure shows the `Form` as it first appears to the user, and the right side shows the `Form` after the user has resized it. Notice that in the resized `Form`, `label1` is still the same distance from the top left as it originally was, and `label2` is still the same distance from the bottom right as it originally was. Anchoring is useful when users expect a specific control to always be in the same general location in a container.

Figure 12-31 A `Form` with two `Labels` anchored to opposite corners

Setting the `Dock` property attaches a `Control` to the side of a container so that the `Control` stretches when the container's size is adjusted. Figure 12-32 shows the drop-down `Dock` Properties window for a `Button`. You can select any region in the window. Figure 12-33 shows a `Button` docked to the bottom of a `Form` before and after the `Form` has been resized.

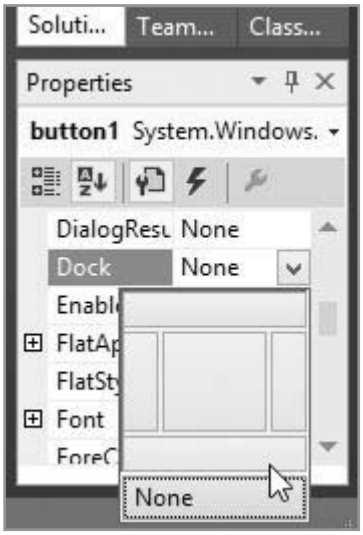

Figure 12-32 The Dock Properties window

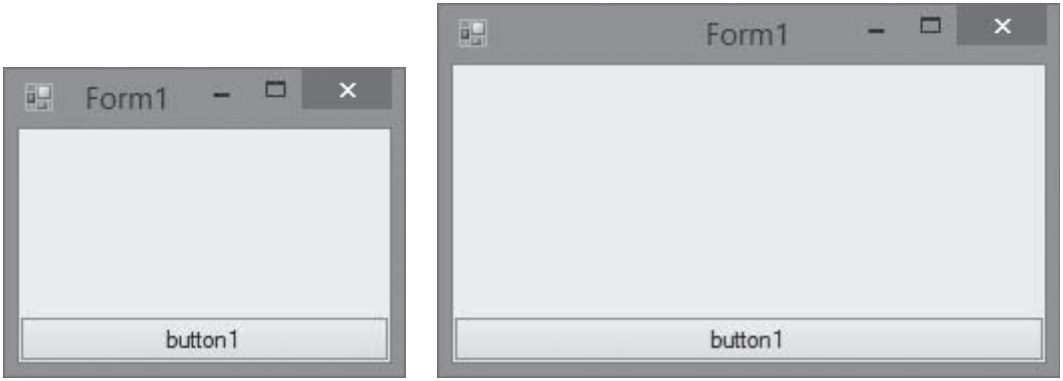

Figure 12-33 A Form with a docked Button before and after resizing

A Form also has a MinimumSize property and a MaximumSize property. Each has two values—Width and Height. If you set these properties, the user cannot make the Form smaller or larger than you have specified. If you do not want the user to be able to adjust a Form's size at all, set the MinimumSize and MaximumSize properties to be equal.

Understanding GroupBoxes and Panels

Many types of ContainerControls are available to hold Controls. For example, a group box or panel can be used to contain a group of other controls and to move them as a group.

To create either of these Controls, you drag it from the Toolbox in the IDE and then drag the Controls you want on top of it. In C#, a **group box** and a **panel** differ in the following ways:

- A group box is created from the GroupBox class, and a panel is created from the Panel class.
- GroupBoxes can display a caption, but Panels cannot.
- Panels can include a scroll bar that the user can manipulate to view Controls; GroupBoxes do not have scroll bars.

You can anchor or dock Controls inside a GroupBox or Panel, and you can anchor or dock a GroupBox or Panel inside a Form. Doing this provides Control groups that can be arranged easily.

If you place several GroupBox Controls on a Form and several RadioButtons in each GroupBox, then a user can select one RadioButton from each GroupBox instead of being able to select just one RadioButton on a Form. In other words, each GroupBox operates independently.

When an application contains multiple GroupBox or Panel Controls on a Form, pressing Tab moves the focus to the next GroupBox or Panel. Then, within the GroupBox or Panel, you use arrow keys to progress to successive RadioButtons.

TWO TRUTHS & A LIE

Working with a Form's Layout

1. Setting the Anchor property causes a Control to remain at a fixed distance from the side of a container when the user resizes it.
2. Setting the Dock property attaches a Control to the side of a container so that the Control's size does not change when the container's size is adjusted.
3. The GroupBox and Panel controls are ContainerControls.

The false statement is #2. Setting the Dock property attaches a Control to the side of a container so that the Control stretches when the container's size is adjusted.

Adding a MenuStrip to a Form

Many programs you use in a Windows environment contain a **menu strip**, which is a horizontal list of general options that appears under the title bar of a Form or Window. When you click an item in a menu strip, you might initiate an action. More frequently, you see a list box that contains more specific options. Each of these might initiate an action or lead to another menu. For example, the Visual Studio IDE contains a horizontal menu strip that begins with the options File, Edit, and View. You have used word-processing, spreadsheet, and game programs with similar menus.

You can add a MenuStrip Control object to any Form you create. Using the Visual Studio IDE, you can add a MenuStrip to a Form by dragging it from the Toolbox onto the Form. This creates a menu bar horizontally across the top of the Form, just below the title bar. The strip extends across the width of the Form and contains a *Type Here* text box. When you click the text box, you can enter a menu item. Each time you add a menu item, new boxes are created so you can see where your next options will go, as shown in Figure 12-34.

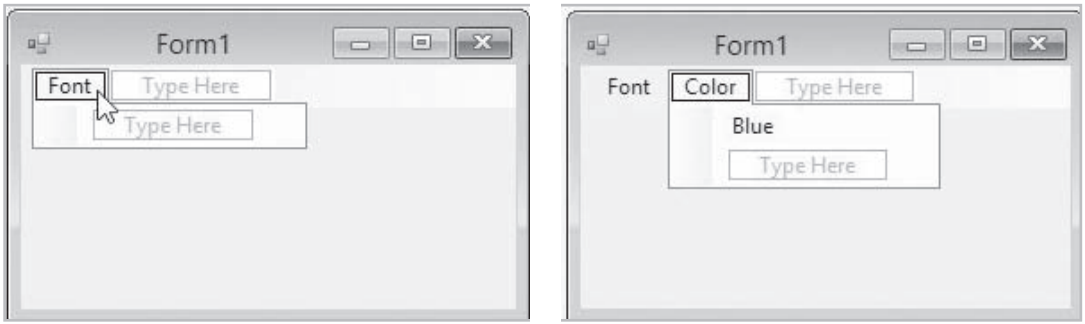

Figure 12-34 Creating a Form with a MenuStrip

If you do not see MenuStrip in your Toolbox, click the Menus & Toolbars group to expose it. You can click the MenuStrip icon at the bottom of the design screen to view and change the properties for the MenuStrip. For example, you might want to change the Font or BackColor for the MenuStrip.

If you create each menu item with an ampersand (&) in front of a unique letter, then the letter becomes an access key, and the user can press Alt and the letter to activate the menu choice. For example, if two choices were Green and Gray, you might want to type &Green and G&ray so the user could type Alt+G to select Green and Alt+R to select Gray.

When you double-click an entry in the MenuStrip, a Click() method is generated. For example, if you double-click *Blue* under *Color* in the menu being created in Figure 12-34, the method generated is blueToolStripMenuItem_Click(). As with all the other controls you have learned

about, you can write any code statements within the method. For example, suppose that a `Label` named `helloLabel` has been dragged onto the `Form`. If choosing the *Blue* menu option should result in a blue forecolor for the label, you might code the method as follows:


```
private void blueToolStripMenuItem_Click(object sender, EventArgs e)
{
   helloLabel.ForeColor = Color.Blue;
}
```

You can work with the other menu items in this program in an exercise at the end of this chapter.

If possible, your main horizontal menu selections should be single words. That way, a user will not mistakenly think that a single menu item represents multiple items. Most applications do not follow this single-word convention for submenus. Also, users expect menu options to appear in conventional order. For example, users expect the far-left option on the main menu bar to be *File*, and they expect the *Exit* option to appear under *File*. Similarly, if an application contains a *Help* option, users expect to find it at the right side of the main menu bar. You should follow these conventions when designing your own menus.

Watch the video *Using a `MenuStrip`*.

TWO TRUTHS & A LIE

Adding a `MenuStrip` to a `Form`

1. When you click an item in a menu strip, the most common result is to initiate an action.
2. When you drag a `MenuStrip Control` object onto a `Form` using the Visual Studio IDE, the `MenuStrip` is added horizontally across the top of the `Form`, just below the title bar.
3. The default event for `MenuStrip` is `Click()`.

The false statement is #1. When you click an item in a menu strip, you might initiate an action. More frequently, you see a list box that contains more specific options.

Using Other Controls

If you examine the Visual Studio IDE or search through the Visual Studio documentation, you will find many other Controls that are not covered in this chapter. If you click Project on the menu bar and click Add New Item, you can add extra Forms, Files, Controls, and other elements to your project. (In the next "You Do It" section, you create a project that adds new Forms that appear after selections are made from a primary Form.) New controls and containers will be developed in the future, and you might even design new controls of your own. Still, all controls will contain properties and methods, and your solid foundation in C# will prepare you to use new controls effectively.

You Do It

Adding CheckBoxes to a Form

In the next steps, you add two CheckBoxes to the BedAndBreakfast Form. These controls allow the user to select an available room and view information about it.

1. Open the BedAndBreakfast project in Visual Studio, if it is not still open on your screen.

2. In the Design view of the BedAndBreakfast project in the Visual Studio IDE, drag a CheckBox onto the Form below the *Check our rates* Label. (See Figure 12-35 for its approximate placement.) Change the Text property of the CheckBox to **BelleAire Suite**. Change the Name of the property to **belleAireCheckBox**. Drag a second CheckBox onto the Form beneath the first one. Change its Text property to **Lincoln Room** and its Name to **lincolnCheckBox**.

(continues)

(continued)

Figure 12-35 The BedAndBreakfast `Form` with two `CheckBoxes`

3. Next, you will create two new `Forms`: one that appears when the user selects the BelleAire `CheckBox` and one that appears when the user selects the Lincoln `CheckBox`. Click **Project** on the menu bar, and then click **Add New Item**. In the Add New Item window, click **Windows Form**. In the Name text box at the bottom of the window, type **`BelleAireForm`**. See Figure 12-36.

(continues)

(continued)

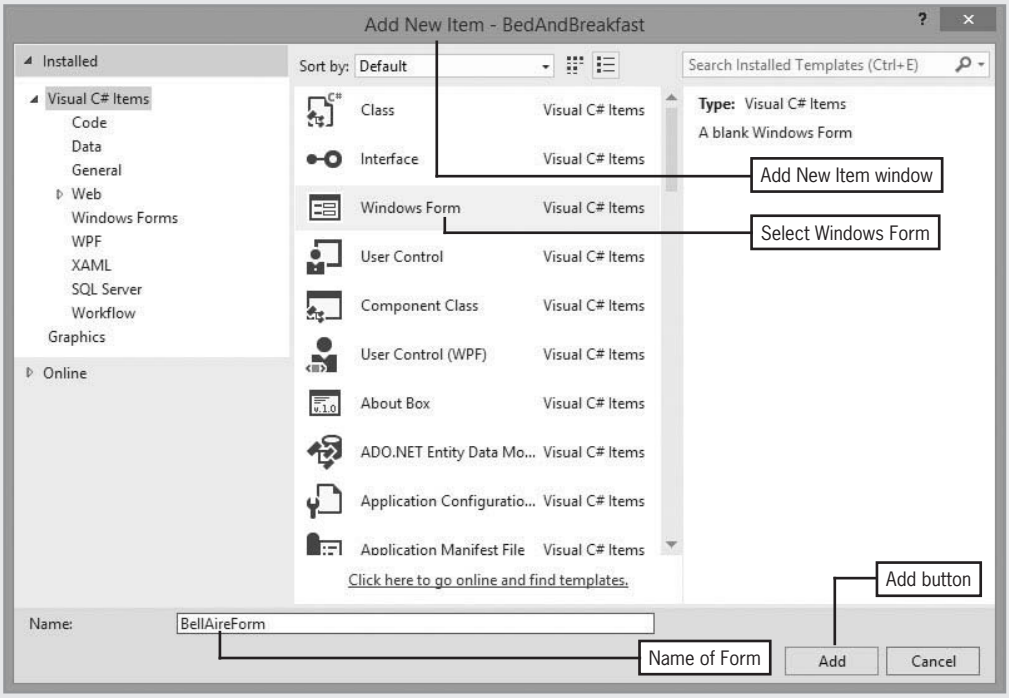

Figure 12-36 The Add New Item window

4. Click the **Add** button. A new `Form` is added to the project, and its `Name` and `Text` (title bar) properties contain *BelleAireForm*. Save the project (and continue to do so periodically).

5. Change the `BackColor` property of the `Form` to `Yellow` to match the color of `BaileysForm`.

6. Drag a `Label` onto the `Form`. Change the `Name` of the `Label` to **`belleAireDescriptionLabel`**. Change the `Text` property of the `Label` to contain the following: **The BelleAire suite has two bedrooms, two baths, and a private balcony**. Click the arrow on the text property to type the long label message on two lines. Adjust the size and position of the `Label` to resemble Figure 12-37. Drag a second `Label` onto the `Form`, name it **`belleAirePriceLabel`**, and type the price as the `Text` property: **$199.95 per night**.

(continues)

(continued)

Figure 12-37 The `BelleAireForm` with two `Labels`

7. Select the **Pointer** tool from the Toolbox at the left of the screen. Drag it to encompass both `Labels`. In the Properties list, select the **`Font`** property to change the `Font` for both `Controls` at once. Choose a suitable `Font`. Figure 12-38 shows **10-point Regular Papyrus**; you might choose a different font. Adjust the positions of the `Labels` if necessary to achieve a pleasing effect.

Figure 12-38 `Font` changed for the `Labels` on the BellAire `Form`

(continues)

(continued)

8. Click the **Form1.cs[Design]** tab at the top of the Designer screen to view the Bailey's Bed and Breakfast Form. Double-click the **BelleAire Suite CheckBox**. The program code (the method shell for the default event of a CheckBox) appears in the IDE main window. Within the belleAireCheckBox_CheckedChanged() method, add an if statement that determines whether the BelleAire CheckBox is checked. If it is checked, create a new instance of BelleAireForm and display it.

```
private void belleAireCheckBox_CheckedChanged(object sender,
   EventArgs e)
{
   if (belleAireCheckBox.Checked)
   {
      BelleAireForm belleAireForm = new BelleAireForm();
      belleAireForm.ShowDialog();
   }
}
```

When a new Form (or other Windows class) is instantiated, it is not visible by default. ShowDialog() shows the window and disables all other windows in the application. The user must dismiss the new Form before proceeding. A secondary window that takes control of a program is a **modal window**; the user must deal with this window before proceeding.

9. Save and then execute the program by selecting **Debug** from the menu bar, and then select **Start Without Debugging**. The main BedAndBreakfast Form appears. Click the **BelleAire Suite CheckBox**. The BelleAire Form appears. Dismiss the Form. Click the **Lincoln Room CheckBox**. Nothing happens because you have not yet written event code for this CheckBox. When you uncheck and then check the **BelleAire Suite CheckBox** again, the BelleAire form reappears. Dismiss the BelleAire Form.

10. When you dismiss the BelleAire Form, the BelleAire CheckBox remains checked. To see it appear as unchecked after its Form is dismissed, dismiss the program's main form (Bailey's Bed and Breakfast) to end the program. Then, add a third statement within the if block in the CheckedChanged() message as follows:

```
belleAireCheckBox.Checked = false;
```

That way, whenever the CheckedChanged() method executes because the belleAireCheckBox was checked, it will become unchecked. Save the project, and then execute it again. When you select the BelleAire CheckBox, view the Form, and dismiss it, the CheckBox appears unchecked and is ready to check again. Dismiss the BedAndBreakfast Form.

(continues)

(continued)

11. Click **Project** on the menu bar, and then click **Add New Item**. Click **Windows Form** and enter its `Name`: **`LincolnForm`**. When the new `Form` appears, its `Name` and `Text` properties will have been set to **LincolnForm**. Change the `Text` property to **Lincoln Room**. Then add two `Labels` to the `Form`, and provide appropriate `Name` properties for them. Change the `Text` on the first `Label` to **Return to the 1850s in this lovely room with private bath**. The second should be **$110 per night**. Change the `Form`'s `BackColor` property to **White**. Change the `Font` to match the `Font` you chose for the BelleAire `Form`. See Figure 12-39.

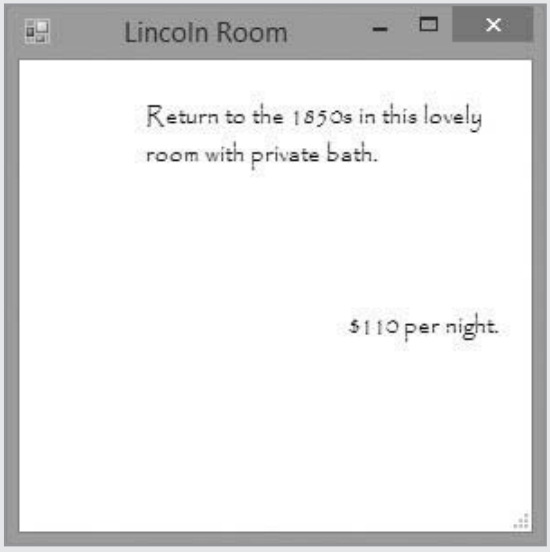

Figure 12-39 The `LincolnForm`

12. From the Toolbox, drag a `PictureBox` onto the `Form`. Select its **`Image`** property. A dialog box allows you to select a resource. Click **Local resource**, and then click the **Import** button to browse for an image. Find the AbeLincoln file in the Chapter.12 folder of your downloadable student files, and double-click it. (Alternately, you can import another image you prefer.) After the image appears in the Select Resource dialog box, click **OK**. The selected image appears in the `PictureBox`. Adjust the size of the `Form` and the sizes and positions of the labels and picture box so that the picture is fully visible and everything looks attractive on the `Form`. See Figure 12-40.

(continues)

(continued)

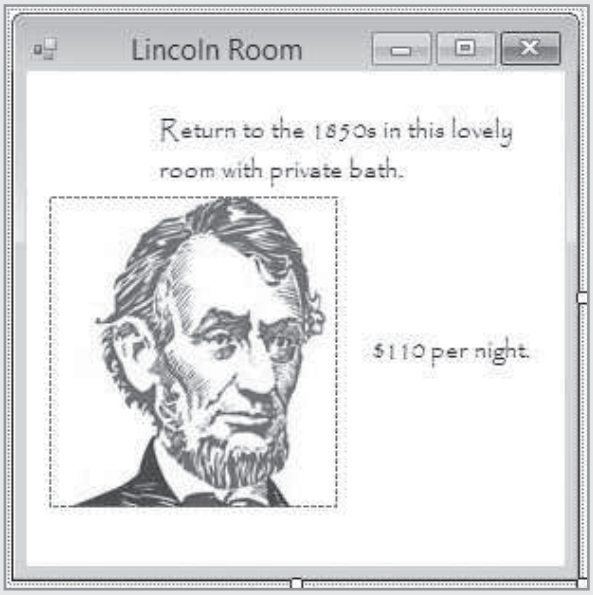

Figure 12-40 The `LincolnForm` with an `Image` in a `PictureBox`

The AbeLincoln file was obtained at www.free-graphics.com. You can visit the site and download other images to use in your own applications. You should also search the Web for "free clip art" and similar phrases.

13. In the Solution Explorer, double-click **`Form1.cs`** or click the **Form1.cs[Design]** tab at the top of the design screen. On the BedAndBreakfast `Form`, double-click the Lincoln Room `CheckBox`, and add the following `if` statement to the `lincolnCheckBox_CheckedChanged()` method:

```
private void lincolnCheckBox_CheckedChanged(object sender,
   EventArgs e)
{
   if (lincolnCheckBox.Checked)
   {
      LincolnForm lincolnForm = new LincolnForm();
      lincolnForm.ShowDialog();
      lincolnCheckBox.Checked = false;
   }
}
```

(continues)

(continued)

14. Save the project and then execute it. When the BedAndBreakfast `Form` appears, click either `CheckBox`—the appropriate informational `Form` appears. Close it and then click the other `CheckBox`. Again, the appropriate `Form` appears.

15. Close all forms.

Adding `RadioButtons` to a `Form`

Next you add more `Controls` to the BedAndBreakfast `Form`. You generally use `RadioButtons` when a user must select from mutually exclusive options.

1. In the Design view of the main `Form` in the BedAndBreakfast project, add a `Button` near the bottom of the `Form`. Change the `Button`'s `Name` property to **`mealButton`** and the `Button`'s `Text` to **Click for meal options**. Adjust the size of the `Button` so that its text is fully visible.

2. From the menu bar, select **Project**, click **Add New Item**, and click **Windows Form**. Name the `Form` **`BreakfastOptionForm`**, and click **Add**. On the new `Form`, make the following changes:

   - Set the `Form`'s `BackColor` to **Yellow**.
   - Drag a `Label` onto the `Form`. Name it appropriately, and set its `Text` to **Select your breakfast option**.
   - Drag three `RadioButtons` onto the `Form`. Set their respective `Text` properties to **Continental**, **Full**, and **Deluxe**. Set their respective `Names` to **`contBreakfastButton`**, **`fullBreakfastButton`**, and **`deluxeBreakfastButton`**.
   - Select an appropriate font for the `Label` and `RadioButtons`.
   - Drag a `Label` onto the `Form`, and then set its `Text` to **Price**: and its `Name` to **`priceLabel`**. Make the `Font` property a little larger than for the other `Form` components.

   See Figure 12-41 for approximate placement of all these `Controls`.

(continues)

(continued)

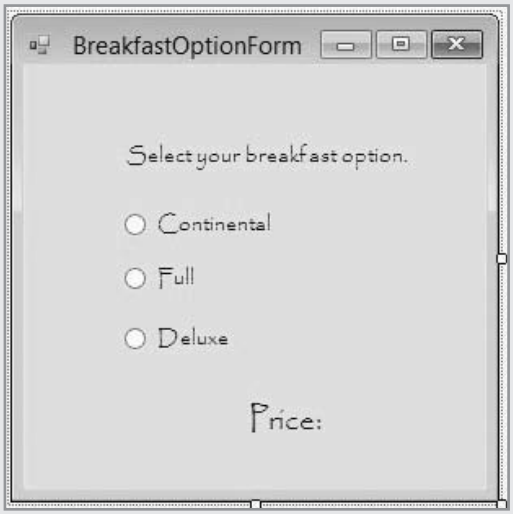

Figure 12-41 Developing the `BreakfastOptionForm`

3. Double-click the title bar of the `BreakfastOptionForm` to generate a method named `BreakfastOptionForm_Load()`. Within this method, you can type statements that execute each time the `Form` is created. Add the following statements within the `BreakfastOptionForm` class, which declare three constants representing prices for different breakfast options. Within the `BreakfastOptionForm_Load()` method, set the `priceLabel Text` property to the lowest price by default when the `Form` loads.

```
public partial class BreakfastOptionForm : Form
{
   private const double CONT_BREAKFAST_PRICE = 6.00;
   private const double FULL_BREAKFAST_PRICE = 9.95;
   private const double DELUXE_BREAKFAST_PRICE = 16.50;
   public BreakfastOptionForm()
   {
      InitializeComponent();
   }
   private void BreakfastOptionForm_Load
      (object sender, EventArgs e)
   {
      priceLabel.Text = "Price: " +
         CONT_BREAKFAST_PRICE.ToString("C");
   }
}
```

(continues)

(continued)

4. Return to the Design view for the `BreakfastOptionForm`, and double-click the **`contBreakfastButton RadioButton`**. When you see the generated `CheckedChanged()` method, add a statement that sets `priceLabel` to the continental breakfast price when the user makes that selection:

```
private void contBreakfastButton_CheckedChanged
   (object sender, EventArgs e)
{
   priceLabel.Text = "Price: " +
      CONT_BREAKFAST_PRICE .ToString("C");
}
```

5. Return to the Design view for the `BreakfastOptionForm`, double-click the **`fullBreakfastButton RadioButton`**, and add a statement to the generated method that sets the `priceLabel` to the full breakfast price when the user makes that selection:

```
private void fullBreakfastButton_CheckedChanged
   (object sender, EventArgs e)
{
   priceLabel.Text = "Price: " +
      FULL_BREAKFAST_PRICE.ToString("C");
}
```

6. Return to the Design view for the `BreakfastOptionForm`, double-click the **`deluxeBreakfastButton RadioButton`**, and add a statement to the generated method that sets the `priceLabel` to the deluxe breakfast price when the user makes that selection:

```
private void deluxeBreakfastButton_CheckedChanged
   (object sender, EventArgs e)
{
   priceLabel.Text = "Price: " +
      DELUXE_BREAKFAST_PRICE.ToString("C");
}
```

7. In the Solution Explorer, double-click the **Form1.cs file** to view the original `Form`. Double-click the **`mealButton Button`**. When the `Click()` method is generated, add the following code so that the `BreakfastOptionForm` is loaded when a user clicks the `Button`:

```
private void mealButton_Click(object sender,
   EventArgs e)
{
   BreakfastOptionForm breakfastForm = new
      BreakfastOptionForm();
   breakfastForm.ShowDialog();
}
```

(continues)

(continued)

8. Save the project and execute it. When the BedAndBreakfast Form appears, confirm that the BelleAire Suite and Lincoln Room CheckBoxes still work correctly, displaying their information Forms when they are clicked. Then click the **Click for meal options Button**. By default, the Continental breakfast option is chosen, as shown in Figure 12-42, so the price is $6.00. Click the other RadioButton options to confirm that each correctly changes the breakfast price.

Figure 12-42 The BreakfastOptionForm with Continental breakfast RadioButton selected

9. Dismiss all the Forms, and close Visual Studio.

Chapter Summary

- The `Control` class provides definitions for GUI objects such as `Form`s and `Button`s. There are 26 direct descendents of `Control` and additional descendents of those classes. Each `Control` has more than 80 `public` properties and 20 `protected` ones.
- When you design GUI applications using the Visual Studio IDE, much of the code is automatically generated.
- You use the `Font` class to change the appearance of printed text on your `Form`s.
- A `LinkLabel` is similar to a `Label`; it is a child of `Label`, but it provides the additional capability to link the user to other sources, such as Web pages or files.
- The `Color` class contains a wide variety of predefined `Color`s that you can use with your `Control`s.
- `CheckBox` objects are GUI widgets the user can click to select or deselect an option. When a `Form` contains multiple `CheckBox`es, any number of them can be checked or unchecked at the same time. `RadioButton`s are similar to `CheckBox`es, except that when they are placed on a `Form`, only one `RadioButton` can be selected at a time.
- A `PictureBox` is a `Control` in which you can display graphics from a bitmap, icon, JPEG, GIF, or other image file type.
- `ListBox`, `ComboBox`, and `CheckedListBox` objects descend from `ListControl` and enable you to display lists of items that the user can select by clicking. The `MonthCalendar` and `DateTimePicker Control`s allow you to retrieve date and time information.
- When you place `Control`s on a `Form` in the IDE, you can drag them to any location to achieve the effect you want. Blue snap lines help you align new `Control`s with others already in place. You also can use the `Location`, `Anchor`, `Dock`, `MinimumSize`, and `MaximumSize` properties to customize a `Form`'s appearance. You can use a `GroupBox` or `Panel` to group related `Control`s on a `Form`.
- Many programs you use in a Windows environment contain a menu strip, which is a horizontal list of general options that appears under the title bar of a `Form` or `Window`. When a user clicks an item in a `MenuStrip Control`, a list box that contains more specific options is displayed frequently.
- If you examine the Visual Studio IDE or search through the Visual Studio documentation, you will find many `Control`s. If you click Project on the menu bar and click Add New Item, you can add extra `Form`s, `File`s, `Control`s, and other elements to a project.

Key Terms

Graphical control elements, or, more simply, **controls**, are the components through which a user interacts with a GUI program.

Widgets are GUI controls.

A **link label** is a control with text that links a user to other resources such as Web pages or files.

The **default event** for a `Control` is the one generated when you double-click the `Control` while designing it in the IDE and is the method you are most likely to alter when you use the `Control`, as well as the event that users most likely expect to generate when they encounter the `Control` in a working application.

A **check box** is a GUI widget the user can click to select or deselect an option.

A **radio button** is an option in a group in which only one can be selected at a time—selecting any radio button automatically deselects the others.

An **access key** provides a shortcut way to make a selection using the keyboard.

A **picture box** is a GUI element that can display graphics.

A **list box** is a GUI element that displays a list of items that the user can select by clicking.

A **combo box** is a GUI element that is a combination of a list box and an editing control that allows a user to select from the list or enter new text.

Snap lines appear in a design environment to help you align new `Control`s with others already in place.

A **group box** is a GUI element that contains other GUI elements; it is similar to a panel but does not have a scroll bar and can contain a caption.

A **panel** is a GUI element that contains other GUI elements; it is similar to a group box but does not have a caption and can contain a scroll bar.

A **menu strip** is a horizontal list of general options that appears under the title bar of a `Form` or `Window`.

A **modal window** is a secondary window that takes control from a primary window and that a user must deal with before proceeding.

Review Questions

1. `Label`s, `Button`s, and `CheckBox`es are all _____ .
   a. GUI objects
   b. `Control`s
   c. widgets
   d. all of these

2. All `Control` objects descend from _____ .

   a. `Form`
   b. `Component`
   c. `ButtonBase`
   d. all of these

3. Which of the following is most like a `RadioButton`?

   a. `ListControl`
   b. `CheckedListBox`
   c. `PictureBox`
   d. `Button`

4. Which of the following is not a commonly used `Control` property?

   a. `BackColor`
   b. `Language`
   c. `Location`
   d. `Size`

5. The `Control` you frequently use to provide descriptive text for another `Control` object is a _____ .

   a. `Form`
   b. `Label`
   c. `CheckBox`
   d. `MessageBox`

6. Which of the following creates a `Label` named `firstLabel`?

   a. `firstLabel = new firstLabel();`
   b. `Label = new firstLabel();`
   c. `Label firstLabel = new Label();`
   d. `Label firstLabel = Label();`

7. The property that determines what the user reads on a `Label` is the _____ property.

   a. `Text`
   b. `Label`
   c. `Phrase`
   d. `Setting`

8. Which of the following correctly creates a `Font`?

   a. `Font myFont = new Font("Arial", 14F, FontStyle.Bold);`
   b. `Font myFont = new Font("Courier", 13.6);`
   c. `myFont = Font new Font("TimesRoman", FontStyle.Italic);`
   d. `Font myFont = Font(20, "Helvetica", Underlined);`

9. The default event for a `Control` is the one that _____ .
   a. occurs automatically whether or not a user manipulates the `Control`
   b. is generated when you double-click the `Control` while designing it in the IDE
   c. requires no parameters
   d. occurs when a user clicks the `Control` with a mouse

10. Assume that you have created a `Label` named `myLabel`. Which of the following sets `myLabel`'s background color to green?
    a. `myLabel = BackColor.System.Drawing.Color.Green;`
    b. `myLabel.BackColor = System.Drawing.Color.Green;`
    c. `myLabel.Green = System.DrawingColor;`
    d. `myLabel.Background = new Color.Green;`

11. What is one difference between `CheckBox` and `RadioButton` objects?
    a. `RadioButton`s descend from `ButtonBase`; `CheckBox`es do not.
    b. Only one `RadioButton` can be selected at a time.
    c. Only one `CheckBox` can appear on a `Form` at a time.
    d. `RadioButton`s cannot be placed in a `GroupBox`; `CheckBox`es can.

12. The `Checked` property of a `RadioButton` can hold the values _____ .
    a. `true` and `false`
    b. `Checked` and `Unchecked`
    c. `0` and `1`
    d. `Yes`, `No`, and `Undetermined`

13. The `Control` in which you can display a bitmap or JPEG image is a(n) _____ .
    a. `DisplayModule`
    b. `ImageHolder`
    c. `BitmapControl`
    d. `PictureBox`

14. `ListBox`, `ComboBox`, and `CheckedListBox` objects all descend from which family?
    a. `ListControl`
    b. `List`
    c. `ButtonBase`
    d. `ListBase`

15. Which of the following properties is associated with a `ListBox` but not a `Button`?
    a. `BackColor`
    b. `SelectedItem`
    c. `Location`
    d. `IsSelected`

16. With a `ListBox` you can allow the user to choose _____ .
    a. only a single option
    b. multiple selections
    c. either of these
    d. none of these

17. You can add items to a `ListBox` by using the _____ method.
    a. `AddList()`
    b. `Append()`
    c. `List()`
    d. `AddRange()`

18. A `ListBox`'s `SelectedItem` property contains _____ .
    a. the position of the currently selected item
    b. the value of the currently selected item
    c. a Boolean value indicating whether an item is currently selected
    d. a count of the number of currently selected items

19. When you create a `ListBox`, by default its `SelectionMode` is _____ .
    a. `Simple`
    b. `MultiExtended`
    c. `One`
    d. `false`

20. A horizontal list of general options that appears under the title bar of a `Form` or `Window` is a _____ .
    a. task bar
    b. subtitle bar
    c. menu strip
    d. list box

Exercises

Programming Exercises

1. Create a project named **DayNight**. Include a `Form` that contains two `Button`s, one labeled *Day* and one labeled *Night*. Add a `Label` telling the user to click a button. When the user clicks *Day*, change the `BackColor` of the `Form` to `Yellow`; when the user clicks *Night*, change the `BackColor` of the `Form` to `DarkBlue`.
2. Create a project named **FiveColors**. Its `Form` contains at least five `Button` objects, each labeled with a color. When the user clicks a `Button`, change the `BackColor` of the `Form` appropriately.
3. Create a project named **FiveColors2**. Its `Form` contains at least five `RadioButton` objects, each labeled with a color. When the user clicks a `RadioButton`, change the `BackColor` of the `Form` appropriately.
4. Create a project named **MyFlix**. Its `Form` contains a `ListBox` with the titles of at least six movies or TV shows available to purchase. Provide directions that tell users they can choose as many downloads as they want by holding down the Ctrl key while making selections. When the user clicks a `Button` to indicate the choices are final, display the total price, which is $1.99 per download. If the user selects or deselects items and clicks the button again, make sure the total is updated correctly.
5. Create a project named **FontSelector**. Its `Form` contains two `ListBox`es—one contains at least four `Font` names, and the other contains at least four `Font` sizes. Let the first item in each list be the default selection if the user fails to make a selection. Allow only one selection per `ListBox`. After the user clicks a `Button`, display *Hello* in the selected `Font` and size.
6. Create a project named **DavesDriveways** that contains a `Form` for a driveway installation company. Allow the user to choose a material (gravel, asphalt, cement, or brick) and a number of square feet. After the user makes selections, display the total price, which is $10 per square foot for gravel, $12 for asphalt, $14 for cement, and $17 for brick. Use the `Controls` that you think are best for each function. Label items appropriately, and use fonts and colors to achieve an attractive design.
7. Create a project named **VacationPlanner** for a tropical resort that offers all-inclusive vacation packages. The project contains a `Form` that allows the user to choose one option from at least three in each of the following categories—departure city, room type, and meal plan. Assign a different price to each selection, and display the total when the user clicks a `Button`. Use the `Controls` that you think are best for each function. Label items appropriately, and use fonts and colors to achieve an attractive design.

8. Create a project named **CarDealer** that contains a `Form` for an automobile dealer. Include options for at least three car models. After users make a selection, proceed to a new `Form` that contains information about the selected model. Use the `Control`s that you decide are best for each function. Label items on the `Form` appropriately, and use fonts and colors to achieve an attractive design.

9. Create a project named **AnnualBudget** that includes a `Form` with two `LinkLabel`s. One opens a spreadsheet for viewing, and the other visits your favorite Web site. Include `Label`s on the `Form` to explain each link. You can create a spreadsheet with a few numbers that represent an annual budget, or you can use the AnnualBudget.xls file in the Chapter.12 folder of the downloadable student files.

10. Create a project named **NinasCookieSource** that includes a `Form` for a company named The Cookie Source. Allow the user to select from at least three types of cookies, each with a different price per dozen. Allow the user to select one-half, one, two, or three dozen cookies. Adjust the final displayed price as the user chooses cookie types and quantities. Also allow the user to select an order date from a `MonthCalendar`. Assuming that shipping takes three days, display the estimated arrival date for the order. Include as many labels as necessary so the user understands how to use the `Form`.

11. Create a project named **MenuStripDemo2** that is based on the `MenuStripDemo` project in the Chapter.12 folder of your downloadable student files. (See Figure 12-34 earlier in this chapter). Add appropriate functionality to the currently unprogrammed menu options (the two options in the Font menu and the three options in the Color menu). Add at least three other menu options to the program, either vertically, horizontally, or both.

12. Create a project named **LetsMakeADeal**. In this game, three prizes of varying value are assigned randomly to be hidden behind three "doors" that you can implement as `Button`s. For example, the prizes might be a new car, a big-screen TV, and a live goat. The player chooses a `Button`, and then one of the two other prizes is revealed; the one revealed is never the most desirable prize. The user then has the option of changing the original selection to the remaining unseen choice. For example, consider these two game scenarios:

    - Suppose that the most valuable prize is randomly assigned to the first button. If the user chooses the first button, reveal either of the other two prizes, and ask the user if he wants to change his selection.
    - Suppose that the most valuable prize is assigned to the first button, but the user chooses the second button. Reveal the third prize so that the most valuable prize's location is still hidden, and then ask the user whether he wants to change his selection.

    After the user has chosen to retain his original selection or make a change, reveal what he has won.

Debugging Exercises

1. Each of the following projects in the Chapter.12 folder of your downloadable student files has syntax and/or logical errors. Immediately save the four project folders with their new names before starting to correct their errors. After you correct the errors, save each project using the same name preceded with *Fixed*. For example, DebugTwelve1 will become FixedDebugTwelve1.
   a. DebugTwelve1
   b. DebugTwelve2
   c. DebugTwelve3
   d. DebugTwelve4

Case Problems

1. Throughout this book, you have created programs for the Greenville Idol competition. Now create an interactive advertisement named **GreenvilleAdvertisement** that can be used to recruit contestants. Include at least three `Control`s that you studied in this chapter, and use at least two different `Font`s and two different `Color`s.
2. Throughout this book, you have created programs for Marshall's Murals. Now create an interactive advertisement named **MarshallsAdvertisement** that can be used to advertise the available murals. Include at least three `Control`s that you studied in this chapter, and use at least two different `Font`s and two different `Color`s.