WEEK 2

DAY 16

Creating Windows Forms

The base class libraries from Microsoft provide a number of classes for creating and working with forms-based windows applications, including the creation of Windows forms and controls. Today, you

- Learn how to create a Windows form
- Customize the look and feel of a form
- Ad controls to a Windows form
- Work with text boxes, labels, and more
- Customize the look of a control by setting its properties
- Associate events with a control

Working with Windows and Forms

Most operating systems today use event-driven programming and forms to interact with users. If you have done development for Microsoft Windows, you most likely used a set of routines within the Win32 libraries that helped you to

create windows and forms. Yesterday you learned about the Base Class Libraries (BCL). Within the BCL is a set of classes for doing similar windows and forms development. The benefit of the classes in the base classes is that they can be used by any of the programming languages within the framework. Additionally, they have been created to make developing forms-based applications simple.

Creating Windows Forms

To create a windows form application, you create a class that inherits from the `Form` class. The `Form` class is located within the `System.Windows.Forms` namespace. Listing 16.1 presents firstfrm.cs, which is the code required to create what is probably the most minimal windows form application.

LISTING 16.1 firstfrm.csA Simple Windows Form Application

```
 1:  // firstfrm.cs - A super simplistic windows form application
 2:  //---------------------------------------------------------------------
 3:
 4:  using System.Windows.Forms;
 5:
 6:  public class frmHelloApp : Form
 7:  {
 8:      public static void Main( string[] args )
 9:      {
10:         frmHelloApp frmHello = new frmHelloApp();
11:         Application.Run(frmHello);
12:      }
13:  }
```

As you can see, this listing is extremely short when you consider what it can do. To see what it can do though, you need to compile it. In the next section, you'll see what you need to do in order to compile this listing.

Compiling Options

Compiling Listing 16.1 needs to be done differently than you have done before. You might need to include a reference in the `compile` command to the base classes you are using, which was briefly covered yesterday.

The `Form` classes are contained within an assembly called System.Windows.Forms.dll. You might need to include a reference to this assembly when you compile the program. Just including the `using` statement at the top of a listing does not actually include any

files in your program; it provides only a reference to a point within the namespace stored in the file. As you have learned and seen, this enables you to use a shortened version of the name rather than a fully qualified name. Most of the common windows form controls and forms functionality is within this assembly.

To ensure that this assembly is used when you compile your program, you use the reference command-line parameter when you compile. This is `/reference:`*`filename`*, where filename is the name of the assembly. Using the `Forms` assembly to compile the firstfrm.cs program in Listing 16.1, you type the following command line:

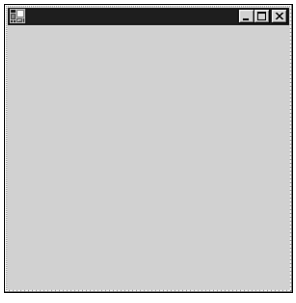

```
csc /reference:System.Windows.Forms.dll firstfrm.cs
```

Alternatively, you can shorten `/reference:` to just `/r:`. When you execute the `compile` command, your program will execute.

If you execute the firstfrm application from the command prompt, you will see the window in Figure 16.1 displayed.

FIGURE 16.1

The firstfrm application's form.



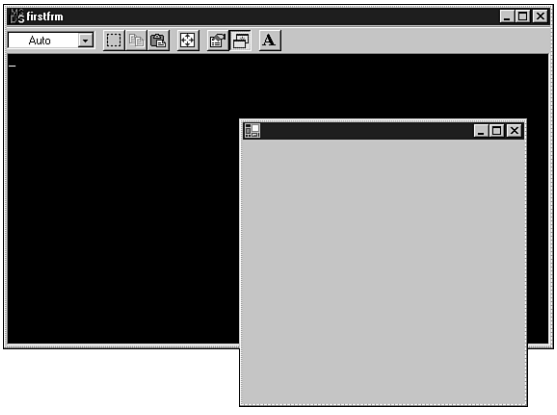
This is exactly what you want. But wait. If you run this program from directly within an operating system such as Microsoft Windows, you will notice a slightly different result. The result will be a command-line box as well as the windows form (See Figure 16.2). The command-line dialog is not something you want created.

To stop this from displaying, you need to tell the compiler that you want the program created to be targeted to a Microsoft Windows operating system. This is done using the `/target:` flag with the `winexe` option. You can use `/t:` as an abbreviation. Recompiling the firstfrm.cs program in Listing 16.1 with the following command results in the solution wanted:

```
csc /r:System.Windows.Forms.dll /t:winexe firstfrm.cs
```

When you execute the program, it does not first create a command window.

FIGURE 16.2
The actual display from the firstfrm application.



You should be aware that some of the assemblies might be automatically included when you compile. For example, development tools such as Microsoft Visual C# include a few assemblies by default. If an assembly is not included, you get an error when you compile, stating that an assembly might be missing.

Analyzing Your First Windows Form Application

Now that you can compile and execute a windows form application, you should begin understanding the code. Look back at the code in Listing 16.1.

In line 4, the listing uses the `System.Windows.Forms` namespace, which enables the `Form` and `Application` class names to be shortened. In line 6, this application is in a class called `frmHelloApp`. The new class you are creating inherits from the `Form` class, which provides all the basic functionality of a Windows form.

As you will learn in today's lesson, the `System.Windows.Forms` namespace also includes controls, events, properties, and other code that will make your windows forms more usable.

With the single line of code (line 4), you have actually created the Forms application class. In line 10, you instantiate an object from this class. In line 11, you call the `Run` method of the `Application` class. This is covered in more detail in a moment. For now, know that it causes the application to display the form and keep running until you close

the form. You could call the `Show` method of the `Form` class instead by replacing line 11 with the following:

```
frmHello.Show();
```

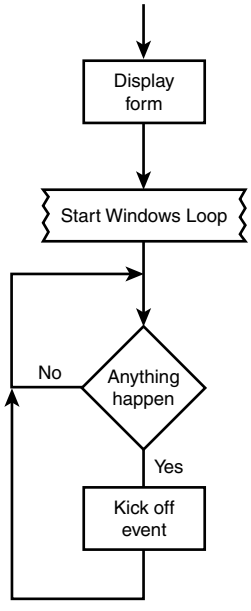
Although this seems more straightforward, you will find the application ends with a flaw. The program shows the form and then moves on to the next line, which is the end of the program. Because the end of the program is reached, the processing ends and the form closes. This is not the result you want. The `Application` class gets around this problem.

Later today, you will learn about a form method that will display a form and wait.

The `Application.Run` Method

A windows application is an event-driven program that will generally display a form containing controls. The program then spins in a loop until the user does something on the form or within the windowed environment. Messages are created whenever something occurs. These messages cause an event to occur. If there is an event handler for a given message, it will be executed. If there is not, the loop will continue. Figure 16.3 illustrates this looping.

FIGURE 16.3
Flow of a standard windows program.



As you can see, the loop never seems to end. Actually, an event can end the program. The basic form that you inherit from (`Form`) includes the close button as well as a close item in the Command menu. These controls can kick off an event that closes the form and ends the loop.

By now you should be guessing what the `Application` class does for you—or more specifically what the `Application` class's `Run` method does for you. The `Run` method takes care of creating the loop and keeping the program running until an event that ends the program loop is executed. In the case of Listing 16.1, selecting the close button on the form or selecting the Close option on the command menu causes an event to be fired that will end the loop and thus close the form.

The `Application.Run` method also displays a form for you. Line 11 of Listing 16.1 receives a form object—`frmHello`. This is an object derived from the `Form` class. The `Application.Run` method displays this form and then loops.

The loop created by the `Application` class's `Run` method actually processes messages that are created. These messages can be created by the operating system, your application, or other applications that are running. The loop will process these methods. For example, when you click a button, there will be a number of messages created. This will include messages for a mouse down, mouse up, button click, and more. If a message matches with an event handler, the event handler will be executed. If no event handler is defined, the message is ignored.

Customizing a Form's Look and Feel

In the previous listing, you saw a basic form presented. There are a number of properties, methods, and events associated with the `Form` class—too many to cover in this book. However, it is worth touching on a few of them. You can check the online documentation for a complete accounting of all the functionality available with this class.

Caption Bar on a Form

Listing 16.1 presented a basic, blank form. The next few listings continue to work with this blank form; however, with each listing in today's lesson, you will learn to take a little more control of the form.

The form from Listing 16.1 comes with a number of items already available, including the control menu and the minimize, maximize, and close buttons. You can control whether these features are on or off with your forms by setting properties:

`ControlBox`	Determines whether the control box is displayed.
`HelpButton`	Indicates whether a help button is displayed on the caption of the form. This will be displayed only if both the `MaximizeBox` and `MinimizeBox` values are false.
`MaximizeBox`	Indicates whether the maximum button is included.
`MinimizeBox`	Indicates whether the minimize button is included.
`Text`	The `Caption` for the form.

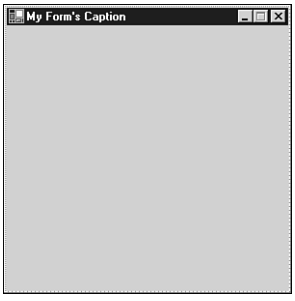
Some of these values impact others. For example, the `HelpButton` will display only if both the `MaximizeBox` and `MinimizeBox` properties are false (turned off). Listing 16.2 gives you a short listing that enables you to play with these values; Figure 16.4 shows the output. Enter this listing, compile it, and run it. Remember to include the `/t:winexe` flag when compiling.

LISTING 16.2 form2.cs—Sizing a Form

```
 1:  // form2.cs - Caption Bar properties
 2:  //---------------------------------------------------------------------
 3:
 4:  using System.Windows.Forms;
 5:
 6:  public class frmHelloApp : Form
 7:  {
 8:      public static void Main( string[] args )
 9:      {
10:         frmHelloApp frmHello = new frmHelloApp();
11:
12:         // Caption bar properties
13:         frmHello.MinimizeBox = true;
14:         frmHello.MaximizeBox = false;
15:         frmHello.HelpButton = true;
16:         frmHello.ControlBox = true;
17:         frmHello.Text = @"My Form's Caption";
18:
19:         Application.Run(frmHello);
20:      }
21:  }
```

OUTPUT

FIGURE 16.4
Output for Listing 16.2.



ANALYSIS This listing is easy to follow. In line 6, a new class is created called `frmHelloApp` that inherits from the `Form` class. In line 10, a new form object is instantiated from the `Application` class. This form has a number of caption bar values set in lines 13 to 17. In line 19, the `Run` method of the `Application` class is called to display the form. You should look at the output in Figure 16.4. Both the Maximize and Minimize buttons are displayed; however, the Maximize button is to be active. This is because you set it to `false` in line 14. If you set both values to `false`, neither button will show.

You should also notice that the Help button is turned to true in line 15. The Help button displays only if both the Minimize and Maximize buttons are turned off (false). This means that line 15 is ignored. Change the property in line 13 so that the resulting properties in lines 14 to 16 are the following:

```
13:         frmHello.MinimizeBox = false;
14:         frmHello.MaximizeBox = false;
15:         frmHello.HelpButton = true;
16:         frmHello.ControlBox = true;
```

Recompile and run this program. The new output will be Figure 16.5.

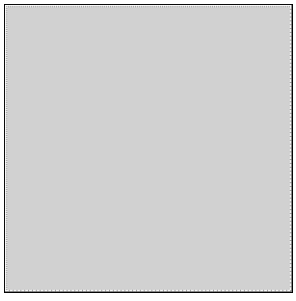
FIGURE 16.5
Output with a help button.

As you can see, the output reflects the values that have been set.

One additional combination is worth noting. When you set `ControlBox` to `false`, the Close button and the Control Box are both hidden. Additionally, if `ControlBox`, `MinimizeBox`, and `MaximizeBox` are all set to false and if there is no text for the caption, the caption bar will be completely gone. Remove line 17 from Listing 16.2 and set the values for the properties in lines 13 to 16 to false. Recompile and run the program. The output you will receive is displayed in Figure 16.6.

You might wonder why you would want to remove the caption bar. One possible reason is to display a splash screen. You'll learn more about creating a splash screen later.

FIGURE 16.6
Output without the caption bar.



In Microsoft Windows, Alt+F4 closes the current window. If you disable the Control Box, you end up removing the close button as well. You'll need Alt+F4 to close the window!

The Size of a Form

The next thing to take control of is the form's size. There are a number of methods and properties that can be used to manipulate the form's shape and size. Table 16.1 presents the ones used here.

TABLE 16.1 Sizing Functionality in the `Form` Class

`AutoScale`	The form automatically adjusts itself, based on the font and/or controls used on it.
`AutoScaleBaseSize`	The base size used for autoscaling the form.
`AutoScroll`	The form will have the automatic capability of scrolling.
`AutoScrollMargin`	The size of the margin for the auto-scroll.
`AutoScrollMinSize`	The minimum size of the auto-scroll.

TABLE 16.1 continued

`AutoScrollPosition`	The location of the auto-scroll position.
`ClientSize`	The size of the client area of the form.
`DefaultSize`	The protected property that sets the default size of the form.
`DesktopBounds`	The size and location of the form.
`DesktopLocation`	The location of the form.
`Height`	The height of the form
`MaximizeSize`	The maximum size for the form.
`MinimizeSize`	The minimum size for the form.
`Size`	The size of the form. `set` or `get` a `Size` object that contains an `x`, `y` value.
`SizeGripStyle`	The style of the size grip used on the form. A value from the `SizeGripStyle` enumerator. Values are `Auto` (automatically displayed when needed), `Hide` (hidden), `Show` (always shown).
`StartPosition`	The starting position of the form. This is a value from the `FormStartPosition` enumerator. Possible `FormStartPosition` enumeration values are `CenterParent` (centered within the parent form), `CenterScreen` (centered in the current display screen), `Manual` (location and size determined by starting position), `WindowsDefaultBounds` (Positioned at the default location), and `WindowsDefaultLocation` (positioned at the default location, with dimensions based on specified values for the size).
`Width`	The width of the form

The items listed in Table 16.1 are only a few of the methods and properties available that work with a form's size. Listing 16.3 presents some of these in another simple application; Figure 16.7 shows the output.

LISTING 16.3 form3.cs—Sizing a Form

```
1:  // form3.cs - Form Size
2:  //---------------------------------------------------------------------
3:
4:  using System.Windows.Forms;
```

LISTING 16.3 continued

```
 5:  using System.Drawing;
 6:
 7:  public class frmHelloApp : Form
 8:  {
 9:      public static void Main( string[] args )
10:      {
11:         frmHelloApp myForm = new frmHelloApp();
12:         myForm.Text = "Form Sizing";
13:
14:         myForm.Width = 400;
15:         myForm.Height = 100;
16:
17:         Point FormLoc = new Point(200,350);
18:         myForm.StartPosition = FormStartPosition.Manual;
19:         myForm.DesktopLocation = FormLoc;
20:
21:
22:         Application.Run(myForm);
23:      }
24:  }
```


OUTPUT

FIGURE 16.7

Positioning and sizing the form.

ANALYSIS Setting the size of a form is simple. Lines 14 and 15 set the size of the form in Listing 16.3. As you can see, the `Width` and `Height` properties can be set. You can also set both of these at the same time by using a `Size` object.

Positioning the form takes a little more effort. In line 17, a `Point` object is created that contains the location on the screen that you want the form positioned. This is then used in line 19 by applying it to the `DesktopLocation` property. To use the `Point` object without fully qualifying its name, you need to include the `System.Drawing` namespace, as in line 5.

In line 18, you see that an additional property has been set. If you leave line 18 out, you will not get the results you want. You must set the starting position for the form by setting the `StartPosition` property to a value in the `FormStartPosition` enumerator. Table 16.1 contained the possible values for this enumerator. You should note the other values for `FormStartPosition`. If you want to center a form on the screen, you can replace lines 17 to 19 with one line:

```
myForm.StartPosition = FormStartPosition.CenterScreen;
```

This single line of code takes care of centering the form on the screen regardless of the screen's resolution.

Colors and Background of a Form

Working with the background color of a form requires setting the `BackColor` property to a color value. The color values can be taken from the `Color` structure located in the `System.Drawing` namespace. Table 16.2 lists some of the common colors.

To set a color is as simple as assigning a value from Table 16.2:

```
myForm.BackColor = Color.HotPink;
```

Of equal value to setting the form's color is to place a background image on the form. An image can be set into the form's `BackgroundImage` property. Listing 16.4 sets an image onto the background; Figure 16.8 shows the output. The image placed is passed as a parameter to the program.

Be careful with this listing. For brevity, it does not contain exception handling. If you pass a filename that doesn't exist, the program will throw an exception.

TABLE 16.2 Colors

AliceBlue	AntiqueWhite	Aqua	Aquamarine	Azure	Beige
Bisque	Black	BlanchedAlmond	Blue	BlueViolet	Brown
BurlyWood	CadetBlue	Chartreuse	Chocolate	Coral	CornflowerBlue
Cornsilk	Crimson	Cyan	DarkBlue	DarkCyan	DarkGoldenrod
DarkGray	DarkGreen	DarkKhaki	DarkMagenta	DarkOliveGreen	DarkOrange
DarkOrchid	DarkRed	DarkSalmon	DarkSeaGreen	DarkSlateBlue	DarkSlateGray
DarkTurquoise	DarkViolet	DeepPink	DeepSkyBlue	DimGray	DodgerBlue
Firebrick	FloralWhite	ForestGreen	Fuchsia	Gainsboro	GhostWhite
Gold	Goldenrod	Gray	Green	GreenYellow	Honeydew
HotPink	IndianRed	Indigo	Ivory	Khaki	Lavender
LavenderBlush	LawnGreen	LemonChiffon	LightBlue	LightCoral	LightCyan
LightGoldenrodYellow	LightGray	LightGreen	LightPink	LightSalmon	LightSeaGreen
LightSkyBlue	LightSlateGray	LightSteelBlue	LightYellow	Lime	LimeGreen
Linen	Magenta	Maroon	MediumAquamarine	MediumBlue	MediumOrchid
MediumPurple	MediumSeaGreen	MediumSlateBlue	MediumSpringGreen	MediumTurquoise	MediumVioletRed
MidnightBlue	MintCream	MistyRose	Moccasin	NavajoWhite	Navy
OldLace	Olive	OliveDrab	Orange	OrangeRed	Orchid
PaleGoldenrod	PaleGreen	PaleTurquoise	PaleVioletRed	PapayaWhip	PeachPuff
Peru	Pink	Plum	PowderBlue	Purple	Red
RosyBrown	RoyalBlue	SaddleBrown	Salmon	SandyBrown	SeaGreen
SeaShell	Sienna	Silver	SkyBlue	SlateBlue	SlateGray
Snow	SpringGreen	SteelBlue	Tan	Teal	Thistle
Tomato	Transparent	Turquoise	Violet	Wheat	White
WhiteSmokeYellow	YellowGreen				

LISTING 16.4 form4.cs—Using Background Images

```
 1:  // form4.cs - Form Backgrounds
 2:  //---------------------------------------------------------------------
 3:
 4:  using System.Windows.Forms;
 5:  using System.Drawing;
 6:
 7:  public class frmApp : Form
 8:  {
 9:      public static void Main( string[] args )
10:      {
11:         frmApp myForm = new frmApp();
12:         myForm.BackColor = Color.HotPink;
13:         myForm.Text = "Form4 - Backgrounds";
14:
15:         if (args.Length >= 1)
16:         {
17:            myForm.BackgroundImage = Image.FromFile(args[0]);
18:
19:            Size tmpSize = new Size();
20:            tmpSize.Width = myForm.BackgroundImage.Width;
21:            tmpSize.Height = myForm.BackgroundImage.Height;
22:            myForm.ClientSize = tmpSize;
23:
24:            myForm.Text = "Form4 - " + args[0];
25:         }
26:
27:         Application.Run(myForm);
28:      }
29:  }
```

OUTPUT

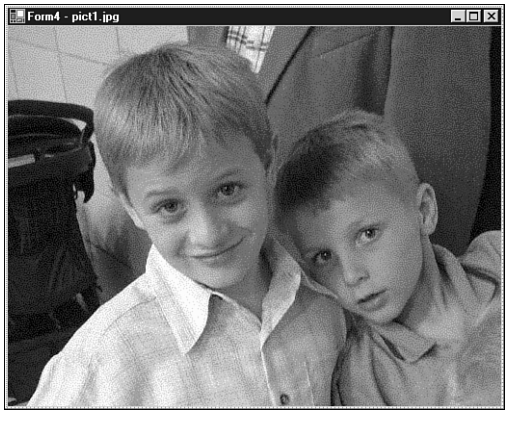


FIGURE 16.8
Using a background image.

ANALYSIS This program presents an image on the form background. This image is provided on the command line. If no image is entered on the command line, the background color is set to Hot Pink. I ran the listing using a picture of my nephews. The command line I entered was:

```
form4 pict1.jpg
```

The pict1.jpg was in the same directory as the form4 executable. If it was in a different directory, I would have needed to enter the full path. You can pass a different image as long as the path is valid. If you enter an invalid filename, you get an exception.

Looking at the listing, you can see that to create an application to display images is extremely easy. The framework classes take care of all the difficult work for you. In line 12, the background color was set to be Hot Pink. This is done by setting the form's `BackColor` property with a color value from the `Color` structure.

In line 15, a check is done to see whether a value was included on the command line. If a value was not included, lines 17 to 24 are skipped and the form is displayed with a hot pink background. If a value was entered, this programming makes the assumption (which your programs should not do) that the parameter passed was a valid graphics file. This file is then set into the `BackgroundImage` property of the form. The filename needs to be converted to an actual image for the background by using the `Image` class. More specifically, the `Image` class includes a `static` method, `FromFile`, that will take a filename as an argument and return an `Image`. This is exactly what is needed for this listing.

If you want a specific image for your background, you could get rid of the if statement and replace line 17's arg[0] value with the hard-coded name of the file you want as the background.

The `BackgroundImage` property holds an `Image` value. Because of this, properties and methods from the `Image` class can be used on this property. The `Image` class includes `Width` and `Height` properties that are equal to the width and height of the image contained. Lines 20 and 21 use these values to a temporary Size variable that will in turn be assigned to the form's client size in line 22. The size of the form's client area is set to the same size as the image. The end result is that the form displayed will always display the full image. If you don't do this, you will see either only part of the image or tiled copies of the image.

Borders

Controlling the border will not only impact the look of the form, but will also determine whether the form can be resized. To modify the border, you set the `Form` class's `BorderStyle` property with a value from the `FormBorderStyle` enumeration. Possible values for the `BorderStyle` property are listed in Table 16.3. Listing 16.5 presents a form with the border modified; Figure 16.9 shows the output.

TABLE 16.3 `FormBorderStyle` Enumerator Values

Value	*Description*
`Fixed3D`	Fixed, 3D border
`FixedDialog`	Fixed, thick border
`FixedSingle`	Fixed, single-line border
`FixedToolWindow`	Non-resizable, tool window border
`None`	No border
`Sizeable`	Resizable
`SizeableToolWindow`	Resizable tool window border

LISTING 16.5 border.cs—Modifying a Form's Border

```
 1:  // border.cs - Form Borders
 2:  //-------------------------------------------------------------
 3:
 4:  using System.Windows.Forms;
 5:  using System.Drawing;
 6:
 7:  public class frmApp : Form
 8:  {
 9:      public static void Main( string[] args )
10:      {
11:         frmApp myForm = new frmApp();
12:         myForm.BackColor = Color.SteelBlue;
13:         myForm.Text = "Borders";
14:
15:         myForm.FormBorderStyle = FormBorderStyle.Fixed3D;
16:
17:         Application.Run(myForm);
18:      }
19:  }
```

OUTPUT

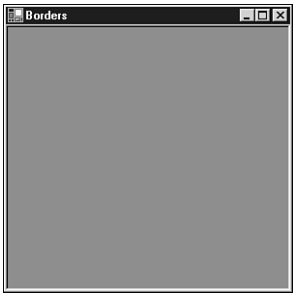


FIGURE 16.9
Modifying a form's border.

ANALYSIS As you can see, the border is fixed in size. If you try to resize the form at runtime you will not be able to.

If you do make the form resizable, you have another option you can set as well: `SizeGripStyle`. `SizeGripStyle` determines whether the form will be marked with a resize indicator. Figure 16.10 has the resize indicator circled. You can set your form to automatically show this indicator or to always hide or always show it. This is done using one of three values in the `SizeGripStyle` enumerator: `Auto`, `Hide`, or `Show`. The indicator in Figure 16.10 was shown by including the line

```
myForm.SizeGripStyle = SizeGripStyle.Show;
```

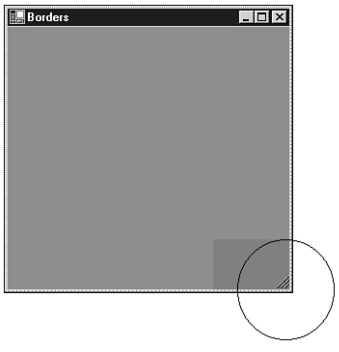


FIGURE 16.10
The size grip.

Caution

Don't get confused by using conflicting properties. For example, if you use a fixed-sized border and you set the size grip to display, your results will not match these settings. The fixed border means that the form cannot be resized; therefore, the size grip will not display regardless of how you set it.

Adding Controls to a Form

Up to this point, you have been working with the look and feel of a form; however, without controls a form is virtually worthless. Controls make a windows application usable.

A control can be a button, a list box, a text box, an image, or even simple plain text being displayed. The easiest way to add such controls is to use a graphical development tool such as Microsoft's Visual C#. A graphical tool enables you to drag and drop controls onto a form. It also adds all the basic code needed to display the control.

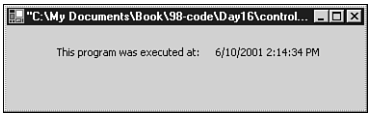
A graphical development tool, however, is not needed. Even if you use a graphical tool, it is still valuable to understand what the tool is doing for you. Some of the standard controls provided in the framework are listed in Table 16.4. Additional controls can be created and used as well.

TABLE 16.4 Some Standard Controls in the Base Class Libraries

`Button`	`CheckBox`	`CheckedListBox`	`ComboBox`
`ContainerControl`	`DataGrid`	`DateTimePicker`	`DomainUpDown`
`Form`	`GroupBox`	`HScrollBar`	`ImageList`
`Label`	`LinkLabel`	`ListBox`	`ListView`
`MonthCalendar`	`NumericUpDown`	`Panel`	`PictureBox`
`PrintReviewControl`	`ProgressBar`	`PropertyGrid`	`RadioButton`
`RichTextBox`	`ScrollableControl`	`Splitter`	`StatusBar`
`StatusBarPanel`	`TabControl`	`TabPage`	`TabStrip`
`TextBox`	`Timer`	`ToolBar`	`ToolBarButton`
`ToolTip`	`TrackBar`	`TreeView`	`VScrollBar`
`UserControl`			

FIGURE 16.11

The control architecture in the .NET classes.



The controls in Table 16.4 are defined in the `System.Windows.Forms` namespace. The following sections cover some of these controls. Be aware, however, that the coverage here is very minimal. There are hundreds of properties, methods, and events associated with the controls listed in Table 16.4. It would take a book bigger than this one to cover

all the details of each control. Here, you will learn how to use some of the key controls. The process of using the other controls will be very similar to those presented here. Additionally, you will see only a few of the properties. All the properties can be found in the help documentation available with the C# compiler or with your development tool.

Working with Labels and Text Display

You use the `Label` control to display simple text on the screen. The `Label` control is in the `System.Windows.Forms` namespace with the other built-in controls.

To add a control to a form, you first create the control. Then you can customize the control via its properties and methods. When you have made the changes you want, you can then add it to your form.

NEW TERM A *label* is a control that displays information to the user, but does not allow the user to directly change its values. You create a label like any other object:

```
Label myLabel = new Label();
```

After it's created, you have an empty label that can be added to your form. Listing 16.6 illustrates a few of the label's properties, including setting the textual value with the `Text` property; Figure 16.11 shows the output. To add the control to your form, you use the `Add` method with the `Controls` property of your form. Simply shown, to add the `myLabel` control to the `myForm` you've used before, you use

```
myForm.Controls.Add(myLabel);
```

To add other controls, you replace `myLabel` with the control's name.

LISTING 16.6 Control1.cs—Using a `Label` Control

```
 1:  // control1.cs - Working with controls
 2:  //---------------------------------------------------------------------
 3:
 4:  using System;
 5:  using System.Windows.Forms;
 6:  using System.Drawing;
 7:
 8:  public class frmApp : Form
 9:  {
10:      public static void Main( string[] args )
11:      {
12:         frmApp myForm = new frmApp();
13:
14:         myForm.Text = Environment.CommandLine;
15:         myForm.StartPosition = FormStartPosition.CenterScreen;
16:
```

LISTING 16.6 continued

```
17:         // Create the controls...
18:         Label myDateLabel = new Label();
19:         Label myLabel = new Label();
20:
21:         myLabel.Text = "This program was executed at:";
22:         myLabel.AutoSize = true;
23:         myLabel.Left = 50;
24:         myLabel.Top = 20;
25:
26:         DateTime currDate = DateTime.Now;;
27:         myDateLabel.Text = currDate.ToString();
28:
29:         myDateLabel.AutoSize = true;
30:         myDateLabel.Left = 50 + myLabel.PreferredWidth + 10;
31:         myDateLabel.Top = 20;
32:
33:         myForm.Width = myLabel.PreferredWidth + myDateLabel.PreferredWidth +
➥110;
34:         myForm.Height = myLabel.PreferredHeight+ 100;
35:
36:         // Add the control to the form...
37:         myForm.Controls.Add(myDateLabel);
38:         myForm.Controls.Add(myLabel);
39:
40:         Application.Run(myForm);
41:      }
42:  }
```

OUTPUT

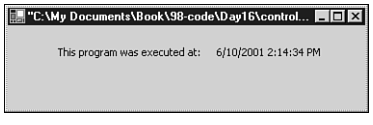


FIGURE 16.11

Using a `Label` *control.*

ANALYSIS This program creates two label controls and displays them in your form. Rather than just plopping the labels anywhere, this listing positions them somewhat centered in the form.

Stepping back, you can see that the program starts by creating a new form in line 12. The title on the control bar for the form is set equal to the command-line value from the `Environment` class. If you remember this from yesterday, you will know that this is the program name along with the complete path. In line 15, the `StartPosition` property for the form is set to center the form on the screen. At this point, no size for the form has been indicated. That will be done in a minute.

In lines 18 and 19, two label controls are created. The first label, `myDateLabel`, will be used to hold the current date and time. The second label control will be used to hold descriptive text. Recall that a label is a control that displays information to the user, but does not allow the user to directly change its values—so these two uses of a label are appropriate.

In lines 21 to 24, properties for the `myLabel` label are set. In line 21, the text to be displayed is assigned to the `Text` property of the label. In line 22, the `AutoSize` property is set to `true`. You can control the size of the label or you can let it determine the best size for itself. Setting the `AutoSize` property to `true` gives the label the ability to resize itself. In lines 23 and 24, the `Left` and `Top` properties are set to values. These values are for the location on the form that the control should be placed. In this case, the `myLabel` control will be placed 50 pixels from the left side of the form and 20 pixels down into the client area of the form.

The next two lines of the listing (26 and 27) are a roundabout way to assign the current date and time to the `Text` property of your other label control, `myDateLabel`. As you can see, a `DateTime` object is created and assigned the value of `Now`. This value is then converted to a string and assigned to the `myDateLabel`.

In line 29, the `AutoSize` property for the `myDateLabel` is also set to `true` so that the label will be displayed in an appropriately sized manner. In lines 30 and 31, the position of the `myDateLabel` are set. The `Top` position is easy to understand—it will be at the same vertical location as the other label—but the `Left` position is a little more complex. The `myDateLabel` label is to be placed to the right of the other label. To place it to the right of the other label, you need to move it over a distance equal to the size of the other label plus any offset from the edge of the window to the other label. This would be 50 plus the width of the `myLabel` label. Because you have said to auto size your labels, the width will be equal to the preferred width. A label's preferred width can be obtained from the `PreferredWidth` property of the control. The end result is that to place the `myDateLabel` to the right of `myLabel`, you add the preferred width of `myLabel` plus the offset added to `myLabel`. To add a little buffer between the two labels, an additional 10 pixels are added. Figure 16.12 helps illustrate what is happening in line 30.

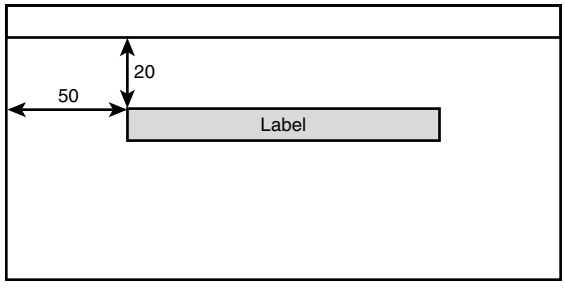


FIGURE 16.12
Positioning of the `Label`.

Lines 33 and 34 set the width and height of the form. As you can see, the Width is set to center the labels on the form. This is done by balancing the offsets and using the widths of the two labels. The height is set to make sure there is a lot of space around the text.

In lines 37 and 38, you see that adding these controls to the form is a simple call. The `Add` method of the `Controls` property is called for each of the controls. The `Run` method of the `Application` is then executed in line 40 so that the form is displayed. The end result is that you now have text displayed on your form!

For the most part, this same process is used for all other types of control. This involves creating the control, setting its properties, and then adding it to the form.

A Suggested Approach for Using Controls

The process presented in the previous section is appropriate for using controls. The most common development tool for creating windowed applications is expected to be Microsoft Visual Studio .NET, and thus Microsoft Visual C# for C# applications. This development tool provides a unique structure to programming controls. Although not necessary, this structure does organize the code so that the graphical design tools can better follow the code. Because the amount of effort to follow this approach is minimal, it is worth considering. Listing 16.7 represents Listing 16.6 in this slightly altered structure. This structure is similar to what is generated by Microsoft Visual C#.

LISTING 16.7 ctrl1b.cs—Structuring Your Code for Integrated Development Environments

```
 1:  // cntrl1b.cs - Working with controls
 2:  //---------------------------------------------------------------------
 3:
 4:  using System;
 5:  using System.Windows.Forms;
 6:  using System.Drawing;
 7:
 8:  public class frmApp : Form
 9:  {
10:     public frmApp()
11:     {
12:        InitializeComponent();
13:     }
14:
15:     private void InitializeComponent()
16:     {
17:         this.Text = Environment.CommandLine;
18:         this.StartPosition = FormStartPosition.CenterScreen;
19:
20:         // Create the controls...
```

LISTING 16.7 continued

```
21:         Label myDateLabel = new Label();
22:         Label myLabel = new Label();
23:
24:         myLabel.Text = "This program was executed at:";
25:         myLabel.AutoSize = true;
26:         myLabel.Left = 50;
27:         myLabel.Top = 20;
28:
29:         DateTime currDate = new DateTime();
30:         currDate = DateTime.Now;
31:         myDateLabel.Text = currDate.ToString();
32:
33:         myDateLabel.AutoSize = true;
34:         myDateLabel.Left = 50 + myLabel.PreferredWidth + 10;
35:         myDateLabel.Top = 20;
36:
37:         this.Width = myLabel.PreferredWidth + myDateLabel.PreferredWidth +
➥110;
38:         this.Height = myLabel.PreferredHeight+ 100;
39:
40:         // Add the control to the form...
41:         this.Controls.Add(myDateLabel);
42:         this.Controls.Add(myLabel);
43:     }
44:
45:     public static void Main( string[] args )
46:     {
47:        Application.Run( new frmApp() );
48:     }
49:  }
```

ANALYSIS The output for this listing is identical to that shown in Figure 16.12 for the previous listing. This listing illustrates a different structure for coding. Again, I include this listing and analysis so you won't be surprised if you use a tool such as Visual C# and see that it followed a different structure than what I had previously presented.

Looking at this listing, you can see that the code is broken into a couple of methods instead of being placed into the `Main` method. Additionally, you can see that rather than declaring a specific instance of a form, an instance is created at the same time the `Application.Run` method is called.

When this application is executed, the `Main` method in lines 45 to 48 is executed first. This method has one line of code that creates a new `frmApp` instance and passes it to the `Application.Run` method. This one line of code kicks off a series of other activities. The

first thing to happen is that the `frmApp` constructor is called to create the new `frmApp`. A constructor has been included in lines 10 to 13 of the listing. The constructor again has one simple call, `InitializeComponent`. This call causes the code in lines 17 to 43 to execute. This is the same code that you saw earlier, with one minor exception. Instead of using the name of the form, you use the `this` keyword. Because you are working within an instance of a form, `this` refers to the current form. Everywhere you referred to the `myForm` instance in the previous listing, you now refer to `this`. When the initialization of the form items is completed, control goes back to the constructor, which is also complete. Control is therefore passed back to `Main`, which then passes the newly initialized `frmApp` object to the `Application.Run` method. This displays the form and takes care of the windows looping until the program ends.

The nice thing about this structure is that it moves all your component and form initialization into one method that is separate from a lot of your other programming logic. In larger programs, you will find this more beneficial.

Working Buttons

One of the most common controls used in windows applications are buttons. Buttons can be created using the—you guessed it—`Button` class! Buttons differ from labels, so you will most likely want an action to occur when the user clicks on a button.

Before jumping into creating button actions, it is worth taking a minute to cover creating and drawing buttons. As with labels, the first step to using a button is to instantiate a button object with the class:

```
Button myButton = new Button();
```

After you've created the button object, you can then set properties to customize it to the look and feel you want. As with the `Label` control, there are too many properties, data members, and methods to list here. You can get the complete list from the help documents. Table 16.5 lists a few of the properties.

TABLE 16.5 A Few Button Properties

Property	*Description*
`BackColor`	Returns or sets the background color of the button.
`BackgroundImage`	Returns or sets an image that will display on the button's background.
`Bottom`	Returns the distance between the bottom of the button and the top of the container the button resides in.
`Enabled`	Returns or sets a value indicating whether the control is enabled.
`Height`	Returns or sets a value indicating the height of the button.

TABLE 16.5 continued

Property	*Description*
`Image`	Returns or sets an image on the button.
`Left`	Returns or sets the position of the left side of the button.
`Right`	Returns or sets the position of the right side of the button.
`Text`	Returns or sets the text on the button.
`TextAlign`	Returns or sets the button's text alignment.
`Top`	Returns or sets a value indicating the location of the top of the button.
`Visible`	Returns or sets a value indicating whether the button is visible.
`Width`	Returns or sets the width of the button.

Take a close look at the properties in Table 16.5. These should look like some of the same properties you used with `Label`. There is a good reason for this similarity. All the controls inherit from a more general `Control` class. This class enables all the controls to use the same methods or the same names to do similar tasks. For example, `Top` is the property for the top of a control regardless of whether it is a button, text, or something else.

Button Events

Recall that buttons differ from labels; you generally use a button to cause an action to occur. When the user clicks on a button, you want something to happen. To cause the action to occur, you use events.

After you create a button, you can associate one or more events to it. This is done in the same manner that you learned on Day 14, "Indexers, Delegates, and Events." First, you create a method to handle the event, which will be called when the event occurs. As you learned on Day 14, this method must take two parameters, the object that caused the event and a `System.EventArgs` variable. This method must also be protected and of type `void`. The format is

```
protected void methodName( object sender, System.EventArgs args )
```

When working with windows, you generally name the method based on what control caused the event followed by what event occurred. For example, if button ABC was clicked, the method name for the handler could be `ABC_Click`.

To activate the event, you need to associate it to the appropriate delegate. A delegate object called `System.EventHandler` takes care of all the windows events. By associating your event handlers to this delegate object, they will be called when appropriate. The format is

```
ControlName.Event += new System.EventHandler(this.methodName);
```

where *`ControlName.Event`* is the name of the control and the name of the event for the control. `this` is the current form, and *`methodName`* is the method that will handle the event (as mentioned previously).

Listing 16.8 presents a modified version of Listing 16.7; Figure 16.13 shows the output. You will see that the date and time are still displayed in the form. You will also see, however, that a button has been added. When the button is clicked, an event fires that will update the date and time. Additionally, four other event handlers have been added to this listing for fun. These events are kicked off whenever the mouse moves over or leaves either of the two controls.

LISTING 16.8 button1.cs—Using Buttons and Events

```
 1:  // button1.cs - Working with buttons and events
 2:  //-------------------------------------------------------------
 3:
 4:  using System;
 5:  using System.Windows.Forms;
 6:  using System.Drawing;
 7:
 8:  public class frmApp : Form
 9:  {
10:     private Label myDateLabel;
11:     private Button btnUpdate;
12:
13:     public frmApp()
14:     {
15:        InitializeComponent();
16:     }
17:
18:     private void InitializeComponent()
19:     {
20:         this.Text = Environment.CommandLine;
21:         this.StartPosition = FormStartPosition.CenterScreen;
22:         this.FormBorderStyle = FormBorderStyle.Fixed3D;
23:
24:         myDateLabel = new Label();      // Create label
25:
26:         DateTime currDate = new DateTime();
27:         currDate = DateTime.Now;
28:         myDateLabel.Text = currDate.ToString();
29:
```

LISTING 16.8 continued

```
30:         myDateLabel.AutoSize = true;
31:         myDateLabel.Location = new Point( 50, 20);
32:         myDateLabel.BackColor = this.BackColor;
33:
34:         this.Controls.Add(myDateLabel);  // Add label to form
35:
36:         // Set width of form based on Label's width
37:         this.Width = (myDateLabel.PreferredWidth + 100);
38:
39:         btnUpdate = new Button();    // Create a button
40:
41:         btnUpdate.Text = "Update";
42:         btnUpdate.BackColor = Color.LightGray;
43:         btnUpdate.Location = new Point(((this.Width/2) - (btnUpdate.Width /
2)),
44:                                        (this.Height - 75));
45:
46:         this.Controls.Add(btnUpdate);  // Add button to form
47:
48:         // Add a click event handler using the default event handler
49:         btnUpdate.Click += new System.EventHandler(this.btnUpdate_Click);
50:         btnUpdate.MouseEnter += new
➥System.EventHandler(this.btnUpdate_MouseEnter);
51:         btnUpdate.MouseLeave += new
➥System.EventHandler(this.btnUpdate_MouseLeave);
52:
53:         myDateLabel.MouseEnter += new
➥System.EventHandler(this.myDataLabel_MouseEnter);
54:         myDateLabel.MouseLeave += new
➥System.EventHandler(this.myDataLabel_MouseLeave);
55:     }
56:
57:     protected void btnUpdate_Click( object sender, System.EventArgs e)
58:     {
59:         DateTime currDate =DateTime.Now ;
60:         this.myDateLabel.Text = currDate.ToString();
61:     }
62:
63:
64:     protected void btnUpdate_MouseEnter( object sender, System.EventArgs e)
65:     {
66:         this.BackColor = Color.HotPink;
67:     }
68:
69:     protected void btnUpdate_MouseLeave( object sender, System.EventArgs e)
70:     {
71:         this.BackColor = Color.Blue;
72:     }
73:
```

LISTING 16.8 continued

```
74:     protected void myDataLabel_MouseEnter( object sender, System.EventArgs
➥e)
75:     {
76:         this.BackColor = Color.Yellow;
77:     }
78:
79:     protected void myDataLabel_MouseLeave( object sender, System.EventArgs
➥e)
80:     {
81:         this.BackColor = Color.Green;
82:     }
83:
84:
85:     public static void Main( string[] args )
86:     {
87:        Application.Run( new frmApp() );
88:     }
89: }
```

OUTPUT

FIGURE 16.13

Using a button and events.

ANALYSIS This listing uses the windows designer format even though a designer was not used. This is a good way to format your code, so I follow the format here.

You will notice that I made a change to the previous listing. In lines 10 and 11, the label and button are declared as members of the form rather than members of a method. This enables all the methods within the form's class to use these two variables. They are `private`, so only this class can use them.

The `Main` method and the constructor are no different from the previous listing. The `InitializeComponent` method has changed substantially; however, most of the changes are easy to understand. Line 31 offers the first new item. Instead of using the `Top` and `Left` properties to set the location of the `myDateLabel` control, a `Point` object was used. This `Point` object was created with the value `(50, 20)` and immediately assigned to the `Location` property of the label.

You might find that creating an object and immediately assigning it can be easier to follow than doing multiple assignments. Either method works. Use whichever you are most comfortable with or whichever is easiest to understand.

In line 39, a button called `btnUpdate` is created. It is then customized by assigning values to several properties. Don't be confused by the calculations in lines 43 and 44. This is just like line 31, except that instead of using literals, calculations are used. Also keep in mind that `this` is the form, so `this.Width` is the width of the form.

Line 46 adds the button to the form. As you can see, this is done exactly the same way that any other control would be added to the form.

In lines 49 to 54, you see the fun part of this listing. These lines are assigning handlers to various events. On the left side of these assignments, you see the controls and one of their events. This event is assigned to the method name that is being passed to the `System.EventHandler`. For example, in line 49, the `btnUpdate_Click` method is being assigned to the `Click` event of the `btnUpdate` button. In lines 50 and 51, events are being assigned to the `MouseEnter` and `MouseLeave` events of `btnUpdate`. Lines 53 and 54 assign events to the `MouseEnter` and `MouseLeave` events of `myDataLabel`. Yes, a label control can have events too! Virtually all controls have events.

There are too many events associated with each control type to list in this book. To know which events are available, check the help documentation.

For the event to work, you must actually create the methods you associated to them. In lines 57 to 82, you see a number of very simple methods. These are the same methods that were associated in lines 49 to 54.

Creating an OK Button

A common button that can be found on many forms is an OK button. This button is clicked when users complete what they are doing. The result of this button is that the form is usually closed.

If you created the form and are using the `Application` class's `Run` method, you can create an event handler for a button click that ends the `Run` method. This method can be as simple as

```
protected void btnOK_Click( object sender, System.EventArgs e)
{
    // Final code logic before closing form
    Application.Exit();   // Ends the Application.Run message loop.
}
```

If you don't' want to exit the entire application or application loop, then you can use the `Close` method on the form instead. The `Close` method will close the form.

There is an alternative method for implementing the logic of OK. This involves taking a slightly different approach. First, instead of using the `Application` class's `Run` method, you can use a `Form` object's `ShowDialog` method. The `ShowDialog` method displays a dialog and waits for the dialog to complete. A dialog is simply a form. All other logic for creating the form is the same.

In general, if a user presses the Enter key on a form, the form will activate the OK button. You can associate the Enter key with a button using the `AcceptButton` property of the form. You set this property equal to the button that will be activated when the Enter key is pressed.

Working with Text Boxes

Another popular control is the text box. The text box control is used to obtain text input from the users. Using a text box control and events, you can obtain information from your users that you can then use. Listing 16.9 illustrates the use of text box controls; Figure 16.14 shows the output.

LISTING 16.9 text1.csUsing Textbox Controls

```
 1:  // text1.cs - Working with text controls
 2:  //-------------------------------------------------------------
 3:
 4:  using System;
 5:  using System.Windows.Forms;
 6:  using System.Drawing;
 7:
 8:  public class frmGetName : Form
 9:  {
10:     private Button btnOK;
11:
12:     private Label  lblFirst;
13:     private Label  lblMiddle;
14:     private Label  lblLast;
15:     private Label  lblFullName;
16:     private Label  lblInstructions;
17:
18:     private TextBox txtFirst;
19:     private TextBox txtMiddle;
```

LISTING 16.9 continued

```
20:     private TextBox txtLast;
21:
22:     public frmGetName()
23:     {
24:        InitializeComponent();
25:     }
26:
27:     private void InitializeComponent()
28:     {
29:         this.FormBorderStyle = FormBorderStyle.Fixed3D;
30:         this.Text = "Get User Name";
31:         this.StartPosition = FormStartPosition.CenterScreen;
32:
33:         // Instantiate the controls...
34:         lblInstructions = new Label();
35:         lblFirst    = new Label();
36:         lblMiddle   = new Label();
37:         lblLast     = new Label();
38:         lblFullName = new Label();
39:
40:         txtFirst    = new TextBox();
41:         txtMiddle   = new TextBox();
42:         txtLast     = new TextBox();
43:
44:         btnOK = new Button();
45:
46:         // Set properties
47:
48:         lblFirst.AutoSize = true;
49:         lblFirst.Text     = "First Name:";
50:         lblFirst.Location = new Point( 20, 20);
51:
52:         lblMiddle.AutoSize = true;
53:         lblMiddle.Text     = "Middle Name:";
54:         lblMiddle.Location = new Point( 20, 50);
55:
56:         lblLast.AutoSize = true;
57:         lblLast.Text     = "Last Name:";
58:         lblLast.Location = new Point( 20, 80);
59:
60:         lblFullName.AutoSize = true;
61:         lblFullName.Location = new Point( 20, 110 );
62:
63:         txtFirst.Width = 100;
64:         txtFirst.Location = new Point(140, 20);
65:
66:         txtMiddle.Width = 100;
67:         txtMiddle.Location = new Point(140, 50);
```

LISTING 16.9 continued

```
68:
69:         txtLast.Width = 100;
70:         txtLast.Location = new Point(140, 80);
71:
72:         lblInstructions.Width = 250;
73:         lblInstructions.Height = 60;
74:         lblInstructions.Text = "Enter your first, middle, and last name." +
75:                                "\nYou will see your name appear as you
➥type." +
76:                                "\nFor fun, edit your name after entering
➥it.";
77:         lblInstructions.TextAlign = ContentAlignment.MiddleCenter;
78:         lblInstructions.Location =
79:             new Point(((this.Width/2) - (lblInstructions.Width / 2 )), 140);
80:
81:         this.Controls.Add(lblFirst);    // Add label to form
82:         this.Controls.Add(lblMiddle);
83:         this.Controls.Add(lblLast);
84:         this.Controls.Add(lblFullName);
85:         this.Controls.Add(txtFirst);
86:         this.Controls.Add(txtMiddle);
87:         this.Controls.Add(txtLast);
88:         this.Controls.Add(lblInstructions);
89:
90:         btnOK.Text = "Done";
91:         btnOK.BackColor = Color.LightGray;
92:         btnOK.Location = new Point(((this.Width/2) - (btnOK.Width / 2)),
93:                                    (this.Height - 75));
94:
95:         this.Controls.Add(btnOK);  // Add button to form
96:
97:         // Event handlers
98:         btnOK.Click += new System.EventHandler(this.btnOK_Click);
99:         txtFirst.TextChanged += new
➥System.EventHandler(this.txtChanged_Event);
100:         txtMiddle.TextChanged += new
➥System.EventHandler(this.txtChanged_Event);
101:         txtLast.TextChanged += new
➥System.EventHandler(this.txtChanged_Event);
102:     }
103:
104:     protected void btnOK_Click( object sender, System.EventArgs e)
105:     {
106:        Application.Exit();
107:     }
108:
109:     protected void txtChanged_Event( object sender, System.EventArgs e)
110:     {
```

LISTING 16.9 continued

```
111:        lblFullName.Text = txtFirst.Text + " " + txtMiddle.Text + " " +
➥txtLast.Text;
112:     }
113:
114:     public static void Main( string[] args )
115:     {
116:        Application.Run( new frmGetName() );
117:     }
118:  }
```


OUTPUT

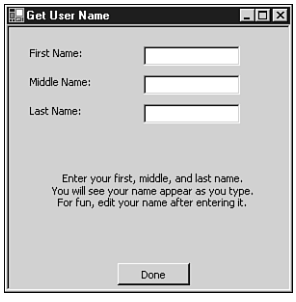


FIGURE 16.14

Using the text box control.

ANALYSIS As you can see by looking at the output of this listing, the applications you are creating are starting to look useful. The text box controls in this listing enable your users to enter their name. This name is concatenated and displayed to the screen.

Although Listing 16.9 is long, much of the code is repetitive because of the three similar controls for first, middle, and last names. In lines 10 to 20, a number of controls are declared within the `frmGetName` class. These controls are instantiated (lines 34 to 44) and assigned values within the `InitializeComponent` method. In lines 48 to 58, the three labels for first, middle, and last names are assigned values. They first have their `AutoSize` property set to `true` so the control will be large enough to hold the information. The text value is then assigned. Finally, each are positioned on the form. As you can see, they are each placed 20 pixels from the edge. They also are spaced vertically at different positions.

In lines 60 to 61, the full name label is declared. Its `Text` property is not assigned a value at this point. It will obtain its `Text` assignment when an event is called.

Lines 63 to 70 assign locations and widths to the text box controls that are being used in this program. As you can see, these assignments are done in the same manner as for the controls you've already learned about.

In lines 72 to 79, instructions are added via another label control. Don't be confused by all the code being used here. In line 74, three lines of text are being added to the control;

however, this is really just one very long string of text that has been broken to make it easier to read. The plus sign concatenates the three pieces and assigns them all as a single string to the `lblInstructions.Text` property. Line 77 uses another property you have not seen before. This is the `TextAlign` property that aligns the text within the label control. This property is assigned a value from the `ContentAlignment` enumeration. In this listing, `MiddleCenter` was used. Other valid values from the `ContentAlignment` enumerator include `BottomCenter`, `BottomLeft`, `BottomRight`, `MiddleLeft`, `MiddleRight`, `TopCenter`, `TopLeft`, and `TopRight`.

Although different controls have properties with the same name, such properties might not accept the same values. For example, the label control's `TextAlign` property is assigned a value from the `ContentAlignment` enumeration. The text box control's `TextAlign` is assigned a `HorizontalAlignment` enumeration value.

Lines 98 to 101 add exception handlers. As you can see, line 98 adds a handler for the `Click` event of the `btnOK` button. The method called is in lines 104 to 107. This method exits the application loop, thus helping end the program.

Lines 99 to 101 add event handlers for the `TextChanged` event of the text box buttons. Whenever the text within one of the three text boxes is changed, the `txtChanged_Event` will be called. As you can see, the same method can be used with multiple handlers. This method concatenates the three name fields and assigns the result to the `lblFullNameText` control.

Working with Other Controls

Listing 16.9 provides the basis of what you need to build basic applications. There are a number of other controls that you can use. For the most part, basic use of the controls is similar to the use you've seen in the listings in today's lessons. You create the control, you modify the properties to be what you need, you create event handlers to handle any actions you want to react to, and finally you place the control on the form. Some controls, such as list boxes, are a little more complex for assigning initial data, but overall the process of using such controls is the same.

As mentioned earlier, covering all the controls and their functionality would be a very, very thick book on its own. The online documentation is a great starting point for working the details of these. Although it is beyond the scope of this book to go into too much depth, the popularity of windows-based programming warrants covering a few additional windows topics in tomorrow's lesson before moving on to Web forms and services.

Summary

Today's lesson was a lot of fun. As you have learned, using the classes, methods, properties, and events defined in the `System.Windows.Forms` namespace can help you create windows-based applications with very little code. Today, you learned how to create and customize a form. You also learned how to add basic controls to the form and how to work with events to give your forms functionality. Although only a few of the controls were introduced, you will find that using the other controls is similar in a lot of ways to working with the ones presented today.

Tomorrow, you continue to expand on what you learned today. On Day 18, "Web Development," you'll learn how windows forms differ from Web forms.

Q&A

Q Where can I learn more about Windows forms?

A You can learn more about Windows forms from the documentation that comes with the .NET SDK. This includes a Windows Forms Quick Start.

Q I noticed that `Form` is listed in the table of controls. Why?

A A form is a control. Most of the functionality of a control is also available to a form.

Q Why didn't you cover all the properties, events, and methods for the controls presented today?

A There are over 40 controls within the framework classes. Additionally, many of these controls have well over a hundred methods, events, and properties. To cover over 4,000 items with just a line each would take roughly 80 pages.

Workshop

The Workshop provides quiz questions to help you solidify your understanding of the material covered and exercises to provide you with experience in using what you've learned. Try to understand the quiz and exercise answers before continuing to the next day's lesson. Answers are provided in Appendix A, "Answers."

Quiz

1. What is the name of the namespace where most of the windows controls are located?
2. What method can be used to display a form?

3. What are the three steps involved in getting a control on a form?
4. What do you enter on the command line to compile the program xyz.cs as a windows program?
5. If you want to include the assembly myAssmb.dll when you compile the program xyz.cs, what do you enter on the command line?
6. What does the `Show()` method of the `Form` class do? What is the problem with using this method?
7. Which of the following causes the `Application.Run` method to end?
    a. A method
    b. An event
    c. The last line of code in the program is reached
    d. It never ends
    e. None of the above
8. What are the possible colors you can use for a form? What namespace needs to be included to use such colors?
9. What property can be used to assign a text value to a label?
10. What is the difference between a text box and a label?

Exercises

1. Write the shortest Windows application you can.
2. Create a program that centers a 200-x-200–pixel form on the screen.
3. Create a form that contains a text field that can be used to enter a number. When the user presses a button, display a message in a label that states whether the number is from 0 to 1000.
4. **BUG BUSTER**: The following program has a problem. Enter it in your editor and compile it. Which lines generate error messages?

```
 1:  using System.Windows.Forms;
 2:
 3:  public class frmHello : Form
 4:  {
 6:      public static void Main( string[] args )
 7:      {
 8:         frmHello frmHelloApp = new frmHello();
 9:         frmHelloApp.Show();
10:      }
11:  }
```