Mastering™ Visual Basic® .NET

Evangelos Petroutsos

Chapter 5: Working with Forms

ISBN: 0-7821-2877-7

Sybex Inc.
1151 Marina Village Parkway
Alameda, CA 94501
U.S.A.
Phone: 510-523-8233
www.sybex.com

Chapter 5

Working with Forms

In Visual Basic, the *form* is the container for all the controls that make up the user interface. When a Visual Basic application is executing, each window it displays on the desktop is a form. In previous chapters, we concentrated on placing the elements of the user interface on forms, setting their properties, and adding code behind selected events. Now, we'll look at forms themselves and at a few related topics, such as menus (forms are the only objects that can have menus attached), how to design forms that can be automatically resized, and how to access the controls of one form from within another form's code. The form is the top-level object in a Visual Basic application, and every application starts with the form.

NOTE *The terms* form *and* window *describe the same entity. A window is what the user sees on the desktop when the application is running. A form is the same entity at design time. The proper term is a Windows form, as opposed to a Web form, but I will refer to them as forms.*

Forms have built-in functionality that is always available without any programming effort on your part. You can move a form around, resize it, and even cover it with other forms. You do so with the mouse, or with the keyboard through the Control menu. As you will see, forms are not passive containers; they're "intelligent" objects that are aware of the controls placed on them and can actually manipulate the controls at runtime. For example, you can instruct the form to resize certain controls when the form itself is resized. Forms have many trivial properties that won't be discussed here. Instead, let's jump directly to the properties that are unique to forms and then look at how to manipulate forms from within an application's code.

The forms that constitute the visible interface of your application are called *Windows forms;* this term includes both the regular forms and dialog boxes, which are simple forms you use for very specific actions, such as to prompt the user for a specific piece of data or to display critical information. A *dialog box* is a form with a small number of controls, no menus, and usually an OK and a Cancel button to close it. For more information on dialog boxes, see the section "Forms vs. Dialog Boxes" later in this chapter. Everything you'll read about forms in the following sections applies to dialog boxes as well, even if some form features (such as menus) are never used with dialog boxes.

VB6 ➠ VB.NET

The Form Designer is one of the most improved areas of VB.NET. For the first time, you can design forms that can be easily resized—anyone who has programmed in earlier versions of VB knows what a hassle the resizing of forms could be. The Anchor and Dock properties allow you to anchor controls on the edges of the form and dock them on the form. When the form is resized, the controls on it can be either resized or moved to new locations, so that they remain visible.

If the controls can't fit the form, scroll bars can appear automatically, so that users can scroll the form in its window and bring another section into view, if the form's AutoScroll property is True. Scrolling forms are also new to VB.NET.

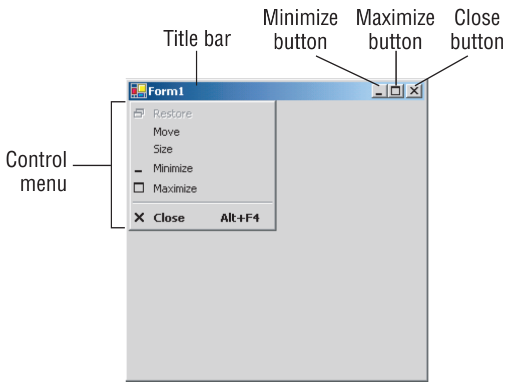
A new special control was added, whose sole purpose is to act as a pane separator on forms: the Splitter control. This control is a thin horizontal or vertical stripe that allows you to resize two adjacent controls. If two TextBox controls on the same form are separated by a Splitter control, users can shrink one TextBox to make more room for the other. Again, no code required.

Of course, many things have changed too. You can no longer show a form by calling its Show method. You must first create an instance of the form (a variable of the Form type) that you want to show and then call the Show method of this variable. You no longer have arrays of controls. This isn't much of a problem, though, because with VB.NET you can create instances of new controls from within your code and position them on the form.

The Appearance of Forms

Applications are made up of one or more forms (usually more than one), and the forms are what users see. You should craft your forms carefully, make them functional, and keep them simple and intuitive. You already know how to place controls on the form, but there's more to designing forms than populating them with controls. The main characteristic of a form is the title bar on which the form's caption is displayed (see Figure 5.1).

FIGURE 5.1
The elements of the form



Clicking the icon on the left end of the title bar opens the Control menu, which contains the commands shown in Table 5.1. On the right end of the title bar are three buttons: Minimize, Maximize, and Close. Clicking these buttons performs the associated function. When a form is maximized, the Maximize button is replaced by the Restore button. When clicked, this button resets the form to the size and position before it was maximized. The Restore button is then replaced by the Maximize button.

Table 5.1: Commands of the Control Menu

Command	Effect
Restore	Restores a maximized form to the size it was before it was maximized; available only if the form has been maximized
Move	Lets the user move the form around with the mouse
Size	Lets the user resize the form with the mouse
Minimize	Minimizes the form
Maximize	Maximizes the form
Close	Closes the current form

Properties of the Form Control

You're familiar with the appearance of the forms, even if you haven't programmed in the Windows environment in the past; you have seen nearly all types of windows in the applications you're using every day. The floating toolbars used by many graphics applications, for example, are actually forms with a narrow title bar. The dialog boxes that display critical information or prompt you to select the file to be opened are also forms. You can duplicate the look of any window or dialog box through the following properties of the Form object.

AcceptButton, CancelButton

These two properties let you specify the default Accept and Cancel buttons. The Accept button is the one that's automatically activated when you press Enter, no matter which control has the focus at the time, and is usually the button with the OK caption. Likewise, the Cancel button is the one that's automatically activated when you hit the Esc key and is usually the button with the Cancel caption. To specify the Accept and Cancel buttons on a form, locate the AcceptButton and CancelButton properties of the form and select the corresponding controls from a drop-down list, which contains the names of all the buttons on the form. You can also set them to the name of the corresponding button from within your code.

AutoScale

This property is a True/False value that determines whether the controls you place on the form are automatically scaled to the height of the current font. When you place a TextBox control on the form, for example, and the AutoScale property is True, the control will be tall enough to display a single line of text in the current font. The default value is True, which is why you can't make the controls smaller than a given size. This is a property of the form, but it affects the controls on the form. If you change the Font property of the form after you have placed a few controls on it, the existing controls won't be affected. The controls are adjusted to the current font of the form the moment they're placed on it.

AutoScroll

This is one of the most needed of the Form object's new properties. The AutoScroll property is a True/False value that indicates whether scroll bars will be automatically attached to the form (as seen in Figure 5.2) if it's resized to a point that not all its controls are visible. This property is new to VB.NET and will help you design large forms without having to worry about the resolution of the monitor on which they'll be displayed.

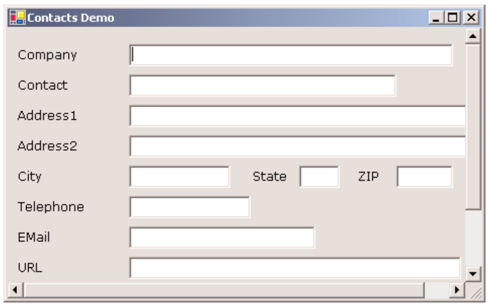


FIGURE 5.2
If the controls don't fit in the form's visible area, scroll bars can be attached automatically.

The AutoScroll property is used in conjunction with three other properties, described next: AutoScrollMargin, AutoScrollMinSize, and AutoScrollPosition.

AutoScrollMargin

This is a margin, expressed in pixels, that's added around all the controls on the form. If the form is smaller than the rectangle that encloses all the controls adjusted by the margin, the appropriate scroll bar(s) will be displayed automatically.

If you expand the AutoScrollMargin property in the Properties window, you will see that it's an object (a Size object, to be specific). It exposes two members, the Width and Height properties, and you must set both values. The default value is (0,0). To set this property from within your code, use statements like these:

```
Me.AutoScrollMargin.Width = 40
Me.AutoScrollMargin.Height = 40
```

AutoScrollMinSize

This property lets you specify the minimum size of the form, before the scroll bars are attached. If your form contains graphics you want to be visible at all times, set the Width and Height members of the AutoScrollMinSize property accordingly. Notice that this isn't the form's minimum size; users can make the form even smaller. To specify a minimum size for the form, use the Minimum-Size property, described later in this section.

Let's say the AutoScrollMargin properties of the form are 180 by 150. If the form is resized to less than 180 pixels horizontally or 150 pixels vertically, the appropriate scroll bars will appear automatically, as long as the AutoScroll property is True. If you want to enable the AutoScroll feature when the form's width is reduced to anything less than 250 pixels, set the AutoScrollMinSize property to (250, 0). Obviously, if the AutoScrollMinSize value is smaller than the dimensions of the form that will automatically invoke the AutoScroll feature, AutoScrollMinSize has no effect. In this example, setting `AutoScrollMinSize.Width` to anything less than 180 or `AutoScrollMinSize.Height` to anything less than 150 will have no effect on the appearance of the form and its scroll bars.

AutoScrollPosition

This property lets you read (or set) the location of the auto-scroll position. The AutoScrollPosition is the number of pixels by which the two scroll bars were displaced from their initial locations. You can read this property to find out by how much the scroll bars were moved, or to move the scroll bars from within your code.

Use this property in very specialized applications, because the form's scroll bars are adjusted automatically to bring the control that has the focus into view. As long as the users of the application press the Tab key to move the focus to the next control, the focused control will be visible.

BorderStyle

The BorderStyle property determines the style of the form's border and the appearance of the form; it takes one of the values shown in Table 5.2. You can make the form's title bar disappear altogether by setting the form's BorderStyle property to FixedToolWindow, the ControlBox property to False, and the Text property to an empty string. However, a form like this can't be moved around with the mouse and will probably frustrate users.

Table 5.2: The FormBorderStyle Enumeration

Value	Effect
`None`	Borderless window that can't be resized; this setting should be avoided.
`Sizable`	(default) Resizable window that's used for displaying regular forms.
`Fixed3D`	Window with a visible border, "raised" relative to the main area. Can't be resized.
`FixedDialog`	A fixed window, used to create dialog boxes.

Continued on next page

TABLE 5.2: THE FORMBORDERSTYLE ENUMERATION *(continued)*

VALUE	EFFECT
`FixedSingle`	A fixed window with a single line border.
`FixedToolWindow`	A fixed window with a Close button only. It looks like the toolbar displayed by the drawing and imaging applications.
`SizableToolWindow`	Same as the FixedToolWindow but resizable. In addition, its caption font is smaller than the usual.

ControlBox

This property is also True by default. Set it to False to hide the icon and disable the Control menu. Although the Control menu is rarely used, Windows applications don't disable it. When the ControlBox property is False, the three buttons on the title bar are also disabled. If you set the Text property to an empty string, the title bar disappears altogether.

KeyPreview

This property enables the form to capture all keystrokes before they're passed to the control that has the focus. Normally, when you press a key, the KeyPress event of the control with the focus is triggered (as well as the other keystroke-related events), and you can handle the keystroke from within the control's appropriate handler. In most cases, we let the control handle the keystroke and don't write any form code for that.

If you want to use "universal" keystrokes in your application, you must set the KeyPreview property to True. Doing so enables the form to intercept all keystrokes, so that you can process them from within the form's keystroke events. The same keystrokes are then passed to the control with the focus, unless you "kill" the keystroke by setting its Handled property to True when you process it on the form's level. For more information on processing keystrokes at the Form level and using special keystrokes throughout your application, see the Contacts project later in this chapter.

MinimizeBox, MaximizeBox

These two properties are True by default. Set them to False to hide the corresponding buttons on the title bar.

MinimumSize, MaximumSize

These two properties read or set the minimum and maximum size of a form. When users resize the form at runtime, the form won't become any smaller than the dimensions specified with the MinimumSize property and no larger than the dimensions specified by MaximumSize. The MinimumSize property is a Size object, and you can set it with a statement like the following:

```
Me.MinimumSize = New Size(400, 300)
```

Or, you can set the width and height separately:

```
Me.MinimumSize.Width = 400
Me.MinimumSize.Height = 300
```

The `MinimumSize.Height` property includes the height of the Form's title bar; you should take that into consideration. If the minimum usable size of the Form is 400 by 300, use the following statement to set the MinimumSize property:

```
me.MinimumSize = New Size(400, 300 + SystemInformation.CaptionHeight)
```

Tip *The height of the caption is not a property of the Form object, even though you will find it useful in determining the useful area of the form (the total height minus the caption bar). Keep in mind that the height of the caption bar is given by the CaptionHeight property of the SystemInformation object.*

SizeGripStyle

This property gets or sets the style of sizing handle to display in the bottom-right corner of the form; it can have one of the values shown in Table 5.3. By default, forms are resizable, even if no special mark appears at the bottom-right corner of the form. This little mark indicating that a form can be resized is new to VB.NET and adds a nice touch to the look of the form.

Table 5.3: The SizeGripStyle Enumeration

Value	Effect
`Auto`	(default) The SizeGrip is displayed as needed.
`Show`	The SizeGrip is displayed at all times.
`Hide`	The SizeGrip is not displayed, but the form can still be resized with the mouse (Windows 95/98 style).

StartPosition

This property determines the initial position of the form when it's first displayed; it can have one of the values shown in Table 5.4.

Table 5.4: The FormStartPosition Enumeration

Value	Effect
`CenterParent`	The form is centered in the area of its parent form.
`CenterScreen`	The form is centered on the monitor.
`Manual`	The location and size of the form will determine its starting position. See the discussion of the Top, Left, Width, and Height properties of the form, later in this section.

Continued on next page

TABLE 5.4: THE FORMSTARTPOSITION ENUMERATION *(continued)*

VALUE	EFFECT
`WindowsDefaultBounds`	The form is positioned at the default location and size determined by Windows.
`WindowsDefaultLocation`	The form is positioned at the Windows default location and has the dimensions you've set at design time.

TOP, LEFT

These two properties set or return the coordinates of the form's top-left corner in pixels. You'll rarely use these properties in your code, since the location of the window on the desktop is determined by the user at runtime.

TOPMOST

This property is a True/False value that lets you specify whether the form will remain on top of all other forms in your application. Its default property is False, and you should change it only in rare occasions. Some dialog boxes, like the Find and Replace dialog box of any text processing application, are always visible, even when they don't have the focus. To make a form remain visible while it's open, set its TopMost property to True.

WIDTH, HEIGHT

These two properties set or return the form's width and height in pixels. They are usually set from within the form's Resize event handler, to keep the size of the form at a minimum size. The form's width and height are usually controlled by the user at runtime.

Placing Controls on Forms

As you already know, the second step in designing your application's interface is the design of the forms (the first step being the analysis and careful planning of the basic operations you want to provide through your interface). Designing a form means placing Windows controls on it, setting their properties, and then writing code to handle the events of interest. Visual Studio.NET is a rapid application development (RAD) environment. This doesn't mean that you're expected to develop applications rapidly. It has come to mean that you can rapidly prototype an application and show something to the customer. And this is made possible through the visual tools that come with VS.NET, especially the new Form Designer.

To place controls on your form, you select them in the Toolbox and then draw, on the form, the rectangle in which the control will be enclosed. Or, you can double-click the control's icon to place an instance of the control on the form. All controls have a default size, and you can resize the control on the form with the mouse. Next, you set the control's properties in the Properties window.

Each control's dimensions can also be set in the Properties window, through the Width and Height properties. These two properties are expressed in pixels. You can also call the Width and Height properties from within your code to read the dimensions of a control. Likewise, the Top and Left properties return (or set) the coordinates of the top-left corner of the control. In the section

"Building Dynamic Forms at Runtime," later in this chapter, you'll see how to create new controls at runtime and place them on a form from within your code. You'll use these properties to specify the location of the new controls on the form in your code.

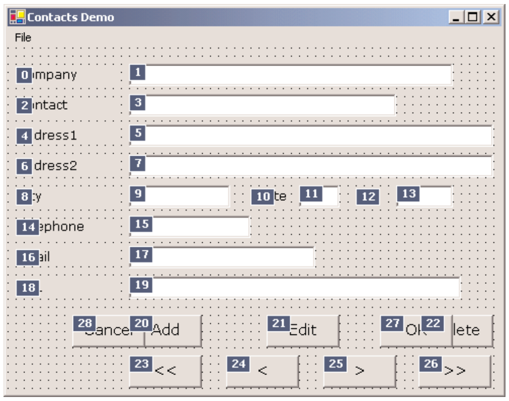
Setting the TabOrder

Another important issue in form design is the tab order of the controls on the form. As you know, pressing the Tab key at runtime takes you to the next control on the form. The order of the controls isn't determined by the form; you specify the order when you design the application, with the help of the TabOrder property. Each control has its own TabOrder setting, which is an integer value. When the Tab key is pressed, the focus is moved to the control whose tab order immediately follows the tab order of the current control. The TabOrder of the various controls on the form need not be consecutive.

To specify the tab order of the various controls, you can set their TabOrder property in the Properties window, or you can select the Tab Order command from the View menu. The tab order of each control will be displayed on the corresponding control, as shown in Figure 5.3 (the form shown in the figure is the Contacts application, which is discussed shortly). Notice that some of the buttons at the bottom of the form are not aligned as they should be. The OK and Cancel buttons should be on top of the Add and Delete buttons, hiding them. I had to displace them to set the tab order of all controls on the form and then align some of the buttons again.

FIGURE 5.3

Setting the TabOrder of the controls on the main form of the Contacts project



To set the tab order of the controls, click each control in the order in which you want them to receive the focus. Notice that you can't change the tab order of a few controls only. You must click all of them in the desired order, starting with the first control in the tab order. The tab order need not be the same as the physical order of the controls on the form, but controls that are next to each other in the tab order should be placed next to each other on the form as well.

NOTE *The default tab order is the same as the order in which you place the controls on the form. Unless you keep the tab order in mind while you design the form, you'll end up with a form that moves the focus from one control to the next in a totally unpredictable manner. Once all the controls are on the form, you should always check their tab order to make sure it won't confuse users.*

As you place controls on the form, don't forget to lock them, so that you won't move them around by mistake as you work with other controls. You can lock the controls in their places either by setting their Locked property to True, or by locking all the controls on the form with the Format ➢ Lock Controls command.

VB6 ➠ VB.NET

Many of the controls in earlier versions of Visual Basic exposed a Locked property too, but this property had a totally different function. The old Locked property prevented users from editing the controls at runtime (entering text on a TextBox control, for example). The new Locked property is effective at design time only; it simply locks the control on the form, so that it can't be moved by mistake.

Designing functional forms is a crucial step in the process of developing Windows applications. Most data-entry operators don't work with the mouse, and you must make sure all the actions can be performed with the keyboard. This doesn't apply to graphics applications, of course, but most applications developed with VB are business applications. If you're developing a data-entry form, for example, you must take into consideration the needs of the users in designing these forms. Make a prototype and ask the people who will use the application to test-drive it. Listen to their objections carefully, collect all the information, and then use it to refine your application's user interface. Don't defend your design—just learn from the users. They will uncover all the flaws of the application, and they'll help you design the most functional interface.

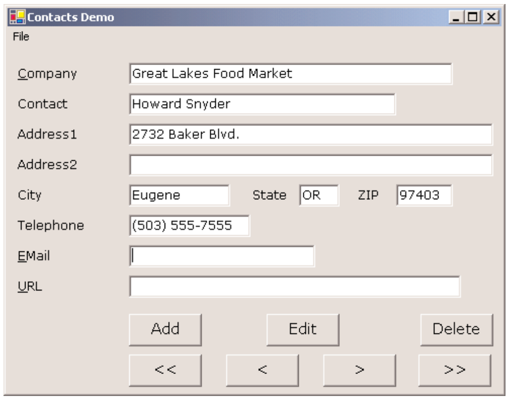
The process of designing forms is considered to be the simplest step by most beginners, but a bad user interface might force you to redesign the entire application later on—not to mention that an inefficient interface will discourage people from using your application. Take your time to think about the interface, the controls on your forms, and how users will navigate. I'm not going to discuss the topic of designing user interfaces in this book. Besides, this is one of the skills you'll acquire with time.

VB.NET at Work: The Contacts Project

I would like to conclude this section with an example of a simple data-entry form that demonstrates many of the topics discussed here, as well as a few techniques for designing easy-to-use forms. Figure 5.4 shows a data-entry form for contact information, and I'm sure you will add your own fields to make this application more useful. You can navigate through the contacts using the buttons with the arrows, as well as add new contacts or delete existing ones by clicking the appropriate buttons. When you're entering a new contact, the buttons shown in Figure 5.4 are replaced by the usual OK and Cancel buttons. The action of adding a new contact must end by clicking one of these two buttons. After committing a new contact, or canceling the action, the usual navigation buttons will appear again.

Once the controls are on the form, the first step is to set their tab order. You must specify a TabOrder even for controls that never receive focus, such as the labels. In addition to the tab order of the controls, we'll also use shortcut keys to give the user quick access to the most common fields. The shortcut keys are displayed as underlined characters on the corresponding labels, as you can see in Figure 5.4.

FIGURE 5.4
A simple data-entry screen



Company	Great Lakes Food Market
Contact	Howard Snyder
Address1	2732 Baker Blvd.
Address2	
City	Eugene
State	OR
ZIP	97403
Telephone	(503) 555-7555
EMail	
URL	

To set the TabOrder of the controls, use the View ➢ Tab Order command. Click all the controls in the order you want them to receive the focus, starting with the first label. The proper order of the controls is shown back in Figure 5.3. You can change the order of the buttons, if you want, but the labels and text boxes must have consecutive settings. Don't forget to include the buttons in the tab order. Then open the View menu again and select the Tab Order command to return to the regular view of the Form Designer.

If you run the application now, you'll see that the focus moves from one TextBox to the next and the labels are skipped. Since the labels don't accept any data, they receive the focus momentarily and then the focus is passed to the next control in the tab order. After the last TextBox control, the focus is moved to the buttons, and then back to the first TextBox control. To add a shortcut key for the most common fields, determine which of the fields will have their own shortcut key and which keys will be used for that purpose. Being the Internet buffs that we all are, let's assign shortcut keys to the Company, EMail, and URL fields. Locate each label's Text property in the Properties window and insert the & symbol in front of the character you want to act as a shortcut for each Label. The Text properties of the three controls should be &Company, &EMail, and &URL.

Shortcut keys are activated at runtime by pressing the shortcut character while holding down the Alt key. The shortcut key will move the focus to the corresponding Label control, but because labels can't receive the focus, it's passed immediately to the next control in the tab order, which is the adjacent TextBox control. For this technique to work, you must make sure that all controls are properly arranged in the tab order.

TIP *By the way, if you want to display the & symbol on a Label control, prefix it with another & symbol. To display the string "Tom & Jerry" on a Label control, assign the string "Tom && Jerry" to its Text property.*

If you run the application now, you'll be able to quickly move the focus to the Company, EMail, and URL boxes by pressing the shortcut key while holding down the Alt key. To access the other fields (the TextBoxes without shortcuts), the user can press Tab to move forward in the tab order or Shift+Tab to move backward. Try to move the focus with the mouse and enter data with the keyboard, and you'll soon understand what kind of interface a data-entry operator would rather work with.

The contacts are stored in an ArrayList object, which is similar to an array but a little more convenient. We'll discuss ArrayList in Chapter 11; for now, you can ignore the parts of the application that manipulate the contacts and focus on the design issues.

Now enter a new contact by clicking the Add button, or edit an existing contact by clicking the Edit button. Both actions must end with the OK or Cancel button. In other words, we won't allow users to switch to another contact while adding or editing a contact. The code behind the various buttons is straightforward. The Add button hides all the navigational buttons at the bottom of the form and clears the TextBoxes. The OK button saves the new contact to an ArrayList structure and redisplays the navigational buttons. The Cancel button ignores the data entered by the user and likewise displays the navigational buttons. In either case, when the user switches back to the view mode, the TextBoxes are also locked, by setting their ReadOnly properties to True.

Don't worry about the statements that manipulate the ArrayList with the contacts or the statements that save the contacts to a disk file and load them back to the application from a disk file. We'll come back to this project in Chapter 11, where we'll discuss ArrayLists. Just focus on the statements that control the appearance of the form.

For now, you can use the commands of the File menu to load or save a set of contacts. These commands are quite simple: they load the same file, `CONTACTS.BIN` in the application's folder. After reading about the Open File and Save File dialog controls, you can modify the code so that it prompts the user about the file to read from or write to. The `CONTACTS.BIN` file you will find on the CD contains a few contacts I created from the Northwind sample database.

The application keeps track of the current contact through the *currentContact* variable. As you move with the navigation keys, the value of this variable is increased or decreased accordingly. When you edit a contact, the new values are stored in the Contact object that corresponds to the location indicated by the *currentContact* variable. When you add a new contact, a new Contact object is added to the current collection, and its order becomes the new value of the *currentContact* variable. Most of the project's code performs trivial tasks—hiding and showing the buttons at the bottom of the form, displaying the fields of the current contact on the TextBox control, clearing the same controls to prepare them to accept a new contact, and so on. We'll come back to this project in Chapter 11, where I'll show you how to manipulate ArrayLists. There you'll find more information about storing data in an ArrayList, as well as how to save an ArrayList to a disk file and how to load the data from the file back to the ArrayList.

HANDLING KEYSTROKES

The last topic demonstrated in this example is how to capture certain keystrokes, regardless of the control that has the focus. We'll use the F10 keystroke to display the total number of contacts entered so far. Set the form's KeyPreview property to True and then enter the following code in the form's KeyDown event:

```
If e.Keycode = keys.F10 Then
   MsgBox("There are " & Contacts.Count.ToString & " contacts in the database")
   e.Handled = True
End If
```

The form captures all the keystrokes and processes them. After it's done with them, it may allow the keystrokes to be passed to the control that has the focus. The processing is quite trivial. It compares the key pressed to the F10 key and, if F10 was pressed, it displays the number of contacts entered so far in a message box. Then, it stops the keystroke from propagating back to control with the focus by setting its Handled property to True. Listing 5.1 is the complete event handler; if you omit that statement in the listing, the F10 keystroke will be passed to the control with the focus—the control that would receive the notification about the keystroke by default, if the form's Key-Preview property was left to its default value. Of course, the key F10 isn't processed by the TextBox control, so it's not necessary to "kill" it before it reaches the control.

Listing 5.1: Handling Keystrokes in the Form's KeyDown Event Handler

```
Public Sub Form1_KeyDown(ByVal sender As Object, _
              ByVal e As System.WinForms.KeyEventArgs) Handles Form1.KeyUp
   If e.Keycode = Keys.F10 Then
      MsgBox("There are " & Contacts.Count.ToString & _
             " contacts in the database")
      e.Handled = True
   End If
   If e.KeyCode = Keys.Subtract And e.Modifiers = Keys.Alt Then
      bttnPrevious_Click(sender, e)
   End If
   If e.KeyCode = Keys.Add And e.Modifiers = Keys.Alt Then
      bttnNext_Click(sender, e)
   End If
End Sub
```

The KeyDown event handler contains a little more code to capture the Alt+Plus and Alt+Minus key combinations as shortcuts for the buttons that move to the next and previous contact respectively. If the user has clicked the Plus button while holding down the Alt button, the code calls the event handler of the Next button. Likewise, pressing Alt and the Minus key activates the event handler of the Previous button.

The KeyCode property of the *e* argument returns the code of the key that was pressed. All key codes are members of the Keys enumeration, so you need not memorize them. The name of the button with the plus symbol is `Keys.Add`. The Modifiers property of the same argument returns the modifier key(s) that were held down while the key was pressed. Also, all possible values of the Modifiers property are members of the Keys enumeration and will appear in a list as soon as you type the equal sign. The name of the Alt modifier is `Keys.Alt`.

If you run the Contacts application, you'll see that it's not trivial. To add or modify a record, you must click the appropriate button, and while in edit mode, the navigational buttons disappear. The reason is that data-entry operators want to know the state of the application at each moment. With this design, you can't move to another record while editing the current one, as discussed previously.

Another interesting part of the project is the handler of the KeyPress event. This event takes place when a normal key (letter, digit, or punctuation symbol) is pressed. If the OK button is invisible at the time, it means that the user can't edit the current record and the program "chokes" the keystroke, preventing it from reaching the control that has the focus. The form's KeyPress event is handled by the subroutine shown in Listing 5.2.

Listing 5.2: Handling Keystrokes in the Form's KeyPress Event Handler

```
Private Sub Form1_KeyPress(ByVal sender As Object, _
               ByVal e As System.Windows.Forms.KeyPressEventArgs) _
               Handles MyBase.KeyPress
   If bttnOK.Visible = False Then
      e.Handled = True
   End If
End Sub
```

The Contacts project contains quite a bit of code, which will be discussed in more detail in Chapter 11. It's included in this chapter to demonstrate some useful techniques for designing intuitive interfaces, and I've only discussed the sections of the application that relate to the behavior of the form and the controls on it as a group.

Anchoring and Docking

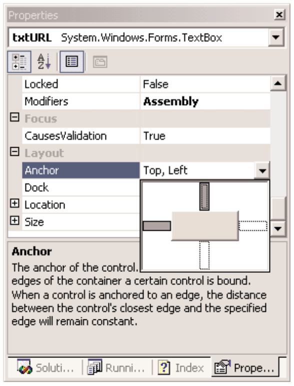
One of the most tedious tasks in designing user interfaces with Visual Basic before VB.NET was the proper arrangement of the controls on the form, especially on forms that users were allowed to resize at runtime. You design a nice form for a given size, and when it's resized at runtime, the controls are all clustered in the top-left corner. A TextBox control that covered the entire width of the form at design time suddenly "cringes" on the left when the user drags out the window. If the user makes the form smaller than the default size, part of the TextBox is invisible, because it's outside the form. You can attach scroll bars to the form, but that doesn't really help—who wants to type text and have to scroll the form horizontally? It makes sense to scroll vertically, because you get to see many lines at once, but if the TextBox control is wider than the form, you can't see an entire line.

Programmers had to be creative and resize and/or rearrange the controls on a form from within the form's Resize event. This event takes place every time the user resizes the form at runtime, and, quite often, we had to insert code in this event to resize controls so that they would continue to take up the entire form's width. You may still have to insert a few lines of code in the Resize event's handler, but a lot of the work of keeping controls aligned is no longer needed. The Anchor and Dock properties of the various controls allow you specify how they will be arranged with respect to the edges of the form when the user resizes it.

The Anchor property lets you attach one or more edges of the control to corresponding edges of the form. The anchored edges of the control maintain the same distance from the corresponding edges of the form. Place a TextBox control on a new form and then open the control's Anchor property in the Properties window. You will see a little square within a larger square, like the one in Figure 5.5, and four pegs that connect the small control to the sides of the larger box. The large box is the form, and the small one is the control. The four pegs are the anchors, which can be either white or gray. The gray anchors denote a fixed distance between the control and the form. By default, the control is placed at a fixed distance from the top-left corner of the form. When the form is resized, the control retains its size and its distance from the top-left corner of the form.

FIGURE 5.5

The settings of the Anchor property



Properties

txtURL System.Windows.Forms.TextBox

Locked	False
Modifiers	**Assembly**
Focus	
CausesValidation	True
Layout	
Anchor	Top, Left
Dock	
Location	
Size	

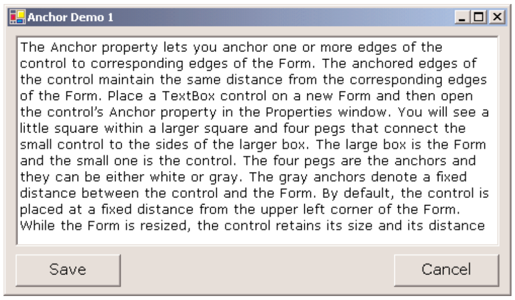
Anchor

The anchor of the control. edges of the container a certain control is bound. When a control is anchored to an edge, the distance between the control's closest edge and the specified edge will remain constant.

Let's say we want our control to fill the width of the form, be aligned to the top of the form, and leave some space for a few buttons at the bottom. Make the TextBox control as wide as the control (allowing, perhaps, a margin of a few pixels on either side). Then place a couple of buttons at the bottom of the form and make the TextBox control tall enough that it stops above the buttons, as shown in Figure 5.6. This is the form of the Anchor project on the CD.

FIGURE 5.6

This form is filled by three controls, regardless of the form's size at runtime.



Anchor Demo 1

The Anchor property lets you anchor one or more edges of the control to corresponding edges of the Form. The anchored edges of the control maintain the same distance from the corresponding edges of the Form. Place a TextBox control on a new Form and then open the control's Anchor property in the Properties window. You will see a little square within a larger square and four pegs that connect the small control to the sides of the larger box. The large box is the Form and the small one is the control. The four pegs are the anchors and they can be either white or gray. The gray anchors denote a fixed distance between the control and the Form. By default, the control is placed at a fixed distance from the upper left corner of the Form. While the Form is resized, the control retains its size and its distance

Now open the TextBox control's Anchor property and make the all four anchors gray by clicking them. This action tells the Form Designer to resize the control accordingly at runtime, so that the distances between the sides of the control and the corresponding sides of the form are the same as you've set at design time.

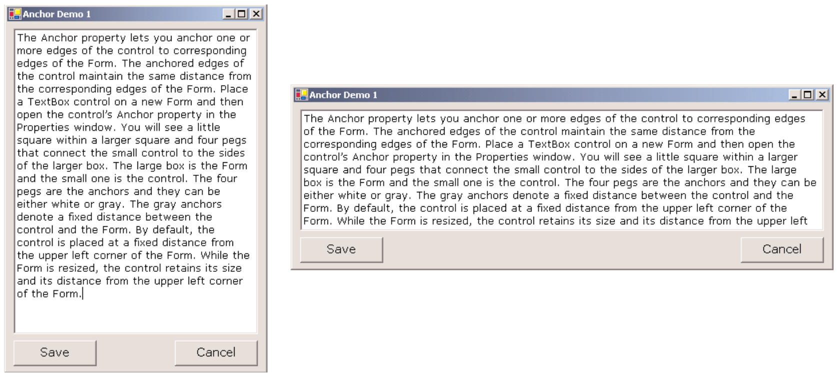
Resize the form at design time, without running the project. The TextBox control is resized according to the form, but the buttons remain fixed. Let's do the same for the two buttons. The two buttons must fit in the area between the TextBox control and the bottom of the form, so we must anchor them to the bottom of the form. Select both controls on the form with the mouse and then open their Anchor property. Make the anchor at the bottom gray and the other three anchors white; this will anchor the two buttons to the bottom of the form. If you resize the form now, the TextBox control will fill it, leaving just enough room for the two buttons at the bottom of the form.

We need to do something more about the buttons. They're aligned vertically, but their horizontal location doesn't change. Select the button to the left, open its Anchor property, and click the left anchor. This will anchor the button to the left side of the form—which is the default behavior anyway.

Now select the button to the right, open its Anchor property, and click the right anchor. This will anchor the second button to the right side of the control. Resize the form again and see how all controls are resized and rearranged on the form at all times. This is much better than the default behavior of the controls on the form. Figure 5.7 shows the same form in two very different sizes, with the TextBox taking up most of the space on the form and leaving room for the buttons, which in turn are repositioned horizontally as the form is resized.

FIGURE 5.7

The form of Figure 5.6 in two different sizes



Anchor Demo 1

The Anchor property lets you anchor one or more edges of the control to corresponding edges of the Form. The anchored edges of the control maintain the same distance from the corresponding edges of the Form. Place a TextBox control on a new Form and then open the control's Anchor property in the Properties window. You will see a little square within a larger square and four pegs that connect the small control to the sides of the larger box. The large box is the Form and the small one is the control. The four pegs are the anchors and they can be either white or gray. The gray anchors denote a fixed distance between the control and the Form. By default, the control is placed at a fixed distance from the upper left corner of the Form. While the Form is resized, the control retains its size and its distance from the upper left corner of the Form.

Anchor Demo 1

The Anchor property lets you anchor one or more edges of the control to corresponding edges of the Form. The anchored edges of the control maintain the same distance from the corresponding edges of the Form. Place a TextBox control on a new Form and then open the control's Anchor property in the Properties window. You will see a little square within a larger square and four pegs that connect the small control to the sides of the larger box. The large box is the Form and the small one is the control. The four pegs are the anchors and they can be either white or gray. The gray anchors denote a fixed distance between the control and the Form. By default, the control is placed at a fixed distance from the upper left corner of the Form. While the Form is resized, the control retains its size and its distance from the upper left

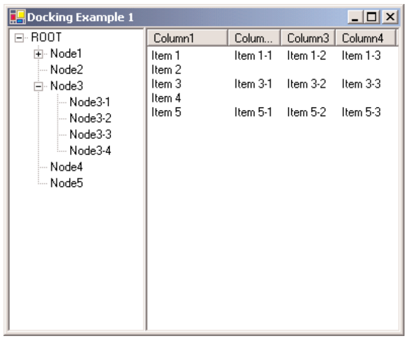
Yet, there's a small problem: if you make the form very narrow, there will be no room for both buttons across the form's width. The simplest way to fix this problem is to impose a minimum size for the form. To do so, you must first decide the form's minimum width and height and then set the MinimumSize property to these values.

In addition to the Anchor property, most controls provide the Dock property, which determines how a control will dock on the form. The default value of this property is None. Create a new form, place a TextBox control on it, and then open the control's Dock property. The various rectangular shapes are the settings of the property. If you click the middle rectangle, the control will be docked over the entire form: it will expand and shrink both horizontally and vertically to cover the entire form. This setting is appropriate for simple forms that contain a single control, usually a TextBox, and sometimes a menu. Try it out.

Let's create a more complicated form with two controls (it's the Docking project on the CD). The form shown in Figure 5.8 contains a TreeView control on the left and a ListView control on the right. The two controls display generic data, but the form has the same structure as a Windows Explorer window, with the directory structure in tree form on the left pane and the files of the selected folder on the right pane.

Place a TreeView control on the left side of the form and a ListView control on the right side of the form. Then dock the TreeView to the left and the ListView to the right. If you run the application now, then as you resize the form, the two controls remain docked to the two sides of the form, but their sizes don't change. If you make the form wider, there will be a gap between the two controls. If you make the form narrower, one of the controls will overlap the other.

End the application, return to the Form Designer, select the ListView control, and anchor the control on all four sides. This time, the ListView will change size to take up all the space to the right of the TreeView.



Column1	Colum...	Column3	Column4
Item 1	Item 1-1	Item 1-2	Item 1-3
Item 2			
Item 3	Item 3-1	Item 3-2	Item 3-3
Item 4			
Item 5	Item 5-1	Item 5-2	Item 5-3

FIGURE 5.8
Filling a form with two controls

NOTE *When you anchor a control to the left side of the form, the distance between the control's left side and the form's left edge remains the same. This is the default behavior of the controls. If you dock the right side of the control to the right side of the form, then as you resize the width of the form, the control is moved so that its distance from the right side of the form remains fixed—you can even push the control out of the left edge of the form. If you anchor two opposite sides of the control (top and bottom, or left and right), then the control is resized, so that the docking distances of both sides remain the same. Finally, if you dock all four sides, the control is resized along with the form. Place a multiline TextBox control on a form and try out all possible settings of the Dock property.*

The form behaves better, but it's not what you really expect from a Windows application. The problem with the form of Figure 5.8 is that users can't change the relative widths of the controls. In other words, you can't make one of the controls narrower to make room for the other, which is a fairly common concept in the Windows interface. The narrow bar that allows users to control the relative sizes of two controls is a *splitter*. When the cursor hovers over a splitter, it changes to a double arrow to indicate that the bar can be moved. By moving the splitter, you can enlarge one of the two controls while shrinking the other.

The Form Designer provides a special control for placing a splitter between pairs of controls, and this is the Splitter control. We'll design a new form identical to that of Figure 5.8, only this time we'll place a Splitter control between them, so that users can change the relative size of the two controls. First, place a TextBox control on the form and set its Multiline property to True. You don't need to do anything about its size, because we'll dock it to the left side of the form. With the TextBox control selected, locate its Dock property and set it to Left. The TextBox control will fill the left side of the form, from top to bottom.

Then place an instance of the Splitter control on the form, by double-clicking its icon on the Toolbox. The Splitter will be placed next to the TextBox control. The Form Designer will attempt to dock the Splitter to the left side of the form. Since there's a control docked on this side of the form already, the Splitter will be docked left against the TextBox.

Now place another TextBox control on the form, to the right of the Splitter control. Set the TextBox's Multiline property to True and its Dock property to Fill. We want the second TextBox to fill all the area to the right of the Splitter. Now run the project and check out the functionality of the Splitter. Paste some text on the two controls and then change their relative size by sliding the Splitter between them, as shown in Figure 5.9. You will find this project, called Splitter1, in this chapter's folder on the CD.

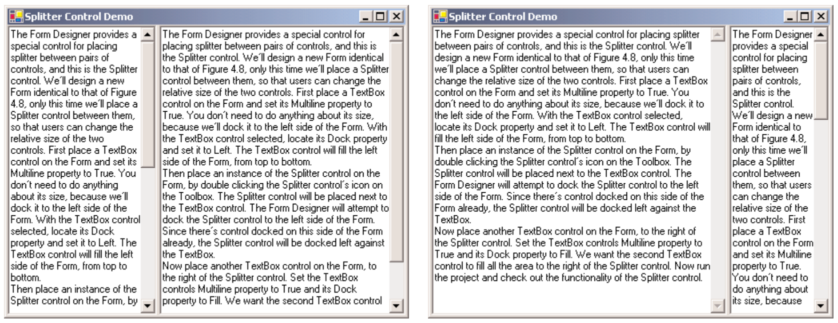


FIGURE 5.9

The Splitter control lets you change the relative size of the controls on either side.

Splitter Control Demo

The Form Designer provides a
special control for placing
splitter between pairs of
controls, and this is the Splitter
control. We'll design a new
Form identical to that of Figure
4.8, only this time we'll place a
Splitter control between them,
so that users can change the
relative size of the two
controls. First place a TextBox
control on the Form and set its
Multiline property to True. You
don't need to do anything
about its size, because we'll
dock it to the left side of the
Form. With the TextBox control
selected, locate its Dock
property and set it to Left. The
TextBox control will fill the left
side of the Form, from top to
bottom.
Then place an instance of the
Splitter control on the Form, by

The Form Designer provides a special control for
placing splitter between pairs of controls, and this is
the Splitter control. We'll design a new Form identical
to that of Figure 4.8, only this time we'll place a Splitter
control between them, so that users can change the
relative size of the two controls. First place a TextBox
control on the Form and set its Multiline property to
True. You don't need to do anything about its size,
because we'll dock it to the left side of the Form. With
the TextBox control selected, locate its Dock property
and set it to Left. The TextBox control will fill the left
side of the Form, from top to bottom.
Then place an instance of the Splitter control on the
Form, by double clicking the Splitter control's icon on
the Toolbox. The Splitter control will be placed next to
the TextBox control. The Form Designer will attempt to
dock the Splitter control to the left side of the Form.
Since there's control docked on this side of the Form
already, the Splitter control will be docked left against
the TextBox.
Now place another TextBox control on the Form, to
the right of the Splitter control. Set the TextBox
controls Multiline property to True and its Dock
property to Fill. We want the second TextBox control

Splitter Control Demo

The Form Designer provides a special control for placing splitter
between pairs of controls, and this is the Splitter control. We'll
design a new Form identical to that of Figure 4.8, only this time
we'll place a Splitter control between them, so that users can
change the relative size of the two controls. First place a TextBox
control on the Form and set its Multiline property to True. You
don't need to do anything about its size, because we'll dock it to
the left side of the Form. With the TextBox control selected,
locate its Dock property and set it to Left. The TextBox control will
fill the left side of the Form, from top to bottom.
Then place an instance of the Splitter control on the Form, by
double clicking the Splitter control's icon on the Toolbox. The
Splitter control will be placed next to the TextBox control. The
Form Designer will attempt to dock the Splitter control to the left
side of the Form. Since there's control docked on this side of the
Form already, the Splitter control will be docked left against the
TextBox.
Now place another TextBox control on the Form, to the right of
the Splitter control. Set the TextBox controls Multiline property to
True and its Dock property to Fill. We want the second TextBox
control to fill all the area to the right of the Splitter control. Now run
the project and check out the functionality of the Splitter control.

The Form Designer
provides a special
control for placing
splitter between
pairs of controls,
and this is the
Splitter control.
We'll design a new
Form identical to
that of Figure 4.8,
only this time we'll
place a Splitter
control between
them, so that users
can change the
relative size of the
two controls. First
place a TextBox
control on the Form
and set its Multiline
property to True.
You don't need to
do anything about
its size, because

Let's design a more elaborate form with two Splitter controls, like the one shown in Figure 5.10 (it's the form of the Splitter2 project on the CD). This form is at the heart of the interface of Outlook, and it consists of a TreeView control on the left (where the folders are displayed), a ListView control (where the selected folder's items are displayed), and a TextBox control (where the selected item's details are displayed). Since we haven't discussed the ListView and TreeView controls yet, I'm using three TextBox controls. The process of designing the form is identical, regardless of the controls you put on it.

Before explaining the process in detail, let me explain how the form shown in Figure 5.10 is different from the one in Figure 5.9. The vertical Splitter allows you to change the size of the TextBox on the left; the remaining space on the form must be taken by the other two controls. A Splitter control, however, must be placed between two controls (no more, no less). By placing a Panel control on the right side of the form, we use the vertical Splitter to separate the TextBox control to the left and the Panel control to the right. The other two TextBox controls and the horizontal Splitter are arranged on the Panel as you would arrange them on a form. Let's build this form.

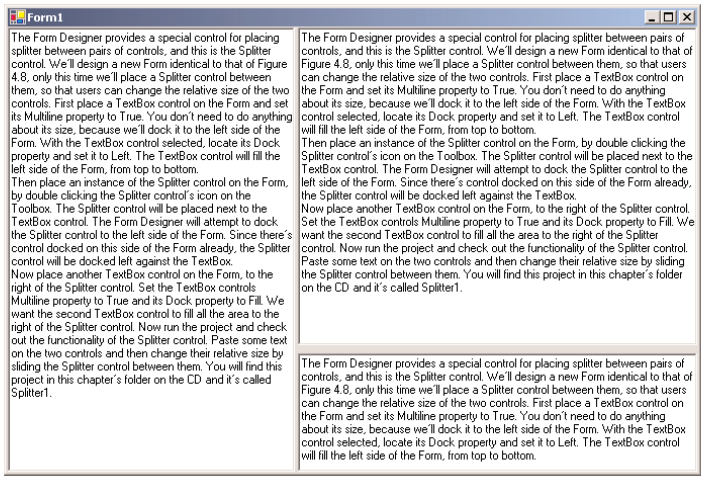
First, place a multiline TextBox control on the form and dock it to the right. Then place a Splitter control, which will be docked to the left by default. Since there's a control docked to the left of the form already, the Splitter control will be docked to the right side of that control. Then place a Panel control to the left of the Splitter control and set its Dock property to Fill. So far, you've done exactly what you did in the last example. If you run the application now, you'll be able to resize the two controls on the form.

Now we're going to place two TextBox controls on the Panel control, separated by a horizontal Splitter control. Place the first multiline TextBox control and dock it to the top of the Panel. Then place a Splitter control on the Panel. The Form Designer will attempt to dock it to the left of the control, so there's no point in trying to resize the Splitter control with the mouse. Just change its Dock property from Left to Top. Finally, place the third TextBox on the Panel, and set its Multiline property to True and its Dock property to Fill. The last TextBox will fill the available area of the Panel below the Splitter. Run the application, paste some text on all three TextBox controls, and then use the two Splitter controls to resize the TextBoxes any way you like. Any VB6 programmer will tell you that this is a very elaborate interface—they just can't guess how many lines of code you wrote.

So far, you've seen what the Form Designer and the Form object can do for your application. Let's switch our focus to programming forms.

FIGURE 5.10

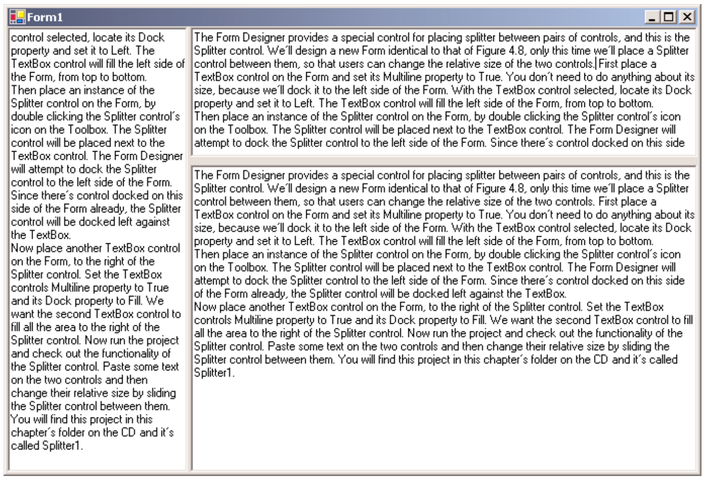
An elaborate form with two Splitter controls.


Form1

The Form Designer provides a special control for placing splitter between pairs of controls, and this is the Splitter control. We'll design a new Form identical to that of Figure 4.8, only this time we'll place a Splitter control between them, so that users can change the relative size of the two controls. First place a TextBox control on the Form and set its Multiline property to True. You don't need to do anything about its size, because we'll dock it to the left side of the Form. With the TextBox control selected, locate its Dock property and set it to Left. The TextBox control will fill the left side of the Form, from top to bottom.
Then place an instance of the Splitter control on the Form, by double clicking the Splitter control's icon on the Toolbox. The Splitter control will be placed next to the TextBox control. The Form Designer will attempt to dock the Splitter control to the left side of the Form. Since there's control docked on this side of the Form already, the Splitter control will be docked left against the TextBox.
Now place another TextBox control on the Form, to the right of the Splitter control. Set the TextBox controls Multiline property to True and its Dock property to Fill. We want the second TextBox control to fill all the area to the right of the Splitter control. Now run the project and check out the functionality of the Splitter control. Paste some text on the two controls and then change their relative size by sliding the Splitter control between them. You will find this project in this chapter's folder on the CD and it's called Splitter1.

The Form Designer provides a special control for placing splitter between pairs of controls, and this is the Splitter control. We'll design a new Form identical to that of Figure 4.8, only this time we'll place a Splitter control between them, so that users can change the relative size of the two controls. First place a TextBox control on the Form and set its Multiline property to True. You don't need to do anything about its size, because we'll dock it to the left side of the Form. With the TextBox control selected, locate its Dock property and set it to Left. The TextBox control will fill the left side of the Form, from top to bottom.
Then place an instance of the Splitter control on the Form, by double clicking the Splitter control's icon on the Toolbox. The Splitter control will be placed next to the TextBox control. The Form Designer will attempt to dock the Splitter control to the left side of the Form. Since there's control docked on this side of the Form already, the Splitter control will be docked left against the TextBox.
Now place another TextBox control on the Form, to the right of the Splitter control. Set the TextBox controls Multiline property to True and its Dock property to Fill. We want the second TextBox control to fill all the area to the right of the Splitter control. Now run the project and check out the functionality of the Splitter control. Paste some text on the two controls and then change their relative size by sliding the Splitter control between them. You will find this project in this chapter's folder on the CD and it's called Splitter1.

The Form Designer provides a special control for placing splitter between pairs of controls, and this is the Splitter control. We'll design a new Form identical to that of Figure 4.8, only this time we'll place a Splitter control between them, so that users can change the relative size of the two controls. First place a TextBox control on the Form and set its Multiline property to True. You don't need to do anything about its size, because we'll dock it to the left side of the Form. With the TextBox control selected, locate its Dock property and set it to Left. The TextBox control will fill the left side of the Form, from top to bottom.


Form1

control selected, locate its Dock property and set it to Left. The TextBox control will fill the left side of the Form, from top to bottom.
Then place an instance of the Splitter control on the Form, by double clicking the Splitter control's icon on the Toolbox. The Splitter control will be placed next to the TextBox control. The Form Designer will attempt to dock the Splitter control to the left side of the Form. Since there's control docked on this side of the Form already, the Splitter control will be docked left against the TextBox.
Now place another TextBox control on the Form, to the right of the Splitter control. Set the TextBox controls Multiline property to True and its Dock property to Fill. We want the second TextBox control to fill all the area to the right of the Splitter control. Now run the project and check out the functionality of the Splitter control. Paste some text on the two controls and then change their relative size by sliding the Splitter control between them. You will find this project in this chapter's folder on the CD and it's called Splitter1.

The Form Designer provides a special control for placing splitter between pairs of controls, and this is the Splitter control. We'll design a new Form identical to that of Figure 4.8, only this time we'll place a Splitter control between them, so that users can change the relative size of the two controls. First place a TextBox control on the Form and set its Multiline property to True. You don't need to do anything about its size, because we'll dock it to the left side of the Form. With the TextBox control selected, locate its Dock property and set it to Left. The TextBox control will fill the left side of the Form, from top to bottom.
Then place an instance of the Splitter control on the Form, by double clicking the Splitter control's icon on the Toolbox. The Splitter control will be placed next to the TextBox control. The Form Designer will attempt to dock the Splitter control to the left side of the Form. Since there's control docked on this side

The Form Designer provides a special control for placing splitter between pairs of controls, and this is the Splitter control. We'll design a new Form identical to that of Figure 4.8, only this time we'll place a Splitter control between them, so that users can change the relative size of the two controls. First place a TextBox control on the Form and set its Multiline property to True. You don't need to do anything about its size, because we'll dock it to the left side of the Form. With the TextBox control selected, locate its Dock property and set it to Left. The TextBox control will fill the left side of the Form, from top to bottom.
Then place an instance of the Splitter control on the Form, by double clicking the Splitter control's icon on the Toolbox. The Splitter control will be placed next to the TextBox control. The Form Designer will attempt to dock the Splitter control to the left side of the Form. Since there's control docked on this side of the Form already, the Splitter control will be docked left against the TextBox.
Now place another TextBox control on the Form, to the right of the Splitter control. Set the TextBox controls Multiline property to True and its Dock property to Fill. We want the second TextBox control to fill all the area to the right of the Splitter control. Now run the project and check out the functionality of the Splitter control. Paste some text on the two controls and then change their relative size by sliding the Splitter control between them. You will find this project in this chapter's folder on the CD and it's called Splitter1.

The Form's Events

The Form object triggers several events, the most important of them being Activate, Deactivate, Closing, Resize, and Paint.

The Activate and Deactivate Events

When more than one form is displayed, the user can switch from one to the other with the mouse or by pressing Alt+Tab. Each time a form is activated, the Activate event takes place. Likewise, when a form is activated, the previously active form receives the Deactivate event.

The Closing Event

This event is fired when the user closes the form by clicking its Close button. If the application must terminate because Windows is shutting down, the same event will be fired as well. Users don't always quit applications in an orderly manner, and a professional application should behave gracefully under all circumstances. The same code you execute in the application's Exit command must also be executed from within the Closing event as well. For example, you may display a warning if the user has unsaved data, or you may have to update a database, and so on. Place the code that performs these tasks in a subroutine and call it from within your menu's Exit command, as well as from within the Closing event's handler.

You can cancel the closing of a form by setting the `e.Cancel` property to True. The event handler in Listing 5.3 displays a message box telling the user that the data hasn't been saved and gives them a chance to cancel the action and return to the application.

Listing 5.3: Cancelling the Closing of a Form

```
Public Sub Form1_Closing(ByVal sender As Object, _
              ByVal e As System.ComponentModel.CancelEventArgs) _
              Handles Form1.Closing
   Dim reply As MsgBoxResult
   reply = MsgBox("Current document has been edited. Click OK to terminate " & _
                  "the application, or Cancel to return to your document.", _
                  MsgBoxStyle.OKCancel)
   If reply = MsgBoxResult.Cancel Then
      e.Cancel = True
   End If
End Sub
```

The Resize Event

The Resize event is fired every time the user resizes the form with the mouse. With previous versions of VB, programmers had to insert quite a bit of code in the Resize event's handler to resize the controls and possibly rearrange them on the form. With the Anchor and Dock properties, much of this overhead can be passed to the form itself.

Many VB applications used the Resize event to impose a minimum size for the form. To make sure that the user can't make the form smaller than, say 300 by 200 pixels, you would insert these lines into the Resize event's handler:

```
Private Form1_Resize(ByVal sender As Object, ByVal e As System.EventArgs) _
           Handles Form1.Resize
   If Me.Width < minWidth Then Me.Width = minWidth
   If Me.Height < minHeight Then Me.Height = minHeight
End Sub
```

There's a better approach to imposing a minimum form size, the MinimumSize property, discussed earlier in this chapter. If you want the two sides of the form to maintain a fixed ratio, you will have to resize one of the dimensions from within the Resize event handler. Let's say the form's width must have a ratio of 3:4 to its height. Assuming that you're using the form's height as a guide, insert the following statement in the Resize event handler to make the width equal to three fourths of the height:

```
Private Form1_Resize(ByVal sender As Object, ByVal e As System.EventArgs)
   Me.Width = (0.75 * Me.Height)
End Sub
```

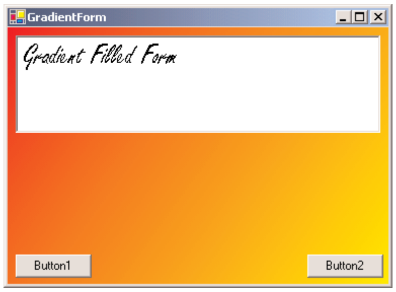
You may also wish to program the Resize event to redraw the form. Normally, this action takes place from within the Paint event, which is fired every time the form must be redrawn. The Paint event, however, isn't fired when the form is reduced in size.

The Paint Event

This event takes place every time the form must be refreshed. When you switch to another form that partially or totally overlaps the current form and then switch back to the first form, the Paint event will be fired to notify your application that it must redraw the form. In this event's handler, we insert the code that draws on the form. The form will refresh its controls automatically, but any custom drawing on the form won't be refreshed automatically. We'll discuss this event in more detail in Chapter 14.

In this section, I'll show you a brief example of using the Paint event. Let's say you want to fill the background of your form with a gradient that starts with red at the top-left corner of the form and ends with yellow at the bottom-right corner, like the one in Figure 5.11. This is the form of the Gradient project, which you will find in this chapter's folder on the CD. Each time the user resizes the form, the gradient must be redrawn, because its exact coloring depends on the form's size and aspect ratio. The PaintForm() subroutine, which redraws the gradient on the form, must be called from within the Paint and Resize events.

FIGURE 5.11
Filling a form's background with a gradient



Before presenting the PaintForm() subroutine, I should briefly discuss the Graphics object. The surface on which you will draw the gradient is a Graphics object, which you can retrieve with the `Me.CreateGraphics` method. The FillRectangle method, which you'll use in this example, is one of the methods of the Graphics object, and it fills a rectangle with a gradient.

To draw on a surface, you must create a brush object (the instrument you'll draw with). One of the built-in brushes is the LinearGradientBrush, which creates a linear gradient. The following statement declares a variable to represents a brush that draws a linear gradient:

```
Dim grbrush As System.Drawing.Drawing2D.LinearGradientBrush
```

To initialize the *grbrush* variable, you must specify the properties of the gradient: its span and its starting and ending colors. One form of the Brush object's constructor is the following:

```
New Brush(origin, dimensions, starting_color, ending_color)
```

The last two arguments are the gradient's starting and ending colors, and they're obvious. The first argument is a point (a pair of *x* and *y* coordinates), while the dimensions of the gradient determine its direction and size. If the height of the gradient is zero, the gradient will be horizontal, and if the width is the same as its height, the gradient is diagonal. If the gradient's dimensions are smaller than the area you want to fill, the gradient will be repeated. To draw a red-to-yellow gradient that fills the form diagonally, the gradient's origin must be the form's top-left corner—the point (0, 0)—and the gradient's dimensions must be the same as the form's dimensions. The following statement creates the brush for the desired gradient:

```
grbrush = New System.Drawing.Drawing2D.LinearGradientBrush(New Point(0, 0), _
                     New Point(Me.Width, Me.Height), Color.Red, Color.Yellow)
```

Finally, the FillRectangle method will draw a filled rectangle with the specified brush. The Fill-Rectangle method accepts as arguments the brush it will use to draw the gradient, the origin of the rectangle, and its dimensions:

```
Me.CreateGraphics.FillRectangle(grbrush, New Rectangle(0, 0, _
                                                       Me.Width, Me.Height))
```

To fill a form with a gradient, enter a RepaintForm() subroutine in the form's code window, then call this subroutine from within the Form's Resize and Paint event handlers, as shown in Listing 5.4.

LISTING 5.4: REPAINTING A GRADIENT ON A FORM

```
Sub RepaintForm()
   Dim grbrush As System.Drawing.Drawing2D.LinearGradientBrush
   grbrush = New System.Drawing.Drawing2D.LinearGradientBrush(New Point(0, 0), _
              New Point(Me.width, Me.height), Color.Red, Color.Yellow)
   Me.CreateGraphics.FillRectangle(grbrush, New Rectangle(0, 0, _
              Me.Width, Me.Height))
End Sub
Public Sub Form1_Paint(ByVal sender As Object, _
              ByVal e As System.WinForms.PaintEventArgs) Handles Form1.Paint
   RepaintForm()
End Sub
Public Sub Form1_Resize(ByVal sender As Object, _
              ByVal e As System.EventArgs) Handles Form1.Resize
   RepaintForm()
End Sub
```

As mentioned earlier, the Paint event is fired every time the form must be redrawn, but not when the form is resized to smaller dimensions. Because the visible area of the form doesn't include any new regions, the Paint event isn't fired—Windows thinks there's nothing to redraw, so why fire a Paint event? The example with the gradient, however, is a special case. When the form is reduced in size, the gradient's colors must also change. They must go from red to yellow in a shorter span, which means that even though the end colors of the gradient will be the same, the actual gradient will look different. Therefore, the statements that draw the form's gradient must be executed from within the Resize event as well.

To experiment with the Paint and Resize events, comment out the call to the subroutine RepaintForm() in one of the two event handlers at a time. Then resize the form, overlap it totally and partially by another form, and bring it to the foreground again. You will notice that unless both event handlers are executed, the form's background gradient isn't properly drawn at all times.

You can request that the Paint event is fired when the form is resized by calling the form's SetStyle method with the following arguments:

```
Me.SetStyle(ControlStyles.ResizeRedraw, True)
```

If you insert this statement in the form's Load event handler, then you need not program the Resize event. The Paint event will be fired every time the form is resized by the user. You'll read a lot about painting and drawing with VB.NET in Chapter 14. In the mean time, you can place background images on your forms by setting the BackgroundImage property.

Loading and Showing Forms

One of the operations you'll have to perform with multi-form applications is to load and manipulate forms from within other forms' code. For example, you may wish to display a second form to prompt the user for data specific to an application. You must explicitly load the second form, read the information entered by the user, and then close the form. Or, you may wish to maintain two forms open at once and let the user switch between them. The entire process isn't trivial, and it's certainly more complicated than it used to be with VB6. You have seen the basics of handling multiple forms in an application in Chapter 2; in this chapter, we'll explore this topic in depth.

To access a form from within another form, you must first create a variable that references the second form. Let's say your application has two forms, named Form1 and Form2, and that Form1 is the project's startup form. To show Form2 when an action takes place on Form1, first declare a variable that references Form2:

```
Dim frm As New Form2
```

This declaration must appear in Form1 and must be placed outside any procedure. (If you place it in a procedure's code, then every time the procedure is executed, a new reference to Form2 will be created. This means that the user can display the same form multiple times. All procedures in Form1 must see the same instance of the Form2, so that no matter how many procedures show Form2, or how many times they do it, they'll always bring up the same single instance of Form2.)

Then, to invoke Form2 from within Form1, execute the following statement:

```
frm.Show
```

This statement will bring up Form2 and usually appears in a button's or menu item's Click event handler. At this point, the two forms don't communicate with one another. However, they're both on the desktop and you can switch between them. There's no mechanism to move information from Form2 back to Form1, and neither form can access the other's controls or variables. To exchange information between two forms, use the techniques described in the section "Controlling One Form from within Another," later in this section.

The Show method opens Form2 in a modeless manner. The two forms are equal in stature on the desktop, and the user can switch between them. You can also display the second form in a modal manner, which means that users won't be able to return to the form from which they invoked it. While a modal form is open, it remains on top of the desktop and you can't move the focus to the any other form of the same application (but you can switch to another application). To open a modal form, use the statement

```
frm.ShowDialog
```

The modal form is, in effect, a dialog box, like the Open File dialog box. You must first select a file on this form and click the Open button, or click the Cancel button, to close the dialog box and return to the form from which the dialog box was invoked. Which brings us to the topic of distinguishing forms and dialog boxes.

A dialog box is simply a modal form. When we display forms as dialog boxes, we change the border of the forms to the setting FixedDialog and invoke them with the ShowDialog method. Modeless forms are more difficult to program, because the user may switch among them at any time. Not only that, but the two forms that are open at once must interact with one another. When the user acts on one of the forms, this may necessitate some changes in the other, and you'll see shortly how this is done. If the two active forms don't need to interact, display one of them as a dialog box.

When you're done with the second form, you can either close it by calling its Close method or hide it by calling its Hide method. The Close method closes the form, and its resources are no longer available to the application. The Hide method sets the Form's Visible property to False; you can still access a hidden form's controls from within your code, but the user can't interact with it. Forms that are displayed often, such as the Find and Replace dialog box of a text processing application, should be hidden, not closed. To the user, it makes no difference whether you hide or close a form. If you hide a form, however, the next time you bring it up with the Show or ShowDialog methods, its controls are in the state they were the last time. This may not be what you want, however. If not, you must reset the controls from within your code before calling the Show or ShowDialog method.

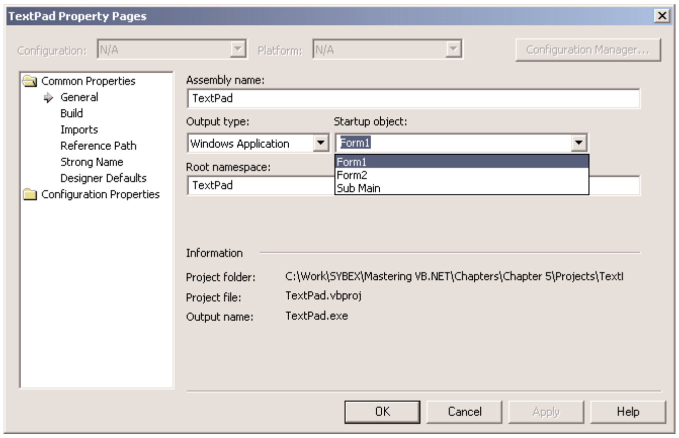
The Startup Form

A typical application has more than a single form. When an application starts, the main form is loaded. You can control which form is initially loaded by setting the startup object in the Project Properties window, shown in Figure 5.12. To open this, right-click the project's name in the Solution Explorer and select Properties. In the project's Property Pages, select the Startup Object from the drop-down list. You can also see other parameters in the same window, which are discussed elsewhere in this book.

By default, Visual Basic suggests the name of the first form it created, which is Form1. If you change the name of the form, Visual Basic will continue using the same form as startup form, with its new name.

FIGURE 5.12

In the Properties window, you can specify the form that's displayed when the application starts.



Assembly name: TextPad
Output type: Windows Application
Startup object: Form1
Form1
Form2
Sub Main
Root namespace: TextPad

Information
Project folder: C:\Work\SYBEX\Mastering VB.NET\Chapters\Chapter 5\Projects\TextI
Project file: TextPad.vbproj
Output name: TextPad.exe

You can also start an application with a subroutine without loading a form. This subroutine must be called Main() and must be placed in a Module. Right-click the project's name in the Solution Explorer window and select the Add Item command. When the dialog box appears, select a Module. Name it StartUp (or anything you like; you can keep the default name Module1) and then insert the Main() subroutine in the module. The Main() subroutine usually contains initialization code and ends with a statement that displays one of the project's forms; to display the AuxiliaryForm object from within the Main() subroutine, use the following statements (I'm showing the entire module's code):

```
Module StartUpModule
   Sub Main()
      System.Windows.Forms.Application.Run(New AuxiliaryForm())
   End Sub
End Module
```

Then, you must open the Project Properties dialog box and specify that the project's startup object is the subroutine Main(). When you run the application, the form you specified in the Run method will be loaded.

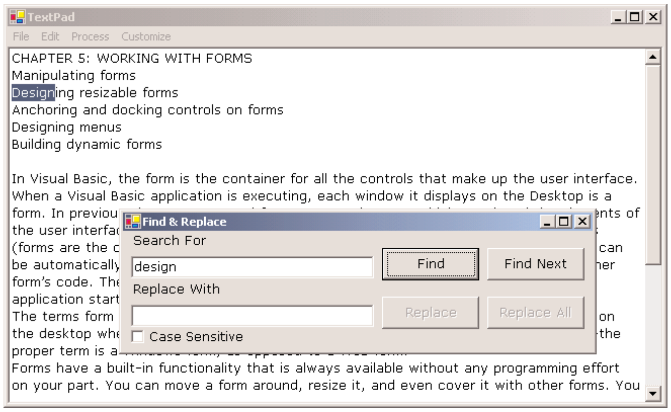
Controlling One Form from within Another

Loading and displaying a form from within another form's code is fairly trivial. In some situations, this is all the interaction you need between forms. Each form is designed to operate independently of the others, but they can communicate via public variables (see the next section, "Private vs. Public Variables"). In most situations, however, you need to control one form from within another's code. Controlling the form means accessing its controls and setting or reading values from within another form's code.

Look at the two forms in Figure 5.13, for instance. These are forms of the TextPad application, which we are going to develop in Chapter 6. TextPad is a text editor that consists of the main form and an auxiliary form for the Find & Replace operation. All other operations on the text are performed with the commands of the menu you see on the main form. When the user wants to search for and/or replace a string, the program displays another form on which they specify the text to find, the type of search, and so on. When the user clicks one of the Find & Replace form's buttons, the corresponding code must access the text on the main form of the application and search for a word or replace a string with another. The Find & Replace dialog box not only interacts with the TextBox control on the main form, it also remains visible at all times while it's open, even if it doesn't have the focus, because its TopMost property was set to True. You'll see how this works in Chapter 6. In this chapter, we'll develop a simple example to demonstrate how you can access another form's controls.

FIGURE 5.13

The Find & Replace form acts on the contents of a control on another form.



```
CHAPTER 5: WORKING WITH FORMS
Manipulating forms
Designing resizable forms
Anchoring and docking controls on forms
Designing menus
Building dynamic forms

In Visual Basic, the form is the container for all the controls that make up the user interface.
When a Visual Basic application is executing, each window it displays on the Desktop is a
form. In previou                                                                   ents of
the user interfac
(forms are the c                                                                   can
be automatically                                                                   her
form's code. The
application start
The terms form                                                                     on
the desktop whe                                                                    the
proper term is a
Forms have a built-in functionality that is always available without any programming effort
on your part. You can move a form around, resize it, and even cover it with other forms. You
```

Search For: design

SHARING VARIABLES BETWEEN FORMS

The simplest method for two forms to communicate with each other is via *public variables*. These variables are declared in the form's declarations section, outside any procedure, with the keywords `Public Shared`. If the following declarations appear in Form1, the variable *NumPoints* and the array *DataValues* can be accessed by any procedure in Form1, as well as from within the code of any form belonging to the same project.

```
Public Shared NumPoints As Integer
Public Shared DataValues(100) As Double
```

To access a public variable declared in Form1 from within another form's code, you must prefix the variable's name by the name of the form, as in:

```
FRM.NumPoints = 99
FRM.DataValues(0) = 0.3395022
```

where *FRM* is a variable that references the form in which the public variables were declared. You can use the same notation to access the controls on the form represented by the *FRM* variable. If the form contains the TextBox1 control, you can use the following statement to read its text:

```
FRM.TextBox1.Text
```

Another technique for exposing the controls of a form to the code of the other forms of the application is to create `Public Shared` variables that represent the controls to be shared. The following declaration makes the *TBox* variable of Form1 available to all other forms in the application:

```
Public Shared TBox As TextBox
```

To make this variable represent a TextBox control, assign to it the name of the control:

```
TBox = TextBox1
```

This statement appears usually in the form's Load() subroutine, but it can appear anywhere in your code. It just has to be executed before you show another form. To access the TextBox1 control on Form1 from within another form's code, use the following expression:

```
Form1.TBox
```

This expression represents a TextBox control, and you can call any of the TextBox control's properties and methods:

```
Form1.TBox.Length                ' returns the length of the text
Form1.TBox.Append("some text")   ' appends text
```

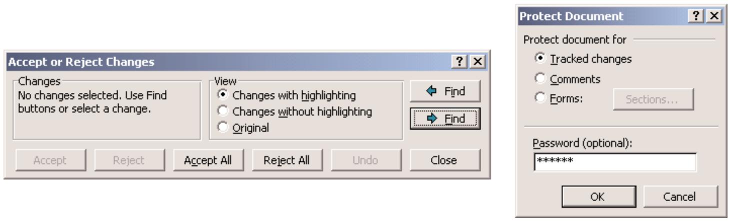
Keep in mind that the controls you want to access from within another form's code must be declared with as `Public Shared`, not just `Public`.

Forms vs. Dialog Boxes

Dialog boxes are special types of forms with rather limited functionality, which we use to prompt the user for data. The Open and Save dialog boxes are two of the most familiar dialog boxes in Windows. They're so common, they're actually known as *common dialog boxes*. Technically, a dialog box is a good old Form with its BorderStyle property set to FixedDialog. Like forms, dialog boxes may contain a few simple controls, such as Labels, TextBoxes, and Buttons. You can't overload a dialog box with controls and functionality, because you'll end up with a regular form.

Figure 5.14 shows a few dialog boxes you have certainly seen while working with Windows applications. The Protect Document dialog box of Word is a modal dialog box: You must close it before switching to your document. The Accept or Reject Changes dialog box is modeless, like the Find and Replace dialog box. It allows you to switch to your document, yet it remains visible while open even if it doesn't have the focus.

Notice that some dialog boxes, such as Open, Color, and even the humble MessageBox, come with the .NET Framework, and you can incorporate them in your applications without having to design them.



No changes selected. Use Find buttons or select a change.

View: [✓ Changes with highlighting] Changes without highlighting Original

Protect document for: [✓ Tracked changes] Comments Forms:

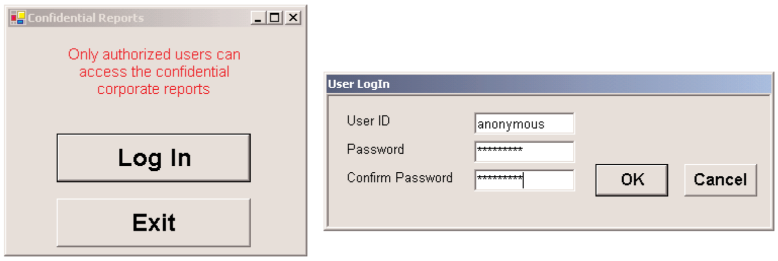
Password (optional): ******

FIGURE 5.14
Typical dialog boxes used by Word

Another difference between forms and dialog boxes is that forms usually interact with each other. If you need to keep two windows open and allow the user to switch from one to the other, you need to implement them as regular forms. If one of them is modal, then you should implement it as a dialog box. A characteristic of dialog boxes is that they provide an OK and a Cancel button. The OK button tells the application that you're done using the dialog box and the application can process the information on it. The Cancel button tells the application that it should ignore the information on the dialog box and cancel the current operation. As you will see, dialog boxes allow you to quickly find out which button was clicked to close them, so that your application can take a different action in each case.

In short, the difference between forms and dialog boxes is artificial. If it were really important to distinguish between the two, they'd be implemented as two different objects—but they're the same object. So, without any further introduction, let's look at how to create and use dialog boxes.

To create a dialog box, start with a Windows Form, set its BorderStyle property to FixedDialog and set the ControlBox, MinimizeBox, and MaximizeBox properties to False. Then add the necessary controls on the form and code the appropriate events, as you would do with a regular Windows form. Figure 5.15 shows a simple dialog box that prompts the user for an ID and a password. The dialog box contains two TextBox controls, next to the appropriate labels, and the usual OK and Cancel buttons. The Cancel button signifies that the user wants to cancel the operation, which was initiated in the form that displayed the dialog box. The forms of Figure 5.15 are the Password project on the CD.



Only authorized users can access the confidential corporate reports

User ID: anonymous
Password: *********
Confirm Password: *********

FIGURE 5.15
A simple dialog box that prompts users for a username and password

Start a new project, rename the form to MainForm, and place a button on the form. This is the application's main form, and we'll invoke the dialog box from within the button's Click event handler. Then add a new form to the project, name it PasswordForm, and place on it the controls shown in Figure 5.15.

We have the dialog box, but how do we initiate it from within another form's code? The process of displaying a dialog box is no different than displaying another form. To do so, enter the following code in the event handler from which you want to initiate the dialog box (this is the Click event handler of the main form's button):

```
Private Sub Button1_Click(ByVal sender As System.Object, _
               ByVal e As System.EventArgs) Handles Button1.Click
   Dim DLG as new PasswordForm()
   DLG.ShowDialog
End Sub
```

Here, *PasswordForm* is the name of the dialog box. The ShowDialog method displays a dialog box as modal; to display a modeless dialog box, use the Show method instead. An important distinction between modal and modeless dialog boxes has to do with the calling application. When you display a modal dialog box, the statement following the one that called the ShowDialog method is not executed. The statements from this point to the end of the event handler will be executed when the user *closes* the dialog box. Statements following the Show method, however, are executed immediately as soon as the dialog box is displayed.

You already know how to read the values entered on the controls of the dialog box. You also need to know which button was clicked to close the dialog box. To convey this information from the dialog box back to the calling application, the Form object provides the DialogResult property. This property can be set to one of the values shown in Table 5.5 to indicate what button was clicked. The `DialogResult.OK` value indicates that the user has clicked the OK button on the form. There's no need to place an OK button on the form; just set the form's DialogResult property to `DialogResult.OK`.

Table 5.5: The DialogResult Enumeration

Value	Description
`Abort`	The dialog box was closed with the Abort button.
`Cancel`	The dialog box was closed with the Cancel button.
`Ignore`	The dialog box was closed with the Ignore button.
`No`	The dialog box was closed with the No button.
`None`	The dialog box hasn't been closed yet. Use this option to find out whether a modeless dialog box is still open.
`OK`	The dialog box was closed with the OK button.
`Retry`	The dialog box was closed with the Retry button.
`Yes`	The dialog box was closed with the Yes button.

The dialog box need not contain any of the buttons mentioned here. It's your responsibility to set the value of the DialogResult property from within your code to one of the settings shown in the table. This value can be retrieved by the calling application. Notice also that the action of assigning a value to the DialogResult property also closes the dialog box—you don't have to call the Close method explicitly.

Let's say your dialog box contains a button named Done, which signifies that the user is done entering values on the dialog box. The Click event handler for this button contains a single line:

```
Me.DialogResult = DialogResult.OK
```

This statement sets the DialogResult property, which will be read by the code of the form that invoked the dialog box, and also closes the dialog box. The event handler of the button that displays this dialog box should contain these lines:

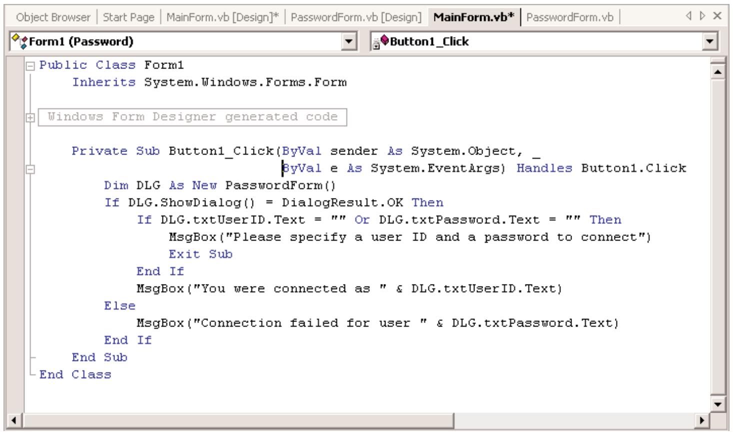
```
Dim DLG as Form = new PasswordForm
If DLG.ShowDialog = DialogResult.OK Then
```

```
    { process the user selection }
End If
```

Figure 5.16 demonstrates how this is done in the Password project.

FIGURE 5.16

The code window of the Password project's main form



```
Public Class Form1
    Inherits System.Windows.Forms.Form

 Windows Form Designer generated code 

    Private Sub Button1_Click(ByVal sender As System.Object, _
                        ByVal e As System.EventArgs) Handles Button1.Click
        Dim DLG As New PasswordForm()
        If DLG.ShowDialog() = DialogResult.OK Then
            If DLG.txtUserID.Text = "" Or DLG.txtPassword.Text = "" Then
                MsgBox("Please specify a user ID and a password to connect")
                Exit Sub
            End If
            MsgBox("You were connected as " & DLG.txtUserID.Text)
        Else
            MsgBox("Connection failed for user " & DLG.txtPassword.Text)
        End If
    End Sub
End Class
```

The dialog box may actually contain two buttons, one of them called Activate or Register Now and the other called Cancel or Remind Me Later. In addition, the dialog box may contain any number of buttons. You decide which buttons will close the form and enter the statement that sets the DialogResult property in their Click event handlers. The value of the DialogResult property is usually set from within two buttons—one that accepts the data and one that rejects them. Depending on your application, you may allow the user to close the dialog box by clicking more than two buttons. Some of them must set the DialogResult property to `DialogResult.OK`, others to `DialogResult.Abort`.

NOTE *Of course, you can read the values of the controls on the dialog box anyway—it's your application and you can do whatever you wish with it. If the user has closed the dialog box with the Cancel button, however, the information is incorrect, and any results your application generates based on these values will also be incorrect.*

The DialogResult property applies to buttons as well. You can close the dialog box and pass the appropriate information to the calling application by setting the DialogResult property of a button to one of the members of the DialogResult enumeration in the Properties window. If you also set one of the buttons on the form to be the Accept button and another to be the Cancel button, you don't have to enter a single line of code in the modal form. The user can enter values on the various controls and then close the dialog box by pressing the Enter or Cancel key. The dialog box will close and will return the `DialogResult.OK` or `DialogResult.Cancel` value.

The dialog box doesn't contain a single line of code. Just make sure the Form's AcceptButton property is bttnOK, the CancelButton property is bttnCancel, and the DialogResult properties of the two buttons are OK and Cancel, respectively. The AcceptButton sets the form's DialogResult property to `DialogResult.OK` automatically, and the CancelButton sets the same property to `DialogResult.Cancel`. Any other button must set the DialogResult property explicitly. Listing 5.5 shows the code behind the Log In button on the main form.

LISTING 5.5: PROMPTING THE USER FOR AN ID AND A PASSWORD

```
Private Sub Button1_Click(ByVal sender As System.Object, _
               ByVal e As System.EventArgs) Handles Button1.Click
   Dim DLG As New PasswordForm()
   If DLG.ShowDialog() = DialogResult.OK Then
      If DLG.txtUserID.Text = "" Or DLG.txtPassword.Text = "" Then
         MsgBox("Please specify a user ID and a password to connect")
         Exit Sub
      End If
      MsgBox("You were connected as " & DLG.txtUserID.Text)
   Else
      MsgBox("Connection failed for user " & DLG.txtPassword.Text)
   End If
End Sub
```

The code of the main form reads the values of the controls on the dialog box through the *DLG* variable. If the dialog box contains many controls, it's better to communicate the data back to the calling application through properties. All you have to do is create a Property procedure for each control and then read the values entered by the user as properties. The topic of Property procedures is discussed in detail in Chapter 8, but it's nothing really complicated. To keep the complexity to a minimum, you can also implement the properties with `Public Shared` variables. Let's say that the dialog box prompts the user to select a state on a ComboBox control. To create a State property, use the following declaration:

```
Public Shared State As String
```

This variable will be exposed by the dialog box as a property, and the application that invoked the dialog box can read the value of the State property with a statement like `DLG.State`.

The value of the *State* variable must be set each time the user selects a state on the ComboBox control, from within the control's SelectedIndexChanged event handler:

```
State = cmbStates.Text
```

where *cmbStates* is the name of the ComboBox control. The user may change their mind and repeat the action of selecting a state. The most recently selected state's name will be stored in the variable *State,* because the SelectedIndexChanged event takes place every time the user makes another selection.

You can invoke the dialog box and then read the value of the *State* variable from within your code with the following statements:

```
Dim Dlg as StatesDialogBox = new StatesDialogBox
Dlg.ShowDialog
If Dlg.DialogResult = DialogResult.OK Then
   Console.WriteLine(Dlg.State)
End If
```

This is a good place to demonstrate how to design multiple interacting forms and dialog boxes with an example.

VB.NET at Work: The MultipleForms Project

It's time to write an application that puts together the most important topics discussed in this section. There's quite a bit to learn about projects with multiple forms, and this is the topic of the following sections. Most of the aspects discussed here are demonstrated in the MultipleForms project, which you will find on the CD. I suggest you follow the steps outlined in the text to build the project on your own.

The MultipleForms project consists of a main form, an auxiliary form, and a dialog box. All three components of the application's interface are shown in Figure 5.17. The buttons on the main form display both the auxiliary form and the dialog box.

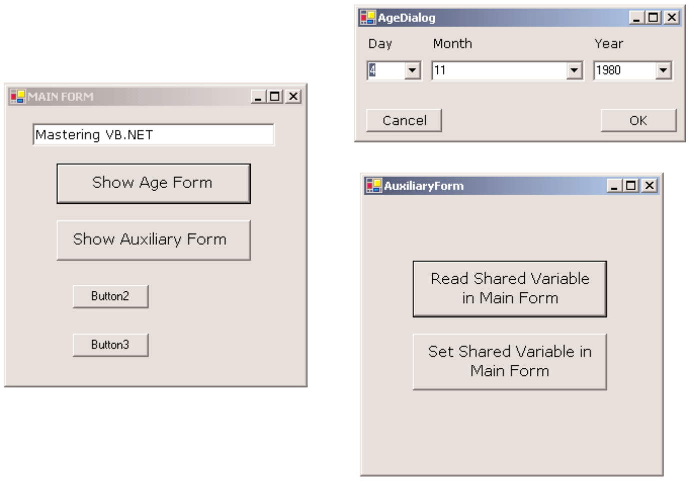


FIGURE 5.17 The MultipleForms project's interface

Let's review the various operations we want to perform—they're typical for many situations, not specific to this application. At first, we must be able to invoke both the auxiliary form and the dialog box from within the main form; the Show Auxiliary Form and Show Age Form buttons do this. The main form contains a variable declaration, *strProperty*. This variable is, in effect, a property of the main form and is declared with the following statement:

```
Public Shared strProperty As String = "Mastering VB.NET"
```

The main form's code declares a variable that represents the auxiliary form and then calls its Show method to display the auxiliary form. The declaration must appear in the form's declarations section:

```
Dim FRM As New AuxiliaryForm()
```

The Show Auxiliary Form button contains a single statement, which invokes the auxiliary form by calling the Show method of the *FRM* variable.

The auxiliary-form button named Read Shared Variable In Main Form reads the *strProperty* variable of the main form with the following statement:

```
Private Sub bttnReadShared_Click(ByVal sender As System.Object, _
            ByVal e As System.EventArgs) Handles bttnReadShared.Click
```

```
    MsgBox(MainForm.strProperty, MsgBoxStyle.OKOnly, "Public Variable Value")
End Sub
```

Using the same notation, you can set this variable from within the auxiliary form. The following event handler prompts the user for a new value and assigns it to the shared variable of the main form. You can confirm that the value has changed by reading it again.

```
Private Sub bttnSetShared_Click(ByVal sender As System.Object, _
              ByVal e As System.EventArgs) Handles bttnSetShared.Click
    Dim str As String
    str = InputBox("Enter a new value for strProperty")
    MainForm.strProperty = str
End Sub
```

The two forms communicate with each other through public variables. Let's make this communication a little more elaborate by adding an event. Every time the auxiliary form sets the value of the *strProperty* variable, it will raise an event to notify the main form. The main form, in turn, will use this event to display the new value of the string on the TextBox control as soon as the code in the auxiliary form changes the value of the variable and before it's closed.

To raise an event, you must declare the event's name in the form's declaration section. Insert the following statement in the auxiliary form's declarations section:

```
Event strPropertyChanged()
```

Now add a statement that fires the event. To raise an event, we call the RaiseEvent statement passing the name of the event as argument. This statement must appear in the Click event handler of the Set Shared Variable In Main Form button, right after setting the value of the shared variable. Listing 5.6 shows the revised event handler.

Listing 5.6: Raising an Event

```
Private Sub bttnSetShared_Click(ByVal sender As System.Object, _
              ByVal e As System.EventArgs) Handles bttnSetShared.Click
    Dim str As String
    str = InputBox("Enter a new value for strProperty")
    MainForm.strProperty = str
    RaiseEvent strPropertyChanged
End Sub
```

The event will be raised, but it will go unnoticed if we don't handle it from within the main form's code. To handle the event, you must change the declaration of the *FRM* variable from

```
Dim FRM As New AuxiliaryForm()
```

to

```
Dim WithEvents FRM As New AuxiliaryForm()
```

The WithEvents keyword tells VB that the variable is capable of raising events. If you expand the drop-down list with the objects in the code editor, you will see the name of the *FRM* variable, along with the other controls you can program. Select *FRM* in the list and then expand the list of events for the selected item. In this list, you will see the `strPropertyChanged` event. Select it, and the definition of an event handler will appear. Enter these statements in this event's handler:

```
Private Sub FRM_strPropertyChanged() Handles FRM.strPropertyChanged
   TextBox1.Text = strProperty
   Beep()
End Sub
```

It's a very simple handler, but it's adequate for demonstrating how to raise and handle custom events. If you run the application now, you'll see that the value of the TextBox control changes as soon as you change the value in the auxiliary form.

Of course, you can update the TextBox control on the main form directly from within the auxiliary form's code. Use the expression `MainForm.TextBox1` to access the control and then manipulate it as usual. Events are used when we want to perform some actions on a form when an action takes place in one of the other forms of the application.

Let's see now how the main form interacts with the dialog box. What goes on between a form and a dialog box is not exactly "interaction"—it's a more timid type of behavior. The form displays the dialog box and then waits until the user closes the dialog box. Then, it looks at the value of the DialogResult property to find out whether it should even examine the values passed back by the dialog box. If the user has closed the dialog box with the Cancel (or an equivalent) button, the application ignores the dialog box settings. If the user closed the dialog box with the OK button, the application reads the values and proceeds accordingly.

Before showing the dialog box, the code of the Show Dialog Box button sets the values of certain controls on it. In the course of the application, it usually makes sense to suggest a few values on the dialog box, so that the user can accept the default values. The main form selects a date on the controls that display the date, and then displays the dialog box with the statements given in Listing 5.7.

LISTING 5.7: DISPLAYING A DIALOG BOX AND READING ITS VALUES

```
Protected Sub Button3_Click(ByVal sender As Object, _
                 ByVal e As System.EventArgs)
' Preselects the date 4/11/1980
   DLG.cmbMonth.Text = "4"
   DLG.cmbDay.Text = "11"
   DLG.CmbYear.Text = "1980"
   DLG.ShowDialog()
   If DLG.DialogResult = DialogResult.OK Then
      MsgBox(DLG.cmbMonth.Text & " " & DLG.cmbDay.Text & ", " & _
             DLG.cmbYear.Text)
   Else
      MsgBox("OK, we'll protect your vital personal data")
   End If
End Sub
```

The *DLG* variable is declared on the Form level with the following statement:

```
Dim DLG As New AgeDialog()
```

The dialog box is modal: you can't switch to the main form while the dialog box is displayed. To close the dialog box, you can click one of the OK or Cancel buttons. Each button sets the Dialog-Result property to indicate the action that closed the dialog box. The code behind the two buttons is shown in Listing 5.8.

Listing 5.8: Setting the Dialog Box's DialogResult Property

```
Protected Sub bttnOK_Click(ByVal sender As Object, ByVal e As System.EventArgs)
   Me.DialogResult = DialogResult.OK
End Sub
Protected Sub bttnCancel_Click(ByVal sender As Object, _
                               ByVal e As System.EventArgs)
   Me.DialogResult = DialogResult.Cancel
End Sub
```

Since the dialog box is modal, the code in the Show Dialog Box button is suspended at the line that shows the dialog box. As soon as the dialog box is closed, the code in the main form resumes with the statement following the one that called the ShowDialog method of the dialog box. This is the If statement in Listing 5.7 that examines the value of the DialogResult property and acts accordingly.

Designing Menus

Menus are one of the most common and characteristic elements of the Windows user interface. Even in the old days of character-based displays, menus were used to display methodically organized choices and guide the user through an application. Despite the visually rich interfaces of Windows applications and the many alternatives, menus are still the most popular means of organizing a large number of options. Many applications duplicate some or all of their menus in the form of toolbar icons, but the menu is a standard fixture of a form. You can turn the toolbars on and off, but not the menus.

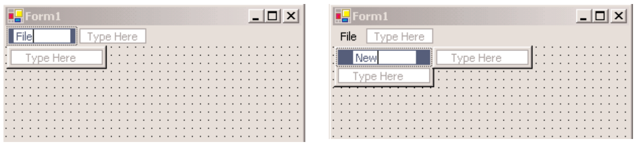
The Menu Editor

Menus can be attached only to forms, and they're implemented through the MainMenu control. The items that make up the menu are MenuItem objects. As you will see, the MainMenu control and MenuItem objects give you absolute control over the structure and appearance of the menus of your application.

The IDE provides a visual tool for designing menus, and then you can program their Click event handlers. In principle, that's all there is to a menu: you design it, then you program each command's actions. Depending on the needs of your application, you may wish to enable and disable certain commands, add context menus to some of the controls on your form, and so on. Because each item (command) in a menu is represented by a MenuItem object, you can control the application's menus from within your code by manipulating the properties of the MenuItem objects. Let's start by designing a simple menu, and I'll show you how to manipulate the menu objects from within your code we go along.

Double-click the MainMenu icon on the Toolbox. The MainMenu control will be added to the form, and a single menu command will appear on your form. Its caption will be Type Here. If you don't see the first menu item on the Form right away, select the MainMenu control in the Components tray below the form. Do as the caption says; click it and enter the first command's caption, **File**, as seen in Figure 5.18. As soon as you start typing, two more captions appear: one on the same level (the second command of the form's main menu, representing the second pull-down menu) and another one below File (representing the first command on the File menu). Select the item under File and enter the string **New**.

FIGURE 5.18

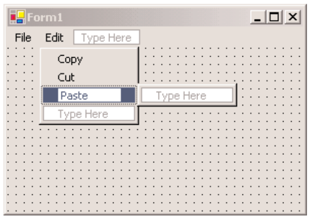


As soon as you start entering the caption of a menu or menu item, more items appear to the left and below the current item.

Enter the remaining items of the File menu—**Open**, **Save**, and **Exit**—and then click somewhere on the form. All the temporary items (the ones with the Type Here caption) will disappear, and the menu will be finalized on the form. At any point, you can add more items by right-clicking one of the existing menu items and selecting Insert New.

To add the Edit menu, select the MainMenu icon to activate the visual menu editor and then click the File item. In the new item that appears next to that, enter the string **Edit**. Press Enter and you'll switch to the first item of the Edit menu. Fill the Edit menu with the commands shown in Figure 5.19. Table 5.6 shows the captions (property Text) and names (property Name) for each menu and each command.

FIGURE 5.19



Type these standard commands on the Edit menu.

The left-most items in Table 5.6 are the names of the first-level menus (File and Edit); the captions that are indented in the table are the commands on these two menus. Each menu item has a name, which allows you to access the properties of the menu item from within your code. The same name is also used in naming the Click event handler of the item. The default names of the menu items you add visually to the application's menu are MenuItem1, MenuItem2, and so on. To change the default names to something more meaningful, you can change the Name property in the Properties window. To view the properties of a menu item, select it with the left mouse button, then right-click it and select Properties from the context menu.

TABLE 5.6: THE CAPTIONS AND NAMES OF THE FILE AND EDIT MENUS

CAPTION	NAME
File	FileMenu
New	FileNew
Open	FileOpen
Save	FileSave
Exit	FileExit
Edit	EditMenu
Copy	EditCopy
Cut	EditCut
Paste	EditPaste

Alternatively, you can select Edit Names from the context menu. This action lets you edit the names of the menu items right on the menu structure, as if you were changing the captions. The captions appear next to the names of the items, but you can only edit the names. Figure 5.20 shows a menu structure in name-editing mode. When you're done renaming the items, right-click somewhere on the menu and select the Edit Names command again. The check mark next to the Edit Names option will clear, and you'll be switched back to editing the captions.

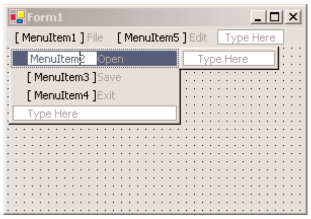


FIGURE 5.20
Editing the names of the items in your menu

To create a separator bar in a menu, right-click the item you want to display *below* the separator and select Insert Separator. Separator bars divide menu items into logical groups, and even though they have the structure of regular menu commands, they don't react to the mouse click. You can also create a separator bar by setting the item's caption to a dash (-).

As you will notice, the menus expand by default to the bottom and to the right. To insert a menu command to the left of an existing command, or to insert a menu item above an existing menu item, right-click the item *following* the one you want to insert and select Insert New.

The MenuItem Object's Properties

The MenuItem object represents a menu command, at any level. If a command leads to a submenu, it's still represented by a MenuItem object, which has its own collection of MenuItem objects. Each individual command is represented by a MenuItem object. The MenuItem object provides the following properties, which you can set in the Properties window at design time or manipulate from within your code:

Checked Some menu commands act as toggles, and they are usually checked to indicate that they are on or unchecked to indicate that they are off. To initially display a check mark next to a menu command, right-click the menu item, select Properties, and check the Checked box in its Properties window. You can also access this property from within your code to change the checked status of a menu command at runtime. For example, to toggle the status of a menu command called FntBold, use the statement:

```
FntBold.Checked = Not FntBold.Checked
```

DefaultItem This property is a True/False value that indicates whether the MenuItem is the default item in a submenu. The default item is displayed in bold and is automatically activated when the user double-clicks a menu that contains it.

Enabled Some menu commands aren't always available. The Paste command, for example, has no meaning if the Clipboard is empty (or if it contains data that can't be pasted in the current application). To indicate that a command can't be used at the time, you set its Enabled property to False. The command then appears grayed in the menu, and although it can be highlighted, it can't be activated. The following statements enable and disable the Undo command depending on whether the TextBox1 control can undo the most recent operation.

```
If TextBox1.CanUndo Then
   cmdUndo.Enabled = True
Else
   cmdUndo.Enabled = False
End If
```

cmdUndo is the name of the Undo command in the application's Edit menu. The CanUndo property of the TextBox control returns a True/False value indicating whether the last action can be undone or not.

IsParent If the menu command, represented by a MenuItem object, leads to a submenu, then that MenuItems object's IsParent property is True. Otherwise, it's False. The IsParent property is read-only.

Mnemonic This read-only property returns the character that was assigned as an access key to the specific menu item. If no access key is associated with a MenuItem, the character 0 will be returned.

Visible To remove a command temporarily from the menu, set the command's Visible property to False. The Visible property isn't used frequently in menu design. In general, you should prefer to disable a command to indicate that it can't be used at the time (some other action is required to enable it). Making a command invisible frustrates users, who may try to locate the command in another menu.

MDIList This property is used with Multiple Document Interface (MDI) applications to maintain a list of all open windows. The MDIList property is explained in Chapter 19.

Programming Menu Commands

Menu commands are similar to controls. They have certain properties that you can manipulate from within your code, and they trigger a Click event when they're clicked with the mouse or selected with the Enter key. If you double-click a menu command at design time, Visual Basic opens the code for the Click event in the code window. The name of the event handler for the Click event is composed of the command's name followed by an underscore character and the event's name, as with all other controls.

To program a menu item, insert the appropriate code in the MenuItem's Click event handler. A related event is the Select event, which is fired when the cursor is placed over a menu item, even if it's not clicked. The Exit command's code would be something like:

```
Sub menuExit(ByVal sender As Object, ByVal e As System.EventArgs) _
             Handles menuExit.Click
   End
End Sub
```

If you need to execute any clean-up code before the application ends, place it in the CleanUp() subroutine and call this subroutine from within the Exit item's Click event handler:

```
Sub menuExit(ByVal sender As Object, ByVal e As System.EventArgs) _
             Handles menuExit.Click
   CleanUp()
   End
End Sub
```

The same subroutine must also be called from within the Closing event handler of the application's main form, as some users might terminate the application by clicking the form's Close button.

An application's Open menu command contains the code that prompts the user to select a file and then open it. You will see many examples of programming menu commands in the following chapters. All you really need to know is that each menu item is a MenuItem object, and it fires the Click event every time it's selected with the mouse or the keyboard. In most cases, you can treat the Click event handler of a MenuItem object just like the Click event handler of a Button.

You can also program multiple menu items with a single event handler. Let's say you have a Zoom menu that allows the user to select one of several zoom factors. Instead of inserting the same statements in each menu item's Click event handler, you can program all the items of the Zoom menu with a single event handler. Select all the items that share the same event handler (click them with the mouse while holding down the Shift button). Then click the Event button on the Properties window and select the event that you want to be common for all selected items.

The handler of the Click event of a menu item has the following declaration:

```
Private Sub Zoom200_Click(ByVal sender As System.Object, _
               ByVal e As System.EventArgs) Handles Zoom200.Click
End Sub
```

This subroutine handles the menu item 200%, which magnifies an image by 200%. Let's say the same menu contains the options 100%, 75%, 50%, and 25%, and that the names of these commands are Zoom100, Zoom75, and so on. The common handler for their Click event will have the following declaration:

```
Private Sub Zoom200_Click(ByVal sender As System.Object, _
               ByVal e As System.EventArgs) Handles Zoom200.Click, _
               Zoom100.Click, Zoom75.Click, Zoom50.Click, Zoom25.Click
End Sub
```

The common event handler wouldn't do you any good, unless you could figure out which item was clicked from within the handler's code. This information is in the event's *sender* argument. Convert this argument to the MenuItem type, then look up all the properties of the MenuItem object that received the event. The following statement will print the name of the menu item that was clicked (if it appears in a common event handler):

```
Console.WriteLine(CType(sender, MenuItem).Text)
```

When you program multiple menu items with a single event handler, set up a `Select Case` statement based on the caption of the selected menu item, like the following:

```
Select Case sender.Text
   Case "Zoom In"
      { statements to process Zoom In command }
   Case "Zoom Out"
      { statements to process Zoom Out command }
   Case "Fit"
      { statements to process Fit command }
End Select
```

It's also common to manipulate the MenuItem's properties from within its Click event handler. These properties are the same properties you set at design time, through the Menu Editor window. Menu commands don't have methods you can call. Most menu object properties are toggles. To change the Checked property of the FontBold command, for instance, use the following statement:

```
FontBold.Checked = Not FontBold.Checked
```

If the command is checked, the check mark will be removed. If the command is unchecked, the check mark will be inserted in front of its name. You can also change the command's caption at runtime, although this practice isn't common. The Text property is manipulated only when you create dynamic menus, as you will see in the section "Adding and Removing Commands at Runtime." You can change the caption of simple commands such as Show Tools and Hide Tools. These two captions are mutually exclusive, and it makes sense to implement them with a single command. The code behind this MenuItem examines the caption of the command, performs the necessary operations, and then changes the caption to reflect the new state of the application:

```
If ShowMenu.Text = "Show Tools" Then
   { code to show the toolbar }
   ShowMenu.Text = "Hide Tools"
Else
   { code to hide the toolbar }
   ShowMenu.Text = "Show Tools"
End If
```

Using Access and Shortcut Keys

Menus are a convenient way of displaying a large number of choices to the user. They allow you to organize commands in groups, according to their function, and are available at all times. Opening menus and selecting commands with the mouse, however, can be an inconvenience. When using a word processor, for example, you don't want to have to take your hands off the keyboard and reach for the mouse. To simplify menu access, Visual Basic supports access keys and shortcut keys.

Access Keys

Access keys allow the user to open a menu by pressing the Alt key and a letter key. To open the Edit menu in all Windows applications, for example, you can press Alt+E. E is the Edit menu's *access key*.

Once the menu is open, the user can select a command with the arrow keys or by pressing another key, which is the command's access key. Once a menu is open, the Alt key isn't needed. For example, with the Edit menu open, you can press P to invoke the Paste command or C to copy the selected text.

Access keys are designated by the designer of the application, and they are marked with an underline character. The underline under the character E in the Edit menu denotes that E is the menu's access key and that the keystroke Alt+E opens the Edit command. To assign an access key, insert the ampersand symbol (&) in front of the character you want to use as access key in the MenuItem's Text property.

NOTE *If you don't designate access keys, Visual Basic will use the first character in each top-level menu as its access key. The user won't see the underline character under the first character, but will be able to open the menu by pressing the first character of its caption while holding down the Alt key. If two or more menu captions begin with the same letter, the first (left-most and top-most) menu will open.*

Because the & symbol has a special meaning in menu design, you can't use it as is. To actually display the & symbol in a caption, prefix it with another & symbol. For example, the caption &Drag produces a command with the caption Drag (the first character is underlined because it's the access key). The caption Drag && Drop will create another command whose caption will be Drag & Drop. Finally, the string &Drag && Drop will create another command with the caption Drag & Drop.

Shortcut Keys

Shortcut keys are similar to access keys, but instead of opening a menu, they run a command when pressed. Assign shortcut keys to frequently used menu commands, so that users can reach them with a single keystroke. Shortcut keys are combinations of the Ctrl key and a function or character key. For example, the usual *access* key for the Close command (once the File menu is opened with Alt+F) is C; but the usual *shortcut* key for the Close command is Ctrl+W.

To assign a shortcut key to a menu command, drop down the Shortcut list in the MenuItem's Properties window and select a keystroke. You don't have to insert any special characters in the command's caption, nor do you have to enter the keystroke next to the caption. It will be displayed next to the command automatically. To view the possible keystrokes you can use as shortcuts, select a MenuItem in the Form Designer and expand the drop-down list of the Shortcut property in the Properties window.

TIP *When assigning access and shortcut keys, take into consideration well-established Windows standards. Users expect Alt+F to open the File menu, so don't use Alt+F for the Format menu. Likewise, pressing Ctrl+C universally performs the Copy command; don't use Ctrl+C as a shortcut for the Cut command.*

Manipulating Menus at Runtime

Dynamic menus change at runtime to display more or fewer commands, depending on the current status of the program. This section explores two techniques for implementing dynamic menus:

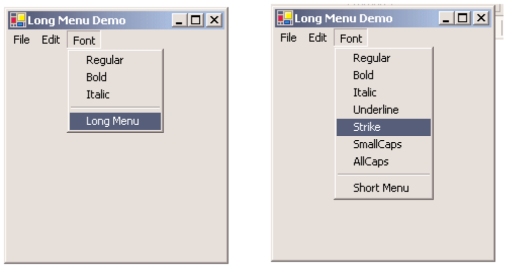
- Creating short and long versions of the same menu
- Adding and removing menu commands at runtime

Once the menu is in place and you have named all the items—you can use the default names, but this makes the code harder to read—you can program them by setting their properties from within your code. Each item in the menu is represented by a MenuItem object, which you program as usual.

CREATING SHORT AND LONG MENUS

A common technique in menu design is to create long and short versions of a menu. If a menu contains many commands, and most of the time only a few of them are needed, you can create one menu with all the commands and another with the most common ones. The first menu is the long one, and the second is the short one. The last command in the long menu should be Short Menu, and when selected, it should display the short version. The last command in the short menu should be Long Menu, and it should display the long version. Figure 5.21 shows a long and a short version of the same menu (from the LongMenu project, which you will find on the CD). The short version omits infrequently used commands and is easier to handle.

FIGURE 5.21



The two versions of the Font menu of the LongMenu application

To implement the LongMenu command, start a new project and create a menu that has the structure shown in Table 5.7. Listing 5.9 is the code that shows/hides the long menu in the MenuSize command's Click event.

TABLE 5.7: LONGMENU COMMAND STRUCTURE

COMMAND NAME	CAPTION
FontMenu	Font
mFontBold	Bold
mFontItalic	Italic

Continued on next page

Table 5.7: LongMenu Command Structure *(continued)*

Command Name	Caption
mFontRegular	Regular
mFontUnderline	Underline
mFontStrike	Strike
mFontSmallCaps	SmallCaps
mFontAllCaps	AllCaps
Separator	- (hyphen)
MenuSize	Short Menu

Listing 5.9: The MenuSize Menu Item's Click Event

```
Protected Sub menuSize_Click(ByVal sender As Object, _
                 ByVal e As System.EventArgs)
   If MenuSize.text = "Short Menu" Then
      MenuSize.text = "Long Menu"
   Else
      MenuSize.text = "Short Menu"
   End If
   mFontUnderline.Visible = Not mFontUnderline.Visible
   mFontStrike.Visible = Not mFontStrike.Visible
   mFontSmallCaps.Visible = Not mFontSmallCaps.Visible
   mFontAllCaps.Visible = Not mFontAllCaps.Visible
End Sub
```

The subroutine in Listing 5.9 doesn't do much. It simply toggles the Visible property of certain menu commands and changes the command's caption to Short Menu or Long Menu, depending on the menu's current status. Notice that because the Visible property is a True/False value, we don't care about its current status; we simply toggle the current status with the Not operator.

Adding and Removing Commands at Runtime

We'll conclude our discussion of menu design with a technique for building dynamic menus, which grow and shrink at runtime. Many applications maintain a list of the most recently opened files in their File menu. When you first start the application, this list is empty, and as you open and close files, it starts to grow.

The RunTimeMenu project demonstrates how to add items to and remove items from a menu at runtime. The main menu of the application's form contains the Run Time Menu submenu, which is initially empty.

The two buttons on the form add commands to and remove commands from the Run Time Menu. Each new command is appended at the end of the menu, and the commands are removed from the bottom of the menu first (the most recently added commands). To change this order, and display the most recent command at the beginning of the menu, use a large initial index value (like 99) and increase it with every new command you add to the menu. Listing 5.10 shows the code behind the two buttons that add and remove menu items.

LISTING 5.10: ADDING AND REMOVING MENUITEMS AT RUNTIME

```
Protected Sub bttnRemoveOption_Click(ByVal sender As Object, _
                 ByVal e As System.EventArgs)
   If RunTimeMenu.MenuItems.Count > 0 Then
      RunTimeMenu.MenuItems.Remove(RunTimeMenu.MenuItems.count - 1)
   End If
End Sub
Protected Sub bttnAddOption_Click(ByVal sender As Object, _
                 ByVal e As System.EventArgs)
   RunTimeMenu.MenuItems.Add("Run Time Option " & _
                 RunTimeMenu.MenuItems.Count.toString, _
                 New EventHandler(AddressOf Me.OptionClick))
End Sub
```

The Remove button's code uses the Remove method to remove the last item in the menu by its index, after making sure the menu contains at least one item. The Add button adds a new item, sets its caption to "Run Time Option *n*", where *n* is the item's order in the menu. In addition, it assigns an event handler to the new item's Click event. This event handler is the same for all the items added at runtime; it's the OptionClick() subroutine.

Adding menu items with the simpler forms of the Add method is trivial. The new menu items, however, would be quite useless unless there was a way to program them as well. The code uses the following form of the Add method, which accepts two arguments: the caption of the item and an event handler:

```
Menu.MenuItems.Add(caption, event_handler)
```

The event handler is the address of a subroutine, which will be invoked when the corresponding menu item is clicked, and it's specified as a New EventHandler object. The AddressOf operator passes the address of the OptionClick() subroutine to the new menu item, so that it knows which subroutine to execute when it's clicked.

As you can understand, all the runtime options invoke the same event handler—it would be quite cumbersome to come up with a separate event handlers for different items. In the single event handler, you can examine the name of the MenuItem object that invoked the event handler and act accordingly. The OptionClick() subroutine used in this example (Listing 5.11) displays the name of the menu item that invoked it. It doesn't do anything, but it shows you how to figure out the item of the Run Time Menu that was clicked:

LISTING 5.11: PROGRAMMING DYNAMIC MENU ITEMS

```
Private Sub OptionClick(ByVal sender As Object, ByVal e As EventArgs)
   Dim itemClicked As New MenuItem()
   itemClicked = CType(sender, MenuItem)
   Console.WriteLine("You have selected the item " & itemClicked.Text)
End Sub
```

CREATING CONTEXT MENUS

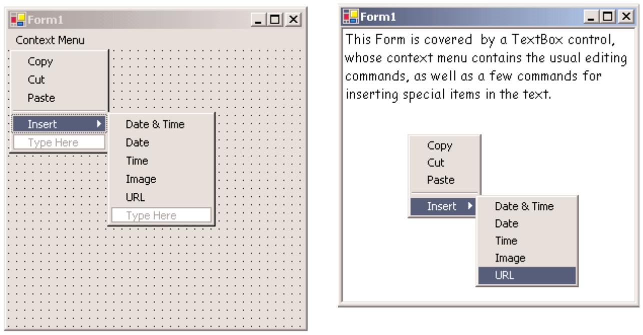
Nearly every Windows application provides a *context menu* that the user can invoke by right-clicking a form or a control. (It's sometimes called a *shortcut menu* or *pop-up menu*.) This is a regular menu, but it's not anchored on the form. It can be displayed anywhere on the form or on specific controls. Different controls can have different context menus, depending on the operations you can perform on them at the time.

To create a context menu, place a ContextMenu control on your form. The new context menu will appear on the form just like a regular menu, but it won't be displayed there at runtime. You can create as many context menus as you need by placing multiple instances of the ContextMenu control on your form and adding the appropriate commands to each one. To associate a context menu with a control on your form, set the *control's* ContextMenu property to the name of the corresponding context menu.

Designing a context menu is identical to designing a MainMenu. The only difference is that the first command in the menu is actually the context menu's name, and it's not displayed along with the menu. Figure 5.22 shows a context menu at design time and how the same menu is displayed at runtime. Context Menu is the menu's name, not a menu item.

FIGURE 5.22

A context menu, (left) at design time and (right) at runtime



This Form is covered by a TextBox control, whose context menu contains the usual editing commands, as well as a few commands for inserting special items in the text.

You can create as many context menus as you wish on a form. Each control has a ContextMenu property, which you can set to any of the existing ContextMenu controls. Select the control for which you want to specify a context menu and, in the Properties window, locate the ContextMenu property. Expand the drop-down list and select the name of the desired context menu.

To edit one of the context menus on a form, select the appropriate ContextMenu control at the bottom of the Designer. The corresponding context menu will appear on the form's menu bar, as if it were a regular form menu. This is temporary, however, and the only menu that will appears on the form's menu bar at runtime is the one that corresponds to the MainMenu control (and there can be only one of those on each form).

You can also merge two menus to create a new one that combines their items. This technique is used with MDI forms, where we want to add the commands of the child form to the parent form. For more information on the Merge method, see Chapter 19.

Iterating a Menu's Items

The last menu-related topic in this chapter demonstrates how to iterate through all the items of a menu structure, including their submenus at any depth. The main menu of an application can be accessed by the expression `Me.Menu`. This is a reference to the top-level commands of the menu, which appear in the form's menu bar. Each command, in turn, is represented by a MenuItem object. All the MenuItems under a menu command form a MenuItems collection, which you can scan and retrieve the individual commands.

The first command in a menu is accessed with the expression `Me.Menu.MenuItems(0)`; this is the File command in a typical application. The expression `Me.Menu.MenuItems(1)` is the second command on the same level as the File command (typically, the Edit menu).

To access the items *under* the first menu, use the MenuItems collection of the top command. The first command in the File menu can be accessed by the expression

```
Me.Menu.MenuItems(0).MenuItems(0)
```

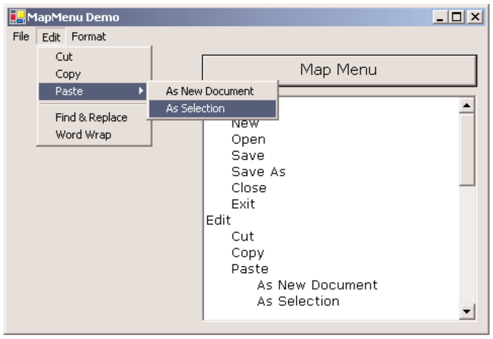
The same items can be accessed by name as well, and this is how you should manipulate the menu items from within your code. In unusual situations, or if you're using dynamic menus to which you add and subtract commands at runtime, you'll have to access the menu items through the MenuItems collection.

VB.NET AT WORK: THE MAPMENU PROJECT

The MapMenu project demonstrates how to access the items of a menu from within your application's code. The project's main form, shown in Figure 5.23, contains a menu, a TextBox control, and a Button that prints the menu's structure on the TextBox. You can edit the menu before running the program, and the code behind the Button will print the structure of the menu items without any modifications.

FIGURE 5.23

The MapMenu application



```
    New
    Open
    Save
    Save As
    Close
    Exit
Edit
    Cut
    Copy
    Paste
        As New Document
        As Selection
```

The code behind the Map Menu button (Listing 5.12) iterates through the items of a MainMenu object and prints all the commands in the Output window. It scans all the items of the menu's MenuItems collection and prints their captions. After printing each command's caption, it calls the PrintSubMenu() subroutine, passing the current MenuItem as argument. The PrintSubMenu() subroutine iterates through the items of the collection passed as argument and prints their captions.

Listing 5.12: Printing the Top-Level Commands of a Menu

```
Protected Sub MapMenu_Click(ByVal sender As Object, ByVal e As System.EventArgs)
   Dim itm As MenuItem
   For Each itm In Me.Menu.MenuItems
      Console.WriteLine(itm.Text)
      PrintSubMenu(itm)
   Next
End Sub
```

The PrintSubMenu() subroutine, shown in Listing 5.13, goes through the MenuItems collection of the MenuItem object passed to it as argument and prints the captions of the submenu it represents. At each iteration, it examines the value of the property `itm.MenuItems.Count`. This is the number of commands under the current menu items. If it's a positive value, the current item leads to a submenu. To print the submenu's items, it calls itself, passing the *itm* object as argument. This simple technique scans all the submenus, at any depth. The PrintSubMenu() subroutine is a recursive routine, because it calls itself.

Listing 5.13: Printing Submenu Items

```
Sub PrintSubMenu(ByVal MItem As MenuItem)
   Dim itm As New MenuItem()
   For Each itm In MItem.MenuItems
      Console.WriteLine(itm.Text)
      If itm.MenuItems.Count > 0 Then PrintSubMenu(itm)
   Next
End Sub
```

Tip *There's a tutorial on coding recursive routines in Chapter 18 of this book, and you will find more examples of recursive routines in the course of the book. If you're totally unfamiliar with recursive routines, you can come back and examine the code more carefully after reading this chapter.*

Open the MapMenu application, edit the menu on its form, run the project, and click the Map Menu Structure button. The few lines of the PrintSubMenu() subroutine will iterate through all the items in the form's menu and submenus, at any depth.

Building Dynamic Forms at Runtime

There are situations when you won't know in advance how many instances of a given control may be required on a form. This isn't very common, but if you're writing a data-entry application and you want to work with many tables of a database, you'll have to be especially creative. Since every table consists of different fields, it will be difficult to build a single form to accommodate all the possible tables a user may throw at your application.

Another good reason for adding or removing controls at runtime is to enable certain features of your application, depending on the current state or the user's privileges. For these situations, it is possible to design *dynamic forms,* which are populated at runtime. The simplest approach is to create more controls than you'll ever need and set their Visible property to False at design time. At runtime, you can display the controls by switching their Visible property to True. As you know already, quick-and-dirty methods are not the most efficient ones. You must still rearrange the controls on the form to make it look nice at all times. The proper method to create dynamic forms at runtime is to add and remove controls with the techniques discussed in this section.

Just as you can create new instances of forms, you can also create new instances of any control and place them on a form. The Form object exposes the Controls collection, which contains all the controls on the form. This collection is created automatically as you place controls on the form at design time, and you can access the members of this collection from within your code. It is also possible to add new members to the collection, or remove existing members, with the Add and Remove statements accordingly.

VB6 ➠ VB.NET

VB.NET doesn't support arrays of controls, which used to be the simplest method of adding new controls on a form at runtime. With VB.NET, you must create a new instance of a control, set its properties, and then place it on the form by adding it to the form's Controls collection.

The Form.Controls Collection

To understand how to create controls at runtime and place them on a form, you must first learn about the Controls collection. All the controls on a form are members of the Controls property, which is a collection. The Controls collection exposes members for accessing and manipulating the controls at runtime, and these members are:

Add method Adds a new element to the Controls collection. In effect, it adds a new control on the current form. The Add method accepts a control as argument and adds it to the collection. Its syntax is:

```
Controls.Add(controlObj)
```

where *controlObj* is an instance of a control. To place a new Button control on the form, declare a variable of the Button type, set its properties, and then add it to the Controls collection:

```
Dim bttn As New System.WinForms.Button
```

```
bttn.Text = "New Button"
bttn.Left = 100
bttn.Top = 60
bttn.Width = 80
Me.Controls.Add(bttn)
```

Remove method Removes an element from the Controls collection. It accepts as argument either the index of the control to be removed, or a reference to the control to be removed (a variable of the Control type that represents one of the controls on the form). The syntax of these two forms is:

```
Me.Controls.Remove(index)
Me.Controls.Remove(controlObj)
```

Count property Returns the number of elements in the Controls collection. The number of controls on the current form is given by the expression `Me.Controls.Count`. Notice that if there are container controls, the controls in the containers are not included in the count. For example, if your form contains a Panel control, the controls on the panel won't be included in the value returned by the Count property.

All method Returns all the controls on a form (or in a container control) as an array of the `System.WinForms.Control` type. You can iterate through the elements of this array with the usual methods exposed by the Array class.

Clear method Removes all the elements of the Controls array.

The Controls collection is also a property of any control that can host other controls. Most of the controls that come with VB.NET can host other controls. The Panel control, for example, is a container for other controls. As you recall from our discussion of the Anchor and Dock properties, it's customary to place controls on a panel and handle them collectively, as a section of the form. They are moved along with the panel at design time, and they're rearranged as a group at runtime. The panel belongs to the form's Controls collection. The element that corresponds to the Panel control provides its own Controls collection, which lets you access the controls on the panel.

If a panel is the third element of the Controls collection, you can access it with the expression `Me.Controls(2)`. To access the controls *on* this panel, use the following Controls collection:

```
Me.Controls(2).Controls
```

VB.NET at Work: The ShowControls Project

The ShowControls project (Figure 5.24) demonstrates the basic methods of the Controls array. Open the project and add any number of controls on its main form. You can place a panel to act as a container for other controls as well. Just don't remove the button at the top of the form (the Scan Controls On This Form button), which contains the code to list all the controls.

The code behind the Scan Controls On This Form button enumerates the elements of the form's Controls collection. The code doesn't take into consideration containers within containers. This would require a *recursive* routine, which would scan for controls at any depth. You will read a lot about recursive routines in this book and you will find a tutorial on the topic in Chapter 18. After you're familiar with recursion (if you aren't already), you can revisit this project and adjust its code accordingly. The code that iterates through the form's Controls collection and prints the names of the controls in the Output window is shown in Listing 5.14.

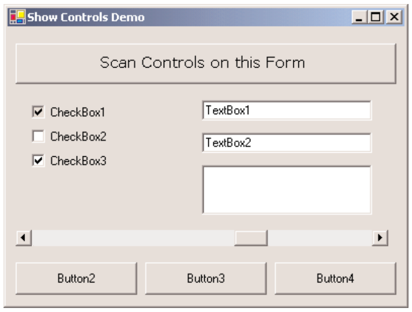


FIGURE 5.24 Accessing the controls on a form at runtime

LISTING 5.14: ITERATING THE CONTROLS COLLECTION

```
Private Sub Button1_Click(ByVal sender As System.Object, _
               ByVal e As System.EventArgs) Handles Button1.Click
   Dim i As Integer
   For i = 0 To Me.Controls.Count - 1
     Console.WriteLine(Me.Controls(i).ToString)
     If Me.Controls(i).GetType Is GetType(system.Windows.Forms.Panel) Then
       Dim j As Integer
       For j = 0 To Me.Controls(i).Controls.Count - 1
         Console.WriteLine(Me.Controls(i).Controls(j).ToString)
       Next
     End If
   Next
End Sub
```

The form shown in Figure 5.24 produced the following output:

```
System.Windows.Forms.HScrollBar, Minimum: 0, Maximum: 100, Value: 60
System.Windows.Forms.CheckedListBox
System.Windows.Forms.TextBox, Text: TextBox2
System.Windows.Forms.CheckBox, CheckState: 1
System.Windows.Forms.CheckBox, CheckState: 0
System.Windows.Forms.CheckBox, CheckState: 1
System.Windows.Forms.TextBox, Text: TextBox1
System.Windows.Forms.Button, Text: Button4
System.Windows.Forms.Button, Text: Button3
System.Windows.Forms.Button, Text: Button2
```

Each member of the Controls collection exposes the GetType method, which returns the control's type, so that you can know what control is stored in each collection element. To compare the control's type returned by the GetType method, use the GetType() function passing as argument a control type. The following statement examines whether the control in the first element of the Controls collection is a TextBox:

```
If Me.Controls(0).GetType Is GetType(system.WinForms.TextBox) Then
   MsgBox("It's a TextBox control")
End If
```

Notice the use of the Is operator in the preceding statement. The equals operator will cause an exception, because objects can be compared only with the Is operator. Do not use string comparisons to find out the control's type. A statement like the following won't work:

```
If Me.Controls(i).GetType = "TextBox" Then ...   ' WRONG
```

The elements of the Controls collection are of the Control type, and they expose the properties of the control they represent. Their Top and Left properties read (or set) the position of the corresponding control on the form. The following expressions move the first control on the form to the specified location:

```
Me.Controls(0).Top = 10
Me.Controls(0).Left = 40
```

To access other properties of the control represented by an element of the Controls collection, you must first cast it to the appropriate type. If the first control of the collection is a TextBox control, use the CType() function to cast it to a TextBox variable, and then request its Text property:

```
If Me.Controls(i).GetType Is GetType(system.WinForms.TextBox) Then
   Console.WriteLine(CType(Me.Controls(0), TextBox).Text)
End If
```

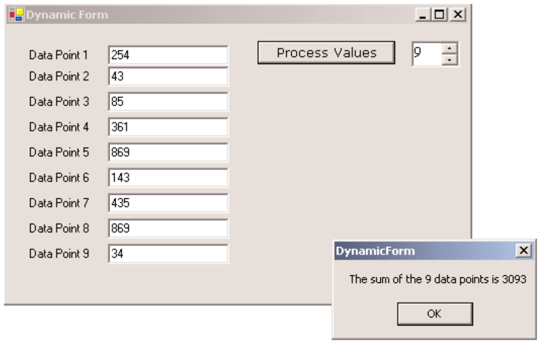
The If statement is necessary, unless you can be sure that the first control is a TextBox control. If you omit the If statement and attempt to convert it to a TextBox, a runtime exception will be thrown if the object `Me.Controls(0)` isn't a TextBox control.

VB.NET at Work: The DynamicForm Project

To demonstrate how to handle controls at runtime from within your code, I've included the DynamicForm project (Figure 5.25), a simple data-entry window for a small number of data points. The user can specify at runtime the number of data points they wish to enter, and the number of TextBoxes on the control changes.

FIGURE 5.25

The DynamicForm project



Data Point 1	254
Data Point 2	43
Data Point 3	85
Data Point 4	361
Data Point 5	869
Data Point 6	143
Data Point 7	435
Data Point 8	869
Data Point 9	34

The sum of the 9 data points is 3093

The control you see at the top of the form is the NumericUpDown control. All you really need to know about this control is that it fires the ValueChanged event every time the user clicks one of the two arrows or types another value in its edit area. This event handler's code adds or removes controls on the form, so that the number of TextBoxes (as well as the number of the corresponding labels) matches the value on the control. Listing 5.15 shows the handler for the ValueChanged event of the NumericUpDown1 control. The ValueChanged event is fired when the user clicks one of the two arrows on the control or types a new value in the control's edit area.

LISTING 5.15: ADDING AND REMOVING CONTROLS AT RUNTIME

```
Private Sub NumericUpDown1_ValueChanged(ByVal sender As System.Object, _
               ByVal e As System.EventArgs) Handles NumericUpDown1.ValueChanged
   Dim TB As New TextBox()
   Dim LBL As New Label()
   Dim i, TBoxes As Integer
' Count all TextBox controls on the form
   For i = 0 To Me.Controls.Count - 1
      If Me.Controls(i).GetType Is GetType(System.Windows.Forms.TextBox) Then
         TBoxes = TBoxes + 1
      End If
   Next
' Add new controls if number of controls on the form is less
' than the number specified with the NumericUpDown control
   If TBoxes < NumericUpDown1.Value Then
      TB.Left = 100
      TB.Width = 120
      TB.Text = ""
      For i = TBoxes To NumericUpDown1.Value - 1
         TB = New TextBox()
         LBL = New Label()
         If NumericUpDown1.Value = 1 Then
            TB.Top = 20
         Else
            TB.Top = Me.Controls(Me.Controls.Count - 2).Top + 25
         End If
         Me.Controls.Add(TB)
         LBL.Left = 20
         LBL.Width = 80
         LBL.Text = "Data Point " & i
         LBL.Top = TB.Top + 3
         TB.Left = 100
         TB.Width = 120
         TB.Text = ""
         Me.Controls.Add(LBL)
         AddHandler TB.Enter, _
               New System.EventHandler(AddressOf TBox_Enter)
         AddHandler TB.Leave, _
               New System.EventHandler(AddressOf TBox_Leave)
      Next
   Else
         For i = Me.Controls.Count - 1 To _
                 Me.Controls.Count - 2 * (TBoxes - NumericUpDown1.Value) Step -2
            Me.Controls.Remove(Controls(i))
            Me.Controls.Remove(Controls(i - 1))
         Next
   End If
End Sub
```

First, the code counts the number of TextBoxes on the form, then it figures out whether it should add or remove elements from the Controls collection. To remove controls, the code iterates through the last *n* controls on the form and removes them. The number of controls to be removed, *n,* is:

```
2 * (TBoxes - NumericUpDown1.Value)
```

where *TBoxes* is the total number of controls on the form minus the value specified in the Numeric-UpDown control.

If the value entered in the NumericUpDown control is less than the number of TextBox controls on the form, the code removes the excess controls from within a loop. At each step, it removes two controls, one of them being a TextBox and the other being a Label control with the matching caption (that's why the loop variable is decreased by two). The code also assumes that the first two controls on the form are the Button and the NumericUpDown controls. If the value entered by the user exceeds the number of TextBox controls on the form, the code adds the necessary pairs of TextBox and Label controls to the form.

To add controls, the code initializes a TextBox (*TB*) and a Label (*LBL*) variable. Then, its sets their locations and the label's caption. The left coordinate of all labels is 20, their width is 80, and their Text property (the label's caption) is the order of the data item. The vertical coordinate is 20 pixels for the first control, and all other controls are three pixels below the control on the previous row. Once a new control has been set up, it's added to the Controls collection with one of the following statements:

```
Me.Controls.Add(TB)    ' adds a TextBox control
Me.Controls.Add(LBL)   ' adds a Label control
```

The code contains a few long lines, but it isn't really complicated. It's based on the assumption that, except for the first few controls on the form, all others are pairs of Label and TextBox controls used for data entry.

To use the values entered by the user on the dynamic form, we must iterate the Controls collection, extract the values in the TextBox controls and use them. Listing 5.16 shows how the Process Values button scans the TextBox controls on the form performs some very basic calculations with them (counting the number of data points and summing their values).

LISTING 5.16: READING THE CONTROLS ON THE FORM

```
Private Sub Button1_Click(ByVal sender As System.Object, _
               ByVal e As System.EventArgs) Handles Button1.Click
   Dim ctrl As Object
```

```
      Dim Sum As Double = 0, points As Integer = 0
      Dim iCtrl As Integer
      For iCtrl = 0 To Me.Controls.Count - 1
         ctrl = Me.Controls(iCtrl)
         If ctrl.GetType Is GetType(system.Windows.Forms.TextBox) Then
            If IsNumeric(CType(ctrl, TextBox).Text) Then
               Sum = Sum + CType(ctrl, TextBox).Text
               points = points + 1
            End If
         End If
      Next
      MsgBox("The sum of the " & points.ToString & " data points is " & _
             Sum.ToString)
   End Sub
```

You can add more statements to calculate the mean and other vital statistics, or process the values in any other way. You can even dump all the values into an array and then use the array notation to manipulate them.

You can also write a `For Each…Next` loop to iterate through the TextBox controls on the form, as shown in Listing 5.17. The Process Values button at the bottom of the form demonstrates this alternate method of iterating through the elements of the `Me.Controls` collection. Because this loop goes through all the elements, we must examine the type of each control in the loop and process only the TextBox controls.

LISTING 15.17: READING THE CONTROLS WITH A FOR EACH…NEXT LOOP

```
   Private Sub bttnProcess2_Click(ByVal sender As System.Object, _
               ByVal e As System.EventArgs) Handles bttnProcess2.Click
      Dim TB As Control
      Dim Sum As Double = 0, points As Integer = 0
      For Each TB In Me.Controls
         If TB.GetType Is GetType(Windows.Forms.TextBox) Then
            If IsNumeric(CType(TB, TextBox).Text) Then
               Sum = Sum + CType(TB, TextBox).Text
               points = points + 1
            End If
         End If
      Next
      MsgBox("The sum of the " & points.ToString & " data points is " & _
             Sum.ToString)
   End Sub
```

Creating Event Handlers at Runtime

You've seen how to add controls on your forms at runtime and how to access the properties of these controls from within your code. In many situations, this is all you need: a way to access the properties of the controls (the text on a TextBox control, or the status of a CheckBox or RadioButton control). What good is a Button control, however, if it can't react to the Click event? The only problem with the controls you add to the Controls collection at runtime is that they don't react to events. It's possible, though, to create event handlers at runtime, and this is what you'll learn in this section. Obviously, this isn't a technique you'll be using every day; you can come back and read this section when the need arises.

To create an event handler at runtime, create a subroutine that accepts two arguments—the usual *sender* and *e* arguments—and enter the code you want to execute when a specific control receives a specific event. Let's say you want to add one or more buttons at runtime on your form and these buttons should react to the Click event. Create the ButtonClick() subroutine and enter the appropriate code in it. The name of the subroutine could be anything; you don't have to make up a name that includes the control's or the event's name.

Once the subroutine is in place, you must connect it to an event of a specific control. The ButtonClick() subroutine, for example, must be connected to the Click event of a Button control. The statement that connects a control's event to a specific event handler, is the AddHandler statement, whose syntax is:

```
AddHandler control.event, New System.EventHandler(AddressOf subName)
```

For example, to connect the ProcessNow() subroutine to the Click event of the Calculate button, use the following statement:

```
AddHandler Calculate.Click, New System.EventHandler(AddressOf ProcessNow)
```

Let's add a little more complexity to the DynamicForm application. We will program the Enter and Leave events of the TextBox controls added at runtime through the `Me.Controls.Add` method. When a TextBox control receives the focus, we'll change its background color to a light yellow, and when it loses the focus we'll restore the background to white, so that the user knows which box has the focus at any time. We'll use the same handlers for all TextBox controls, and the code of the two handlers are shown in Listing 5.18.

Listing 5.18: Event Handlers Added at Runtime

```
Private Sub TBox_Enter(ByVal sender As Object, ByVal e As System.EventArgs)
   CType(sender, TextBox).BackColor = color.LightCoral
End Sub
Private Sub TBox_Leave(ByVal sender As Object, ByVal e As System.EventArgs)
   CType(sender, TextBox).BackColor = color.White
End Sub
```

The event handlers use the *sender* argument to find out which TextBox control received or lost the focus, and they set the appropriate control's background color (property BackColor). We write one handler per event and associate it with any number of controls added dynamically. Technically, the

TBox_Enter() and TBox_Leave() subroutines are not event handlers—at least, not before we associate them with an actual control and a specific event. This is done in the same segment of code that sets the properties of the controls we create dynamically at runtime. After adding the control to the `Me.Controls` collection, call the following statements to connect the new control's Enter and Leave events to the appropriate handlers:

```
AddHandler TB.Enter, New System.EventHandler(AddressOf TBox_Enter)
AddHandler TB.Leave, New System.EventHandler(AddressOf TBox_Leave)
```

Run the DynamicForm application and see how the TextBox controls handle the focus-related events. With a few statements and a couple of subroutines, we were able to create event handlers at runtime, from within our code.

Summary

In this chapter, you learned the most useful and practical techniques for designing forms. The Windows Form Designer that comes with VS.NET is leaps ahead of the equivalent designer of VB6, and it allows you to design truly elaborate interfaces with very little code (in some cases, no code at all). At the very least, you must make sure that the controls on the form will fit nicely when the form is resized at runtime by setting the Anchor and Dock properties accordingly.

Building applications with multiple forms is a bit more involved than it used to be, but not really complicated. In the following chapter, we're going to discuss in detail the basic components of the user interface, which are the controls—the basic building blocks of the application. If you think forms come with a lot of built-in functionality, wait until you find out the functionality built into the controls.