Visual Basic 2008 Programming

Business Applications with a Design Perspective

Jeffrey J. Tsay

Table of Contents

Chapter 11

Menus and Multiple-Form Applications

The previous chapters focused on projects involving only one form. This chapter deals with topics related to large projects that traditionally involve many forms. Many similar activities and functions will be called for in different forms, so there are plenty of opportunities to reduce redundant code. This requires careful analysis and design of the project. Because many program functionalities must be provided, menus are used in place of buttons. The purpose of this chapter is to present techniques to handle all these aspects.
After completing this chapter, you should be able to:

- Create menus,
- Apply proper guidelines in creating menus,
- Add additional forms and modules to a project,
- Designate a startup object for the project,
- Design and implement ways to share data and code among forms,
- Create and code MDI applications, and
- Articulate design principles for large projects.

11.1 The Main Menu

When your application needs to provide many functions from which the user can choose, it might not be practical to use buttons to trigger these functions. Too many buttons can clutter the form, making it look messy and difficult for the user to locate the needed one. In such cases, a menu will be a good alternative. This section introduces the menu.

Creating a Menu

You have seen menus in action in various Windows applications. The VS IDE with which you have been working has a menu bar on which many menus are presented. You can create a similar menu bar and menus in your applications by using the Menu Strip.

Menu Design Considerations

Before you start to create menus for your application, you should carefully analyze your application to identify all capabilities it needs to provide. Each capability and function that the user can choose to perform will typically become an option in one of the menus. You will then decide how these capabilities should be grouped. There can be various ways to group them, but the most important consideration should be how *logical* and intuitive the menus appear to the user. Another consideration is *consistency* with all other Windows applications.

To illustrate, consider an accounts receivable application system for a family medical clinic. The application is used to keep track of patients' visits and the resulting fees. In most cases, the patients are insured. An insurance company pays the major portion of the fees, and the patient is responsible for a fixed co-pay amount. To obtain payments from the insurance company, the clinic needs to file an insurance claim form for each patient visit, which requires various types of information concerning the patient, the insured, the diagnosis, and the treatment. In many cases, the insured person responsible for payment on the account is not the patient, but rather somebody else such as the patient's parent. The application system must maintain records concerning not only the visits, but also all data pertinent to the patients, accounts, insurance companies, as well as the relationships among these entities. In addition, to support the recording of patient visits, the application system needs to maintain code tables for diagnoses as well as treatments (services provided).

The maintenance of records for all the aforementioned activities and entities requires the following:

- Screens (forms) for data-entry operations
- Reports of various activities (patient visits and payments as well as insurance remittance) for review and verification
- Screens to view activities and entities on files
- Claim forms for fee reimbursement
- Other utility functions for file maintenance operations and system customizations

How should all these functions be grouped for menus? Programmers tend to group functions by their program types; for example, report-generating procedures are grouped together under the menu, Reports; however, this approach of grouping may not be most intuitive to the user, who tends to think along the line of business activities. For example, the user most likely would rather see all the software capabilities related to insurance grouped under the menu, Insurance. This menu can include procedures and forms that perform maintenance of insurance company information, list insurance companies, print claim forms, as well as generate reports for all outstanding claims.

When designing menus for an application, keep in mind that you develop the application for the user to use. Only those applications that fit the user's needs and meet their application requirements are used and considered successful.

Along the line of designing a menu system that appears intuitive to the user, you should also consider its consistency with other applications. For example, most users are likely to be well acquainted with the word processing software as well as other Windows applications. These applications have the familiar File, Edit, and Help menus; therefore, as long as your application needs to provide similar functions, they should be incorporated in a similar manner. This will alleviate the user's needs to learn to navigate around your software system.

Using the Menu Strip

After you have designed the menus for your application, you can use the *Menu Strip* to create the menus. To use the Menu Strip, either double-click the MenuStrip icon in the Toolbox, or click the icon and draw it on the form. Either way, the menu named MenuStrip1 will appear in the area below the form, and a rectangle with the text "Type Here" will appear on the upper left corner of the form (Figure 11-1). Whatever you type in the rectangle will become the text for the first menu item. Additional rectangles will appear for you to add items in the menu.

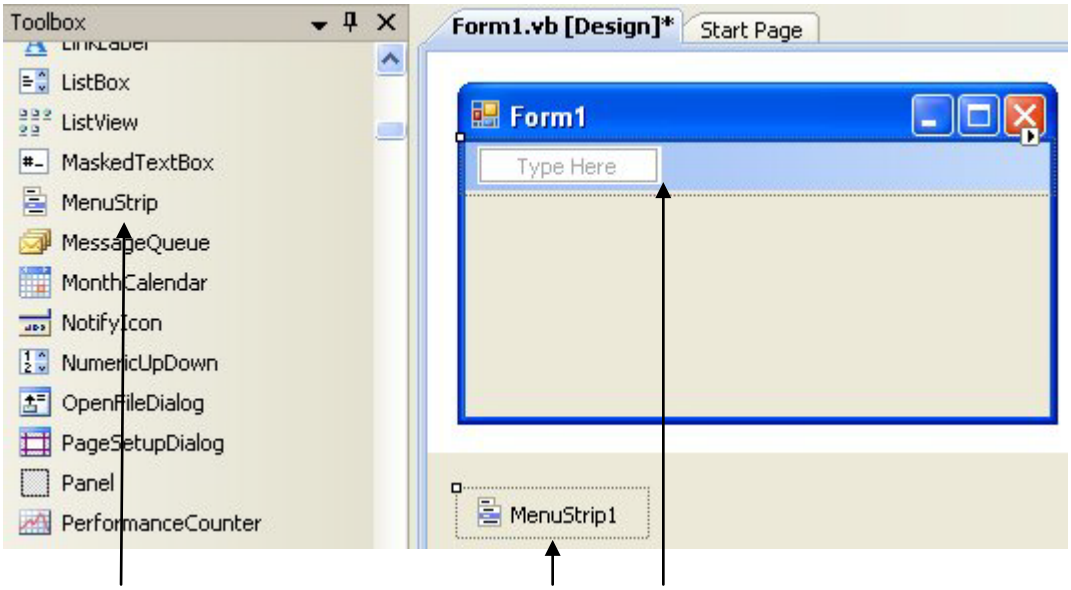
Creating a File Menu Step by Step

Suppose you need to create a File menu that consists of four options: New, Open, Close, and Exit. Follow these steps:

- Double-click the MenuStrip icon in the Toolbox.
- Enter "&File" in the rectangle, and press the Enter key when complete. You should see the letter F in File is underlined. You should see two additional rectangles showing Type Here, as shown in Figure 11-2. The rectangle on the right of File allows you to create another item on the menu bar, and the rectangle below File allows you to create a menu option under File.
- Enter "&New" in the rectangle below File. Again, you should see two additional rectangles. The rectangle on the right allows you to create additional submenus, while the rectangle below allows you to create additional submenu items. Press the Enter key when complete.

Do the same to add Open, Close and Exit as the additional submenu items. (Note that x, not E is underlined in Exit. The & symbol should be placed before x.)

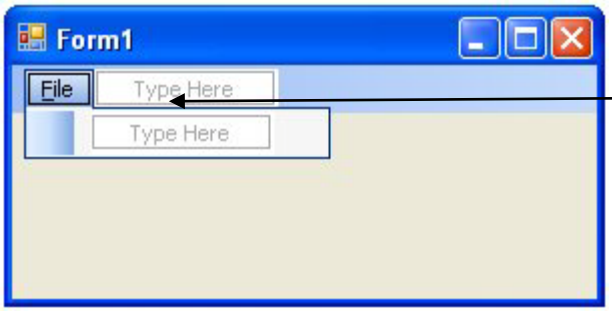
Figure 11-1
The Menu Strip icon and appearance



Menu strip in the toolbox

Menu strip when dragged onto a form will appear in the area below the form and show a rectangle in the form.

Figure 11-2
Editing the menu



After you enter &File, you should see File (with F underlined) and two additional rectangles. The one on the right allows you to create another menu on the menu bar and the one below allows you to create submenu items (menu options) under File.

Adding a Separator Line

When you click the form, you should be able to see File appear in the menu bar of your form. If you click the menu, you should be able to see all the menu options (New, Open, and so on) in the drop-down menu list. Examine the menu list carefully. You may want to add a line (separator) between Exit and all other items above it because this item and others appear to logically belong to different categories. Right-click on Exit and then select the Insert Separator option to add a separator line between Exit and Close.

Naming Menu Items

The menu items you have created so far are automatically given the names that begin with the texts you have entered followed by the suffix of ToolStripMenuItem; for example, if you enter Save As for menu item, it will be named SaveAsToolStripMenuItem. To change these names, click on each item and then change the name in the Properties window. In accordance with the naming conventions, menu items are given the prefix mnu; therefore, the menu for File should be given the name mnuFile. The submenu name should include the name of its upper level menu. For example, the name for the menu item New should be mnuFileNew; for Open, mnuFileOpen; etc.

Adding New Menu Items in the Menu Bar

When you are ready to add another menu to the menu bar (such as create another top-level menu), click the File menu on the form, the one you have just created (not the File menu in the IDE). You will see two Type Here rectangles. Enter the menu text (say, Edit) in the rectangle on the right of File to create a new menu.

To add new menu items at the end of a menu, enter the item in the rectangle below a particular menu item. You can also perform Insert New, Delete, Cut, and Paste by right-clicking on the menu list and then selecting the proper option.

Context (Pop-Up) Menus

In some cases, you may find that certain menus do not belong to the menu bar but should be available only in specific situations. Because these menus appear only under certain contexts, they are recognized as *context menus*. Context menus appear only when the user clicks the right mouse button on a particular control. To create a context menu, draw a *Context Menu strip* on the form, or simply double-click the Context Menu Strip icon in the Toolbox. Editing items in the context menu is similar to that for editing a main menu. Here are a few additional points:

- There is no top level menu bar for the context menu.

- You can edit the menu names and items by right-clicking a menu item and then selecting the desired option in the same manner as the main menu.
- You can create as many context menus as you need. For each additional context menu, add a Context Menu Strip to the form. To edit a particular context menu, click the Context Menu Strip to make its menu items appear on the form.
- The name of the Context Menu Strip (the one that appears in the area below the form) should be meaningful. You can change the name using the control's Properties window.

As mentioned previously, a context menu is triggered when the user right-clicks a control, which must be associated with this context menu. To associate a control with a context menu, set the *Context Menu Strip property* of the control to the context menu. For example, suppose you have created a context menu with the name cmnEdit (which has Copy, Cut, and Paste options) and you want to trigger this menu when the user right-clicks on a text box named txtDocument. You should set the Context Menu Strip property of txtDocument to cmnEdit. A context menu can be associated with many controls, but a control can have only one context menu.

Assigning Access and Short Keys

As you probably have guessed, when you enter an & symbol before a letter in a menu text, the letter becomes the access key for the menu item. For example, &File, which appears in the form as File, has an access key F. The user can press Alt+F to access the File menu. (*Note:* when the program is running, the user may need to press the Alt key for the F letter to show the underline.) In addition to the access key, you can also assign a shortcut key to a menu item by selecting the (shortcut) key in the menu item's Shortcut property in the menu item's Properties window. For example, if you select Ctrl+C as the shortcut for the Close option, the user will be able to activate the Close option by pressing Ctrl+C. Note that access keys behave differently from shortcut keys in several respects. A menu item is accessible through its access key only when the item is visible; the user presses the Alt key plus the underlined letter to activate the menu item. In contrast, a menu item is accessible through the shortcut key anytime the designated shortcut key is pressed regardless of whether the menu item is visible. Also, allowable shortcut keys are formed by a combination of Ctrl, Alt, and Shift keys plus a letter, while access keys are formed by an Alt key plus a character, not limited to a letter (e.g., @).

Invoking an Action

The menu has a Click event that occurs when one of the following three situations occurs:

- The menu item is clicked.
- The menu's access key (Alt+<the underlined letter in the Text>) is pressed.
- The menu's shortcut key is pressed.

Coding the event is the same as coding for the Click event procedure of a button. For example, the code for the menu mnuFileExit with the text Exit will be as follows:

```
Private Sub mnuFileExit_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
  Handles mnuFileExit.Click
    Me.Close()
End Sub
```

Levels of Menus

You can create as many levels of menus as you practically want or need; however, keep in mind that as the levels of menu increase, they could become confusing to the user. When you are designing the menu structure, avoid a design that goes beyond three levels.

Occasionally, a submenu is used for the choice of settings of the operating environment such as the background color or font for an object. In such cases, a custom dialog box (a form created to ascertain specifications/instructions/information from the user) with radio buttons and check boxes can be used as a substitute, and is usually a better alternative.

11.2 Multiple-Form Applications

In most practical applications a project can involve many forms, each of which performs a special task such as displaying a report or working as a data-entry screen. This section considers issues related to multiple-form applications.

Adding a Form to a Project

When you start a new VB project, a new form is automatically provided. To add an additional new form to the project:

- Click the Project menu.
- Select the Add Windows Form option. The Add New Item dialog box will appear.
- Select the Windows Form in the Templates pane.
- Enter the name for the new form at the bottom of the dialog box.
- Click the Open button.

Tip

Another way to invoke the Add New Item dialog box is to click the project name in the Solution Explorer, select the Add option in the context menu, and click the Add Windows Form option in the submenu.

Switching Among Forms and Code Windows

The form that you add to the project most recently is the one that will appear on top in the center of the IDE. The code window (module) will also be associated with this form.

To switch to another form, double-click the form name in the Solution Explorer window or click the form name and then click the Form icon. To switch to another code window directly, click its form name and then click the View Code icon in the Solution Explorer window. All the form and code windows currently open also appear as horizontal tabs right between the tool bar and the main window area. A quick way to switch to one of these windows is to click the tab of the window of your choice. You can also use the Window menu to select the desired form. The list of forms in the project is shown at the bottom of the Window menu.

Tip

The list of forms and their associated code windows that are currently open is also shown as tabs across top of the Designer's design area where the code appears. The quickest and simplest way to switch among these forms and code windows is to click the tab of the desired object on this list.

Starting Up and Calling a Form

When your project starts, only one form will automatically open. The default startup form is the first form that you create in the project. To start up with another form in the project, you will need to specify it in the Project Properties dialog box. To start the dialog box, right-click the Project name in the Solution Explorer window and then click Open. The dialog box appears as shown in Figure 11-3. Select the Application tab on the left side of the dialog box. You can then click the drop-down arrow on the Startup form box, and select the target startup form.

Displaying and Closing Another Form

At run time, only the startup form will initially appear. To invoke another form, use its Show method. For example, supposed you have added a form named frmAccount. To invoke it, you code:

```
frmAccount.Show()
```

To close the same form from another form, you code:

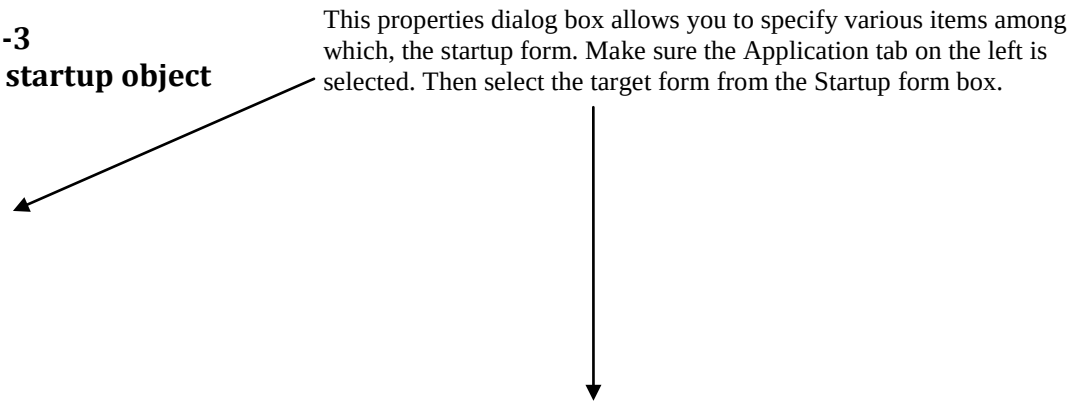
```
frmAccount.Close()
```

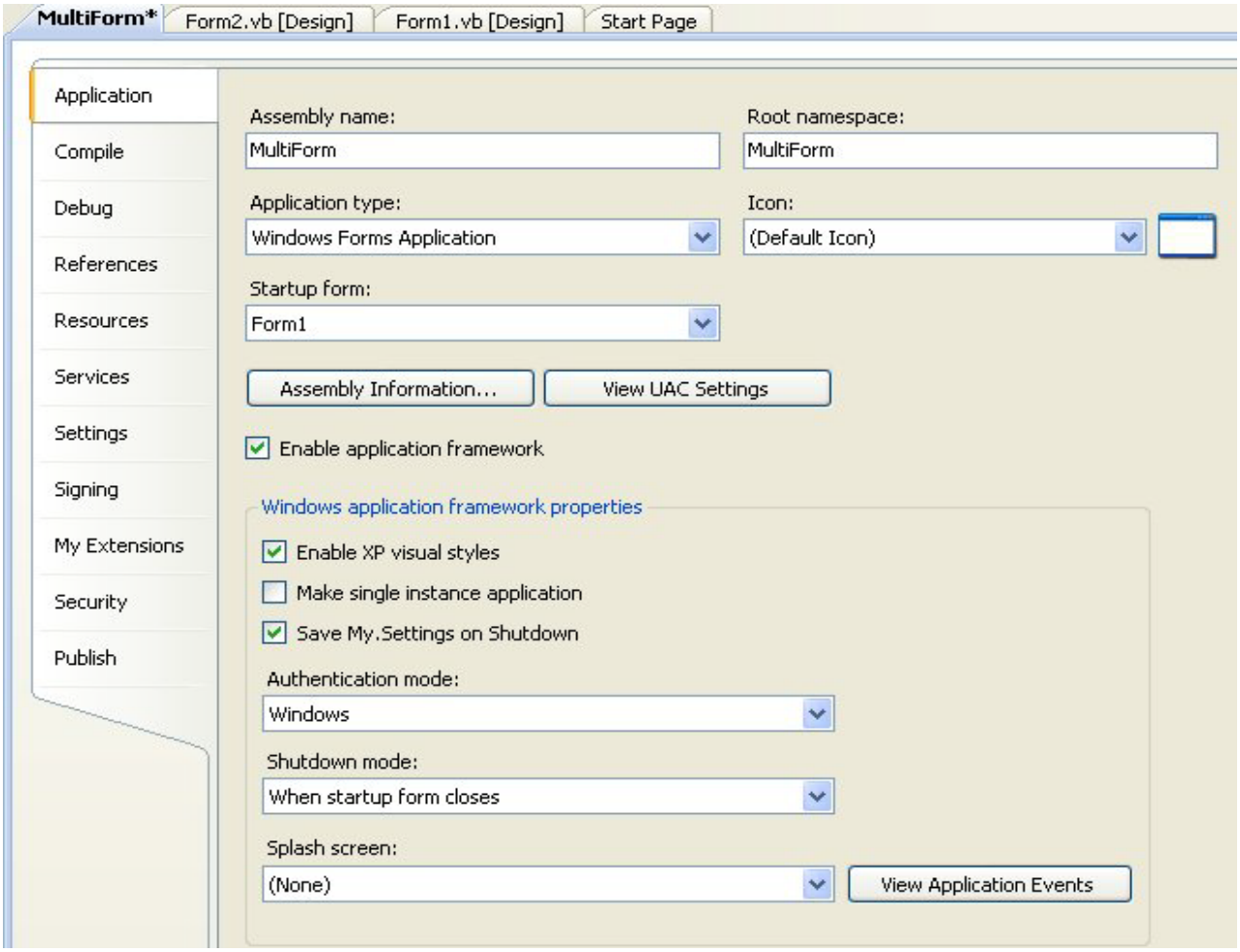
Of course, the code in frmAccount to close frmAccount is still:

```
Me.Close
```

A form can also be closed when the user clicks the Close button on the form's title bar.

Figure 11-3
Changing startup object





Modal and Modeless Forms

The preceding code will show the form in *modeless style*, the default. A form can be displayed in either *modal* or *modeless style*. When displayed as a modeless form, it acts independently of the form that invokes it. All remaining code in the procedure that invokes the form will continue to execute. In addition, the user can change focus among all forms that appear by clicking on the form of interest.

Modal Form and Custom Dialog Box

A modal form will stay on top of the current application. No other forms in the same project can get focus and obtain input until this form is either closed or hidden. In addition, the remaining code in the procedure that invokes the modal form will not be executed until the modal form is either closed or hidden.

Modal forms are usually used as custom dialog boxes. A *custom dialog box* is a form designed to solicit special input from the user. For example, in an account data-entry form, the user may have difficulty recalling the account number of an account although the account name is known. In such a case, you may provide a special lookup button that the user can click to look for the account number given an account name. When the button is clicked, another form is displayed to help the user search for

the account number. The form designed for such purposes is recognized as a custom dialog box, as opposed to the common dialog box provided by VB to help your code obtain certain information commonly required for input/output operations. To display a form as modal, use the form's *ShowDialog method* instead of the Show method. The modal form will be discussed later in this chapter.

Hiding and Closing Forms

You can make a form disappear from the screen either by closing it or by hiding it. To hide a form, use the *Hide method*, which has the following syntax:

```
Object.Hide
```

where Object represents the form to hide. Again, if the code is in the code module of the same form, to hide the form, you will code the following:

```
Me.Hide
```

Hiding a form is equivalent to setting its Visible property to False. Although not visible, the form continues to exist and function. It can be made to reappear with its Show method.

Differences Between Closing and Hiding a Form

Both closing and hiding a modeless form will make the form disappear; however, there are some differences between the two actions:

- The data values in a closed form are reset, while those in a hidden form are preserved.
- When you use the Show method to invoke the form again, it takes a bit longer to make a closed form to reappear because it has to be reconstructed.

Note also that there are no differences between closing and hiding a modal form. Data values in the form are preserved and continue to be accessible.

Ending a Project

Chapter 2, "Introduction to Visual Basic Programming," briefly discussed how to end a project. It was suggested that the form's Close method be used instead of using the End statement. The End statement ends the project abruptly. No events such as FormClosing or FormClosed will be raised when the End statement is executed. This can cause problems. For example, the form may still have unsaved data. Because the FormClosing event will not be raised, there will be no way to warn the user or to save the data. On the other hand, the Close method will not cause this problem.

Note also although closing the startup form will trigger the form's own Closing event, all other forms will terminate without triggering their respective FormClosing events. When you do use the FormClosing events in other forms, this can be a serious problem. For example, you may have written code to close a StreamWriter in the FormClosing event; but if the event is not triggered when the project ends, the data written by the StreamWriter will be lost. The next sub-subsection presents a solution.

Ensuring the Closing Event in Each Form to Execute

To ensure the FormClosing event of each open form to execute properly when the project terminates involves the use of the OpenForms property of the Application object, which provides various static methods and properties to manage your application. The OpenForms property (object) automatically keeps track of all open forms in your project. Its Item collection contains all forms currently open. For example, the following code will display the names of all forms which are currently open:

```
Dim I As Integer
```

```
For I = 0 To Application.OpenForms.Count - 1
    MsgBox(Application.OpenForms.Item(I).Name)
Next
```

If you execute the code, you will notice that Item(0) is always the startup form. Thus, to close all other forms before the startup form, you can place the following code in the FormClosing event of the startup form:

```
Dim I As Integer
For I = Application.OpenForms.Count - 1 To 1 Step -1
    Application.OpenForms.Item(I).Close()
Next
```

Note that the loop counter (and therefore the collection's index in the loop) starts with the collection Count property minus 1 and goes down to 1. Because the routine is being executed in the FormClosing event, you need not to close the startup form again. Also, when an item with a lower index is removed from the collection, the higher indices are adjusted down; therefore, if the item with an index value of 1 is removed first, the highest index will no longer be in the collection, causing an error when the counter reaches that value. Removing the item with the highest index first (as the way your loop does) will not cause this problem.

Tip

An alternative to the above countdown loop, you can remove Item(1) in the loop instead of an index of varying value as shown following. It will work because as explained in the text, the collection's indices are adjusted as soon as an item is removed from its list.

```
For I = 1 To Application.OpenForms.Count - 1
    Application.OpenForms.Item(1).Close()
Next
```

Sharing Data Between Forms

In multiple-form applications, you may have a need to refer to a property of a control in another form. Controls are declared to have a Friend scope by default and therefore are accessible by other forms in the same project. A control property in one form can be referenced in another form by the syntax:

```
FormName.ControlName.Property
```

To display the Text property of the text box txtName on the form, frmAccount, you can code in another form (such as a form to prepare a report) as follows:

```
MsgBox("The account name is " & frmAccount.txtName.Text)
```

Declaring Variables for Broader Access

Variables in a form declared with a Dim or Private statement cannot be accessed by another form. To make a variable accessible by other forms, you need to declare it with either the *Public* or *Friend modifier*. A variable of Public scope is accessible to any entity without any restriction. A variable with Friend scope is accessible to all modules in the current project. To make the variable AccountName

available to other forms, you can code either of the following statements in the general declaration section as follows:

```
Public AccountName As String, or
Friend AccountName As String
```

In other forms, you can refer to this variable by the following syntax:

```
FormName.VariableName
```

To display the AccountName in frmAccount in another form, you will code the following:

```
MsgBox("The account name in form account is " & frmAccount.AccountName)
```

Potential Problems with Accessing Data in a Form Directly

When you refer to data in another form directly as previously shown, you can encounter unexpected problems because you can never be sure that your code is referencing to a form that the user has closed. As mentioned previously, if a form is opened with the ShowDialog method (therefore is used as a dialog box), it is safe to reference the dialog box's contents even if is closed; however, if a form is opened with the Show method, its contents will be lost after it is closed. Whatever your code retrieves can be invalid.

Keeping a Form from Unloading

Ensuring that a form remains in memory can become fairly complicated. For example, although you may use the form's Hide method in code to ensure that the form you will make reference to later remains in memory, this may not be enough. The user may use the Close (x) button to close the form. You can solve this problem by one of the following approaches:

- Set the form's *FormBorderStyle property* to None. In this case, the form will show no title bar (therefore, no control box). The form's appearance may not be exactly what you want.
- Set the form's *ControlBox property* to False to minimize the user's flexibility to maximize or minimize the form.
- Place the code in the FormClosing event to hide the form and cancel the closing operation as shown below; however, it takes extra code and effort.

```
        Me.Hide() 'Hide the form
        e.Cancel = True 'Do not close
```

As you can see, the code is fairly simple, but you will need to take care of this for every form whose control properties will be accessed by other forms. You also need to find a way to truly close the form at the end of the project. A more secure (and easier) way to share data among forms is to use a standard module, discussed in the next subsection.

The Standard Module

The code you place in a form's code window is kept in a *code module*, which will be called the form module. Other kinds of code modules also exist. The *class module* is used to store code for classes, which are the template of objects. The use of the class module is discussed in Chapter 12, "Object-Based Programming." Finally, the *standard module* is used to contain code and data common to other code modules.

The standard module contains only code and variables. It has no form or any container in which you may place any control or visible object; however, you can use it to keep procedures and data that are

global in nature. These elements can then be accessible to all modules (form, standard, and class modules) in the project.

Adding Standard Modules to a Project

To add a new standard module to a project, follow these steps:

- Click the Project menu.
- Select the Add Module option to open the Add New Item dialog box.
- Enter the proper name for the module in the name box (begin the name with the mod prefix); then click the Open button.

A project can have as many standard modules as you need. In a large project, you may find it desirable to include more than one module, each specialized in a particular aspect. Procedures that are useful to all projects should be placed in a module (or modules) separated from procedures that are to be used only by forms in a particular project. Modules arranged in this manner can enhance their reusability.

Declaring Variables and Procedures in a Standard Module

You can declare variables and procedures in a standard module in the same way as in a form module. Similar rules for scope and duration apply:

- Variables declared within a procedure are recognized only within the procedure.
- Variables declared with a Dim statement in the procedure are reinitialized each time the procedure is called.
- Variables declared with a Static statement retain their values until the project ends.
- Variables declared at the module level with a Dim or Private statement are recognized by all procedures in the module, but are not accessible to other forms or modules. These variables retain their values until the project ends.
- Variables declared with a Public modifier are recognized globally.
- Variables declared with a Friend modifier are accessible to all procedures in the current project.
- Both Public and Friend variables retain their values until the project ends.
- Public, Friend, and Private variables can be declared only in the general declaration area.
- Static variables can be declared only inside a procedure.

These rules are the same as those explained in Chapter 4, “Data, Operations, and Built-In Functions.”

Duration of the Standard Module and Form Module

One difference between the standard module and the form is their time duration. The standard module exists throughout the entire life of the project, whereas the form can be closed and destroyed before the project ends (or loaded after the project starts). Global and module-level variables in the standard module exist for the entire life of the project; in the form, they exist for the entire life of the form, which can be shorter than the life of the project.

Public, Friend, and Private Procedures

The modifiers Public, Friend, and Private are also used to define the scope of Sub and Function procedures and will have exactly the same scope as the variables declared with the same modifiers. For example, consider the following two procedures:

```
Public Sub FirstProc()
    ‘Statements
```

```
End Sub
Private Sub SecondProc()
    'Statements
End Sub
```

FirstProc can be called from any other modules, but SecondProc can be used only within the module. Also, a procedure that is not preceded by the Public, Friend, or Private keyword is a Public procedure by default. These rules of declaration for procedures apply to procedures in forms as well.

Calling Procedures in Another Module

To call a Public or Friend procedure in a standard module, the syntax is exactly the same as if the procedure were in the same form (or standard module); therefore, to call FirstProc from a form, you will code the following:

```
FirstProc()
```

To call a Public procedure in a form, you must qualify the procedure with the form name. For example, assume in a form named frmAccount that there is a public function ComputeBalance, which requires an account number as its parameter and returns a value as the account balance. To use the function from another module (form or standard module), you can code the following:

```
AccountBalance = frmAccount.ComputeBalance(AccountNumber)
```

Of course, here you assume that AccountBalance and AccountNumber have been properly declared, and that AccountNumber has been assigned a proper value.

Another Way to Share Variables Among Modules

Potential problems with accessing data in a form directly have been mentioned because of the possibility that the form has been closed prematurely. One possible solution is to use a standard module in which you place all variables to be shared by various forms. The form that provides the value can then assign it to the variable in the standard module. After this is done, whether this form is closed is no longer a concern. All other forms or modules can access the value simply by referencing the variable in the standard module.

In the previous example, suppose various forms need to access the Text property of txtName in the form frmAccount. To use the standard module as the bridge to keep the data, you can first add a declaration in the module as follows:

```
Public AccountName As String
```

In frmAccount, you can then assign the value as follows:

```
AccountName = txtName.Text
```

From then on, in all other forms or modules, you can use the value regardless of the state of frmAccount. Your code in these modules can appear as follows:

```
MsgBox("Account name is " & AccountName)
```

Notice that the (standard) module name does not need to be referenced. The variable name alone is sufficient in most cases.

Variables and Procedures in Multiple Modules

As noted, you can have more than one standard module in a project. Can you then declare the same Public or Friend variable (and procedure) names in these modules? Yes. If you declare the same Public or Friend variable (procedure) name in two standard modules, this name actually represents two different variables (or procedures) in the two modules. To refer to each variable or procedure, you will need to qualify each with the module name. For example, suppose the variable name AccountName is

declared as Public in two modules: modAccounts and modInsurance. To refer to that variable in modAccounts from another module, you will need to code:

```
modAccounts.AccountName
```

The syntax in referring to these variables/procedures is the same as that for those Public or Friend variable/procedure names declared in a form. Failure to qualify an identifier declared in multiple modules with the proper module name will result in an ambiguous reference error.

A Multiple-Form Example

To illustrate how to code an application with multiple forms, consider an example that involves a number guessing game. When the project starts, the program displays an instruction explaining how to play the game, and instructs the game player (user) to enter his/her name and click the Ready button to proceed—to go on to the next form. When the user clicks the Quit button, the program tells the user how many games he/she has played and the amount of time used.

A label in the second form gives more specific instructions about the game. The form has two buttons: Start A Game to start playing a game, and Bye. When the first button is clicked, a label and a text box are enabled. In addition, a number between 1 and 1,000 is selected at random, which will be the number for the player to guess. After the user enters a number in the textbox, the program checks whether the entered number is equal to the selected random number. If so, it displays the message "That's right!" with the player's name, and informs the player of the number of times he/she tried and the amount of time used. If the guessed number is not equal to the random number selected, a label on the right of the text box gives a hint indicating whether the actual number is higher or lower. The player can play as many games as he/she wishes. Behind the scene, a streamwriter is used to record the player name, times attempted, and time used. This file is open when the second form is loaded and is not closed until this form is closed.

In addition to showing how each form is started and the variables that can be shared in a multiple form project, this project also touches upon a few technical aspects discussed in the preceding subsections.

Setting Up the Guessing Game Project

This project will involve one standard module and two forms. Follow these steps:

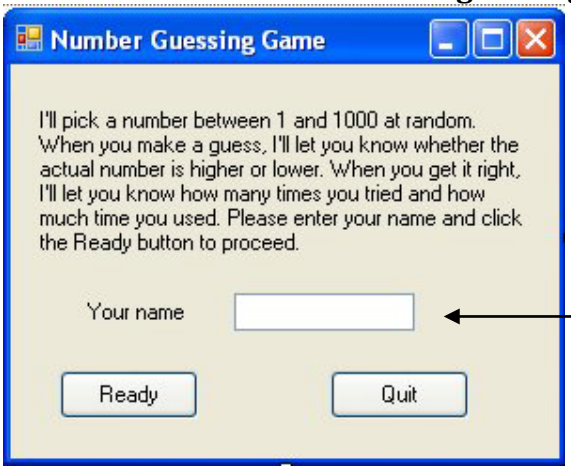
- Create a new project and name it GuessingGame.
- Rename the form as frmGreet. Right click the Form1.vb icon in the Solution Explorer, select Rename from the context menu, and change Form1.vb to frmGreet.vb.
- Add another form. Click the Project menu and then select the Add Windows Form option. The Add New Item dialog box will appear. Enter **frmGuess** in the Name box and then click the Open button.
- Add a module. Click the Project menu and then select the Add Module option. The Add New Item dialog box will appear. Enter **modGuessingGame** in the Name box and then click the Open button.

Designing the Visual Interface

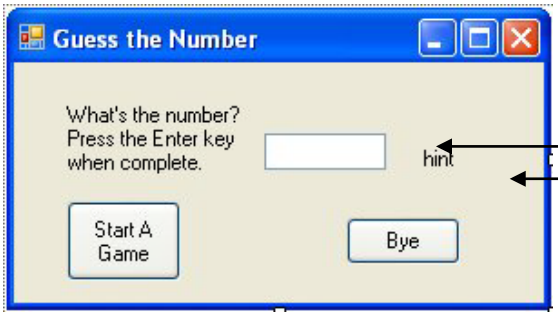
The two forms of this project appear as in Figure 11-4. The following table explains the names and purposes of the forms and controls.

Control	Name	Purpose
In frmGreet		
Text box	txtPlayerName	To obtain player's name
Button	btnReady	To invoke the game form frmGame
Button	btnQuit	To end the program
In frmGame		
Label	Label1	To show instructions for the game
Text box	txtNumber	For the user to enter the guessed number (*Note:* Set its Enabled property to False)
Label	lblGuess	The label for txtNumber (*Note:* Also set its Enabled property to False)
Label	lblHint	To display hint telling the player whether the actual number is higher or lower
Button	btnStart	To start a game
Button	btnBye	To make the game form disappear

Figure 11-4
Visual interface of the number guessing game



This startup form contains instructions to play the game and prompts for the player's name.



The player will play the number guessing game with this form. The label named lblHint tells the player whether the actual number is higher or lower.

Coding the Project

The project involves one standard module and two forms. Let's consider the code in each module in the following order:

- The standard module
- The startup form (frmGreet)
- The game form (frmGame)

Code in the Standard Module

The standard module provides a code container for the variables and procedures to be shared by all forms. Three variables for this purpose are declared as follows:

```
Module modGuessingGame
    'Variables for all forms to share
    Public PlayerName As String
    Public GameCount As Integer
    Public TotalSeconds As Integer
End Module
```

The number of games played (GameCount) and total time used (TotalSeconds) are declared here with a Public scope. These variables are updated in the second form (frmGame) when the games are played. Having them declared here simplifies sharing the values between forms. You will never have to worry about whether the second form is closed and invoked again by the user even though such actions will cause data values in this form to reset.

The player's name obtained from the user in frmGreet (through txtPlayerName) will be referenced in frmGame. As long as the program is running, the startup form, frmGreet will exist. Therefore, there will be no problem to directly reference txtPlayerName from the second form, frmGame. It is not necessary to declare another variable for the player name here. Its declaration here is simply for the sake of uniformity in sharing data among forms.

Code in the Startup Form

As you would expect, when the project starts, the startup form (frmGreet) should appear and wait for the player to enter their name and click the Ready button. The Click event procedure should:

- Check to ensure the player's name is entered.
- Proceed (if the name is entered) to:
    - Display the second form (frmGame).
    - Move the player's name to the Public variable in the standard module for the other form to access.

The procedure should appear as follows:

```
Private Sub btnReady_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs)
  Handles btnReady.Click
    If Len(Trim(txtPlayerName.Text)) = 0 Then
        'Name has not been entered. Display a message
        MsgBox("Please enter your name to proceed.")
        txtPlayerName.Focus() 'set the focus on the name box
        Exit Sub
    End If
    'Display the game form
    frmGame.Show()
    'Keep name in standard module for all forms to share
```

```
    PlayerName = txtPlayerName.Text
End Sub
```

The If block tests whether the player's name is present. If not, a message is displayed and the focus is set on the text box for the user to enter the name. The last statement in the procedure moves the entered player's name to the PlayerName variable, which is declared in the standard module as a Public string variable.

When the user clicks the Quit button in the main form, the program should close. Thus, you code:

```
Private Sub btnQuit_Click(ByVal sender As System.Object, ByVal e As
  System.EventArgs) Handles btnQuit.Click
    Me.Close()
End Sub
```

Ensuring All Forms Are Closed Properly

When the user quits, the program will inform the user the number of games played and time used. This statement is coded in the FormClosing event.

As explained previously, if you do not do anything about the other forms when the startup form is closing, none of the other forms' FormClosing events will be triggered. This may cause problems if your code in the FormClosing events is to perform important activities at closing; therefore, the loop to close the other forms (as explained in the preceding subsection) should also be included:

```
Private Sub frmGreet_FormClosing(ByVal sender As Object, ByVal e As
  System.Windows.Forms.FormClosingEventArgs) Handles Me.FormClosing
    Dim I As Integer
    MsgBox("You played " & GameCount & " times and used " & TotalSeconds & " seconds
      today.")
    For I = Application.OpenForms.Count - 1 To 1 Step -1
        Application.OpenForms.Item(I).Close()
    Next
End Sub
```

Notice that these statements are placed in this form's FormClosing event, not in the Quit button's Click event. The code in the FormClosing event gets executed either when the code initiates the closing (as in the Quit button's Click event) or when the user clicks the Close button in the control box. In contrast, the code in the Quit button's Click event is executed only when the user clicks that button.

Notice also that there is no declaration for the Application object. The system automatically creates and recognizes it. This routine to close all forms in a project is generally applicable. Any forms that you may add to the project in the future will be automatically closed here, requiring no future maintenance chore. On the other hand, if you simply code the following in place of this routine, you will need to modify the code when you add more forms to the project.

```
    frmGame.Close() 'Not as good for code maintenance
```

Coding the Game Form

As explained in the game introduction, the program records the player, number of guesses, and time used for each game. The streamwriter to perform the recording is opened when the game form starts and closed when this form closed. This object (StatFile) should be declared at the class level for different event procedures to access. In addition, the number to be guessed and the time that the game starts are set when the user clicks the Start button and updated/used when the user enters the guessed number and

therefore for the same reason should be declared at the class level. The declaration for these variables appears:

```
Dim StatFile As System.IO.StreamWriter
Dim TheNumber As Integer
Dim StartTime As Date
```

As soon as the game form loads, the streamwrite (StatFile) should be associated with the physical file and ready to record any game results. The code appears:

```
Private Sub frmGame_Load(ByVal sender As System.Object, ByVal e As System.EventArgs)
  Handles MyBase.Load
    StatFile = New System.IO.StreamWriter("GameStat.txt", True)
End Sub
```

The Closing Event in the Game Form

When the second form (frmGame) is being closed, the file used to record game results should be closed. The code appears as follows:

```
Private Sub frmGame_FormClosing(ByVal sender As Object, ByVal e As
  System.Windows.Forms.FormClosingEventArgs) Handles Me.FormClosing
    StatFile.Close()
End Sub
```

Keep in mind that if the file is not closed when the form is closed, the data added during the current session will be lost. As a reminder, this event will not be triggered if this form is not either closed by code or by the user's clicking the Close button of this form's control box. Specifically, if this form is still open and the startup form is closed without invoking any code to close this form, then this event will not be triggered. This is why it is necessary to provide the code in the startup form's FormClosing event to close all other forms.

Starting the Game

The text box (txtNumber) to enter the guessed number, and its label (lblGuess) should be disabled at design time. This way, the player cannot enter the number before the program is ready for a game—the number to be guessed needs to be set first. When the Click here to start a game button is clicked, the preceding two controls are enabled with the following statements:

```
txtNumber.Enabled = True
lblGuess.Enabled = True
```

A random number in the range of 1 to 1,000 (inclusive) should be picked and kept in the variable TheNumber. Suppose the variable for the Random object is declared to be RandomEngine. The statement should appear as follows:

```
‘ Pick a random number between 1 and 1000
TheNumber = RandomEngine.Next(1, 1001)
```

The text box should also be cleared, and the focus set on it, ready for the user to guess the number:

```
txtNumber.Text = “”
txtNumber.Focus
```

Finally, the start time should be assigned by current time, Now:

```
StartTime = Now ‘Start counting elapse time
```

The complete procedure appears as follows:

```
Private Sub btnStart_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs)
  Handles btnStart.Click
    Dim RandomEngine As New Random()
    'Enabled the number box for guessing
    txtNumber.Enabled = True
    lblGuess.Enabled = True
    'Pick a random number
    TheNumber = RandomEngine.Next(1, 1001)
    txtNumber.Text = ""
    txtNumber.Focus()
    StartTime = Now 'Start counting time
End Sub
```

Responding to an Entered Number

The player is instructed to press the Enter key after entering a number in the text box (see the interface design); therefore, the code to check whether the number entered is correct is placed in the text box's KeyPress event. The procedure will be executed only when the Enter key is pressed. The entered number should be compared with the number to be guessed (TheNumber). If the two match, a "That's right!" message along with the player's name is displayed. The game results are recorded, count for the number of trials is reset, and total games played and cumulative time are computed; otherwise, the program shows a hint, either "Higher" or "Lower" depending on the comparison. The code appears as follows:

```
Private Sub txtNumber_KeyPress(ByVal sender As Object, ByVal e As
  System.Windows.Forms.KeyPressEventArgs) Handles txtNumber.KeyPress
    Static GuessCount As Integer
    Dim TheAnswer As Integer
    Dim KeyCode As Integer
    Dim Duration As Integer

    KeyCode = Asc(e.KeyChar)
    'check the number only if this is a return key or there is number to check
    If KeyCode <> Keys.Return OrElse Len(Trim(txtNumber.Text)) = 0 Then
        Exit Sub
    End If
    e.Handled = True 'Suppress the Return key
    GuessCount += 1
    TheAnswer = CInt(txtNumber.Text)
    If TheNumber = TheAnswer Then
        'Right answer, give number of attempts and time
        lblHint.Text = ""
        Duration = DateDiff(DateInterval.Second, StartTime, Now)
        MsgBox("That's right! " & PlayerName & "." & vbCrLf & _
        "You tried " & GuessCount & " times " & vbCrLf & _
        "and used " & Duration & " seconds " & vbCrLf & _
        "to get the number.")
        StatFile.WriteLine(PlayerName & vbTab & GuessCount & vbTab & Duration)
        GuessCount = 0 'Clear the counter
        txtNumber.Text = "" 'Clear the text box
        'Force the user to click the Start a Game button again
        lblGuess.Enabled = False
        txtNumber.Enabled = False
```

```
        TotalSeconds += Duration 'Track total seconds
        GameCount += 1 'One additional completed game
    ElseIf TheNumber < TheAnswer Then
        lblHint.Text = "Lower"
    Else
        lblHint.Text = "Higher"
    End If
    'Highlight the previous number
    txtNumber.Select(0, Len(txtNumber.Text))
End Sub
```

The meaning of the code should be apparent. Pay particular attention to the way the player's name is referenced in the first line of the MsgBox statement. The PlayerName variable is used because before this form starts, that variable (declared as Public in the standard module) has already been assigned the Text property of the text box that accepts the player's name in the first (startup) form. You can also reference the text box directly. The control's name should be qualified by the form name (with a dot). The text property of that text box can be coded as follows:

```
frmGreet.txtPlayerName.Text
```

Notice also that GameCount is increased by 1 each time the player guesses the right number. So is TotalSeconds added by the time used. These two variables are declared in the module. Their values are preserved for the duration of the project even when the game form is closed more than one time.

Quitting the frmGame Form

When the player clicks the Bye button, the form should be closed:

```
Private Sub btnBye_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
  Handles btnBye.Click
    Me.Close()
End Sub
```

Additional Notes on Multiple Form Applications

The above example shows how to deal with the additional issues pertaining to multiple form applications. These include invoking other forms, sharing data among forms, and ensuring that the non-startup forms' FormClosing events are triggered. The example, however, does not involve the use of the customized dialog box, which is lustrated in the next section.

11.3 MDI Applications

In the preceding section, the forms used in a project have no parent-child relationship with each other. Each form presents itself as a single document; that is, only one document of a similar nature appears in the project at one time. Applications involving forms of this kind are referred to as *single document interface (SDI) applications*. Most real-world applications are of this type. Some applications involve forms of *multiple document interface (MDI)*. Several documents of similar nature are loaded in different windows of the same project at the same time. The familiar applications of this kind are word processing and spreadsheet applications. Additional examples of this style of application include the following:

- Real estate property listing in which several real properties (each in a different window) are shown for review and comparison

- Graphic design in which multiple pictures are shown for cropping, cutting, pasting, and other drawing/painting operations
- Music composition in which multiple pieces of music are displayed for the composer to analyze or to cut and paste like in word processing operations

In these applications, multiple documents can be loaded. Each document appears on a window (form) and handles different data (document), but behaves in the same manner and shares the same menu bar. All these documents are child forms of the MDI (parent) form. There is a parent-child relationship between these documents and the MDI form.

Differences Between MDI and SDI in Behavior

You can tell whether an application is SDI or MDI by the manner in which the forms (documents) interact with each other. Here is a selected list of the differences in their behavior and restrictions:

- MDI child forms stay within the boundary of their parent form; SDI forms can appear on any part of the screen, independent of each other's location.
- When the MDI parent form is minimized, all child forms are also minimized; minimizing an SDI form does not affect any other forms.
- When an MDI child form is minimized, its icon stays within the boundary of the MDI (parent) form. When an SDI form is minimized, its icon appears on the taskbar of the Windows system.

Creating an MDI Application

Creating an MDI application involves several issues that are not encountered in an SDI application. You will need to be aware of at least the following How To topics:

- Set up an MDI application. The application itself involves at least two forms: one to serve as the parent; and the other as child form. You will need to know how to set up the parent-child relationship.
- Create copies of child windows (form). When the project starts, only the parent (startup) form will be present. You need to provide code for the user to either create a new window (form; document) or load an existing document.
- Merge menus. The parent form and the child form each may contain different menu items and options. Often you want to present them as if they belonged to the same menu when both forms appear.
- Present the child window list and arrange these windows in the parent window. When multiple copies (documents) of the child window are present, the user needs a way to navigate around different documents. This is typically treated by providing Window menu with a child window list as well as a list of methods to arrange them in the parent window (such as cascade or tile).
- Reference a current document. When your code to work on the current document (copy of the child form) is located in a different module, you will need a way to reference the current active document.

The following section discusses these issues by a hands-on example.

11.4 Coding an MDI Project: An Example

To illustrate how to handle those special issues involved in an MDI project mentioned in the preceding section, let's consider a simplified example. You will create a program that behaves similar to Notepad except that the program will be able to handle multiple documents; therefore, you are creating an MDI project. This will involve the following the major steps:

- Designing the features of the program,
- Creating the MDI interface, and
- Implementing the features.

Features of the Project

Here is the list of features/capabilities the project is expected to provide:

- As a Notepad application, the program will work mainly with a text file with a .txt extension. It will also allow the user to use other file extensions as long as the file itself contains only text.
- When the project starts, no document will be displayed until the user chooses either to create a new document (New) or open an existing document (Open) from a File menu. There will be only one menu (the File menu) on the menu bar when the program starts, or when there is no open document.
- When there is (are) open document(s), two additional menus, Find and Window, will appear on the menu bar. The Find menu enables the user to search on the current document for a text string with a custom Search dialog box. The Window menu provides options to arrange the documents, such as cascade or tile. In addition, it also provides a list of open documents from which the user can select one to work on. Finally, the File menu will be expanded to include the Save and Close options.
- When a document is being closed (child form being unloaded), the program will prompt the user whether to save the document. When the application (project) is being closed, all open documents will be handled similarly.

Creating the Interface

As explained previously, an MDI application needs at least two forms. In addition, this project needs a form to serve as the Find dialog box. Therefore, your project will include the following forms:

Form name	**Comments**
mdiNotepad	Startup form To serve as the MDI container (parent)
FrmMdiChild	MDI child form
FrmFind	To serve as the dialog box to search for text in a document window

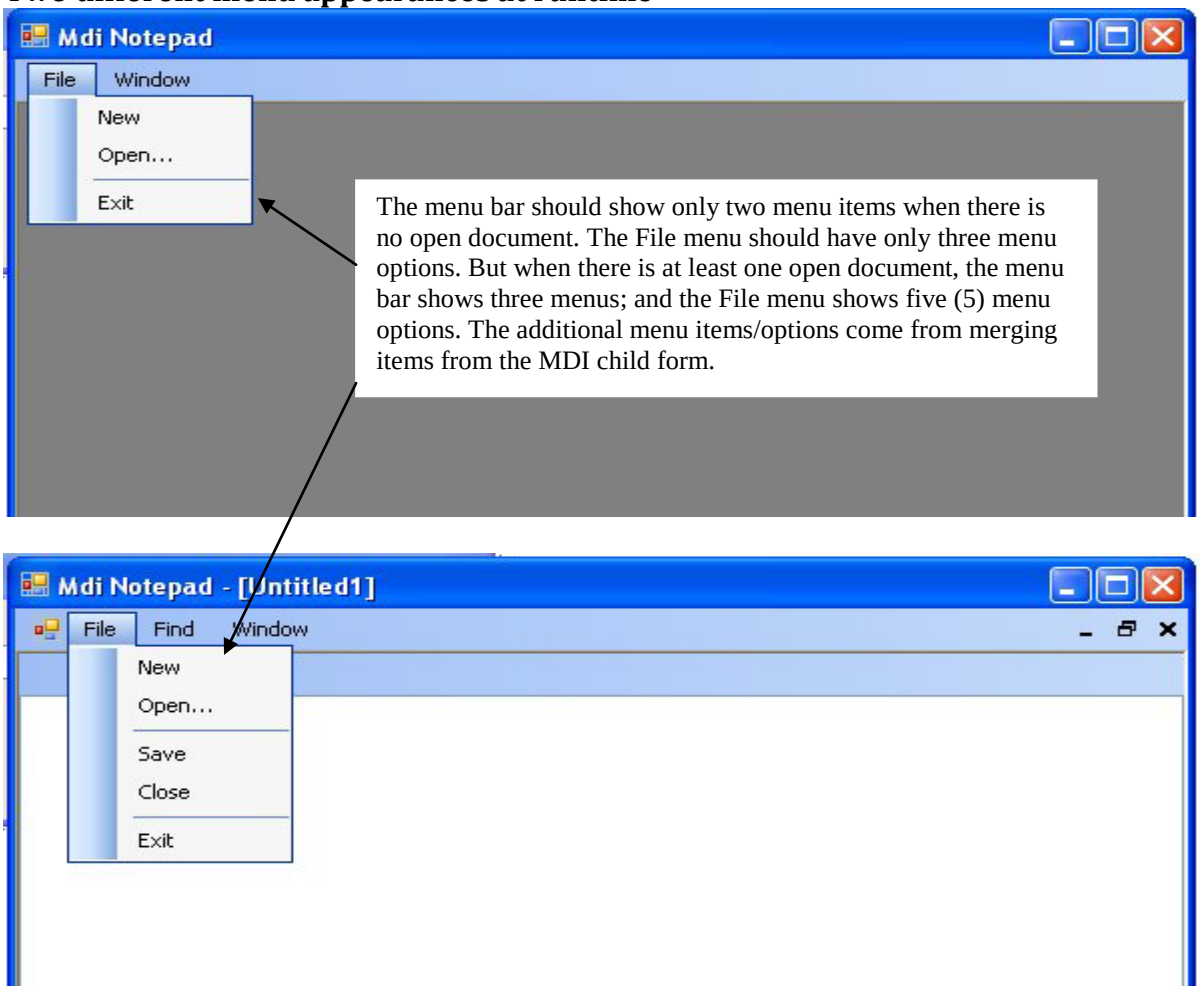
Designing the Menu Bar(s)

There are several ways to handle the menu bar. Perhaps the most convenient way for coding is to use two menu strips, one in the MDI (parent) form, one in the child form, each handles its own

requirements. The menu strip in the parent form can be set up to contain the File and Window menus. The file menu will have the New, Open and Exit options to create new or load existing documents as well as quit the application, while the Window menu can be used to provide options to arrange the child windows as well as show the list of current open documents.

The menu strip in the child form can be set up to have the menu File and Find menus. The file menu will have the Save and Close options to save or close the current child window, while the Find menu will be used to invoke the frmFind dialog box. At run time, when there is no open document (child window), only File the menu in the parent form will appear. However, when there is at least one open document, the menus from both parent and child forms will be presented as one menu as shown in Figure 11-5. You will use this approach.

Figure 11-5
Two different menu appearances at runtime



Creating, Setting Up, and Merging the Menus

To create the two menus, draw a menu strip on the parent form (frmNotepad) and another on the child form (frmMidChild). To name the menu strip, click the main menu below the form and then enter the name in the Name property in the Properties window. Name the two menu strips **mnuParent** and

mnuChild. To set up the menu items for mnuParent, click this control (below the form) first; then proceed to enter the text for each menu item, and set its name as described in Section 11.1. Do the same for mnuChild in the child form. Be sure that you click the control below the form before doing anything for the menu strip.

The following table shows the structure of each main menu.

In mnuParent			
First level menu Item: Name	First level menu Item: Text	Second level menu Item: Name	Second level menu Item: Text
MnuFile	&File	mnuFileNew	&New
		mnuFileOpen	&Open
		Separtor1 (ToolSrtipSeparator1)	
		mnuFileExit	E&xit
MnuWin	&Window	mnuWinCascade	&Cascade
		mnuWinHorizontal	Tile &Horizontal
		mnuWinVertical	Tile &Vertical
In mnuChild			
MnuFile	&File	Separator1 (ToolStripSepartor1)	
		mnuFile1Save	&Save
		mnuFile1Close	&Close
MnuFind	Fi&nd		

Additional settings are needed in order for the forms and their menus to behave as described. They are listed in the following table:

Form	**Control or menu item**	**Property**	**Setting and comment**
frmMdiNotepad (parent)	(form itself)	isMdiContainer	True (to make this form the MDI parent)
	mnuParent	AllowMerge	True (default)
		MdiWindowListItem	mnuWin (so that mnuWin will display a list of open (child) documents)
	mnuWin	Visible	False (so that it will appear only when there is open child document as handled by code)
frmMidChild	mnuChild	AllowMerge	True (default)
	mnuFile	Visible	False (so that File will not appear in the child window menu)
		MergeAction	MatchOnly (so that its options are combined with mnuFile in the parent form)
	Separator1	MergeAction	Insert
		MergeIndex	2 (so that this separator will appear right below mnuOpen in the parent File menu)
	mnuFileSave	MergeAction	Insert
		MergeIndex	3

	mnuFileClose	MergeAction	Insert
		MergeIndex	4
	mnuFind	MergeAction	Insert
		MergeIndex	1 (so that this item will appear right after mnuFile)

The isMdiContainer property of frmMdiNotePad is set to true so that it can serve as the container of this Mdi application. You will notice the form's background color is changed to darker gray as soon as you get this property set. Also, in versions beginning with VB.NET you can have as many Mdi parent forms as your application requires, unlike the previous VB versions, which restricts each application to only one parent form. As you will see in the next subsection, at runtime, you will also need to set the MdiParent property of the child form to this parent form in order to establish the parent-child relationship.

To make the Window menu display the list of open documents, the menu strip, mnuParent's MdiWindowListItem property should be set to that menu item, which is named mnuWin. Since you do not want this menu to appear when there is no open document, its Visible property is initially set to False.

Merging menu items from different menu strips involves setting the MergeAction and MergeIndex properties of the menu items involved. The MergeAction property of mnuFind is set to Insert so that this item will be added to the menu bar. Its MergeIndex is set to 1 so that it will appear as the second item (first item has an index value of 0) on the bar. Note that if you set the MergeAction to Append, this item will appear at the end of the menu bar no matter what value you set for the MergeIndex. Since you want to combine the File menu of the child form with that in the parent form, the MergeAction property of the File menu in the child form is set to MatchOnly. Its options' MergeAction and MergeIndex properties are set with values such that they will appear as specified in the project requirements.

The Document on the Child Form

Because the child form will be used to handle the text document, the control to display the document should be placed in this form. The controls that are suitable for this purpose include the text box and the *rich text box control*. For this purpose, the familiar text box will be used to display the document. Add a text box on the child form, and name it **txtDocument**. Set its MultiLine property to True, Dock property to Fill, and the ScrollBars property to Vertical. This will enable the text box to display and handle the document properly.

The Custom Search Dialog Box

Details of this form are covered in the subsection "Building the Find Dialog Box" later in this chapter.

Coding the Project

You now have three forms to write code. The events and procedures will be discussed in the order the parent form, child form, and the dialog box. As you proceed, be aware of the specific module in which the code is to be placed.

Handling Events in the MDI Parent Form

Now consider the code in the MDI (parent) form. This form has two menu items: File and Window. The File menu (named mnuFile) has three options New, Open, and Exit. The Window menu (named mnuWin) has Cascade, Tile Horizontal, and Tile Vertical options. It is fairly straightforward to code the event procedure for Exit. Like the SDI application, the Close method should be used. When the MDI (parent) form closes, all its child forms will also close. However, unlike the SDI application, the FormClosing event in each child form will be raised. Thus, there is no need to close each existing child form explicitly (with code) in order to trigger the FormClosing event in the child form. The mnuFileExit event procedure should contain the familiar statement to close the form as follows:

```
Private Sub mnuFileExit_Click(ByVal sender As System.Object, ByVal e As
  System.EventArgs) Handles mnuFileExit.Click
    Me.Close()
End Sub
```

Handling the New Menu Option

The New option in the File menu creates a new child window (form; document), which should also identify its parent form. It is also customary to give this new document a title in the title bar (the form's Text property). The following code serves these purposes:

```
Private Sub mnuFileNew_Click(ByVal sender As System.Object, ByVal e As
  System.EventArgs) Handles mnuFileNew.Click
    'Create a new form of the frmMdihild type
    Dim frmDocument As New frmMdiChild
    frmDocument.MdiParent = Me
    frmDocument.Text = "Untitled" & Application.OpenForms.Count
    frmDocument.Show()
    mnuWindow.Visible = True 'Make window list available
End Sub
```

Notice how a new child form (document) is created. The new child form is declared to have the name, frmDocument. The New keyword specifies that this form should be created with the type of frmMdiChild, which you have set up at design time. In the previous SDI example, a form created at design time is referenced and used directly. It is treated both as an object (something that actually exists like a textbox you draw on a form) and type (something that exists only in concept like the textbox icon in the toolbox). Here the child form from the design time is used as a type. Notice also since you use the New keyword, each time this procedure is invoked, another new document will be created. The statement:

```
frmDocument.MdiParent = Me
```

establishes the relationship between this newly created document and the parent form in which you are currently coding.

The new child form's title bar is given the title of "Untitled" followed by a number provided by the application based on the number of open forms (not counting the new one because it has not been opened yet). Finally, the new form's Show method opens and displays the form itself.

Loading an Existing Document

Now consider the OpenExistingDoc procedure. This procedure is expected to open an existing document. The specific steps should include the following:

- Prompt for the filename, and open the file for input.

- Create a new child window.
- Read the entire file into the text box in the child form.
- Assign the child form's Text property with the filename.
- Make the MDI form the parent of the child form.
- Maximize and display the form.
- Close the file.

The following code serves these purposes:

```
Private Sub mnuFileOpen_Click(ByVal sender As System.Object, ByVal e As
  System.EventArgs) Handles mnuFileOpen.Click
    Dim TheTextFile As IO.StreamReader
    Dim ThefileName As String
    If cdlOpenFile.ShowDialog = Windows.Forms.DialogResult.Cancel Then
        Exit Sub
    End If

    ThefileName = cdlOpenFile.FileName
    TheTextFile = New IO.StreamReader(ThefileName)
    'Create a new window if everything goes right
    Dim frmDocument As New frmMdiChild
    With frmDocument
        .txtDocument.Text = TheTextFile.ReadToEnd() 'Read the entire file
        .Text = ThefileName 'Show the filename in the title bar
        .MdiParent = Me 'Set this window as a child form
        .WindowState = FormWindowState.Maximized
        .Show()
    End With
    TheTextFile.Close()
    mnuWindow.Visible = True 'Make window list available
End Sub
```

Referencing the Active Form

Each time the above two procedures are executed, a new form is created. Note that the variable frmDocument is declared within a procedure and is recognized only in that procedure. From the parent form, to refer to the specific child form that the user is currently working, you should code:

```
    ActiveMdiChild
```

From other forms or modules, you code:

```
    mdiNotepad.ActiveMdiChild
```

Handling the Window Menu Events

The Window menu has three options: Cascade, Tile Horizontal, and Tile Vertical, which refer to how the child forms should be arranged in the MDI workspace. When several documents are open, this Window menu allows the user to choose a different layout of the documents. For this purpose you can use the MDI form's *LayoutMdi method*, which has the following syntax:

```
MdiForm.LayoutMdi(layout)
```

The named constants for the cascade, tile horizontal, and tile vertical arrangements are MdiLayout.Cascade, MdiLayout.TileHorizotal, and MdiLayout.TileVertical, respectively. The code to handle the Window Click events will appear as follows:

```
Private Sub mnuWinCascade_Click(ByVal sender As System.Object, ByVal e As
  System.EventArgs) Handles mnuWinCascade.Click
    Me.LayoutMdi(MdiLayout.Cascade)
End Sub

Private Sub mnuWinHorizontal_Click(ByVal sender As System.Object, ByVal e As
  System.EventArgs) Handles mnuWinHorizontal.Click
    Me.LayoutMdi(MdiLayout.TileHorizontal)
End Sub

Private Sub mnuWinVertical_Click(ByVal sender As System.Object, ByVal e As
  System.EventArgs) Handles mnuWinVertical.Click
    Me.LayoutMdi(MdiLayout.TileVertical)
End Sub
```

Handling Events in the Child Form

Once a child window is loaded, the user can write and edit the document contents in the textbox named txtDocument. The form also has two menu items: File (which has Save and Close options) and Find. The menu settings suggested at the beginning of this section allows the system to deal with menu appearances automatically without any code; i.e., these menu items will appear along with the menu items in the parent form's menu bar when a child form is loaded and disappear from it when there no child form. The event procedures to handle clicks of the menu options in the child form are discussed in the following subsections.

Saving the Document and Closing the Window

When the user clicks the Save menu option, a separate sub procedure to save the document is invoked. The code appears as follows:

```
Private Sub mnuFileSave_Click(ByVal sender As Object, ByVal e As System.EventArgs)
  Handles mnuFileSave.Click
    SaveFile()
End Sub

Private Sub SaveFile()
    Dim Filenname As String
    If InStr(Me.Text, "Untitled") > 0 Then
        If cdlSaveFile.ShowDialog = Windows.Forms.DialogResult.Cancel Then
            Exit Sub
        Else
            Filenname = cdlSaveFile.FileName
        End If
    Else
        Filenname = Me.Text
    End If
    Dim TheFile As New System.IO.StreamWriter(Filenname)
    TheFile.Write(txtDocument.Text)
    TheFile.Close()
End Sub
```

The routine to save the document is written as a separate procedure because the same routine need also be invoked when the form is being closed. The routine involves prompting the user for the filename when one is absent and save the document with a stream writer. Refer to Chapter 6 for information concerning the use of stream writer.

When the user clicks the Close menu item, the child form should be closed with the following event procedure:

```
Private Sub mnuFileClose_Click(ByVal sender As System.Object, ByVal e As
  System.EventArgs) Handles mnuFileClose.Click
    Me.Close()
End Sub
```

Recall that when a document (child window) is being closed, the user is prompted for whether to save the document. This should be handled in the form's FormClosing event as follows:

```
Private Sub frmMdiChild_FormClosing(ByVal sender As Object, ByVal e As
  System.Windows.Forms.FormClosingEventArgs) Handles Me.FormClosing
    Dim Ans As Integer
    'changes in the textbox has not been saved, ask
    Select Case MsgBox("Save the current document?", _
      MsgBoxStyle.Question Or MsgBoxStyle.YesNoCancel)
        Case MsgBoxResult.Yes
        SaveFile()
        Case MsgBoxResult.No
            'don't save
        Case MsgBoxResult.Cancel
            e.Cancel = True 'don't close form
            Exit Sub
    End Select
End Sub
```

The code actually offers the user three choices: to save, not to save, or cancel the closing. If the user chooses to save, the SaveFile Sub is invoked. In either the case to save or not to save, the child form proceeds to close. Otherwise, when the user chooses to cancel, the event's Cancel property (e.Cancel) is set to True; and the form will not be closed.

Invoking the Find Dialog Box

The Find menu's Click event should bring up the Find dialog box (form) for the search of a specified text. Recall that forms used as dialog boxes should be invoked with the ShowDialog method. The code appears as follows:

```
Private Sub mnuFind_Click(ByVal sender As System.Object, ByVal e As
  System.EventArgs) Handles mnuFind.Click
    frmFind.ShowDialog()
End Sub
```

This discussion completes all required code for both the MDI parent and child forms. The following subsection deals with the invoked dialog box.

Building the Find Dialog Box

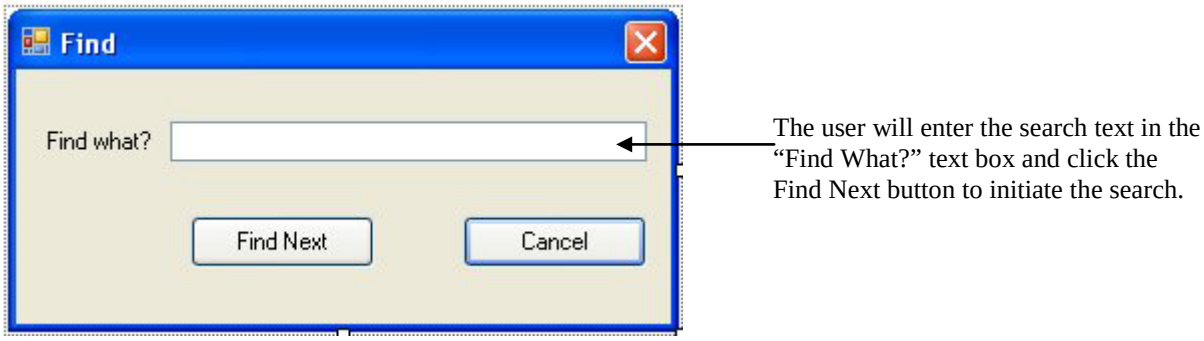
Figure 11-6 shows the visual interface of this custom dialog box, which should behave in the following manner:

- When the user enters the search text in the text box and clicks the Find Next button, the dialog box should find and highlight the text in the document that matches the search text.
- When the user clicks the Find Next button another time, the dialog box should highlight the next match. When there is no (or no more) text in the document that matches the search text, a message "Text not found" is displayed.
- When the user clicks the Cancel button or the Close button on the title bar, the form should disappear.
- When the user clicks the Find menu in the MDI form again, this dialog box will show the previous search text in the text box. It should also "remember" the position of the text in the document that was previously highlighted so that any repeated search will start from there.

This form should have the following controls:

- A text box accompanied by a label with proper text, which will be used by the user to enter the text to find. Name it **txtSearch**.
- A button with the text Find Next. The user will click this button when they finish entering the search text and are ready to search. Name this button **btnFindNext**.
- Another button with the text Cancel. The user will click this button when they are done with the search. Name this button **btnCancel**.

Figure 11-6
The Find custom dialog box



Setting Up the Visual Interface

Take a look at Figure 11-6 again. Notice that the form does not have either the Maximize or Minimize box. This is the typical appearance of a dialog box as there is no need to maximize or minimize the dialog box. You can get this effect by setting the form's *MaximizeBox* and *MinimizeBox properties* to False. In addition, a dialog box usually also accepts the results, disappears when the user presses the Enter key and cancels the results, and disappears when the user presses the Esc key. You can obtain these effects by setting the form's *AcceptButton* and *CancelButton* properties to the respective buttons. In this case, when the user is done with the text search, they can press either the Enter key or the Esc key to close the dialog box. The Cancel button will close the form, so you can set both the AcceptButton and the CancelButton properties to **btnCancel**.

Closing the Form

The form will be closed when the user clicks the Cancel buttons:

```
Private Sub btnCancel_Click(ByVal sender As System.Object, ByVal e As
  System.EventArgs)Handles btnCancel.Click
    Me.Close()
End Sub
```

Preparing the Text Box in the Child Form for Highlighting Selected Text

This dialog box is to be used to search text in the child form's txtDocument text box. While the child form on top can be referenced as ActiveMdiChild, there is no way to directly reference the text box with ActiveMdiChild. The only way to refer to the text box is through a variable of the frmMdiChild type, so the active child form must be converted to the frmMdiChild type first. Assume a form variable, frmCurrent has been declared to be of the frmMdiChild type. The following statement should achieve this:

```
    frmCurrent = frmNotepad.ActiveMdiChild
```

In search for a specified text string, the dialog box should highlight the matched text in txtDocument. A text string is highlighted when it is selected by the Select method of the textbox. However, when the textbox's HideSelection property is set to True, the highlight will show only when the textbox has the focus. This will not be the case with the presence of a dialog box because it always gets the focus, taking the focus away from all other forms. For this reason, the txtDocument's HideSelection property must be set to False in order for the selected text to remain highlighted. The following code takes care of the preparation for highlighting selected text in the textbox of the active document.

```
Dim frmCurrent As frmMdiChild
Dim HideState As Boolean
Private Sub frmFind_Load(ByVal sender As Object, ByVal e As System.EventArgs)
  Handles Me.Load
    'Keep a reference of the active child form
    'Note that frmCurrent can refer to txtDocument but the ActiveMdiChild can't
    frmCurrent = frmNotepad.ActiveMdiChild
    'Keep the state of HideSelection
    HideState = frmCurrent.txtDocument.HideSelection
    'To make sure that selection will be shown
    frmCurrent.txtDocument.HideSelection = False
End Sub
```

Inside the Load event procedure, a variable named frmCurrent is declared to be of the frmMdiChild type, which is then assigned the active MDI child form. This conversion allows the reference to the child form's text box by qualifying the control's name with the form variable (frmCurrent.txtDocument). Note that the active child form is qualified with the form name, frmNotepad.

Coding for the Find Next Operation

The Find Next event procedure should take care of the following situations:

- It should check to ensure that there is search text in the text box. If there is no search text, the procedure should display the message "Please enter search text" and then exit.
- It should also check to see whether the current search is a continuation of a previous search. This can be determined by comparing the current search text with the previous one. If they are the same, it is a continuation and the search should start from the previously highlighted position plus the length of the search text; otherwise, it is a new search and should start from position one.

The actual search should use the start position just determined. If no text in the document is found to match the search text, the message “Text not found” should be displayed; otherwise, the text found in the document should be highlighted. As explained in Chapter 3, “Visual Basic Controls and Events,” highlighting can be accomplished by the Select method.

The previous search text and the highlighted position should be preserved for the next round. The previous text is kept in a variable named PreviousSearchWord, while the previous position is kept in a variable named Pos. Both are declared at the class level.

The code to handle the FindNext Button Click event appears as follows:

```
Dim PreviousSearchWord As String
Dim Pos As Integer

Private Sub btnFindNext_Click(ByVal sender As System.Object, ByVal e As
  System.EventArgs) Handles btnFindNext.Click
    Dim SearchWord As String
    SearchWord = txtSearch.Text
    If Len(Trim(SearchWord)) = 0 Then
        MsgBox("Please enter search text", MsgBoxStyle.Information)
        Exit Sub
    End If
    If SearchWord = PreviousSearchWord Then
        'Same search word, continue from previous pos
    Else
        'New search word, start from the beginning
        Pos = 0
    End If
    With frmCurrent.txtDocument
        Pos = InStr(Pos + 1, .Text, SearchWord, CompareMethod.Text)
        If Pos > 0 Then
            'text found, highlight
            .Select(Pos - 1, Len(SearchWord))
        Else
            MsgBox("Text not found")
        End If
        'Keep search word for future comparison
        PreviousSearchWord = SearchWord
    End With
End Sub
```

Pay particular attention to the reference to the active child form. It is referenced in this procedure as frmCurrent, which was declared in the general declaration area and assigned properly in the Form Load event presented previously.

Preserving Previous Search Text

Because frmFind is invoked with the ShowDialog method, when it is closed and invoked later, all data values (including the controls) in the form are preserved; that is, values for PreviousSearchWord, Pos as well as the text in the txtDocument textbox should still exist when the form is invoked again after being closed. There is no special issue in this respect.

Creating MDI Applications: A Recap

The MDI Notepad application involves many programming aspects. The following table summarizes the essentials to the creation of an MDI application.

Objective	What to Do
To create an MDI parent	Set the form's IsMdiContainer Property to True.
To make a form a child window of the MDI application	Set the form's MDIParent property to the Mdi form; e.g., `frmMdiChild.MdiParent = frmMdiParent`
To merge menus of the parent and child forms	Set AllowMerge Property of menu strips of all forms to True (default) Set MergeAction of each menu item to either Append or Insert Set MergeIndex to the position you want the item to appear
To make a menu item show the list of child documents	Set the MdiWindowListItem property of the menu strip to the menu item in which to show the list
To create a new document (child window)	Declare a form variable of the child form type with a New keyword; e.g., `MyForm = New frmTextDoc` Set the MdiParent property of the child form to the Mdi parent form; e.g., `frmChild.MdiParent = Me`
To open an existing document	Create a new document. Open the existing file Read the file contents into the textbox in the child form

To reference a control in the active child form	Convert the active child form to the same form type; e.g.,
	`Dim frmDoc As frmMdiChild`
	`frmDoc=MdiNotepad.ActiveMdiChild`
	Qualify the control with the form variable name; e.g., `frmDoc.txtDoc`

To cascade or tile child windows	Use the MDI form's LayoutMdi method; e.g., `mdiNotePad.LayoutMdi(MdiLayout.Cascade)`

Additional Remarks

As you start to review this notepad application, you may realize that MDI is an interface style choice. If the user needs to refer to more than one document in an application that handles only a single document, such as the MS Notepad, the user can load another copy of the application. If your application often involves multiple documents, however, it will be much easier for the user to work with an MDI interface. Your interface style choice should be based on this consideration.

Admittedly, the project is highly simplified. The main purpose is to illustrate the key issues particularly pertaining to an MDI project. Hopefully, by working with the example, you have gained a good understanding of the use and design of MDI applications.

You can, of course, add several enhancements to the program. For example, you can do the following:

- Add the find-and-replace capability to the custom dialog box.
- Include an Edit menu. Note, though, that the Edit context menu is automatically supported by VB for the text box.

- Show the saved document names (filenames) at the bottom of the File menu.
- Replace the text box with the rich text box, which supports varying fonts and other capabilities.

All of these refinements are left to you.

11.5 Designing a Large Project

As the size of a project grows, various issues that do not seem obvious in a small project will start to surface. In some cases, efforts to patch the code can induce even more problems. Most of these problems relate to duplicate code, entangled code, and the hidden bugs that appear to be too many to ignore but too undetectable to catch. In developing a large project, it pays to put in a lot of effort upfront by carefully analyzing the requirements and designing the framework before any single line of code is started. The following are a few pointers worth careful consideration.

Modular Design

With the huge size of code, it is easy to get into a situation in which code in different procedures are entangled; that is, code in two or more procedures or modules may make reference to each other directly. The logic for such code structure becomes difficult to trace. Such code structure can cause great difficulty for maintenance. A minor change in the program requirements can call for a huge amount of maintenance effort.

For example, imagine a form named frmEmployee with a text box to enter the employee number. A button is provided to invoke a custom dialog box named frmSearchEmpNum, which allows the user to search for the employee number given a name. The dialog box is invoked in a typical manner:

```
Private Sub btnEmpSearch_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs)Handles btnEmpSearch.Click
    frmSearchEmpNum.ShowDialog()
End Sub
```

The Form Closing event procedure in frmSearchEmpNum, however, appears as follows:

```
Private Sub frmSearchEmpNum_FormClosing(ByVal sender As System.Object,
  ByVal e As System.ComponentModel.CancelEventArgs) Handles MyBase.FormClosing
    frmEmployee.txtEmpNum.Text = txtEmpNum.Text
End Sub
```

Notice that this event procedure places the employee number in the target text box before the custom dialog box closes. Such a design can cause several problems. If the target text box, for whatever reason, has to change its name or to be moved to another form, the code in the dialog box will have to be changed. Also, the dialog box is extremely specialized. It can provide service to only one control. If another text box in the same project needs the same service, either another dialog box with nearly the same code will have to be written or the code in the dialog box must be modified to identify the new target text box, making the code unnecessarily complex. Its maintainability and general applicability become an issue.

When developing code for large projects, be particularly careful in the design phase to ensure that code in each procedure is independent of the code in all other procedures. As implied in the previous discussion, to accomplish this, you should observe these rules:

- Avoid referencing objects in other forms or modules directly.
- Avoid declaring and sharing variables of a broader scope than necessary. In other words, declare and use variables with a scope as narrow as possible.

Procedures designed this way can then become the building block of a large project. Such a design is recognized as modular design.

Following this design approach, several alternatives to the preceding problem example can be considered:

- The dialog box places the result (employee number) in a Public variable either in the form module or in a standard module. The calling procedure can then obtain the result from the Public variable.
- The dialog box provides a Public function procedure, which returns the employee number being searched for. The calling procedure calls this function in the custom dialog box (instead of invoking the dialog box itself) and then assigns the result in the target control. In this way, any procedure in any form can use the same function in the dialog box.

In either of these design alternatives, any change to the name of the control that needs the service in the calling procedure will require no change in the code in the dialog box. In addition, any other text box requiring the same service can conveniently call the same dialog box.

Factoring to Minimize Code

Another problem in large projects is the potential of duplicate code. Several forms can require the same computations and perform similar activities. Even in the same form, several procedures can require similar code to perform certain activities. Copying similar code to handle the repetitive needs can not only be boring and time-consuming, but can also be error prone. In addition, the sheer number of lines of code can create problems for maintenance. Each time a revision of the repeated code is called for, the same correction will have to be done many times, increasing yet another possibility of introducing more errors.

One way to alleviate this problem is to analyze the program requirements carefully and code the required activities as Public general procedures in a standard module. Any event that needs to handle the activities can then simply call that procedure. An important advantage of this factoring approach is that if there is any error or required revision of the code, you need to look at only one place for correction. All procedures that use the function will automatically be corrected. The effort required for the revision is minimized, with much less chance to introduce additional errors.

Layered Standard Modules

The concept of modular design and code reusability can be carried out further by using multiple standard modules in a project. Some standard modules can be used to contain code and data that are particularly related to the current project, whereas others can be used to contain procedures that are useful to many projects. Procedures in each type of standard modules can be further classified by their commonality. For example, procedures that deal with files can be placed in one module. Modules of this nature can be added to any project that needs their capabilities.

Object-Oriented Programming

An additional approach to handling a large project is the use of the object-oriented programming technology. An object is a unit of code with data. It presents itself as a single unit (object or entity). A programmer using the object does not need to know anything about its inner working other than its interfaces: properties, methods, and events. A large project that involves a team can be divided into

several subprojects (units). These units can then be coded as objects. The project can be assembled by putting the objects together, allowing all to interact with each other through their defined interfaces. Topics related to developing templates for objects are discussed in detail in Chapters 12 and 13.

Summary

- A typical large project can involve many forms and modules and its main form employs menus to invoke other forms.
- The menu strip allows the programmer to easily create a menu bar on a form. You can conveniently enter the text for a menu item in a rectangle that the menu strip displays and set the item's name property in its property window.
- The context menu strip allows the programmer to create a context menu that pops up when the user right-clicks on a control. The context menu can be set up in a way similar to the menu strip. Each context menu can be associated with any number of controls, but a control can have only one context menu. To associate a control with a context menu, set the control's Context Menu property in its properties window.
- The user can access a menu item by clicking on the item or the use of an access key or shortcut key. You can set up an access key by placing an &before a letter in the item's text. The user can then access the menu item by pressing Alt+<letter>. You can set up a shortcut key for a menu item by selecting a key in its Shortcut property in the properties window. The user can access the menu item by pressing the designated shortcut key.
- When a menu item is clicked, or accessed through an access or shortcut key, its Click event is triggered.
- In a multiple form application, you can choose a form as the startup object.
- To invoke a modeless form, use the form Show method.
- A form can also be invoked with the ShowDialog method. When invoked this way, the form stays on top of the application and is recognized as a modal form. Typically, a form invoked this way is used as a custom dialog box.
- A form can be made to disappear by its Close or Hide method. A closed modeless form no longer exists in memory; a hidden form stays in the memory, and can be shown again with the Show method.
- Controls and variables with Public or Friend scope in a form are accessible to other modules; however, the form must be still in memory—either appearing on the screen or being hidden with the Hide method. If you are not sure whether the form will still exist at time of access, assign the data to Public or Friend variables in a standard module. Other modules can then obtain the data by referencing the variables in the standard module.
- You can declare the same Public (or Friend) variable names in different standard modules. In that case, to reference the variables, qualify their names with the module name.
- When the startup form closes, all other forms are also closed without triggering their respective FormClosing event. In general, this is the event that you place code to perform finishing touches for each form, such as saving unsaved data. To ensure that the FormClosing event is triggered when the startup form closes, place the code to close all open forms in the application (Application.OpenForms) in the startup form's FormClosing event.

- The number guessing game example is used to illustrate how a multiple form project can be developed.
- If an application can have many forms (windows) of the same nature appearing in a container form at one time, the application is a multiple document interface (MDI) application. The MDI application is technically different from the single document interface (SDI) application in that the former has a container form with one or more child forms, while the SDI does not have the parent-child relationship between forms.
- To create an MDI application, set the container form's IsMdiContainer property to True. Also by code, set the child form's MdiParent property to the container form.
- To reference the active MDI child form, use the syntax:

```
ParentForm.ActiveMdiChild
```

- To cascade or tile the child windows in the MDI container, use the container form's LayoutMdi method.
- If you set a form's AcceptButton property to a button, the button's Click event will be triggered when the user presses the Enter key. If you set a form's CancelButton property to a button, the button's Click event will be triggered when the user presses the Esc key.
- The MdiNotepad project is used to illustrate how to design and code an MDI project.

Explore and Discover

11-1. **Sequence of Mouse Click Events.** Four events can be associated with a mouse click: Click, MouseDown, MouseUp, and MouseMove; however, which event precedes which? Use the Console WriteLine method in each of these events for a form to find out; for example, place the following code in the Form Click event:

```
Debug.WriteLine ("Form click event is raised.")
```

Code similarly for the other event procedures.

Run the project and then click the form. List the events in sequence of their occurrence.

11-2. **Who Gets the Mouse Click Events?** Add a text box on the form in the preceding exercise. Add the three mouse event procedures for the text box, and paraphrase the Debug.WriteLine statement; for example, "Text box Click event is raised." Try the following actions and observe the messages in the immediate window:

- Press and then release the mouse button in the form.
- Press and then release the mouse button in the text box.
- Press the mouse button in the form, and release it in the text box.
- Press the mouse button in the text box, and release it in the form.

11-3. **Mouse Movement Events.** When the user moves the mouse on and around an object, several mouse events can be triggered: MouseMove, MouseEnter, MouseHover, and MouseLeave. In each of these events for the form, enter some code to identify the event; for example enter the following code in the form MouseMove event:

```
Debug.WriteLine("Form MouseMove")
```

Try the following actions while pressing the left or right button and not pressing any button; then observe the messages in the immediate window:

- Move the mouse on the form.

- Move the mouse away from the form.
- Move the mouse back onto the form, and gradually leave the form again.

11-4. **Are Menus Included?** In a new form, set up a File menu with Open, Close, and Exit as its options. Draw a text box, a radio button, and check box on the form. Also, draw a button. Name it **btnShowControls,** and set its Text property to **Show Control Names**. Enter the following sub procedure, and call it from the button's Click event with the statement ListControls(Me).

```
Sub ListControls(ByVal TheControl As Control)
    Dim I As Integer
    For I = 0 To TheControl.Controls.Count - 1
        Debug.WriteLine(TheControl.Controls(I).Name)
        If TheControl.Controls(I).Controls.Count > 0 Then
            ListControls(TheControl.Controls(I))
        End If
    Next I
End Sub
```

Run the project and then click the button. What do you see? Are the menus listed? The menu strip is included but not those items within it.

11-5. **Resizable and Nonresizable Forms.** The form has a FormBorderStyle property that may interest you. Run a project with a new form. Can you resize the form by dragging an edge from one of its sides? End the program.

Set the form's FormBorderStyle property to FixedSingle, and run the project again. This time, can you resize the form?

The form's FormBorderStyle has several other settings. Some affect the title bar, while some affect the appearance of the outer edge of the form. Try each setting, and run the project to see the effect.

11-6. **Is the Minimized Form Idle or Active?** Does a minimized form continue to work? To test, enter the following code in the form Resize event:

```
    Dim ComeBackTime As Date
    If Me.WindowState = FormWindowState.Minimized Then
        ComeBackTime = DateAdd(DateInterval.Second, 5, Now)
        Do Until Now > ComeBackTime
            Application.DoEvents()
        Loop
        Me.WindowState = FormWindowState.Normal
     End If
```

Run the project. Click the Minimize button on the form's title bar. The form should come back in five seconds. A minimized form is not really idle, is it?

11-7. **Ending a Project.** How does the Close method differ from the End statement in terminating a project? To test, draw two buttons on the form. Name them **btnClose** and **btnEnd**. Set their Text properties to **Close** and **End**, respectively. Enter the following code:

Code	Event Procedure
`End`	btnEnd_Click
`Me.Close`	btnClose_Click
`MsgBox("Closing the form")`	Form1_FormClosing

Run the project, and click the End button. Do you see any message? The program terminates abruptly. Run the project again, and click the Close button. Do you see any message? Run the project again. This time, click the Close (X) button instead of the Close button. Do you see any message?

You can place code in the form's FormClosing or FormClosed event to take care of unfinished business for the form before it is closed; however, the code will be triggered only if your code uses the form's Close method—not the End statement—to quit.

11-8. **How Many Forms Can You Create?** Add a new form to your new project, and name the second form **Form2Template**. In the startup form, add a button; name it **btnShow**, and set its Text property to **Show**. In the same form, enter the following code:

Code	Code Placement
`Dim Form2 As New Form2Template`	(General declaration area)
`Form2.Show`	BtnShow_Click

Run the project. Click the Show button several times without closing the second form. How many of the second form do you see? You should see only one. The Show method does not create a new form object.

Now change the code in the startup form as follows:

Code	Code Placement
`Dim Form2 As Form2Template`	(General declaration area)
`Form2 = New Form2Template`	BtnShow_Click
`Form2.Show`	

Run the project, and click the Show button on the startup form several times. How many forms do you see? You should see as many as the number of times you click. The New keyword creates (initializes) a new form. Now change the code in the startup form as follows:

Code	Code Placement
(None)	(General declaration area)
`Dim Form2 As Form2Template`	BtnShow_Click
`Form2 = New Form2Template()`	
`Form2.Show`	

Run the project, and click the Show button on the startup form several times. How many forms do you see? You should see as many as the number of times you click. The New keyword creates a new form.

So, what's the difference between this way and the preceding way of declaration? In the last way of declaration, the Form2 variable is not recognizable by any other procedure, while in the preceding one, Form2 is recognizable by other procedures. Note, however, it is associated only with the most recently created form. Set up another procedure in the startup form to verify.

11-9. **The Activated Event.** The Activated event can be triggered unexpectedly. Add a new form to a new project, and name the second form **Form2Template**. In the startup form, add a button, name it **btnShow**, and set its Text property to **Show**. Enter the following code:

Code	Event Procedure
`Form2.Show`	BtnShow_Click
`Debug.WriteLine("Startup form activated")`	Form1_Activated

In the second form, add the following code in the form's Activated event:

```
    Debug.WriteLine("Form2 activated")
```

Run the project, and click the Show button on the startup form. Look into the immediate window. You should see activated messages from both forms. Now close the second form by clicking its Close button in the control box. Do you see another activated message from the startup form? Be aware of this situation to avoid unexpected execution of the code in that event.

11-10. **Are the Child Forms Really Closed Gracefully?** You know that in an SDI application, when the startup form is closed, all the other forms end abruptly; however, the text claims that the child forms will close gracefully with each form's FormClosing event triggered when the MDI (parent) form closes. Design a project to verify.

11-11. **Start Position for the Form.** The form has a StartPosition property that allows five different settings. Run the project, and observe the form position after selecting a setting. Try all five settings. This property provides a neat way to set the form start position without any single line of code.

Exercises

11-12. **Creating Menus for a General Ledger Application.** In a new project, create a menu system that has the following:

- On the menu bar, there are two items: File and Maintenance with both items' first letter underlined.
- In the File menu, there are four options: New, Open, Close, and Exit, with the N, O, C, and x underlined in each respective item. There should be a separator between Exit and Close.
- In the Maintenance menu, there are three options: Account Classification, Account Entry/Edit, and Statement Format. The letters underlined for the respective items are C, A, and S.

Add the code so that the program will end when the user clicks the Exit option at run time.

11-13. **Creating a Menu for a Clinic.** In a new project, create a menu system that has the following:

- On the menu bar, there are three items: Code Maintenance, Service, and Account/Patient. The first letter of each item is underlined.
- In the Code Maintenance menu, there are three items: Service Code, Diagnosis Code, and Billing Adjustment Code. The first letter of each item should be underlined.
- In the service menu, there are two items: Service and Reports (first letter of each is underlined). There is a separator between the two items. The reports item has three options (third-level menu items): Daily Service by Patient Report (S in Service is underlined; shortcut key is Ctrl+S), Daily Service by Service Code Report (C in code is underlined), and Daily Patient Payment Report (P in payment is underlined; shortcut key is Ctrl+P).
- In the Account/Patient menu, there are two items: Account Entry/Edit and Patient Entry/Edit. The first letter of each item is underlined.

11-14. **Creating a Pop-Up Menu for a Clinic.** (continued from 11-13) Add a context menu strip to the clinic project. Name the menu mnuOther, which should have three options: Doctor, Hospital, and Lab. The menu should pop up when the form is a right-clicked.

11-15. **An Application with Two Forms**. Refer to exercise 14. Modify the project so that it does the following:

- As soon as the project starts, a login form appears to solicit a password from the user. The text that the user enters into the password text box should be masked with "*." The password should be case sensitive.
- If the password entered by the user is not "Master Key," a "password incorrect" error message will be displayed. Also, if the user fails three times, the program will terminate.
- If the password entered is correct, this login form is closed. Another form (the form designed for exercise 11-14) is displayed until the user decides to quit.

11-16. **GL Application with Two Forms. Refer to 11-12**. Add a new form for Account Classification maintenance, which will be triggered when the user clicks the Account Classification option of the Maintenance menu in the main form. The new form will be used to maintain account types (which should remain fairly static) of the GL system. The DB table (named AccountClassification) has the following data fields:

Field	Data Type	Remark
Classification Code	Integer	Primary key; 1 thru 5
Account Type	Text (255)	1=Asset, 2=Liability; etc…

Add the necessary visual design and code for the user to conveniently enter new account classifications as well as modify existing ones.

11-17. **GL Application with Three Forms. Refer to 11-16.** Add a new form for Account Maintenance, which will be triggered when the user clicks the Account Entry/Edit option of the Maintenance menu in the main form. The new form will be used to maintain general ledger accounts of the GL system. The DB table (named Account) has the following data fields:

Field	Data Type	Remark
Account Number	Long Integer	Primary key; 5 digits
Account Name	Text(255)	
Account Type	Integer	Account classification
Contra Account	Yes/no	True=contra account
Affects cash flow	Yes/no	

Add the necessary visual design and code for the user to conveniently enter new accounts as well as modify existing accounts.

11-18. **To Save Only When There Is a Change**. Refer to the MDI example in section 11.4. In the Closing event of the child form, the program prompts the user whether to save the document indiscriminately. Design a scheme so that the program will prompt the user only when some changes have been made to the document but not yet saved. (Hint: Use a class level Boolean variable to track changes in the TextChanged event, set it to False in the routine that saves the document, and test this variable is true in the FormClosing event.)

11-19. **The Find and Replace Dialog Box.** (Refer to exercise 11-18.) The MDI project has a Find dialog box. Suppose the Find menu in the child form is revised to include Find and Find & Replace as its menu options. Add the Find and Replace capability to the dialog box.

11-20. **A Generalized Find Dialog Box: Design for Code Reusability.** The Find dialog box in the MDI example makes specific reference to the form and its text box. Such a design makes the dialog box difficult to use by other forms or projects. One way to avoid referring to the calling form is not to call the dialog box directly. The mnuFind Click event procedure in the MDI form will instead call a Sub (call it FindText) in the dialog box form, and pass the text box (the object, not just the Text property) to be searched as a parameter. Its header should look similar to the following:

```
Public Sub FindText(TextBoxFromCaller As TextBox)
```

This Sub then sets this text box to yet another class-level text box variable (call it WorkingTextBox) so that other procedures in this form can refer to it. In addition, the FindText Sub should invoke the dialog box as a modal form. That is:

```
    WorkingTextBox = TextBoxFromCaller
    Me.ShowDialog()
```

where WorkingTextBox is declared at the form level as follows:

```
Dim WorkingTextBox As TextBox
```

In this way, WorkingTextBox is actually referring to the text box in the form that calls the FindText Sub. Any searching or highlighting performed on the variable, WorkingTextBox is actually performed on the text box in the calling form. The btnFindNext event procedure can access this variable because it is declared at the class level. This scheme has the advantage that neither the FindText Sub nor the btnFindNext event procedure in the dialog box makes any direct reference to the MDI child form or the text box therein.

Notice that the FindText Sub does not really perform any Find Text, but sets the text box to the working text box variable and invokes the form's ShowDialog method; however, this Sub is what another developer sees to provide the search text service. Because the dialog box form is named frmSearch, and the text box to search on (in the calling form) is named txtDocument, this Sub can be invoked with the following statement:

```
    frmSearch.FindText(txtDocument)
```

Modify the code for the text search functionality in the MDI project by implementing this design. (Note: an exercise in Chapter 12 provides yet another possible solution.)

Projects

11-21. **A Form with a Custom Dialog Box (Sales Application).** (Refer to exercise 9-8 in Chapter 9, "Database and ADO.NET," for the product table of the inventory database) Create another table, Sales, with four columns: OrderID, Date, ProductID, and Quantity Sold; then design a project for the sales order entry application. The form should have all the four fields (use a masked edit box for Date and text boxes for all other fields) and three buttons (Save, Search, and Quit).

When the Save button is clicked, data will be saved. The Search button is placed by the ProductID. When it is clicked, a custom dialog box will appear to help the user find the ProductID given a product name. When the Quit button is clicked, the program quits.

The dialog box has a text box for the user to enter a portion of the product name. When the user clicks the Search button by the text box, all products matching the name will appear in a grid (use the data grid), which has two columns: Product Name and Product ID. When the user clicks a row in the grid, the corresponding data will show in the two labels above the grid. When the user clicks the OK button, this dialog box disappears.

The data-entry form should incorporate all data validation routines necessary to ensure data validity before the record is saved into the Sales table.

11-22. **An Application with a Menu Form and Two Additional Forms** (continued from Project 11-21). Modify project 11-21 in the following manner:

- The project will start with a menu bar that appears as follows:

File	**Maintenance**	**Transaction**
Open	Product	Sales
Exit	Customer	Purchase

- The Open option of the File menu will invoke an open dialog box for the user to specify the location of the Inventory database. The Exit option will end the application.
- The Product option of the Maintenance menu will invoke the form you created for exercise 9-8.
- The Sales option of the Transaction menu will invoke the form you created for exercise 11-21.
- When the Customer or Purchase option is clicked, display the message “Sorry! But the application is still under development.”