Visual Basic 2008 Programming

Business Applications with a Design Perspective

Jeffrey J. Tsay

Table of Contents

Chapter 6

Input, Output, and Procedures

This chapter introduces two topics. Introduction to input and output provides introductory topics to input and output features available in VB, which are essential to your understanding of input and output concepts. In addition, some of these features provide a convenient means for you to design sound interactions between your program and the user. Also, the section on procedures discusses the uses of general procedures in structuring VB code. The advantages of using general procedures are too significant to overemphasize, and are outlined at the end of that section after you gain a good working knowledge of general procedures. This chapter concludes with an example that represents an attempt to show how all the pieces you learn from this chapter can be put together. After completing this chapter, you should be able to:

- Use the MsgBox function to ascertain the user's response when your program needs a direction from the user,
- Develop code to handle text files for input and output,
- Use file dialog boxes to prompt for file paths,
- Articulate the benefits of creating and using general procedures,
- Write and use general procedures,
- Differentiate between the situations in which a sub and a function should be written, and
- Determine when a parameter is needed for a general procedure.

6.1 Introduction to Input and Output

When your program starts, the variables in the program must be initialized first either with constants or with values provided from external sources, such as users' actions (keystrokes or mouse clicks) or an existing file. The process of obtaining data from a source external to the central processing unit (CPU) is referred to as *input***.**

The data that the variables contain are stored in memory. As soon as your program terminates, all these data will no longer exist in the CPU. Some of the data are temporary in nature and can be discarded without any problem. Others, however, need to be displayed (for the user to view their values), printed on a hard copy, or saved in some intermediate storage device, such as hard disks for future uses. The process of sending the data in the CPU to an external device is referred to as *output*. This section introduces a few possible means that you can use in VB for input and output.

VB provides various ways that your program can obtain input or produce output. Recall in Chapter 3, "Some Visual Basic Controls and Events," you learned that at least six VB controls could be used to obtain input from the user. A text box, for example, can be used for the user to enter data. Indeed, some of those controls can also be used to display output. For example, you can display any data in the same text box. This section explores three additional ways for input and output:

- The MsgBox function
- Files
- File dialog boxes

The MsgBox Function

You have seen MsgBox used to display a pop-up message. When used to display a message (as output), the MsgBox function has the following syntax:

```
    MsgBox(Prompt[, Buttons] [, Title])
```

where *Prompt* = a text string that is displayed on the message box.

Buttons = a message box style enumeration value specifying the button(s) and icon the pop-up message box is to display; when omitted, an OK button is displayed. To specify the enumeration, code `MsgBoxStyle` followed by a dot and the enumeration name, such as `MsgBoxStyle.OkOnly`.

Title = an optional text string to be displayed as the title of the pop-up message box; when omitted, the project name is displayed.

The buttons parameter specifies what button(s) and icon the message box is to display. Some of the sample values are listed in the following table:

MsgBoxStyle Constant	Display	Type
OKOnly	OK only	Button(s)
YesNoCancel	Yes, No, and Cancel	
YesNo	Yes and No	
RetryCancel	Retry and Cancel	
Critical	Critical Message	Icon
Question	Warning Query	
Information	Information Message	

You can specify one group of buttons and one icon to display at the same time. To display a group of buttons and an icon together, insert an "Or" logical operator between the two values as the second parameter. If nothing is specified, the OK default button will be displayed. Two examples of the uses of MsgBox follow.

Simple Message

Suppose you want to display a message in your payroll program informing the user that the maximum allowable number of hours worked per week is 60; you can code the following:

```
MsgBox("Maximum allowable hours worked per week is 60.", _
  MsgBoxStyle.Information, "Payroll Entry")
```

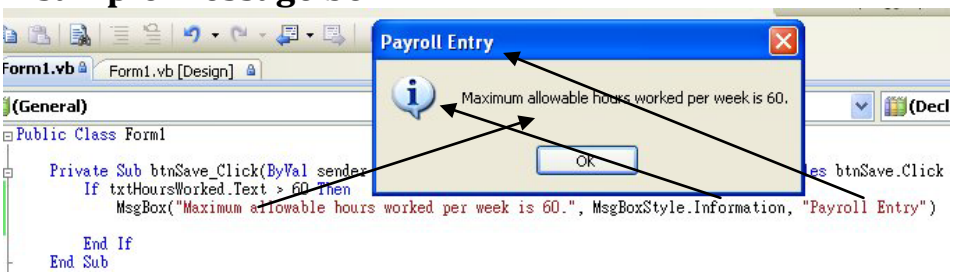
The code will display in the message box not only the message, but also an Information icon. It will also show "Payroll Entry" as its title. Because nothing is said about the buttons to display, the default OK button will be displayed (see Figure 6-1).

Notice again that you can specify an icon along with a group of buttons to display in the message box by adding the icon and buttons values together for the second parameter. For example, you can make it display Yes and No buttons instead of the default OK button as well as the Information icon by coding the following:

```
MsgBox("Maximum allowed hours worked per week is 60.", _
  MsgBoxStyle.Information Or MsgBoxStyle.YesNo, "Payroll Entry")
```

The Yes and No buttons, however, do not seem to make any sense in this message. They are usually used when the message box is used to prompt for the user response rather than to display a straight message.

Figure 6-1
A sample message box



This figure shows where each parameter in MsgBox appears in the message box. Notice that the information icon is displayed because of the second parameter.

Tip

If you set Option Strict Off, you can code the following

```
MsgBoxStyle.Information + MsgBoxStyle.YesNo
```

in place of MsgBoxStyle.Information Or MsgBoxStyle.YesNo in the preceding code; however, technically, the Or operator is the correct operator to use. The values in the MsgBoxStyle enumeration are actually flags that represent the on and off states of different bits. Refer to Appendix C, "Number Systems and Bit-wise Operations," for flags and bit-wise operations.

Using MsgBox to Obtain the User Response

The MsgBox function can be used to obtain the response from the user. When used for this purpose, its has the following syntax:

```
Response = MsgBox(prompt[, buttons] [, title])
```

Depending on the button the user clicks on the message box, the MsgBox returns a different value. The possible returned values are enumerated as MsgBoxResult-type values with the same names as the buttons. The following table is a sample of the possible responses:

Button Clicked	MsgBoxResult Constant
Cancel	Cancel
Abort	Abort
Yes	Yes
No	No

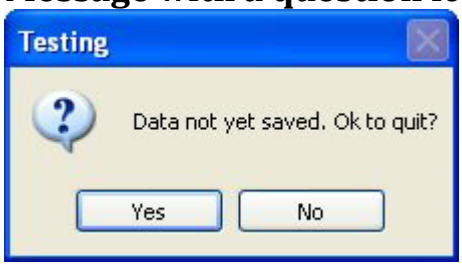
Suppose your form has a text box named txtName, which is cleared when data on the form are saved. When the user clicks the Close (X) button on the title bar, if the control contains data, it would mean that data have not yet been saved. You would like to warn and then ask the user if it is OK to quit. One solution is to place the code in the form's Closing event to handle this situation. The *Form Closing event* occurs

immediately before the form is being closed. It occurs when the user clicks the form's Close button, or when the form's Close method is being executed. The event has a Cancel argument—e.Cancel—that when set to True will cancel the Close operation.
Using this event procedure, the code can appear as follows:

```
Private Sub Form1_Closing(ByVal sender As System.Object, ByVal e As
  System.ComponentModel.CancelEventArgs) Handles MyBase.Closing
    Dim Response As Integer
    If Len(Trim(txtName.Text)) > 0 Then
        'The text box contains data.
        'Ask the user whether he or she really means to quit.
        Response = MsgBox("Data not yet saved. Ok to quit?", _
          MsgBoxStyle.Question Or MsgBoxStyle.YesNo, "Testing")
            'Display the "?" icon as well as yes and no buttons
        If Response = MsgBoxResult.Yes Then
            ' User clicked the Yes button;
            ' Do nothing (allow program to proceed to close)
        Else
            ' User clicked the No button, cancel the close operation
            e.Cancel = True
        End If
    End If
End Sub
```

In this procedure, the first If statement checks if the txtName text box contains any nonblank text. If so, the message box displays the warning message with the question mark icon and the Yes and No buttons (see Figure 6-2). Note that the buttons parameter specifies to display the question icon as well as the Yes and No buttons. If the user clicks the Yes button, a value MsgBoxResult.Yes will be returned. (Notice that this enumeration is different form MsgBoxStyle, which is used for displaying message and not for comparing results.) The program will proceed normally (close); otherwise, True is assigned to the Cancel parameter and the closing process will be canceled.

Figure 6-2
Message with a question icon and two buttons



Specifying the second parameter in MsgBox as MsgBoxStyle.Question Or MsgBoxStyle.YesNo will make the message box display both the question mark icon and the Yes and No buttons.

The MsgBox function provides a facility for simple dialogs between your program and the user. It is most appropriate in the following situations:

- When your program needs to convey a simple message to the user. Examples include messages that do the following:
    - Indicate that a file has been saved
    - Inform the user that an entry is not acceptable
    - Instruct the user to take a certain action, such as inserting a disk in a drive

- When your program needs a direction (among a few available choices) from the user. Examples include the following:
    - Inform the user of the failure of an operation, such as unable to read a specified file and prompt whether to try again, skip the operation, or quit
    - Warn the user of the possibility of losing data, such as when instructed to delete a file and prompt whether to proceed the operation

Look It Up

Search the keyword MsgBox Function (Visual Basic language reference) in the Index tab of the Help file. It explains in more details of all the parameters. It also provides a complete list of button values you can specify for the box to display as well as the values indicating the button clicked.

Files

VB provides different ways to handle files. In this section, you will explore the use of two objects that can be used to read and write text files. These two objects are available under the System.IO namespace, which is explained later in this chapter.
When you work with files using objects in System.IO, several key steps are involved:

- Declare an object variable to indicate the object type by which your code is to perform input or output. The two objects that can be used to read or write text files are StreamReader and StreamWriter.
- Create the object. In this step, your program associates the object in your program with the physical file residing in the intermediate storage device, such as the hard drive or floppy disk. Traditionally, this step is recognized as Open the File. A file that is associated with the StreamReader is said to be opened for input, while a file that is associated with the StreamWriter is said to be opened for output.
- Use the object's proper method to perform file operations, such as reading/writing data from/onto the file. Most of the file activities of your program relate to this aspect.
- Close the object. This process dissociates your program from the physical file, and ensures that all I/O operations are completed by the system.

Declaring the Object Variable

If you want to read data from an existing file, you will use the StreamReader. If you want to write data onto a file, you use the StreamWriter. The syntax appears as follows:

```
Dim InputFileName As System.IO.StreamReader
Dim OutputFileName as System.IO.StreamWriter
```

where *InputFileName* and *OutputFileName* can be any legitimate VB identifiers.

The declaration indicates that the variable is of the object type, System.IO.StreamReader (System.IO.StreamWriter). Assuming that you want to read a file containing names and phone numbers, you can declare a phone file as follows:

```
Dim PhoneFileI As System.IO.StreamReader
```

As discussed in Chapter 4, "Data, Operations, and Built-in Functions," be aware that this declaration simply indicates that the variable, PhoneFileI is of the StreamReader type but a StreamReader object has not yet been created.

Creating the Object

In this step, you create the object and associate it with a physical file in your computer systems. For example, assume the phone file is located at C:\Temp\PhoneFile.txt. You can use the following code to create the object (open the file).

```
    PhoneFileI = New System.IO.StreamReader("C:\Temp\PhoneFile.txt")
```

The New keyword creates a new StreamReader object and associates the object with the physical file specified in the string (opens the file). The resulting object is assigned to PhoneFileI. From this point on, when you make any reference to PhoneFileI, you are actually referring to the object that deals with the file. Notice that the physical file, such as C:\Temp\PhoneFile.txt, must exist before you can associate it with the StreamReader. If the file does not exist, execution of the preceding statement will result in a run time error.

Combining Declaration and Creation of An Object

In the preceding discussion, you are shown two steps to associate an object variable with an object. As explained in Chapter 4, you can actually combine the two steps in some cases. For example, the following statement will declare the variable PhoneFileI and associate it with the physical file at the same time:

```
Dim PhoneFileI As New System.IO.StreamReader("C:\Temp\PhoneFile.txt")
```

Notice the New keyword in the statement. This keyword works exactly the same way as in the previous assignment statement. Which way is preferable? It depends on how the variable is going to be used. The one-step approach appears to be convenient; however, if the variable is declared in a procedure, the object will not be accessible to another procedure, and will be out of scope as soon as the procedure ends. If it is declared at the class level, the filename must be known at design time. Apparently, its applicability is limited. The two-step approach is more flexible if the object needs to be accessible to more than one procedure.

Reading Data from a File

The StreamReader provides several methods that you can use to read data from the file. The following table shows methods that are of particular interest:

Method	Explanation
Peek	Returns the next character in the stream but does not advance the current position; returns –1 if there is no more data (end of file is reached)
Read	Reads a number of specified characters starting from current position
ReadLine	Reads one line of data starting from current position
ReadToEnd	Reads the remainder of the entire file starting from current position
Close	Dissociates the object from the physical file

Depending on the organization of the existing file and the application, you will find each of these methods useful. The ReadLine method will be most useful when you organize your data on a line by line basis. For example, suppose that you have created the PhoneFile object shown in the preceding code example. The file contains your friends' names and phone numbers and appears as follows:

```
817-222-3838, John Dole
214-555-9999, Jane Smith
```

To read your friends' data one person at a time, the ReadLine method should be most appropriate. As an example, assume your form has a button named btnRead. When you click the button, you want to read one line of data into the variable PhoneAndName and then display the result. Your code can appear as follows:

```
Dim PhoneFileI As System.IO.StreamReader
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
  System.EventArgs) Handles MyBase.Load
    PhoneFileI = New System.IO.StreamReader("C:\Temp\PhoneFile.txt")
End Sub
Private Sub btnRead_Click(ByVal Sender As System.Object, ByVal e As
  System.EventArgs) Handles btnRead.Click
    Dim PhoneAndName As String
    PhoneAndName = PhoneFileI.ReadLine()
    MsgBox(PhoneAndName) 'Show the line
End Sub
```

The first time the procedure is executed, the PhoneFileI object will read the first line of data into the variable PhoneAndName, so the variable will hold the value "817-222-3838, John Dole". The second time, the variable will have the data on the second line.

Of course, if you want to separate the phone from the name, you will need to parse the string. A possible solution is to add the following code:

```
Dim TheName As String
Dim ThePhone as String
Dim P As Integer
P = Instr(PhoneAndName, ",") 'find position of comma
ThePhone = Microsoft.VisualBasic.Left(PhoneAndName, P – 1)
TheName = Mid(PhoneAndName, P + 2) 'There's a space after comma
```

As a technical side note, at the end of each line in a text file, there is an end of line marker that the computer uses to separate one line from the other. The marker consists of two character codes: Return (13) and Line Feed (10). VB provides the named constant vbCrLf for this marker. Recall that you used this constant in Chapter 3, "User Interface Design: Some Visual Basic Controls and Events," to display texts on different lines. Although the marker is not visible to you, it is important to be aware of its existence to understand how the text file is handled internally.

Testing for the End of File Condition

In the preceding example, if you continue to click the Read button and execute the ReadLine method, the file will eventually run out of data. Further attempt to read the file will result in a run time error. Your code should check for the end of file condition before executing the ReadLine method. One way to perform this test is to use the Peek method. As indicated in the preceding table, the Peek method previews the next character in the input stream without advancing the current position. When the end of file is reached, this method returns a value –1. The complete code to test the end of file condition, read and parse the data, and display the result is given as follows:

```
Private Sub btnRead_Click(ByVal Sender As System.Object, ByVal e As
  System.EventArgs) Handles btnRead.Click
    Dim PhoneAndName As String
    Dim TheName As String
    Dim ThePhone As String
    Dim P As Integer
```

```
    If PhoneFileI.Peek() = -1 Then
        'end of file has been reached
        MsgBox("No more data") 'display the eof message
        Exit Sub 'leave the procedure
    End If
    PhoneAndName = PhoneFileI.ReadLine()
    P = InStr(PhoneAndName, ",") 'find position of comma
    ThePhone = Microsoft.VisualBasic.Left(PhoneAndName, P - 1)
    TheName = Mid(PhoneAndName, P + 2) 'There's a space after comma
    MsgBox("The phone is " & ThePhone & ". The name is " & TheName &
      ".")
End Sub
```

Pay particular attention to the If block in the code. The If statement uses the Peek method to test against –1. If it is true, a message indicating no more data is displayed. The next statement, *Exit Sub*, will leave the procedure without executing any of the remaining code in the event procedure. Additional explanation of this statement is given in the next section of this chapter. To test the above procedure, be sure to include the code to declare the PhoneFileI variable and the form Load procedure to create the object to associate with the physical file.

Reading an Entire File

The ReadToEnd method reads from the current position to the end of the file. This method is handy when you need to read the entire file in one operation. All you have to do is to create a StreamReader object and then use this method to read the file. For example, suppose you would like to read the entire file into a text box named txtDoc from "C:\Temp\PhoneFile.txt." You can code the following:

```
Dim TheInputFile As System.IO.StreamReader
TheInputFile = New System.IO.StreamReader("C:\Temp\PhoneFile.txt")
txtDoc.Text = TheInputFile.ReadToEnd()
```

You can even allow the user to edit the text as in the word processing operations. Of course, to display the text properly, you should set the text box's MultiLine property to True and its Scrollbars property to Vertical.

Note that as previously discussed, you can declare and create an object at the same time. The preceding code can be modified as follows:

```
Dim TheInputFile As New System.IO.StreamReader("C:\Temp\PhoneFile.txt")
txtDoc.Text = TheInputFile.ReadToEnd()
```

The Close Method

When you are done with a file, you should dissociate it from the StreamReader object for several reasons. You may need to use the same file with a different mode, such as from the input to output mode. The dissociation eliminates the possibility that your program will accidentally perform unexpected operations on the file. It releases the computer resources associated with that file, and ensures that all file operations you have performed are actually carried out by the computer. The StreamReader's *Close method* dissociates the file from the object. To close TheInputFile in the preceding example, you will code:

```
TheInputFile.Close()
```

Output with Files

To output data onto a text file, you can create a StreamWriter object and then use its Write or WriteLine method to write the text. The syntax to create the StreamWriter appears as follows:

```
StreamWriterObject = New System.IO.StreamWriter(FileName, AppendMode)
```

where StreamWriterObject = a variable that is declared to be of the StreamWriter Type.

FileName = a string giving the file path, and

AppendMode = a Boolean value. Regardless of this value, if the physical file does not exist, a new file is created. On the other hand, if the file already exists and if the Boolean value is True, new data will be written at the end of the existing contents (old data preserved); otherwise, the existing contents will be erase.

The following statements will associate the StreamWriter, PhoneFileO with the Phone file discussed in the previous example. Note that its related StreamReader, PhoneFileI must be closed if you have previously created the association for other purposes.

```
Dim PhoneFileO As System.IO.StreamWriter ‘Place this line in the
general declaration area
‘Place the following line in an event procedure to create a
‘StreamWriter with the append mode
PhoneFileO = New System.IO.StreamWriter(“C:\Temp\PhoneFile.txt”, True)
```

You can then use the WriteLine method to add more phone numbers and names.

The WriteLine Method

The *WriteLine method* will write the text data on the file, and add an “end of line” maker; that is, vbCrLf. The next output operation will start from the next line. This method has the following syntax:

```
    StreamWriterObject.WriteLine(StringToOutput)
```

where *StreamWriterObject* = the object created to write the stream, and *StringToOutput* = a string expression to be written to the file.

For example, you have a masked text box named mskPhone, and a text box named txtFriend in a form. You would like to save the data entered by the user onto the file associated with PhoneFileO. You can use the following statement:

```
    PhoneFileO.WriteLine(mskPhone.Text & “, “ & txtFriend.Text)
```

The line of output data in the file will appear as follows:

```
214-666-8392, Allen Jones
```

Recall that the ReadLine method of the StreamReader explained previously in this chapter reads one line of data at a time. You can think of the ReadLine and WriteLine methods as complementary I/O methods.

Notice a difference between the Write and the WriteLine method. As discussed, the WriteLine method always adds an end of line marker, but the Write method does not; therefore, if you use the Write method to output to a file, the next output operation (either using the Write or WriteLine method) will start at where the previous output ends, not at the beginning of a new line—even if the file is closed and opened again for output operations.

Close the StreamWriter

After you use the StreamWriter to write data to the file, and before your program ends, *be sure you have code to close the StreamWriter*. This is imperative. Without closing the

StreamWriter, the output operations may not be complete, and your data may be lost in the process. In the Explore and Discover exercises at the end of this chapter, you can explore the consequence of not closing the StreamWriter and learn alternatives to ensure that the data are written to the file. Still, closing the StreamWriter is the most direct and efficient way to achieve this purpose. To close the StreamWriter in the preceding example, you code:

```
    PhoneFileO.Close()
```

The event to place this statement depends on the application at hand. Assume in the preceding example, a phone number and a name are written on the file each time a button is clicked. This would suggest that the StreamWriter should be active as long as the program is in action. The most appropriate time to close the StreamWriter will probably be just before the form is closed. In such a case, the code to close the StreamWriter should be placed in the form closing event.

Tip

Always make sure that you have code to close a StreamWriter before the program ends to ensure all data are written to the file. This is the most direct and efficient way to ensure that all necessary output operations are complete.

About System.IO

You have seen that some objects and functions in your code are qualified with other names. For example, the StreamReader and StreamWriter are qualified with System.IO. This pertains to the way VS 2008 is organized. The huge software contains many class libraries (libraries that provide classes [templates for objects]), commands, and so forth that are developed by different groups. Because of the huge vocabulary of the software, it is unavoidable that some identifiers in one part conflict with those in another part. To resolve this issue, the software is organized into *namespaces*. Within each namespace, identifier names must be unique, but the same identifier names can be used in another namespace. To differentiate an identifier (of the same name) in one namespace from the one in another namespace, you qualify the name with the name space. Recall that the Left function discussed in Chapter 4 must be qualified with Microsoft.VisualBasic because Left is also a property of the form.

All namespaces coming with VS 2008 begin with System and Microsoft. When VS 2008 starts, it automatically includes some of these namespaces for references based on the profile setting for your project. Among these are Microsoft.VisualBasic, which provides most of the VB functionality, and System.Windows.Forms, which provides functionality for forms and controls. As you may guess by now, the StreamReader and StreamWriter are in the System.IO namespace, which is not automatically included in the project. That is why your code needs to qualify them with the namespace System.IO. You can leave out the namespace qualification for the two objects if you *import* the namespace into your project. To import System.IO, place the following code as the first line in the module (before the Class statement):

```
Imports System.IO
```

After you have this line in place, you will no longer need to qualify the StreamReader or StreamWriter. For example, you can code the following line without encountering any error:

```
Dim MyFileReader As StreamReader
```

File Dialog Boxes

The files created using the StreamWriter can be viewed and edited by the use of the Microsoft Notepad program. Files created by Notepad can also be input to your program using the StreamReader with the ReadLine and ReadToEnd methods.

Talking about Notepad, you probably notice that each time you use it to open or save a file, it displays a dialog box for you to specify the file in your computer system. Wouldn't it be nice if you can have the same facilities in VB to specify the file? VB 2008 provides the open file dialog and save file dialog controls to do these.

The open file dialog is used when you want to open a file; therefore, its default title is Open. The save file dialog is used when you want to save a file, so its default title is Save As. Similar to the timer control, these two controls are visible at design time, allowing the programmer to set their properties. At run time, however, they do not appear until invoked by code.

These two controls have a few different properties; however, they also have several common properties that you can set at design time or by code to perform the services you need. The following table lists some of these properties:

Property	Explanation	Example	Effect
Filter	Show only the types of file extension as specified.	`.Filter =“Text File (*.*) |*.txt”`	Only file with the txt extension will be shown in the dialog box
Title	A string to be displayed on the title bar of the dialog box in place of the default Open or Save As	`.Title =“Where is the File?”`	The dialog box title bar will display “Where is the File?”
FileName	Sets or returns the filename specified by the user	`MyFileName = .FileName`	MyFileName will hold the current FileName property
AddExtension	If set to True, file extension will be added in the file specification	`.AddExtension = True`	

The Filter Property

The Filter property allows you to specify what types of files you would like the file dialog to display. For each type of file you want to specify, you must give the description and the filter, separated by a | (pipe) symbol. For example, assume that you have an OpenFileDialog named cdlOpenFile, and you want it to show only files with the txt extension. (The cdl name prefix is the acronym for common dialog boxes, which include the file, font, color, and print dialog boxes.) You can specify the following:

```
    cdlOpenFile.Filter = “Text Files (*.txt)|*.txt”
```

In the preceding code, the string “Text Files (*.txt)” gives the description of the type of file your user will be looking for. The filter “*.txt” (following the |) specifies that the

system will display all files with the txt file extension. The * symbol is a wildcard specification that indicates to ignore the matching (all names are considered a match). You can also separate additional filters by additional | symbols. The following line will allow either text files or all files to be possible filters, depending on which filter the user chooses.

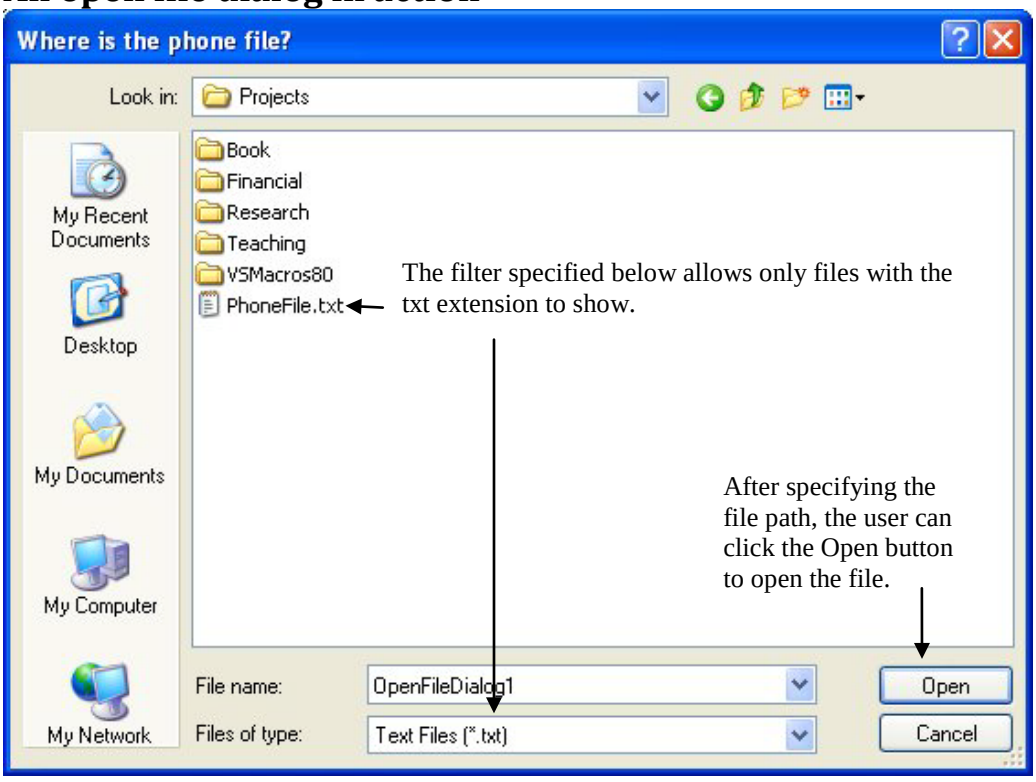
```
cdlOpenFile.Filter = "Text File (*.txt)|*.txt|All Files (*.*)|*.*"
```

To illustrate how one of these dialog boxes can be used for file specifications, suppose in the previous phone file example (for input) that all you know is that there is such a file. The file can be located at any folder with any name, so you want your user to specify the location for input using the OpenFileDialog. Again, assume the control has been named cdlOpenFile. Your code to open the file (associate the file with a StreamReader) can appear as follows:

```
'Title on the dialog box
cdlOpenFile.Title = "Where is the phone file?"
'Display only txt files
cdlOpenFile.Filter = "Text Files (*.txt)|*.txt"
cdlOpenFile.ShowDialog()  ' Display the open dialog box
' cdlFile.FileName below is obtained from the ShowDialog method
PhoneFileI = New System.IO.StreamReader(cdlOpenFile.FileName)
```

When the ShowDialog method is executed, the dialog box will appear as in Figure 6-3.

Figure 6-3
An open file dialog in action



After the user browses through the computer and selects the proper file, the filename is stored in the control's FileName property, which can then be used as the string for the file specification in the StreamReader creation statement.

Tip

When you want the file dialog to accept multiple extensions, separate each extension by a semicolon. For the example, the following filter will display all files with extensions of txt, html, htm, and css:

```
"Text files and HTML files|*.txt;*.html;*.htm;*.css"
```

Additional Notes

Both the OpenFileDailog and SaveFileDialog have *CheckFileExists* and *CheckPathExists* properties. When these Boolean properties are Set to True, the controls will check whether the file (folder) entered by the user in the dialog box exists. If not, the dialog box displays a message box to inform the user that the file (folder) must exist before it will accept the entry. Of course, setting the CheckFileExists property to True is more meaningful for the OpenFileDailog than for the SaveFileDialog. Set this property to True for the OpenFileDialog. It can save your program from failure when the user fails to provide a filename for an existing file.

The SaveFileDialog has the *OverwritePrompt property*, which when set to True will also check whether the specified file already exists. If it does, the dialog box warns the user that the file already exists and then asks the user whether the file should be overwrriten. Setting the property to True can protect the user against accidental loss of valuable data.

The preceding code example assumes that the user will always choose or enter a valid filename when the OpenFileDialog displays the dialog box. What if the user simply clicks the Cancel button? The FileName property will contain a zero length string. The creation of the StreamReader will fail, resulting in an execution time error. To guard against this potential problem, you can test the returned value from the showDialog method of the control before attempting to create the StreamReader object. The method will return a value, DialogResult.Cancel, when the user clicks the Cancel button.The code can appear as follows:

```
    If cdlOpenFile.ShowDialog = Windows.Forms.DialogResult.Cancel Then
        Exit Sub
    End If
    PhoneFileI = New System.IO.StreamReader(cdlOpenFile.FileName)
    'Other statements
```

6.2 Procedures: Subs and Functions

Consider the case of student registration application. The user first enters the student's ID number as well as all the courses and section numbers that the student intends to enroll. When the user clicks the Process button, the program will proceed to process the request by going through the following steps:

- Verify that the student number is valid and the student is allowed to enroll
- Look for all previous unpaid invoices, campus police tickets, and library fines owed by the student
- Verify for each course the student requests to enroll:
    - that the course and the section number are valid
    - that the class is not yet full
    - that the student has the prerequisites to take the course

- Compute the total tuition and previous balance
- Issue the invoice
- Update the student's record
- Update all the class records

The list can go even longer, but the point is simple: In this situation, if you place all the code in the button's Click event procedure, it will be hard to follow the program flow of logic. The program will be too long and difficult to read, understand, debug, change, or maintain. The solution to this problem is to divide the program into several subprograms, each handling a predefined special task, such as one of the major steps outlined. The button's Click event procedure will then provide just an outline of the steps by calling these subprograms. This arrangement will dramatically reduce the number of lines in the event procedure. Because each subprogram handles a smaller task, it is easier to develop code and to understand.

A partial, conceptual solution to the previous problem would probably be similar to the following:

```
Public Class Enrollment
    Private Sub btnProcess_Click(ByVal Sender As System.Object, ByVal e
      As System.EventArgs) Handles btnProcess.Click
        If BadStudentID(IDNumber) Then
            MsgBox("Needs a valid student ID number to proceed.")
            Exit Sub
        End If
        FindUnPaidRecords(IDNumber)
          .
          .
    End Sub
    Private Function BadStudentID(ByValue IDNumber As Integer) As
      Boolean
        'Routine to determine whether the student ID is valid
          .
          .
    End Function
    Private Sub FindUnPaidRecords(ByVal IDNumber As Integer)
        'Routine to locate student's unpaid records
          .
          .
    End Sub
    'Other event or general procedures
    .
    .
End Class
```

Each procedure referenced inside the event procedure can be written separately as a *Sub* or as a *Function* procedure, and can be placed anywhere in the class in the same area as the event procedures. Both types of procedures are different from the event procedure in that the event procedure is typically invoked by the occurrence of an event, whereas the former are invoked by a reference of their names in the code. These procedures are recognized as general (or separate) procedures, as a distinction from the event procedure. The key difference between Sub and Function procedures is that the latter returns a value, whereas the former does not. Additional distinctions are explained in the subsection "Additional Notes on General Procedures," later in this chapter. For now, the classification of procedures can be summarized as follows.

Procedure Type	How Invoked	Returns a Value?	Remark
Event	Triggered by event	No	
Sub	Referenced by name	No	Collectively recognized as general or separate procedures
Function	Referenced by name	Yes	

Writing a Sub Procedure

So, how do you create a Sub procedure? A *Sub* procedure is declared with a Sub keyword with the following syntax:

```
[Private|Friend|Public] Sub SubProgramName(Parameter List)
    Statements for the sub program
End Sub
```

where *SubProgramName* can be any valid identifier name, and *ParameterList* represents a list of parameters.

The parameter list can range from none to many and has the following syntax:

```
ByVal|ByRef Param1 As DataType1, ByVal|ByRef Param2 As DataType2,…
```

The *ByVal* and *ByRef* keywords indicate whether the parameter will be passed by value or by reference (address). When ByVal is specified, a copy of the value of the parameter is passed to the procedure. On the other hand, if ByRef is specified, the address of the parameter is passed. The default is ByVal. The following examples are valid subprogram headers.

```
Private Sub BringData()
Private Sub SearchPreviousInvoice(ByVal IDNumber as Long)
Public Sub Enroll(ByRef ID As Long, ByVal CourseSection As String)
```

The *Private, Friend, or Public Modifier* declares the scope of the procedure and has the same effect as a variable of the same scope.

Tip

Although the number of parameters for a procedure is determined by the task requirement, it is always difficult to use a procedure with too many parameters. Limiting the number to 5 or fewer appears to be a good rule of thumb; otherwise, try your best to order the list in a logical way so that you can easily remember the parameters needed one after another. Alternatively, try to redesign the code structure to see if it is possible to reduce the number of parameters needed.

The statements inside the procedure will be executed line by line in sequence when the procedure is invoked. The following Sub procedure computes and displays the area for a circle given its radius.

```
Private Sub ComputeArea(ByVal Radius As Double)
    ‘ This Sub computes and displays the area of a circle
    ‘given a radius, which is expected to be of the Double type
    Dim Area As Double
    Area = Math.Pi * Radius * Radius
    MsgBox( “The area of a circle with a radius of “ _
      & CStr(Radius) & “ is “ & CStr(Area))
End Sub
```

In this example, you declare the procedure to be Private; that is, the procedure will be recognizable and accessible only in this form. The procedure is named ComputeArea. It expects a parameter of the Double type and the parameter will be recognized in this

procedure as Radius. Inside the procedure, this parameter is used to compute the area of a circle, and the resulting area is displayed by the MsgBox statement.

Calling a Sub Procedure

You can call (invoke) a Sub procedure by using the following syntax:

```
    [Call] SubName(argument list)
```

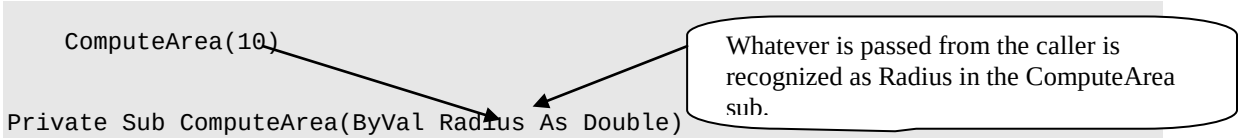
To call the Sub just written, you can code the following:

```
    Call ComputeArea(10)
```

or

```
    ComputeArea(10)
```

Notice that you must enclose the argument(s) with a pair of parentheses in either case. Notice also that you pass an argument with a value 10 to the Sub, ComputeArea, but the argument is recognized in ComputeArea as the parameter, Radius. That is, Radius is a symbolic name for any value that the calling procedure passes to the Sub. If you change the value to 8, 8 will still be recognized by the Sub as Radius. The relationship between an argument and a parameter in this context can be depicted as the following diagram:



In addition, you can also use different variables as arguments to call the Sub. For example, assume that you have declared both A and B as Double and assigned proper values to both variables. You can obtain the proper results by using the following statements:

```
    ComputeArea(A)
    ComputeArea(B)
```

To test how it works, enter the Sub procedure in the module in the same area as you would for event procedures; then draw a new button on the form. Name it btnCompute, set its Text property to Compute, and enter the following code:

```
Private Sub btnCompute_Click(ByVal Sender As System.Object, ByVal e As
  System.EventArgs) Handles btnCompute.Click
    Dim A As Double
    Dim B As Double
    A = 10
    B = 8
    ComputeArea(A)
    ComputeArea(B)
End Sub
```

Run the project and then click the button. The message box should first display the area of the circle for a radius 10; then, for 8.

Code Reusability with the General Procedure

This simple illustration has also demonstrated a powerful use of general procedures. Much like the use of variables to handle different data values, you can use a procedure to even handle different variables that require the same computational formulas or processing steps. You can use the same Sub repetitively anytime you need to handle a similar situation, even when the variables involved are not the same; therefore, general procedures enhance the reusability of your code.

To further illustrate this point, suppose you have a program that allows the user to set the Text, ForeColor, and BackColor properties of certain controls. Rather than writing the code for each specific control, you can write a generic Sub for this purpose, and call it wherever you need to perform the task. The Sub can appear as follows:

```
Private Sub SetControlProperties(ByRef TheControl As Control, ByVal
  TheText As String, ByVal TheForeColor As Color, ByVal TheBackColor As
  Color)
    With TheControl
        .Text = TheText
        .ForeColor = TheForeColor
        .BackColor = TheBackColor
    End With
End Sub
```

This Sub takes four parameters. The first parameter is a control, the second is the text, and third and the fourth are the colors, as indicated in the header. Inside the procedure, the specified control's Text, ForeColor, and BackColor properites are assigned the values given by the parameters.

If you want to set the properties for the control named txtName, you can call the Sub using the following code:

```
    SetControlProperties(CType(txtName, Control), "Jeff Tsay",
      Color.Blue, Color.Aqua)
```

If you want to set the properties for a button named btnSave, you can use this same Sub:

```
    SetControlProperties(CType(btnSave, Control), "Save", Color.Blue,
      Color.Red)
```

Notice how the first argument in each statement is specified. With Option Strict On, the specific type of control, such as textbox, cannot be passed as a generic control. The CType function is used to explicitly convert each specific type to the generic control.

Position and Type of Argument

An important point you must remember in calling a Sub is that the data or objects you pass as arguments are recognized by their positions, not by their names. With SetControlProperties's header as defined previously, if you try to call it with a statement such as the following:

```
    SetControlProperties("Jeff Tsay",Color.Blue, Color.Aqua, txtName)
```

you will get three parameters underlined by the compiler. The Sub expects the first parameter to be a control, but "Jeff Tsay" is a string, not a control. The same reason goes for the second and fourth parameters.

You may be wondering why then the statement:

```
    ComputeArea(10)
```

did not result in the same error. (Recall that ComputeArea expects a Double type parameter, but the number 10 is an integer.) The explanation is that even under Option Strict, the compiler allows implicit data conversion from a narrower range (Integer) to a wider range (Double).

Passing Data to Sub Procedures

In the preceding discussion, all data are passed by position. They, however, can be passed either by position or by name. The following discussion provides additional details.

Passing Data By Position

The preceding subsection stresses that data passed to the Sub procedure are recognized by position, not by name. This is true even when the data types of the arguments match those parameters expected by the Sub. For example, consider the following Sub:

```
Private Sub MakeTwice(ByRef Two As Double, ByVal One As Double)
    Two = One + One
End Sub
```

This procedure doubles the value of the second parameter, and assigns the result to the first one. If in another procedure you have two variables, such as X and Y (both declared to be the Double data type and assigned with proper values), and you use the following statements:

```
    MakeTwice(Y, X)
    MsgBox(X & Y)
```

you should see that Y has a value twice of X. In addition, if you use the statements:

```
    MakeTwice(X, Y)
    MsgBox(X & Y)
```

you should see that X has a value twice of Y.

This holds even if you use the variables Two and One in the calling procedure. To test, try the following code, and be sure to include the MakeTwice Sub.

```
Private Sub Form1_Click(ByVal sender As System.Object, ByVal e As
  System.EventArgs) Handles MyBase.Click
    Dim One as Double, Two As Double
    One = 1
    Two = 2
    MakeTwice(Two, One)
    MsgBox("One = " & One & " Two " & Two)
    MakeTwice(One, Two)
    MsgBox("One = " & One & " Two = " & Two)
End Sub
```

Run the project. When you click the form, you should see the first message box display two numbers: 1 and 2 the first time for the variables One and Two. The results are obtained as expected when the arguments are passed to the Sub as shown next.

```
        MakeTwice(Two, One)

Private Sub MakeTwice(ByRef Two As Double, ByVal One As Double)
```

The second message box, however, will display the numbers 4 and 2 for the variables One and Two. Again, the results are obtained (perhaps, not as expected for some beginners) when the arguments are passed as shown next:

```
       MakeTwice(One, Two)

Private Sub MakeTwice(ByRef Two As Double, ByVal One As Double)
```

The variable Two in the calling procedure is passed to MakeTwice as One, which is then doubled inside the procedure and assigned to the parameter named Two; however, the corresponding variable in the calling procedure to this parameter is the variable One. The result in the calling procedure is that One has twice the value of Two.

Passing Data by Name

There is actually a way to pass data by the parameter name. Using this approach, the position of the parameter will no longer be important. The syntax to pass data by name is as follows:

```
SubName(Parameter1 := Argument1[, Parameter2 := Argument2] . . .)
```

The := symbol is used to indicate that the left side is the parameter name, whereas the right side is the argument. Parameter1 can be a parameter in any position in the parameter list in the Sub procedure's header.
Consider the following statement in previous example again.

```
    MakeTwice(X, Y)
```

Using the named parameter syntax, you can have the same result by coding:

```
    MakeTwice(Two := X, One := Y)
```

or

```
    MakeTwice(One := Y, Two := X)
```

Of course, you can also code the following for the first statement in the above Form Click event procedure:

```
    MakeTwice(Two := Two, One := One)
```

or

```
    MakeTwice(One := One, Two := Two)
```

Again, keep in mind that the name on the left of the := symbol is the parameter name and the name on the right is the argument to be passed.

ByVal and ByRef Again

A derived question in the previous discussion is, "Why can a procedure change the value of a variable in *another* procedure?" This is because in your code, you specified ByRef for the parameter, Two. The argument's address, rather than its value, was passed. Whatever operations performed on the parameter are actually performed on the argument. Any change made on the parameter is actually made on the argument.

If you do not want a procedure to accidentally change the value of any variables passed to it, use the *ByVal* keyword (default) in its header. For example, the header:

```
Private Sub MakeTwice(ByVal Two As Double, ByVal One As Double)
```

will cause only the values of the arguments to be passed to this procedure, not their addresses. That is, a copy of the argument, not the argument itself, is passed to the procedure. The effect is this: Whatever changes made to the parameters inside the Sub procedure will not cause any change in the variables used as arguments in the calling procedure.

In the previous example, the use of ByVal will make no sense because there will be no way to obtain the intended result because ByVal nullifies all desired effect of doubling the value of one of the variables; however, not all the Subs are written for the purpose of changing the values of some parameters.
ByVal is compatible with other languages. It is also true that in general, passing data ByVal executes faster than ByRef.

Tip

There is another way to have the effect of ByVal if you do not want the variable passed to a procedure gets its value changed. Enclose the variable with a pair of parentheses. For example, you can code:

```
      MakeTwice ((Y), (X))
```

The arguments then will be taken as expressions, forcing the compiler to create two dummy variables to pass the data to the called procedure. Whatever changes made to the dummy variables are discarded as soon as the called procedure ends.

Terminating a Procedure Before Reaching the End: Exit Sub or Return

In some cases, your program logic in a procedure may be such that when certain conditions arise, there is no need to proceed further to execute the remainder of the code in the Sub. In such cases, you can use the Exit Sub or the Return statement to terminate the Sub procedure. The effect will be the same as if the Sub has reached the End Sub statement. For example, assume you have a Sub to save an inventory part in a file. The part number must be greater than 100; otherwise, it is considered invalid. Your Sub may appear as follows:

```
Private Sub SavePart(ByVal ID As Integer, ByVal PartName As String)
    If ID <= 100 Then
        MsgBox("The ID number is not valid. Part not saved.")
        Exit Sub 'You can substitute Return for Exit Sub here
    End If
    ' The routine to save the part will continue from here
End Sub
```

Exit Sub Versus End Sub

If both End Sub and Exit Sub (and Return) statements terminate a procedure, what is the difference between the two? The End Sub statement actually serves two purposes. It not only terminates a Sub procedure when the execution control reaches that point but also physically defines the boundary of a Sub procedure. It tells the compiler that the procedure ends here. No other statements beyond this point should be considered a part of this procedure. On the other hand, the Exit Sub (Return) statement terminates the Sub at that point and returns execution control to the calling procedure. Because no other statements inside the procedure will be executed after the control reaches this statement, the Exit Sub (Return) statement is usually included in an If block; that is, you would want to terminate a Sub only when certain conditions occur.

Event Procedures and General Sub Procedures

At this point, you may wonder about the differences between the event procedures and the general Sub procedures. After all, both have the keyword Sub. The two differ in several aspects:

- Event procedures are triggered by events. As you recall, these events are results of something occurring in the systems or the results of the user's actions. There is no explicit call in the code to invoke an event procedure. In contrast, the general Sub procedures must be triggered (invoked) by code. That is, somewhere in the code, the name of the general Sub procedure must be referenced, and only when this line of code is executed will the general Sub be invoked. Note, however, an event procedure can also be invoked by an explicit call like the way a general Sub is called.
- A corollary of the preceding difference is that in general, you will find general Subs are called from an event procedure or another general Sub procedure; however, it is rare that a general Sub calls an event procedure even though it is permissible to do so.

- By convention, the name of an event procedure follows a specific syntax structure. The first part of it is the name of the object with which the event is associated and the second part is the name of the event. These two parts are connected with an underscore. For example, the name for the Form1's Load event procedure is Form1_Load. Form1 is the name of the object and Load is the event. Between Form1 and Load, there is the underscore ("_"). More importantly, at the end of the name header, there is a Handles clause that begins with the keyword Handles followed by the name of the object, a dot (.), and the name of the event. In this clause, the name must be the object name and the event must be one recognized by the object. (Additional variations in naming event handlers are discussed in Chapter 12, "Object-Based Programming.") In contrast, you can create any name for a general Sub procedure as long as the name can be considered a legitimate identifier name.
- The number, order, and data type of parameters for event procedures are predefined by the system. You cannot change these elements in any event procedure. In contrast, you have complete freedom in deciding these elements in the general procedures that you create.
- In terms of scope, the event procedure must be Private, whereas the Sub procedure can either be Private, Friend, or Public.

These differences are summarized in the following table.

Difference in	Event Procedure	General Sub Procedure
Trigger	Event (triggered by the occurrence of the event)	Referenced by code (called by another procedure)
Caller	Seldom called from another procedure	Must be called by another procedure, which can be an event or general procedure
Name syntax	Should observe the following structure: ObjectName_EventName (variations discussed in chapter 12)	Can use any legitimate identifier name
Parameter list	Predefined by VB	Designed by the programmer
Scope	Private	Public (default), friend, or Private

Function Procedures

Consider the previous MakeTwice Sub procedure example again. The example was used to illustrate how arguments passed to a Sub are handled. If you carefully reexamine the design, you should realize that there is a better way to structure the code. Recall that in the Sub, one parameter (One) was used to compute the value of the other (Two), which could then be used by the calling procedure. That is, the Sub procedure was used to generate one result based on the value of another parameter. In such a case, a Function (instead of a Sub) procedure is a better choice. The key difference between a Sub and a Function procedure is that the latter returns a value, whereas the former does not.

The Function Procedure Header

So, how do you create a Function procedure? The Function procedure is declared with the *Function* keyword with the following syntax:

```
[Public|Friend|Private] Function Name(parameter list) [As Type]
    'Statements (including Exit Function or Return)
    Name = Expression
End Function
```

As you can see in the syntax, the Function procedure header has the scope modifier and the parameter list (similar to the Sub procedure); however, the function header accepts a Type declaration, whereas the Sub does not. The Type declaration at the end of the header specifies the type of data that the function will return. Inside the Function procedure, there must be at least a Return statement or an assignment statement with the function name on the left side. The value so assigned is the value returned to the caller. The Return statement returns a value to the caller and has the following syntax:

```
    Return(Expression)
```

The Differences Between Subs and Functions

There are several differences between Sub and Function procedures.

- Because the Function procedure is expected to return a value to the calling procedure, you should provide a Type declaration at the end of the header. (With Option Strict On, you must specify the type.) The type declared here is the data type of the value to be returned by the function. The Sub procedure is not expected to return a value. A Type declaration in its header is not allowed.
- Both types of procedures will contain statements that carry out computations. In a Function procedure, however, at least one statement should have the effect of returning a value. This can be either an assignment statement that has the function name on the left side or a Return statement with an expression as its parameter. The execution of this statement will cause the value of the expression to be returned to the calling procedure. On the other hand, the name of the Sub procedure must not appear in any assignment statement.
- Because the Function procedure returns a value, its name is typically used as a part of an expression, whereas the Sub procedure is used as a statement (when referenced without the keyword, ”Call”) or a part of a statement (when referenced with the keyword ”Call”).

Writing a Function Procedure

To illustrate the structure of a function, you can rewrite the MakeTwice Sub into a Function procedure as follows:

```
Private Function MakeTwice(ByVal One As Double) as Double
    MakeTwice = One + One
End Function
```

or

```
Private Function MakeTwice(ByVal One As Double) as Double
    Return(One + One)
End Function
```

The function is declared to be a Double type because it is expected to return a value of this type. Inside the first example procedure, the parameter One is doubled, and the result is assigned to MakeTwice—the name of the Function procedure. This statement enables

the function to return the value computed. In the second example procedure, the Return statement first doubles the parameter value of One and then returns the result to the caller. If none of the statements inside a Function procedure either assigns a value to the function name (in this case, MakeTwice) or has a Return statement with an expression, the function will return a value of zero (or a zero-length string if the function is expected to return a string).

The preceding function can then be used by other procedures as follows:

```
Two = MakeTwice(10)
```

or, assuming a proper declaration and assignment of value have been made for A,

```
MsgBox("Twice of " & A & " is " & MakeTwice(A))
```

Note that similar to the Sub procedure, the argument passed to the function does not have to be named One (the name used for the parameter) but can be any constant, variables, or expressions of the same type.

As another example, the previous Sub to compute the area of a circle can be rewritten as a function as follows:

```
Private Function AreaOfCircle(ByVal Radius As Double) As Double
    ' This function computes and returns the area of a circle
    'given a radius, which is expected to be of the Double type
    AreaOfCircle = Math.Pi * Radius * Radius
End Function
```

In this function, you make the function only compute and return the area, but have omitted the code to display the result. This choice is by design, not by the restriction on a function.

Using the Function Procedure

From the preceding examples, you must have noticed that the way to invoke a function is by the reference of the Function procedure's name. Arguments passed to a function must be enclosed in a pair of parentheses. Because the function is expected to return a value, the name of the function typically appears in an expression. That is, a function can be used exactly like a constant or a variable, in exactly the same way as built-in functions. But a Sub cannot be used in the same manner. The only difference between a built-in function and a general function procedure is that the former is predefined (built-in) in the system, whereas you are the one to define the latter whenever you choose to use one.

Additional Notes on General Procedures

You have been introduced to the basic syntax and concepts on general procedures. This subsection considers additional important issues and features regarding these kinds of procedures.

Function Versus Sub

Those examples in the discussion of Function procedures may make you wonder when you should create a Sub and when you should create a Function procedure. The key distinction between the two is that one returns a value and the other does not. In general, when you need the procedure to return a value, you should write it as a function. This includes the following situations:

- When the value returned from a function will be used like a variable in an expression; therefore, all computational problems that obtain a result based on a list of parameters naturally fall in this category.
- When there is a need to determine whether the computations/actions performed by the procedure are successful. For example, the execution of the procedure may fail because of some external conditions, such as required files are missing or not available, or an operation is canceled by the user. In such cases, the procedure can be written as a function and then returns an execution code to indicate whether the execution was successful, which can be used by the calling procedure to determine the proper courses of actions.

If the procedure is not required to return any value but just to carry out some activities, it will be more appropriate to create this procedure as a Sub.

Naming the General Procedure

All identifier names should be descriptive. Names for procedures are no exception. Because the Sub procedures will be more related to actions, it is advisable to use names that begin with a verb; for example, SaveCustomerRecord, DisplayAssetItem, and ClearScreen. The names for Function procedures are a bit trickier. Those functions for computational results are usually given the nomenclature of the values they return; for example, AreaOfCircle, CubeRoot, and GrossPay. Those functions used to carry out actions but expected to provide a return code are usually given names that begin with a verb, too. They can be easily confused with Subs. Fortunately, the confusion does not cause coding problems. When you do not need a return value from a function of this kind, you can use it as if it were a Sub by referencing the name without assigning the result to another variable. For example, suppose you have a function CheckDate that returns a value True and displays an error message when a String parameter passed to it is not a valid date. The function appears as follows:

```
Function CheckDate(TheDate As String) As Boolean
    ‘This function checks if a string contains a valid date
    ‘in the mm/dd/yyyy format and
    ‘returns a value True when the date is invalid
    If IsDate(TheDate) AndAlso Val(Microsoft.VisualBasic.Left(TheDate,
      2)) <= 12 Then
        ‘This appears to be a valid date. Do nothing
        Return(False) ‘Return False to caller
    End If
    ‘Execution will reach here only if it fails the above test.
    ‘Display an error message and set the return code to True
    MsgBox(“Must have a valid date to proceed.”)
    Return(True)
End Function
```

But suppose in your code, all you need is to display an error message if the string variable BirthDate is invalid. Its return code will not be used. You can still use the function just like a Sub as follows:

```
    CheckDate(BirthDate)
```

You can even invoke it by using the Call statement syntax as follows:

```
    Call CheckDate(BirthDate)
```

The CheckDate Function

Perhaps some explanation of the code in the CheckDate function is in order. The function takes a string, TheDate, as its only parameter. It assumes that the date is in the mm/dd/yyyy format. In the procedure, it first uses the IsDate function to check whether the string appears to be a valid date. If so, it further checks to see whether the first two digits in the string are not greater than 12. This is necessary because IsDate accepts strings in several different date formats, including the dd/mm/yyyy format. The string "31/12/2009" will be considered valid, although it should be considered invalid in the CheckDate function. The second expression in the If statement ensures that the month portion of the string is not greater than 12.

If a string passes both tests, it should be a valid date. The Return statement will return a value False. If the string fails either test of the IsDate or the second expression, it is considered invalid. The MsgBox function is used to display an error message. In addition, the Return statement will return a value True, indicating the string is not a valid date.

Documenting General Procedures

A procedure typically is written to perform a much more complex task than the examples illustrated previously; therefore, it is important to document the procedure properly with remarks (comments) when it is being developed. These comments should include at least the following:

- The purposes of the procedure
- What is returned if the procedure is a function
- A description of the required parameters
- Assumptions made
- The algorithm used if the problem is fairly complex

All the examples given previously are coded with this in mind. If you did not notice the comments given at the beginning of each procedure, you may want to take another look now. Of course, additional comments pertinent to any code in the procedure should be added, just as you would typically do for an event procedure.

Recursion

All the VB procedures can be recursive; that it, each procedure can call itself. The code in the procedure remains the same but the data passed through each call are supposed to be different. This feature can be a powerful programming tool. A complex programming problem can be solved with an algorithm that recursively (repetitively) divides the problem into a smaller problem of the same nature until the latter is simple enough to solve.

A classical example of the use of recursion is the computation of n factorial (n!). When n is large enough, the problem calls for a fairly long series of computations; however, the problem can be subdivided into a problem of n (n - 1)! That is, if you are able to obtain the result of (n - 1)!, all you have to do to obtain the result is to multiply n by the factorial of one less. You can now solve the problem of (n - 1)! in the same manner; that is, if you think of (n - 1) as n', n'! can be solved with the same algorithm. This divide-and- conquer method can be continued until the n at hand is a value one (1); then, there is no need to divide the problem any further. The result is 1. This value can be

returned to the previous level, which can then multiply, obtain the result, and return to yet another previous level. The process repeats until it reaches the original problem level. There the problem is solved.

The following table shows the process of arriving at the solution for 4! based on the description in the preceding paragraph.

Problem-Solving Description	Example	Solution Return Process	Solution
1. Original problem	4!	10. Obtain result and return	24
2. Divide the problem	4 x 3!	9. Obtain result and return	4 x 6
3. Divide the next problem 3!	3 x 2!	8. Obtain result and return	3 x 2
4. Divide the next problem 2!	2 x 1!	7. Obtain result and return	2 x 1
5. Solve the problem 1!	1	6. Return result to preceding level	1

Column 1 shows the solution process, and column 2 shows the numeric example based on the process. Column 3 shows how the result is obtained at each subsequent step, and should be read from bottom to the top with its corresponding numeric results in column 4. How can this process be implemented in VB? The following code provides the solution:

```
Private Function Factorial(N As Integer) As Long
    ‘ This function computes the value of N!
    ‘ where N is an integer
    ‘ Assumption: N is a number not to exceed 20
    ‘ Otherwise, overflow will result
    If N > 1 Then
        Factorial = N * Factorial (N - 1)
    Else
        Factorial = 1
    End If
End Function
```

This function literally implements the algorithm just described. When n is greater than 1, the function calls itself to find the result of the factorial of 1 less than the current n. The result returned from this call is then multiplied and returned to the caller, which can be itself. Only when n reaches a value not greater than 1 does the function consider the problem simple enough to find the solution (1), and return to the previous level.
The following table shows the execution process for an initial parameter value 4.

N	Function Call	Call Return (from Bottom to Top)
Initial call	Factorial (4)	24
4	4 * Factorial(3)	4 * 6
3	3 * Factorial(2)	3 * 2
2	2 * Factorial(1)	2 * 1
1	Return 1	1

To use a recursive procedure is no different from any other typical procedure. For example, after entering the code for the previous Factorial function, if you place the

following statement in a button Click event, you should see a message box displaying, “Factorial(4) = 24”.

```
    MsgBox(“Factorial(4) = “ & Factorial(4))
```

Cascading an Event Procedure

Even the event procedure is recursive; that is, an event procedure can trigger the same event and invoke itself within itself. In general, recursion for event procedures occurs by accident, not by design. They are written to handle events. Triggering an event to handle the event itself just does not make much sense. You should take care not to code an event procedure that causes recursion of itself. As an illustration of event cascading, consider the following code:

```
Private Sub txtNumber_TextChanged(ByVal Sender As System.Object, ByVal
  e As System.EventArgs) Handles txtNumber.TextChanged
    txtNumber.Text = CStr(1 + Val(txtNumber.Text))
End Sub
```

The code is placed in the text box’s *TextChanged event* that occurs when the value of the text box changes. The programmer may think the code will increase the number entered by the user by 1; however, when you run the project and press a numeric key into the text box, you will encounter an unhandled error. Why? Initially, the number entered by the user is increased by 1 and then assigned to the Text property of the text box. This action causes a change in the text box and thus triggers another TextChanged event, invoking the event procedure itself before the current event procedure ends. This process continues until the computer runs out of stack memory.

Actually, the code was placed in that event procedure because of the misunderstanding of the TextChanged event, which occurs after *each* key is entered or after an assignment of data to the text box. If the intent was to increase the value in the text box by 1 after the field has been properly entered, the code should be placed in either the text box’s Validating or Leave event when the user leaves the field.

Tip

If you encounter an unhandled error in your program, consider recursion as a possible source of error. If you use a recursive procedure, make sure it can exit without calling itself. If the error occurs in an event procedure, look at the left side of the assignment statements for the control name associated with the event. When you find one, check to ensure that the statement does not trigger the same event.

Optional Parameters

The general procedure can have *optional parameters* that can be omitted when the procedure is called. An optional parameter is specified with the keyword Optional, and must be provided with a default value. All parameters following an optional parameter must be optional; the syntax to specify the header for a general procdure is as follows:

```
[Public|Private] {Sub|Function}([required parameter list][,optional
  parameter list])
```

where the optional parameter list can be specified as:

```
Optional {ByVal|ByRef} Name1 As Type1[, Optional {ByVal|ByRef} Name2 As
  Type2]….
```

The following are valid examples of specifying optional parameters:

```
Sub AnyGeneralProc(ByVal A As Integer, Optional ByVal B As single =
  -9999, Optional ByVal C as Double = -9999)

Private Function QuickCount(Optional ByVal FirstCount As Integer = 0,
  Optional ByVal SecondCount As Integer = 0 ) As Integer
```

Because both of the QuickCount parameters are optional, when you invoke the function, you can provide both parameters or omit either or both of the parmeters as follows:

```
    QuickCount(X, Y)
    QuickCount(X)
    QuickCount(, Y)
    QuickCount()
```

The omitted parameter will have a value zero as specified in the header.

You have seen built-in functions that allow optional parameters. For example, the Rnd function allows omission of its only parameter, and the Mid function allows the omission of its last parameter. Typically, for the convenience of coding the calling procedure, a procedure can be written to allow optional parameters if their values can be assumed or derived. Consider a function that computes the net pay for employees. The net pay depends on the total earnings that include gross pay, commissions, and tips. But not all employees have all three elements. In this case, you can make the commissions and tips optional and code the header as follows:

```
Fucntion NetPay(ByVal GrossPay As Double, Optional ByVal Commission As
  Double = 0, Optional ByVal Tip As Double = 0) As Double
    'Statements to compute net pay
End Function
```

To compute the net pay for an employee who has gross pay and tips but no commission, you can then code:

```
    Amount = NetPay(GrossPay,, Tip)
```

Overloading Procedures

You can write two or more different procedures (Sub or Function) with the same name as long as they have different parameter lists in number, type, or both. For example, suppose you want to write a generalized function that gives you the smaller of two values that can be either of Long type of String type. You can code the function as follows:

```
Function Smaller(ByVal A As Long, ByVal B As Long) As Long
    If A <= B Then
        Return (A)
    Else
        Return (B)
    End If
End Function
Function Smaller(ByVal A As String, ByVal B As String) As String
    If A <= B Then
        Return (A)
    Else
        Return (B)
    End If
End Function
```

You can then use the function to obtain the smaller of two values as long as the two parameters are both either of the Long type or the String type similar to the following:

```
    MsgBox("The smaller of 23 and –10 is " & Smaller(23, -10))
    MsgBox("The smaller of X and b is " & Smaller("X", "b"))
```

Of course, you can add additional functions to handle other data types. This feature, recognized as *overloading*, makes it convenient to call procedures of similar functionality without having to differentiate between data types or the number of parameters. (Note that the Math.Min function does exactly what our example here attempts to do. Smaller is used as an example because of its simplicity.)
To explicitly indicate that a procedure overloads another, precede the Function or Sub keyword with the Overloads keyword. A more explicit header for the Smaller function should be:

```
Public Overloads Function Smaller(ByVal A As Long, ByVal B As Long) As
  Long
```

Note that if you precede one of the overloading procedures with the Overloads keyword, you must do the same for all procedures with the same name.

Uses of General Procedures

You may wonder why go through all the trouble of creating general procedures. After all, you may end up with more lines of code. In addition, the calls and returns between procedures impose overhead in handling all the housekeeping tasks inside the computer, which can slow down the program. These drawbacks, however, must be weighed against the advantages of creating and using these procedures. Here are some of their important advantages:

- General procedures facilitate top-down and modular design. A huge, complex task can be broken down into smaller, manageable tasks, each written as a procedure. When designed this way, the main program is much easier to read, and its logic is much easier to trace. The programmer can focus on requirements of the main task at the beginning, and then take care of the details of smaller tasks later. Several programmers can even work together on the same project, each taking care of some of the lower-level procedures. This is the advantage cited at the beginning of this section.
- Problems can be isolated, making the program easier to understand, debug, and revise.
- Procedures can be made reusable. Carefully designed general (public) procedures can be called from any part of the project, eliminating the need to repeat similar code in different parts of the project. This not only reduces the size of the program, but also simplifies the coding. Later, if the procedure needs to be revised or corrected, the programmer needs to focus on only one place (the procedure). Any change made to the procedure applies to all parts of the project that call the procedure, making it much easier (and more efficient and effective) to maintain the code.
- Function procedures can make code easier to understand. A function procedure returns a value that can be used just like a constant or variable in a formula. The meaning of the formula can be clear to the developer and the reader/reviewer. The complexity of the computations is isolated from the main focus of the problem. This structure of code will be much easier to review and understand than one that includes complex code to carry out the computations of the formula.

- Recursion simplifies problem solving. Wherever a complex problem can be subdivided into simpler problems using the same algorithm, a recursive procedure can be developed. This makes the program much shorter and easier to code.

6.3 The Contacts Project: An Application Example

This section presents a project that will use some of the features you have learned in this chapter. This project involves uses of file dialog boxes, file input and output, and general procedures. It also considers a few code design issues.

The Contacts Project

This project, Contacts, will be used to maintain a file that contains your friends' phone numbers and names. This file has been created as a text file, so it will be easy to handle the file with the StreamReader and StreamWriter. Note that a file can only be opened either for input or for output (or append) mode; that is, it cannot be associated with the StreamReader and the StreamWriter at the same time. When a file needs to be updated (edit, add, or delete records), you will need to handle it as follows:

- Open (create) a new output file.
- Read the original file, and let the user inspect the records one at a time.
    - If a record needs no correction, it can be saved in the new file as is; otherwise, the user can make the change and then save the record.
    - If a record needs to be deleted, the user will choose not to save it in the new file.
    - To add a new record, the user enters the new data and then saves it.

The User Interface

The form should contain data fields for the user to enter the phone number and name. These two fields can also be used to display the record read from the existing phone file.

- The field for the phone number is a masked text box.
- The field for the name is a text box.

The form should also contain four buttons for the user to initiate actions to do the following:

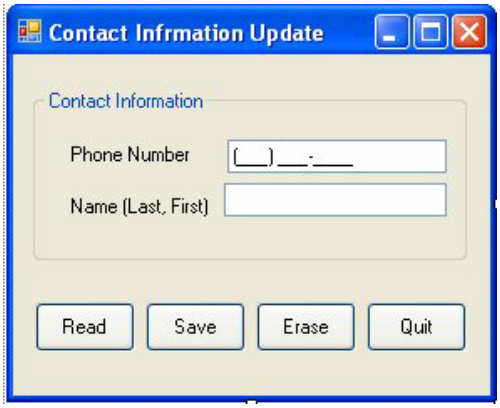
- Read a record from the existing file.
- Save the record displayed on the fields. These can be either from the file or entered by the user.
- Clear the fields so that the user can enter new data.
- Quit.

The form should also include an open file dialog named cdlOpenFile to prompt for file path to open for input, as well as a save file dialog named cdlSaveFile to prompt for file path to save (for output). Finally, this form should include a group box to hold the data fields (with their proper labels). The resulting visual interface appears as Figure 6-4.

The following table provides a list of controls along with selected properties referenced in the code.

Control	Property	Setting	Remark
Open file dialog	Name	cdlOpenFile	To prompt for input file path
	CheckFileExists	True	To make sure the file exists
Save file dialog	Name	cdlSaveFile	To prompt for output file path
	OverwritePrompt	True	To ensure an existing file is not accidentally erased
Masked text box	Name	mskPhone	Phone field
	Mask	(999) 000-0000	
Text box	Name	TxtName	Name field
Button	Name	BtnRead	To read a record
	Text	Read	
Button	Name	BtnSave	To save a record
	Text	Save	
Button	Name	btnClearScreen	To clear the screen
	Text	Erase	
Button	Name	BtnQuit	To close file and quit
	Text	Quit	

Figure 6-4
Visual interface for the Contacts project



In this application, existing contact data are read from an existing file when the Read button is clicked. Whatever contact information on the screen will be saved onto another file when the Save button is clicked. An open file dialog and a save file dialog are used to prompt for the file paths.

The program should behave in the following manner:

- As soon as the program starts, it should display an open file dialog box to prompt the user for the phone file and then a save file dialog box to prompt for the name to save as the new file.
- When the user clicks the Read button, it reads a record from the existing file and then places the record in the masked text box and the text box. When the file runs out of records, the program displays a message to such an effect.
- When the user clicks the Save button, the data is saved on the new file; then, the data fields onscreen are cleared.

- When the user clicks the Erase button, the data fields onscreen are cleared.

Designing the Code Structure

If you inspect the program's requirements closely, several points should draw your attention:

- The open and save file dialog boxes will be used to prompt for file paths: the open file dialog for the input file, and the save file dialog for the output file. Recall that it takes several lines of code before the dialog box's ShowDialog method is used to prompt for a file; however, most of the statements will be the same for both situations. This condition presents itself as a possibility to create a general procedure that can be called to handle both prompts. Should this procedure be a Sub or a function? Recall that the key difference between the two is in whether the procedure returns a value. It seems that requiring the procedure to return the filename will make the procedure more convenient to use. The programmer will not need to look for or remember the name of the file dialog box to obtain the result; therefore, the procedure should and will be coded as a function, and will be called GetTheFileName.
- There are two situations in which clear screen is called for—when the data is saved, and when the Erase button is clicked. Again, creating a general procedure to handle the clearing will shorten the code. There is no need for the procedure to return a value. This procedure will be coded as a Sub, and will be named ClearScreen.
- All other requirements can be implemented in the event procedures directly.

Given these considerations, you will create two general procedures and handle all other requirements in the event procedures.

Determining the Parameters for the General Procedures

The preceding discussion suggests that two general procedures should be created: ClearScreen and GetTheFileName. What parameters, if any, should each procedure have? Obviously, the action to clear the screen will be the same regardless of the situation that triggers the call; there will be no need for ClearScreen to require any parameters. In determining the parameters for the GetTheFileName function, several factors should be considered:

- There are two file dialogs in the program. The function needs to know which one to use. One easy way to handle this is to pass the specified dialog as a parameter.
- The program will prompt for two different files. The user should be properly informed of which file the program is looking for. Recall that both file dialog boxes have the Title property that can be used to specify the title. The title can be used to display the prompt. The calling procedure can specify the title. This procedure can then display the proper one, suggesting the title should be one of the parameters required by the procedure.
- What types of files should appear in the dialog box? Recall that the file dialog boxes have a Filter property by which you can specify the type of files to display in the dialog box. For this project, you could allow only the text file and hard-code the file filter in the procedure; however, if you would like the procedure to

be more general, you should also allow different file filters to be specified. This means another required parameter should be the file filter.

- You should also consider the file's existence status. When the function is prompting for the existing phone file for input, it should expect that the file has already existed. Because the open file dialog will be used in this case, its CheckFileExists property should be set to True at design time as shown in the preceding table. On the other hand, when it prompts for the output file, if the file already exists, it should caution the user that the file will be written over. Because the save file dialog will be used in this case, its WriteOverPrompt property should be set to True, again as shown in the preceding table. These two situations can be handled at design time. No code or parameter for the function is called for.

Coding the Project

Based on the preceding discussion, you are now ready to write the code for the project. Consider the two general procedures first.

The GetTheFileName Function

This procedure should be a function instead of a Sub. It should also take three parameters—one for the file dialog, another for the title, and the other for the file filter. The header of the procedure should appear as follows:

```
Private Function GetTheFileName(TheDialog As FileDialog, TheTitle As
  String, TheFilter As String) As String
```

Notice that the type of the function is declared to be As String because you expect the function to return a filename of the String type. The function is declared as a Private procedure because you do not expect it to be called by other forms or modules.

The calling procedure will then supply as its arguments: the file dialog (TheDialog), the dialog title (TheTitle) and the file filter (TheFilter) when calling this function. Inside this function, the TheTitle parameter can then be assigned to the dialog box's Title property, and the TheFilter parameter, the Filter property. The complete procedure should appear as follows:

```
Private Function GetTheFileName(ByVal TheDialog As FileDialog, ByVal
  TheTitle As String, ByVal TheFilter As String) As String
    With TheDialog
        .Title = TheTitle
        .Filter = TheFilter
        If .ShowDialog() = Windows.Forms.DialogResult.Cancel Then
            Return("")
        End If
        Return (.FileName) 'Return the filename to caller
    End With
End Function
```

The detail working of most statements in this procedure has been explained in the first section of this chapter. The way to use this procedure is further discussed in the next subsection, "Coding the Event Procedures."

Notice the following statements:

```
        If .ShowDialog() = Windows.Forms.DialogResult.Cancel Then
            Return("")
        End If
        Return (.FileName) 'Return the filename to caller
```

The If statement checks if the file dialog's SowDialog method returns a value Cancel, indicating the user has canceled the dialog. In such a case, an empty string is returned (`Return("")`); otherwise, the filename obtained is returned (`Return(.FileName)`).

The ClearScreen Procedure

The ClearScreen procedure clears the masked text box and the text box. The code should be fairly straightforward and should appear as follows:

```
Sub ClearScreen()
    Clear the masked text box
    mskPhone.Text = "" ' Restore the mask
    ' Clear the text box
    txtName.Text = ""
End Sub
```

Coding the Event Procedures

Recall that as soon as the project starts, the program will prompt the user for the path of the input file and then for the output file. In addition, there are four buttons to allow the user to initiate various activities.

Prompting for the File Paths

Because the program will prompt for the file paths as soon as the project starts, the code should be placed in the Form Load event. The prompt will involve calls to the GetTheFileName function, which requires three parameters, one of which is the file filter. As discussed previously, the expected file type should be the text file; the file extension should be txt. Just in case the user might use a different file extension, you will also allow all other file extensions. The filter string should be just a constant for the purpose of this example. It is suggested that constants be declared at a broader scope so that other procedures can use the same. You will place the declaration of the filter string in the general declaration area.

```
Const FileFilter As String =  "Text Files (*.txt)|*.txt|All Files
  (*.*)|*.*"
```

After the file paths are obtained, the input file should be associated with a StreamReader; the output file should be associated with a StreamWriter. These files should remain open throughout the life of the project because the program will continue to read and write while the project is running. This consideration should also hint that the StreamReader and StreamWriter associated with the files should be recognized by all the event procedures dealing with the files; therefore, the two object variables should be declared as the class-level variables. On the other hand, filenames are needed only at the time they are to be opened. Recall that in general, all variables should be declared with a scope as narrow as possible, so you will declare the filenames in the Form Load procedure while declaring the StreamReader and StreamWriter in the general declaration area.

In summary, the general declaration area should contain the declaration of two object variables and a constant as follows:

```
Const FileFilter As String = "Text Files (*.txt)|*txt|All Files
  (*.*)|*.*"
Dim ContactReader As System.IO.StreamReader
Dim ContactWriter As System.IO.StreamWriter
```

The *Form Load procedure* will declare the variables, prompt for the file paths, and open the files (associate files with the StreamReader and StreamWriter). It should appear as follows:

```
Private Sub frmContacts_Load(ByVal sender As System.Object, ByVal e As
  System.EventArgs) Handles MyBase.Load
    Dim InFileName As String
    Dim OutFileName As String
    'Prompt for the existing input file path
    InFileName = GetTheFileName(cdlOpenFile, "Where is the Phone
      File?", FileFilter)
    'Prompt for the output file path
    OutFileName = GetTheFileName(cdlSaveFile, "Specify Filename to
      Save", FileFilter)
    'Open phone file to read
    ContactReader = New System.IO.StreamReader(InFileName)
    ' Open output file to write
    ContactWriter = New System.IO.StreamWriter(OutFileName)
End Sub
```

Using the GetTheFileName Function

Notice how the GetTheFileName function is invoked. In the first case, to prompt for the existing input file, the open file dialog (cdlOpenFile) is passed as the file dialog, and the constant literal, “Where is the Phone File?” is passed as the TheTitle parameter. This should make the open file dialog display a title to prompt for the phone file. For simplicity, the program assumes the phone file has already existed and takes no steps in creating a new one. Before testing this project, create a text file with one phone number and name, using Notepad.

In the second case, to prompt for the output file path, the save file dialog (cdlSaveFile) is passed as the file dialog, and the constant literal, “Specify Filename to Save” is passed as the TheTitle parameter so that the dialog box can display it to prompt for the output file.

The two files are then opened as (associated with) ContactReader and ContactWriter, which are declared as class-level object variables to be recognizable by other event procedures. Note that in this event procedure, there is no testing for file validity. Several things can go wrong in this procedure. For example, the user may click the Cancel button in response to the prompt for the file path. This will result in a zero-length string being returned as the filename, causing an invalid filename error when the procedure attempts to open the file. Also, the user may specify the same filename for both input and output by mistake. This will cause a conflict in an attempt to open the file both for input and output. Adding additional If blocks to guard against these potential errors is left to you.

Reading the Phone File

When the user clicks the Read button, your program should read a record such as the phone number and name, and display the data on the masked text box and text box controls. The code to read and write the file depends on how the data appear in the data file. You expect the file to have one record per line. Each record has a phone number and name that are separated with a comma followed by a blank space. A sample file appears as follows:

```
8172726009, Tsay, Jeffrey J.
3333838, Handler, Henry H.
4809359999, Tuff, Tiffany T.
.
.
```

If the file runs out of records, a message should be displayed and the execution for the procedure should be terminated with the Exit Sub or Return statement. The code for the event procedure should appear as follows:

```
Private Sub btnRead_Click(ByVal Sender As System.Object, ByVal e As
System.EventArgs) Handles btnRead.Click
    Dim PhoneName As String
    Dim PhoneNo As String
    Dim Name As String
    Dim P As Integer
    If ContactReader.Peek() = -1 Then
        MsgBox("No more phone records.")
        Exit Sub
    End If
    PhoneName = ContactReader.ReadLine()
    P = InStr(PhoneName, ",")
    PhoneNo = Microsoft.VisualBasic.Left(PhoneName, P - 1)
    Name = Mid(PhoneName, P + 2) 'skip the space after comma to get the
    ‘name
    mskPhone.Text = Format(Val(PhoneNo), "(000)000-0000")
    txtName.Text = Name
End Sub
```

On the Format String

Notice how the phone number is formatted. The masked text box has the mask “(999)000-0000,” as shown in the visual interface. If the actual number is 3333838, the resulting string will be “(333)383-8.” On the other hand, the format string “(000)000-0000” will force zeros to fill in the leading missing digits, resulting in a string “(000)333-3838.”

Saving the Data Fields

When the user clicks the Save button, the program is expected to save the data onscreen onto the output file. The data can come from the Read operation either with or without the user’s correction and from the user’s direct entry into the data fields. A simple WriteLine statement should take care of adding a record. To be sure that there are data to be saved, the program should also check whether the name field is blank. In addition, after the record is saved, the data fields should be cleared by calling the ClearScreen procedure. The complete code for the event procedure appears as follows:

```
Private Sub btnSave_Click(ByVal Sender As System.Object, ByVal e As
  System.EventArgs) Handles btnSave.Click
    Dim PhoneNo As String
    If Len(Trim(txtName.Text)) = 0 Then
        MsgBox("Please enter a valid name.")
        Exit Sub
    End If
    PhoneNo = Replace(mskPhone.Text, ”(“, ””) ‘Remove left paren
    PhoneNo = Replace(PhoneNo, “)”, ””) ‘Remove right paren
    PhoneNo = Replace(Phone, “-“, ””) ‘Remove -
    ‘Write the phone number and name
```

```
    ContactWriter.WriteLine(PhoneNo & ", " & txtName.Text)
    ClearScreen()
End Sub
```

Notice the statements to remove the extra characters from the masked text box's text property. The mask's literals (parentheses and dashes) are part of the Text property. However, those phone numbers stored in the file do not carry these extra characters. The Replace function (explained in Chapter 4) is used to remove these characters.

Erasing the Data Fields

The user can choose to erase the data fields when he or she sees no need to preserve the record. The user can then proceed to read another record, or enter his or her own data. The event procedure can be easily taken care of by a call to the ClearScreen Sub:

```
Private Sub btnClearScreen_Click(ByVal Sender As System.Object, ByVal e
  As System.EventArgs) Handles btnClearScreen.Click
    ClearScreen()
End Sub
```

Terminating the Program

Finally, terminating the program is a simple procedure that should close the form.

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

Also, as noted previously, the ContactReader and ContactWriter objects associated with the filesshould be closed, too. Beware that the form can also be closed if the user clicks the Close button in the form's title bar. In this case, the Quit button's click event will not be triggered. Placing the code there will not close the objects.

The form closing event will always be triggered regardless how the form is closed, by code or by the user's click on the Close button. Those objects should be closed in this event. The code appears as follows:

```
Private Sub frmContacts_FormClosing(ByVal sender As System.Object,
  ByVal e As System.ComponentModel.CancelEventArgs)
  HandlesMyBase.Closing
    ContactReader.Close()
    ContactWriter.Close()
End Sub
```

Additional Remarks

The preceding procedures should complete the requirements for the project. Before leaving this example, keep in mind that it is presented only for the purpose of illustrating how the various parts you have learned in this chapter can be pieced together. Please note the following points in particular:

- This program assumes that the phone file has already existed. To test the code for this example, you should create a text file first, perhaps with Notepad. The phone number should be the first field, and the name should be the second field of each record (line). Be sure to separate the phone and the name with a comma followed by a space.
- When you work with the program as the user, you should feel very nervous about the accuracy of the results. For example, if you repetitively click the Read button,

the program will continue to read the records from the existing file. There is no warning that the previous record has not yet been saved. Also, the records in the old file that have not been read before the program quits will not be saved. All these situations can certainly cause the new file to miss records that should have been kept. The needed refinements to guard against potential losses of data are left to you.

- In general, you would rather have a way to update a record randomly; that is, you would rather have the ability to recall a record for edit at your random choice instead of having the program dictate the sequence in which records are retrieved. The approach used in the example will take too long and cost unnecessary effort and attention (plus potential errors) of the user. A better approach would be to structure the file so that it is capable of being randomly accessed, or to save the records in a database table. In either case, a record can be retrieved and updated directly. New records can also be added to the file (database table) without any need to deal with the existing records. Databases are discussed in Chapter 9, "Database and ADO.NET."
- Despite the shortcomings of the example project just noted, you should be able to appreciate the reasons for creating those general (separate) procedures and the design for the parameter(s) called for by each procedure. In addition, you should also notice the contexts in which various statements such as Exit Sub can be used.

Summary

- MsgBox can be used not only to display messages, but also prompt the user for direction to guide the flow of control in your program.
- The StreamReader can be used to associate with (to open) a file for input purposes. It provides various input capabilities:
    - The ReadLine method can read one line at a time.
    - The ReadToEnd method can read the entire file in one operation.
    - The Peek method offers a way to test whether the end of file has been reached.
- The StreamWriter can be used to associate with (to open) a file for output purposes:
    - Its WriteLine method will add an end of line marker, ensuring the next output operation begins with a new line.
    - Its Write method will write the text without adding the end of line marker, so the next output operation will start from where the previous output ends (will continue on the same line).
- When no additional input (output) operation is needed, the StreamReader (StreamWriter) should be closed to release the resources associated with the object. In the case of StreamWriter, closing the object ensures that all data are properly written to the file.
- VS 2008 is organized into various namespaces. Object names that are in a namespace not imported into the project must be qualified by the namespace; for example, StreamReader must be qualified by System.IO.

- You can import a namespace by placing an Imports statement at the beginning of the module.
- File dialogs (open file dialog and save file dialog) can be used to prompt the user to specify the filename. Always set the Filter property so that only pertinent files are displayed in the dialog box.
- General procedures (also recognized as separate procedures) provide many advantages. Before writing large block of code in an event procedure, analyze the problem and then determine whether the code can be simplified by creating separate Subs and Functions. These separate procedures can be invoked anywhere in the event procedure.
- General procedures differ from event procedures in that the former must be invoked by a reference of the procedure name; the latter are triggered by events.
- Subs differ from functions in that the former do not return any value; the latter do.
- General procedures enhance code reusability because they can accept different variables as parameters as long as the data types match or conversions are allowed.
- In general, parameters are passed by position. With proper syntax, parameters can be passed by name.
- Parameters can be passed by value (ByVal) or by reference (ByRef). By value is the default.
- Sub procedures can be terminated before the flow of control reaches the procedure end (End Sub) by the use of Exist Sub or Return statement.
- Function procedures can be terminated before the flow of control reaches the procedure end (End Function) by the use of Exit Function or Return statement. The former does not return a value. The latter can be coded with an expression, such as Return(X), to return a value.
- A function can be used in an expression like a constant, a variable, or a built-in function. The only difference between a general function and a built-in function is that the former is defined by the programmer; but the latter is pre-defined and provided by the system.
- As in all cases, the names of general procedures (Subs and functions) should be meaningful. Inside each procedure, proper comments should be inserted to enhance it understandability.
- All procedures (including event procedures) in VB are recursive. Recursion can be powerful in solving complex programming problems when these problems can be divided into smaller problems of the same nature; but invoking an event procedure recursively seldom has practical meaning.
- A procedure can have optional parameters. All parameters following the first optional parameter must also be optional.
- More than one procedure can have the same procedure name. Procedures coded this way are said to be overloaded. The parameters of these procedures must be different in *numbe*r, in *type*, or in both.

Explore and Discover

6-1. Default Button for the MsgBox Function. Enter the following code in the form click event of a new project. Note the different way of specifying the buttons and icon).

```
Dim Answer As Integer
Answer = MsgBox("Cannot find the file. Proceed?", vbYesNoCancel Or
  vbQuestion)
MsgBox(Answer)
```

Run the project and click the form. What do you see the message box display as the default button? Press the spacebar. What value do you see in the second message box?

Revise the MsgBox statement as follows:

```
Answer = MsgBox("Cannot find the file. Proceed?", _
    vbYesNoCancel Or vbQuestion Or vbDefaultButton2)
```

Run the project again and tclick the form. What do you see the message box display as the default button? Press the spacebar. What value do you see in the second message box this time?

6-2. Closing a Program without Closing a StreamWriter. Place the following code in the form click event. Note that to test this project, you need to have a Temp folder in Drive C. Change the path to some other folder if you do not want to create a Temp folder. Run the project, click the form, and end the program. Use Notepad to view the file. Do you see any text in the file?

```
Dim MyFileWriter As New System.IO.StreamWriter("c:\temp\test.txt")
MyFileWriter.WriteLine("This is a test line")
```

The line written by the object MyFileWriter was not actually written on the file when your program ends. Now, add the following line to the code.

```
MyFileWriter.Close()
```

Again, run the program and click the form. Use Notepad to view the file. What do you see this time? It is important to close the StreamWriter to ensure that data are actually written onto the file.

6-3. The AutoFlush Property of the StreamWriter. Modify the code in exercise 6-2 as follows:

```
Dim MyFileWriter As New System.IO.StreamWriter("c:\temp\test.txt",
  True)
MyFileWriter.AutoFlush = True
MyFileWriter.WriteLine("This is a test line")
```

Run the program and then click the form. Use Notepad to view the file. What do you see this time? When the StreamWriter's AutoFlush property is set to True, the StreamWriter automatically flushes its buffer, or forces the operating system to write on the file and clear the data in the file buffer. The data are actually written onto the file even when you do not close the StreamWriter; however, in this manner, the file buffer is not efficiently used. Setting the property to True slows down your computer, especially when your program has large amount of data to output.

6-4. The Flush Method of the StreamWriter. Modify the code in exercise 6-3 as follows:

```
Dim MyFileWriter As New System.IO.StreamWriter("c:\temp\test.txt",
  True)
```

```
MyFileWriter.WriteLine("This is a test line")
MyFileWriter.Flush()
```
Run the program and click the form. Use Notepad to view the file. What do you see this time? The StreamWriter's flush method flushes the file buffer and causes the data to be actually written onto the file; however, its effect is still not the same as closing the object. The resources associated with the file are not released. The file is not actually closed; if you click the form again, your program will encounter an error for attempting to open a file already in use. This same problem holds with setting the AutoFlush property to True. What lesson have you learned from this exercise? Always close the StreamWriter and StreamReader when it is no longer needed.

6-5. Append Or Overwrite. Draw two buttons on a new form. Name the first one btnAppend and set its Text property to Append. Name the second one btnOverwrtie and set its Text property to Overwrite. Enter the following code:

```
Private Sub btnOverwrite_Click(ByVal Sender As System.Object, ByVal e
  As System.EventArgs) Handles btnOverwrtie.Click
    Dim MyFileWriter As New
      System.IO.StreamWriter("C:\Temp\TestFile.txt", False)
    MyFileWriter.WriteLine("ABCD")
    MyFileWriter.Close()
End Sub

Private Sub btnAppend_Click(ByVal Sender As System.Object, ByVal e As
  System.EventArgs) Handles btnAppend.Click
    Dim MyFileWriter As New
      System.IO.StreamWriter("C:\Temp\TestFile.txt", True)
    MyFileWriter.WriteLine("ABCD")
    MyFileWriter.Close()
End Sub
```

Note that the only difference between the two procedures is in the second parameter in creating the object—in the Dim statement where the object is created. One specifies False; the other, True.

Run the project. Click the Append button. Each time you click the Append button, you should see one more line of ABCD when you view the file with Notepad.

Click the Overwrite button and then inspect the file again. What do you see? Click as many times as you want. You should always see only one line of ABCD in the file. The Overwrite mode (the second parameter set to False) erases the previous content of the file; the Append mode (the second parameter set to True) appends data at the end of the existing file.

6-6. Creating the StreamWriter at the Class Level? Can you create the StreamReader or StreamWriter at the class level? Place the following code in the general declaration area to find out.

```
Dim MyFileWriter As New System.IO.StreamWriter("C:\Temp\TestFile.txt", True)
```

6-7. Using The Color Dialog. Draw a color dialog and a button on a new form. Name the button btnSetColor and set its Text property to Set Color. Enter the following code:

```
Private Sub btnSetColor_Click(ByVal sender As System.Object, ByVal e As
  System.EventArgs) Handles btnSetColor.Click
    ColorDialog1.ShowDialog()
```

```
    Me.BackColor = ColorDialog1.Color
End Sub
```

Run the project, click the Set Color button, and select a color. What do you see? The color dialog obtains the setting for its Color property from the user by its ShowDialog method. Its Color property setting can then be used in the program.

6-8. The InputBox Function. Draw a text box onto a new form. Enter the following code in the form click event.

```
TextBox1.Text = InputBox("What is your name?",  "Name Prompt", "Debbra
    Solomon")
```

Run the project, click the form, and carefully inspect the dialog box. Can you associate the code with the text in the dialog box? After you click the form, do the following (one each time), and inspect the result in the text box:

- Click the OK button.
- Press the Enter key.
- Click the Cancel button.
- Enter your name in the box and then click the OK button.
- Enter your name in the box and then click the Cancel button.

Have you got a good feel about how the InputBox function behaves? This function allows the program to obtain data from the user on the fly, and can be used conveniently to test your program logic under certain circumstances. It cannot, however, be used in a serious data-entry application because it pops up only when called, and input from the function can be verified only afterward (not while the data is being typed in); therefore, it is not further discussed in the text.

6-9. Relationship of Argument and Parameter. Use two buttons. Name one btnTestA; and the other btnTestB. Set their Text properties to “Test A” and “Test B,” respectively. Enter the following code.

```
Sub Messenger(ByVal C As Integer)
    MsgBox("I get a value " & C)
End Sub

Private Sub btnTestA_Click(ByVal sender As System.Object, ByVal e As
  System.EventArgs) Handles btnTestA.Click
    Dim A As Integer
    A = 100
    Call Messenger(A)
End Sub

Private Sub btnTestB_Click(ByVal sender As System.Object, ByVal e As
  System.EventArgs) Handles btnTestB.Click
    Dim B As Integer
    B = 120
    Messenger(B)
End Sub
```

Run the project and then click each button. What message do you see? Notice that you can use two different syntactical expressions to call a Sub procedure. Also, how are A and B related to C in this illustration?

6-10. Positions of Arguments and Parameters. Use a button, and name it btnTest. Set its Text property to Test. In its Click event procedure and a separate Sub procedure named Divider, give the code as follows:

```
Private Sub btnTest_Click(ByVal sender As System.Object, ByVal e As
  System.EventArgs) Handles btnTest.Click
    Dim A As Single
    Dim B As Single
    A = 4
    B = 2
    Divider(B, A)
End Sub

Sub Divider(ByVal A As Single, ByVal B As Single)
    MsgBox("The quotient is " & A / B)
End Sub
```

Click the button. Is there anything puzzling to you? Recall that arguments passed to a Sub procedure are recognized by their positions, not by their names. What is recognized in the event procedure as A is recognized as B in Divider. It's the position, not the name, of the variable in the argument list that matters.

6-11. Passing Data by Name. (continued from exercise 6-10). If you pass the arguments without referencing the names of the parameters, they are recognized by position, as shown in the preceding exercise. As explained in the text, however, there is a way in which you can pass the arguments by the parameter names. For example, without any change to the Divider Sub in the preceding exercise, you can rewrite the Test Click event procedure as follows:

```
Private Sub btnTest_Click(ByVal sender As System.Object, ByVal e As
  System.EventArgs) Handles btnTest.Click
    Dim A As Single
    Dim B As Single
    A = 4
    B = 2
    Divider (B := B, A := A)
End Sub
```

Run the program and then click the button. Do you see the correct result? Inspect the syntax carefully. The := symbol is used to indicate the parameter assignment. The name on the left side of the := symbol is the parameter name. The name on its right is the argument.

Replace the line in the event procedure to call Divider as follows:

```
    Divider(B := 2, A := 4)
```

Run the project again. Do you still obtain the correct result?

When you use this syntax, the positions of the parameters no long matter. They are recognized by name. Can you think of situations in which the use of named parameters is more convenient?

6-12. Optional Parameters. VB provides a way to write procedures that require optional parameters. For example, suppose you are interested in a function that will return the largest value from a list of two or three arguments. You can code the function as follows:

```
Function Biggest(ByVal X1 As Long, ByVal X2 As Long, Optional ByVal X3
  As Long = -9999999999) As Long
    Dim X As Long
```

```
    If X1 >= X2 Then
        X = X1
    Else
        X = X2
    End If
    If X3 = -9999999999 Then
    ElseIf X3 > X Then
        X = X3
    End If
    Biggest = X
End Function
```

Note that the third parameter starts with the keyword Optional, which denotes that the parameter can be omitted. A default value must be specified as shown. To check whether it is present (passed from the calling procedure), your code checks against the default value.

Try the preceding code with the following code placed in the form click event:

```
    MsgBox(Biggest(12, 34, 53))
```

Are there other situations in which optional parameters can be useful?

6-13. Overloading a Procedure. Add the following code to the project in exercise 6-12.

```
Function Biggest(ByVal X1 As String, ByVal X2 As String, Optional ByVal
  X3 As String = "") As String
    Dim X As String
    If X1 >= X2 Then
        X = X1
    Else
        X = X2
    End If
    If X3 = "" Then
    ElseIf X3 > X Then
        X = X3
    End If
    Biggest = X
End Function
```

Place the following code in the form click event:

```
    MsgBox(Biggest(12, 34, 53))
    MsgBox(Biggest("d","V"))
```

Run the program and then click the form. What do you see? Although you have two functions of the same name in the class, the compiler knows which one to use because the two functions have different number or types of parameters. Notice also that you are not required to specify the third parameter because it is optional.

6-14. Effect of ByVal and ByRef. Draw a button on a new form, name it btnTest, and set its Text property to Test. In its Click event procedure and a separate Sub procedure named Setter, give the following code:

```
Sub Setter(ByVal C As Integer, ByRef D As Integer)
    C = 10
    D = 10
End Sub

Private Sub btnTest_Click(ByVal sender As System.Object, ByVal e As
  System.EventArgs) Handles btnTest.Click
```

```
    Dim A As Integer
    Dim B As Integer
    Setter(A, B)
    MsgBox(“After setting the values, A = “ & A & “ and B = “ & B))
End Sub
```

Click the button. Why don’t you get the same result for A and B? Recall that a ByVal parameter uses a copy of the value of the argument. Any change to that parameter will not cause a corresponding change to the argument passed to the Sub procedure.

6-15. Expressions as Arguments. (Continued from exercise 6-14). Suppose the previous Setter procedure is revised as follows:

```
Sub Setter(ByRef C As Integer, ByRef D As Integer)
    C = 10
    D = 10
End Sub
```

The parameters will be passed by reference; however, you code the calling event procedure as follows:

```
Private Sub btnTest_Click(ByVal Sender As System.Object, ByVal e As
  System.EventArgs) Handles btnTest.Click
    Dim A As Integer
    Dim B As Integer
    Setter((A), (B))
    MsgBox("After setting the values, A = " & A & " and B = " & B)
End Sub
```

What results do you obtain when you test the program? Why are A and B’s values not set to 10? The pairs of parentheses enclosing the variables make each argument an expression, not the original variable. Expressions are passed to procedures by value, not by reference, no matter how the parameters are declared in the header of the procedure.

Exercises

6-16. Using the MsgBox Function. Add the masked text box control to your project. Draw a masked text box and a button on the form. Name the masked text box mskPhone, and set its Mask property to “(999)000-0000” (Phone Number). Set its Prompt Character property to a blank space, set the button’s Text property to Save, and name it btnSave.

Provide code so that when the button is clicked, your program will check whether the user has entered a 10-digit number. If not, your program should display a message (use the MsgBox function) indicating that the phone number field is incomplete and then ask the user whether it is okay to save. If the user clicks the No button, the focus is set on the masked text box and the Save procedure is exited. When the masked text box has a legitimate number or when the user indicates that it is okay to save, the program displays a message stating that the record has been saved. (Do not provide code to actually save the record.)

6-17. Ensuring Data Are Saved. Add code to the Contacts example in Section 6.3 so that when the Read button is clicked, your program will check whether the name field has any data. If so, your program will display a message box asking the user whether to save the data before reading another record. The message box should display three buttons: Yes, No, and Cancel. If the user clicks the Yes button, your program should save the data fields before reading in another record. If the user clicks the No button, your program

should proceed to read another record. If the Cancel button is clicked, your program should neither save the record nor read another record.

6-18. Ensuring Data Are Saved (continued from exercise 6-17). Refer to the Contacts example in Section 6.3 again. Add code to the project so that the remaining records in the original file can be saved when the user wants to quit. The additional code should work as follows:

- When the user clicks the Quit or Close button, the program checks to see whether the original file contains any unread data. If not, the program quits; otherwise, the program proceeds to the next step.
- Display a message (use the MsgBox function) warning the user of the unread/unsaved data and asks the user whether the data should be copied into the new file. The MsgBox function should display a Question icon with three buttons: Yes, No, and Cancel.
- If the Yes button is clicked, your code will copy the remaining records in the old file to the new one and then quit. If the No button is clicked, your program quits. If the Cancel button is clicked, your program does not quit. (*Hint:* Place your code in the Form Closing event.)

6-19. The WhatFollows Function. Write a function that will take either the Integer type or String type as its parameter. If the parameter is of the Integer type, the function increases the magnitude of the value by 1, and returns the result (that is, if the parameter value is 3, the function returns 4; if the value is –3, the function returns –4). If the parameter is a letter, the function returns the next letter, for example if the parameter value is c, the function returns d; and if the letter is z, it returns a. If the string parameter is not a letter, it returns the same value as the original parameter. (*Hint:* Use the overloading feature. Write two functions with the same name.)

6-20. A Personal Memo Project. Design and write code for a project that will open your personal memo (a file with the path "C:\My Documents\MyMemo.txt") and display the content in a text box. Set its MultiLine property to True. You can then edit the content, and save the updated version with the same filename.

6-21. Save a Record (Data Entry Screen). Suppose that a form has a masked text box and two text boxes. They are named mskSSN, txtLastName, and txtFirstName, respectively. You would like for the data entered to be saved as a record in fixed-length format (one record per line); that is, data in each field are to have the same length (pad with blank spaces, if necessary). The lengths for the fields should be 9, 12, and 12, respectively. Write a Sub to save data entered.

From what event procedure should the Save Record procedure be called? Is there a reason why the Save routine should be written as a separate procedure?

6-22. Read and Display a Record. Write a general Sub procedure to read the data saved in the preceding exercise and then display the results in the three VB controls.

From what event procedure should the Read Record procedure be called? Is there a reason why the Read routine should be written as a separate procedure?

6-23. The Missing Data Function. Write a function to return a Boolean value True if the text string passed to this function contains only blank spaces or is zero length; otherwise, the function returns False. Name the function Missing. Its header should appear as follows:

```
Function Missing(TheText As String) As Boolean
```

6-24. Random Uppercase Letters. Write a function that will return a random uppercase letter. Test the function as follows:

- Draw a text box, with proper label, and a button on a new form.
- Name the button btnGenerate.
- Provide the code so that when the user clicks the button, the function is called and the letter it generates is displayed in the text box.

6-25. The Month-End Function. Write a Function that will take the month and year as its parameter, and return a string that represents the last day of that month; for example, MonthEnd(2, 2004) will return the string "02/29/2004." (*Hint:* Use the DateAdd function. The last day of a month is one day before the first day of the next month.)

6-26. The Julian Date Function. Write a function that will return a Julian date given a valid date string such as "01/01/1999." A Julian date is a date represented by a five-digit number whose first two digits represent the year, and the last three digits give the sequence number of the day in the year. For example, the Julian date for 01/01/1999 is 99001.

6-27. The GetTheFileName Function. Use an OpenFileDialog and a SaveFileDialog in a form. Write a function that will prompt the user for a filename, and return it as a string. The function should have the following header:

```
Function GetTheFileName(ForWhat As String, Filter As String) _
  As String
```

The parameter ForWhat expects an "O" (for "Open") or "S" (for "Save"). The parameter Filter expects a string specifying the file filter, as explained in the chapter. Make sure that when "O" is specified, the dialog box will insist that the file must exist; and when "S" is specified, the user will be warned if the file already exists.

The function will return the filename (a string) specified by the user in the dialog box. If the user clicks the Cancel button in the dialog box, the function should return a zero-length string.

6-28. The Grade Function. Write a function that will return a letter grade. The function expects an integer parameter that represents the score. The grading system is based on the typical 90, 80, 70, and so on, cut-off criteria.

6-29. String of Specified Length. Write a function that will take a string parameter and an integer parameter, and return a string of the length as specified in the second parameter. If the string parameter is longer than the specified length (L), the function will return the first L characters of the string. If the string is shorter than the specified length, the function will return the string with additional spaces padded at the end.

6-30. Numeric String of Specified Length. Write a function that will take two integers as its parameters. The first parameter represents the number to be converted to a string and returned. The second parameter specifies the length of the returned string. If the string to be returned is longer than the specified length, its extra characters on the left are truncated. If the string is shorter than the specified length, it should be padded with spaces on its left.

6-31. The Present Value of an Annuity Function. The present value of the annuity of one dollar ($1), P, can be computed by the following formula:

$$P = (1 - (1 + R)^{-n}) / R$$

where *R* is the rate of interest per period, and *N* is the number of periods.
Write a function with the following header to perform the computation:

```
Function PresentValue(R As Double, N As Integer) As Double
```

Compare the results obtained from this function with the results from using the built-in PV function. Use the keyword PV function to search the Index tab of the online Help file for additional information on how to use the function.

6-32. Recursive Function. Write a recursive function to compute the following series.

$$S = 1 + 2 + 3 + \ldots + N$$

The function should have the following header.

```
Function Series(N As Integer) As Long
```

(*Hint:* Rewrite the formula as follows:

$$S = N + (N - 1) + (N - 2) + \ldots + 2 + 1)$$

6-33. Recursive Function. Write a recursive function to approximate the following infinite series.

$$S = B + B / D + B / D^2 + B / D^3 + \ldots$$

Where *D* is greater than 1.
The function should have the following header.

```
Function Series(B As Double, D As Double) As Double
```

(*Hint:* Rewrite the formula as follows:

$$S = B + (B + (B / D + (B / D^2 + \ldots))) / D$$

The series can be evaluated by the beginning value (B) plus the series that begins with the current beginning value divided by D. Terminate the function when B/D is very small—say, .1E-10.

Projects

6-34. Revisiting the FICA Computation Project. Refer to exercise 5-32 in the previous chapter. Modify the FICA withholding computation project so that the previous data come from an existing file, and the resulting year-to-date (y-t-d) data are saved onto a new file.

The existing file contains the y-t-d data on all employees who have been paid previously. Each record consists of employee Social Security number (SSN), employee name, y-t-d pay, and FICA withholding. A sample record from the data file appears as follows:

```
"111-11-1111", "John Smith", 10000, 465
```

Your program should have buttons to Read, Compute, Save, and Quit. These buttons should align across the top of the form. You should add controls to display the employee SSN and name in the form, while all other data fields in your visual interface remain the same.

Your project should behave in the following manner.

- When the project starts, it should prompt for the existing (y-t-d) file and the file to which updated data will be saved. To simplify your code, create an empty y-t-d file using Notepad for the first round before testing your program.
- When the user clicks the Read button, your program will read a record from the existing file. If the file runs out of records, your program should display a message to that effect. The pay data will be displayed in the group box for the previous period, and the employee's SSN and name should be displayed in the proper controls. Before your program proceeds to read, it should check to see whether the data in the current screen have been saved. If not, it should prompt the user whether to save and should save the data if so instructed. (See requirement below for instruction on what to save.)
- The user should enter current pay in the group box for current data and then click the Compute button. Your program will then display the results in the proper fields as described in exercise 5-32.
- When the user clicks the Save button, the data in the group box for cumulative total should be saved in the same format as the previous sample record. All fields on the form should then be cleared.
- Data on new employees can be entered when all the fields are cleared. The user should enter the employee's SSN, name, and current pay. Click the Compute button and then click the Save button. (This is what you should do to create a y-t-d file for the first round.)
- When the user clicks the Quit or Close button on the title bar, your program should check whether the existing file has any unread data. If so, it should copy all these records to the new file, and display a message informing the user of this action.

(*Note:* This project emulates a sequential file processing system although it is much simplified.)

6-35. A Mini Word Processor. Develop a project that will provide the following word processing capabilities: opening a file; allowing the user to edit, search, and replace text; and saving text onto a file. The user interface should have buttons for Open, Search, Save, and Quit across the top of the form; a text box for text editing and processing at the center of the form; and a group box for search and replace. The group box should be placed below the text box, and should contain a text box for search text, another text box for replace text, a button with a Find Next text, another with a Replace text, and the other with an OK text. The text boxes in the search group box should be accompanied with proper labels.

The application should behave as follows:

- When the project starts, all the buttons across the top of the form and the text box (txtDoc) for word processing should appear. The form should have space enough to show all these controls. (The aforementioned group box is not visible. In

addition, there is no space in the form to hint its existence.) The user can start entering and editing text or click the Open button to bring text from an existing file. (*Hint:* Set the group box's Visible property to False. Compute the form's Height property using the Top and Height properties of txtDoc.)

- When the user clicks the Open button, a dialog box should appear to prompt for an existing file; then the data should be retrieved and displayed in the text box. Note that before your program retrieves data from an existing file, if the text box contains text and has been changed, your program should prompt the user whether to save the change. (*Hint:* Use a class-level variable to keep track of the change. Set it to False when the file is first open, or when the text is saved; then set it to True when the text changes.)
- When the user clicks the Save button, your program should prompt for an output file path. The text will then be saved in the specified file path.
- When the user clicks the Search button, the group box for Search and Replace will appear below txtDoc. (*Hint:* Compute the form's Height property using the group box's Top and Height property.) After the group box appears:
    - When the user clicks the Find Next button, your program will search the text in txtDoc for the text specified in the Search text box. If a match is found, it will be highlighted; otherwise, the program displays a Not Found message.
    - When the user clicks the Replace button, whatever appears in the Replace text box will replace the highlighted text in txtDoc. (*Hint:* See exercise 4-10 in Chapter 4 for hints for replacing text.)
    - When the user clicks the OK button, the group box disappears and the form goes back to its previous size.
- When the user clicks the Quit button, the program should check to ensure that the changes in the text have been saved. If not, it should prompt the user whether to save. If so, handle the saving operation as specified in the previous requirement.