CHAPTER 24

EXCEL AND VB.NET

In 2002, Microsoft released the first version of its development suite **Visual Studio.NET (VS.NET)** together with the **.NET Framework**. Since then, Microsoft has released new versions of the Framework and development suite in quick succession. Microsoft has strongly indicated that .NET is the flagship development platform now and for the foreseeable future.

Visual Basic.NET (VB.NET) is part of VS.NET, and despite its similarity in the name with **Classic VB** (VB6), the two have little in common. VB.NET is the successor to Classic VB and as such it provides the ability to create more technically modern solutions, a large group of new and updated controls, and a new advanced IDE. Moving from Classic VB to VB.NET is a non-trivial process, primarily because VB.NET is based on a new and completely different technology platform.

Excel developers also face the situation where applications created with the new .NET technology need to communicate with applications based on the older COM technology, for example, VB.NET applications communicating with Excel. Because Excel is a COM-based application it cannot communicate directly with code written in .NET.All .NET code that communicates with Excel must cross the .NET □ COM boundary. This is important to keep in mind because it is a challenge to manage and can have significant performance implications.

In the first part of this chapter, VB.NET is introduced along with the .NET Framework. The second part of this chapter focuses on how we can automate Excel with VB.NET. Finally we cover ADO.NET, which is used to connect to and retrieve data from various data sources. ADO.NET is the successor to classic ADO on the .NET platform.

To provide a better understanding of VB.NET, we develop a practical solution, the **PETRAS Report Tool.NET**. This solution is a fully functional Windows Forms based reporting tool. It retrieves data from the PETRAS SQL Server database and uses Excel templates to present the reports.

VB.NET, ADO.NET, and the .NET Framework are book-length topics in their own right; what we examine here and in the two following chapters merely scratches the surface. At the end of this chapter you find some recommended books and online resources that provide additional detail on these subjects.

.NET Framework Fundamentals

The .NET Framework is the core of .NET. Before we can develop or run any .NET-based solutions, the Framework must be installed and available. The Framework provides the foundation for all .NET software development. The .NET Framework is also responsible for interoperability between .NET solutions and COM servers and components. This topic is covered later in the chapter. For the purposes of our discussion, we can think of the .NET Framework architecture as consisting of two major parts:

- **A huge collection of base class libraries and interfaces—**This collection contains all the class libraries and interfaces required for .NET solutions. **Namespaces** are used to organize these class libraries and interfaces into a hierarchical structure. The namespaces are usually organized by function, and each namespace usually has several child namespaces. Namespaces make it easy to access and use different classes and simplify object references. We discuss namespaces in more detail when presenting VB.NET later in this chapter.
- **Common Language Runtime (CLR)—**This is the engine of the .NET Framework, and it is responsible for all .NET base services. It controls and monitors all activities of .NET applications, including memory management, thread management, **structured exception handling (SEH)**, **garbage collection**, and security. It also provides a **common data type system (CTS)** that defines all .NET data types.

The rapid evolution of the .NET Framework is reflected in the large number of versions available. Different Framework versions can coexist on one computer, and multiple versions of the Framework can be run side-by-side simultaneously on the same computer. However, an application can only use one version of the .NET Framework at any one time. The Framework version that becomes active is determined by which version is required by the .NET-based program that is loaded first. A general recommendation is to only have one version of the Framework installed on a target computer.

Because there are several different Framework versions in common use and we may not be able to control the version available on the computers we target, we need to apply the same strategy to the .NET Framework as we do when targeting multiple Excel versions: Develop against the lowest Framework version we plan to target. Of course there will also be situations that dictate the Framework version we need to target, such as corporate clients who have standardized on a specific version.

As of this writing, the two most common Framework versions are 2.0 and 3.0. Both versions can be used on Windows XP, and version 3.0 is included with Windows Vista and Windows Server 2008. Visual Studio 2008 (VS 2008) includes both of these Framework versions as well as version 3.5. By providing all current Framework versions, VS 2008 makes it easy to select the most appropriate version to use when building our solutions. Versions 3.0 and 3.5 of the .NET Framework are backward compatible in a similar manner as the latest versions of the Excel object libraries.

The .NET Framework can run on all versions of Windows from Windows 98 forward, but to develop .NET-based solutions we need to have Windows 2000 or later. If we plan to target Windows XP or earlier we need to make sure the desired version of the .NET Framework is installed on the target computer, because these Windows versions do not include the Framework preinstalled. All versions of the Framework are available for download from the Microsoft Web site and can be redistributed easily. To avoid confusion, we only use version 2.0 of the .NET Framework in this chapter and the next.

NOTE The standard version 3.5 .NET Framework distribution is around 197MB in size. Microsoft provided a lighter edition of about 25MB in size that can be installed on the target computers instead. To find out more about this edition, search for the phrase ".NET Framework Client Profile Deployment Guide" at www.microsoft.com.

Visual Basic.NET

With VS.NET we can create Web applications, server applications, database applications, console applications, Windows desktop applications, setup and deployment projects, and much more. VS.NET ships with the following programming languages: Visual C#, VB.NET, and Visual C++.

VB.NET is distributed in all VS.NET packages as well as in a standalone version. However, not all capabilities are present in all distributions.

You need to select the version of VB.NET that fits your requirements best. Table 24-1 shows the capabilities related to Excel development and the distributions in which they are available.

Table 24-1 Available Tools in Different Versions of VS.NET

	VB.NET Express	VS.NET Standard	VS.NET Professional
Automate Excel	3	3	3
Shared Add-in Template (To create managed COM add-ins with.)		3	3
Office templates			3
Visual Studio Tools for Office System (VSTO)			3

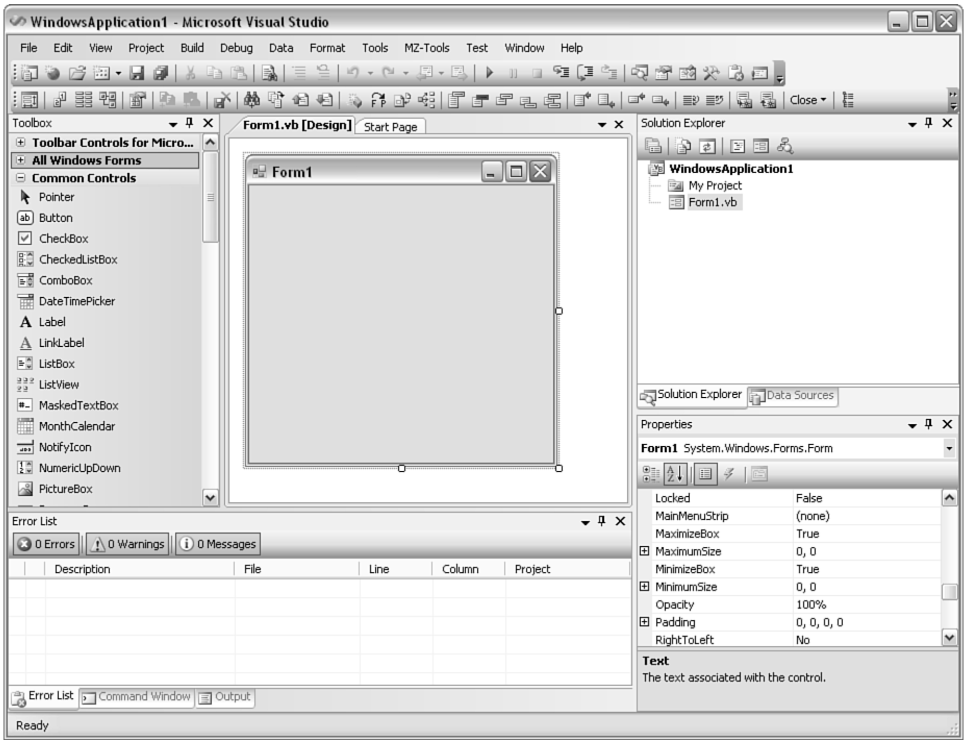
For a full comparison among the versions, see http://msdn.microsoft.com/en-us/vs2008/products/cc149003.aspx. If you just want to try out VB.NET you can download the free Express Edition from Microsoft's Web site. VS.NET Professional is required if you plan to develop **managed COM add-ins** and VSTO solutions. It is also required to follow the discussions here and in the next two chapters.

VB.NET was the first version of VB that broke backward compatibility badly enough that you could not even open a project created in an earlier version of VB. If you have non-trivial Classic VB projects that you would like to transfer to VB.NET, the best choice is to create them from scratch in VB.NET. Microsoft has some tools to ease the transition, but for larger VB projects they cannot do all the work. On the other hand, you may also consider keeping your Classic VB solutions for as long as it is still possible to run them on the Windows versions your solution targets. VB.NET is the first BASIC language version that fully supports object oriented programming (OOP). It means that with VB.NET we can fully utilize encapsulation, inheritance, and polymorphism.

Code that targets the .NET runtime is described as **managed code** while code that cannot be hosted by the .NET runtime is described as **unmanaged code**. **Assemblies** are the binary units (*.DLL or *.EXE) that contain the managed code. Since it is common that one .NET assembly contains only one binary unit, it is safe to refer to .NET-based DLL files as assemblies.

The Visual Studio IDE

The **Visual Studio IDE (VS IDE)** is shared by all .NET programming languages. The VS IDE is a complex development environment, even for developers who are very familiar with the Classic VB IDE. Figure 24-1 shows the VS IDE with a simple VB.NET Windows Forms project open.



Locked	False
MainMenuStrip	(none)
MaximizeBox	True
⊞ MaximumSize	0, 0
MinimizeBox	True
⊞ MinimumSize	0, 0
Opacity	100%
⊞ Padding	0, 0, 0, 0
RightToLeft	No

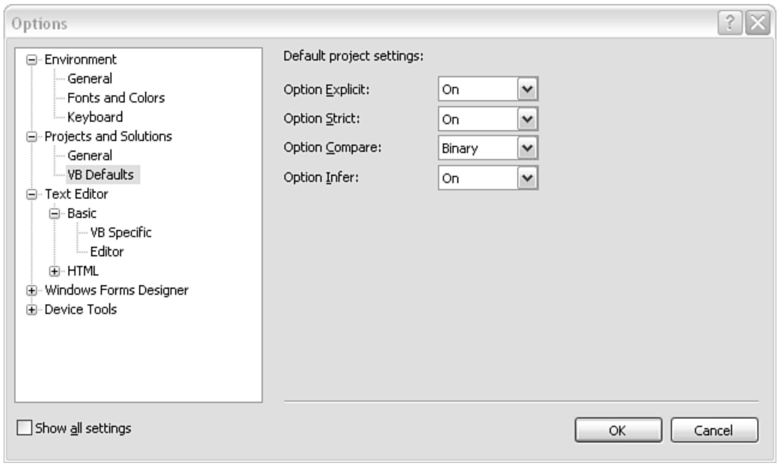
Text
The text associated with the control.

FIGURE 24-1 The Visual Studio .NET IDE

When you first run VS.NET, you are prompted to select a development category for VS.NET to use in customizing the environment. If your previous experience is with Classic VB or VBA, you will probably want to allow VB.NET to be your first choice of programming language. In this case, choose the **Visual Basic Development Settings**. The VS IDE is also highly customizable by the user, but before beginning to customize it you should learn the basics using the default configuration.

General Configuration of the VS IDE

After launching the VS IDE, you should change some general configuration settings for the development environment. Start by selecting *Tools > Options...* from the menu. This displays the Options dialog shown in Figure 24-2.



Default project settings:

Option Explicit: On

Option Strict: On

Option Compare: Binary

Option Infer: On

FIGURE 24-2 The general Options dialog

The Options dialog organizes its settings in a tree view on the left side. The *VB Defaults* section under *Projects and Solutions* contains four of the more important settings for VB.NET development. We recommend that you set them exactly as shown in Figure 24-2. A detailed description of each setting follows:

- **Option Explicit—**Determines whether VB.NET requires us to declare all variables before using them.
- **Option Strict—**Turning on this setting disallows late binding (to improve performance), implicit data type conversion, and provides **strong typing** (strict use of type rules with no exceptions).
- **Option Compare—**Specifies the default method used for string comparisons. It can either be Binary (case-sensitive) or Text (case-insensitive). The default value is Binary, which provides the same text comparison behavior as Classic VB. See Chapter 3, "Excel and VBA Development Best Practices," for more information.
- **Option Infer—**When this setting is turned on it allows us to omit the data type when declaring a variable and instead let VB.NET

identify ("infer") the data type. Listing 24-1 shows a simple example. The right-hand value tells the compiler the data type is an Integer. Declaring a variable and giving it a value at the same time in this manner is fully supported in VB.NET.

Listing 24-1 Omitting the Data Type When Declaring a Variable

```
Dim iCountRows = 225
```

When working with **VB.NET solutions** (a solution can contain one or more projects), these settings can be overridden at the code module level. This means, for example, that if we really need to use late binding in one code module we can modify the Option Strict setting at the top of that code module. Listing 24-2 shows how to turn off the Option Strict setting and also change comparisons to Text.

Listing 24-2 Changing Settings at the Code Module Level

```
Option Compare Text
Option Strict Off
```

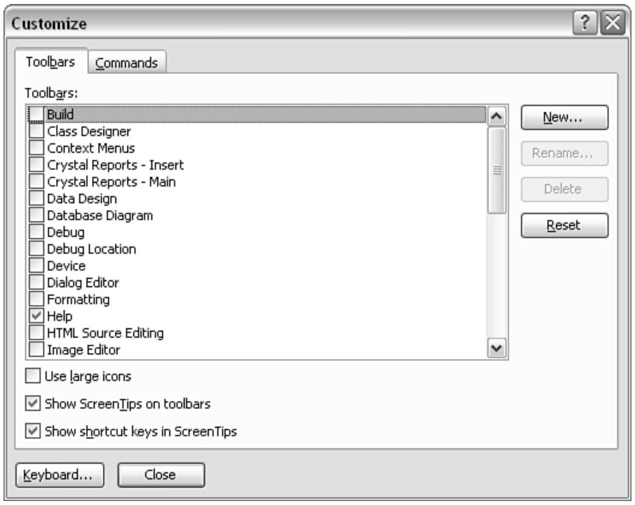
Adding line numbers to your code can make many development tasks easier, the debugging process in particular. To activate this option, expand the *Text Editor* section in the Options dialog and select the *Basic* section below it. Check the option *Line numbers* and then close the dialog.

Next we make screentips and keyboard shortcuts available in the IDE. Choose *Tools > Customize...* from the menu. This displays the Customize dialog shown in Figure 24-3. Check the two options *Show ScreenTips on toolbars* and *Show shortcut keys in ScreenTips* and then close the dialog.

The final setting is to make various docked windows in the IDE hide themselves when they are not being used. This provides us with a workspace that is not cluttered with open windows not relevant to the current context.

1. Click on the window you want to hide so it gets the focus.
2. On the *Window* menu click on the option *Auto Hide* or click on the pushpin icon on the title bar for the window.
3. Repeat these steps for every window that you want to auto hide.

When an auto-hidden window loses focus, it automatically slides back to its tab on the edge of the IDE.



Toolbars:

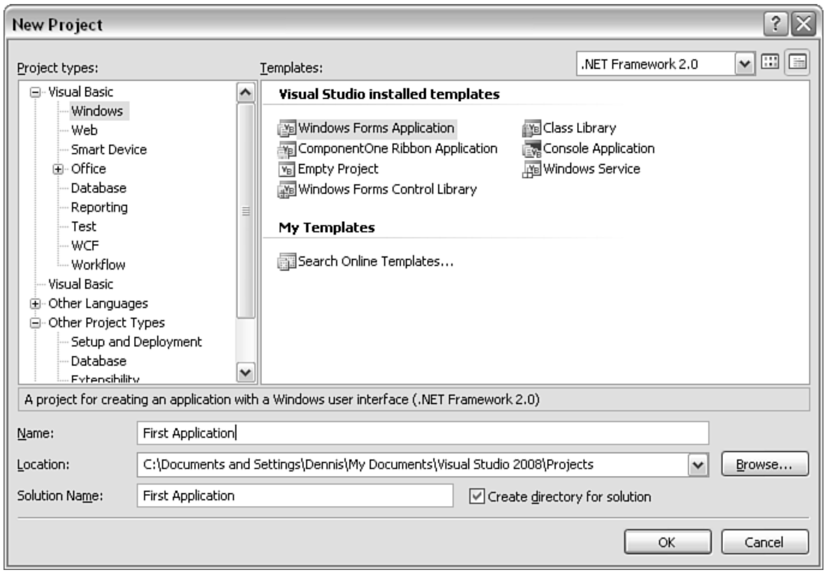
☐ Build
☐ Class Designer
☐ Context Menus
☐ Crystal Reports - Insert
☐ Crystal Reports - Main
☐ Data Design
☐ Database Diagram
☐ Debug
☐ Debug Location
☐ Device
☐ Dialog Editor
☐ Formatting
☑ Help
☐ HTML Source Editing
☐ Image Editor

☐ Use large icons
☑ Show ScreenTips on toolbars
☑ Show shortcut keys in ScreenTips

FIGURE 24-3 The Customize dialog

Creating a VB.NET Solution

We create a new VB.NET project by selecting the *File > New Project...* from the menu. This displays the New Project dialog shown in Figure 24-4.



.NET Framework 2.0

A project for creating an application with a Windows user interface (.NET Framework 2.0)

Name: First Application

Location: C:\Documents and Settings\Dennis\My Documents\Visual Studio 2008\Projects

Solution Name: First Application

☑ Create directory for solution

FIGURE 24-4 The New Project dialog

Since we are creating a Windows based-solution, select *Windows* in the *Project types* section and then select the *Windows Forms Application* template. We also select the version of .NET Framework to target using the combo box in the upper-right corner. Next, enter the name "First Application" in the *Name* box. By default, the solution name is the same as the name entered in the *Name* box, as shown in Figure 24-4. The solution name is also used to name the main folder of the project. Finally, click the OK button to create the solution.

The **Solution Explorer** window provides the workspace for working with files and projects inside VB.NET solutions. Figure 24-5 shows the workspace for our solution. A single Windows Form has been added to the solution and we have right-clicked on the form to display the shortcut menu containing the various actions available to perform on that object.

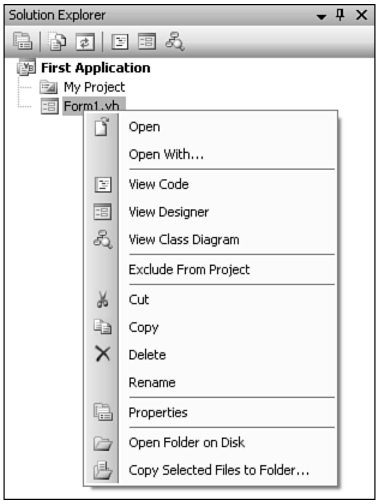


FIGURE 24-5 The Solution Explorer window

Windows Forms are the basic building block of many solutions. They provide us with a graphical user interface to which we can add controls. Windows Forms and all other Windows controls are contained in the `System.Windows.Forms` namespace. Windows Forms are in many ways identical to their counterpart Forms in Classic VB but are more modern and offer more properties to work with.VB.NET provides a large number of Windows controls for various purposes. However, use the new controls with good judgment. They exist to create a friendly user interface, not confuse the user.

Although VB.NET is designed to use Windows Forms controls, we can still use ActiveX controls. Therefore, if we own expensive third-party ActiveX controls, we can still use them in VB.NET. To add a control to a Windows Form, click the control's icon in the Toolbox and then drag and drop over the area on the form where you want the control to be placed. For our solution we add a label control, combo box, and two buttons to the Windows Form and resize the form itself. Figure 24-6 shows how the final Windows Form looks.

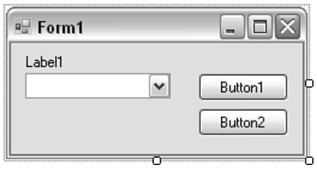


FIGURE 24-6 The Windows Form

Before we add code to the Windows Form, we set the tab order for the controls. Select *View > Tab Order* from the menu. The tab order for each control is now displayed visually on the form. Clicking on a control's tab number increases the number. Change the tab order so that it matches the order shown in Figure 24-7. To exit the tab order view, select *View > Tab Order* from the menu again.

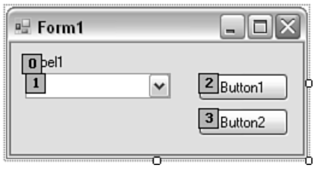


FIGURE 24-7 The tab order for the form

As a final step, we add code to the solution. Select *View > Code* from the menu. This opens the class module for the Windows Form. The first event we use is the Load event of the Windows Form. This is created by first selecting *(Form1 Events)* from the combo box in the upper-left corner of the module and then selecting *Load* from the combo box in the upper-right corner of the module. Listing 24-3 shows the code in the Load event.

Listing 24-3 The Code for the Load Event of the Windows Form

```
Private Sub Form1_Load(ByVal sender As Object, _
                       ByVal e As System.EventArgs) _
                       Handles Me.Load

        'Create and populate the array with names.
        Dim sArrNames() As String = {"Rob Bovey", _
                                     "Stephen Bullen", _
                                     "John Green", _
                                     "Dennis Wallentin"}
        With Me
            'The caption of the Form.
            .Text = "First Application"

            'The captions of the label and button controls.
            .Label1.Text = "Select the name:"
            .Button1.Text = "&Show value"
            .Button2.Text = "&Close"

            'Populate the combobox control with the list
            'of names.
            With .ComboBox1
                .DataSource = sArrNames
                .SelectedIndex = -1
            End With

        End With
End Sub
```

In this code, we create a string array, set values for various control properties, and then add the array as a data source for the combo box control. We use a single dimension array to populate the combo box with the list of names. It is a perfectly accepted practice to declare and initialize an array at the same time in VB.NET, as shown in Listing 24-3. When using this approach we do not need to specify the size of the array because this is inferred from the number of items within the scope of the curly brackets.

The next step is to get the selected value from the combo box and display it in a message box. Before doing that we need to import the namespace `System.Windows.Forms` into the code module, which gives us a shortcut to the .NET `MessageBox` class. Importing namespaces saves keystrokes each time we refer to objects that are part of the imported namespaces. It also makes our code easier to read and maintain by making it less verbose.

The `Imports` statements tell the compiler which namespaces the code uses. Usually we first set a reference to a namespace and then we import it to one or more code modules. Here we only do the latter because the `System` namespace is referenced by default in all new VB.NET solutions. This is because Visual Studio automatically adds a reference to the `System` namespace when a new VB.NET project is created. At the top of the Form's class module we add the `Imports` statement shown in Listing 24-4.

Listing 24-4 The Imports Statement

```
'To use the messagebox object.
Imports System.Windows.Forms
```

The namespace `Microsoft.VisualBasic` also belongs to the namespaces that are referenced by default in all new VB.NET solutions. This namespace is also globally imported, which means we do not need to import it into individual code modules to use it. From a practical standpoint this means we can use the well-known MsgBox function instead of its .NET variant. However, in Listing 24-5 we use .NET `MessageBox` class in the Click event for Button1, which displays the selected name in a message box.

Listing 24-5 Show Selected Name

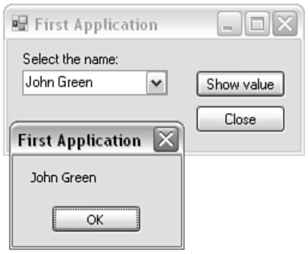
```
Private Sub Button1_Click(ByVal sender As System.Object, _
                                ByVal e As System.EventArgs) _
                                Handles Button1.Click

        'Make sure that a name has been selected.
        If Me.ComboBox1.SelectedIndex <> -1 Then

            'Show the selected value.
            MessageBox.Show( _
                    text:=Me.ComboBox1.SelectedValue.ToString(), _
                    caption:="First Application")
        End If

End Sub
```

The final piece of the puzzle is to add a command to close (unload) the Windows Form in the Button2 Click event. Listing 24-6 shows the required code.

Listing 24-6 Unload the Windows Form

```
Private Sub Button2_Click(ByVal sender As System.Object, _
                                ByVal e As System.EventArgs) _
                                Handles Button2.Click
        Me.Close()
End Sub
```

To begin testing the application, we just have to press the F5 key. Figure 24-8 shows the First Application in action after we select a value in the combo box and then click the Show value button.



John Green

First Application

John Green

FIGURE 24-8 Our first application in action

Whenever we execute the application in the debugger, the VS IDE creates a number of new files, including an executable file for our application. These files are located in the *..\First Application\bin\Debug* folder. A working example of this solution can be found on the companion CD in the *\Concepts\Ch24 - Excel & VB.NET\First Application* folder. If you just want to run the application without opening it in Visual Studio, the First Application executable file can be found in the *\Concepts\Ch24 - Excel & VB.NET\First Application\First Application\bin\Debug* folder on the CD.

Structured Exception Handling

When an unexpected condition occurs in managed code, the CLR creates a special object called an **exception**. The exception object contains properties and methods that give detailed information about the unexpected condition. Because we deal with exceptions rather than errors in .NET development, we use the expression exception handling rather than error handling.

Exception handling covers the techniques used to detect exceptions and take appropriate actions after they are detected. **Structured exception handling (SEH)**, is the term used to describe how we implement exception handling in managed code. Although it is possible to use the Classic VB error handling approach in VB.NET, we strongly encourage the use of SEH because it gives us much better options for dealing with exceptions. SEH consists of the following building blocks:

- **`Try`—**We place the code we want to execute in this block. This code may create one or more exceptions.
- **`Catch`—**In this block we place the code that handles the exceptions. It is possible to place several `Catch` blocks within the same structure to handle different types of exceptions. `Catch` blocks are optional.
- **`Finally`—**Code placed in this block always is executed, which makes this block a perfect place for code to clean up and release references to objects like COM objects and ADO.NET objects. This block is also optional.
- **`End Try`—**Ends the SEH structure.

Listing 24-7 shows the skeleton structure of SEH in code. When we enter a `Try` statement in a code module, the VS IDE automatically adds the `Catch` block and `End Try` statement. The `Finally` block must be typed manually.

Listing 24-7 The Building Blocks of SEH

```
Private Function iDiscount(ByVal iPrice As Integer) As Integer

        Try
            'Do the calculation here.

        Catch ex As Exception
            'In case of any unexpected scenarios take
            'some action here, like a message to the user.

        End Try
End Function
```

Most of the namespaces in the .NET Framework class library include their own specific exception classes, which make it possible to catch them in separate `Catch` blocks. All built-in exception classes extend the built-in `System.Exception` class. `Catch` blocks are executed (or tested for execution) in the order in which they are coded. .NET works its way through the `Catch` blocks trying to find a matching exception type. Therefore the preferred approach is to implement the `Catch` blocks with more specific exception types first, followed by the `Catch` blocks with the more generic exception types. Listing 24-8 shows an example using two `Catch` blocks.

Listing 24-8 Using Several Catch Blocks and the Finally Block

```
Try
    frmSaveFile = New SaveFileDialog

    With frmSaveFile
        .Filter = "XML File|*.xml"
        .Title = "Save report to XML file"
        .FileName = sFileName
    End With

    dtTable.WriteXml(fileName:=sFileName)

    dtTable.WriteXmlSchema( _
           fileName:=Strings.Left(sFileName, _
         Len(sFileName) - 4) & ".xsd")

Catch XMLexc As Xml.XmlException

    MessageBox.Show(text:=sMESSAGENOTSAVEDXML, _
                    caption:=swsCaption, _
                    buttons:=MessageBoxButtons.OK, _
                    icon:=MessageBoxIcon.Stop)

Catch COMExc As COMException

    MessageBox.Show(text:= _
                    sERROR_MESSAGE & _
                    sERROR_MESSAGE_EXCEL, _
                    caption:=swsCaption, _
                    buttons:=MessageBoxButtons.OK, _
                    icon:=MessageBoxIcon.Stop)

Catch Generalexc As Exception
```

```
    MessageBox.Show(text:=sMESSAGENOTSAVEDGENERAL, _
                      caption:=swsCaption, _
                      buttons:=MessageBoxButtons.OK, _
                      icon:=MessageBoxIcon.Stop)

Finally

    frmSaveFile.Dispose()
    frmSaveFile = Nothing

End Try
```

The first `Catch` block handles any `XmlException` exceptions. The second block catches COM exceptions that might occur when working with COM servers like Excel. The final `Catch` block is generic and handles all other exceptions. The example also shows how we can use the `Finally` block to release an object. Listing 24-8 also shows how to use custom error messages to respond to each exception type.

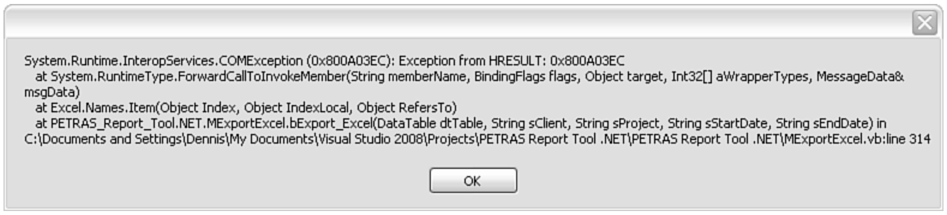
During development we need to see the underlying technical details for all exceptions. In Listing 24-9 the previously customized end user messages have been replaced with the exception object and its method `ToString` in each `Catch` block. The `ToString` method gives a textual summary of the exception. You can also use the `GetBaseException` method to return the first exception in the chain.

Listing 24-9 Displaying Exception Descriptions

```
Catch XMLexc As Xml.XmlException

    MessageBox.Show(XMLexc.ToString())

Catch COMExc As COMException

    MessageBox.Show(COMExc.ToString())
    MessageBox.Show(COMExc.ErrorCode.ToString())

Catch Generalexc As Exception

    MessageBox.Show(Generalexc.ToString())
```

When VB.NET receives an exception from a COM server like Excel, it checks the **COM exception** code and tries to map that code to one of the

.NET exceptions classes. If this fails, which is the most common outcome, VB.NET throws a large and mostly unhelpful HRESULT message like the one shown in Figure 24-9.



```
System.Runtime.InteropServices.COMException (0x800A03EC): Exception from HRESULT: 0x800A03EC
   at System.RuntimeType.ForwardCallToInvokeMember(String memberName, BindingFlags flags, Object target, Int32[] aWrapperTypes, MessageData&
msgData)
   at Excel.Names.Item(Object Index, Object IndexLocal, Object RefersTo)
   at PETRAS_Report_Tool.NET.MExportExcel.bExport_Excel(DataTable dtTable, String sClient, String sProject, String sStartDate, String sEndDate) in
C:\Documents and Settings\Dennis\My Documents\Visual Studio 2008\Projects\PETRAS Report Tool .NET\PETRAS Report Tool .NET\MExportExcel.vb:line 314
```

FIGURE 24-9 The COM exception message

The line of code that generates this message is the first `MessageBox.Show` line under the COM exception block in Listing 24-9. COM exceptions are wrapped into generic `COMException` objects when .NET does not have a matching exception class for the HRESULT error generated by a COM component.

In SEH, it is possible to exit a `Try` block with the `Exit Try` statement. This statement can be placed either in the `Try` block or in any `Catch` block. Any code in a `Finally` block is still executed after the `Exit Try` statement.

Another option is to use nested `Try` structures. A nested SEH can be added either to the `Try` block or to a `Catch` block. When using nested exception handlers the **InnerException** property of the exception object becomes very important. It helps us determine the cause of the nested exception and allows us to obtain the chain of exceptions that led to that exception.

We can use the **Throw** statement to communicate exceptions to the calling code. `Throw` is usually used within a `Catch` block only if the exception is to be bubbled up the call stack. A `Throw` statement causes code execution to be intentionally interrupted. The `Throw` statement also allows us to create our own exceptions, but this topic is beyond the scope of this chapter.

Modules and Methods, Scope and Visibility

When we make a declaration at the module level (module here stands for module, class, or structure), the access level we choose determines the scope of the thing being declared. In VB.NET we can use the keywords `Public` and `Private`, which have the same scopes as in Classic VB, but VB.NET also provides the following additional keywords to specify module scope and visibility:

- **Friend—**A data member or **method** (function or subroutine) declared with the Friend modifier can be accessed from any part of the program containing the declaration. This is not a new keyword, as it is also available in Classic VB. However, if we do not explicitly include a scope in our declaration, then the default scope is Friend in VB.NET, while in Classic VB the default scope is Public.
- **Protected—**Data members or methods declared with Protected scope are only accessible from the module itself or from derived classes.
- **Protected Friend—**This scope is equivalent to the union of Protected and Friend access. A data member or method declared as Protected Friend is accessible from anywhere in the program in which the declaration occurs, or from any derived class containing the declaration.

Declare Variables and Assign Values

In VB.NET, we declare local variables using the keyword `Dim`, module-level variables using the keyword `Private`, solution-level variables using the keyword `Friend`, and public variables using the keyword `Public`. All .NET programming languages provide the option to declare variables and assign values to them at the same time.

The first two lines in Listing 24-10 show how we can declare variables and initialize them with values using one line of code. The third line creates three String variables without assigning any values to them. Since they don't have assigned values, these String objects are marked as unused local variables by the VS IDE. This is a result of the `Option Strict` setting being on. Good coding practice in .NET says that we should always assign known values to variables, even if they initially will not have any "real" values. Lines 4 through 6 show how we can achieve this in practice.

Listing 24-10 Declare Variables and Assign Values to Them

```
1  Dim sTitle As String = "PETRAS Report Tool"
2  Dim iPrice As Integer = 100

3  Dim sAddress, sCity, sCountry As String

4  Dim sName = String.Empty
5  Dim bReportStatus = Nothing
6  Dim iNumberOfRecords As Integer = Nothing

7  Dim iNumberOfColumns As Integer = dtTable.Columns.Count - 5
8  Dim iNumberOfRows As Integer = dtTable.Rows.Count - 1
9  Dim obDataArr(iNumberOfRows, iNumberOfColumns) As Object
```

Lines 7 and 8 in Listing 24-10 contain two variables that hold the number of columns and rows of a **DataTable** (an ADO.NET object covered later in this chapter). These two variables are then used as parameters to dimension the array of data type Object in line 9. The data type Object is the VB.NET counterpart to the data type Variant in Classic VB. An Object array behaves in roughly the same manner as a Variant array.

VB.NET also offers the ability to declare variables anywhere in the code. Listing 24-11 shows an example where we have declared a variable within a `Try` block in conjunction with a `For...Next` loop.

Listing 24-11 Block Scope Variable Declaration

```
Try
    For iCountRows As Integer = 0 To 9
        'Do the iteration.
    Next iCountRows

Catch ex As Exception

     MessageBox.Show(ex.ToString())

End Try
```

Block scope can also be achieved by declaring variables within `With...End With` blocks, `For...Next` blocks, and `Do...Loop` blocks. In Listing 24-12 we show a variable that is declared in a `Do...Loop`.

Listing 24-12 Block Scope within a Do...Loop

```
'Declaration of a variable with
'a block scope of Do...Loop.
Do
    Dim iMonth As Integer = 1
    'Other code goes here...
Loop
```

However, declaring variables using this method may cause unexpected problems. This is because the scope of variables declared in this manner is limited to the block in which they are declared. This means we cannot access these variables or use them outside that block. Code that uses this method can also be more difficult to debug and maintain. In general we should avoid this approach. Good coding practice suggests that all variables used within a method should be declared at the beginning of that method.

Creating New Instances of Objects

We can create new instances of objects in VB.NET using the same techniques as in Classic VB. The only difference is that we do not use the `Set` keyword in VB.NET. Listing 24-13 shows two methods of creating objects in VB.NET. The `Nothing` keyword is a way of telling the system that the variable does not currently have any value but still may use memory.

Listing 24-13 Declare and Instantiate Objects

```
'The classic approach.
Dim frmSaveDialog As SaveFileDialog = Nothing
frmSaveDialog = New SaveFileDialog

'.NET approach.
Dim frmSaveDialog As New SaveFileDialog
```

The .NET approach is singled out in the second example in Listing 24-13, which shows that we declare and set the variable to a new instance of the SaveFileDialog class with one line of code. Although the .NET approach may look attractive, we still recommend using the classic approach. This is also outlined as the best practice in Chapter 3.

Using the .NET approach can cause unwanted exceptions because of the block scoping of variables. For example, if we create a new instance of the SaveFileDialog component and we want to trap any exceptions that may occur (or we want to throw an exception), block scoping of the variable itself causes an exception. This is demonstrated in Listing 24-14, where we have declared and instantiated the `frmSaveDialog` object variable in the `Try` block. However, because the scope of this variable is limited to the `Try` block, the VS.IDE displays a compile error for the two lines of code inside the `Finally` block.

Listing 24-14 Using the .NET Approach

```
    Sub Show_Save_Dialog()

        Try
        Dim frmSaveDialog As New SaveFileDialog
        frmSaveDialog.ShowDialog()


    Catch ex As Exception

    Finally

        frmSaveDialog.Dispose()
        frmSaveDialog = Nothing

    End Try

End Sub
```

To correct this problem, we modify the code to use the classic approach as shown in Listing 24-15. The `frmSaveDialog` variable can now be seen throughout the `Try` block, and it traps any exceptions that may occur.

Listing 24-15 Using the Classic Approach

```
Sub Show_Save_Dialog()

    Dim frmSaveDialog As SaveFileDialog = Nothing

    Try

        frmSaveDialog = New SaveFileDialog
        frmSaveDialog.ShowDialog()


    Catch ex As Exception

        MessageBox.Show(ex.ToString())

    Finally

        frmSaveDialog.Dispose()
        frmSaveDialog = Nothing

    End Try

End Sub
```

Using ByVal or ByRef

Unlike Classic VB, procedure arguments in VB.NET are passed ByVal by default ***not*** ByRef. If we do not explicitly specify procedure arguments as either ByVal or ByRef, the VB.NET default is ByVal. However, good practice states that we should always explicitly specify the keyword we want to use.

Using Wizards in VB.NET

Compared to the wizards in Classic VB, the wizards in VB.NET have been significantly improved. New wizards have also been added to the VS IDE. The advantage of using a wizard is that we get the desired result in a fast and reliable way without needing to have a deep understanding of the process. The wizard takes care of the details. The disadvantage of using a wizard is that the wizard works in "black box" mode, which means we do not have much control over the process. Developing real-world applications requires you to be in control and to understand your solutions inside and out. You can explore the wizards in the VS IDE, but for any non-trivial solution you should avoid them.

Data Types in VB.NET

Compared with Classic VB, some data types are new in VB.NET. Table 24-2 shows most of the VB.NET data types but not all of them.

Table 24-2 Data Types in VB.NET

Data Type	Size	Values
Boolean	2 bytes	True or False.
Short	2 bytes	-32,768 to 32,768.
Integer	4 bytes	-2,147,483,648 to 2,147,483,648.
Long	8 bytes	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,808.
Decimal	16 bytes	It provides the greatest number of significant digits for a number.
Double	8 bytes	It provides the largest and smallest possible magnitudes for a number.

Table 24-2 Data Types in VB.NET

Data Type	Size	Values
String	Variable	A string can hold 0 to 2 billion Unicode characters.
Date	8 bytes	January 1, 0001 0:0:00 to December 31,9999 11:59:59.
Object	4 bytes	Point to any type of data.

As we can see in Table 24-2, the data type **Short** includes the interval -32,768 to 32,768, and the **Integer** data type now covers a much greater interval than it does in Classic VB. The Currency data type is no longer available in VB.NET. It has been replaced by the new **Decimal** data type, which can handle more digits on both sides of the decimal point. The Byte data type from Classic VB has no counterpart in VB.NET. The data type **Object** is the universal data type in VB.NET, taking the place of the Variant data type in Classic VB.

String Manipulation

As previously mentioned, whenever a new .NET solution is created the namespace `Microsoft.VisualBasic` is included by default. This provides access to the .NET versions of the well-known string functions in Classic VB. The .NET Framework also provides us with a `System.String` class to manipulate strings. However, using the old functions has no negative impact on solution performance, so using the old familiar functions is completely acceptable.

Using Arrays in VB.NET

The .NET Framework provides us with powerful new options for creating and using arrays and collections in VB.NET. There are two basic kinds of VB.NET arrays. Arrays that we declare as array variables of a specific data type by using parentheses after the variable name are normal arrays. We can also use the Array class, which provides us with a new array data type that offers methods for managing items in arrays as well manipulating arrays. Arrays in VB.NET inherit from the Array class in the System namespace, so methods of the Array class can also be used with normal arrays.

In this section, we discuss normal arrays along with some methods of the Array class. In VB.NET, all arrays are zero-based. This is important to keep in mind, especially when working with Excel objects or Classic VB code that may have 1-based arrays. We already showed one way to use an array in Listing 24-3, where we used an array to populate a combo box control. In Listing 24-16 we use the same approach to populate a list box control and then add the selected items to an array.

Listing 24-16 Populate an Array with Selected Items from a List Box

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

'Make sure that at least one item is selected.
If Me.ListBox1.SelectedIndex <> -1 Then

   'Grab the number of selected items.
   Dim iCountSelectedItems As Integer = _
       Me.ListBox1.SelectedItems.Count - 1

   'Declare and dimension the one-dimensional array.
   Dim sArrSelectedItems(iCountSelectedItems) As String

   'Populate the array.
   For iCountSelectedItems = 0 To iCountSelectedItems
      sArrSelectedItems(iCountSelectedItems) = _
      Me.ListBox1.SelectedItems(iCountSelectedItems).ToString()
   Next iCountSelectedItems

  'Show the number of items in the array.
   MessageBox.Show(CStr(sArrSelectedItems.GetLength(0)))

   'Show the lower bound of the array.
   MessageBox.Show(CStr(sArrSelectedItems.GetLowerBound(0)))

   'Show the upper bound of the array.
   MessageBox.Show(CStr(sArrSelectedItems.GetUpperBound(0)))

   'Iterate through the array and display each value.
   For iCountSelectedItems = sArrSelectedItems.GetLowerBound(0) _
                            To sArrSelectedItems.GetUpperBound(0)
       MessageBox.Show(text:= _
                  sArrSelectedItems(iCountSelectedItems).ToString())
   Next iCountSelectedItems

End If

End Sub
```

When working with arrays we should always specify which dimension we are targeting. Since we are working with a one-dimensional array in this example, the dimension we are targeting is zero.

One of the more resource-intensive processes in VB development is redimensioning arrays, so we should always look for ways to reduce or eliminate this process. Listing 24-16 shows how VB.NET allows us to do this easily. We first retrieve the number of selected list items and then declare and dimension the array all at once. Note that in Listing 24-16 we use the **GetLowerBound** and **GetUpperBound** methods to return the lower bound and upper bound index values for the array. Both these methods are part of the Array class. In some scenarios we may not know the bounds for an array initially, but we can get the necessary information later. Listing 24-17 shows how we can initialize an array after declaring it.

Listing 24-17 Declare an Array and Initialize It Later

```
Dim iNumberOfHouses() As Integer
...
...
iNumberOfHouses = New Integer() {10, 15, 20}
```

Listing 24-16 shows one way to iterate an array, but we could actually enumerate it as shown in Listing 24-18.

Listing 24-18 Enumerating an Array

```
Dim iNumberOfHouses() As Integer = {10, 15, 20}
Dim iItem As Integer

For Each iItem In iNumberOfHouses
    Debug.WriteLine(iItem)
Next iItem
```

The Array class also provides methods that allow us to manipulate the items in different ways. Among the more common actions we might want to perform on an array are reversing the order of items in the array, sorting the array, removing items from the array, returning specific array items, and copying items from one array to another. Listing 24-19 shows how to perform these operations using methods of the Array class.

Listing 24-19 Methods of the Array Object

```
Dim sArrProjects() As String = _
                 {"Upgrade","Investment", "Maintenance"}

Array.Reverse(sArrProjects)

Array.Sort(sArrProjects)

Array.Clear(sArrProjects, 0, 1)

Dim sItem As String = sArrProjects.GetValue(1).ToString()

Dim sArrProjectsCopy(sArrProjects.GetLength(0)) As String

Array.Copy(sArrProjects, sArrProjectsCopy, _
                    sArrProjects.GetLength(0))
```

The first example shows how to reverse the order of the items in an array. The second example sorts the array in ascending order. The third example shows how to delete the first item from an array. Note that deleting an item from an array in this manner does not resize the array or move any of the other items into new positions in the array.

To get a specific item value from an array you use the **GetValue** method, as shown in the fourth example. And as the final example shows, we can even copy one array to another using the Copy method. The last argument of this method allows us to specify the number of items to be copied. This can be a good alternative to the redimension approach when resizing an array. In this example we copy all items from the first array into the second array.

Next we demonstrate how to search for a value in an array using the **BinarySearch** method. This method is useful when you want to determine whether a specific value exists in an array. To use this method the items in the array must be sorted. The result of executing the BinarySearch method is an integer that represents the index number of the value you are searching for within the array. If the result is -1 the value you are searching for does not exist. If the value you are searching for exists more than once within the array, the index number of the last occurrence is returned.

Listing 24-20 shows how to use the BinarySearch method to locate the index number of an item in an array. There are also several other methods of the array object that allow us to find specific items and work with them in various ways.

Listing 24-20 The BinarySearch Method

```
Dim sArrProjects() As String = _
    {"Upgrade", "Investment", "Maintenance"}

Dim sSearchedValue As String = "Investment"

Array.Sort(sArrProjects)

Dim iSearchedIndex As Integer = _
    Array.BinarySearch(sArrProjects, sSearchedValue)

MessageBox.Show(CStr(iSearchedIndex))
```

A good alternative to the normal array is the **ArrayList** class, which is part of the `System.Collection` namespace. By using this class we can dynamically increase a list, hold several different data types in one list, manipulate the elements in a list, and manipulate ranges of elements in one operation. The ArrayList is something of a hybrid between the Array and Collection objects. In Listing 24-21 we demonstrate the use of an ArrayList object.

Listing 24-21 Working with the ArrayList Object

```
Dim Arrlst As New ArrayList(7)
Dim oArrlstObject As Object = Nothing

Debug.Print(Arrlst.Capacity.ToString())

With Arrlst
      .Add("Dennis")
      .Add(True)
      .Add(12)
End With

Debug.Print(Arrlst(1).GetType.ToString())

Dim sNames() As String = {"Rob Bovey", _
                          "Stephen Bullen", _
                          "John Green", _
                          "Dennis Wallentin"}

Arrlst.AddRange(sNames)
Arrlst.RemoveRange(0, 3)

Arrlst.TrimToSize()

Debug.Print(Arrlst.Capacity.ToString())

For Each oArrlstObject In Arrlst
       Debug.Print(oArrlstObject.ToString())
Next oArrlstObject

Me.CheckedListBox1.DataSource = Arrlst
```

We first create a new ArrayList object and dimension it to hold seven items. Expanding an ArrayList is a resource-intensive process, so we want to try and create it with the capacity to hold as many items as we will need. The first debug print command gives us the current capacity of the ArrayList. We then populate the ArrayList object with items that represent different data types, in this case a string value, a boolean value, and an integer value. To verify that the ArrayList actually holds different data types we print the data type of the second item to the Immediate window using the **GetType** method.

Next we add a range of values to the ArrayList using the **AddRange** method. Our ArrayList already has the capacity to hold these new items, but if an ArrayList does not have sufficient capacity to hold the number of items being added it automatically expands itself. The **RemoveRange** method enables us to remove several items at once, so next we use this method to remove the first three items we added to it. At this stage the ArrayList object still has a capacity of seven items, but since we no longer need them all we resize it by using the **TrimToSize** method. Using the debug print command to check the capacity of the ArrayList after resizing it should show a capacity of four items. Just to check which values the ArrayList now holds we iterate over all its items using a `For...Each` loop. Finally, the collection of items in the ArrayList is added to a **CheckedBoxList** control.

In addition to the ArrayList, the .NET Framework provides additional data structures like **Stack** and **Queue**. The Stack class is a data structure that allows adding and removing objects from one position only. This position is referred to as the “Top” of the stack. The last object placed on the stack is the first one to be removed. This is a **Last In First Out** (**LIFO**) data access method. The Queue class is a data structure that allows us to add objects to the back and remove objects from the front. This is a **First in First Out** (**FIFO**) data access method.

Debugging

The most important task in development is to debug non-trivial solutions. The VS IDE offers a large number of tools to assist you in this task. Depending on the complexity of the solution, debugging can be quite difficult and time consuming. One of the best features of the VS IDE is that we can interact with it during debugging sessions.

Selecting the *Debug* menu reveals the available tools and options. As we can see, most of the commands and windows are familiar from Classic VB. During the debugging process, and while in break mode, additional tools become available as shown in Figure 24-10. Although a detailed walkthrough is beyond the scope of this chapter, we focus on the most important new and updated debugging tools that the VS IDE provides. See the Chapter 16, "VBA Debugging," for a more detailed discussion of the debugging process.

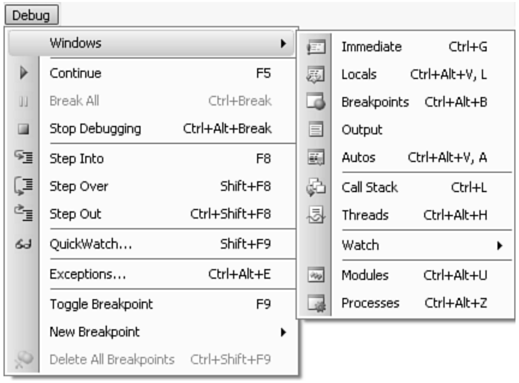
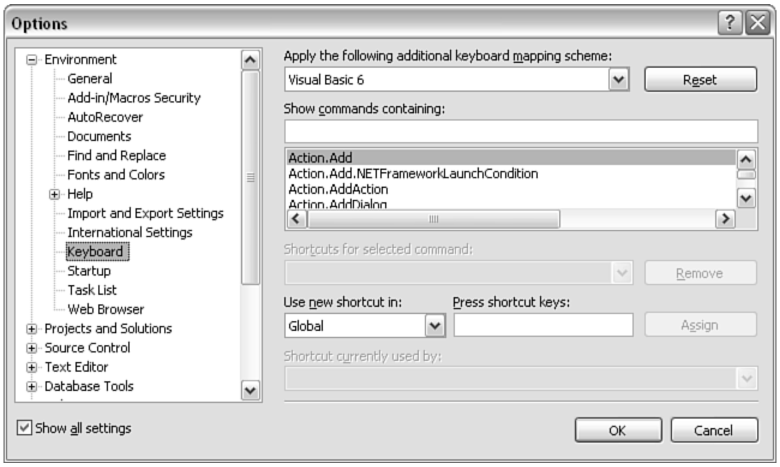


FIGURE 24-10 Debugging tools available in break mode

Set Keyboard Shortcuts

Before we start to explore the many tools for debugging, we customize our keyboard shortcuts. Select *Tools > Options...* from the menu to display the

Options dialog. In the Options dialog tree view, select the *Keyboard* section under *Environment*, as shown in Figure 24-11.



Apply the following additional keyboard mapping scheme: Visual Basic 6

Use new shortcut in: Global

☑ Show all settings

FIGURE 24-11 Keyboard shortcuts

This section allows us to set the mapping scheme for keyboard shortcuts. Changing the mapping scheme to *Visual Basic 6.0* provides access to all the well-known VB6 keyboard shortcuts in the VS IDE. This change is global, meaning it will be applied for all VB.NET solutions in the VS IDE. The keyboard shortcuts mentioned in the rest of this chapter assume this setting has been made in your environment.

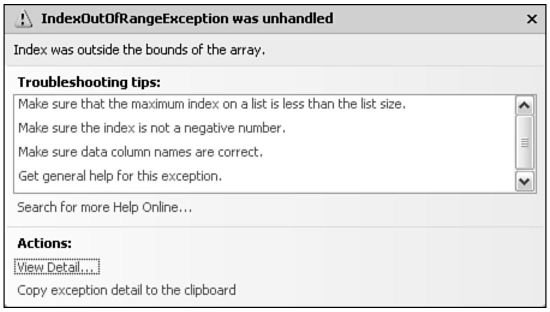
Enable Unmanaged Code Debugging

If we do a lot of interoperability development, that is, calls to COM objects, the option *Enable unmanaged code debugging* gives us the possibility to debug the native code. Select *Project > [Solution Name] Properties...* from the menu to display the Properties window; then select the *Debug* tab and check this option.

The Exception Assistant

Whenever a runtime exception is thrown, the Exception Assistant highlights the line of code that caused the exception and displays a dialog with suggestions on how to solve the problem. Figure 24-12 shows the Exception Assistant in action.

The Exception Assistant attempts to provide context-sensitive help related to the exception, and it allows the developer to perform certain



IndexOutOfRangeException was unhandled

Index was outside the bounds of the array.

Troubleshooting tips:

Make sure that the maximum index on a list is less than the list size.

Make sure the index is not a negative number.

Make sure data column names are correct.

Get general help for this exception.

Search for more Help Online...

Actions:

View Detail...

Copy exception detail to the clipboard

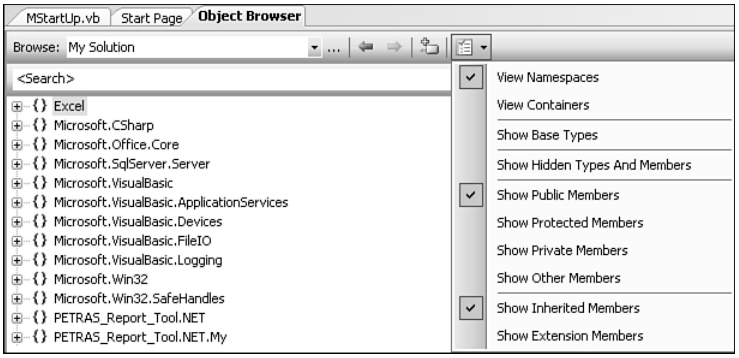
FIGURE 24-12 The Exception Assistant

actions, such as viewing details of the exception and copying exception information to the Clipboard. For COM exceptions, however, the information provided by the Exception Assistant is of limited value.

We can provide troubleshooting tips for our own exception types by creating an XML file containing the information in the correct ExceptionAssistantContent directory under C:\Program Files\Microsoft Visual Studio 9.0\Common7\IDE\ExceptionAssistantContent.

The Object Browser (F2)

The Object Browser is one of the most valuable development resources. The VS IDE ships with a modern Object Browser that can be customized by selecting the *Object Browser Settings* icon on its toolbar, as shown in Figure 24-13. We can also add components to the *Custom Component Set Browsing scope* by selecting the *Edit Custom Component Set* button directly to the right of the *Browse* drop-down in the Object Browser toolbar.



Excel
Microsoft.CSharp
Microsoft.Office.Core
Microsoft.SqlServer.Server
Microsoft.VisualBasic
Microsoft.VisualBasic.ApplicationServices
Microsoft.VisualBasic.Devices
Microsoft.VisualBasic.FileIO
Microsoft.VisualBasic.Logging
Microsoft.Win32
Microsoft.Win32.SafeHandles
PETRAS_Report_Tool.NET
PETRAS_Report_Tool.NET.My

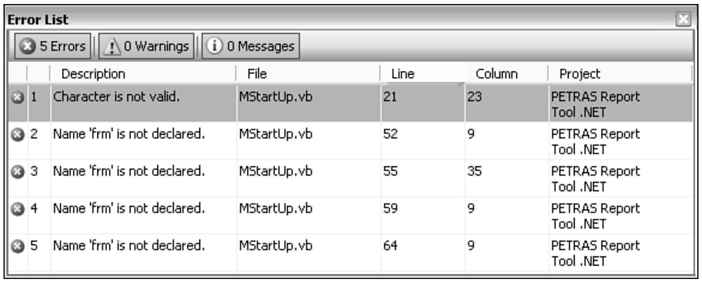
☑ View Namespaces
☐ View Containers
☐ Show Base Types
☐ Show Hidden Types And Members
☑ Show Public Members
☐ Show Protected Members
☐ Show Private Members
☐ Show Other Members
☑ Show Inherited Members
☐ Show Extension Members

FIGURE 24-13 The Object Browser

The *Browse* drop-down is used to limit the scope of items displayed in the Object Browser. One of the selections available is to browse *My Solution*, as shown in Figure 24-13. This option allows us to browse the objects in our solution as well as any outside namespaces the solution references.

The Error List Window (Ctrl+W Ctrl+E)

The **Error List window** shows errors, warnings and other messages that result from attempting to compile the active project. It detects most common syntax and deployment errors. Figure 24-14 shows an example Error List window displaying some errors. Double-clicking on an item in the list takes you to the module and line of code it refers to.



		Description	File	Line	Column	Project
⊗	1	Character is not valid.	MStartUp.vb	21	23	PETRAS Report Tool .NET
⊗	2	Name 'frm' is not declared.	MStartUp.vb	52	9	PETRAS Report Tool .NET
⊗	3	Name 'frm' is not declared.	MStartUp.vb	55	35	PETRAS Report Tool .NET
⊗	4	Name 'frm' is not declared.	MStartUp.vb	59	9	PETRAS Report Tool .NET
⊗	5	Name 'frm' is not declared.	MStartUp.vb	64	9	PETRAS Report Tool .NET

FIGURE 24-14 The Error List window

The keyboard shortcut to display the Error List window requires two steps, Ctrl+W followed by Ctrl+E. It may feel a bit odd to use two instructions to access a feature, but this reflects how many features the VS IDE contains.

The Command Window (Ctrl+Alt+A) and Immediate Window (Ctrl+G)

The **Command window** and Immediate window overlap each other to some degree, but they actually have two different tasks to accomplish. The Command window allows you to execute VS IDE commands instead of going through the menus and toolbars. It can also execute commands to open other windows.

Suppose we have started a debugging session and we are running in break mode. If we enter the command shown in Listing 24-22 into the Command window the variable bExport will be added to the Watch window.

Listing 24-22 Add a Watch Using the Command Window

```
>Debug.AddWatch bExport
```

If we want to see all the command aliases, or command shortcuts, defined by the VS IDE, we can run the command >*Alias* in the Command window to produce a list.

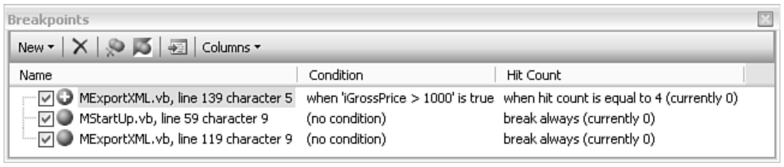
The Immediate window in the VS IDE behaves much like its counterpart in Classic VB. We can assign variables, run procedures, and invoke methods in standard VB.NET syntax in the Immediate window.

The Output Window (Ctrl+Alt+O)

The **Output window** displays compilation results and the text output from several tools such as Debug and Trace. The *Show output from:* drop-down in the toolbar allows you to show the output from either the debug or the build process. It is also possible to save the output to a text file by clicking anywhere inside the Output window and then using the keyboard shortcut Ctrl+S.

Break Points (Ctrl+Alt+B)

To insert a new break point, use the keyboard shortcut Ctrl+B. Compared with its older sibling in Classic VB, the break points feature has been improved significantly in VS.NET. First, VS.NET provides a **Breakpoints window** that displays the location and settings for all break points in the solution, as shown in Figure 24-15.



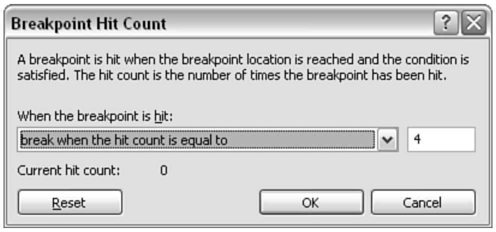
Name	Condition	Hit Count
☑ MExportXML.vb, line 139 character 5	when 'iGrossPrice > 1000' is true	when hit count is equal to 4 (currently 0)
☑ MStartUp.vb, line 59 character 9	(no condition)	break always (currently 0)
☑ MExportXML.vb, line 119 character 9	(no condition)	break always (currently 0)

FIGURE 24-15 The Breakpoints window

Second, we can set conditions for a break point by right-clicking on that break point and selecting *Condition...* from the shortcut menu. In Figure 24-15 we set a condition for the first break point. When the break point is reached, this condition is evaluated to determine whether it is true or false.

If the condition is true the break point is triggered; otherwise, the break point is skipped.

Third, we can add a **hit count** for a break point by right-clicking on that break point and selecting *Hit Count...* from the shortcut menu. This provides us with an additional parameter to control whether code execution should stop at break points. Figure 24-16 shows us defining a hit count for our first break point in the Breakpoint Hit Count dialog.



A breakpoint is hit when the breakpoint location is reached and the condition is satisfied. The hit count is the number of times the breakpoint has been hit.

When the breakpoint is hit:

break when the hit count is equal to 4

Current hit count: 0

FIGURE 24-16 Defining a break point hit count

The Call Stack (Ctrl+L)

The Call Stack window displays the method calls that are currently on the stack. It is a useful debugging tool because it allows you to see the specific execution path that led to the current position in your code.

The Quick Watch and Watch Windows

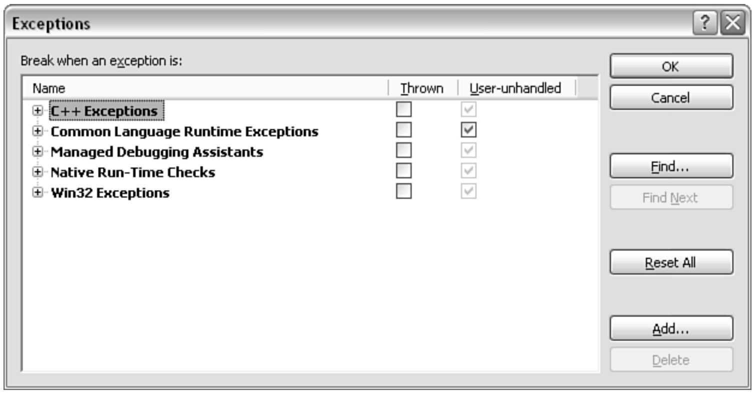
Once our code is in break mode we have access to the Quick Watch and Watch windows. The Watch window, accessed by selecting the *Debug* > *Windows* > *Watch* menu while in break mode, provides four different Watch tabs. It is easy to add watches. You can drag and drop an object or expression onto the Watch window or select the object or expression in the code editor, right-click on it, and choose *Add Watch* from the shortcut menu. To delete a watch, select it in the Watch window, right-click on it, and choose *Delete Watch* from the shortcut menu. You can add as many different watches as you want. To access one of the Watch windows during a debugging session, press Ctrl+Alt+W followed by a digit between 1 and 4.

Quick Watch works the same way as the Watch window except that it can only handle one watch variable at the time.

Exceptions (Ctrl+Alt+E)

The Exceptions dialog is an advanced debugging tool that allows us to specify what types of exceptions we want VS.NET to throw during debugging.

The debugger stops whenever the selected type of exception occurs. Figure 24-17 shows this dialog.



Name	Thrown	User-unhandled
⊞ C++ Exceptions	☐	☑
⊞ Common Language Runtime Exceptions	☐	☑
⊞ Managed Debugging Assistants	☐	☑
⊞ Native Run-Time Checks	☐	☑
⊞ Win32 Exceptions	☐	☑

FIGURE 24-17 The Exceptions dialog

The *Thrown* option causes the debugger to break unconditionally when the specified exception type occurs. If we check the *Thrown* option for the **Common Language Runtime Exceptions**, we ensure that when a Common Language Runtime exception is thrown it breaks into the debugger, overriding any custom SEH we may have defined. The *User-unhandled* option causes the debugger to stop for the specified exception type only if no error handler is active when the exception occurs.

We can also configure specific exception types below the top-level namespaces by clicking on the plus sign (+) to the left of a namespace. This expands the namespace node to show all exceptions within the namespace that can be configured.

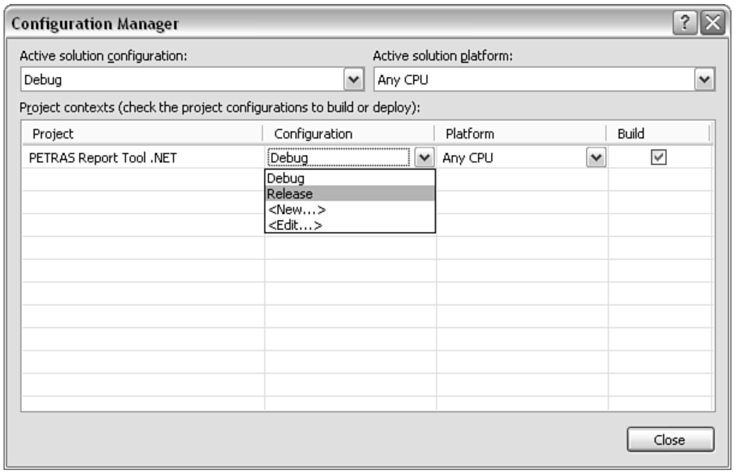
Conditional Compilation Constants

Chapter 16 introduced the concept of conditional compilation constants, so in this section we only cover conditional compilation topics that are specific to the .NET platform.

VB.NET provides several predefined conditional compilation constants, including the Boolean constant DEBUG. When DEBUG is set to true we have a **debug build**, and when it is set to false we have a **release build**. When compiling a release build we do not need to manually remove any debugging information. VS.NET handles this automatically when the DEBUG constant is set to false. Debugging information is also ignored when running a release build in the VS IDE. To compile a release build we need to use the Configuration Manager. Verify that the Configuration Manager is available in the following manner:

1. Select the *Tools > Options...* menu from the VS IDE.
2. Select *Projects and Solutions* from the tree view in the Options dialog.
3. Check the *Show advanced build configurations* check box.
4. Click the OK button to close the Options dialog.

We can then access the Configuration Manager by selecting *Build > Configuration Manager...* from the VS IDE menu, as shown in Figure 24-18. By changing the configuration we can switch between debug and the release builds. We can also use the Configuration Manager to specify which platform to target.



Configuration Manager

Active solution configuration: Debug

Active solution platform: Any CPU

Project contexts (check the project configurations to build or deploy):

Project	Configuration	Platform	Build
PETRAS Report Tool .NET	Debug	Any CPU	☑

FIGURE 24-18 The Configuration Manager

We can execute code conditionally based on the value of the DEBUG constant as shown in Listing 24-23.

Listing 24-23 Using DEBUG in Code

```
#If DEBUG then
'Do some evaluation.
#End If
```

Using Assertions

The Debug.Assert method is used exactly the same way in the VS IDE as it is in Classic VB. Chapter 16 already covered the use of this method, so we do not discuss it further here.

Useful Development Tools

The VS IDE ships with a large number of useful development tools. Although it is beyond the scope of this chapter to discuss them all, we cover some of the most important tools in this section.

Code Region

The **Code Region** feature allows us to expand and collapse different sections, or regions, in our code modules. We can use this feature to create logical groups of methods that expand and collapse together. We can then collapse all regions in a code module that are unrelated to the one we are working with.

To create a region, we enter #Region followed by the name of the region in double quotes on a blank line above where the region should start. We then move to the next blank line below the code we want included in the region and enter #End Region (or select it from the IntelliSense list when we are prompted). Listing 24-24 shows an example of a code region.

Listing 24-24 A Code Region

```
#Region "Export data to Excel"
'Many lines of code here
#End Region
```

The Code Snippets Manager (Ctrl+K Ctrl+B)

Code snippets are small, reusable pieces of code. They are stored in a snippet library and managed using the **Code Snippets Manager**. The VS IDE includes a large number of code snippets already written and stored in the Code Snippets Manager. Code snippets are particularly easy to use because they are exposed as part of the VS IDE IntelliSense feature. Code snippets are stored in text files in XML format. This makes it easy to use them on other computers as well as to share them with other developers.

You can insert a code snippet into your code module in the following manner:

1. Place the cursor at the position where you want to insert the code snippet.
2. Right-click and select *Insert Snippet...* from the shortcut menu.
3. Select the desired category.
4. Select the desired code snippet.

Figure 24-19 shows the Insert Snippet command in action.

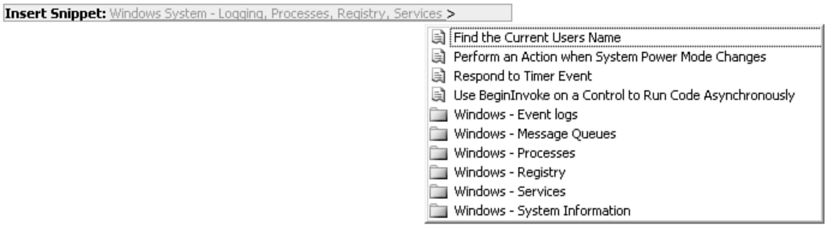

Insert Snippet: Windows System - Logging, Processes, Registry, Services >
Find the Current Users Name
Perform an Action when System Power Mode Changes
Respond to Timer Event
Use BeginInvoke on a Control to Run Code Asynchronously
Windows - Event logs
Windows - Message Queues
Windows - Processes
Windows - Registry
Windows - Services
Windows - System Information

FIGURE 24-19 Inserting a code snippet

Instead of using the menu to insert code snippets, we can use **code shortcuts**. First we need to find out which code shortcuts are available in the Code Snippets Manager. The Code Snippets Manager can be accessed from the *Tools > Code Snippets Manager...* menu. Next we type the shortcut text, for instance `ForEach`, in the code editor and press the Tab key to execute it. Listing 24-25 first shows the shortcut text and then the result after we press the Tab key.

Listing 24-25 Using a Shortcut to Insert a Code Snippet

```
'The shortcut.
ForEach

'The result.
For Each Item As String In CollectionObject

Next Item
```

As we can see in Listing 24-25, we need to fix the code snippet before it can be properly used. On first consideration it may seem like too much effort to remember all the shortcuts as well as correct the code that is actually inserted into the code editor. However, the code snippets are completely customizable, so it is worth your effort to spend some time and make the changes required to suit your needs.

The built-in Code Snippets tool is rather primitive and doesn't provide a very user-friendly interface. If you find yourself working extensively with code snippets the free **Snippet Editor** may be a better tool. As of this writing, it is available at www.codeplex.com/SnippetEditor.

Insert File as Text

Insert File as Text is not a standalone tool but rather a built-in function of the VS IDE. It can be used to import code from plain text files. To display the Insert File dialog select *Edit > Insert File as Text...* from the menu. The default file extension is **.vb* so we need to change the file extension to **.txt* before we can select a text file. The code in the selected text file is imported into the active code module at the current cursor position. Using text files to manage complete and reusable class modules, standard modules, and methods requires only a simple text editor, making it a portable, light-weight solution.

Task List (Ctrl+Alt+K)

The **Task List** is a simple but handy tool for managing the To-Do list for a solution. Using the only button on its toolbar we can create different tasks and set flags indicating their priority. By right-clicking on the list we can also sort, copy, and delete tasks.

Automating Excel

At the most fundamental level, automating Excel from .NET solutions does not differ from automating Excel from Classic VB. What must be taken into consideration is that the .NET Framework cannot communicate directly with Excel because of differences between .NET technology and the COM technology Excel is built on. It is necessary to create a bridge between these two technologies for us to be able to automate Excel from the .NET platform. The bridge between .NET and COM is mostly provided by features contained in two .NET Framework namespaces: **System.Runtime.InteropServices** and **System.EnterpriseServices**.

However, there are additional components required to allow interoperability that we need to discuss further.

Primary Interop Assembly (PIA)

When we set a reference to a COM type library in a .NET solution, the VS IDE automatically creates a default **Interop Assembly (IA)**. The autogenerated IA is a .NET-based assembly that acts as a wrapper for the COM type library. The IA provides us with basic access to the COM type library, and it contains type definitions (as metadata) of types implemented by COM. A **Primary Interop Assembly (PIA)** is a prebuilt, vendor-supplied assembly. The difference between an IA and a PIA is more or less semantic.

Microsoft has released PIAs for all Excel versions beginning with Excel 2002 as part of the Microsoft Office PIAs. The PIAs have **strong names** and are digitally signed by Microsoft. The use of strong names makes it possible to install PIAs in the **Global Assembly Cache (GAC)**. The GAC is a machinewide .NET assembly cache for the CLR. Assemblies that should have only one version on the system should be installed in the GAC.

One important point to understand is that only one version of the PIAs can be used on a system, although multiple versions can be installed side by side in the GAC. In addition, PIAs are registered in the Windows registry. If multiple versions of the PIAs are installed, only the latest version is registered, and the entries for any previous version are overwritten.

When we set a reference to Excel in a .NET solution the VS IDE reads the registry and adds a reference to the PIA instead of generating a new IA. This guarantees that we always use the PIAs if they are available. As a practical matter, when we are automating Excel from .NET we are always developing against the PIA and not the Excel COM type library.

The PIAs are optimized for Excel and you should always use the official Microsoft versions. The PIAs are also Excel version-specific. This means you cannot automate Excel 2002 using the PIA for Excel 2003. Therefore, you must be sure the correct version of the PIA is installed on your development computer. Whether or not the PIA is already installed on a computer depends on the following:

- For Excel 2002 on Windows XP or Windows Vista you need to manually download and install the redistributable PIA package from the

Microsoft Web site. If you run Windows XP, then the .NET Framework must be installed prior to installing the PIA package.

- For Excel 2003 or Excel 2007 on Windows XP, if Microsoft Office has been installed ***before*** the .NET Framework, then you must install the PIA package manually. You can either download the redistributable PIA package from the Microsoft Web site or install it from the Office CD.
- For Excel 2003 or Excel 2007 on Windows Vista you do not need to take any action at all. Because version 3.0 of the .NET Framework is shipped with Windows Vista, the PIAs are automatically installed when Office is installed.

Since no official PIA exists for Excel 2000, we must compile our own IA using the **TlbImp.exe** tool that is shipped as part of the .NET Framework SDK. It takes the Excel9.olb file as its input and generates a .NET assembly as its output. When automating Excel from .NET you should always develop against the earliest versions of the PIA and Excel that you plan to target and the earliest version of the .NET Framework you intend to use.

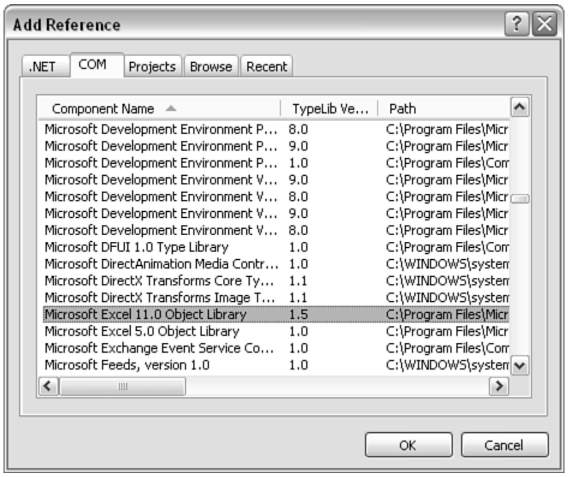
You need to be aware of the code execution overhead for all kinds of .NET solutions, especially when it comes to interaction between .NET and COM. Compared with Classic VB, .NET solutions require more components and therefore require more overhead to run. These components include

- The COM interop layer (PIA)
- The CLR
- The .NET Framework

As we see in the next section, there are additional aspects we need to consider to maintain acceptable performance for .NET solutions that automate Excel. If high performance is critical to your solution you may even consider using Classic VB if it is available and is an acceptable development platform.

Using Excel Objects in .NET Solutions

Create a Windows Forms solution and name it “Automate Excel.” Add a button to the form and name it “Automate Excel.” Next, add a reference to the Excel 2003 PIA or later. Choose *Project > Add Reference...* from the VS IDE menu to display the Add Reference dialog. Select the *COM* tab and scroll down to locate the Microsoft Excel 11.0 Object Library as shown in Figure 24-20. The reference is added when you close the dialog by clicking the OK button.



Component Name	TypeLib Ve...	Path
Microsoft Development Environment P...	8.0	C:\Program Files\Micr
Microsoft Development Environment P...	9.0	C:\Program Files\Micr
Microsoft Development Environment P...	1.0	C:\Program Files\Com
Microsoft Development Environment V...	9.0	C:\Program Files\Micr
Microsoft Development Environment V...	8.0	C:\Program Files\Micr
Microsoft Development Environment V...	9.0	C:\Program Files\Micr
Microsoft Development Environment V...	8.0	C:\Program Files\Micr
Microsoft DFUI 1.0 Type Library	1.0	C:\Program Files\Com
Microsoft DirectAnimation Media Contr...	1.0	C:\WINDOWS\system
Microsoft DirectX Transforms Core Ty...	1.1	C:\WINDOWS\system
Microsoft DirectX Transforms Image T...	1.1	C:\WINDOWS\system
Microsoft Excel 11.0 Object Library	1.5	C:\Program Files\Micr
Microsoft Excel 5.0 Object Library	1.0	C:\Program Files\Micr
Microsoft Exchange Event Service Co...	1.0	C:\Program Files\Com
Microsoft Feeds, version 1.0	1.0	C:\WINDOWS\system

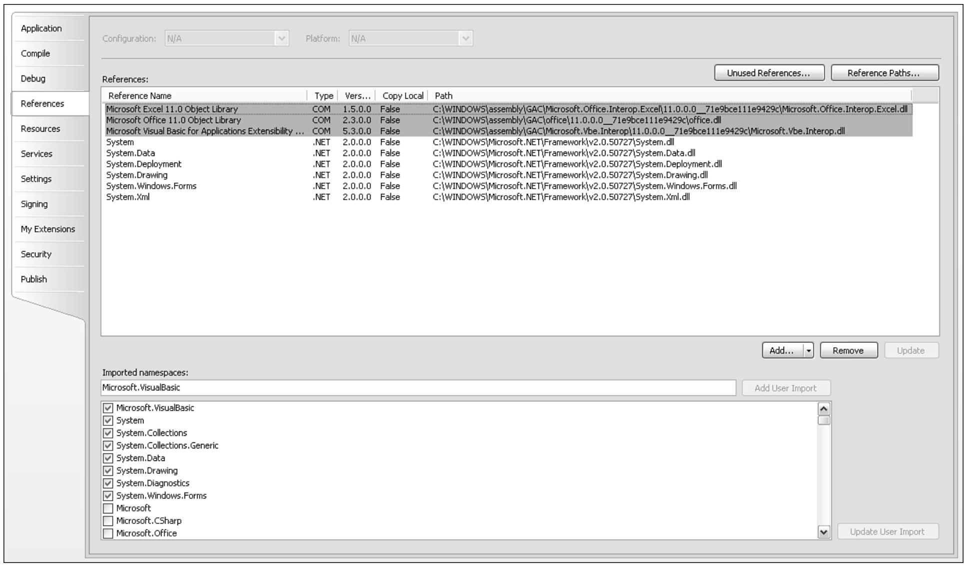
FIGURE 24-20 Adding a reference to the Excel Object Library

Choose *Project > Automate Excel Properties...* from the VS IDE menu. Select the *References* tab, and you see that three new Excel-related references have been added: the Excel Object Library, the Office Object Library, and the VBA Extensibility Object Library. Figure 24-21 shows the current list of references in the solution.

The System references are added by default. These give us access to the most commonly used .NET Framework class libraries. The imported namespaces are automatically included in all new .NET solutions. These are globally available in a solution. Open the Windows Form class module. When working with namespaces like Excel it is a good development practice to create a **namespace alias** for it at the top of the code module. We also add another `Imports` statement that is required as shown in Listing 24-26.

Listing 24-26 Namespace Alias and Imports Statements

```
'Namespace alias for Excel.
Imports Excel = Microsoft.Office.Interop.Excel

'To release COM objects and catch COM errors.
Imports System.Runtime.InteropServices
```



Configuration: N/A
Platform: N/A

References:

Reference Name	Type	Vers...	Copy Local	Path
Microsoft Excel 11.0 Object Library	COM	1.5.0.0	False	C:\WINDOWS\assembly\GAC\Microsoft.Office.Interop.Excel\11.0.0.0__71e9bce111e9429c\Microsoft.Office.Interop.Excel.dll
Microsoft Office 11.0 Object Library	COM	2.3.0.0	False	C:\WINDOWS\assembly\GAC\office\11.0.0.0__71e9bce111e9429c\office.dll
Microsoft Visual Basic for Applications Extensibility ...	COM	5.3.0.0	False	C:\WINDOWS\assembly\GAC\Microsoft.Vbe.Interop\11.0.0.0__71e9bce111e9429c\Microsoft.Vbe.Interop.dll
System	.NET	2.0.0.0	False	C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\System.dll
System.Data	.NET	2.0.0.0	False	C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\System.Data.dll
System.Deployment	.NET	2.0.0.0	False	C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\System.Deployment.dll
System.Drawing	.NET	2.0.0.0	False	C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\System.Drawing.dll
System.Windows.Forms	.NET	2.0.0.0	False	C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\System.Windows.Forms.dll
System.Xml	.NET	2.0.0.0	False	C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\System.Xml.dll

Imported namespaces:
Microsoft.VisualBasic

☑ Microsoft.VisualBasic
☑ System
☑ System.Collections
☑ System.Collections.Generic
☑ System.Data
☑ System.Drawing
☑ System.Diagnostics
☑ System.Windows.Forms
☐ Microsoft
☐ Microsoft.CSharp
☐ Microsoft.Office

FIGURE 24-21 References in the automate Excel solution

Declaring and instantiating some Excel COM objects, like Workbook and Range objects, requires that we **cast** the object reference to the precise type using the **CType** function. This is because the Option Strict setting prevents us from using code that might fail at runtime due to type conversion errors. The VS IDE actually helps us with this task by visually marking the objects that need to be cast.

Next, add a Click event handler for the button. Listing 24-27 shows the code required to get the Excel automation started. Put this code in the button's Click event. As you can see, we implemented an SEH but intentionally left out any exception handling code. At this stage we also did not add the code required to release any of the Excel objects we used.

Listing 24-27 Declare and Instantiate Excel Objects

```
Dim xlApp As Excel.Application = Nothing
Dim xlWkbNew As Excel.Workbook = Nothing
Dim xlWksMain As Excel.Worksheet = Nothing
Dim xlRngData As Excel.Range = Nothing
Dim sData() As String = {"Hello", "World", "!"}
```

```
Try
     'Instantiate a new Excel session.
     xlApp = New Excel.Application

     'Add a new workbook.
     xlWkbNew = xlApp.Workbooks.Add

     'Reference the first worksheet in the workbook.
     xlWksMain = CType(xlWkbNew.Worksheets(Index:=1), _
                      Excel.Worksheet)

     'Reference the range to which we will write some
      data to.
     xlRngData = CType(xlWksMain.Range("A1:C1"),  _
                      Excel.Range)

     'Write the data to the range.
     xlRngData.Value = sData

     'Save the workbook.
     xlWkbNew.SaveAs(Filename:="c:\Test\New.xls")

     'Make Excel visible for the user.
     With xlApp
          .UserControl = True
          .Visible = True
     End With

Catch COMex As COMException

Catch ex As Exception

End Try
```

As shown in Listing 24-27, we must explicitly use the `Value` property of the Excel Range object in VB.NET. This is because VB.NET does not recognize default properties.

Whenever Excel objects are instantiated at runtime the CLR creates so called **Runtime Callable Wrapper (RCW)** for each underlying COM object in the memory. It is the group of RCWs that constitute the runtime proxies, or bridges, between a .NET solution and the COM type libraries it references. This is important to keep in mind because the more Excel COM objects we use, the more memory our solution consumes at runtime. It is a good development practice to clean up the RCW reference counts so we don't end up with a large number orphaned RCWs.

Let's take a closer look at the code in Listing 24-27. Initially it looks like we are only using four objects: the Application object, the Workbook object, the Worksheet object, and the Range object. But we indirectly reference the Workbooks collection, the Worksheets collection, and the Range collection, so we actually use seven objects. The objects used indirectly are out of our control but must be managed anyway.

On the .NET platform the **Garbage Collector (GC)**, is responsible for all memory management. The GC uses a managed memory scheme that periodically traces live references. When the trace is complete, all unreachable objects are released, and the GC reclaims the memory they previously used. The GC operates in a nondeterministic manner, so we never know exactly when it will perform its memory management tasks.

For pure .NET solutions this is not a problem, but it becomes an issue when trying to release COM objects properly. When releasing Excel objects we must be sure to release ***all*** the objects we have used. Otherwise, we may end up in a situation where Excel remains in memory and continues to consume resources even after our application has ended.

The first step in a practical solution is to explicitly call the GC from our .NET code. Calling the GC is a time-consuming process, but one that may be necessary when automating Excel because it is the only way to release all the Excel COM objects referenced indirectly. Each RCW has a **finalizer** that is responsible for releasing its COM object from memory. This finalizer needs to be called twice to fully remove the COM object from memory. Therefore, if we call the GC twice it releases our three indirectly referenced Excel objects.

The second step in a practical solution is to call the `Marshal.FinalReleaseComObject` method for every Excel COM object. Note that Excel objects must be released in the reverse order in which they were created, with the Excel Application object released last. Listing 24-28 shows the code in our solution used to release all the Excel COM objects. This should normally be performed when we are closing the application.

Listing 24-28 Releasing Excel COM Objects with a Function

```
      'In the calling sub procedure.
      '...
        Finally

            'Calling the Garbage Collector twice.
            GC.Collect()
            GC.WaitForPendingFinalizers()
            GC.Collect()
            GC.WaitForPendingFinalizers()

            'Releasing the Excel objects.
            ReleaseCOMObject(xlRngData)
            ReleaseCOMObject(xlWksMain)
            ReleaseCOMObject(xlWkbNew)
            ReleaseCOMObject(xlApp)

        End Try

    End Sub

    Private Sub ReleaseCOMObject(ByVal oxlObject As Object)
        Try
            Marshal.ReleaseComObject(oxlObject)
            oxlObject = Nothing
        Catch ex As Exception
            oxlObject = Nothing
        End Try

    End Sub
```

Note how we use the custom `ReleaseCOMObject` function to release the Excel objects and set them to Nothing. This example also shows why the `Finally` block is so useful; it ensures that the code required to clean up our Excel objects will always run.

The Automate Excel example can be found on the companion CD in *\Concepts\Ch24 - Excel & VB.NET\Automate Excel* folder. If you just want to run the example, the Automate Excel executable file can be found in the *\Concepts\Ch24 - Excel & VB.NET\Excel Automate\Excel Automate\bin\Debug* folder on the CD.

Using Late Binding

Whenever possible, we should use early binding and declare all variables as specific types. The reasons for this are simple:

- Our .NET solutions run faster because it is not necessary to perform type conversion on any variables.
- The compiler can detect and display exceptions and therefore prevent runtime exceptions.
- We get IntelliSense support and dynamic help during the development process.

Unfortunately, it is common for developers to have the latest version of an application such as Microsoft Office while end users have earlier versions. However, given access to desktop virtualization software such as **WMware** (commercial software) and **Microsoft Virtual PC** (free tool) it is now much easier for developers to use the same versions of software as the end users they develop for. This makes it possible for developers to use early binding in their applications.

Resources in .NET Solutions

On the .NET platform we can add images, icons, strings, and text files as resources to our solutions. To add resources we select the *Resources* tab from the .NET solution Properties window and click the *Add Resource* button on its toolbar. All resources associated with a solution become part of the EXE or DLL file upon compilation of the solution.

NOTE VS 2008 ships with a large group of images and icons. These are contained in the file VS2008ImageLibrary.zip that is located in the folder *\Program Files\Microsoft Visual Studio 9.0\Common7\VS2008ImageLibrary\1033*.

To work with resources in code, we use `My.Resources` together with the name of the resource file. Listing 24-29 shows how we use an icon resource in code.

Listing 24-29 Associate an Icon Resource File to a Windows Form

```
Me.Icon = My.Resources.PetrasIcon
```

In this example, the `Me` keyword refers to a Windows Form, and `PetrasIcon` refers to an icon resource file. The `My` keyword refers to the `My` namespace that the .NET Framework makes available for all VB.NET solutions. This namespace exposes seven objects that allow us to work with various resources and features. Table 24-3 lists the `My` namespace objects along with the purpose of each.

Table 24-3 Objects in the My Namespace

Object	Purpose
`My.Application`	Provides information about the application such as path, assembly information, and environment variables.
`My.Computer`	Provides features for manipulating computer components such as audio, the clock, the keyboard, the file system, and so on.
`My.Forms`	Provides access to all Windows Forms in the solution.
`My.Resources`	Provides access to resources used by the solution.
`My.Settings`	Allows reading and storing application configuration settings.
`My.User`	Provides access to information about the current user, including whether or not the user belongs to a special user group.
`My.WebServices`	Provides features for creating and accessing a single instance of each XML Web service referenced by the solution.

Retrieving Data with ADO.NET

Despite the similarity in the name, ADO.NET is something totally different from classic ADO on the unmanaged platform. For instance, it does not include a Recordset object, and the Excel `CopyFromRecordset` method is not supported. This is covered in more detail in Chapter 25, "Writing Managed COM Add-ins with VB.NET." Another major difference is that ADO.NET has strong support for XML data representation. VS 2008 ships with version 3.5 of the ADO.NET class library.

ADO.NET is one of the default namespaces included in all Windows Forms based solutions, so to use it we just need to add `Imports` statements to the top of code modules from which ADO.NET will be called. However, to complicate things ADO.NET can be used in two different ways: **connected mode** and **disconnected mode**.

Before we can examine these two different approaches we need to first discuss **.NET Data Providers**. Data providers are used to connect to databases, execute commands, and provide us with the results. Each database, like SQL Server, Oracle, MySQL, and so on requires its own unique data provider. Some data providers are available by default in the .NET Framework, including SQL Server, Oracle, and OLE DB. Other data providers can be obtained from specific database vendors. For Microsoft Access and other databases that support ODBC, the OLE DB Data Provider can be used.

Connected mode means that we work with an open connection to the database. In this mode we explicitly use command objects and the **DataReader** object. A DataReader object retrieves a read-only, forward-only stream of data from a database. It can also handle multiple result sets. To do this, the connection must be open during the whole data retrieval process. Connected mode provides a performance advantage if we need to work with database records one at a time because the DataReader object retrieves and stores them in memory. However, the drawback is that connected mode creates more network traffic and requires having an active connection open during the whole database operation.

In Listing 24-30, we use a SQL Server database and therefore we import the namespace `System.Data.SqlClient`, which gives us access to the .NET Data Provider for SQL Server. We also use the ADO.NET class library and therefore we import the namespace `System.Data`.

Listing 24-30 Using a DataReader Object

```
'At the top of the code module.
Imports System.Data
Imports System.Data.SqlClient

Friend Function Retrieve_Data_With_DataReader() As ArrayList

        'SQL query in use.
        Const sSqlQuery As String = _
            "SELECT CompanyName AS Company " & _
            "FROM Customers " & _
```

```
     "ORDER BY CompanyName;"

'Connection string in use.
Const sConnection As String = _
     "Data Source=PED\SQLEXPRESS;" & _
     "Initial Catalog=Northwind;" & _
     "Integrated Security=True"

'Declare and initialize the connection.
Dim sqlCon As New SqlConnection(connectionString:= _
                                   sConnection)
'Declare and initialize the command.
Dim sqlCmd As New SqlCommand(cmdText:=sSqlQuery, _
                                connection:=sqlCon)
'Define the command type.
sqlCmd.CommandType = CommandType.Text

'Explicitly open the connection.
sqlCon.Open()

'Populate the DataReader with data and
'explicit close the connection.
Dim sqlDataReader As SqlDataReader = _
sqlCmd.ExecuteReader(behavior:= _
                       CommandBehavior.CloseConnection)

'Variable for keeping track of number of rows in the
'DataReader.
Dim iRecordCounter As Integer = Nothing

'Get the number of columns in the DataReader.
Dim iColumnsCount As Integer = sqlDataReader.FieldCount

'Declare and instantiate the ArrayList.
Dim DataArrLst As New ArrayList

'Check to see that it has at least one
'record included.
If sqlDataReader.HasRows Then

     'Iterate through the collection of records.
     While sqlDataReader.Read

          For iRecordCounter = 0 To iColumnsCount - 1
```

```
                'Add data to the ArrayList's variable.
                DataArrLst.Add(sqlDataReader.Item _
                               (iRecordCounter).ToString())

            Next iRecordCounter

        End While
    End If

    'Clean up by disposing objects, closing and
    'releasing variables.
    sqlCmd.Dispose()
    sqlCmd = Nothing

    sqlDataReader.Close()
    sqlDataReader = Nothing

    sqlCon.Close()
    sqlCon.Dispose()
    sqlCon = Nothing

    'Send the list to the calling method.
    Return DataArrLst

End Function
```

We first create a SqlConnection object and then a SqlCommand object. Next we explicitly open the connection, create the DataReader object, and iterate through the collection of records in the DataReader object by using its `Read` method. Within the loop we populate an ArrayList object with the data from the DataReader object. Finally, we close and clean up the objects we've used and return the data in the ArrayList to the calling method. The Northwind database used in this example can be found on the companion CD in *\Applications\Ch24 - Excel & VB.NET \Northwind*.

When working in disconnected mode we make use of the DataAdapter, DataSet, and DataTable objects, which are supported by all .NET Data Providers. A DataAdapter acquires the data from the database and populates the DataTable(s) in a DataSet. The DataAdapter object includes commands to automatically connect to and disconnect from the database. It also includes commands to select, insert, update, and delete data. The DataAdapter object runs these commands automatically. The DataSet is an in-memory representation of the data, and like the DataReader object it can handle multiple SQL queries at the same time.

The advantages of using disconnected mode are that it creates less network traffic because it acquires the data in one go, and it does not require an open connection to the database once the data has been retrieved. It also allows us to first update the retrieved data and then return the updated data to the database.

Listing 24-31 shows a complete function, including SEH, which first creates the Connection object together with the DataAdapter object. It then creates and initializes a new DataSet. Next it initializes the DataAdapter object, which automatically establishes a connection, retrieves the data, and closes the connection. The DataSet is filled with the retrieved data and finally the function returns the first DataTable in the DataSet.

Listing 24-31 Using DataAdapter and DataSet Objects

```
'On top of the code module.
Imports System.Data
Imports System.Data.SqlClient

    Friend Function Retrieve_Data_With_DataAdapter() As DataTable

        'SQL query in use.
        Const sSqlQuery As String = _
            "SELECT CompanyName AS Company " & _
            "FROM Customers " & _
            "ORDER BY CompanyName;"

        'Connection string in use.
        Const sConnection As String = _
            "Data Source=PED\SQLEXPRESS;" & _
            "Initial Catalog=Northwind;" & _
            "Integrated Security=True"

        'Declare the connection variable.
        Dim SqlCon As SqlConnection = Nothing

        'Declare the DataAdapter variable.
        Dim SqlAdp As SqlDataAdapter = Nothing

        'Declare and initialize a new empty DataSet.
        Dim SqlDataSet As New DataSet

        Try
            'Initialize the connection.
            SqlCon = New SqlConnection(connectionString:= _
                                          sConnection)
              'Initialize the DataAdapter.
              SqlAdp = New SqlDataAdapter(selectCommandText:= _
                                            sSqlQuery, _
                                            selectConnection:= _
                                            SqlCon)

              'Fill the DataSet.
              SqlAdp.Fill(dataSet:=SqlDataSet, srcTable:="PED")

              'Return the datatable.
              Return SqlDataSet.Tables(0)

          Catch Sqlex As SqlException
              'Exception handling for the communication with
              'the SQL Server Database.

              'Tell it to the calling method.
              Return Nothing

          Finally

              'Releases all resources the variable has consumed from
              'the memory.
              SqlDataSet.Dispose()

              'Release the reference the variable holds and
              'prepare it to be collected by the Garbage Collector
              '(GC) when it comes around.
              SqlDataSet = Nothing

              SqlCon.Dispose()
              SqlCon = Nothing

              SqlAdp.Dispose()
              SqlAdp = Nothing

          End Try

      End Function
```

The function returns a DataTable object from the ADO.NET class, but we do not need to cast it into a DataTable object from the DataSet class before returning it. The exception handler catches any exceptions that occur in the SQL Server Data Provider. In the `Finally` block we dispose all object variables and set them to nothing. A working example of this solution can be found on the companion CD in *\Concepts\Ch24 - Excel & VB.NET\Northwind* folder.

ADO.NET may be a new technology for developers who are working with the .NET platform for the first time. But for Microsoft, the latest technology is **.NET Language Integrated Query (LINQ)**, which is part of the .NET Framework 3.5 and was released with VS 2008. LINQ is a set of .NET technologies that provide built-in language querying functionality similar to SQL for accessing data from any data source. Instead of using string expressions that represent SQL queries, we can use a rich SQL-like syntax directly in our VB.NET code to query databases, collections of objects, XML documents, and more.

The future will tell us more about how well LINQ will succeed. Developers who are coming from classic ADO are more likely to first adopt ADO.NET and later perhaps also begin to use LINQ.

Further Reading

When it comes to the .NET Framework, VB.NET, and ADO.NET we have only scratched the surface. These technologies are all book-length topics in their own right. The following books are sources that we have found to be useful for a general introduction to VB.NET and to ADO.NET.

Programming Microsoft Visual Basic .NET Version 2003

Authored by Francesco Balena
ISBN# 0735620598—Microsoft Press
Unfortunately, this book has not been updated since VB.NET 2003 was released. However, it provides an excellent introduction to the .NET Framework and to VB.NET, as well as to other related technologies such as ADO.NET. It explicitly targets Classic VB developers who are moving to the .NET platform.

Visual Basic 2008 Programmer's Reference

Authored by Rod Stephens
ISBN# 0470182628—Wrox

This book offers a light introduction to VB.NET that explicitly targets beginning to intermediate level developers. This is a practical book about the .NET Framework, VS IDE, and VB.NET, written well in plain English. The only thing that may be annoying is that some screen shots are oversized. Hopefully this will be corrected in later editions of the book.

Additional Development Tools

The authors have no financial interest in these tools and are not connected to their vendors. The recommendations are based on our own daily use of these tools as .NET developers.

MZ-Tools

MZ-Tools 6.0 is an add-in to the VS IDE. It works with all current versions of VS.NET except for the Express edition. It adds many tools and functions to the VS IDE that are designed to simplify development work and increase productivity. For more information see www.mztools.com.

VSNETCodePrint

VSNETCodePrint 2008 is an add-in to the VS IDE that helps developers document their solutions. With this tool we can print, preview, and export a complete solution, selected projects, project items, classes, modules, and procedures in several file formats. It can save you a significant amount of time when you need to document solutions and inspect code. For more information see www.starprint2000.com.

It should be noted that MZ-Tools provides features to generate documentation using either HTML or XML file formats that overlap the features in VSNETCodePrint to some degree but are less advanced.

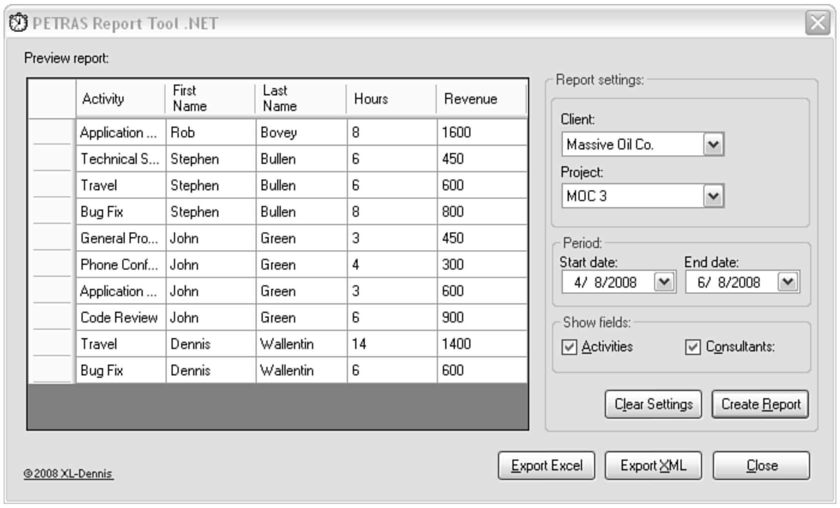
Q&A Forums

There are many general public VB.NET Q&A forums, but the Microsoft MSDN section for VB.NET is one of the best at http://forums.msdn.microsoft.com/en-US/tag/visualbasic/forums/. The VB.NET section at Xtreme VB Talk is also good, and it includes a subforum for .NET Office automation at www.xtremevbtalk.com/forumdisplay.php?f=97.

Practical Example—PETRAS Report Tool .NET

PETRAS Report Tool .NET is a practical case study that demonstrates a more complex VB.NET application than is possible to cover in a single chapter. In Chapter 25, the tool is converted into a managed COM add-in for Excel. The tool is a standalone, fully functional reporting solution. It retrieves data from a SQL Server database (created in Chapter 19, "Programming with Access and SQL Server") based on the user selection in the main Windows Form. It then populates predefined Excel report templates with the data. It can export reports either to Excel or to XML files. The solution can be found on the companion CD in *\Applications\Ch24 - Excel & VB.NET\PETRAS Report Tool.NET*. Please read the Read Me First.txt file located in the *\Applications\Ch24 - Excel & VB.NET* folder. You will find it helpful to open this solution in the VB IDE so that you can reference it while reading this section.

When the tool starts up, it first tries to establish a connection to the database. A custom Windows Form is displayed while the tool is trying to connect. If the connection attempt is successful, the main Windows Form shown in Figure 24-22 is displayed. If the connection attempt fails, an error message is displayed.



Activity	First Name	Last Name	Hours	Revenue
Application ...	Rob	Bovey	8	1600
Technical S...	Stephen	Bullen	6	450
Travel	Stephen	Bullen	6	600
Bug Fix	Stephen	Bullen	8	800
General Pro...	John	Green	3	450
Phone Conf...	John	Green	4	300
Application ...	John	Green	3	600
Code Review	John	Green	6	900
Travel	Dennis	Wallentin	14	1400
Bug Fix	Dennis	Wallentin	6	600

Client: Massive Oil Co.
Project: MOC 3
Start date: 4/ 8/2008
End date: 6/ 8/2008
☑ Activities ☑ Consultants:

FIGURE 24-22 PETRAS Report Tool .NET user interface

Use the following steps to create a report in the main form:

1. Select a *Client*.
2. Select a *Project*.
3. Select the reporting time period by entering a *Start date* and an *End date*.
4. Uncheck or keep the fields *Activities* and *Consultants*.
5. Click on the *Create Report* button to preview the report in the DataGrid.
6. Click the appropriate button to export to an Excel report or to an XML file.
7. If export to Excel is selected, Excel is launched and a copy of one of the four predefined report templates is created.
8. If export to XML is selected, a Save File dialog is displayed so you can specify a filename and location where the XML file should be saved.
9. If the export is successful, the selections you made become the new default values for all controls on the Windows Form. It is possible to clear these settings by selecting the *Clear Settings* button.
10. To close the Windows Form, click the *Close* button.

The .NET Solution

Although we only use one main Windows Form, our .NET solution includes some additional modules and files. Table 24-4 shows a summary of what the solution contains.

Table 24-4 Contents for the PETRAS Report Tool.NET Solution

Module Name	Type and Function
app.config	XML configuration file containing the connection string
frmConnecting.vb	Windows Form displayed while connecting to the database
frmMain.vb	Windows Form that is the main form for the solution
MCommonFunctions.vb	Standard module containing general functions for the tool
MDataReports.vb	Standard module containing all database functions
MExportExcel.vb	Standard module containing all the functions required to export data to Excel
MExportXML.vb	Standard module containing all the functions required to export data to XML files

Table 24-4 Contents for the PETRAS Report Tool.NET Solution

Module Name	Type and Function
MSolutions Enumerations Variables.vb	Standard module containing all the enumerations used in the solution
MStartUp.vb	Standard module containing the Main procedure for the solution

As you can see in Table 24-4, the solution does not include any class modules. Creating well-designed class modules is covered in Chapter 25. In addition to the components shown in Table 24-4, the solution uses four different Excel report templates. Depending on the user selections, one of them is used to create the requested report:

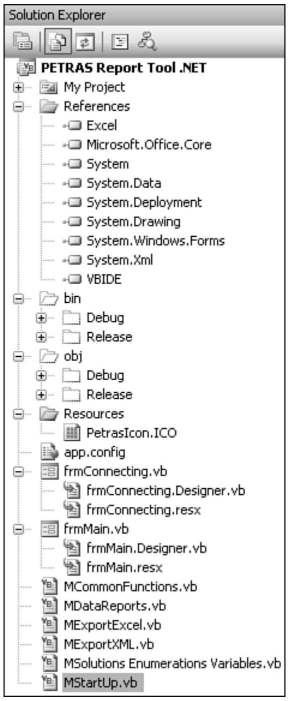
- **PETRAS Report Activities.xlt—**Used when only the Activities control is checked
- **PETRAS Report Activities Consultants.xlt—**Used when both the Activities and Consultants controls are checked
- **PETRAS Report Consultants.xlt—**Used when only the Consultants control is checked
- **PETRAS Report Summary.xlt—**Used when neither the Activities nor the Consultants controls are unchecked

If we click the *Show All Files* button in the Solution Explorer toolbar, it displays an expanded tree view. If we then expand the References item in the tree view we can see all references for the solution, as shown in Figure 24-23. Most hidden files are system files that we rarely need to work with, but it's a good exercise to explore all the files included in the solution.

In any non-trivial real-world application where we initially load a Windows Form, we usually need to ensure that certain conditions are met before loading it. In VB.NET we can use the same approach as with Classic VB. We create a `Main` subroutine in a standard code module that is used as the startup subroutine.

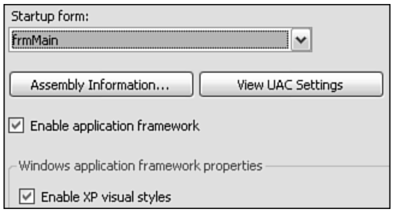
But in VB.NET, we need to change some additional settings in the solution before this will work correctly. After creating the new Windows Forms application, open the solution Properties window, and select the *Application* tab. Figure 24-24 shows the original startup settings for the PETRAS Report Tool.NET solution.

We add a standard code module to the solution that we name MStartup.vb. We add the `Main` subroutine and its code to this module, as shown in Listing 24-32.



- PETRAS Report Tool .NET
  - My Project
  - References
    - Excel
    - Microsoft.Office.Core
    - System
    - System.Data
    - System.Deployment
    - System.Drawing
    - System.Windows.Forms
    - System.Xml
    - VBIDE
  - bin
    - Debug
    - Release
  - obj
    - Debug
    - Release
  - Resources
    - PetrasIcon.ICO
  - app.config
  - frmConnecting.vb
    - frmConnecting.Designer.vb
    - frmConnecting.resx
  - frmMain.vb
    - frmMain.Designer.vb
    - frmMain.resx
  - MCommonFunctions.vb
  - MDataReports.vb
  - MExportExcel.vb
  - MExportXML.vb
  - MSolutions Enumerations Variables.vb
  - MStartUp.vb

FIGURE 24-23 The tree view in Solution Explorer



Startup form: frmMain

☑ Enable application framework

☑ Enable XP visual styles

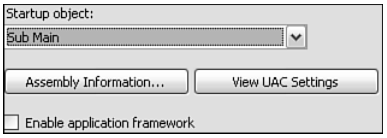
FIGURE 24-24 Default settings for the solution

Listing 24-32 Code for the Main Subroutine

```
Sub Main()
       'Enable Windows XP's style.
       Application.EnableVisualStyles()

       'Declare and instantiate the Windows Form.
       Dim frm As New frmMain

       'Set the position of the main Windows Form.
```

```
    frm.StartPosition = FormStartPosition.CenterScreen

    'Show the main Windows Form.
    Application.Run(mainForm:=frm)

    'Releases all resources the variable has consumed from
    'the memory.
    frm.Dispose()

    'Release the reference the variable holds and prepare it
    'to be collected by the Garbage Collector when it
    'comes around.
    frm = Nothing
End Sub
```

Now we return to the *Application* tab of the solution Properties window, where we uncheck the option *Enable application framework* and change the *Startup object* to the Main subroutine as shown in Figure 24-25.



Startup object: Sub Main

☐ Enable application framework

FIGURE 24-25 Modified startup settings

Unchecking the *Enable application framework* option implicitly removes the option to use **Windows XP styles**. Therefore, we need enable this option manually in the startup code, which is done in the first line of our `Main` procedure in Listing 24-32.

The `Main` subroutine is also a good place to put code to position the Windows Form before it is loaded. The `Main` subroutine is also an acceptable place to put code for connecting to a database, but in the PETRAS Report Tool.NET we use a different approach that is covered soon. When the user closes the main Windows Form we dispose its class and set the variable to nothing.

Windows Forms Extender Providers

The .NET Framework provides so-called **extender providers** to Windows Forms. These components can only be used with visual controls.

By adding them to our Windows Forms we get additional properties to work with. Extender providers are added to a Windows Form in exactly the same way as regular controls. However, the extender providers appear in the form's **Component Tray** rather than on the surface of the form itself.

Figure 24-26 shows the Component Tray for the main form of the PETRAS Report Tool.NET. The components used are the **ErrorProvider**, **HelpProvider**, and **ToolTip** components, for the main Windows Form, the **BackgroundWorker** component, which we cover later, and the SaveFileDialog component that was introduced earlier in the chapter.

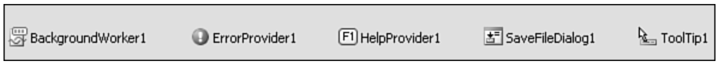


FIGURE 24-26 Extender providers in the PETRAS Report Tool.NET

The first extender provider in use is the ErrorProvider, which provides us with the option to set validation errors. It can be used with one or more controls on the Windows Form as each of them have the **Validating** event.

When a control's input is not valid the ErrorProvider places an error icon next to the control and displays an error message when the user hovers the mouse over the icon. Listing 24-33 shows how this is implemented in the PETRAS Report Tool.NET solution. As the code shows, we can create a single event that hooks the `Validating` events of all the targeted controls on the form.

Listing 24-33 The Validating Event Subroutine for Several Controls

```
Private Sub Client_Project_Validating(ByVal sender As Object, _
            ByVal e As System.ComponentModel.CancelEventArgs) _
            Handles cboClients.Validating, _
            cboProjects.Validating


Const sMESSAGECLIENTERROR As String = _
      "You need to select a client."

Const sMESSAGEPROJECTERROR As String = _
      "You need to select a project."

Dim Ctrl As Control = CType(sender, Control)

   If Ctrl.Text = "" Then
```

```
        Select Case Ctrl.Name

         Case "cboClients"
         Me.ErrorProvider1.SetError(control:=Ctrl, _
                                value:=sMESSAGECLIENTERROR)

         Case Else
         Me.ErrorProvider1.SetError(control:=Ctrl, _
                               value:=sMESSAGEPROJECTERROR)

        End Select

    Else

        Me.ErrorProvider1.SetError(control:=Ctrl, value:="")

    End If

End Sub
```

If one of the controls being validated has the focus when the user clicks the *Clear Settings* button, the validation handling code is executed. To prevent this we must add one line of code to the load event of the main Windows Form. This is shown in Listing 24-34.

Listing 24-34 Code to Prevent Validation when the Clear Settings Button Is Clicked

```
Me.cmdClearSettings.CausesValidation = False
```

We can prevent the entry of bad data into a control by writing handlers for the key press event as well.

Looking more closely at the code in Listing 24-33 may raise the question of why we do not use a control array as we would in Classic VB. This is because VB.NET does not currently support control arrays, and it does not appear as if this feature will be implemented in any future version. The solution shown is the closest workaround in VB.NET. The second extender provider, HelpProvider, is used to associate a help file (either a .chm or .htm file) with our application. Whenever our application is running and has focus, the HelpProvider associates the F1 button with our application's help file. For the PETRAS Report Tool.NET we use a simple **form-based help** system, meaning that we associate the help file with our main Windows Form. It is much easier to set this up using Windows Form properties manually at design time than to do it at runtime with code. The design-time property settings required to create a form-based help system are the following:

- Set the HelpKeyword property on HelpProvider1 to the value About.htm.
- Set the HelpNavigator property on HelpProvider1 to the value Topic.

One property of the HelpProvider that should be set in code is the HelpNameSpace property. Doing this provides us with a more flexible solution because we can change the location of the help file dynamically. Listing 24-35 shows the code in the main Windows Form load event required to set the HelpNameSpace property.

Listing 24-35 Setting the Path and Name to the Help File

```
'The help file in use.
Const sHELPNAMESPACE As String = "PETRAS_Report_Tool.chm"

'Setting the helpfile to the HelpProvider component.
Me.HelpProvider1.HelpNamespace = swsPath + sHELPNAMESPACE
```

The swsPath is a global enumeration member that holds the path to the application EXE file for the PETRAS Report Tool.NET.

The third extender provider is the ToolTip component. It provides us with the option to add a Tooltip to each control in a Windows Form. Whenever the user hovers over a control with the mouse the control's Tooltip is displayed.

Threading

With .NET we can leverage multithreading to create more powerful solutions. It is beyond the scope of this chapter to cover multithreading in detail, but we demonstrate a simple example. The .NET Framework includes an extender provider, BackgroundWorker, which allows us to run code on a separate, dedicated thread, meaning we can run our project in multithreading mode. This extender provider is normally used for time-consuming operations, but as this case shows, we can use it for other tasks as well.

In the PETRAS Report Tool.NET, we use the BackgroundWorker component to run the code that connects to the database. By using two of its events, `BackgroundWorker1_DoWork` and `BackgroundWorker1_RunWorkerCompleted`, we attempt to connect to the database in the background and be notified about the outcome. Listing 24-36 shows the code for the load event of the main Windows Form followed by the code for the two events of the BackgroundWorker component.

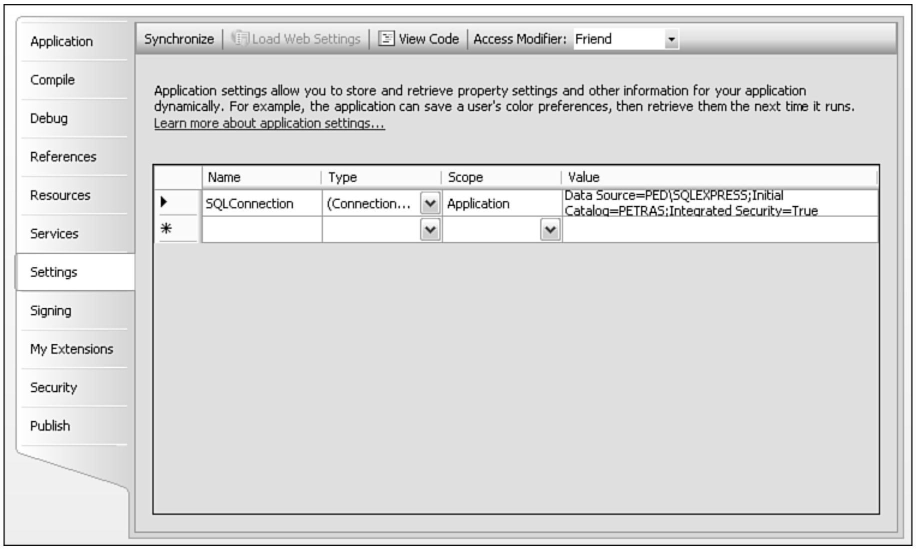
Listing 24-36 Code in Use for the BackgroundWorker

```
Private Sub Form1_Load(ByVal sender As System.Object, _
                             ByVal e As System.EventArgs) _
                             Handles MyBase.Load

'...

'Settings for the BackgroundWorker component.
With Me.BackgroundWorker1
             'Makes it possible to cancel the operation.
             .WorkerSupportsCancellation = True
             'Start the background execution.
             .RunWorkerAsync()
End With

'Change the cursor while waiting to BackgroundWorker
'component has been finished.
Me.Cursor = Cursors.WaitCursor

End Sub

Private Sub BackgroundWorker1_DoWork(ByVal sender As Object, _
            ByVal e As System.ComponentModel.DoWorkEventArgs) _
            Handles BackgroundWorker1.DoWork

        'Instantiate a new instance of the connecting
        'Windows Form.
        mfrmConnecting = New frmConnecting

        'Position the Windows Form and display it.
        With mfrmConnecting
            .StartPosition = FormStartPosition.CenterScreen
            .Show()
        End With
        'Can we connect to the database?
        If MDataReports.bConnect_Database() = False Then

            'OK, we cannot establish a connection to the
            'database so we cancel the background operation.
            Me.BackgroundWorker1.CancelAsync()

            'Let us tell it for the other backgroundWorker
            'event - the RunWorkerCompleted.
            mbIsConnected = False

        Else

            'Let us tell it for the other backgroundWorker
            'event - the RunWorkerCompleted.
            mbIsConnected = True

        End If

        'Close the connecting Windows Form.
        mfrmConnecting.Close()

        'Releases all resources the variable has consumed
        'from the memory.
        mfrmConnecting.Dispose()

        'Release the reference the variable holds and prepare
        'it to be collected by the Garbage Collector (GC) when
        'it next time comes around.
        mfrmConnecting = Nothing

End Sub

Private Sub BackgroundWorker1_RunWorkerCompleted _
            (ByVal sender As Object, _
            ByVal e As System.ComponentModel. _
            RunWorkerCompletedEventArgs) _
            Handles BackgroundWorker1.RunWorkerCompleted

 'If we have managed to connect to the database then we can continue.
  If mbIsConnected Then

        '...
  End If

 'Restore the cursor.
  Me.Cursor = Cursors.Default

End Sub
```

On its surface, the use of the BackgroundWorker component may look attractive. However, multithreaded application development is complex and easy to get wrong, so it should only be used in situations where it is absolutely necessary to run code outside the main process.

Retrieving the Data

A database connection string can be created using several different methods. For the PETRAS Report Tool.NET we create a solutionwide connection string using an application setting. This is accomplished in the *Settings* tab of the solution Properties windows, as shown in Figure 24-27.



Access Modifier: Friend

Application settings allow you to store and retrieve property settings and other information for your application dynamically. For example, the application can save a user's color preferences, then retrieve them the next time it runs.
Learn more about application settings...

	Name	Type	Scope	Value
▶	SQLConnection	(Connection...	Application	Data Source=PED\SQLEXPRESS;Initial Catalog=PETRAS;Integrated Security=True
*				

FIGURE 24-27 A solutionwide connection string

We first create a name for the setting and then select the type *(Connection string)*. The scope is now automatically set to Application. After placing the cursor in the *Value* field a button appears on the right side. Clicking this button displays a very useful built-in wizard for creating connection strings.

If we look in the Solution Explorer window, we notice that a new app.config XML file has been created and added to the solution. The app.config file will not be compiled into the executable file when we develop standalone applications like the PETRAS Report Tool.NET. Instead, it is a separate XML file that is installed alongside the PETRAS Report Tool.NET executable. This allows us to easily update the connection string by simply opening and editing the XML file. When we compile the solution the VS IDE creates an XML file based on the solution name, PETRAS Report Tool .NET.exe.xml, for example, instead of using the name app.config.

When creating a DLL, the app.config file is compiled into the DLL, which makes it more difficult to update the connection string. This is addressed in Chapter 25. Listing 24-37 shows how to read the connection string setting from within our application code.

Listing 24-37 Reading the Application Setting for the Connection String

```
'Read the connection string into a module variable.
Private ReadOnly msConnection As String = _
My.Settings.SQLConnection.ToString()
```

Next we use it to initialize a new SqlConnection object, as shown in Listing 24-38.

Listing 24-38 Function to Create New SqlConnection

```
Friend Function sqlCreate_Connection() As SqlConnection

    Return New SqlConnection(connectionString:=msConnection)

End Function
```

All functions that retrieve data using disconnected mode expect the DataSet object to contain one DataTable at the time. We use a module-level DataTable variable to populate the DataGridView control. If the user decides to either create an Excel report or export the data to an XML file, the same DataTable is used as an argument to one of the export functions.

Exporting Data

The `MExportExcel.vb` module contains all the functions required to export data to Excel using one of the four predefined Excel templates described earlier. The main export function, shown in Listing 24-39, takes several arguments. Since the query has already been executed we can get the results as a DataTable from the DataGridView control on the main Windows Form. The other arguments provide information about the options specified by the user when the data was retrieved from the database.

Listing 24-39 The Main Export to Excel Function

```
    Friend Function bExport_Excel(_
                ByVal dtTable As DataTable, _
                ByVal sClient As String, _
                ByVal sProject As String, _
                ByVal sStartDate As String, _
                ByVal sEndDate As String) As Boolean
```

Because the PETRAS Report Tool.NET is a standalone application not related to Excel, we first need to determine whether Excel exists and if so, determine which version of Excel is available. To accomplish this we examine the value of a critical Excel-related registry entry and use it to determine the current Excel version.

The lowest version of Excel that we can support is version 2002, meaning the tool cannot be used if version 2000 is installed. The function uses an enumeration of Excel versions, which is defined in the `MSolutions Enumerations Variables.vb` code module. To provide access to the .NET Framework functions that allow us to read the Windows registry, we import the namespace `Microsoft.Win32`. We also use regular expressions to complete this task, so the namespace `System.Text.RegularExpressions` also is imported into the code module. Listing 24-40 shows the code for the function.

Listing 24-40 Determine Which Version of Excel Is Available

```
'At the top of the module.
'To read the Windows Registry subkey.
Imports Microsoft.Win32
'To use regular expressions.
Imports System.Text.RegularExpressions
```

```
Friend Function shCheck_Excel_Version_Installed() As Short

Const sERROR_MESSAGE As String = _
                    "An unexpected error has occurred " + _
                    "when trying to read the registry."

 'The subkey we are interested in is located in the
 'HKEY_CLASSES_ROOT Class.
 'The subkey's value looks like the following:
 'Excel.Application.10
 Const sXL_SUBKEY As String = "\Excel.Application\CurVer"

 Dim rkVersionkey As RegistryKey = Nothing
 Dim sVersion As String = String.Empty
 Dim sXLVersion As String = String.Empty

 'The regular expression which is interpreted as:
 'Look for integer values in the interval 8-9
 'in the end of the retrieved subkey's string value.
 Dim sRegExpr As String = "[8-9]$"

 Dim shStatus As Short = Nothing

 Try
   'Open the subkey.
   rkVersionkey = Registry.ClassesRoot.OpenSubKey _
                    (name:=sXL_SUBKEY, writable:=False)

   'If we cannot open the subkey then Excel is not available.
   If rkVersionkey Is Nothing Then
      shStatus = xlVersion.NoVersion
   End If

   'Excel is installed and we can retrieve the wanted
   'information.
   sXLVersion = CStr(rkVersionkey.GetValue(name:=sVersion))

   'Compare the retrieved value with our defined regular
   'expression.
   If Regex.IsMatch(input:=sXLVersion, pattern:=sRegExpr) Then
    'Excel 97 or Excel 2000 is installed.
        shStatus = xlVersion.WrongVersion
   Else
```

```
         'Excel 2002 or later is available.
         shStatus = xlVersion.RightVersion
    End If

Catch Generalexc As Exception

     'Show the customized message.
     MessageBox.Show(text:=sERROR_MESSAGE, _
                     caption:=swsCaption, _
                     buttons:=MessageBoxButtons.OK, _
                     icon:=MessageBoxIcon.Stop)


    'Things didn't worked out as we expected so we set the
    'return variable to nothing.
    shStatus = Nothing

Finally

    If rkVersionkey IsNot Nothing Then

         'We need to close the opened subkey.
         rkVersionkey.Close()

        'Release the reference the variable holds and prepare it
        'to be collected by the Garbage Collector (GC) when it
        'comes around.
         rkVersionkey = Nothing
    End If

End Try

'Inform the calling procedure about the outcome.
Return shStatus

End Function
```

The module `MExportExcel.vb` also contains a function to verify that the Excel templates exist in the same folder as the executable file.

The function that exports data to an XML file also creates the Schema file for it. Listing 24-41 shows the two lines of code required to generate these files. We actually use the methods of the DataTable object to generate the XML files. This is because ADO.NET uses XML as its underlying data representation scheme. Both of these XML files can be opened and studied in more detail.

Listing 24-41 Creating XML and Schema Files

```
...
'Write the data to the XML file.
dtTable.WriteXml(fileName:=sFileName)

'Create the Schema file for the XML file.
dtTable.WriteXmlSchema(fileName:=Strings.Left( _
        sFileName, Len(sFileName) - 4) & ".xsd")
...
```

Summary

In this chapter, we provided a brief introduction to the .NET Framework, VB.NET, data access using ADO.NET, and Excel automation from VB.NET. Compared to Classic VB, the .NET Framework is a completely new and different platform. It is also a modern, advanced development platform with a great set of tools for creating user-friendly solutions. To fully utilize the .NET platform you must be prepared to invest significant time exploring and learning it. As we all know, there are no real shortcuts to learning new technology. Only hard work can accomplish the task. But the reward, in addition to the new knowledge itself, is that we can leverage all the knowledge from this chapter in the two chapters that follow.