3

Introduction to Managed Code

Technology is dominated by two types of people: those who understand what they do not manage, and those who manage what they do not understand.
—PUTT'S LAW

Topics Covered in This Chapter

What Is Managed Code?

Code that runs within the .NET Framework is considered *managed code.* This includes applications written in languages such as Visual C# and Visual Basic 2005. Code that is not managed by the .NET Framework is typically referred to as *unmanaged code.* This includes applications written in programming languages such as C++, Visual Basic 6.0, and VBA.

All Office solutions created using VSTO are written in managed code. VSTO supports both Visual Basic 2005 and Visual C#; however, we refer only to Visual Basic 2005 in text and in code examples in this book because we believe it is easier to transition from VBA to Visual Basic 2005. Keep in mind that there is much to learn about the .NET Framework and managed code; this chapter only scratches the surface.

Following are some benefits of using managed code:

- **Improved security.** Often, security permissions are enabled or disabled by end users or determined by the identity of the user who is attempting to run code. However, code managed by the .NET Framework uses the security model code access security (CAS), which is based on the code's identity and location.
- **Automatic resource management.** Code managed by the .NET Framework eliminates common programming errors such as memory leaks and memory corruption. To eliminate memory leaks, the .NET Framework releases all memory used by the program when it closes.
- **Verification of type safety.** The .NET Framework ensures that code is type safe before it is run. A programming language that is type safe prevents operations from being performed that are not appropriate for that type.
- **Code reuse.** The .NET Framework improves productivity by providing a class library that gives you thousands of classes you can use in your code. You can also create your own custom class library.
- **Language interoperability.** Applications that use managed code can be written in multiple languages that are supported by the .NET Framework. For VSTO, this includes Visual Basic and Visual C#.
- **Partial classes.** Visual Basic 2005 supports the use of partial classes, thereby separating designer-generated code from your own code. The partial classes are merged into one unified class when the code is compiled.

For VBA developers moving to managed code, an additional advantage is the ability to use all the coding and debugging tools in the Visual Studio IDE and the ability to design solutions using a true object-oriented programming language.

Introduction to Object-Oriented Programming

Object-oriented programming is a type of programming that relates coding constructs to *objects.* The objects that are created in code can have similar characteristics to objects in the real world. You define *properties* for an object to define its characteristics. For example, light bulbs might have a color property. The *value* of this property might differ for each individual light bulb; some might be white, and others yellow.

You define *methods* for an object to describe the actions it can take. Using our light bulb example, the methods might be TurnOn, Adjust-Brightness, and TurnOff. You can define *event handlers* for an object so that certain actions are performed when a particular *event* on the object occurs. For example, if a BeforeTurnOff event is raised, it would enable you to first decrease the brightness of the light bulb before turning it off. You also define the type of data the object can contain and any logic that is required to manipulate the data.

Understanding Classes and Objects

Classes contain methods, events, and properties that enable access to data. These methods, events, and properties of a class are known as its *members.* In VBA, you use *procedural programming,* writing most of your code within a code module (although classes are available in VBA). In object-oriented programming, most, if not all, of your code is contained within classes.

A class is often described as being a sort of cookie cutter or blueprint for an object. You can also think of a class as a template for an object.

Think about how you would use a Word template. You can add boiler-plate text and special formatting (styles) to the template. Then when a document is created based on that template, the new document contains the same characteristics that were applied to the template. The document has access to all the styles defined in the template, but an end user can make additional changes to the document and apply styles to different sections of the document. In this same way, a class contains the base functionality of an object, and yet you can later change properties of each object to make them different. Although a class contains a base set of functionality such as methods, properties, and events, these class members can be used, and the data associated with the object can be accessed, only when you've created an *instance* of the class. An object is an instance of a class, and the process is known as *instantiation*.

You learned in Chapter 1 about the extended objects that VSTO provides for Word and Excel. One of these is a NamedRange. A NamedRange is actually a class, and each time you add one to the worksheet, you are creating an instance of that class, or a NamedRange object. VSTO generates a unique name for each instance by appending an incremental number to the end of the class name: NamedRange1, NamedRange2, and so on. If you want to provide a different name, you can make the change in the Properties window. If you also change the value of the properties of NamedRange1—such as setting a specific font or adding a border—then the appearance of NamedRange1 will differ from that of NamedRange2. Even though each NamedRange is unique, the two of them share the same characteristics (properties) because both of them were created from the same NamedRange class.

When you create a VSTO application using Visual Basic 2005, you have numerous objects to work with. There are objects such as Windows Forms and controls, and there are Word, Excel, and Outlook objects such as documents, list objects, and e-mail items. Additionally, the .NET Framework contains a class library that you can use to create objects in your code.

You use the New keyword to create an instance of a class. In the case of a NamedRange, VSTO automatically creates the instance of the class in the auto-generated code of the worksheet's hidden partial class whenever you add a NamedRange (or any other control) to the worksheet. When you create your own class, you can store data privately; to do that, you create variables, known as *private member variables,* to store data. Then you create public properties so that the data can be accessed by other methods outside the class. This gives you complete control over the access to this data.

Creating Properties

To create a property, you add a Property statement. Private member variables are accessible only from outside the class when the Get and Set property procedures are accessed. These private member variables are also known as *fields.* The Get property procedure returns the value of the field, and the Set property procedure enables you to assign a value to the field. You can create a property in Visual Basic by typing the Property statement, such as the following, and then pressing the ENTER key.

```
Property Text() as String
```

Visual Basic automatically creates the Get and Set statements for you, as the code example in Listing 3.1 shows.

Listing 3.1. *Creating a property*

```
Property Text() As String
    Get

    End Get
    Set(ByVal value As String)

    End Set
End Property
```

Notice that the value field is created for you as a parameter of the Set property procedure. To assign the value to the member variable in the Set property procedure, you must create a member variable and write code. You must also write code to return the member variable in the Get property procedure. You can create a read-only property by using the ReadOnly keyword before the property. In the case of a read-only property, you need only provide a Get property procedure.

So far, the Text property you created lets you set and get a value for the Text property. You cannot use these properties or store any data in the class until you have actually created an instance of the class (an object). Each object that you create can hold a different value for the Text property.

Creating Classes

In this section you will create a simple Sentence class in a Word solution. As in Chapter 1, you will save the VSTO solutions that are described in this book in the Samples directory at the root of the C:\ drive.

1. Open Visual Studio 2005.
2. On the File menu, point to New and then click Project.
3. In the New Project dialog box, select Word Document in the Templates pane.
4. Name the solution SampleDocument, and set the location to C:\Samples.
5. In the Visual Studio Tools for Office Project Wizard, select Create a New Document, and then click OK.
6. In Solution Explorer, right-click the solution node, point to Add, and select New Item.
7. In the New Item dialog box, select Class, name the class Sentence.vb, and click Add.

Visual Studio creates the Sentence class and then opens the class file in code view.

8. Add the code in Listing 3.2 to the Sentence class.

Listing 3.2. *Creating a property named Text for the Sentence class*

```
Public Class Sentence
    Private TextValue as String
    Property Text() As String
        Get
            Return TextValue
        End Get
        Set (ByVal value As String)
           TextValue = value
        End Set
    End Property
End Class
```

The variable TextValue is a private member variable. You can retrieve and set its value only by using the public Text property.

Instantiating Objects

Now that you have a Sentence class, you will create two instances of the class, assigning a different value to the Text property of each class. Finally, you'll retrieve the value of the Text property for each Sentence object and insert it into your document. Follow these steps:

1. In Solution Explorer, right-click ThisDocument.vb and select View Code.
2. The Code Editor opens, and two default event handlers are visible. The first is the Startup event handler, and the second is the Shutdown event handler for the document.
3. Add the code in Listing 3.3 to the Startup event handler of ThisDocument.

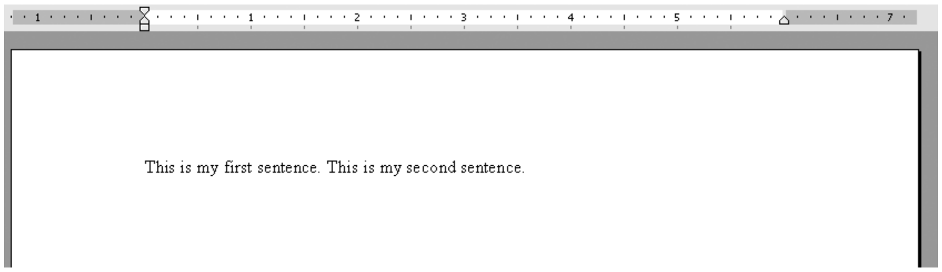
The code concatenates the text in Sentence1 and Sentence2 and then uses the InsertAfter method to insert the text into the first paragraph of the document. Because the code is added to the ThisDocument class, the Me keyword is used to represent the VSTO document (Microsoft.Office.Tools.Document, which wraps the native Document class).

Listing 3.3. *Creating two Sentence objects*

```
Dim Sentence1 As New Sentence()
Dim Sentence2 As New Sentence()
Sentence1.Text = "This is my first sentence. "
Sentence2.Text = "This is my second sentence. "
Me.Paragraphs(1).Range.InsertAfter( _
    Sentence1.Text & Sentence2.Text)
```

4. Press F5 to run the solution.

   When the solution runs and the Word document is created, the Startup event handler is raised and two instances of the Sentence class are created. The code then assigns a different string to each Sentence object. Finally, the value of each Sentence object is retrieved and inserted into the first paragraph of the document, as shown in Figure 3.1.



This is my first sentence. This is my second sentence.

Figure 3.1. *Text inserted into the document using the Sentence class*

5. Stop execution of the solution code by clicking Stop Debugging on the Debug menu, and then close the solution.

Creating and Calling Constructors

As you saw in the example in Listing 3.3, you use the New keyword to create an instance of a class. The New keyword calls the constructor of the class. A *class constructor* describes how to initialize the properties and methods of the class. Every class has a default constructor that is automatically generated for you as a method called Sub New.

You can override this default constructor by adding your own procedure named Sub New. Constructors can take parameters and can also be overloaded so that you can create an instance of the class in several ways. *Overloading* means that there are multiple versions of the same method, each with different parameters. Visual Basic 2005 enables you to create constructors that take one or more parameters so that you can pass data when you create an instance of the class. To overload the constructor, you add multiple Sub New methods that take different parameters.

1. Add the constructors in Listing 3.4 to your Sentence class.

Listing 3.4. *Adding two constructors to a class*

```
Public Sub New()
    TextValue = "Hello World! "
End Sub

Public Sub New(ByVal myText as String)
    TextValue = myText
End Sub
```

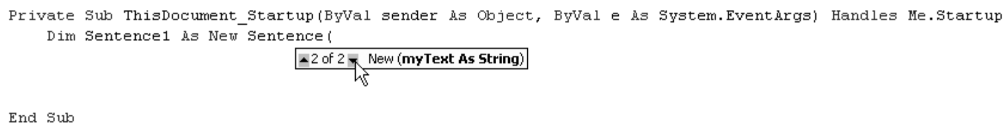
The first constructor overrides the default parameterless constructor and assigns the text “Hello World” to the member variable, TextValue. If you instantiate the class without passing any text, “Hello World” will be the value of the Sentence class. The second constructor takes a string as a parameter and assigns the string to the member variable.

2. Replace the code in the Startup event handler of ThisDocument with the code in Listing 3.5.

Listing 3.5. *Passing parameters to a constructor*

```
Dim Sentence1 As New Sentence()
Dim Sentence2 As New Sentence("This is my second sentence.")
Me.Paragraphs(1).Range.InsertAfter( _
    Sentence1.Text & Sentence2.Text)
```

Notice that when you type the open parenthesis after the word Sentence, IntelliSense lists the overloaded methods and displays the required parameter (myText As String) in method 2 of 2, as shown in Figure 3.2.



```
Private Sub ThisDocument_Startup(ByVal sender As Object, ByVal e As System.EventArgs) Handles Me.Startup
    Dim Sentence1 As New Sentence(
                                  ▲2 of 2▼ New (myText As String)


End Sub
```

Figure 3.2. *IntelliSense displays a parameter required by the constructor of the Sentence class.*

3. Press F5 to run the solution.

This time, when the solution runs, the value to be assigned to Sentence2 is passed to the constructor of the Sentence class when the class is instantiated. Although you could assign a value to the Text property of Sentence2 after it's instantiated (as shown in Listing 3.7), this example (Listing 3.5) uses the default value "Hello World!"

Adding Methods

You can also add methods to your class to perform an operation on the data. If you want the method to be accessible from the instance of the class, you must declare the method as a public method; to make it accessible from instances of the class in the same assembly, declare it as Friend. Private methods are available only to other members within the class.

1. Add the method in Listing 3.6 to your Sentence class. This method calls the ToUpper method of a String, which is provided by the .NET Framework.

Listing 3.6. *Creating a public method for a class*

```
Public Sub UpperCase()
    TextValue = TextValue.ToUpper()
End Sub
```

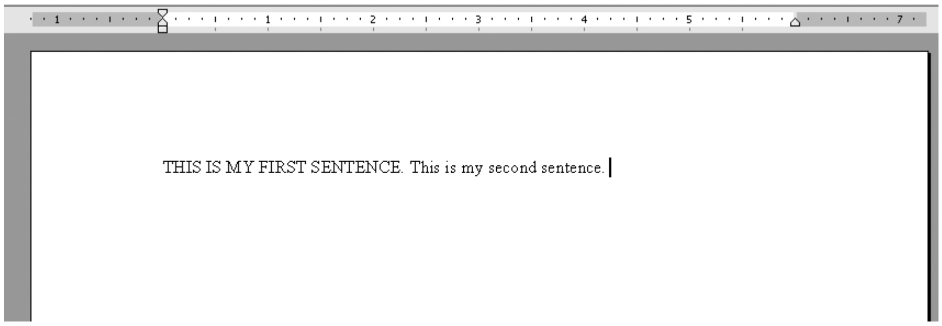
2. Replace the code in the Startup event handler of ThisDocument with the code in Listing 3.7 so that the UpperCase method is called only on Sentence1.

Listing 3.7. *Calling the UpperCase method of the Sentence class*

```
Dim Sentence1 as New Sentence()
Dim Sentence2 As New Sentence("This is my first sentence.")
Sentence1.Text = "This is my first sentence. "
Sentence1.UpperCase()
Me.Paragraphs(1).Range.InsertAfter( _
    Sentence1.Text & Sentence2.Text)
```

3. Press F5 to run the solution.

When the solution runs, the code in Listing 3.6 passes text to the constructor for the second object, but it uses the default (parameterless) constructor for the first object and then reassigns a value to the Text property of Sentence1. After the call to the UpperCase method on the first object, the first sentence that is inserted into the document appears in uppercase, and the second sentence appears in sentence case, as shown in Figure 3.3.



THIS IS MY FIRST SENTENCE. This is my second sentence.

Figure 3.3. *Text inserted into the document using the Sentence class*

Adding Events

You can add events to your class to indicate that objects created from this class can raise the events you've added.

1. Add the code in Listing 3.8 to your Sentence class. This code adds an event statement and replaces the UpperCase method.

Listing 3.8. *Creating an event for a class*

```
Public Event CaseChanged()

Public Sub UpperCase()

    TextValue = TextValue.ToUpper()
    RaiseEvent CaseChanged()

End Sub
```

2. Replace the code in the Startup event handler of ThisDocument with the code in Listing 3.9 so that a message box is shown when the event is raised.

   You can create an event handler for the CaseChanged event by declaring the variables for the Sentence objects with the WithEvents keyword, as shown in Listing 3.9. This code also adds a method that handles the OnChanged event for the Sentence1 and

McGrath.book Page 101 Thursday, December 7, 2006 10:04 AM

Sentence2 classes. Notice that the Sentence_ChangeCase method lists both Sentence1.CaseChanged and Sentence2.CaseChanged in the Handles clause.

Listing 3.9. *Displaying a message box when an event is raised*

```
WithEvents Sentence1 as New Sentence()
WithEvents Sentence2 As New Sentence( _
    "This is my first sentence.")

Private Sub ThisDocument_Startup(ByVal sender As Object, _
    ByVal e As System.EventArgs) Handles Me.Startup

    Sentence1.Text = "This is my first sentence. "
    Sentence1.UpperCase()
    Me.Paragraphs(1).Range.InsertAfter( _
        Sentence1.Text & Sentence2.Text)

End Sub

Sub Sentence_CaseChange() Handles Sentence1.CaseChanged, _
    Sentence2.CaseChanged

    MsgBox("Case changed.")

End Sub
```

3. Press F5 to run the code.

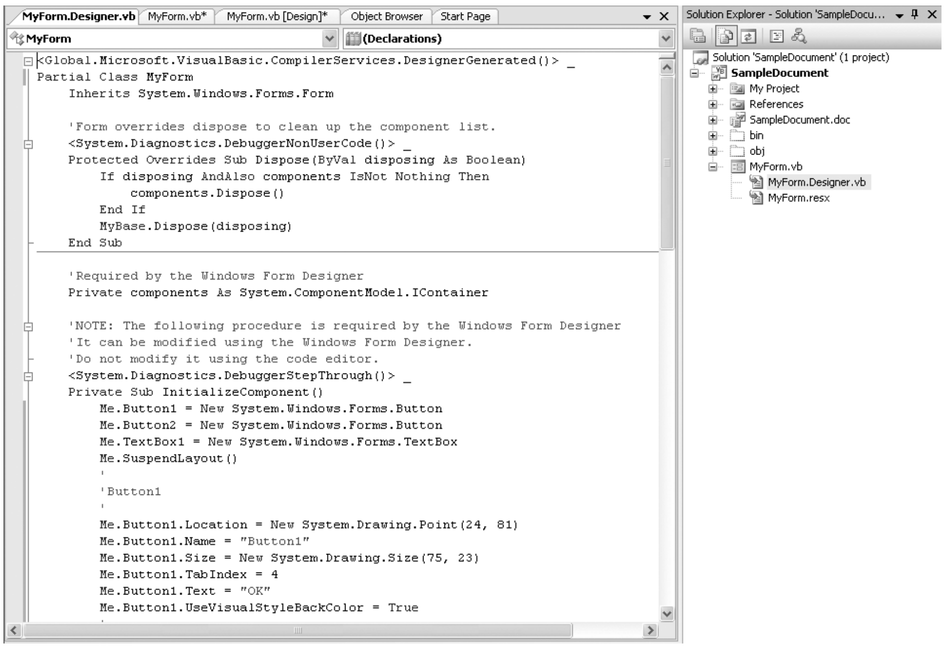
Only one message box is displayed because only Sentence1 called the UpperCase method, which raised the CaseChange event. If you add code to call UpperCase on Sentence2, the event will be raised on both objects, and therefore two messages will be displayed.

Partial Classes

Partial classes are a new feature in .NET Framework 2.0 and are supported in Visual Basic 2005. The Partial keyword enables you to split a class into separate source files. You can also define partial structures and interfaces.

You learned in Chapter 2 that there is a hidden code file behind the ThisDocument.vb file in Word solutions (and behind ThisWorkbook.vb, Sheet1.vb, Sheet2.vb, and Sheet3.vb in Excel solutions). These code files are partial classes. VSTO uses partial classes as a way to separate auto-generated code from the code that you write so that you can concentrate on your own code. Partial classes are also used in Windows Forms to store auto-generated code when you add controls to a Windows Form.

Figure 3.4 shows the partial class named MyForm, which contains the code that is generated whenever you add a control to the form. This code is stored in the MyForm.Designer.vb file; in contrast, the code you

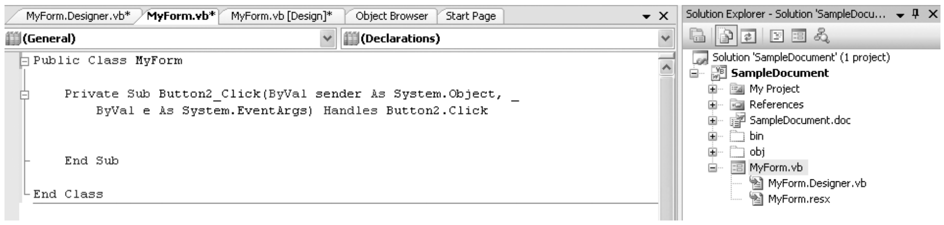


```
<Global.Microsoft.VisualBasic.CompilerServices.DesignerGenerated()> _
Partial Class MyForm
    Inherits System.Windows.Forms.Form

    'Form overrides dispose to clean up the component list.
    <System.Diagnostics.DebuggerNonUserCode()> _
    Protected Overrides Sub Dispose(ByVal disposing As Boolean)
        If disposing AndAlso components IsNot Nothing Then
            components.Dispose()
        End If
        MyBase.Dispose(disposing)
    End Sub

    'Required by the Windows Form Designer
    Private components As System.ComponentModel.IContainer

    'NOTE: The following procedure is required by the Windows Form Designer
    'It can be modified using the Windows Form Designer.
    'Do not modify it using the code editor.
    <System.Diagnostics.DebuggerStepThrough()> _
    Private Sub InitializeComponent()
        Me.Button1 = New System.Windows.Forms.Button
        Me.Button2 = New System.Windows.Forms.Button
        Me.TextBox1 = New System.Windows.Forms.TextBox
        Me.SuspendLayout()
        '
        'Button1
        '
        Me.Button1.Location = New System.Drawing.Point(24, 81)
        Me.Button1.Name = "Button1"
        Me.Button1.Size = New System.Drawing.Size(75, 23)
        Me.Button1.TabIndex = 4
        Me.Button1.Text = "OK"
        Me.Button1.UseVisualStyleBackColor = True
```

Figure 3.4. *Partial class for a Windows Form named MyForm, where auto-generated code is stored*

write to set properties or handle the events of the control should be written in MyForm.vb, as shown in Figure 3.5.

Notice that the class definition for MyForm.Designer.vb is Partial Class MyForm. The class definition for MyForm.vb does not contain the Partial keyword. Instead, it is simply Public Class MyForm.



```
Public Class MyForm

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


    End Sub

End Class
```

Figure 3.5. *Partial class for a Windows Form named MyForm, where developer-written code is stored*

The Partial keyword is not needed for the main class definition; it is needed only for any additional class definitions that share the same class name. When you compile the code, Visual Basic automatically merges the code from the partial classes with the code for the main class.

Another way you might use partial classes is to divide a programming task between two developers. If each developer writes code in a separate class file, you can then add each class to the main project and Visual Studio will automatically merge the classes during the build process as if they were a single class file.

Generic Classes

Generic classes are a new feature in the .NET Framework and are supported in Visual Basic 2005. A *generic class* is a single class that provides functionality for different data types, without the need to write a separate class definition for each data type. You can also define generic methods, structures, and interfaces.

A generic class uses type parameters as placeholders for the data types. The code example in Listing 3.10 shows the declaration of a generic class using *t* to represent the type parameter. You can specify more than one parameter by separating the parameters with commas. When you want to instantiate the class, you must specify the data type, rather than the type parameter, in the declaration, as shown in Listing 3.10.

Listing 3.10. *Creating a Generic class*

```
Public Class MySampleClass(Of t)

    ' Implementation code for the class goes here.

End Class


Sub CreateGenericClasses()

    Dim myStringClass As New mySampleClass(Of String)
    Dim myIntegerClass As New mySampleClass(Of Integer)

End Sub
```

In the System.Collections.Generic namespace, the .NET Framework provides a number of generic collection classes that correspond to existing (nongeneric) collection classes. For example, you can use a Dictionary class to specify the data type for a key-value pair (rather than use a Hashtable), and a List is the generic class that corresponds to an ArrayList.

Interfaces

Like classes, *interfaces* define properties, methods, and events of an object. Unlike classes, interfaces do not provide any implementation, and you cannot create an instance of an interface. A class can implement one or more interfaces.

Any class that implements an interface must implement all the members of the interface as they are defined. An interface should not change after it has been deployed in your solution, because any such change could possibly break existing code.

You declare interfaces using the Interface statement. For example, the code in Listing 3.11 defines an interface that must be implemented with a method that takes an Integer argument and returns an Integer. When you implement this interface in a class, the data type for the arguments and the return value of the method must match those of the method defined (in this case, Integer). You can implement this interface within a class by using the Implements keyword, as shown in Listing 3.11.

Listing 3.11. *Creating and implementing an interface*

```
Public Interface ISampleInterface
    Function SampleFunction(ByVal Count As Integer) As Integer
End Interface


Public Class SampleClass
    Implements ISampleInterface

    Function SampleFunction(ByVal Count As Integer) As Integer _
        Implements ISampleInterface.SampleFunction

        ' Add code to perform the function here.
    End Function

End Class
```

Code Modules

Code modules in Visual Basic work in the same way as they do in VBA. A *code module* is a container for global methods and properties that can be used by other parts of your application. Unlike classes, you do not need to create a new instance of a module in order to call the methods.

Nor do you need to fully qualify your call unless the same method exists in multiple modules. To fully qualify the method name you would use moduleName.methodName().

Modules are a simple way to create code that can be accessed from anywhere in your application, but in general it is better to use classes and object-oriented techniques. In the next section you will learn about the benefits of object-oriented programming.

Object Orientation

To be considered a true object-oriented language, a language should support the following features:

- Encapsulation
- Inheritance
- Polymorphism

All these features were available in VBA except inheritance. This is one reason many people never considered VBA a true object-oriented programming language. This isn't to say, however, that Visual Basic 2005 is merely VBA plus inheritance. Many more capabilities and enhancements have been made to Visual Basic 2005.

In this section we look at these encapsulation, inheritance, and polymorphism requirements of object-oriented programming.

Encapsulation

Encapsulation enables you to control the access to data within a class. For example, suppose your class has a number of methods that work on some data. Code that calls into the instantiated class (the object) need not know how a particular operation functions. To perform an action, the calling code need know only that the functionality exists and that it needs to call it. By not allowing direct external access to those methods and by hiding the logic used in the class, you are following the principle of encapsulation.

You can hide the implementation of your class by using access modifiers that prevent code outside the class from modifying data within the class or calling its methods. For example, you can use the Private keyword with a property or method that you don't want outside code to access. However, if you want to manipulate data from outside the class, you must provide public properties or methods. This was illustrated in the Sentence class you created earlier in this chapter. The Text property of the Sentence class had a Get property procedure and a Set property procedure that enabled you to write code to assign values to and retrieve values from the property. The actual data, however, was stored in a private member variable that was not directly accessible from outside the class.

The value of this feature becomes clearer if we add logic along with setting the internal value, such as checking the spelling of the sentence. The developer who sets the text property doesn't have to know *how* the Sentence object is checking the spelling, only that it *does* check the spelling when it sets the value.

Inheritance

Using *inheritance,* you can create a class that is based on an existing class, giving your new class all the behavior and functionality of the existing class. The class that you inherit from is known as the *base class,* and the class that is inheriting the functionality is known as the *derived class.* You can extend the functionality of the derived class by adding properties or methods that did not exist in the base class, or you can override inherited properties or methods so that they behave differently in the derived class.

Visual Basic 2005 supports inheritance, although it doesn't support multiple inheritance. A derived class can have only one base class.

Using inheritance, you can reuse existing code that performs most of the functionality you require. You modify only a portion of the code to meet your needs, instead of having to reinvent the wheel. Whenever you need functionality in your application, you should look at the .NET Framework class library to see whether the functionality exists or whether you can inherit the base functionality from one of the classes. For example, if you want to extend an existing Textbox control, you can create a new class that derives from the Windows Forms Textbox control, as shown in Listing 3.12.

Listing 3.12. *Inheriting from an existing Windows Forms control*

```
Public Class MyTextBox

    Inherits System.Windows.Forms.TextBox

    ' Add code to override existing TextBox functionality.

End Class
```

Polymorphism

Polymorphism is the ability to create identically named methods or properties within a number of derived classes that perform different functions. You can implement polymorphism by using interfaces or inheritance. For inheritance-based polymorphism, you override methods in a base class with new implementations of the methods in the derived class. For interface-based polymorphism, you implement an interface differently in multiple classes. You saw an example of this when you created multiple constructors for your Sentence class.

Exploring the .NET Framework

The .NET Framework is made up of the common language runtime and a set of code libraries called the Framework Class Library. This framework is a platform for building .NET applications such as Windows applications, Web-based applications, and VSTO customizations. Before delving into the common language runtime, we briefly cover three concepts that are important to programming in managed code: assemblies, namespaces, and application domains.

Assemblies

An *assembly* is a collection of classes and functionality that is stored as an executable file (.exe) or a library (.dll). When you compile a .NET language such as Visual Basic 2005, your code is not compiled directly into machine language, or binary code. Instead, it is compiled into an assembly-like language known as Intermediate Language (IL). No matter which language you use (Visual Basic 2005 or C#) to create a VSTO solution, the build process compiles the code into IL. The assembly contains both IL and metadata that describes each class and its members, along with information about the assembly itself, such as the assembly name, version, and any dependencies it has.

Assemblies can be private or shared. A *private* assembly normally contains code that is intended to be used by only one application. These assemblies can reside in the same folder as the application that uses them, or in a subfolder of the application. For example, when you create and build a VSTO solution, the compiled code is saved in a subfolder of your solution.

Shared assemblies, on the other hand, are designed to be shared among many applications. Because any software can access these assemblies, they should be stored in a special directory known as the *global assembly cache* (GAC). An example of a shared assembly is an Office primary interop assembly (PIA) described later in this chapter.

Namespaces

Namespaces help you organize the objects in an assembly, such as classes, interfaces, structures, enumerations, and other namespaces.

Using namespaces helps you avoid problems such as naming collisions or conflicts within your code. For example, let's say you have a class named Math that contains functionality to add or subtract the value of Excel ranges. You could add a reference to an assembly that contains a class also named Math but having different functionality. When you run your application, there would be no way for the .NET Framework to distinguish between your Math class and the referenced Math class.

Creating namespaces for your classes gives you another level of naming that helps disambiguate your classes. In the same way that using people's last names can help distinguish them from others who share the same first name, using a namespace along with a class name (also known as *fully qualifying* a class) helps the .NET Framework runtime distinguish one class from a like-named class. Often a company name is used as an alias of a namespace, so, for example, MyCompany.Employees can be easily distinguished from YourCompany.Employees.

To fully qualify the name of an object, you simply prefix the object with the namespace. For example, the Button class that you add to a Word document is in the Microsoft.Office.Tools.Word.Controls namespace and is referenced as Microsoft.Office.Tools.Word.Controls.Button. In contrast, the Button class you add to a Windows Form is in the System.Windows.Forms namespace and is referenced as System.Windows.Forms.Button.

You can include a namespace in your project by using the Imports statement, and you can optionally provide an alias to be used in place of the namespace. For example, you could add the following line to the top of your code file:

```
Imports Microsoft.Office.Tools.Word
```

Or, to disambiguate this namespace from the Microsoft.Office.Interop.Word namespace, you might use an alias:

```
Imports Tools = Microsoft.Office.Tools.Word
```

In this way, you can refer to an object within that namespace by using the alias. So instead of declaring myBookmark as a Microsoft.Office.Tools.Word.Bookmark, you could declare it as a Tools.Bookmark.

Application Domains

Application domains give the .NET Framework a way to isolate applications that are running in the same process. For example, if you're running multiple add-ins for your application and if one of them needs to be reloaded, you would want to ensure that the other add-ins are not affected. Loading the add-ins into a separate application domain guarantees this isolation. You can run several application domains in a single process and achieve the same level of isolation that exists when you run the applications in separate processes.

You can also set security permissions on an application domain. For example, when an application domain is created for a VSTO solution, the VSTO runtime sets policy for the application domain so that it does not trust the My Computer Zone. This practice ensures that the code in the My Computer Zone has been granted trust explicitly rather than allowing all code to run by default. You'll learn more about security in Chapter 11, Security and Deployment.

Common Language Runtime

The *common language runtime* is a runtime environment that supports multiple .NET Framework programming languages, such as Visual Basic 2005 and Visual C#. The common language runtime manages your code and provides compilation services, exception handling services, reflection services, memory management services, and a security mechanism for running secure code.

Compilation

At run time, the common language runtime compiles IL code into machine code (binary) that is specific to the hardware and operating system the code is currently running on. The common language runtime compiler is known as the Just-In-Time (JIT) compiler because it doesn't go through and compile all the code in the assembly at one time but rather compiles code only as it is being called. If the same method is called again while the solution is running, the common language runtime runs the binary that is already in memory rather than rerun it through JIT compilation. One benefit of this arrangement is that only the code that needs to be run is compiled, saving time and memory compared with compiling it all at once.

Additionally, the common language runtime can read the metadata of the IL stored in the assembly and can verify that the code is type safe before attempting to access memory locations. Note also that the verification process can be skipped if security policy is set to do so.

Exception Handling

The common language runtime provides an exception notification service so that it's easy to determine that an error has occurred. The .NET Framework provides a number of exception classes that describe the most common types of exceptions. With managed code, you should use structured exception handling—such as Try Catch Finally (Try Catch) statements—to check whether an exception is thrown within your code and handle the exception accordingly. Rather than use the method of error handling used in VBA code (On Error GoTo statements), you should instead favor the more robust exception handling of a Try Catch statement.

A Try Catch statement is made up of a Try block, a Catch block, and an End Try statement. You add the code that can possibly cause an exception in the Try block as a way to "try out" the code. Then you "catch" any exceptions that are thrown by handling the exception in the Catch block.

If needed, you can break out of a Try Catch statement by using the Exit Try keyword. The Finally block is always executed, whether or not an error is raised and handled.

You end a Try Catch statement with End Try. For example, the code in Listing 3.13 shows how you can check whether a folder exists. The code sets a folder named Personnel as the current Outlook folder in a Try block and displays an error in the Catch block if an exception is raised. An exception is raised if inbox.Folders does not contain an entry for "Personnel." Note, however, that it is better to specify the type of exception in the Catch statement (if it is known) than to catch all exceptions as in this example.

Listing 3.13. *Try Catch statement*

```
Try
    Me.ActiveExplorer().CurrentFolder = inBox.Folders( _
        "Personnel")
Catch Ex As Exception
    MsgBox(Ex.Message)
End Try
```

Reflection

Using *reflection*, you can discover which types exist in an assembly at run time, as well as examine its methods, properties and events, and attributes. *Attributes* are metadata tags that you can apply to your code. The common language runtime uses classes within the .NET Framework class library that are part of the System.Reflection namespace to programmatically inspect an assembly.

The .NET Framework 2.0 SDK contains a tool named ILDASM that uses reflection to display all the types and members of an assembly. You can also view the assembly's IL. There are other tools that use reflection on an assembly that do not ship with the .NET Framework, such as .NET Reflector.

Garbage Collection

The common language runtime provides automatic memory management known as garbage collection. *Garbage collection* is a process of releasing memory used to store an object or object reference when it is no longer being used. The garbage collector examines variables and objects and checks whether there are any existing references. If the objects are not being referenced, they are not destroyed; rather, they are flagged for garbage collection. The .NET Framework determines the time frame in which the garbage collection actually occurs.

The garbage collector reclaims any memory that is no longer being used. The garbage collector functionality is exposed through the GC class. At times, you should mark an object as being eligible for garbage collection (for example, setting the object variable to Nothing); at other times, you might want to force a garbage collection to free up memory (for example, calling GC.Collect()). Under normal circumstances you shouldn't force garbage collection.

Security

The common language runtime offers *code-based* security, in which permissions are based on the identity of the code, rather than *role-based* security, which is based on the identity, or role, of the user trying to run the code. This security model is known as code access security (CAS), and the security policy set determines what the code can do and how much the code is trusted. Security is covered in more detail in Chapter 11.

Common Language Specification

For code to interact with objects implemented in any language, the objects must expose common features to the common language runtime. The Common Language Specification (CLS) defines the rules that must be adhered to. For example, it specifies that arrays must have a lower bound of zero.

The *common type system,* which defines how types are declared and used, is defined in the CLS. The common type system ensures that objects written in different languages can interact. All types derive from System.Object, and they are typically classified into value types and reference types.

Value Types and Reference Types

There are two main categories of types that are managed by the common language runtime: value types and reference types. The difference between them is that *reference types,* such as objects, are stored on a portion of memory in the computer called the *heap,* whereas *value types,* such as numeric data types, are stored on another portion of memory called the *stack.*

Value types include structures, the numeric data types (Byte, Short, Integer, Long, Single, Double), enumerations, Boolean, Char, and Date. Reference types include classes, delegates, arrays, and Strings, and they can be accessed only through a reference to their location. When you create a variable for a value type without assigning it a value, the type is automatically initialized to a default value.

Table 3.1 lists some common data types, shows how they map to the System namespace in the .NET class library, and lists their default values. When you create a reference type variable, however, its value defaults to Nothing.

Table 3.1. *Default Value for Data Types*

Data Type	Namespace Map	Default Value
Byte	System.Byte	0
Short	System.Int16	0
Integer	System.Int32	0
Long	System.Int64	0

(continues)

Table 3.1. *Default Value for Data Types (Continued)*

Data Type	Namespace Map	Default Value
Single	System.Single	0
Double	System.Double	0
Decimal	System.Decimal	0D
Boolean	System.Boolean	False
Date	System.DateTime	01/01/0001 12:00:00AM
String	System.String	Nothing

.NET Framework Class Library

As the name suggests, the .NET Framework class library is a library of classes that contains popular functionality for use in your code. For example, an XMLReader class in the System.XML namespace gives you quick access to XML data. Instead of writing your own classes or functionality, you can use any of the thousands of classes and interfaces—such as Windows Forms controls and input/output (IO) functions—that are included in the .NET Framework class library. You can also derive your own classes from classes in the .NET Framework. When you're working with the Framework class library, it's important to understand that these classes are organized within the context of namespaces.

The .NET Framework class library is organized into hierarchical namespaces according to their functionality. This arrangement makes it easier to locate functionality within the library and provides a way to disambiguate class names. A number of namespaces are automatically imported, or referenced, when you create a project in Visual Studio. For example, in a Windows Forms application, you do not need to fully qualify the Button class as Microsoft.Windows.Forms.Button, because the Microsoft.Windows.Forms namespace is automatically imported into your project. Instead, you can refer to the Button class directly. When you're using Visual Studio Tools for Office, however, there is a Button class in Excel and Word that differs from the Windows Forms Button class. In this case, you must fully qualify any references to the VSTO Button. The code example in Listing 3.14 illustrates.

Listing 3.14. *Fully qualifying an object*

```
' Declare a variable for a Windows Forms button.
Dim myButton As Button

' Declare a variable for a button to be used on a Word document.
Dim myWordButton As Microsoft.Office.Tools.Word.Button
```

In Visual Basic, you can use an Imports statement at the top of your code file to include a namespace in your project. In this way, you do not have to type the fully qualified namespace every time you reference the classes within that namespace. You can also create an alias for the namespace, as shown in Listing 3.15.

Listing 3.15. *Creating an alias for a namespace*

```
Imports Tools = Microsoft.Office.Tools.Word

Sub Test()
    ' Declare a variable for a Windows Forms button.
    Dim myButton As Button

    ' Declare a variable for a button to be used on a
    ' Word document.
    Dim MyWordButton As Tools.Button
End Sub
```

Table 3.2 lists some of the popular namespaces in the .NET Framework class library that you might use in your VSTO solutions.

To use a .NET Framework class in your code, you must first set a reference to the assembly that contains the class. To set a reference, you click Add Reference from the Project menu. When the Add Reference dialog box appears, you select the component name from the .NET tab

Table 3.2. *Popular Namespaces in the .NET Framework Class Library*

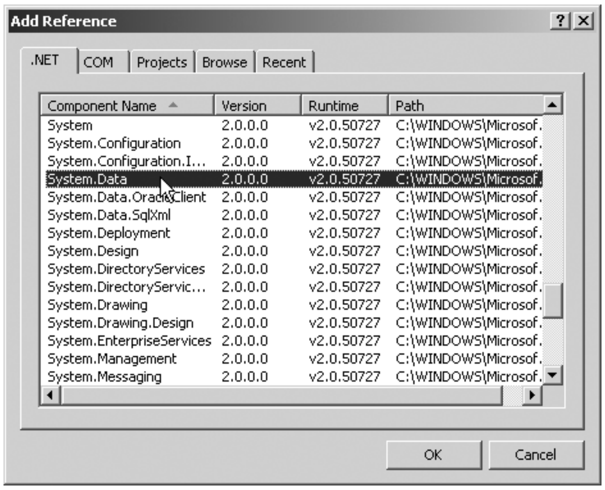
Namespace	**Description**
System	Contains the base data types, such as String, Boolean, and Object. This namespace is automatically included in your project, so you need not qualify any of the types in this namespace. Most languages, including Visual Basic 2005, define their own data types, which typically map to the types in this namespace. This is one reason for some of the language changes (data type changes) between VBA and Visual Basic 2005. For example, a Short in Visual Basic 2005 is equivalent to an Integer in VBA, and the Variant data type in VBA is no longer supported. These types were updated in Visual Basic 2005 to conform to the types in the System namespace. In addition to providing the base data types, the System namespace has classes such as exception classes and the Math class (for computations).
System.Collections	Contains classes and interfaces used to define collections of objects, such as the ArrayList, CollectionBase, and SortedList classes. This namespace is typically used to create collection classes and also contains many generic collection classes.
System.Data	Contains the classes for ADO.NET. You need references to this namespace when creating data binding in VSTO objects. Defined types in this namespace include the IDbConnection interface, IDataAdapter interface, and DataSet class.
System.IO	Contains the classes for reading and writing files synchronously or asynchronously. Objects defined in this namespace include the File, Directory, and Stream classes.

(continues)

Table 3.2. *Popular Namespaces in the .NET Framework Class Library (Continued)*

Namespace	**Description**
System.Text	Contains the StringBuilder class and supports various String manipulation functions, such as insert or remove text and others. You do not need to create a new String object, which the concatenation operator (&) implicitly does, in order to modify a string.
System.Windows.Forms	Contains a number of control classes that can be added to a form to create rich GUI applications. The controls include DateTimePicker, Textbox, Button, and ListBox.
System.Xml	Used for processing XML. This namespace includes a reader for parsing XML and classes such as XmlDocument and XmlNode.

and then click OK to add the reference and close the dialog box. Figure 3.6 shows the Add Reference dialog box in Visual Studio. You can also set references to component object model (COM) type libraries or browse to a particular assembly on your system. All references to your project



Component Name	Version	Runtime	Path
System	2.0.0.0	v2.0.50727	C:\WINDOWS\Microsof.
System.Configuration	2.0.0.0	v2.0.50727	C:\WINDOWS\Microsof.
System.Configuration.I...	2.0.0.0	v2.0.50727	C:\WINDOWS\Microsof.
System.Data	2.0.0.0	v2.0.50727	C:\WINDOWS\Microsof.
System.Data.OracleClient	2.0.0.0	v2.0.50727	C:\WINDOWS\Microsof.
System.Data.SqlXml	2.0.0.0	v2.0.50727	C:\WINDOWS\Microsof.
System.Deployment	2.0.0.0	v2.0.50727	C:\WINDOWS\Microsof.
System.Design	2.0.0.0	v2.0.50727	C:\WINDOWS\Microsof.
System.DirectoryServices	2.0.0.0	v2.0.50727	C:\WINDOWS\Microsof.
System.DirectoryServic...	2.0.0.0	v2.0.50727	C:\WINDOWS\Microsof.
System.Drawing	2.0.0.0	v2.0.50727	C:\WINDOWS\Microsof.
System.Drawing.Design	2.0.0.0	v2.0.50727	C:\WINDOWS\Microsof.
System.EnterpriseServices	2.0.0.0	v2.0.50727	C:\WINDOWS\Microsof.
System.Management	2.0.0.0	v2.0.50727	C:\WINDOWS\Microsof.
System.Messaging	2.0.0.0	v2.0.50727	C:\WINDOWS\Microsof.

Figure 3.6. *The Add Reference dialog box in Visual Studio*

are displayed in the References node in Solution Explorer. You might have to click Show All Files to view the references.

After setting the reference, you can add an Imports statement at the top of your code file so that you don't have to fully qualify an object in the namespace. You can optionally create an alias for the namespace as was shown in Listing 3.15. Some namespaces, such as the System namespace, are automatically included in your solution; therefore, it is not necessary to add a reference or create an alias for these namespaces.

VSTO and Managed Code

When you create VBA solutions for Word and Excel, your code typically is stored in the document or in an attached or global template. To access code in a template separate from your solution, you set a reference from your document to the template.

However, when you create VSTO solutions, your code is stored in an assembly. You can set references in your project to other assemblies, such as .NET Framework assemblies and interop assemblies, but you cannot set a reference to other VSTO solution assemblies. Only one VSTO solution assembly can be associated with a document or workbook. However, multiple documents can reference the same solution assembly. This is the case when you create multiple documents based on a VSTO-customized template. Note that because Outlook solutions are application-level, you can load multiple add-ins into Outlook. The same is true for add-ins created with VSTO 2005 SE.

Primary Interop Assemblies

Office applications, such as Word and Excel, are written in unmanaged code. For your VSTO solution (managed code) to interoperate with the unmanaged COM objects in the Office application, it must use an *interoperability assembly.* Visual Studio can create an interoperability assembly for you when you set a reference to a COM type library, but generating an interoperability assembly in this way is not recommended. Instead, you should use the official interoperability assembly that is provided by the publisher of the type library. This is known as the *primary interop assembly* (PIA). If the PIAs are installed on your computer and you set a reference to the type library, Visual Studio automatically loads the PIA instead of generating a new interop assembly.

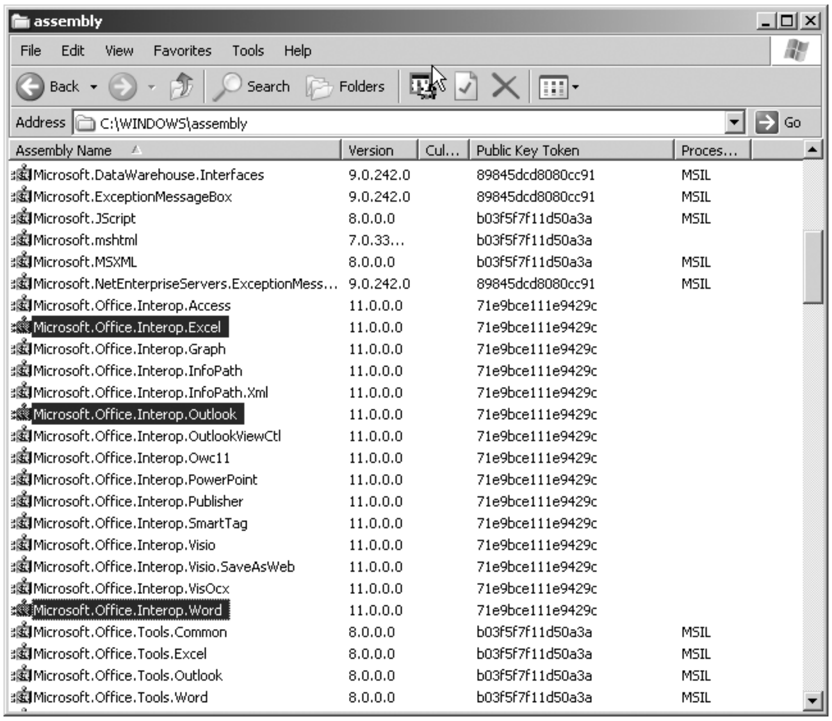
Microsoft provides PIAs for its Office applications. The name of the Excel PIA is Microsoft.Office.Interop.Excel.dll. Word and Outlook follow the same naming convention for their PIAs: Microsoft.Office.Interop.Word.dll and Microsoft.Office.Interop.Outlook.dll.

The PIAs get installed in your computer's GAC when you run a complete installation of Office 2003. Whenever you create a new VSTO solution, Visual Studio automatically adds to your solution a reference to the appropriate Office PIA and any of its dependent assemblies. You can view the contents of the GAC by opening the Windows\assembly directory of your root drive. Figure 3.7 shows the Microsoft PIAs.

The namespace for these PIAs is determined by the name of the assembly. VSTO automatically creates an alias for these namespaces, as shown in Table 3.3.

If the PIAs have not been correctly installed on your development machine because a complete installation of Office 2003 was not performed, you can run a reinstall or repair of Office 2003, assuming that .NET Framework 1.1 or later is already installed on your computer. Alternatively, you can use Add or Remove Features in the Maintenance Mode Options section of the Microsoft Office 2003 Setup, and click Next (see Figure 3.8). For information on redistributing the PIAs when you deploy your solutions, see Chapter 11.

On the next page of the Setup wizard, select Choose Advanced Customization of Applications, and then click Next.



Address C:\WINDOWS\assembly

Assembly Name	Version	Cul...	Public Key Token	Proces...
Microsoft.DataWarehouse.Interfaces	9.0.242.0		89845dcd8080cc91	MSIL
Microsoft.ExceptionMessageBox	9.0.242.0		89845dcd8080cc91	MSIL
Microsoft.JScript	8.0.0.0		b03f5f7f11d50a3a	MSIL
Microsoft.mshtml	7.0.33...		b03f5f7f11d50a3a	
Microsoft.MSXML	8.0.0.0		b03f5f7f11d50a3a	MSIL
Microsoft.NetEnterpriseServers.ExceptionMess...	9.0.242.0		89845dcd8080cc91	MSIL
Microsoft.Office.Interop.Access	11.0.0.0		71e9bce111e9429c	
Microsoft.Office.Interop.Excel	11.0.0.0		71e9bce111e9429c	
Microsoft.Office.Interop.Graph	11.0.0.0		71e9bce111e9429c	
Microsoft.Office.Interop.InfoPath	11.0.0.0		71e9bce111e9429c	
Microsoft.Office.Interop.InfoPath.Xml	11.0.0.0		71e9bce111e9429c	
Microsoft.Office.Interop.Outlook	11.0.0.0		71e9bce111e9429c	
Microsoft.Office.Interop.OutlookViewCtl	11.0.0.0		71e9bce111e9429c	
Microsoft.Office.Interop.Owc11	11.0.0.0		71e9bce111e9429c	
Microsoft.Office.Interop.PowerPoint	11.0.0.0		71e9bce111e9429c	
Microsoft.Office.Interop.Publisher	11.0.0.0		71e9bce111e9429c	
Microsoft.Office.Interop.SmartTag	11.0.0.0		71e9bce111e9429c	
Microsoft.Office.Interop.Visio	11.0.0.0		71e9bce111e9429c	
Microsoft.Office.Interop.Visio.SaveAsWeb	11.0.0.0		71e9bce111e9429c	
Microsoft.Office.Interop.VisOcx	11.0.0.0		71e9bce111e9429c	
Microsoft.Office.Interop.Word	11.0.0.0		71e9bce111e9429c	
Microsoft.Office.Tools.Common	8.0.0.0		b03f5f7f11d50a3a	MSIL
Microsoft.Office.Tools.Excel	8.0.0.0		b03f5f7f11d50a3a	MSIL
Microsoft.Office.Tools.Outlook	8.0.0.0		b03f5f7f11d50a3a	MSIL
Microsoft.Office.Tools.Word	8.0.0.0		b03f5f7f11d50a3a	MSIL

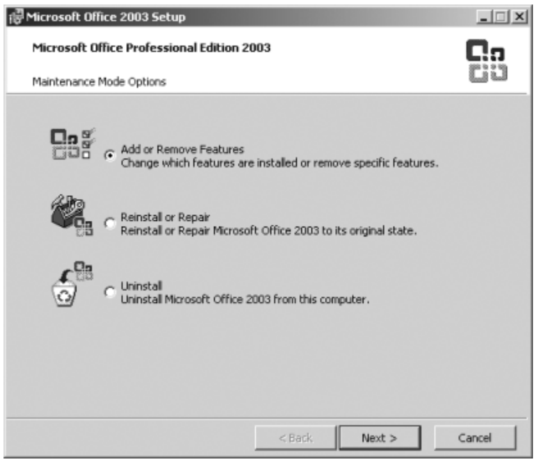
Figure 3.7. *Microsoft Office PIAs displayed in the GAC*

Table 3.3. *Excel, Word, and Outlook PIAs, Namespaces, and Aliases*

PIA	Namespace	Alias
Microsoft.Office.Interop.Excel.dll	Microsoft.Office.Interop.Excel	Excel
Microsoft.Office.Interop.Word.dll	Microsoft.Office.Interop.Word	Word
Microsoft.Office.Interop.Outlook.dll	Microsoft.Office.Interop.Outlook	Outlook

Set each of the following components to Run from My Computer. See Figure 3.9 for an example.

- Microsoft Office Excel | .NET Programmability Support
- Microsoft Office Outlook | .NET Programmability Support


Microsoft Office 2003 Setup
Microsoft Office Professional Edition 2003
Maintenance Mode Options
< Back
Next >
Cancel

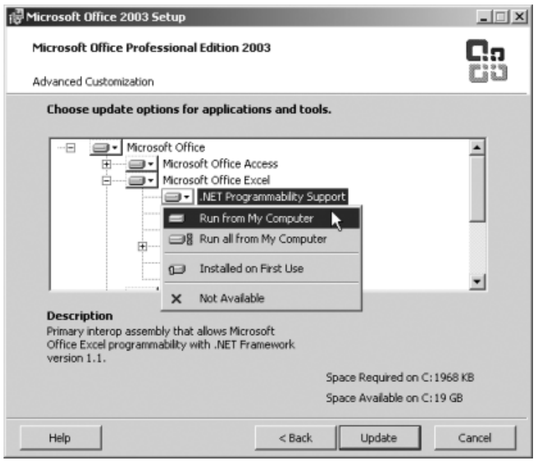
(●) Add or Remove Features
Change which features are installed or remove specific features.

( ) Reinstall or Repair
Reinstall or Repair Microsoft Office 2003 to its original state.

( ) Uninstall
Uninstall Microsoft Office 2003 from this computer.

Figure 3.8. *Maintenance Mode Options in Microsoft Office 2003 Setup*

- Microsoft Office Word | .NET Programmability Support
- Office Tools | Microsoft Forms 2.0 .NET Programmability Support
- Office Tools | Smart Tag .NET Programmability Support
- Office Tools | Microsoft Graph, .NET Programmability Support



Choose update options for applications and tools.

Description
Primary interop assembly that allows Microsoft Office Excel programmability with .NET Framework version 1.1.

Space Required on C: 1968 KB
Space Available on C: 19 GB

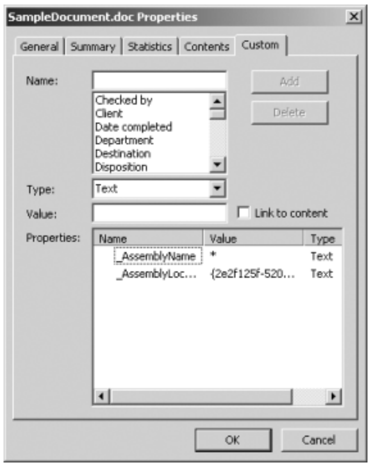
Figure 3.9. *Advanced Customization in Microsoft Office 2003 Setup*

Solution Assemblies

As you learned earlier in this chapter, an assembly is a collection of classes and functionality that is stored as an executable file (.exe) or a library (.dll). When you build a VSTO solution, the code is compiled and stored in a single assembly (DLL file) located in the \bin\debug (or \bin\release) directory of your solution. The assembly isn't stored inside the document, but the document does include an application manifest, which contains information about the name and location of the assembly. There are two manifests: an *application manifest* (which is stored in the document via an embedded control called the Runtime Storage Control) and a *deployment manifest,* which is located in the same directory where the assembly is deployed.

Even though you can associate a VSTO solution assembly only with a particular document (you can have only one solution assembly associated with a document), your solution assembly can reference other assemblies. For example, to add a common group of user controls to the actions pane, you can save a library of user controls in an assembly and then reference that assembly from multiple VSTO solutions.

How do you determine which VSTO assembly is associated with a particular document? You look at the document properties. You can view the document properties of a Word document or Excel workbook by clicking Properties in the File menu. A VSTO-enabled document has two custom document properties that indicate that the document is a VSTO solution. The first custom property is named _AssemblyName. If the value of _AssemblyName is an asterisk (*), it means that the document has an associated VSTO customization (assembly). The second custom property, _AssemblyLocation, stores the globally unique identifier (GUID) of the Runtime Storage Control (which contains the application manifest and information about where to locate the associated assembly). Figure 3.10 shows the custom document properties of a Word solution.



Name:

Checked by
Client
Date completed
Department
Destination
Disposition

Type: Text

Value:

Link to content

Properties:

Name	Value	Type
_AssemblyName	*	Text
_AssemblyLoc…	{2e2f125f-520…	Text

Figure 3.10. *Custom document properties of a Word solution*

Adding Customization to Existing Documents

You can attach an existing solution assembly to an uncustomized document or worksheet in one of two ways. The first is to add the _AssemblyName property with a value of asterisk (*), add the _AssemblyLocation custom property with the location of the deployment manifest as its value, and then save, close, and reopen the document. The second approach involves using the AddCustomization method of the ServerDocument class. These techniques are described in more detail in Chapter 13, Advanced Topics in VSTO 2005.

Running VSTO Solutions

For a VSTO solution to run, the assemblies must have full trust permissions. On the development machine, full trust is automatically granted to your solution's assembly and any referenced assemblies whenever you build the project. The evidence used for this trust is based on location (the URL of the assembly). When you deploy your solution, you need to grant full trust to the document and the assembly by using a strong name or by digitally signing the assembly. This procedure is discussed in greater detail in Chapter 11.

As a developer, you can run your customization by pressing F5 or by clicking Start Debugging from the Debug menu in Visual Studio. You can also open the document or workbook that is stored in the /debug/bin directory of the location in which you saved your solution, or press Ctrl+F5 to run the solution without debugging it.

An end user can run a solution in one of two ways.

1. To run a document-level customization, the user opens the document or workbook that has an associated customization assembly. Alternatively, the user can create a new document or workbook that is based on a template that has an associated customization assembly.
2. To run an application-level customization (add-in), users have two options. They can open the application, which contains instructions to load an add-in when the application starts. Or they can manually enable the add-in in the COM Add-ins dialog box of the application.

Summary

We began this chapter with an introduction to managed code and object-oriented programming. You looked at classes and learned how you can instantiate a class to create an object and then create properties, methods, and events for your class. You then learned how to create partial classes, generic classes, and interfaces. Next, you looked at the .NET Framework and learned that the common language runtime provides a framework for running managed code that includes compilation services, exception handling, reflection, and memory management. We looked at some common namespaces in the .NET Framework class library. Finally, we put this all together by looking at how a VSTO solution uses primary interop assemblies and how solution assemblies are created and run.

Review Questions

1. How do objects differ from classes?
2. What is the default constructor of any class in Visual Basic 2005? How can you override and overload a constructor?
3. How do Windows Forms and VSTO use partial classes in their programming models?
4. Define the three features that an object-oriented language must support.
5. What is IL? Why is it important to .NET programming?
6. Name three services provided by the common language runtime.