Visual Basic 2008 Programming

Business Applications with a Design Perspective

Jeffrey J. Tsay

Table of Contents

Chapter 12

Object-Based Programming

Forms and controls are recognized as objects. So, you have been working with objects since the first day you worked with VB. You can see these objects while you are designing your project. There are also objects such as the stream reader and the dataset of ADO.NET that are not visible on your form. All these objects have been defined and provided by others (Microsoft), not developed by yourself.
This chapter takes a different angle. You will learn how to develop your own templates of objects that you and other programmers can use. Before you can develop these objects, you will need to learn object-specific terminology. Section 12.1 discusses some of these terms. The remaining sections of this chapter deal with features that you can implement for the objects you develop.

After completing this chapter, you should be able to:

- Explain the relationship and differences between a class and an object,
- Develop code to create interfaces (properties, methods, and events) for a class,
- Implement additional features for a class, including enumerated constants, setting initial property values, raising events, and raising errors for improper uses of the object, and
- Create and use an object from the class you have developed.

12.1 Classes and Objects: Basic Concepts

You have seen and worked with objects such as controls, the stream reader, and ADO.NET. An object consists of code and data that work together as a unit. As a programmer who uses objects, you can think of objects as black boxes because you do not know how they work internally. You do not see any code of these objects. As you are aware, however, objects do have defined behaviors. They provide properties and methods that you can use to perform desired activities. They also recognize events to which you can write code in response. These defined features and behaviors are the interfaces of the object, and are exposed to its outside world (externally). All other code and data of the object are insulated from other programs (kept internally). This arrangement is recognized as *encapsulation*.

Object and Class Defined

Objects are derived—or more precisely, instantiated—from classes. A *class* is a code template or blueprint that defines the characteristics of an object; therefore, an object is a special instance of a class. To differentiate between a class and an object, consider the TextBox icon in the Toolbox and a text box in your form. You know the general features that you can derive from the text box while it is in the Toolbox. The appearances and behaviors of the text boxes you draw on a form can be quite different from each other. Each text box on the form is a special instance of the text box in the Toolbox. The TextBox icon in the Toolbox is a template and, thus, a class. On the other hand, a text box in the form is a special instance of the text box class and, thus, is an object.

In daily language, people tend not to differentiate an object from a class. For example, an instance of the text box class is usually referred to as a text box, or just the text box, which in turn can really mean the text box class itself. Such references are harmless because a clear differentiation between the two is not really necessary in their context. In this chapter, however, you do need to

understand the difference between the two. In Section 12.2, you will learn how to build a class—that is, a template. You will then create special instances of that class; that is, objects that your program can use.

Instance and Static Members

As you already know, objects have methods and properties that are defined in the objects' code template, class. Methods and properties are collectively recognized as *members* of the class. Depending on how it is declared in the class, a member may or may not be instantiated. Members that can be instantiated are recognized as *instance members*. Members that cannot be instantiated (can have only one copy for the class) are recognized as *static members*. Typically, members are declared as instance members. Static members are used for special purposes, as you will see later in this chapter.

Advantages of Object-Oriented Programming

When you start to build your own classes and use your own objects, your thinking process and the resulting code can be quite different from the programs you write in the "traditional" way. In essence, you will be doing object-oriented programming. Instead of attacking the programming problem directly, you will be thinking in terms of building a class to solve a generic class of problems and creating an object from the class to handle the problem at hand. While you are developing the class, you are one layer away from a particular programming problem.

Why build classes? Why use objects? The following are the advantages of this programming approach:

- *Encapsulation:* As already explained, each object keeps its own data, free from the interference of any part of the code in a project. The performance and accuracy of an object is independent of the other code in a project.
- *Code Reusability and Maintainability:* A class/object has well-defined interfaces (properties, methods, and events) and boundaries (encapsulation). You can easily incorporate a new object/class into your project. These interfaces are all you need to know about the class/object to use the class properly. In addition, code update and maintenance can be much more convenient. For example, if there is a change in the class/object, the new version can easily replace the old one. When the class/object is provided through a Dynamic Link Library (DLL), replacing the old DLL with the new DLL is all it takes to update all the projects using that class. A project that is compiled using objects provided by a DLL does not embed (statically link) the code of these objects, but rather just makes references to them. These objects are linked to the project dynamically at run time. There is no need to tear down or recompile the project. Code reusability is further enhanced when a class is *inheritable*. An inheritable class allows another class to extend its methods and properties. This feature is recognized as *inheritance*. The first class serves as the base class, and the second (new) class is the derived class. As a programmer, you can add functionality in the derived class without the need to modify the code in the base class; therefore, inheritance greatly enhances code reusability. Inheritance is discussed in details in Chapter 13, "Object-Oriented Programming."
- *Uniform Data Validation Rules:* An interesting application of objects in business is their representation as a business entity. For example, an object can be used to represent a student, an employee, or a product. This appears to be a very different use compared with a text box as an object. All the data associated with such a business object can then be coded as the properties of the entity. You can code all the data validation rules for each property in the object's property procedures as explained in Section 12.2. A company can require that all the programs dealing

with the business entity (object) use that object instead of using their own definition. Imposing such a requirement on all programs will result in uniform data validation rules. The advantages should be obvious: no unexpected exception will occur, and any change of validation rules can be revised in only one location—where the class is defined. This point should become clear after you complete Section 12.2.

- *Easier Project Management:* A big project can be more easily divided into smaller subprojects defined in terms of objects/classes that can be assigned to project team members. Each member can more easily focus on the member's own assignment because the interactions of the member's products are defined by the interfaces of the objects. These subprojects can then be assembled and tested by focusing on the behaviors of the interfaces.

12.2 Building and Using a Class

So, how do you build a class from which you can create objects? You build a class by writing code in a *class module*. You can think of a class module as a form module without its visual element; that is, a class module behaves like the code window of the form.

Differences Between the Class Module and the Standard Module

The class module is also similar to the standard module in that they both can contain only code, with no visual element; however, they are different in several respects:

- Each standard module can have only one copy in a project, and it can contain unrelated data and code (variables and procedures). On the other hand, each class module can have multiple instances (objects) in the same project, but it should not contain unrelated code or data.
- A standard module exists for as long as the project runs. An object (instance of a class) exists when it is created but is destroyed when it is out of scope. For example, if an object is created with an object variable in a procedure, the object is destroyed after the procedure ends. In addition, an object can be destroyed by being set to Nothing, such as `ObjVar = Nothing`. Note that the object may continue to reside in memory until the *garbage collector* (discussed in Section 12.2) is invoked and reclaims the resources that the object used. On the other hand, there is no way to destroy a standard module in a project other than ending the project.
- Public Sub and Function procedures in standard modules are recognized as Subs and Functions accordingly. To invoke a procedure in a standard module, your code makes a reference to the procedure name; however, Public Sub and Function procedures in a class module are methods of the object. They can be accessible only when the object exists. To invoke a method in a class module, your code must qualify the name of the method with the object name, not the class name; that is, the code must have the following syntax:

```
Object.Method
```

- Public variables in a standard module are accessible to all other modules in the duration of the project. Public variables of a class are properties of the object; thus they are accessible only when the object exists, and must be referenced with the object name as the qualifier. That is, you refer to the Public variables of a class module by the following syntax:

```
Object.Variable
```

  Note again that the qualifier of the variable is the object name, not the class name. Recall that you code the Text property of a text box named txtEmployee as follows:

```
txtEmployee.Text
```

  not

```
TextBox.Text
```

- Because public variables and procedures (subs and functions) are properties and methods of the object, there can be as many copies of these properties and methods as the number of instances (objects) of the class. These copies are independent of each other. In contrast, there is only one copy of the variables and procedures for each standard module in the project.

These differences are summarized in the following table.

Difference in	Class Module	Standard Module
Number of copies in a project	Multiple copies of the same module can be loaded.	Only one copy is loaded.
Data and code relationship	All data and code in one module should be related to the same class.	Unrelated code and data can exist in the same module.
Life duration	An object exists when it is created from the class module, and disappears when it is destroyed.	A standard module exists throughout the life of the project.
Copies of Public procedures	As many as the number of the objects created from the same module.	Only one copy exists.
Reference to the Public procedures	`Object.Method` (A Public procedure is a method of an object and can be accessible only when the object exists.)	`ProcedureName` or `ModuleName.ProcedureName`.
Copies of Public variables	As many as the number of the objects created from the same.	Only one copy per project.
Reference to the Public variables	`Object.Variable` (Public variables are properties of the object and can be accessible only when the object exists.)	`VariableName` or `ModuleName.VariableName`.

Recall that properties and methods of a class are collectively recognized as members of the class. Note that the above discussion does not apply to *static members* of the class. Static members are declared with the *Shared* keyword. A class has only one copy of each static member. All instances of the class (objects) share the same static members of that class. The lifetime of the shared (static) members does not depend on any instance of the class. In addition, static members can only be referenced by the class name as follows:

```
ClassName.Member
```

Adding a Class Module to a Project

To build a class, you need to add a class module to your project. Adding a class module is similar to adding a form and can be done by following these steps:

1. Click the Project menu in the IDE.
2. Click the Add Class option to open the Add New Item dialog box.
3. Enter a name for the class in the Name box.
4. Click the Add button. A code window with the following code template should appear in place of the form.

```
Public Class ClassName
End Class
```

You can then place code in the code window (class module) to build a class. You can include as many class modules as you need in a project. Although you can place multiple classes in a class module by

enclosing code within pairs of "Class…End Class" statements, this practice is not recommended. You may have difficulty in the future with locating the classes that you previously developed.

Scope of Class Modules

In general, a class is declared as Public. As such, it is accessible to all other modules in the project as well other projects. You can also declare a class as Friend. In such a case, the class is accessible only to the current project.

Creating the Fixed Asset Class

To illustrate how a class can be created and used, consider the creation of a FixedAsset class step by step. Suppose you would like to create this class with the following interfaces:

- Three properties:
  - Cost
  - Estimated life (in years)
  - The depreciation (accounting) method
- One method: net book value. This method will return the net book value of the fixed asset using the aforementioned three properties and a parameter to be discussed next.

How do you proceed? In general, the major steps are as follows:

1. Add a new class module to a new project, and assign a proper name. This was explained in the preceding subsection.
2. Add property procedures to create properties.
3. Add Public Sub and/or Function procedures for methods.

The following discussion assumes that you have added a class module to a new project and named it FixedAsset. All code should be placed inside the class definition:

```
Public Class FixedAsset

End Class
```

Creating a Property

To create a property, start your code in the class module with the following syntax:

```
Public [ReadOnly|WriteOnly] Property PropertyName() As Type
```

For example, you can declare an Account property as follows:

```
Public Property Account() As String
```

Typically, a property is expected to return a single value. If it is expected to return an array, a pair of parentheses should follow the type declaration of the property definition (similar to the header of a function). For example, if the Accounts property is expected to return a string array, it should be declared as follows:

```
'  Specify that the Accounts property will return an array
Public ReadOnly Property Accounts As String()
```

When you press the Enter key at the end of the statement, the IDE automatically provides you with the following code template:

```
Public Property PropertyName() As Type
    Get

    End Get
```

```
        Set(ByVal Value As Type)

        End Set
    End Property
```

The Get procedure is used to return the value of the property; the Set procedure allows the client code (the code that uses an instance of this class) to set the value (setting) for the property.

Notice that the Property name can be declared with Public, Friend, Protected (discussed in Chapter 13), and Private access. The default is Public. A Public property can be accessible to other modules. Notice also that you can place code only in either the Get or the Set procedure within the property definition. Inside the Set procedure, you should assign the parameter (named Value) to a variable that is private to the class module. You use a Private (instead of Public) variable so that the data passed to the class/object is insulated from other modules. In this way, data can be encapsulated.

In this example, you want the fixed asset class to have a Cost property; therefore, in the class module, you will create the following code template first:

```
    Public Property Cost() As Double
        Get

        End Get
        Set(ByVal Value As Double)

        End Set
    End Property
```

To keep a private copy of the Cost property, you will need to declare a class level Private variable to be associated with the property. For example, you can make the following declaration:

```
    Private mCost As Double
```

The Set procedure can be coded as follows:

```
        Set(ByVal Value As Double)
            mCost = Value
        End Set
```

Using the Class and the Property

To see the effect of this property procedure, you need to make changes to the project's form. First, draw a button on the form, name it **btnCompute,** and set its Text property to **Compute**. In the button's Click event procedure, code the following:

```
    Private Sub btnCompute_Click(ByVal sender As System.Object, ByVal e As
      System.EventArgs) Handles btnCompute.Click
        Dim Land As New FixedAsset()
        Land.Cost = 100000
    End Sub
```

The Dim statement declares a variable Land of the FixedAsset type. It also creates a new instance of FixedAsset, and assigns it to Land. The creation of the Cost property in the FixedAsset class module now allows you to set the Cost property of Land to 100,000.

Compare the preceding code with what you learned in Chapter 9, "Database and ADO.NET," when working with ADO.NET objects. You should notice that there is no difference between the way you declare and use an object from a class you create for yourself and that from a class provided by others (as in the case of ADO.NET).

Returning the Property Value

The Set procedure allows the code in other modules to set the property value of the object. What do you code to make the object return the value of the same property? For example, in your form, you would like to inspect the value of the Cost property by coding the following:

```
MsgBox("Cost of the land is " & Land.Cost)
```

What code/procedure do you need to add in the class module to enable this? (*Note:* Go back to the class module. All code from this point on should be placed in the class module.) It is the Property Get procedure, as mentioned previously, in the Cost property definition. To enable the Cost property to return a value, you can code the following:

```
Get
    Cost = mCost 'Return the property setting
End Get
```

Note that the Cost property name is placed on the left side of the assignment statement so that the property will return a value. This is similar to the way you write code for a function to return a value. You can also use the Return statement to return the value. In that case, you will code:

```
Get
    Return(mCost)  'Return the property setting
End Get
```

Also note that mCost is the Private variable that accepts the value of the same property in the Set procedure discussed previously. Variables that are used to associate their values with Public properties such as mCost with Cost are recognized as the *private copies* of the properties. They are sometimes referenced as member variables and are typically named with a prefix of m or mvar.

Read and Write Capabilities

In general, it is said that the Set procedure enables the code to write (set) the value for the property, whereas the Get procedure enables the code to read (return) the property. As shown previously, when you declare a property without either the *ReadOnly* or *WriteOnly* specification, it is both read- and write-enabled. The IDE editor provides both the Get and Set procedure templates. You can declare a property to be ReadOnly or WriteOnly. In that case, the editor will provide only either the Get or the Set procedure depending on your specification. The read-only specification can be useful for properties such as Count or Balance that should be maintained (computed) in the object internally. The following code shows an example ReadOnly property definition for Balance:

```
Private mBalance As Double
Public ReadOnly Property Balance() As Double
    Get
        Return (mBalance)
    End Get
End Property
```

The Complete Property Procedures

You can proceed in the same manner to code all other property procedures for the other properties (life and depreciation method) as initially planned. The complete code for the property procedures of the FixedAsset class appears as follows:

```
'Declare private copies of the properties
Private mCost As Double
Private mLife As Double
Private mDepreMethod As Integer
'Procedures for Cost property
Public Property Cost() As Double
```

```
        Get
            Cost = mCost
        End Get
        Set(ByVal Value As Double)
            mCost = Value
        End Set
    End Property
    'Procedures for Life property
    Public Property Life() As Double
        Get
            Life = mLife
        End Get
        Set(ByVal Value As Double)
            mLife = Value
        End Set
    End Property
    'Procedures for DepreMethod property
    Public Property DepreMethod() As Integer
        Get
            DepreMethod = mDepreMethod
        End Get
        Set(ByVal Value As Integer)
            mDepreMethod = Value
        End Set
    End Property
```

Creating a Method

How do you create a method for a class? The code for a method of an object is no different from a typical general Sub or Function procedure. In fact, you use exactly the same keywords: Sub and Function. To illustrate, you will create the NetBookValue method for the FixedAsset class to compute, and return the net book value for the asset. In general, the net book value is computed by the following formula:

Net Book Value = Cost - Accumulated Depreciation

The amount of accumulated depreciation depends on the number of years the asset is in use and the depreciation method. For simplicity, assume the company uses only two accounting depreciation methods: no depreciation (0), and straight-line depreciation (1). The net book value of an asset can be computed as follows:

- If DepreMethod is 0, no depreciation needs to be taken for the asset, such as land. The asset's net book value should be the same as its cost.
- If the DepreMethod is 1, the straight-line depreciation is used. The annual depreciation can be computed by the following formula:
  Annual Depreciation = (Cost - Salvage Value) / Life in Years

For simplicity again, assume a zero for the salvage value. The annual depreciation can be computed as follows:

Annual Depreciation = Cost / Life in Years

The accumulated depreciation will be computed as follows:

Accumulated Depreciation = Years in use × annual deprecation

Note that the maximum amount of accumulated depreciation is the cost of the asset. If the number of years the asset in use is greater than the asset's life, the net book value should be zero because the asset

has been fully depreciated; otherwise, the net book value should be equal to the cost minus the annual depreciation times the number of years in use. The years in use should be passed as a parameter to the NetBookValue procedure (method) so that it can perform the computation.
The code for the NetBookValue method should be placed in the class module and should appear as follows:

```
Public Function NetBookValue(ByVal Years As Integer) As Double
    If mDepreMethod = 0 Then
    ' This deprecation method does not depreciate the asset.
        NetBookValue = mCost
    Else
    ' Straight line depreciation method
        If Years >= mLife Then
    'Asset has been fully depreciated; book value should be zero.
            NetBookValue = 0
        Else
        ' Compute net book value by subtracting accumulated
        ' depreciation from the cost
            NetBookValue = mCost - Years * (mCost / mLife)
        End If
    End If
End Function
```

Note that the parameter, Years (representing years in use) has to be passed to the method (function) for the method to compute the net book value. Also, carefully examine the variables used in the formula to compute the net book value. Both the variables mCost and mLife are variables private to the FixedAsset class. Where are the sources of their values? They obtain their values when your code sets the values for the Cost and Life properties through their respective Set procedures. The flow of data can be depicted as in Figure 12-1.

Different Types of Methods

As you can see, when invoked, your NetBookValue method returns a value, the net book value; therefore, you code it as a function. Not all methods need to return a value. If you need a method that only brings about a result, such as displaying data or moving an image, you can code it as a Sub.

Differences Between Methods and General Procedures

So, how are the methods in class modules different from those general procedures in standard modules? In logic and in syntax, there really is no difference; however, as noted, the way the data are handled can and should be different. When designing your code, you always want to encapsulate the data as much as possible; that is, you will take care to eliminate contamination (unintentional interference) in the data you use in a procedure or a method. When dealing with the class/object, you accomplish this by encapsulating the properties. When working with procedures in the standard module, you will need to pass all required data elements as parameters. The number of parameters passed to a general procedure will tend to be more than that for a method, all other things being equal.
Another difference is the way that a method and the procedure are invoked. To invoke a method, you use the following syntax:

```
Object.Method(Parameter list)
```

whereas you usually invoke a procedure with the following syntax:

```
[ModuleName.]ProcedureName(Parameter List)
```

Figure 12-1
How a property setting is passed and used

```
'***********Code in the form*******************
Public Class Form1
    Private Sub btnCompute_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles btnCompute.Click
        Dim Land As New FixedAsset()
        Land.Cost = 100000
    End Sub
End Class

'**********Code in the Class Module**************************
Public Class FixedAsset
    'Declare private copies of the properties
    Private mCost As Double
    Private mLife As Double
    Private mDepreMethod As Integer
    'Procedures for Cost property
    Public Property Cost() As Double
        Get
            Cost = mCost
        End Get
        Set(ByVal Value As Double)
            mCost = Value
        End Set
    End Property
    'The NetBookValue Method
    Public Function NetBookValue(ByVal Years As Integer) As Double
        If mDepreMethod = 0 Then
            ' This deprecation method does not depreciate the asset.
            NetBookValue = mCost
        Else
            ' Straight line depreciation method
            If Years >= mLife Then
                'Asset has been fully depreciated; book value should be zero.
                NetBookValue = 0
            Else
                ' Compute net book value by subtracting accumulated
                ' depreciation from the cost
                NetBookValue = mCost - Years * (mCost / mLife)
            End If
        End If
    End Function
```

These statements in the form create a Land object of the FixedAsset type (class) and assign 10000 to the Cost property of Land.

Notice how the value (setting) of the Cost property flows inside the class module.

Note that to invoke a procedure, you refer to its name directly, or qualify it with the module name, whereas you qualify the method with the object name, not the class name. Each object (instance) of the same class has its own data, so you need not be concerned about data within a method being accidentally altered by another object. On the other hand, a procedure in a standard module deals with only one set of data; therefore, you will need to be more careful about the possibility that some of its data can be a leftover from previous invocation or accidentally altered by other code.

These differences are summarized in the following table.

Difference in	Procedures in a Standard Module	Methods in a Class Module
Number of parameters	More	fewer
Invocation (reference)	*ProcName* or *Module.ProcName*	*Object.Method* (not *Class.Method*)
Potential data contamination	Higher	Lower
Number of copies in a project	One	As many as objects created

Note, however, the preceding comparison pertains to methods declared as Public. Because these methods exist with each object (instance of the class), they are recognized as *instance methods*. You can also declare methods as Public Shared. As discussed previously in this chapter, Shared methods are static and are referred to as *static* (or *shared*) *methods*. Shared methods have characteristics that resemble procedures in standard modules; that is, they have only one copy per class (not per object), and can only be referenced by the class name, using the syntax:

```
ClassName.Method
```

Using the FixedAsset Class

Now that you have created a FixedAsset class, how do you use it in your project? As explained previously, you need to declare object variables of the FixedAsset class and then associate these variables with the instances of the class created using the New keyword. From that point on, using the objects you have created will be exactly the same as using all the objects you have seen before.

For example, suppose you would like to create two fixed assets: Land and Factory. Each will be assigned different values of their properties. You will then use the NetBookValue method to determine their net book values after 10 years in use, and use MsgBox to display the results.

You can rewrite the btnCompute Click event procedure shown in the preceding subsection to satisfy these requirements. The code can appear as follows:

```
' Code in the form
Private Sub btnCompute_Click(ByVal sender As System.Object, ByVal e As
  System.EventArgs) Handles btnCompute.Click
    ' Declare Land and Factory as variables of the FixedAsset class
    Dim Land As FixedAsset
    Dim Factory As FixedAsset
    Land = New FixedAsset()   'Create a new Land object
    ' Set properties for Land
    Land.Cost = 100000
    Land.Life = 1000
    Land.DepreMethod = 0
    Factory = New FixedAsset()   'Create a new Factory object
    ' Set properties for Factory
    Factory.Cost = 300000
    Factory.Life = 20
    Factory.DepreMethod = 1
    ' Display cost and net book value for Land
    MsgBox("Land cost is " & Land.Cost _
      & ". Net book value is  " & Land.NETBookValue(10)) '10 years
    ' Display cost and net book value for Factory
    MsgBox("Factory cost is " & Factory.Cost _
      & ". Net book value is  " & Factory.NETBookValue(10))
End Sub
```

In the code, both Land and Factory are instantiated as the FixedAsset class (type). Notice that you use the NetBookValue method for both Land and Factory objects after their respective properties have been set. This is done intentionally for you to be able to inspect the results. You should see each object retains its assigned cost, and each object has a correct net book value: 100,000 and 150,000, respectively. The data for each object are encapsulated and isolated from each other's so that the property settings in one object will not affect the others. Imagine the net book value for each object is computed in a standard module. You will need to take special care to separate the data for the land from the factory, won't you?

Default Property Setting and the Constructor

In this fixed asset example, if you forget to set the property value for DepreMethod for a fixed asset object, its default value will be zero—(that is, no depreciation)—because all numeric variables will be initialized to zero when an instance of the class is initialized. Most fixed assets are depreciable, however, so the straight-line depreciation (accounting) method would be a better default. Is there a way to set the straight-line depreciation (accounting) method as the default for each fixed asset?

When an object is being created from the class, the Sub New procedure in the class is automatically executed. This *New procedure* cannot be called by any other procedure (and therefore is executed only once for each instance) and is recognized as the *constructor*. This is the procedure that can be used to set the initial states for the object so that it is ready to perform.

You can use the New procedure to set the default value for a property. In this FixedAsset example, you can use it to set the default value of the DepreMethod property to 1 as follows:

```
Sub New()
    mDepreMethod = 1
End Sub
```

After you add this code in the fixed asset class module, all fixed asset objects will have a default straight-line depreciation method.

To see the effect of the preceding code, go back to the form. Add a button, name it **btnShow,** and set its Text property to **Show**; then add the following code:

```
Private Sub btnShow_Click(ByVal sender As System.Object, ByVal e As
  System.EventArgs) Handles btnShow.Click
    Dim Furniture As New FixedAsset()
    MsgBox("DepreMethod for furniture is " & Furniture.DepreMethod)
End Sub
```

Keep in mind that in this event procedure, you have not yet set any value for any property of the Furniture object. If, however, you run the project and then click the button, you will see that the MsgBox displays the following message:

```
DepreMethod for furniture is 1
```

Additional Uses of the Constructor

Similar to all other procedures, the New procedure can be overloaded; that is, you can write more than one New procedure with the same parameter list of different types or with a different parameter list (of the same types or different types). You can then use different New procedures to initialize an instance of the class. Typically, this feature is used to initialize property settings. In the fixed asset example, you can use it to allow the programmer to initialize the value for cost and life. The *additional* New procedure can appear as follows:

```
Sub New(ByVal pCost As Double, Optional ByVal pLife As Double = 0, Optional
  ByVal pDepreMethod As Integer = 1)
    mCost = pCost
    mLife = pLife
```

```
            mDepreMethod = pDepreMethod
        End Sub
```

This procedure will set mCost (the private copy of the Cost property) to the value of the first parameter. The second parameter is written as optional. When it is present, the value will be assigned to mLife, the Private copy of the Life property. If not, 0 will be assigned. Similarly, the absence of the third parameter will cause 1 to be the default setting for the DepreMethod property.

Notice that this New procedure overloads the previous one that sets default value for mDpreMethod. Only one of these New subs is called when an instance of the class is instantiated. The optional third parameter is included in the list so that the default setting for the DepreMethod property will be the same regardless of how an object of the FixedAsset class is instantiated.

In the calling procedure, the following code will initialize the Cost property of Furniture to $3,000:

```
            Dim Furniture As New FixedAsset(3000)
```

Notice the position of the parameter list. Although the New procedure in the class module specifies the parameter list, the list is placed in the pair of parentheses following the class name.

Building and Using a Class: A Recapitulation

The following table summarizes what you need to do to build a class and to use an object created from it.

Objective	**Action**
Code and actions in the class module	
To create a class	Add a class module to the project, and provide the class name in the Class statement; for example, `Public Class FixedAsset`
To create a property	Code the property definition with the following syntax: `Public [ReadOnly|WriteOnly] Property PropertyName() As Type`
To create a method	Write a Function that will return a value, or Sub that does not return a value.
To set default value for a property	Write code in the Sub New procedure.
To initialize property settings at the same time of object instantiation	Write additional New procedures, taking the initial settings as parameters.
Code in other modules (form, standard, or class modules)	

To use an object	Declare an object variable of the class created;, for example,
	`Dim Land As FixedAsset`
	Set the object variable to the object using the New keyword;, for example,
	`Land = New FixedAsset()`
	or simply
	`Dim Land As New FixedAsset()`

To reference a property or method of an object	Use the syntax: `Object.Property` or `Object.Method`; e.g., `Land.Cost`
To initialize the settings of properties at the same time of	Place the parameter list in the pair of parentheses after the class name where the instance is created;, for

object instantiation	example, `Dim Furniture As New FixedAsset(3000, 10)` Note that the overloading procedure must exist in the class to handle the initialization.

12.3 Adding Features to the Class

Perhaps you have noticed some desirable improvements in the preceding example. This section considers several of these improvements.

This textbook has advocated for the use of meaningful names for both constants and variables. The depreciation (accounting) methods were coded with numbers: 0 for no depreciation and 1 for the straight-line depreciation method. Is there a way to represent these numbers with meaningful names? Yes, you can with the Enum statement.

Enumerated Constants

The Enum statement has the following syntax:

```
[Public | Private] Enum name
    membername1 [= constant]
    membername2 [= constant]
    . . .
End Enum
```

where *name* represents a generic name for the data you are enumerating, and *membername* is the name for the specific value you are designating.

For example, instead of using 0 and 1 to represent the depreciation (accounting) method in the previous example, you can place the following code *in the FixedAsset class module*:

```
Public Enum DepreType
    NoDepreciation = 0
    StraightLine = 1
End Enum
```

Note that the assignment of values to the enumerated names is optional. When you just list all names, the first one in the list will be assigned a value of zero. All subsequent names will be assigned a value of 1 greater than the preceding one. You can also assign any unique value to any name. Again, any subsequent names without being assigned a value will be assigned with one increment of its preceding one.

Revising the Property Procedures for the DepreMethod Property

Because you are going to use DepreType to enumerate the available settings with the depreciation (accounting) method property, the property definition for the DepreMethod and its Set procedure should be revised as follows:

```
Private mDepreMethod As DepreType
Public Property DepreMethod() As DepreType
    Get
        DepreMethod = mDepreMethod
    End Get
    Set(ByVal Value As DepreType)
        mDepreMethod = Value
    End Set
End Property
```

Compare the header for the property definition and that for the Set procedure with the ones you had previously. The property definition (first line) and the parameter Value passed to the Property Set procedure now are declared to be the DepreType, instead of the Integer type. The code informs the procedure to expect/accept only one of the two values or names declared in the Enum statement. Note that the declaration for the property (first line) must be consistent with that for the parameter in the Set procedure; otherwise, the compiler will inform you of an error.

Effect of Enumeration on Code

How does this change affect your code? You can use the Enum data in both the class module and the form module. For example, in the class module, the NetBookValue method (function) contains an If block that tests the value of the mDepreMethod. The block of code can be revised as follows:

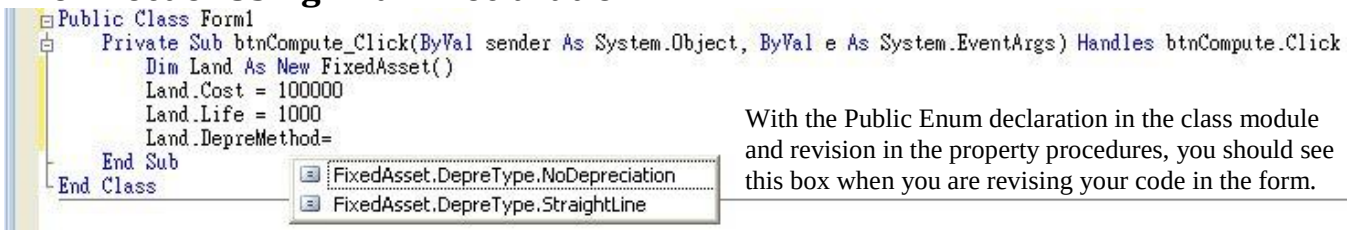
```
If mDepreMethod = DepreType.NoDepreciation Then
    ' This method does not depreciate the asset.
    NetBookValue = mCost
Else
    ' Straight line depreciation
    If Years >= mLife Then
        NetBookValue = 0
    Else
        NetBookValue = mCost - Years * (mCost / mLife)
    End If
End if
```

This revision should make the code more readable. In addition, you can now use the names in the Enum declaration in the form to set the value for the DepreMethod in the button Click event procedure. For example, you can now code the following:

```
Land.DepreMethod = FixedAsset.DepreType.NoDepreciation
Factory.DepreMethod = FixedAsset.DepreType.StraightLine
```

Again, this change should make your code clearer and more readable. Notice that the enumeration is qualified with FixedAsset because the enumeration is defined in that module. While you are revising the code, you should see the IDE IntelliSense displaying available choices as shown in Figure 12-2.

Figure 12-2
The Effect of Using Enum Declaration



Throwing Exceptions

In the preceding example, what if you accidentally set the value of the DepreMethod property to 2 for a fixed asset object? This value is apparently out of the valid range of the DepreType we declared for the property. The invalid setting will be used for the property, but you will not be alerted to the error.

Displaying Error Message in a Property Procedure

One alternative, of course, is to provide an error-checking routine in the Set procedure for the property of interest, and to display an error message when the data is out of range. If you do so, when you test

your program with an inappropriate property setting, you will encounter the message; however, the program continues to execute. You will have difficulty identifying the source of the error, and if another user uses the program, the error message will make no sense.

Throwing an Exception

A better alternative is to throw an exception in your class module, where you can specify the exception. An exception is a run time error. It can be caused by wrong logic in code, wrong data such as bad property setting as in the example, or an action by the user such as failure to insert a storage device in a designated slot. Throwing an exception emulates the exception you encounter in your code. For example, when your code attempts to read beyond the end of a file, the program will encounter the "System.IO.EndOfStreamException." If an error occurs when you throw an exception in a class module, the execution is halted at the point where your code attempts to set an invalid value. The error message will indicate the exception thrown in the class module. This can be helpful to other programmers who are using the class you have developed. How do you throw an exception? You use *the Throw statement*. The syntax appears as follows:

```
Throw New Exception(parameter list)
```

The exception can be a user-defined exception or a system defined exception. Most of the errors encountered are related to setting the property, and can be considered an argument error, as in this case. To help the programmer identify what went wrong, the exception thrown should be as specific as possible. In this example, the most appropriate exception to throw should be the ArgumentOutOfRangeException. Where can you find the list of exceptions that have been already defined? The most convenient place to locate them is the Debug menu. When you click the Debug menu and then select the Exceptions option, the Exceptions dialog box appears as shown in Figure 12-3. In most cases, you should be able to find the exception that is appropriate for the exception that you want to throw.

Incorporating Exception Throwing in a Property Procedure

If you apply this alternative to treating the error for the depreciation (accounting) method, you can revise the code for its Set procedure as follows:

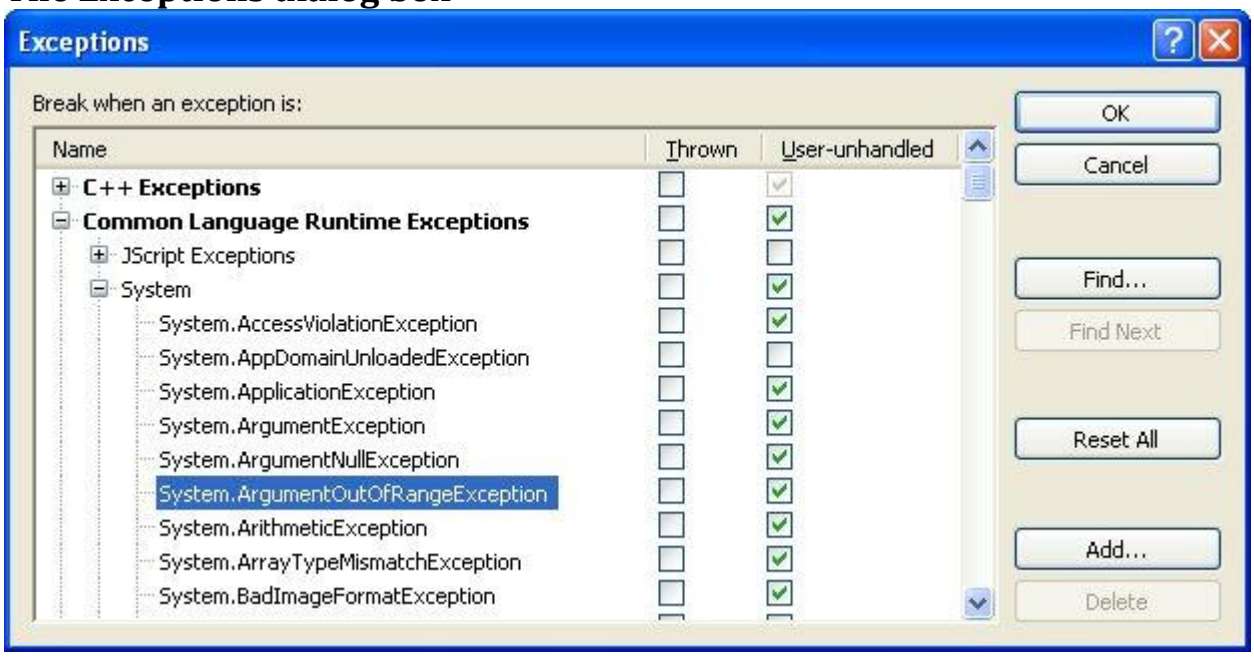
```
Set(ByVal Value As DepreType)
    If Value>=0 AndlAlso Value <= DepreType.StraightLine Then
        mDepreMethod = Value
    Else
        Throw New ArgumentOutOfRangeException("DepreMethod", "Value should be in the
          range of 0 and 1")
    End If
End Set
```

In this procedure, the If statement tests whether the Value parameter is within the valid range. If so, the property setting is assigned to the member variable mDepreMethod; otherwise, the "argument out of range" exception is thrown, with the message, "Value should be in the range of 0 and 1" displayed in the message box. You can explore the effect of this routine by using the following code in your btnCompute_Click event procedure (*Note:* To test the code, turn off Option Strict):

```
  Land.DepreMethod = 2
```

When you test your program, execution will be halted on this line with an error message indicating System.ArgumentOutOfRangeException with "Additional information: Value should be in the range of 0 and 1".

Figure 12-3
The Exceptions dialog box



This dialog box will appear when you click the Debug menu and select the Exceptions option. Click the Expand (+) button by "Common Language Runtime Exceptions" and then expand System. You should see a long list of exceptions from which you can find one that fits your situation.

Benefits of Throwing Exceptions

When you work for a company, most of the programming projects are fairly large. They are typically divided into smaller manageable subprojects, each assigned to different project team members. You will most likely be required to develop classes/objects to handle these subprojects. The classes/objects you have created will be used by your colleagues, while you will be using the classes/objects they have developed. You will appreciate it when your colleagues' classes and objects throw an exception to alert you to an error in your code. For the same reason, your colleagues will appreciate your classes that have similar implementations.

Additional Remarks

Admittedly, the fixed asset class you have created is simple, and can be enhanced in several ways. For example, you probably have noticed that various (accounting) depreciation methods exist. A complete fixed asset object should include all these methods for practical uses. Also, in reality, there are restrictive accounting rules governing the change of the depreciation method. You may want to incorporate an event such as "DepreMethodChanged" in the class when it detects an attempt to change the depreciation method. You can then place some code in that event to alert the user to the restriction when he or she attempts to change the depreciation method for an existing asset. In addition, you can also add methods for the class to compute the depreciation for the current year as well as the cumulative depreciation amount. These enhancements are left to you as an exercise at the end of this chapter. The implementation of events in a class is discussed in the next subsection.

Implementing Events in a Class

Recall that objects have three types of interfaces: properties, methods, and events. You have seen how properties and methods are created in a class, but how do you implement an event? In the class, you use the event declaration statement to declare an event, and the *RaiseEvent statement* to raise an event. The *event declaration statement* has the following syntax:

```
[Public] Event EventProcName(Parameter list)
```

where *EventProcName* is the name of the event procedure such as Click, and *Parameter list* is the list of parameters to pass to the procedure.

Notice that the event must be declared at the class level as Public. All events must be recognized by other modules in order to be used; therefore, an event declared as Private will not make any sense.

The following statement will declare an Insolvent event, with two parameters, both of the Double type:

```
Public Event Insolvent(Cash As Double, PaymentsDue As Double)
```

The *RaiseEvent statement* has the following syntax:

```
RaiseEvent EventProcName[(Parameter List)]
```

where *EventProcName* and *Parameter list* have the same meaning as in the event declaration. If there is no parameter, the pair of parentheses should be included.

The RaiseEvent statement will raise the event, and trigger the event procedure that is written to handle this event. The following code segment shows how the statement can be coded:

```
If Cash < PaymentsDue Then
    RaiseEvent Insolvent(Cash, PaymentsDue)
End If
```

After an event is properly declared and raised in the class module, it can be used in other modules similar to any typical event. The next subsection discusses additional details.

A Class with an Event: An Example

To illustrate how the two event statements are coded in a class, consider a simple example. Suppose you want to create a SortEngine class, in which you will provide a BubbleSort method that can be used to sort any array of the Integer type. Because you will be concerned about the amount of time that the method may take to sort the array, you would like to provide a means to inform the user of the progress in sorting. One way is to implement an event in the class. The event (call it PercentChanged) can be raised at each percentage of completion (1%, 2%, and so on), and will give the percentage as well as an estimated remaining time.

Bubble Sort Without an Event

You will begin with coding the sorting routine in a new project. Because the procedure is a method in a class module, you will need to do the following:

- Add a class module to the new project, and name it **SortEngine**.
- Enter the following code in the SortEngine class:

```
Public Sub BubbleSort(ByVal X() As Integer)
    Dim I As Integer
    Dim J As Integer
    Dim Temp As Integer
    For I = 0 To UBound(X) - 1
        For J = 0 To UBound(X) - I - 1
            If X(J) > X(J + 1) Then
                ' Out of order; swap
                Temp = X(J)
                X(J) = X(J + 1)
```

```
                    X(J + 1) = Temp
                End If
            Next J
        Next I
    End Sub
```

The bubble sort algorithm is explained in Appendix C, "Sorting and Searching."

Declaring the Event

The first step to implement an event is to declare the event using the Event statement. As described previously in this subsection, the PercentChanged event will give two values: the percentage completed, and the remaining time. The Event statement should be declared in the class-level declaration area as follows:

```
    Public Event PercentChanged(Percent As Integer, TimeLeft As Double)
```

As you can see, the declaration looks much like the header of a Sub procedure. The event must be declared as Public because you want it to be known to other modules in the projects. The Percent parameter is declared to be of the Integer type because you would like the number to be 1, 2, and 3 for 1%, 2%, and 3%, respectively. TimeLeft is declared to be a Double variable to show the estimated remaining time (in seconds). This statement should be placed in the general declaration area of the class module.

Raising the Event

The event should be raised each time the Percent variable increases by a percentile. The sorting time depends mainly on the number of comparisons to perform. As discussed in Appendix C, for an array of N elements, the total number of comparisons for the bubble sort algorithm can be computed by the following formula:

```
Comparisons = (N - 1) + (N - 2) + . . . + 2 + 1
            = N * (N - 1) / 2
```

The number of comparisons that has been performed can be computed in the inner loop with a variable (call it Counter) as follows:

```
        Counter += 1
```

The percentage can then be computed by the following formula:

```
        Percent = CInt(100 * (Counter / Comparisons))
```

Assume the time at which the computation started (StartTime) has been obtained; then the remaining time can be estimated as follows:

```
    TimeLeft = (Microsoft.VisualBasic.Timer - StartTime) / CDbl(Percent) * _
    CDbl(100 - Percent)
```

The preceding line explicitly converts Percent and (100 - Percent) to the Double type using the CDbl function. Both TimeLeft and StartTime should be of the Double type to keep track of time in seconds. (Recall that the Timer function returns a Double value representing the number of seconds elapsed since midnight.) Percent should be declared as Integer, however, as implied in the preceding discussion. The use of CDbl function makes the data type of all elements in the expression compatible.

For efficiency, the event should not be raised until the value of Percent actually changes. To detect the change, another variable, OldPercent, can be used. When the procedure (method) is invoked, both Percent and OldPercent will initially be zero; therefore, the following If statement should detect the change:

```
        If Percent > OldPercent Then
        End If
```

Inside the If block, the RaiseEvent statement can be used to fire the event. In addition, OldPercent should be assigned with the value of Percent, ready to check for another change in Percent; therefore, you can code the following:

```
    If Percent > OldPercent Then
        TimeLeft = (Microsoft.VisualBasic.Timer - StartTime) / CDbl(Percent) * _
          CDbl(100  Percent)
        RaiseEvent PercentChanged(Percent, TimeLeft)
        ' Revise OldPercent, so that this block will execute
        ' only when Percent has changed again.
        OldPercent = Percent
    End If
```

Notice that the two arguments enclosed in the pair of parentheses correspond to the event declaration statement. The variable names do not have to be exactly the same as in the event declaration. The relationship of the argument list between RaiseEvent and Event declaration is the same as that between Call and Sub.

The Complete Code for SortEngine

The complete code for the SortEngine class appears as follows:

```
Public Class SortEngine
    Public Event PercentChanged(ByVal Percent As Integer, ByVal TimeLeft As Double)
    Public Sub BubbleSort(ByVal X() As Integer)
        Dim I As Integer
        Dim J As Integer
        Dim Temp As Integer
        Dim N As Integer
        Dim Counter As Integer
        Dim Comparisons As Integer
        Dim Percent As Integer
        Dim OldPercent As Integer
        Dim StartTime As Double
        Dim TimeLeft As Double
        N = X.Length  'Number of elements in array
        Comparisons = N * (N - 1) \ 2 'Number of comparisons
        StartTime = Microsoft.VisualBasic.Timer 'starting time to sort
        For I = 0 To UBound(X) - 1
            For J = 0 To UBound(X) - I - 1
                ' Compare and order
                If X(J) > X(J + 1) Then
                    ' Out of order; swap
                    Temp = X(J)
                    X(J) = X(J + 1)
                    X(J + 1) = Temp
                End If
                ' Code to handle event raising
                Counter += 1 'Count number of comparisons
                Percent = CInt(100 * (Counter / Comparisons))
                If Percent > OldPercent Then
                    TimeLeft = (Microsoft.VisualBasic.Timer - StartTime) / _
                      CDbl(Percent) * CDbl(100 - Percent)
                    RaiseEvent PercentChanged(Percent, TimeLeft)
                    ' Revise OldPercent, so that this block will
                    ' execute only when Percent has changed again.
                    OldPercent = Percent
                End If
```

```
            Next J
        Next I
    End Sub
End Class
```

Declaring and Using the Event

To use the event created in a class, the object must be declared at the module (form) level with the WithEvents keyword. For example, assume an object named Sorter is created from the preceding class to sort arrays in a form. The Sorter object should be declared at the form/class level as follows:

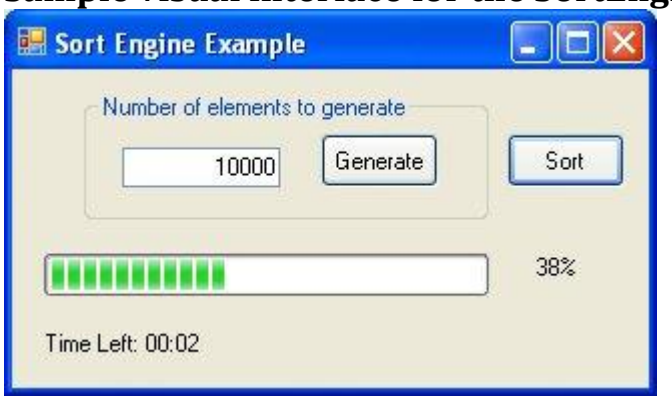
```
    ' This code is placed in the form that will use the SortEngine class
    Dim WithEvents Sorter As New SortEngine()
```

Note that after the preceding line is placed in the code window of the form, you should be able to find the Sorter object in the code window's object box and the event in the procedure box as shown in Figure 12-4. With this declaration, you can place code in the event in exactly the same manner as you can in all other events recognized by other objects.

Figure 12-4
An object variable declared with events

Once the Sorter object is declared WithEvents, you should be able to find it in the object box and its events in the procedure box.

Figure 12-5
Sample visual interface for the SortEngine project



This illustration shows how the PercentChanged event of the Sorter object can be used to display the progress in the Sorter's BubbleSort method. In essence, there is no difference in using the events of those objects provided by VB and the events raised in any class that you develop for yourself.

Completing the Example

To continue and complete the example, go back to the form; then draw a text box, two buttons, two labels, and a progress bar. The text box will be used to specify the upper bound for the array to sort. One button will be used to generate random numbers, and the other will be used to call the BubbleSort method. One label will be used to display the percentage; the other will be used to show the time remaining. The progress bar will be used to indicate the percentage of progress in sorting. The following table indicates how the properties of these controls should be set.

Control	Property	Setting	Remarks
Text box	Name	txtNumber	To specify the upper bound for the array
Button	Name	btnGenerate	To generate the specified random numbers
	Text	Generate	
Button	Name	BtnSort	To call the bubble sort method in the SortEngine class
	Text	Sort	
Label	Name	lblPercent	To display percentage of completion
Label	Name	lblTimeLeft	To show remaining time to sort
Progress bar	Name	prgPercent	To indicate percentage of completion

Figure 12-5 shows a sample visual interface in action.

When the user clicks the Generate button, the computer will generate an array of the Integer type with its upper bound equal to the number specified in the text box txtNumber. The code in the *btnGenerate_Click procedure* appears as follows:

```
Dim A() As Integer
Private Sub btnGenerate_Click(ByVal sender As System.Object, ByVal e As
  System.EventArgs) Handles btnGenerate.Click
    Dim N As Integer
    Dim I As Integer
    Dim Generator As New System.Random()
    If Len(Trim(txtNumber.Text)) = 0 Then
        MsgBox("Please enter a number")
        Exit Sub
    End If
    N = CInt(txtNumber.Text)
    ReDim A(N)
    For I = 0 To N
        'Generate a random number in the range of 0 and N - 1
        A(I) = Generator.Next(0, N)
    Next I
    prgPercent.Value = 0 'Clear the progress bar
    MsgBox(A.Length & " random numbers generated.") 'Tell the user, it's done.
End Sub
```

Note that for user feedback purposes, you also added two statements at the end of the procedure. One statement assigns a value of 0 to the Value property of the progress bar, and should clear the progress bar. The other statement displays a message box to inform the user of the number of random numbers generated.

When the user clicks the Sort button, your code will create a new Sorter object and then use its BubbleSort method to sort the array generated in the btnGenerate Click procedure. The ***btnSort_Click procedure*** can appear as follows:

```
Private Sub btnSort_Click(ByVal sender As System.Object, ByVal e As
  System.EventArgs) Handles btnSort.Click
    Sorter.BubbleSort(A)
End Sub
```

Because the object variable Sorter has been declared and created with New in the general declaration area, it should not be declared again. This procedure simply calls Sorter's BubbleSort method to sort array A.

How do you code the PercentChanged event? This event is triggered when the percentage of comparisons changes by a full percentile. You can use it to display the change in the labels and the progress bar. The code should appear as follows:

```
Private Sub Sorter_PercentChanged(ByVal Percent As Integer, ByVal TimeLeft As
```

```
    Double) Handles Sorter.PercentChanged
      Dim Remaining As DateTime
      lblPercent.Text = Percent & "%"
      prgPercent.Value = Percent
      Remaining = DateAdd(DateInterval.Second, TimeLeft, Today)
      lblTimeLeft.Text = "Time Left: " & Format(Remaining, "mm:ss")
      Application.DoEvents()
  End Sub
```

Notice that you should obtain the event procedure template from the procedure box, not by writing your own. There is no difference between handling this event and handling any events generated by any VB objects. Notice that TimeLeft provided by the Sorter object is a Double value in seconds. You'd like to display the time in the HH:mm:ss format; therefore, that value is converted to a Date/Time value using the DateAdd function. The result is assigned to the Date/Time variable, Remaining, which is declared at the beginning of this procedure and displayed with the Format function. Percent is used to set the progress bar, and is also displayed in lblPercent. Notice also that the Application.DoEvents statement is needed for the operating system to update the appearance (Texts) of the labels.

Test this project by trying several different numbers for the upper bound of the array to get a feel. For a computer with a 1.4 gHz processor with an upper bound set to 40,000, you should see that it takes a few minutes to sort the array. Note that the bubble sort algorithm is used in this example to illustrate how the PercentChanged event can be implemented. Recall that this algorithm is the least efficient among all available sorting algorithms. You can think of its use here as a representation of a process that takes a long time.

Why Events?

The sorting example illustrated thus far should give you a good appreciation of implementing events in classes. Without the PercentChanged event, a programmer using the SortEngine class will have no way to implement code to inform the user of the progress in sorting. One possible alternative is to rewrite the method (BubbleSort) so that it will take two labels and a progress bar as its parameters. It can then display the progress in these controls; however, such a design is inflexible. The programmer who uses the method will be forced to provide these controls. In addition, these controls must be of the classes of the label and the progress bar.

On the other hand, when the PercentChanged event is implemented in the class, the programmer who uses the class has complete flexibility in determining what to do with the object created from the class. The programmer can ignore the event or code the event in any manner the programmer sees fit. The combination of labels and the progress bar used in the preceding example is but one of the many possibilities.

Hooking Event Handlers

So far, you have been coding event handlers in the event procedures designated by the IDE designer. If you examine the procedure header again, you will find that each header ends with the following clause:

```
        Handles Object.Event
```

Does it have any significance? If you delete that portion from the header and test the event again, you will notice that the event procedure no longer gets invoked. This clause hooks the event handler (the event procedure you write) to the event trigger, the statement in the object that raises the event. Recall in the SortEngine class, you have a RaiseEvent statement. This is the statement that looks for an event procedure to execute (call). The *"Handles" clause* indicates which event(s) the event procedure handles.

You can actually use the same event procedure to handle more than one event as long as these events have the same parameter list. For example, suppose you have two text boxes named txtAmount1 and txtAmount2 that you want to respond the same way, such as check for numeric keys, when the user presses a key. You can actually write only one KeyPress event procedure but list the two events after Handles as follows:

```
Private Sub txtAmount_KeyPress(ByVal sender As System.Object, ByVal e As
  System.Windows.Forms.KeyPressEventArgs) Handles txtAmount1.KeyPress,
  txtAmount2.KeyPress
    Dim KeyAscii As Integer
    KeyAscii = AscW(e.KeyChar)
    Select Case KeyAscii
        Case Is < Keys.Space, Keys.D0 To Keys.D9
            'control key or numeric key; do nothing
        Case Else
            MsgBox("Numeric keys only, please.", MsgBoxStyle.Information)
            e.Handled = True 'suppress the key
    End Select
End Sub
```

Pay particular attention to the header. The sub name does not have to be associated with any particular control. It is the Handles clause that indicates what events that the procedure handles. Thus, the Handles clause allows you to hook events to their handlers at design time.

You can also hook an event with its handler at runtime by the use of the *AddHnadler statement*, which has the following syntax:

```
AddHandler Event, AddressOf Handler
```

In the preceding example, if you would rather hook the events to the event handlers at run time, you can remove the Handles clause and add the following two statements:

```
AddHandler txtAmount1.KeyPress, AddressOf txtAmount_KeyPress
AddHandler txtAmount2.KeyPress, AddressOf txtAmount_KeyPress
```

Of course, these two statements should be executed before the event procedures can be invoked. You can have more than one handler hooked to the same event. You can also remove the handler when it is no longer needed by the use of the RemoveHandler statement, which has the following syntax:

```
RemoveHandler Event, AddressOf Handler
```

Hooking Event Handlers with Events of Controls Created during Run Time

The Add Handler statement is most useful when controls/objects are created during run time. Consider the case of the data entry screen for cash flows that was discussed in Chapter 8. It was noted that the text boxes created during run time could not deal with events, but you are now equipped to solve this problem. How? Suppose you have created a KeyPress event handler suitable to handle the Keypress event for all these text boxes (say, txtCashFlow_KeyPress). You can declare a text box variable "WithEvents" at the class level (required); then in the procedure to create the text boxes, you can add an AddHandler statement to hook each of the events to the same handler. The code fragment should appear as follows:

```
'In the general declaration area
Dim WithEvents txtCashFlow As TextBox

'In the procedure that creates text boxes
For I =0 To N
    txtCashFlow = New TextBox() 'Create a new text box
    AddHandler txtCashFlow.KeyPress, AddressOf txtCashFlow_KeyPress
    'Other statements
Next I
```

The additional details in making the project work are left to you as exercise 12-19.

You may wonder how the AddHandler statement is different from the Call statement. In many ways, they are similar; however, recall that events are triggered from objects to which your code is expected to respond. In other words, the call is made from the object to the handler (the Sub you write). The AddHandler statement informs the object of which sub to use, while the Call statement informs your own routine of which sub to use.

12. 4Nested Classes

A class can contain one or more other classes, which can in turn contain yet others forming a class of *nested classes*. Usually, the *inner classes* are exposed as properties of the *outer classes*. This arrangement is similar to what you have seen in ADO.NET. For example, the dataset contains the datatable, which contains data rows and data columns, each in turn having its own methods and properties.

Scope of Inner Class

Recall that a class at the module level must be either Public or Friend. Inner classes, however, can have a narrower scope. They can be declared as Private. A Private class can be referenced only within the class that contains it. Private classes are typically used to support the operations of their outer classes. They tend to contain code that pertains only to the internal operations, and therefore should be encapsulated (hidden from the outside world).

Uses of Nested Classes

Is there any practical business application of nested class? Imagine a depositor object that has properties such as the depositor ID, address, and other profile information. In addition, it can have two accounts: checking and savings. Each of these accounts has its properties such as account number, balance, interest rate, compensating balance, and so on, as well as methods such as deposit and withdraw. These accounts logically belong in the depositor class, and should be treated as such. Notice that although the accounts are properties of the depositor, they have methods, so these properties themselves must be objects to be able to have methods. You can solve this problem in the following manner: Let the depositor class contain the account class with the required properties and methods; then the checking account and savings account objects can be instantiated from the account class, and present themselves as the properties of the depositor class. The following subsection illustrates how the code can be developed.

Developing the Depositor Class

To illustrate how to code the nested classes, let's use the depositor problem described previously as an example. For simplicity, the depositor will have the depositor ID, checking account, and savings account properties. Because the checking account and savings account properties are instantiated from the account class, you will write the account class inside the depositor class, creating a nested class with the depositor as the outer class and the account as the inner class. Call this new project **BankDepositor**.

Before writing any code, add a class module to this new project, and name it **Depositor**. You will first create the property for the DepositorID. This should be fairly straightforward and is left to you to complete. You can now pay full attention to creating the Account class. This is a class inside the depositor class. So, the code structure should appear as follows:

```
Public Class Depositor
    ' Statements for the Depositor ID property
    '
    Public Class Account
    End Class
End Class
```

Notice that the Account class is Public because it needs to be accessible to the code external to the Depositor class as properties.

Exposing the Account class as Properties of the Depositor

Recall that the depositor has two types of accounts: checking and savings, which are exposed as properties. Setting up these two properties is similar to that for other typical properties. Be aware that these two properties should be read-only, and be of the Account class. To create the CheckingAccount property, you can first set up an internal member (inside the depositor but outside the Account class) as follows:

```
    Private mCheckingAccount As New Account()
```

Notice that the variable has been instantiated with the New keyword, so mCheckingAccount is an instance (object) of the Account class (type). The read-only CheckingAccount property is defined as follows:

```
    Public ReadOnly Property CheckingAccount() As Account
        Get
            Return(mCheckingAccount)
        End Get
    End Property
```

The code in effect returns an instance of the Account class as the CheckingAccount property. Because Account is a class, CheckingAccount is an object with all the properties and methods that Account has (will have). In a similar fashion, you can create the SavingsAccount property. The code is shown later in this subsection.

Properties and Methods of the Account Class

Coding the properties and methods for the Account class is no different from that for a typical class. The class has two properties: AccountNo and Balance. Balance should be read-only so that the amount is changed by deposit and withdraw only, not by an accidental setting. The class has two methods: Deposit and Withdraw. The former increases the account balance; the later decreases.

Complete Code for the Depositor Class

The code for the Depositor class should appear as follows:

```
    Public Class Depositor
        Private mDepositorID As Integer
        Private mSavingsAccount As New Account()
        Private mCheckingAccount As New Account()

        Public Property DepositorID() As Integer
            Get
                Return(mDepositorID)
            End Get
            Set(ByVal Value As Integer)
                mDepositorID = Value
            End Set
        End Property
```

```
    Public ReadOnly Property CheckingAccount() As Account
        Get
            Return(mCheckingAccount)
        End Get
    End Property

    Public ReadOnly Property SavingsAccount() As Account
        Get
            Return (mSavingsAccount)
        End Get
    End Property

    Public Class Account
        Private mBalance As Double
        Private mAccountNo As Integer
        Public ReadOnly Property Balance() As Double
            Get
                Return(mBalance)
            End Get
        End Property

        Public Property AccountNo() As Integer
            Get
                Return(mAccountNo)
            End Get
            Set(ByVal Value As Integer)
                mAccountNo = Value
            End Set
        End Property

        Public Function Deposit(ByVal Amount As Double) As Double
            mBalance += Amount
            Return(mBalance)
        End Function
        Public Function Withdraw(ByVal Amount As Double) As Double
            mBalance -= Amount
            Return(mBalance)
        End Function

    End Class
End Class
```

Using the Depositor Class.

To illustrate how the Depositor class as developed can be used in code, go back to the form and create a visual interface as shown in Figure 12-6. The interface is used to keep track of the depositor's activities. For simplicity, assume the depositor is given. The user enters the amount, selects an account (checking or savings), and selects an action (deposit or withdraw). The account is updated accordingly. At anytime, the user can click the Balance button to find out the balance of the account of interest based on the radio button selected.

The following table shows the names of the controls used in code:

Control	Name	Remark
Text box	txtAmount	For the user to enter the amount

Radio button	rdbChecking	To indicate checking account (set its Checked property to True to make it the default)
Radio button	rdbSavings	To indicate savings account
Button	btnDeposit	To initiate deposit
Button	btnWithdraw	To initiate withdraw
Button	btnBalance	To show account balance

Because it is assumed that the depositor is given, the object is created with the New keyword at the form/class level where it is declared:

```
    Dim TheDepositor As New Depositor()
```

Figure 12-6
Visual interface for depositor activity

Referencing the Accounts in the Depositor Class

Recall that you created two Account properties (CheckingAccount and SavingsAccount) for the Depositor class. How should they be referenced in the module that uses the Depositor object? They are referenced literally:

```
    TheDepositor.CheckingAccount
    TheDepositor.SavingsAccount
```

To call the Deposit method for the CheckingAccount property, you will code:

```
    TheDepositor.CheckingAccount.Deposit(Amount)
```

Handling the Button Click Events

When the user clicks the Deposit button, the program should check which radio button is selected and deposit the amount to the proper account. The Withdraw and Balance button Click events can be handled similarly. The code should appear as follows:

```
Private Sub btnDeposit_Click(ByVal sender As System.Object, ByVal e As
  System.EventArgs) Handles btnDeposit.Click
    Dim Amount As Double
    Amount = Val(txtAmount.Text)
    If rdbChecking.Checked Then
        TheDepositor.CheckingAccount.Deposit(Amount)
    Else
        TheDepositor.SavingsAccount.Deposit(Amount)
    End If
End Sub

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

```
    Dim Amount As Double
    Amount = Val(txtAmount.Text)
    If rdbChecking.Checked Then
        TheDepositor.CheckingAccount.Withdraw(Amount)
    Else
        TheDepositor.SavingsAccount.Withdraw(Amount)
    End If
End Sub

Private Sub btnBalance_Click(ByVal sender As System.Object, ByVal e As
  System.EventArgs) Handles btnBalance.Click
    If rdbChecking.Checked Then MsgBox("Checking account balance is " & _
        TheDepositor.CheckingAccount.Balance)
    Else
        MsgBox("Savings account balance is " & TheDepositor.SavingsAccount.Balance)
          End If
      End Sub
```

Additional Remarks

The depositor class is used to illustrate how to code nested classes and create properties from the inner class. Of course, the class can be enhanced in several ways. Additional properties can be included at each level of classes. Events can also be added. For example, in the Withdraw method, you can create an NSF (insufficient fund) event, which is triggered when the amount to be drawn is greater than the account balance. The programmer using your Depositor class can then decide what to do with the situation, such as ignore or prompt for the user direction. These desirable refinements are left to you to complete.

Nested classes can make your code more complex. Use this kind of structure only when it is logical and necessary. Avoid creating very complex nested classes. They are harder to use. In general, a flat object model (class without nesting) is simpler and preferable. As an alternative to the example structure, you can actually make the Account class external to and independent of the Depositor class by moving the Account class outside the Depositor class. All the remaining code in the Depositor class can remain the same and will work exactly the same as when the Account was an internal class. The advantage of this structure is that the Account class can be used by any other modules without involving the Depositor class as the example showed.

Summary

- A class is a code template that defines the characteristics of an object. An object is a special instance of a class.
- The advantages of object-oriented programming include encapsulation, code reusability and maintainability, uniform data validation rules, and easier project management.
- Classes are developed in class modules. A typical class module differs from a form module in that it does not have the visual elements. A class module differs from a standard module in that multiple copies (objects) can be instantiated from the same class module, but there is only one copy of each standard module in the project.
- A class module can be added to the project in a way similar to adding a new form or a standard module.
- A class can be created by placing code within the Class…End Class block on the class module. Code in the class should be related to the purpose of the class and nothing else.

- Properties of a class can be created with Property procedures. The Get procedure returns the value (setting) of the property and the Set procedure allows the client code to set the value for the property. Data validation rules can be placed in the Set procedure to ensure the validity of the property setting.
- Public variables in a class are properties; however, use of public variables in place of property definitions is not advisable because it is not possible to implement data validation rules with public variable declarations.
- Public functions and subs in a class are exposed externally as methods of the class.
- There is no difference in instantiating and using the objects between classes developed in a project and classes provided by others, such as those provided by VB 2008.
- The Sub New method is the constructor of the class, and is automatically executed when an instance of the class is instantiated. It is typically used to initialize properties and data used in the class.
- The FixedAsset class is used as an example to show how elements of a class can be coded.
- The Enum… End Enum block is used to enumerate constants for any particular group of data with meaningful names.
- When code outside of a class can cause an error in the class, one good way to handle the situation is to throw an exception. This allows the developer of the client code to identify and correct the error. The names of most exceptions are already available in the system. The list of exception names can be found in the Exceptions option of the Debug menu although user-defined exceptions can also be created.
- To create an event for a class, the event must be declared at the class level. It can then be raised in a procedure with the RaiseEvent statement.
- To use the event of a class in another module, the instance of the class must be declared with the WithEvents keyword. The event procedure can then be coded in exactly the same manner as all events recognized by those controls discussed in the textbook.
- The SortEngine example is used to illustrate how an event can be implemented in a class and used in another module (form).
- The Handles clause at the end of an event procedure header hooks the event procedure (handler) with the event. You can place more than one event in the Handles clause. If you do so, the event handler (procedure) will be invoked when any of these events is raised.
- Event handlers and events can be dynamically hooked and removed with the AddHandler and RemoveHandler statements. The Cashflow data entry project in Chapter 8 was revisited to illustrate the usefulness of the AddHandler statement to hook event handlers with events associated with controls created during runtime.
- A class can contain one or more other classes, which can in turn contain yet others, forming a class of nested classes. Usually, the inner classes are exposed as properties of the outer classes. To expose an inner class as a property, declare a property as the inner class type; then treat the property in a manner similar to any other types of properties.
- The example of the Depositor class and its inner Account class is used to illustrate how to code and expose an inner class as a property of its outer class.

Explore and Discover

12-1. More Than One Class in a Class Module? Add a class module to a new form; then place the following code in the class module:

```
Public Class Class1
End Class
Public Class Class2
End Class
```

Next, place the following code in the form's Click event:

```
Dim MyObject1 As New Class1()
Dim MyObject2 As New Class2()
```

Run the project. Do you encounter any errors? You should not. You can have more than one class in a class module, and they will work just fine. The problem is how you will locate these classes in the future. Placing more than one class in a class module is not a good practice, although you are allowed to do so.

12-2. Timing of Creation and Destruction of an Object. Add a class module to a new project, and place the following code in the class module in the preceding exercise as follows:

```
Public Class Class1
    Sub New()
        MyBase.new()
        MsgBox("I'm being initialized")
    End Sub
    Protected Overrides Sub Finalize()
        MsgBox("I'm being destroyed")
        MyBase.Finalize()
    End Sub
End Class
```

Place the following code in the form:

```
Dim MyObject1 As Class1
Private Sub Form1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
  Handles MyBase.Click
    MyObject1 = New Class1()
End Sub
```

Notice the location of the object variable declaration.

Run the project. Do you see the "I'm being initialized" message? Now click the form. Do you see the message? This indicates that the Sub New procedure is called automatically when the object is being instantiated.

But do you see the "I'm being destroyed" message? Click the form's Close button. Do you see the message? The Sub Finalize procedure is automatically called when an object is *actually* destroyed. This also appears to suggest that objects are destroyed when their associated variables are out of scope. But wait. Check out the experiment in the following exercise.

12-3. Timing of Creation and Destruction of an Object. (continued from exercise 12-2) Revise the code in the form in the preceding exercise as follows:

```
Private Sub Form1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
  Handles MyBase.Click
    Dim MyObject1 As Class1
    MyObject1 = New Class1()
End Sub
```

Run the project and then click the form. Do you see the "I'm being initialized" message? But do you see "I'm being destroyed" message? Objects are initialized with the New keyword, but they are not necessarily automatically destroyed immediately after their associated names are out of scope although you no longer are able to reference to them.

Click the form's Close button. Do you see the "I'm being destroyed" message? Objects may not be destroyed until your project ends regardless of their scope.

12-4. You Can Raise Events in a Form, Too. Add a new form (Form2) to a new project. Draw a button on this form (Form2), name the button **btnClick**, and set its Text property to **Click**. Place the following code in this form:

```
Public Event ActionOccurred()
Private Sub btnClick_Click(ByVal sender As System.Object, ByVal e As
  System.EventArgs) Handles btnClick.Click
    RaiseEvent ActionOccurred()
End Sub
```

Go back to Form1. Draw a button on the form, name the button **btnShow,** and set its Text property to **Show Form2**; then place the following code in this form:

```
Dim WithEvents ActionForm As Form2
Private Sub btnShow_Click(ByVal sender As System.Object, ByVal e As
  System.EventArgs) Handles btnShow.Click
    ActionForm = New Form2()
    ActionForm.Show()
End Sub
Private Sub ActionForm_ActionOccurred() Handles ActionForm.ActionOccurred
    MsgBox("Someone has clicked the Click button")
End Sub
```

Run the project, click the Show Form2 button, and click the Click button in Form2. Each time you do so, you should see the "Someone has clicked the Click button" message. Can you see the use of this code arrangement? One form (Form1) can be used to monitor the activities in another form (Form2), which can inform the other of what happens by raising an event.

12-5. Beware of the Difference. (continued from exercise 12-4) Add the following code to Form1 in the preceding exercise:

```
Private Sub Form1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
  Handles MyBase.Click
    Dim ActionForm As New Form2()
    ActionForm.Show()
End Sub
```

Run the project and click on Form1—the form itself, not the Show Form2 button. When Form2 appears, click its Click button. Do you see any message displayed? Why?

Click Form1's title bar to set focus on Form1; then click the Show Form2 button to display *another Form2*. Click the Click button on the newly displayed form. Do you see a message? Alternate the focus between the two Form2's and then click the Click button to confirm that one sends messages and the other does not.

How do you explain the difference? The ActionForm object created in the Button Click event procedure is associated with an object variable declared "WithEvents." It is capable of recognizing the event raised from the object. The ActionForm object created in the Form Click event procedure is not declared "WithEvents," and therefore cannot recognize the event.

12-6. What Comes Next? Enter the following code in a new form:

```
Enum TestType
    Test1
    Test2
    Test3
```

```
End Enum
Private Sub Form1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
  Handles MyBase.Click
    Console.WriteLine(TestType.Test1 & TestType.Test2 & TestType.Test3)
End Sub
```

Run the project and then click the form. What numbers do you see in the immediate window? Change Test1 in the Enum block as follows:

```
    Test1 = 1
```

Run the project and then click the form again. This time what do you see?

Change the Enum block so that it will appear as follows:

```
Enum TestType
    Test1
    Test2 = 2
    Test3
End Enum
```

Run the project, and click the form again. What do you see this time? What is your conclusion concerning the items not being assigned with a value? (When nothing is assigned, the first member is zero. If a name is assigned with a value when the next one is not, the next one is one greater than the preceding value.)

12-7. The Handles Clause. Draw a button on a new form. Name the button **btnTest,** and set its text to **Test**. Place the following in code window:

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

Run the project, and click the button to make sure that the "Test" message is displayed.

Add another button to the form. Name the button **btnTest1**, and set its text to Test1. Modify the Handles clause in the preceding procedure as follows:

```
  Handles btnTest.Click, btnTest1.Click
```

Run the project, and click both buttons. Is the "Test" message displayed when each button is clicked? Now remove the Handles clause so that the procedure header appears as follows:

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

Run the program again and then click both buttons. Is the "Test" message displayed when either of the buttons is clicked? Without the Handles clause, the event procedure (handler) is not invoked.

12-8. The AddHandler Statement. (continued from exercise 12-10) Now add the following code in the Form Click event procedure to the preceding project.

```
  AddHandler btnTest1.Click, AddressOf btnTest_Click
```

Run the project, and click both buttons. Is the "Test" message displayed when either of the buttons is clicked? Not yet. Now click the form and then click both buttons again. When is the "Test" message displayed? Notice that the event and the event handler are not hooked to each other until the AddHandler statement is executed. Notice also that in the AddHandler statement, btnTest1.Click is the event hooked to the event procedure. So, when you click the Test1 (not Test) button the message is displayed. The name of the event procedure itself does not determine which control's event is hooked. The AddHandler statement or the Handles clause does.

Now try to make the event procedure handle the click event of both buttons. (Either an AddHandler statement or the Handles clause should do.)

12-9. How the Shared Property Works. Add a class module to a new project. Place the following code in the module:

```
Public Class Class1
    Public Shared Count As Integer
    Public ID As Integer
    Sub New()
        MyBase.new()
        Count += 1
        ID = +1
    End Sub
End Class
```

Place the following code in the form click event procedure:

```
    Console.WriteLine(Class1.Count)
```

Do you see anything underlined? A Shared variable can be referenced with the class name. Now try to add the following statement:

```
    Console.WriteLine(Class1.ID)
```

Is there an error? Public variables are properties and must be referenced with the object name. Remove this line from the code.

Add a button to the form. Name it **btnCreate** and then set its text to **Create Object**. Place the following code in the code window:

```
Private Sub btnCreate_Click(ByVal sender As System.Object, ByVal e As
  System.EventArgs) Handles btnCreate.Click
    Dim MyObject As New Class1()
    Console.WriteLine("Count = " & Class1.Count)
    Console.WriteLine("ID = " & MyObject.ID)
End Sub
```

Run the project and then click the button several times. Inspect the results in the immediate window. You should see the value for Count increases each time you click the button, but the value for ID remains to be 1. Each instance of the class has its own ID property, but all instances share the same Count property because it is declared as Shared.

Also try to qualify Count with MyObject, instead of Class1 and see what the compiler says.

Exercises

12-10. An Employee Class. Create an employee class named Employee that has the following properties:

Field (Property)	Type
SSN	Long
Name	String
HourlyRate	Double

It has a GrossPay method that computes the gross pay of the employee. The method requires the number of hours worked (Double) as the parameter.

Draw a button in the form. Set its Text property to **Compute**. When the user clicks the button, your program will create an employee object named TheEmployee. Set its SSN, Name, and HourlyRate properties to 123456789, John Smith, and 70, respectively; then use its GrossPay method to compute the gross pay for working 50 hours in a week (overtime pay is 1.5 times of the regular rate). Use a message box to display the following message:

```
John Smith (employee number 123-45-6789) worked 50 hours, and is paid $x,xxx
```

(Replace x,xxx with proper amount.)

12-11. A Depositor Class. Create a depositor class, and name it **Depositor**. The class should have the following properties:

Properties	Type
AccountNumber	Integer
Name	String
Balance	Decimal (read-only)

It has two methods: Deposit and Withdraw. Both methods require one parameter (Amount) of the Decimal type. Deposit adds the value of the parameter to Balance, while Withdraw subtracts the value of the parameter from Balance. If the amount to be withdrawn is greater than the Balance, the object should raise an NSF event, which gives the Balance (before the withdrawal) and a Boolean variable, OK (with a default value, False). If OK is set to True in the calling module, the withdrawal is processed; otherwise, it is denied.

Test and use the class in a new project as follows:

1. Draw a text box, two radio buttons, and two buttons on the form. The text box will be used to enter the amount of deposit or withdrawal. The two radio buttons will be used to indicate whether the amount is a deposit or withdrawal. One button will have the text Process Transaction. When it is clicked, the amount in the text box will be processed either by the Deposit method or Withdraw method, depending on the radio button selected. The other button will have the text Quit. When this button is clicked, your program quits.
2. The object's Account Number and Name properties should be set to 1111 and Jane Doe, respectively. The initial balance is zero.
3. When the program runs, the user can enter an amount in the text box, select the proper radio button and then click the Process button. After the amount is processed, the balance is displayed in a message box, and the text box is cleared. When the object raises the NSF event, your program should display a message, ask the user whether to proceed with the transaction, and handle it accordingly.

12-12. A Student Class. Create a Student class that has the following properties:

Property	Type	Description
Number	Long	Student ID number
Name	String	Student name
CourseCount	Long	Returns the number of courses enrolled; read-only

The number and name properties can be set only once in the duration of the object. Attempts to set these properties more than once should cause a "Property already set" exception.

The class has three methods:

- The Enroll method takes Course ID as a parameter, and adds the Course ID to an array that tracks the courses enrolled. (*Hint:* Use `ReDim Preserve` to increase the size of the array to accommodate the new course.) An attempt to enroll a course already in the course list will cause an exception.
- The Drop method removes a course from the Courses list. An attempt to remove a course not already in the list will cause an exception.
- The GetCourses method returns a string array that contains the courses in which the student has enrolled.

Draw a text box and three buttons on the form. The text box with proper label will be used to enter the course ID. One button has the text Enroll. When it is clicked, the course ID in the text box will be added to the student's course list. Another button has the text Drop. When it is clicked, the course ID in the text box will be removed from the course list. Finally, the third button has the text Show. When it is clicked, the courses in which the student has enrolled will be displayed in a message box.
The Number and Name properties will be set to 123456789 and John Doe, respectively, by code as soon as the project starts.

12-13. A Student Class with Database Tables. Modify exercise 12-12 so that when the Number property is set for the Student object, the student name and the courses enrolled are retrieved from the Students database, which contains two tables: Students and StudentCourse Tables.
The Students table has the following fields:

Field	Description
RecordNo	Record number; auto number; primary key
Number	Student ID number (SSN), Integer
Name	Student name, String

The StudentCourse table has the following fields:

Field	Description
Number	Student ID Number (SSN), Integer
Course	Course ID such as Acct5335-001

The form that uses the Student object has the following controls:

- A text box (with a proper label) for the user to enter the student number
- A label to display the student name (with a proper label)
- A text box to enter/display a course ID
- A list box to display the courses enrolled
- A button to enroll a new course
- A button to drop a course
- A button to quit the application

When your program runs, as soon as the user leaves the Student Number field (except for going to the list box or the Quit button), the program sets the Number property of the Student object, which then will proceed to set all other pertinent properties or internal member variable(s) (Courses and CourseCount). (*Hint:* Add a routine in the class module to retrieve the student enrollment data; then use the retrieved data to set mCourseCount and the mCourses array).

A student must already exist before the student can enroll in or drop a course. The student name and courses enrolled should be displayed in the controls described previously. Note that the Enroll and Drop methods should be modified to take care of adding records to and deleting records from the StudentCourse table when called.

12-14. Use of the Constructor. (refer to exercise 11-27 in Chapter 11) Modify the project so that the dialog box will appear and accept the calling form's text box as its parameter when the dialog box is instantiated. This modification should eliminate the need for the FindNext method described in exercise 11-27. (Hint: Use the Sub New constructor. In the Sub New procedure, be sure to include this statement: `InitializeComponents()`, which will draw the controls in the form when the form is being constructed.)

12-15. Revisiting the Fixed Asset Class. Modify the FixedAsset class in the text as follows:

- Add an additional (accounting) depreciation method called the double declining balance method so that the DepreMethod will have three alternative settings (0, no depreciation; 1, straight line depreciation; and 2, double declining balance). This method is explained at the end of this exercise.
- Add the Depreciation (class) method and the CumulativeDepreciation method. Both methods will take Years as their parameter, and return the current depreciation and the cumulative depreciation amount for the specified year(s), respectively.
- Add the DepreMethodChanged event. This event will be raised when the user tries to change the depreciation method for an existing asset; that is, the user is allowed to set the DepreMethod once. Any subsequent attempt to change the depreciation (accounting) method will cause this event to be raised. This event should have three parameters: OriginalMethod, NewMethod, and Cancel. The first two show the pertinent depreciation methods. The last parameter allows the user to cancel the change.

Use the following formula to compute the depreciation of the double declining balance method for a given year:

Depreciation = (2/N) * Current Net Book Value
Where *N* = the life of the asset in years, and
Current Net Book Value equals cost minus cumulative depreciation

If the straight line method on the current net book value results in a higher depreciation amount, the straight line method should be used. That is, If Current net book value/Remaining life > Current net book value * (2/N), Depreciation = Current net book value/Remaining life.

Test the class module by designing a user interface in the form that allows the user to enter/specify the asset's cost, life, years in use, and the deprecation method. Use radio buttons for depreciation methods. Each time a radio button is clicked, the DepreMethod property is set. Clicking these buttons more than once should cause the object to raise the DepreMethodChanged event. Your program should then warn the user of the change and ask if the user means to change. A No answer should cancel the change. When the user clicks the Compute button, your program should show the current depreciation, cumulative deprecation, and book value in three text-boxes, respectively.

12-16. An Apartment Class (Nested). Create an apartment class that has two properties: Bedroom and Kitchen. Each of these properties has the Size property expressed in squared feet, and the FurnitureList property returns a list of furniture and fixtures in the room. Each property also has an AddFurniture method that allows the user to add a furniture item to the furniture list. This method takes a string variable as its parameter, representing the furniture item. Notice that both the Bedroom and Kitchen properties have the same method and properties. You can, therefore, write an inner class and then create the two properties from this class.

12-17. A Simulation Engine. Refer to exercise 7-37 in Chapter 7, "Repetition." Modify the project so that the simulation process is done in an object. The visual interface in the form can then be used to accept input from the user and call the object.

12-18. A Sort Class with an Event. Create a Sort class module with two of your favorite sorting algorithms. The SortAlgorithm property allows the user (or programmer) to specify the sorting algorithm to use. The programmer should be able to specify the sorting algorithm by a named constant, such as sortBubble or sortQuick. One of these algorithms should be set as the default. The class has an

Exec method that will carry out the sorting. This method takes an array as its parameter, which will be sorted. In both sorting routines, implement an event called PercentChanged, which will be fired as the percentile of sort completion changes. The event should provide the following header template:

```
Private Sub Object_PercentChanged(Percent As Integer)
```

Draw two radio buttons, two buttons, a list box, and a label in a new form. The radio buttons will be used to specify the sorting algorithm to use. One button will have the text Generate. When it is clicked, 10,000 random numbers of the Integer type will be generated. Another button will have the text Sort. When it is clicked, it will invoke the sort object's Exec method to perform the sorting. The result will be displayed in the list box. As the sorting is in progress, the label should display the (estimated) percentage of completion.

12-19. Hooking Events to An Event Handler At Run Time. Refer to the Cashflow data entry example in section 8.3 and the subsection on hooking event handlers in section 12.3. Complete the project by adding an event handler that will check for numeric keys and hooking it to the KeyPress event of all text boxes created at run time.

12-20. A Data-Entry Server. Refer to exercise 10-19 in Chaper 10, "Special Topics in Data Entry." The exercise results in a project that includes various Sub procedures to check for valid keys, cumulative results starting from Exercise 10-14. Create a class module (call it EntryServer) that will include all these Subs at its Private procedures. The class module will provide a method, VerifyKey, which takes ActiveControl and KeyAscii as its parameters and returns a Boolean value to indicate whether there is a problem with the key. A True value will indicate that the key is wrong. The wrong key will be suppressed in the method. All controls should have their Tag properties set according to the specifications in those exercises in Chapter 10, "Special Topics in Data Entry."

Include a ClearScreen method in this class module. The method should take a form as its parameter. All the text boxes, masked edit boxes, check boxes, and radio buttons in the form should be cleared. All list boxes and combo boxes in the form should have their SelectedIndex properties set to -1. Also, clear all selections if a list box allows multi-selections because of its SelectionMode setting. Test the class module using a form with a visual interface that includes all text boxes and masked edit boxes needed to test each type of key. In addition, add a check box, two radio buttons, a list box, and a combo box to test the ClearScreen method.