CHAPTER 4

Object-Oriented Programming

From the introduction of Version 4.0 of Visual Basic until the release of Version 6.0, a lively debate raged among developers about whether Visual Basic was or was not an object-oriented programming language. Proponents of the first position could point to Visual Basic's extensive support for objects and interfaces, while advocates of the opposite point of view could point to its lack of inheritance and its overall limited number of object-oriented features. With the release of .NET, however, this debate has become of historical interest only. Visual Basic .NET is clearly an object-oriented language.

Typically, treatments of object-oriented programming begin by discussing the four major characteristics of object-oriented languages: abstraction, encapsulation, inheritance, and polymorphism. This chapter, however, will begin by examining the specific implementation of object-oriented programming features in Visual Basic .NET and the .NET platform.

.NET Types

.NET recognizes six categories of types that can be defined in a namespace:

- *Classes*, which are reference types defined by the Class . . . End Class construct.
- *Arrays*, which are reference types that store objects of another type. The Array class is defined in the System namespace of the .NET Framework Class Library, and array objects can be instantiated in your code; see Chapter 3 for details.
- *Structures*, which are value types defined by the Structure . . . End Structure construct.
- *Interfaces*, which define a contract that implementers must conform to, are defined by the Interface . . . End Interface construct.
- *Delegates*, which are reference types that encapsulate methods with particular signatures. They are defined using the Delegate statement.
- *Enumerations*, which are a collection of related values, defined by the Enum . . . End Enum construct.

TABLE 4-1 The Inheritance Chain of .NET Types

Type	Chain of Inheritance
Arrays	System.Object
Classes	System.Object
Structures	System.Object System.ValueType
Delegates	System.Object System.Delegate or System.MulticastDelegate
Enumerations	System.Object System.ValueType System.Enum
Interfaces	none

.NET includes a Type object that can be used to retrieve information about a particular type. The Type object for a type can be retrieved in Visual Basic in either of two ways:

- By using the GetType operator and providing it with the name of the type in which you're interested as an argument. For example,

  ```
  Dim typ As Type = GetType(Integer)
  ```

  returns a Type object with information about the Visual Basic Integer data type.
- By calling the GetType method of an instance of the type. For example, the code:

  ```
  Dim counter As Integer = 10
  Dim typ As Type = counter.GetType()
  ```

  returns a Type object with information about the type of the *contents* of the counter variable, which happens to be an Integer. If we had declared the variable to be of type Object, as in the following code, the GetType method would still return a Type object representing the Integer type, which is the variable's runtime type at the time the method is called:

  ```
  Dim counter As Object
  counter = 1
  Console.WriteLine(counter.GetType().FullName)
  ```

Ultimately, all .NET types except for interfaces are derived from a single type, System.Object. (That's why Object happens to be .NET's "universal" data type, much as the Variant was in COM-based versions of Visual Basic.) Table 4-1 shows the inheritance chain for each of .NET's six categories of types.

.NET Type Members

Each of the six categories of .NET types can define one or more members. Members define the public interface of a type and either allow you to set or retrieve the data of a .NET type, or provide access to the functionality that a .NET type makes available. Type members include the following:

- *Fields*, which are public constants or variables that allow access to a type's data. Fields can be defined by classes, structures, and enumerations. (In fact, enumerations have only fields.) Since fields allow little opportunity for data validation and offer little protection from inappropriate changes to data values, they are used primarily for read-only data. Often, fields are implemented as constants, which are necessarily read-only, since their value is defined at compile time and cannot be modified in the runtime environment. In other cases, fields are implemented as read-only variables that are, in fact, write-once variables: their values can be defined at runtime by a class constructor but cannot subsequently be modified. For example, field declarations might take the following form:

```
Public Class TestClass
   Public Const Label As String = "Test Class"
   Public ReadOnly CounterStartValue As Integer

   ' Constructor to allow us to assign a value to a read-only variable
   Public Sub New(ctr As Integer)
      CounterStartValue = ctr
   End Sub
End Class
```

- *Properties*, which allow access to the type's data. In Visual Basic .NET, properties can be defined for classes, structures, and interfaces. Most commonly, properties are both readable and writable, although they can also be read-only or write-only (although the latter is rare). Properties are defined with the Property . . . End Property construct. A frequent pattern for assigning and returning property values is shown in the following code, in which the property is responsible for assigning a value to or retrieving a value from a private variable that is otherwise accessible only from within its class:

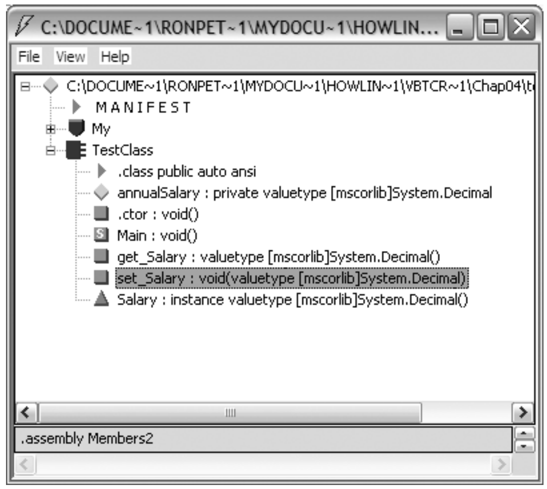
```
Private annualSalary As Decimal

Public Property Salary() As Decimal
   Get
      Return annualSalary
   End Get
   Set
      annualSalary = Value
   End Set
End Property
```

Some properties may require that you provide an index or key that identifies a particular member of an array or a collection. The Item property of the Visual Basic .NET Collection object, which allows you to retrieve a member of the collection by its key or its ordinal position in the collection, provides an excellent example:

```
Dim states As New Collection
states.Add("California", "CA")
states.Add("Michigan", "MI")
states.Add("New York", "NY")
Dim state As String = CStr(states.Item("CA"))
```

FIGURE 4-1
The .NET implementation of properties



Interestingly, properties are actually implemented internally as methods, as Figure 4-1 shows. The Salary property consists of a get accessor method (get_Salary) and a set accessor method (set_Salary), both of which are members of the TestClass class.

- *Methods*, which are the functions (defined by the Function . . . End Function construct) and subroutines (defined by the Sub . . . End Sub construct) that expose the functionality provided by a class, structure, interface, or delegate. Of the major net types, only enumerations cannot have methods. Delegates are a partial exception: although they have methods that they inherit from System.Delegate or System.MulticastDelegate, you cannot define new delegate methods. Functions and subroutines were discussed at length in Chapter 3.
- *Events*, which are function calls from a source to bound delegates that occur in response to some event (such as a mouse click, a press of the keyboard, or a change to the value of a field in a database). Typically, events are passed two parameters: an Object indicating the sender of the event; and an object of type EventArgs or a type derived from it that provides information about the event. Events are discussed in greater detail in Chapter 5.

We'll begin our examination of object-oriented programming with Visual Basic by examining inheritance, since it is the basis of the .NET type system.

Parameterized Properties and CLS Compliance

Parameterized properties are not CLS-compliant. Clients of your components who are using a .NET language that does not support parameterized properties, such as C#, have to use the get_*propertyName*(*index*) or set_*propertyName*(*index*) syntax in order to retrieve or assign a property value. If you want to write CLS-compliant code, you should avoid parameterized properties.

Inheritance

For the most part, the language features that we've covered so far do little to distinguish Visual Basic .NET as a more object-oriented language than its predecessors. There is, however, such a distinguishing feature: *inheritance* is not only the basis of the .NET type system, but the single individual feature that marks Visual Basic .NET as a clearly object-oriented programming language.

Inheritance means that a .NET type is based on, or inherits the members of, the type from which it is derived. If inheritance is not explicitly defined, .NET relies on implicit inheritance; particular types automatically inherit the base types shown earlier in Table 4-1. However, in the case of classes only, inheritance can also be explicit: you can designate a particular class from which a class you're defining derives. Explicit inheritance is indicated by using the Inherits keyword on the line following the class definition. For example:

```
Public Class MyForm
   Inherits System.Windows.Forms.Form
```

.NET supports single inheritance only. That is, a class can inherit directly from only a single other class. In order for a class to inherit from multiple classes, the inheritance must occur in a chain. That is, class D must inherit from class C, which inherits from class B, which inherits from class A.

However, not all classes support explicit inheritance. If a class is *sealed*—which in Visual Basic .NET means that it's marked with the NotInheritable keyword—other classes cannot be derived from it. In an application in which a series of classes are built through inheritance, it is common to mark the final set of inherited classes that the application actually instantiates as sealed, since the application will have no further need to derive classes from them. Often, core classes whose operation is central to an application (or to the system) might also be marked as sealed to prevent the creation of inherited classes with disabled or incorrect functionality.

Conversely, some classes cannot themselves be created (you cannot create an instance of this class using the New keyword) but instead are intended to be used only as a base class from which other classes inherit. Such classes are marked with the MustInherit keyword and are known as *abstract base classes.*

Inheritance automatically makes the attributes and the functionality of a base class available to its derived classes. For example, consider the following code:

```
Public Class EmptyClass
End Class

Public Module modMain
   Public Sub Main()
      Dim ec As New EmptyClass
      Console.WriteLine(ec.ToString())   ' Displays "EmptyClass"
   End Sub
End Module
```

Even though EmptyClass is a class with no members, the Main method is able to call a ToString method, which displays the name of the class. We are able to do this because our seemingly empty class automatically inherits the members of System.Object, the base class for all classes. (For the members of System.Object, see Appendix G.) This functionality of the base class is available for free; we don't have to do anything to get it.

Overriding Members

We don't, however, have to accept all of the data and the behaviors of the base class. In many cases, we can change none, some, or all of them, depending on the needs of the class we're creating. For instance, we might want our class' ToString method to do something other than display its name. In that case, we can override the base class' ToString method by providing a replacement method. This is called *overriding* the base class method, and requires that we use the Overrides keyword when we define our method. For example:

```
Public Class EmptyClass
   Public Overrides Function ToString() As String
      Return "This is an (almost) empty class."
   End Function
End Class

Public Module modMain
   Public Sub Main()
      Dim ec As New EmptyClass
      Console.WriteLine(ec.ToString() )    ' Displays "This is an (almost)
                                           '             empty class."
   End Sub
End Module
```

Very much as you can't inherit from all classes, though, you can't override all base class members. Properties and methods of a base class can be overridden by a derived class only under either of the following two conditions:

- The base class member is marked with the Overridable keyword, which allows you to override it.
- The base class member itself overrides a corresponding member of its base class and is not marked with the NotOverridable keyword.

So, members of base classes are not overridable by default, even if they are not explicitly marked with the NotOverridable keyword. And if a member of a base class can be overridden, that ability to override the member is inherited by derived classes until a particular subclass marks the member as NotOverridable. In other words, unless you have a compelling reason to prevent a class member from being overridden, you should remember to mark it as Overridable.

While there are times when you can't override a member of a class, there are other times when you're required to override a class member. Such members are marked with the MustOverride keyword. Typically, you must override a class member under any of the following conditions:

- The base class provides either no real implementation or a very partial implementation of the method. It relies on derived classes to provide complete implementations.
- The base class itself is an abstract base class—that is, it defines the members that a derived class should have, but it provides no real implementation of them, leaving it to the derived classes to do this.

Overloaded Members

A basic rule of Visual Basic 6.0 and earlier versions was that each member name in a class must be unique, since the Visual Basic compiler relied exclusively on the member name to identify the member. In .NET, that is no longer true. Instead, each member must be uniquely identified by its signature, which includes a combination of its name and the types in its parameter list. (In distinguishing members with the same name, a method's return value or a property type is not part of that member's signature; that is, members with the same name cannot be differentiated by return type alone.)

This makes it possible to *overload* members, which means that multiple members can share the same name but can be distinguished by differing parameter types. In the .NET Framework Class Library, for instance, the Convert.ToString method provides an excellent example of operator overloading; there are 36 different versions of the Convert.ToString method, each of which is distinguished from the other 35 versions by the number and type of its parameters.

Overloading is significant because it allows members to be named in terms of their functionality, rather than the need to find a unique name. And as we saw in Chapter 3, it allows parameter lists to reflect the specific types expected by a method, rather than a weakly typed parameter that can accept an argument of virtually any type.

At the same time, overloading methods in a careless way can lead to a duplication of code, as each of the overloaded methods contains code that performs a more or less identical set of operations. Typically, this problem is solved by performing only the minimum of work that is necessary (typically data conversion and defining default values) before calling the "main" version of the method, where most of the actual work is performed. For example: the Person class in the following code returns an array of Person objects whose name meets the search criteria. The method has three overloads: one that accepts a last name, one that accepts a first and last name, and one that accepts a first, middle, and last name. The first two methods provide a default value for the parameters they don't define and call the third overload.

```
Public Shared Function FindName(lastName As String) As Person()
   Dim middleName As String = String.Empty
   dim firstName As String = String.Empty
   Return FindName(firstName, middleName, lastName)
End Function

Public Shared Function FindName(firstName As String, lastName As String) _
                                As Person()
   Dim middleName As String = String.Empty
   Return FindName(firstName, middleName, lastName)
End Function

Public Shared Function FindName(firstname As String, middleName As String, _
                                lastName As String) As Person()
   ' Query database to return names
   Return matches
End Function
```

Constructors

.NET classes and structures have one or more constructors that can be executed when the class or structure is instantiated. This support for constructors differs in a number of ways from class modules in Visual Basic 6.0 and earlier versions, where a Class_Initialize event procedure appeared to function as a class constructor:

- Constructors are called when the object is instantiated and are executed before any other code belonging to the class. In contrast, the Class_Initialize event was actually fired after the object was created but before it was activated; any code located outside of individual class members that declared and initialized variables was executed first.
- In .NET, constructors can be overloaded; there can be multiple constructors that differ by their parameter list. In contrast, Class_Initialize did not support method arguments.
- .NET constructors can be executed automatically only when an object instance is instantiated, or they can be called from the first line of code in a derived class constructor (a topic that we'll discuss in detail later in this section). In contrast, the Class_Initialize procedure could be called from anywhere in code.
- Structures as well as classes in .NET can have constructors. In contrast, only classes supported the Class_Initialize procedure. (Structures in Visual Basic 6.0 could not have properties or methods.)

In Visual Basic .NET, constructors are subroutines named New, and they can be defined in both classes and structures. Like any method, constructors can have parameter lists and can be overloaded. The overloading of constructors, in fact, is an area of confusion in Visual Basic .NET. If you fail to define any constructors, the Visual Basic .NET compiler automatically implements a parameterless constructor in a class. For example, Figure 4-2 shows the result if we compile the following code and display the resulting assembly in ILDasm:

```
Public Class Animal
   Dim animalName As String

   Public Property Name() As String
      Get
         Return animalName
      End Get
      Set
         animalName = Value
      End Set
   End Property
End Class
```

Note that in addition to the private animalName string variable and the Name property (Name) along with its set (set_Name) and get (get_Name) accessors, ILDasm displays a class constructor (indicated as .ctor) that is a subroutine (it returns void, which means that the method has no return value) with no parameters.

However, if we explicitly define parameterized constructors for a class but fail to define a parameterless constructor, the Visual Basic .NET compiler does not add a parameterless constructor to our class. If we are defining class constructors, we have to remember to include a parameterless constructor if we want one.

In Visual Basic .NET, structures can also have constructors. However, all structure constructors must be parameterized. The Visual Basic .NET compiler does not automatically include a parameterless constructor among a structure's members if you fail to define a parameterized constructor. And the attempt to explicitly define a parameterless constructor generates a compiler error ("Structures cannot declare a non-shared 'Sub New' with no parameters.").

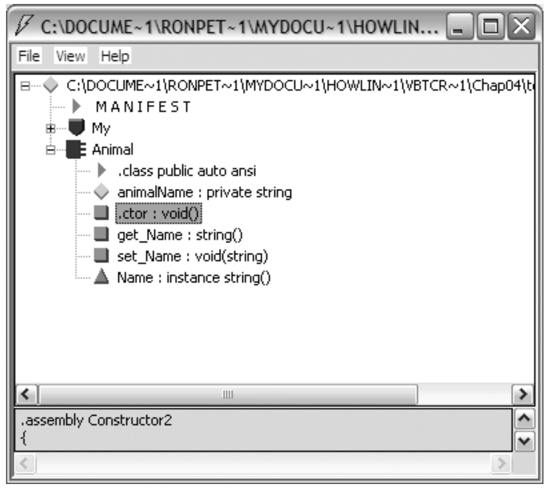


FIGURE 4-2 A compiled class displayed by ILDasm

Constructors of derived classes aren't marked with the Overrides keyword. But the constructors of derived classes must call the base class constructor in the first line of the constructor's code, immediately after the constructor's subroutine definition, unless the base class implements a parameterless constructor. In the latter case, the call to the base class constructor can be omitted. To call the base class constructor, the MyBase keyword is used, as the following code illustrates:

```
Public Class BaseClass
   Private _name As String

   Public Sub New(name As String)
      _name = name
   End Sub

   Public ReadOnly Property Name() As String
      Get
         Return _name
      End Get
   End Property
End Class
Public Class DerivedClass
   Inherits BaseClass

   Public Sub New(name As String)
      MyBase.New(name)
   End Sub
End Class
```

Whether or not a call to one of the base class constructors is required it's typically a good idea to include the call, since, unless the documentation explicitly indicates otherwise, the base class constructor may perform some initialization that is important for the proper functioning of the class.

Destructors

Many object-oriented programming languages and runtime environments support both constructors, which execute when a class is instantiated, and destructors, which execute when a class instance is destroyed. .NET, in fact, defines not just one, but two different destructors that execute when an object either is about to be destroyed or is being destroyed.

The need for two destructors stems from the fact that in .NET, garbage collection—the process whereby .NET destroys unused managed objects and releases their resources—is non-deterministic. This means that, although garbage collection will happen sooner or later, there is no guarantee of precisely when a particular object that has gone out of scope or that is no longer needed will actually be destroyed.

When a class or structure instance is destroyed (something that the .NET runtime environment manages without your having to write any code), the object instance often has to perform some cleanup, such as closing files, saving state information, or releasing resources that are not managed by .NET. .NET's non-deterministic destructor for this purpose is named Finalize. Unlike the class constructor, which the compiler creates automatically if it is not explicitly declared, Finalize is not created automatically by the compiler. Since it involves a performance penalty, you should implement it only if there are resources belonging to an object instance that must be released as the object is destroyed. In addition, Finalize must be declared as a Protected method, with the following signature:

```
Protected Sub Finalize()
```

Because it is protected, the Finalize method can be called only from within the class that defines it or from a derived class. Typically, Finalize can also be overridden by a derived class by using the Overrides keyword. However, it cannot be called from client code that instantiates the class.

Because of the non-deterministic garbage collection system and because maintaining unneeded resources can often be expensive, .NET provides a second destructor that can be called at any time both from within a class, from a derived class, and from client code, and that immediately releases resources. This destructor is implemented as an interface (a topic discussed in greater detail in the section "Interfaces" later in this chapter) named IDisposable that has a single method, Dispose. The Dispose method has the following signature:

```
Sub Dispose()
```

However, unlike Finalize, Dispose is not called automatically by .NET. Instead, it should be called by clients of the implementing class or structure.

There are two recommended patterns for implementing Dispose, one for base classes and one for derived classes. The recommended pattern for base classes is:

```
Public Class Base : Implements IDisposable
   Public Overloads Sub Dispose() Implements IDisposable.Dispose
      Dispose(True)
      GC.SuppressFinalize(Me)
   End Sub

   Protected Overridable Overloads Sub Dispose(ByVal disposing As Boolean)
      If disposing Then
         ' Free other state (managed objects).
      End If
```

```
      ' Free your own state (unmanaged objects).
      ' Set large fields to null.
   End Sub
   Protected Overrides Sub Finalize()
      ' Simply call Dispose(False).
      Dispose(False)
   End Sub
End Class
```

Base classes should define two versions of Dispose, one of which implements IDisposable.Dispose and, because it is part of the class or structure's public interface (i.e., it is defined as a Public method), is callable from outside of the class or structure, including by clients of the class. This version of the Dispose method first calls the second version of the Dispose method and passes it a True value as an argument. The argument causes the block of code in the second Dispose method that frees other managed resources to execute. The public version of the Dispose method then calls the GC (garbage collector) object's SuppressFinalize method, which indicates that finalization should not be handled automatically by .NET.

The second version of Dispose is protected, and so can be called only from within the class or by a derived class; it is not accessible to clients that have references to instances of the class. This version of Dispose has a single Boolean parameter. A True indicates a client call, which causes the class to release both managed and unmanaged resources. A False value indicates that the call comes from the class or structure, or from a derived class, and that only unmanaged resources should be released.

Instead of directly calling Dispose from within the class or a derived class, though, Finalize should be called. Finalize in turn calls the protected version of Dispose, passing it a False value that indicates that the call comes from within the class, so that any other managed resources should not be released (since they already have been by the client's call to the public version of Dispose).

The following is the recommended implementation of Dispose and Finalize for derived classes:

```
  Protected Overridable Overloads Sub Dispose(ByVal disposing As Boolean)
        If disposing Then
            ' Free other state (managed objects).
        End If
        ' Free your own state (unmanaged objects).
        ' Set large fields to null.
    End Sub

    Protected Overrides Sub Finalize()
        ' Simply call Dispose(False).
        Dispose(False)
    End Sub
End Class
```

Note that this version implements only the protected version of Dispose, which leaves the base class as the only class with a public implementation of Dispose that implements the public IDisposable.Dispose interface.

Identically Named Variables with Different Scope

It's best to avoid giving variables with different scope identical names, since the practice is unnecessarily confusing. If you give your locals unique names, you won't have to disambiguate them later.

Internal References: Me and MyClass

In Visual Basic .NET, as in Visual Basic 6.0, the Me keyword refers to the instance of the class upon which the method or property that's currently executing was called. (The equivalent in C++ or C# is the **this** object.) Typically, Me is not required, but it can add clarity by calling out that a particular identifier refers to a class member. However, in some situations, using Me is required. This is the case, for example, if there is a local variable or method parameter of the same name as an instance member, as in the following code:

```
Public Class Airplane
   Private seat As New ArrayList

   Public Function AssignSeat(seat As String) As Boolean
      For Each seatAssignment As String In Me.seat
         If seatAssignment = seat Then
            Return False
         End If
      Next
      Me.seat.Add(seat)
      Return True
   End Function
End Class
```

Here, seat represents both an ArrayList variable that holds the locations of assigned seats, as well as the name of the AssignSeat method's single parameter. If we fail to qualify the references to the ArrayList variable named seat with the Me keyword, the .NET compiler will assume in the first case that we want to use the String argument and in the second will generate a compiler error, since the String class does not have an Add member.

Similarly, when a variable is hidden by another identically named variable with more immediate scope (or, to put it another way, when a variable with more restrictive scope *shadows* a variable with broader scope), Me allows you to reference the otherwise hidden variable. For example, consider the following code (which admittedly reflects rather poor programming practice):

```
Public Class Counters
   Dim ctr As Integer

   Public Sub New()
      For ctr = 1 to 20
         DoSomething()
      Next
   End Sub
```

```
   Public Sub DoSomething()
      For ctr As Integer = 0 to 2
         Console.WriteLine("The value of ctr is {0}, but the value of Me.ctr
is {1}.", _
                           ctr, Me.ctr)
      Next
   End Sub
End Class

Public Module modMain
   Public Sub Main()
      Dim obj As New Counters()
   End Sub
End Module
```

Here, a class named Counters has a variable named ctr that is visible throughout the class. A second variable named ctr, however, is local to the For Each . . . Next construct. Nevertheless, we are able to reference the first variable within the For Each . . . Next construct by using the Me keyword.

Finally, Me can be used as an argument in a method call when you need to pass a reference to the current class instance to some method or property outside of the class. This, in fact, is one of its major uses.

Because Me refers to the current instance of a class or structure, it can't be used to access class members within a shared property or method. (Shared members, which do not require that an instance of the class be created in order to execute, are discussed in the section "Shared Members" later in this chapter.)

Closely related to Me is the MyClass keyword. For the most part, MyClass is identical to Me. Its sole difference arises in cases in which the MyClass keyword is used in a base class to call one of its members, and a derived class overrides that member; in that case, the MyClass keyword causes the overridable method to be treated as if it is not overridable, and invokes the base class member. This is reflected in the following code:

```
Public Class BaseClass
   Public Sub MainMethod()
      Console.WriteLine("Calling Me.Method1...")
      Me.Method1()
      Console.WriteLine("Calling MyClass.Method1...")
      MyClass.Method1()
   End Sub
   Public Overridable Sub Method1()
      Console.WriteLine("BaseClass.Method1...")
   End Sub
End Class

Public Class DerivedClass : Inherits BaseClass
   Public Overrides Sub Method1()
      Console.WriteLine("DerivedClass.Method1...")
   End Sub
End Class
```

```
Public Module modMain
   Public Sub Main()
     Console.WriteLine("Invoking BaseClass.MainMethod")
     Dim bc As New BaseClass
     bc.MainMethod()
     Console.WriteLine()
     Console.WriteLine("Invoking DerivedClass.MainMethod")
     Dim dc As New DerivedClass
     dc.MainMethod()
   End Sub
End Module
```

When Method1 is called from an instance of the BaseClass class, there is, of course, only one method that can be called—Method1 in BaseClass. However, when an instance of DerivedClass calls Method1, the Me keyword causes DerivedClass.Method1 to be called, whereas the MyClass keyword causes BaseClass.Method1 to be called, just as if BaseClass.Method1 were marked NotOverridable, and DerivedClass had never been able to override the method.

Referencing the Base Class: MyBase

As we've already noted in the discussion of constructors, the MyBase keyword refers to the base class from which the current class is derived. As we've noted, all classes are derived from another class either implicitly or explicitly, while all structures implicitly derive from System.ValueType. MyBase causes a member of those base classes to be executed.

MyBase.New can be used to call the base class constructor, which is generally a good idea when the base class performs some initialization when it is instantiated. MyBase also can be used to call methods in the base class from a derived class when they are otherwise overridden or inaccessible.

Polymorphism

A general definition of polymorphism is that it describes something that has many different forms. In object-oriented programming, it refers to the ability of callers to call objects that behave differently depending on the type of the callee. It allows for the creation of black box routines that can operate on a range of types but always call the appropriate method of that type. Explanations of polymorphism often tend to be long, laborious, and thoroughly confusing. An example better illustrates the concept. Consider, for instance, the following code, which defines an abstract base class named Mammal, a derived class named Canine, and two derived classes that inherit from Canine named Dog and Wolf:

```
Public MustInherit Class Mammal
   Protected nocturnal As Boolean
   Protected herbivore As Boolean
   Protected carnivore As Boolean
   Protected omnivore As Boolean

   Public Property SleepsAtNight() As Boolean
      Get
         Return nocturnal
      End Get
      Set
         nocturnal = Value
      End Set
   End Property
```

```
   Public Property IsHerbivore() As Boolean
      Get
         Return herbivore
      End Get
      Set
         herbivore = Value
      End Set
   End Property

   Public Property IsCarnivore() As Boolean
       Get
          Return carnivore
       End Get
       Set
         carnivore = Value
      End Set
   End Property

   Public Property IsOmnivore() As Boolean
      Get
         Return omnivore
      End Get
      Set
         omnivore = Value
      End Set
   End Property

   Public Overridable Function Sound() As String
      Return "This mammal is largely mute."
   End Function
End Class

Public Class Canine : Inherits Mammal
   Public Sub New()
      MyBase.New()
      Me.Carnivore = True
   End Sub

   Public Overrides Function Sound() As String
      Return "Snarl"
   End Function
End Class

Public Class Wolf : Inherits Canine
   Public Overrides Function Sound() As String
      Return "Howl"
   End Function
End Class

Public Class Dog : Inherits Canine
   Public Overrides Function Sound() As String
      Return "Bark"
   End Function
End Class
```

```
Public Module modMammals
   Public Sub Main
      Dim wlf As New Wolf
      Dim dg As New Dog
      DescribeSound(wlf)
      Eats(wlf)

      DescribeSound(dg)
      Eats(dg)
   End Sub

   Private Sub DescribeSound(mamml As Mammal)
      Console.WriteLine("{0} makes a {1}.", mamml.ToString(), mamml.Sound())
   End Sub

   Public Sub Eats(mamml As Mammal)
      If mamml.IsCarnivore Then
         Console.WriteLine("{0} eats meat.", mamml.ToString())
      ElseIf mamml.IsHerbivore Then
         Console.WriteLine("{0} eats plants.", mamml.ToString())
      Else
         Console.WriteLine("{0} eats meat and plants.", mamml.ToString())
      End If
   End Sub
End Module
```

The base class, Mammal, consists of three protected variables, which means that they're accessible within the Mammal class and any classes derived from it. Had we declared the variables to be private, we would not have been able to access them (as we did in the Canine class constructor) from derived classes. Mammal also defines three properties that wrap the three protected variables. Finally, it has a single method, Sound, that simply returns a string indicating that the animal lacks a distinctive sound. Note that the method has been marked Overridable so that classes derived from Mammal can override the function to indicate the animal's sound.

Canine inherits from Mammal and adds a parameterless constructor that sets the value of the Carnivore property to True. (The other properties remain at their default value, which is False for Boolean properties.) It also overrides the Sound method to return the string "Snarl".

The Wolf class inherits from Canine. (It could have inherited from Mammal, but deriving it from Canine eliminates the need to explicitly define the Carnivore property as True, since canines are primarily carnivores, and wolves are canines.) The only code within the Wolf class overrides the Sound method to return the string "Howl".

The Dog class also inherits from Canine and, like the Wolf class, overrides the Sound method, in its case to return the string "Bark".

Finally, the code example includes a module that defines three methods: Main, DescribeSound, and Eats. Main instantiates one Dog object and one Wolf object. Notice that it declares the objects to be of type Mammal and then uses the New keyword to invoke the constructors of the Dog class and the Wolf class and assign a Dog class instance and a Wolf class instance, respectively, to our Mammal variables. We can do this because of polymorphism.

Once the two object variables are instantiated, the code calls the two other methods, once with each object type. DescribeSound accepts an argument of type Mammal (not an argument of type Dog or Wolf) and indicates whether the mammal is a herbivore, a carnivore, or an omnivore.

Sound also accepts an argument of type Mammal and displays a string that describes the sound made by the mammal. Note that in the case of both methods, the single parameter expects an argument of type Mammal, and not an argument of type Dog or Wolf or any other specific mammal.

When the code executes, it produces the following output:

```
Wolf makes a Howl.
Wolf eats meat.
Dog makes a Bark.
Dog eats meat.
```

In this example, the DescribeSound and the Eats methods are polymorphic. They don't care what specific type is passed to them as an argument, as long as that type either is Mammal or is derived from Mammal. Because of inheritance, we can write black box routines that (within limits) don't care about the types passed to them but are nevertheless able to call the correct method and produce the correct result anyway.

Casting Using DirectCast and TryCast

Ordinarily, if you want to convert a variable from one type to another, you use the CType function or one of the other conversion functions implemented by the compiler (like CStr, CInt, CDbl, etc.). However, Visual Basic has two other casting or conversion operators, DirectCast and TryCast, which are designed to handle conversions between types that are related to one another through inheritance or implementation (a topic discussed later, in the section "Interfaces"). Although DirectCast was introduced in .NET 1.0, TryCast is new to Visual Basic 2005. Both have the same syntax as CType:

```
ConvertedType = DirectCast(variable_name, type_name)
Convertedtype = TryCast(variable_name, type_name)
```

DirectCast will fail under either of two conditions:

- The object reference to be converted is not related to *type_name* through a relationship based on inheritance or interface implementation. In this case, Visual Basic generates a compiler error.
- The cast is a narrowing conversion that fails. In this case, the .NET runtime generates an InvalidCastException. .NET considers a narrowing conversion to be any conversion from a base class instance to a derived class, while it considers a widening conversion to be any conversion from a derived class to a base class or interface. For example, the conversion from an instance of the Wolf or Dog class to the Mammal class is a widening conversion. On the other hand, the conversion of an instance of the Mammal class to either the Wolf or Dog class is a narrowing conversion. For instance, the following code instantiates a Dog instance and then calls the DirectCast conversion function twice:

  ```
  Dim dg As New Dog
  ' Cast Dog instance to Mammal
  Dim objMammal As Mammal = DirectCast(dg, Mammal)
  ' Cast Mammal instance to Wolf
  Dim objWolf As Wolf = DirectCast(objMammal, Wolf)
  ```

The conversion from Dog to Mammal always succeeds because it is a widening conversion: every Dog is a Mammal. The conversion from Mammal to Wolf fails (it throws an InvalidCastException) because the original variable before its conversion was of type Dog, and a Dog is not a Wolf. The attempt to convert a Dog to a Wolf in this case violates the IS A relationship.

When using DirectCast, you have two options for dealing with potential errors. The first is to use rigorous type checking before the conversion to prevent the exception. For instance, in our preceding code fragment, we might have checked the Mammal variable as follows before performing the conversion:

```
If TypeOf objD Is Wolf Then Dim objMD As Wolf = DirectCast(objD, Wolf)
```

The second is to use exception handling, as illustrated in the following code:

```
Dim objD As Mammal = DirectCast(dg, Mammal)
Try
   Dim objMD As Wolf = DirectCast(objD, Wolf)
Catch e As InvalidCastException
   Console.WriteLine("Can't cast from " & TypeName(objD) & " to " _
                     & TypeName(objMW) & " here.")
End Try
```

Handling exceptions, however, has a serious impact on a program's performance. To eliminate the need for handling an exception (and even to save yourself from having to examine object types before converting them), you can use the TryCast function instead of DirectCast. Like DirectCast, TryCast performs conversions between two types that are related through either inheritance or interface implementation. Unlike DirectCast, however, TryCast doesn't raise an exception if the conversion fails; it simply returns Nothing. As a result, if you use TryCast, you should always check its return value after attempting the conversion.

Shadowing

In the discussion of the Me keyword, we noted that a variable with more restrictive scope shadows an identically named variable with less restrictive scope, so that the former hides the latter while the former is in scope. Shadowing also applies to any program element declared in a type that is derived from a base type. It means that, when members of the derived type are accessed through a variable of the derived type, the derived type members shadow the base type members.

To use a simple example, the following code defines a class with a ToString member:

```
Public Class ShadowClass
   Private value As String

   Public Sub New(value As String)
      Me.value = value
   End Sub

   Public Shadows Function ToString() As String
      Return "Value of " & Me.GetType.Name & ": " & Me.value
   End Function
End Class
```

This version of ToString is declared with the Shadows keyword, which indicates that it shadows the ToString method found in System.Object, the class from which ShadowClass implicitly inherits. If we declare and then instantiate a variable of type ShadowClass, as in the following code:

```
Dim sc As ShadowClass = New ShadowClass("The Shadow")
Console.WriteLine(sc.ToString())
```

the program's output displays the class name along with the value of its private value variable:

```
Value of ShadowClass: The Shadow
```

However, if we declare the variable to be of type Object and instantiate it using the ShadowClass constructor, as the following code shows:

```
Dim obj As Object = New ShadowClass("The Shadow")
Console.WriteLine(obj.ToString())
```

the result is very different:

```
ShadowClass
```

Because in the first example we've called the ToString method through an instance of ShadowClass, the ShadowClass implementation of ToString shadows the implementation of ToString in the System.Object class. But when we call ToString in a variable declared to be of the base class type, the shadowed base class implementation becomes visible and is executed.

Shadowing is somewhat different that overriding. Any base class member can be shadowed, including those that are not overridable. Second, shadowing is based exclusively on name, and not on signature. Thus, a derived class property can shadow a base class method, for instance, or a derived class constant can shadow a base class property.

Interfaces

Visual Basic 6.0 and its earlier 32-bit versions implemented polymorphism through the use of interfaces, and they continue to be very important in Visual Basic .NET. An interface defines a contract that implementers of the contract must fulfill. Interfaces define a set of public members but have no implementation code; the implementation code is provided by the implementers of the interface. Interfaces are defined using the Interface . . . End Interface construct, as follows:

```
<access_modifier> Interface <interface_name>
End Interface
```

Member declarations within the interface cannot have access modifiers because all members are necessarily public. (The purpose of an interface is precisely to define a set of public members that types must define to properly implement that interface.) In addition, they contain no code, and even omit the terminating End statement. The interface can define methods, properties, and events as members, but it cannot define constructors or destructors.

For example, the following code defines an interface named IBreed (the names of interfaces traditionally begin with the letter "I"):

```
Public Interface IBreed
   Property BreedName As String
   Property Group As String
   Function IsBreedInAmericanKennelClub() As Boolean
End Interface
```

Interfaces are used for a variety of purposes:

- To define a particular service or functionality that is common to a range of classes. In .NET, this is probably the most common use of interfaces. For instance, two interfaces, IEnumerable and IEnumerator, allow you to use the For Each . . . Next loop to iterate various container objects like arrays and collections. The IComparer interface allows two objects to be compared and determines whether they are equal, or whether one is "less than" or "greater than" the other. If you were defining an Automobile class, you might implement the IComparer interface to consider the car that provided the better mileage per gallon of fuel as the "greater" one.
- To apply a particular service or functionality that is common to only a subset of the members of a class. For instance, although all dogs have a breed, IBreed is intended primarily to provide information about purebred dogs.
- To support multiple inheritance in environments (like .NET) that support only single inheritance, or in environments (like Visual Basic 6.0 and earlier versions) that don't support inheritance. This is not a good use of interfaces.
- As a substitute for classes. Again, this is not a good use of interfaces.

As this list suggests, when used properly, interfaces express a "can do" or a "has a" relationship with its implementers: types that implement an interface CAN DO something (such as compare objects using the IComparer interface) or HAVE something (such as a dog's breed, as recognized by the American Kennel Club in the United States and Canada). This differs from the IS A relationship that results when a derived class inherits from a base class. (For example, a Dog is a Mammal.)

Client classes that implement interfaces use the Implements statement on the line following the class definition, or on the line following the Inherits statement. Classes can implement multiple interfaces (hence the tendency to use "interface inheritance" as a work-around for single inheritance); in this case, interface names are separated from one another by commas. In addition, each member that implements an interface member includes the Implements keyword and the name of the interface and member it implements, separated by a dot or period, on the same line as the method definition. The signature of implemented members must conform to the signature defined in the interface.

For example, the following code provides a redefinition of the Dog class we presented earlier to implement the IBreed interface:

```
Public Class Dog : Inherits Canine
   Implements IBreed

   Private IsRecognizedAsBreed As Boolean
   Private Breed As String
   Private BreedGroup As String
```

```
   Public Sub New(breedName As String, breedGroup As String, _
                  isRecognizedAsBreed As Boolean)
      MyBase.New()
      Me.Carnivore = True
      Me.Breed = breedName
      Me.BreedGroup = breedGroup
      Me.IsRecognizedAsBreed = isRecognizedAsBreed
   End Sub

   Public Overrides Function Sound() As String
      Return "Bark"
   End Function

   Public Property BreedName As String  Implements IBreed.BreedName
      Get
         Return Me.Breed
      End  Get
      Set
         Me.Breed = Value
      End Set
   End Property

   Public Property Group As String  Implements IBreed.Group
      Get
         Return Me.BreedGroup
      End Get
      Set
         Me.BreedGroup = Value
      End Set
   End Property

   Public Function IsAKC() As Boolean  _
                   Implements IBreed.IsBreedInAmericanKennelClub
      Return Me.IsRecognizedAsBreed
   End Function
End Class
```

Once we've implemented the interface members in our class, we can instantiate the class and access the implemented members just as we would any other member of the class, as the following code illustrates:

```
Public Sub Main
   Dim malamute As New Dog("Malamute", "Working", True)
   Console.WriteLine(malamute.Group)
   Console.WriteLine(malamute.IsAKC)
End Sub
```

We can, however, require that the interface members be accessed only through an instance of the interface itself. We do this by implementing the interface members as private members of the implementing class, as the following code illustrates:

```
Public Class Dog : Inherits Canine
   Implements IBreed
```

```
   Private IsRecognizedAsBreed As Boolean
   Private Breed As String
   Private BreedGroup As String

   Public Sub New(breedName As String, breedGroup As String, _
                  isRecognizedAsBreed As Boolean)
      MyBase.New()
      Me.Carnivore = True
      Me.Breed = breedName
      Me.BreedGroup = breedGroup
      Me.IsRecognizedAsBreed = isRecognizedAsBreed
   End Sub

   Public Overrides Function Sound() As String
      Return "Bark"
   End Function

   Private Property BreedName As String  Implements IBreed.BreedName
      Get
         Return Me.Breed
      End  Get
      Set
         Me.Breed = Value
      End Set
   End Property

   Private Property Group As String  Implements IBreed.Group
      Get
         Return Me.BreedGroup
      End Get
      Set
         Me.BreedGroup = Value
      End Set
   End Property

   Private Function IsAKC() As Boolean  _
                    Implements IBreed.IsBreedInAmericanKennelClub
      Return Me.IsRecognizedAsBreed
   End Function
End Class
```

Because the BreedName and Group properties and the IsAKC method are now private, we can no longer access them through an instance of the Dog class. Instead, we must instantiate an object of the interface type, assign the instance of the Dog class to it, and then access the implemented members. The following code illustrates this:

```
  Public Sub Main
      Dim malamute As New Dog("Malamute", "Working", True)
      Dim breed As IBreed = malamute

      Console.WriteLine(breed.Group)
      Console.WriteLine(breed.IsBreedInAmericanKennelClub)
   End Sub
```

Note that, because we are accessing the IBreed interface, rather than accessing the IBreed interface members through the Dog class, we cannot access the method that implements IsBreedInAmericanKennelClub by the name we've assigned to it, IsAKC. Instead, we must access it by the name assigned to it in the interface definition, IsBreedInAmericanKennelClub.

Shared Members

In discussing the object-oriented features of classes, we've focused on defining and instantiating classes, and using the object reference to access the class members. In .NET, these are termed *instance members*; in order to access them in code, they require that an instance of the class or structure be created, and each instance of the class has its own set of members with their own values. However, some class members can be accessed or invoked without instantiating an instance of the class. These are *shared members* (in C++ and C#, they're known as static members), and they maintain a single set of values for an application as a whole. In Visual Basic .NET, they're defined using the Shared keyword. Fields, properties, and methods can all be declared as shared.

The System.Math class provides an excellent example of shared members. You can calculate the circumference of a circle, for example, with code like the following:

```
Dim radius As Single = 2
Dim circum As Single = Math.Pi * radius ^ 2
```

Here, we've accessed the shared Pi field of the Math class to compute the circumference. We could also find the absolute value of a number by using the shared Abs method of the Math class:

```
Dim num As Integer = -12
Dim absNum As Integer = Math.Abs(num)
```

In comparison to C#, Visual Basic .NET is somewhat unusual in that you can access shared members either through an instance variable or by specifying the name of the type. For example:

```
Dim mth As System.Math
Dim circum As Single = mth.Pi * radius ^ 2
Dim absNum As Integer = mth.Abs(num)
```

The major restriction when defining shared members is that they can't access instance member of their class. Doing so produces the compiler error, "Cannot refer to an instance member of a class from within a shared method or shared member initializer without an explicit instance of the class." If you want to call an instance method or retrieve the value of an instance property of the class that has the shared member, you have to instantiate an instance of that class. This is a common problem in console applications or executables that have an explicitly defined Sub Main. For example:

```
Public Class SharedClass
   Public Shared Sub Main()
      ' DoSomething()    '-- produces compiler error

      Dim sc As New SharedClass
      sc.DoSomething()
      End Sub
```

```
   Public Sub DoSomething
      ' Code in an instance method
   End Sub
End Class
```

In addition to using the Shared keyword to define members whose values are shared on an application-wide basis, it is also possible to define members whose values are shared on a per-thread basis. This is done using the System.ThreadStaticAttribute class. (An attribute is a class that serves as a label in code to modify the behavior of the compiler at compile time, Visual Studio or some other environment at design time, or the .NET runtime at runtime.) For example, in the following code, the application's main thread and a secondary thread make repeated calls to the IncrementCounter subroutine, which increments a shared per-thread counter:

```
Imports System.Threading

Public Class SharedByThread

   <ThreadStatic> Private Shared Count As Integer

   Public Shared Sub Main()
      Dim secondThread As New thread(AddressOf Thread2Proc)
      secondThread.Start()
      count = 100
For ctr As Integer = 0 to 10
   Console.WriteLine("The value of count in the main thread is {0}.", _
                     count)
   IncrementCounter(200)
Next
   End Sub

   Public Shared Sub Thread2Proc()
      count = 0
For ctr As Integer = 0 to 10
   Console.WriteLine("The value of count in the second thread is " & _
                     "{0}.", count)
   IncrementCounter(250)
Next
   End Sub

   Public Shared Sub IncrementCounter(delay As Integer)
      count +=1
      Thread.Sleep(delay)
   End Sub
End Class
```

When run, the code produces the following output, which shows that the counter has been incremented separately for each thread:

```
The value of count in the main thread is 100.
The value of count in the second thread is 0.
The value of count in the main thread is 101.
The value of count in the second thread is 1.
```

```
The value of count in the main thread is 102.
The value of count in the second thread is 2.
The value of count in the main thread is 103.
The value of count in the second thread is 3.
The value of count in the main thread is 104.
The value of count in the main thread is 105.
The value of count in the second thread is 4.
The value of count in the main thread is 106.
The value of count in the second thread is 5.
The value of count in the main thread is 107.
The value of count in the second thread is 6.
The value of count in the main thread is 108.
The value of count in the second thread is 7.
The value of count in the main thread is 109.
The value of count in the second thread is 8.
The value of count in the main thread is 110.
The value of count in the second thread is 9.
The value of count in the second thread is 10.
```