Bonus chapter for Object-Oriented Programming in VisualBasic.NET by Michael McMillan

Copyright©2004 by Michael McMillan

Inheritance

The real power of OOP lies in being able to create new classes using the definitions of classes that already exist. This ability is called *inheritance,* sometimes also called derivation. In this chapter we examine how to build *derived classes* from previously defined classes, called *base classes*. We also discuss the different forms of inheritance and provide examples of how they are implemented in VB.NET programs.

The Is-a Relationship

Just as we live in a world of complex relationships and networks, the classes we design in our programs are linked in complex networks and interdependencies. One type of relationship that is frequently represented in OOP is the *is-a* relationship.

The *is-a* relationship models the way specialized objects are formed from more general objects. For example, we can say that a truck *is-a* road vehicle, as is a car and a motorcycle. Road vehicle is a general term for a type of automotive device, while trucks, cars, and motorcycles are more specialized types of road vehicles. These road vehicle types all share certain common characteristics (wheels, seats, motors), but they differ from each other in various ways (cars have four seats, some trucks just have one bench seat, and a motorcycle has its own type of seat).

Another characteristic of *is-a* relationships is that they are inherently hierarchical. We can draw a graph that depicts the relationship between the specialized object and its generalizations. Formally, this type of graph is called a *directed acyclic graph* (DAG). Directed means that you can view the flow of the graph from the top to the bottom. Often this is indicated by arrows pointing from the upper levels to the lower levels of the graph. In terms of the is-a relationship, the graph moves from more specialized objects down to more generalized objects. Acyclic means that the flow from specialized to generalized cannot go the other way. In other words, we can't take a general object and make more specialized objects from them.

An example of a DAG for road vehicles is shown in Figure 6-1.

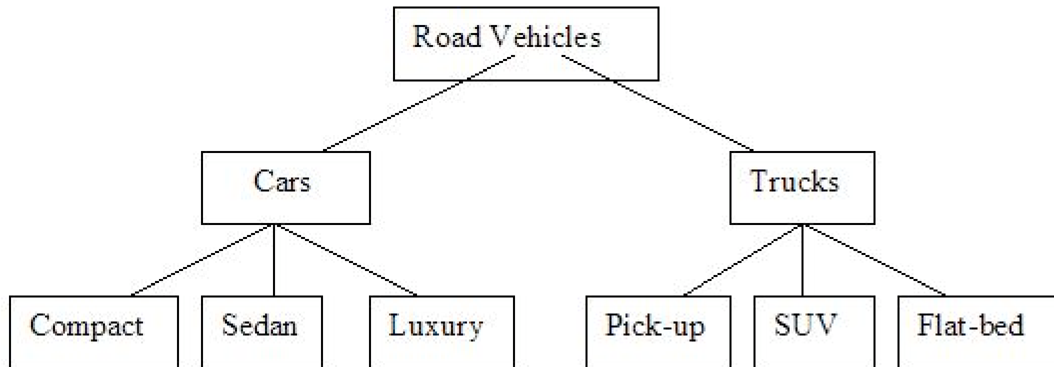

Figure 6-1: a DAG for road vehicles

The top level of the graph holds the general vehicle – road vehicle. The next level down holds the first level of specializations below road vehicle – car, truck, SUV, minivan, and motorcycle. To demonstrate that there can be many levels of specialization in a DAG, the next level of specialization includes different types of trucks, such as pickup trucks and dump trucks.

You've probably noticed that a DAG looks very similar to a company's organizational chart. The difference is that an organizational chart depicts the levels of responsibility in an organization while a DAG depicts the relationships between objects, specifically the relationship between a general object and its specializations. The arrows in the DAG indicate that the graph moves from the general object to more specialized objects.

The term DAG is primarily used in mathematics to refer to one particular type of graph object. The term used in OOP most frequently is *class hierarchy*.

Building Derived Classes

A derived class definition is usually some type of extension of the definition of the base class. Later in this chapter we'll examine in detail the different forms of inheritance. In this section, though, we illustrate the general process of inheritance and discuss many of the techniques used in building derived classes from base classes.

Deriving a PartTimeEmployee Class

Our first example demonstrates how to build a PartTimeEmployee class from an Employee class. This example typifies the most appropriate use of inheritance – to model an "is-a" relationship. An organization usually has several types of employees: full-time employees, part-time employees, managers, and so on. The different types of employee share so many attributes that it makes perfect sense to base the design of the different employee types on an "is-a" basis.

The Employee class includes a set of data members and methods that capture the general behavior of an employee. The data members include the employee's name, their identification number, the department they work in, and their salary. The methods of the class include a constructor for instantiating an Employee object, a Display method, and a method to calculate an employee's pay. The class also include Property methods for setting and getting the Private data member values. The code for the Employee class is shown below.

```
Public Class Employee
   Private pName As String
   Private pID As String
   Private pDept As String
   Private pSalary As Decimal
   Private pPay As Decimal = 0D

   Public Sub New(ByVal initName As String, _
                  ByVal initID As String, _
                  ByVal initDept As String, _
                  ByVal initSalary As Decimal)
      pName = initName
      pID = initID
      pDept = initDept
      pSalary = initSalary
   End Sub

   Public Property Name() As String
      Get
         Return pName
      End Get
      Set(ByVal Value As String)
         pName = Value
      End Set
   End Property

   Public Property ID() As String
      Get
         Return pID
      End Get
      Set(ByVal Value As String)
         pID = Value
      End Set
   End Property

   Public Property Dept() As String
      Get
         Return pDept
      End Get
      Set(ByVal Value As String)
         pDept = Value
      End Set
   End Property

   Public Property Salary() As Decimal
      Get
         Return pSalary
```

```
      End Get
      Set(ByVal Value As Decimal)
         pSalary = Value
      End Set
   End Property

   Public Property Pay() As Decimal
      Get
         Return pPay
      End Get
      Set (ByVal Value As Decimal)
         pPay = Value
      End Set
   End Property

   Public Overridable Function calcPay(ByVal hours As Integer) _
                                    As Decimal
      Return pSalary * hours
   End Function

   Public Overridable Sub Display()
      Console.WriteLine("Name: " & pName)
      Console.WriteLine("ID: " & pID)
      Console.WriteLine("Dept: " & pDept)
      Console.WriteLine("Salary: " & FormatNumber(pSalary, 2))
      Console.WriteLine("Pay: " & FormatNumber(pPay, 2))
   End Sub

End Class
```

The Employee class is developed as a general class for employees, but it doesn't take into account the differences in types of employees. It encapsulates the main attributes of an employee, without specifying the many differences among employees in a company. Part-time employees, for example, work less hours than full-time employees and may not receive benefits. Managers may work on salary and receive a different bonus package than other employees. Because there are different types of employees, yet they share many of the same properties, it makes sense to create derived classes for each employee type.

As a concrete example, let's look at a partial implementation of a derived class for part-time employees. The first thing we have to do is write heading for the new class. When you derive a class from a base class, the second line of the class definition has to specify the base class using the Inherits keyword:

```
Public Class PartTimeEmployee
   Inherits Employee
```

By specifying the Employee class as the base class, we are telling the compiler that we want our new class to have access (at least indirectly) to the properties and methods of the base class. And indeed, because the data members of the base class all have the Private access modifier, the only access we have to the data members of the Employee class is through the public Property methods we defined. We'll see how to use them shortly.

If we have any new data members we want the derived class to store, they are declared next in the class definition. (It should be mentioned that different authors place Private data member declarations in different places. We use the convention of placing them at the beginning of the class definition, but many authors choose to place them at the end of the definition. As always, you may choose to use either convention, but for the sake of those who read your code, be consistent.) We don't need to declare any new data members for this class, so we can move on to defining constructor methods.

The constructor methods of a derived class have a very important responsibility – they must call the constructor to the base class to make sure that the Private data inherited from the base class is initialized properly. And they must also initialize any data members declared in the derived class. For our part-time employee class, we only need to call the constructor for the derived class. To do this, we have a special keyword to use – MyBase. The MyBase keyword, followed by a method name (any method name, not just the constructor), will cause the method (as defined in the base class) to execute. So a call to the constructor in a base class looks like this:

```
MyBase.New(' arguments here)
```

Any arguments that need to be passed to the base constructor have to be passed in the parameter list to the derived class constructor. The constructor method for the PartTimeEmployee class looks like this:

```
Public Sub New(ByVal initName As String, ByVal initID As String, _
               ByVal initDept As String, ByVal initSalary As Decimal)
   MyBase.New(initName, initID, initDept, initSalary)
End Sub
```

It cannot be stressed enough that the call to the base class constructor is necessary to initialize any data that we're inheriting in the derived class.

At this point, we're ready to start defining those methods that describe the behavior of our derived class. Interestingly, though, we don't have any new methods that we need to describe. Generally, whatever methods are defined for an Employee object will work with a PartTimeEmployee object. That does not mean, though, that the methods in the Employee class will work *as is* with the PartTimeEmployee class.

One example is the CalcPay method. A part-time employee can be defined in many different ways, but one way many company's define part-time is working less than 40 hours. The CalcPay method of the Employee class simply takes the hours passed to it and multiplies that value by the employee's salary. To ensure that a number of hours greater than or equal to 40 is not passed to the method, we re-write the CalcPay method in the PartTimeEmployee class to check the number of hours passed to the method. If the value is too large, the user is asked to enter another number and the code loops until a proper number is entered. Here's the code:

```
Public Overrides Function CalcPay(ByVal hours As Integer) As Decimal
   While (hours >= 40)
      hours = CInt(InputBox("Hours cannot be greater than or equal" & _
             " to 40. Enter a new value."))
   End While
```

```
   Return hours * MyBase.Salary
End Function
```

The first thing you should notice about this method is that it uses the Overrides modifier in its heading. As we've discussed before, overriding a method involves reusing a method defined in the base class by redefining the behavior of the method. Let's go back and look at how the method is defined in the base class:

```
Public Overridable Function calcPay(ByVal hours As Integer) As Decimal
   Return pSalary * hours
End Function
```

To allow the method to be overridden in the derived class, we have to mark it Overridable in the base class definition. Without doing so, we cannot redefine the method's behavior in the derived class.

The other interesting feature to note in this method is how we pull the pSalary value out of the base class. You have to remember that all the data for a PartTimeEmployee object is indirectly part of the class because all the data members are declared as Private. We can't make a call like this:

```
MyBase.pSalary
```

to pull out the pSalary value because the data member is declared as a Private member. The only Public access to the Private data is through the Property methods defined in the base class. If we want to access a data member, we have to do so indirectly through a Property method. While this seems like extra work, it is necessary to protect the integrity of the data members of the base class.

The last method we'll look at in this section is the Display method. The base class definition looks like this:

```
Public Overridable Sub Display()
   Console.WriteLine("Name: " & pName)
   Console.WriteLine("ID: " & pID)
   Console.WriteLine("Dept: " & pDept)
   Console.WriteLine("Salary: " & FormatNumber(pSalary, 2))
   Console.WriteLine("Pay: " & FormatNumber(pPay, 2))
End Sub
```

This method does everything we need it to do for both the Employee class and the PartTimeEmployee class, so we don't need to override the method in the derived class, even though we could. In fact, we declare the method as Overridable in the base class just in case a derived class does need to override the method.

We want to use this method as-is in the PartTimeEmployee class, we can call the method directly from the base class. We do this by overriding the Employee class Display method, changing the definition of the method so that it simply makes a call to the base class method, like this:

```
Public Overrides Sub Display()
   MyBase.Display()
```

```
End Sub
```

Building a Graphics Drawing Program Using Class Inheritance

This section describes the beginnings of a graphics drawing applications using an OOP approach. We begin by building a Shape class and then use the class to derive other classes for developing a square. This application again demonstrates how to properly use inheritance to build a hierarchy of related objects that draw on the code written in the parent object.

The first step is to design and build a Shape class from which we'll derive more specific shapes. Our Shape class just needs one data member - the position from the left edge of the screen to start drawing. This is called the offset. We'll have two constructor methods – a default constructor that sets the offset to 0, and a parameterized constructor that allows the user to set the offset. We'll provide a Property method for setting and returning the offset. Next, we need a method that determines where to start drawing and a method that actually writes an asterisk to the screen.

Here's the code for the Shape class:

```
Public Class Shape
   Private pOffset As Integer
   Public Sub New()
      pOffset = 0
   End Sub
   Public Sub New(ByVal offset As Integer)
      pOffset = offset
   End Sub
   Public Property OffSet() As Integer
      Get
         Return pOffset
      End Get
      Set(ByVal Value As Integer)
         pOffset = Value
      End Set
   End Property
   Public Sub DrawAt(ByVal line As Integer)
      Dim count As Integer
      For count = 0 To line - 1
         Console.WriteLine()
      Next
      drawHere()
   End Sub
   Public Overridable Sub DrawHere()
      Dim count As Integer
      For count = 0 To Me.OffSet - 1
         Console.Write(" ")
      Next
      Console.WriteLine("*")
   End Sub
End Class
```

The Shape class now becomes a base class for any specific shapes we might want to draw with our program. To demonstrate how inheritance works for this program, we'll develop a Square class that derives from the Shape class.

The Square class draws a square by first drawing a horizontal line across the top of the square, the two lines for the sides of the square, followed by the horizontal line at the bottom of the square. This algorithm requires us to declare two Private data members for the class – height and width.

The method we call to draw a square is DrawAt. This method is defined in the Shape class, but is Public, so we can call it from within the Square class. This method calls the DrawHere method, which is overridden in the Square class. Let's look first at the definition:

```
Public Overrides Sub DrawHere()
   drawHorizontalLine()
   drawSides()
   drawHorizontalLine()
End Sub
```

The Overrides keyword tells the compiler that this definition supercedes the definition of DrawHere in the base class. A method can only be overridden if the Overridable keyword is used in the definition of the method in the base class. As you recall, the definition of this method in Shape is:

```
Public Overridable Sub DrawHere()
   Dim count As Integer
   For count = 0 To Me.OffSet - 1
      Console.Write(" ")
   Next
   Console.WriteLine("*")
End Sub
```

The DrawHere method body in Shape is completely different than the body defined for the DrawHere method in Square, but notice that the signatures of the methods are the same. Overriding a method involves changing the definition but leaving the method signature the same. The DrawHere method calls three other methods that actually draw the square.

The first method called is DrawHorizontalLine. This method is declared as a Private method, meaning that only code defined within the Square method has access to the method. This method draws the line that forms the top of the square. Here's the code for DrawHorizontalLine:

```
Private Sub DrawHorizontalLine()
   Spaces(Me.OffSet)
   Dim count As Integer
   For count = 0 To width - 1
      Console.Write("*")
   Next
   Console.WriteLine()
End Sub
```

The Spaces method is a Private method that simply writes blank characters from the beginning of a line to the position of the offset. Notice that the argument passed to Spaces is the offset qualified with the keyword Me. The Offset Property method allows us access to the Private data member pOffset because the Property method is declared Public in the Shape class. We have to have an object to qualify our call the Offset method, so we use Me, but we could just as easily have used the MyBase object instead. In this situation, both of these objects give us access to the Offset method.

Once the Spaces method puts us in the right position, the rest of the DrawHorizontalLine method just writes out a series of asterisks that equals the width defined when we instantiated the Square object.

After the top of the square is drawn, DrawHere calls the DrawSides method. This method draws both sides (the vertical lines) within the one loop. If we don't do this, the bottom line will be skewed. The method calls yet another method, DrawOneLineOfSides, which prints an asterisk on the left side of the square, skips over to where the right side is supposed to be, and prints another asterisk. Here's the code for both DrawSides and DrawOneLineOfSides:

```
Private Sub DrawSides()
   Dim count As Integer
   For count = 0 To height - 2
      drawOneLineOfSides()
   Next
End Sub

Private Sub DrawOneLineOfSides()
   Spaces(Me.OffSet)
   Console.Write("*")
   Spaces(width - 2)
   Console.WriteLine("*")
End Sub
```

To end the DrawHere method definition, the DrawHorizontalLine method is called one more time to draw the bottom line of the square.

Polymorphism in Derived Classes

A derived class can contain the same method names as those found in its base class. This polymorphism can be implemented either via overriding and overloading. In this section we look at how method overloading and overriding in derived classes work.

First, though, let's define what we mean by overloading and overriding. When we overload a method, we are giving it additional meanings. As Budd points out, there are many words in the English language that are overloaded. One word that comes to mind is "park." We "park the car" and "take a walk in the park." We know the definition of the word from the context the word is used in. There are many words in our language that have more than one meaning. Overloaded methods in a class are used to perform the same task on different types of objects, as you've already seen with class constructor methods.

Overriding a method means to change the definition of the method for a particular context, or scope. A method in a child class overrides a method in a parent class so that

the method will behave differently than the method in the parent class. Sometimes the derived class method will call the method defined in the parent class and other times the method in the derived class will have a completely unrelated (at least with respect to calling the same method in the base class) to the base class. The child class method that overrides a method in the base class has the same name and signature as the base class method.

Method Overloading By Scope

A method can be overloaded in two different ways – by its scope and by its signature. Both of these overloading techniques play important roles in object-oriented programming, though overloading based on signature is used more often and is what most people think as overloading a method. We'll look first at how methods are overloaded by scope.

When we talk about overloading based on a method's scope, we're usually talking about class scope. Of course, we know that multiple methods can have the same name (but with different signatures) within a single class definition. This is how multiple constructor methods can be defined for the same class. But methods with the same name can exist within unrelated classes that exist within the same program scope. For example, we might have a program that uses objects from a NumberList class and objects from a ApproximateNumber class. Each of these classes might contain a method called Add. The Add method for the NumberList class inserts a new element to the list, while the Add method for the ApproximateNumber class performs addition. Because each method is defined within the scope of different objects, they can coexist within the same program:

```
Dim appNum1  As New ApproximateNumber(23)
Dim appNum2 As New ApproximateNumber(12)
Dim myNums As New NumberList(99)
myNums.Add(appNum1.Add(appNum2))
```

There is no conflict because the compiler can easily resolve each Add method based on its qualifying class type. The Add method for the ApproximateNumber class is not going to be confused with the Add method for the NumList class.

It is considered good programming style to use method overloading in this way. Method names should be short, descriptive, and as natural as possible for the user. In other words, we wouldn't want the code above to look like this:

```
Dim appNum1  As New ApproximateNumber(23)
Dim appNum2 As New ApproximateNumber(12)
Dim myNums As New NumberList(99)
myNums.AddToNumberList(appNum1.AddApproximateNumber(appNum2))
```

Method Overloading By Signature

We have already looked at how method overloading by signature works within a class when we examined class constructor methods. As a review, let's look at the constructor methods for a class that keeps track of time:

```
Public Class Time
    Private pHours As Integer
    Private pMinutes As Integer
    Private pSeconds As Integer

    Public Sub New()
        pHours = 0
        pMinutes = 0
        pSeconds = 0
    End Sub

    Public Sub New(ByVal m As Integer)
        pMinutes = m
        If (pMinutes > 60) Then
            pMinutes -= 60
            pHours += 1
        Else
            pHours = 0
        End If
        pSeconds = 0
    End Sub

    Public Sub New(ByVal h As Integer, ByVal m As Integer, _
                   ByVal s As Integer)
        pHours = h
        pMinutes = m
        pSeconds = s
    End Sub
    Public Sub New(ByVal theTime As Time)
        Me.pHours = theTime.pHours
        Me.pMinutes = theTime.pMinutes
        Me.pSeconds = theTime.pSeconds
    End Sub

    ' ... More class definition code
End Class
```

Each constructor method differs from the other by signature. There is even one more possible constructor method we didn't define – a constructor that has two integer parameters – because we don't want users of the class instantiating objects in that manner.

The constructor examples above involve methods in the same class. Overloading methods in derived classes is achieved in the same way. Let's look at a typical example. The Employee class defined below includes a method for assigning a raise to an employee. The Manager class, which inherits from the Employee class, also includes a Raise method, but it has an extra parameter not found in the Raise definition in the Employee class. This parameter helps determine whether or not a manager gets a regular raise, or a "Bonus" raise. Here's the code:

```
Option Strict On

Public Class Employee
    Private pName As String
```

```
    Private pSalary As Decimal

    Public Sub New(ByVal n As String, ByVal s As Decimal)
        pName = n
        pSalary = s
    End Sub

    Public Sub Raise(ByVal rate As Decimal)
        pSalary *= rate
    End Sub
    Public ReadOnly Property Salary() As Decimal
        Get
            Return pSalary
        End Get
    End Property
End Class

Public Class Manager
    Inherits Employee
    Public Sub New(ByVal n As String, ByVal s As Decimal)
        MyBase.New(n, s)
    End Sub

    Public Overloads Sub Raise(ByVal rate As Decimal, _
                               ByVal type As String)
        If (type = "Regular") Then
            MyBase.Raise(rate)
        ElseIf (type = "Bonus") Then
            MyBase.Raise(rate * 1.05D)
        End If
    End Sub
End Class

Module module1

    Sub main()
        Dim emp1 As New Manager("Jane Doe", 25D)
        Dim emp2 As New Employee("John Doe", 13D)
        emp1.Raise(1.05D, "Bonus")
        emp2.Raise(1.05D)
        Console.WriteLine("Jane's new salary is: " & emp1.Salary)
        Console.WriteLine("John's new salary is: " & emp2.Salary)
        Console.Read()
    End Sub

End Module
```

The Raise method definition in the Manager class includes the keyword Overloads. This lets the compiler know that when the method is called with a Manager object, use the method definition defined in that class, and not the method definition defined in the Employee class. Technically, this type of overloading is called a *redefinition* (Budd, p. 299).

Method Overriding By Replacement

A derived class method that overrides a base class method redefines the behavior of the method from the base class. The methods will have the same name and the same signature, but their definitions (the code in the body of the method) will be different.

The nature of the "redefinition" of the base class method determines whether you use replacement overriding or refinement overriding. Replacement overriding is used when you want to completely change the definition of the base class method in the derived class. As a simple example, look at the method definition below for the Draw() method of the Shape class (this class, which is used to build a text graphics program, and the derived Square class, are defined completely later in the chapter):

```
Public Overridable Sub DrawHere()
   Dim count As Integer
   For count = 0 To Me.OffSet - 1
      Console.Write(" ")
   Next
   Console.WriteLine("*")
End Sub
```

Notice the keyword Overridable in the heading for the method. Any base class method that we will want to override in a derived class must be marked Overridable or we can't use the method name in a derived class.

The method prints a single asterisk at the position computed by the method. The offset is the number of spaces from the left of the console where a shape is to be drawn. A specific shape, such as a square, can't use this method as is, since it only draws a single asterisk. However, you want to use the method name because it exactly defines what you want to happen. Designing classes that use intuitive method names make the interface for that class much easier to use.

Here's the code for the DrawHere method defined in the Square class:

```
Public Overrides Sub DrawHere()
   DrawHorizontalLine()
   DrawSides()
   DrawHorizontalLine()
End Sub
```

The keyword Overrides notifies the compiler that we're overriding the definition of DrawHere found in the base class. The definition of DrawHere in the Square class is completely different, keeping with the concept of replacement. We want to use the same method name for interface consistency, but we need a new definition because this DrawHere needs to do something different than DrawHere in the base class.

Method Overriding By Refinement

When we override a method by refinement, we include a call to the method definition in the base class, adding to it the behavior we want in the derived class. Both sets of code are executed in the method call. While we generally consider constructor methods as

examples of overloading, we still see the technique of refinement used in them. A constructor method defined in a derived class has to make a call to a constructor method in the parent class in order to initialize any data inherited from the parent class.

We can illustrate the principle of refinement with a simple example. The code for a TimeSpan class includes a Display method that shows the date and time stored in a TimeSpan object. We can derive an ExtTimeSpan class from TimeSpan that stores extra information, such as the time zone. First, let's look at the code for the TimeSpan class:

```
Public Class TimeSpan
   Dim pDate As String
   Dim pTime As String

   Public Sub New()
      pDate = Now.Month & "/" & Now.Day & "/" & Now.Year
      pTime = Now.Hour & ":" & Now.Minute & ":" & Now.Second
   End Sub

   Public Overridable Sub Display()
      Console.WriteLine(pDate & " " & pTime)
   End Sub
End Class
```

The behavior that we want to change is in the Display method. In order to override this method in the derived class, we have to designate the method as Overridable. If we don't do this and then try to override a Display method in the derived class, the compiler will complain.

Now let's look at the code for the ExtTimeSpan class:

```
Public Class ExtTimeSpan
   Inherits TimeSpan

   Private pTZone As Zones

   Public Sub New(ByVal tz As Zones)
      MyBase.New()
      pTZone = tz
   End Sub

   Public Overrides Sub Display()
      Dim tzones() As String = {"EST", "CST", "MST", "PST"}
      Dim zone As String = tzones(pTZone)
      MyBase.Display()
      Console.WriteLine(" " & zone)
   End Sub

End Class
```

The Display method in ExtTimeSpan class includes new code, primarily to allow us to pull a string out for the time zone enumeration, and to display the time zone. The call to the base class method Display allows us to access the private data members pDate and pTime.

Polymorphic Variables

A polymorphic variable is a variable that can reference multiple types of objects. There is, of course, a built-in polymorphic data type in VB.NET, Object, but we can also create polymorphic variables from derived classes. In the following sections we discuss how to use polymorphic variables in your object-oriented programs.

The Object Data Type

A variable declared as Object can hold any other data type. When can even assign an instantiated class object to an Object variable legally. For example, given the class definition for ExtTimeSpan given in an earlier chapter, we can write the following lines:

```
Dim thisTime As New ExtTimeSpan(Zones.EST)
Dim thisTimeCopy As Object
thisTimeCopy = thisTime
```

This code works because an Object variable can hold data from any other type, even user-defined class types.

However, just because we can do this doesn't mean we necessarily should. Once we assign a class instance to an Object variable, there's not much we can do with the Object variable. We can't, for instance, access any of the class methods from the instantiated object we assigned to the Object variable. Continuing with the code fragment from directly above, we might want to display the timestamp held in thisTimeCopy. When we try to call the Display method:

thisTimeCopy.Display()

we get an error. While the Object variable can be assigned an instantiated ExtTimeStamp object, it doesn't have access to the methods of the class. We have to either have a declared instance of ExtTimeStamp, or alternatively, we can convert the Object variable temporarily to the proper class type in order to access a class method.

The CType function allows you to convert an object from one type to another type. The format of the function is as follows:

CType(Original-type, Converted-type)

We can use the function in-place, so that once we've performed the conversion we can immediately access a class method or perform some other legal operation based on the type we're converting to. To give the variable thisTimeCopy access to methods from ExtTimeStamp, we can write the following line of code:

```
CType(thisTimeCopy, ExtTimeStamp).Display()
```

which calls the Display method and shows the timestamp stored in thisTimeCopy.

How do we know that an Object variable contains the proper class type we are looking to convert to? The GetType method returns the type of an object. It's general form is:

TypeValue = Object.GetType

where TypeValue is an Object variable.

Using this method, we can write code to test an object for it's type before we call a class method. Here's one way to do this:

```
Sub main()
   Dim thisTime As New ExtTimeStamp(Zones.CST)
   Dim thisTimeCopy, temp As Object
   thisTimeCopy = thisTime
   temp = thisTime.GetType
   If (thisTimeCopy.GetType Is temp) Then
      CType(thisTimeCopy, ExtTimeStamp).Display()
   Else
      Console.WriteLine("Not the correct type")
   End If
   Console.Read()
End Sub
```

The GetType method returns a Reference object, which is why we use the Is keyword rather than the = operator.

Polymorphic Collections of Class Objects

The Object data type gives us lots of flexibility when working with class objects. Many object-oriented programs use the Collection data type to store class objects. The Collection data type is convenient to use because its underlying data structure, the ArrayList, stores all data as Object type. This means that a program that uses many different types of class objects can store them in one data structure for easy access.

This technique is used in Windows applications by Visual Studio.NET. Every form you create in a Windows application has a collection called Controls which contains all the controls you place on the form. Since each control is its own type, the Controls collection has to be able to store any type of object. With all form controls gathered in one place, it is easy to perform tasks on all the forms at once, or at least on similar controls at once, such as clearing the text out of all the textboxes.

There are several ways to gather class objects into a collection. If you are using just one class type, you can add a collection data member to the class and store a new object in the collection every time a constructor method is called. If you make the collection Public, you can access it from the user's code. If you have more than one class type in a program, you can just declare a Collection object in the user code and add new class objects to the collection whenever one is instantiated.

We demonstrate this use of a collection in the program below. We only show the Sub Main() code, since we've previously shown you the code for the TimeStamp class and the ExtTimeStamp class.

```
Sub Main()
   Dim thisTime As New ExtTimeStamp(Zones.CST)
   Dim thatTime As New TimeStamp()
   Dim badTime As DateTime = Now
   Dim times As New Collection()
   Dim theTime, TimeType, ExtType As Object
   TimeType = thatTime.GetType
   ExtType = thisTime.GetType
   times.Add(thisTime)
   times.Add(thatTime)
   times.Add(badTime)
   For Each theTime In times
      If (theTime.GetType Is TimeType) Then
         CType(theTime, TimeStamp).Display()
         Console.WriteLine()
      ElseIf (theTime.GetType Is ExtType) Then
         CType(theTime, ExtTimeStamp).Display()
      Else
         Console.WriteLine("Invalid type")
      End If
   Next
   Console.Read()
End Sub
```

The only tricky part of this code is getting an object to display its timestamp. The problem is each class object, whether it's a TimeStamp object or an ExtTimeStamp object, is stored in the collection as an Object type. We have to use the CType function to get access to the Display method, but we don't know what the class type is when we pull the object out of the collection. We can use the GetType method to pull out the object's underlying class type, but we can't compare it to a string such as "ExtTimeStamp" or "TimeStamp". To solve this, we add two extra variables and assign each class types to the variables:

```
TimeType = thatTime.GetType
ExtType = thisTime.GetType
```

Now we can test these variables with the results of the GetType method to determine whether to use the Display method from the TimeStamp class or the Display method from the ExtTimeStamp method.

Polymorphic Variables Via Inheritance

We can create polymorphic object variables using class objects based on a base class and one or more classes derived from that base class. We can do this due to the principle of substitution, though there are some limits.

An example from the Person-Employee class hierarchy demonstrates how class objects can behave polymorphically. The following lines of code instantiate a Person object and an Employee object:

```
Dim person1 As New Person("Mike", "McMillan", "M", 45)
Dim person2 As New Employee("Terri", "McMillan", "F", 44, _
                           "123", "Nursing", 55000D)
```

Now, even though person1 is instantiated as a Person object, we can assign the object person2 to person1:

person1 = person2

By doing this, we don't gain access to the methods of the Employee class, even though we assigned an Employee object to person1. An instantiated object is limited to the methods of its original type, even if it takes on the type of a different class.

This assignment works because all the data members of the Person class are assigned values from the person2 object of the Employee class. This is what we mean by inheritance substitution – an object of a base class can be used in place of an object of a derived class.

Conversely, we can assign the object person2 to person1, but only if we turn Option Strict off and allow late binding. As we've discussed, this is generally considered a bad programming practice because it can lead to errors in your code and it almost always leads to lowered performance due to the compiler having to perform dynamic type-checking. With Option Strict on, the CLR doesn't allow an implicit conversion from one type to another type, even if they are related by an inheritance chain.

Using Overloading and Overriding to Implement Polymorphism

As you know now, we use the term polymorphism to refer to the ability of an object in VB.NET to have more than one name. There are many different types of polymorphism in VB.NET, including method overloading and method overriding.

Inheritance Forms

There are a number of ways you can use inheritance in your programs. In this section we'll discuss several of the more common forms of inheritance, showing examples where we can (though a couple of the examples must be put off until later chapters).

Specialization and Overriding

When a derived class implements the complete definition of the base class and adds its own functionality to distinguish it from its parent, the derived class is said to be a specialization of the base class. In this form, the principle of substitution is not violated, since an instance of the derived class can be used in place of an instance of the base class. Specialization is the purest form of inheritance and should be the goal of any object-oriented design.

The following VB.NET program demonstrates how specialization works. In the code below, the List class allows the user to keep track of a simple list of string values. The list items are stored in a regular array and when a list is displayed, the items are shown in the order they were entered into the list.

For various reasons, we may want to keep our list items sorted in alphabetical order. Rather than redefine the List class, we choose instead to build a derived class, SortedList, that inherits all the functionality of the List class. In order to keep its data in sorted order, the SortedList class redefines the Add method from the List class so that when an item is added to the list, the array is then sorted.

Redefining a base class definition like this is called *overriding* and is one of the ways polymorphism is implemented in VB.NET. A base class method can be overridden when it is defined as Overridable in the base class body. For example, the Add method in the List class takes the following form:

```
Public Overridable Sub Add(item As String)
  store(pos) = item
  pos += 1
End Sub
```

Now, in the body of the SortedList class, we write a new definition of the Add method:

```
Public Overrides Sub Add(item As String)
  store(pos) = item
  pos += 1
  Array.Sort(store)
End Sub
```

We'll see many more examples of method overriding in this chapter.

Redefining the Add method this way is one way the SortedList class is a specialization of the List class. The Add method, though, is not the only List method that has a new definition in the SortedList class. The Display method also has to be overridden in SortedList so that blank elements of the array are skipped and not written out.

Let's look at all the code for both classes and a program that tests them:

```
Option Strict On
Imports System
Module ListClass
```

```
Public Class List
  Protected store() As String
  Protected pos As Integer = 0

  Public Sub New(n As Integer)
    Redim store(n)
  End Sub

  Public Overridable Sub Add(item As String)
    store(pos) = item
    pos += 1
  End Sub

  ReadOnly Property Count() As Integer
    Get
      Return pos
    End Get
  End Property

  Public Overridable Sub Display()
    Dim i As Integer
    For i = 0 to pos
      Console.WriteLine(store(i))
    Next
  End Sub
End Class

Public Class SortedList : Inherits List

  Public Sub New(n As Integer)
    MyBase.New(n)
  End Sub

  Public Overrides Sub Add(item As String)
    store(pos) = item
    pos += 1
    Array.Sort(store)
  End Sub

  Public Overrides Sub Display()
    Dim i As Integer
    For i = 0 To pos+1
      If (store(i) <> "") Then
        Console.WriteLine(store(i))
      End If
```

```
        Next
      End Sub

    End Class

    Sub Main()
       Dim glist as New SortedList(4)
       glist.Add("Milk")
       glist.Add("Eggs")
       glist.Add("Bread")
       glist.Display()
    End Sub

End Module
```

Specification

After specialization, one of the most common uses of inheritance is to meet the specifications of a base class. A derived class whose definition primarily implements a set of methods and other definitions found in its base class is said to be a *specification* of the base class.

Specification is usually used when a class implements an interface, which is a special class type that specifies only a set of methods that any subclass of the interface must implement. This set of methods is said to form a contract that any subclass must abide by. Interface implementation is examined later in this book.

Interface implementation is not the only type of specification. VB.NET allows the definition of an *abstract* class, which is a class that specifies a set of methods (much like an interface) but cannot be used to create direct instances. An abstract class can only be used as a base class for derived classes, which is quite similar to an interface. The chapters on interfaces and abstract classes discuss the differences between these two class inheritance forms.

Construction

The construction form of inheritance involves designing a derived class from a base class where the methods of the derived class are simply renamed from the base class or the signatures of the methods are changed. This type of inheritance is not the same as specification because the principle of substitution doesn't hold, which is one reason why it is not usually a good idea to use construction. In other words, if class A is derived from class B using construction, you could not use an instance of class A anywhere you would use an instance of class B.

An example of a derived class that is built for construction is a Platypus class that derives from a Mammal class. The platypus is a mammal, but in one nature's little quirks, gives birth by laying eggs rather than by live birth. So, if the method of birth is represented in the Mammal class as a method, we have to override this method in the

Platypus class to ensure that the class properly reflects nature. A VB.NET program that simulates this relationship is shown below.

```
Module Module1
   Public Class Mammal
      Private pHasHair As Boolean

      Public Sub New()
         pHasHair = True

      End Sub
      Public ReadOnly Property HasHair() As Boolean
         Get
            Return pHasHair
         End Get
      End Property
      Public Overridable Sub BirthMethod()
         Console.WriteLine("Live birth")
      End Sub
   End Class
   Public Class Platypus
      Inherits Mammal
      Public Sub New()
         MyBase.new()
      End Sub
      Public Overrides Sub BirthMethod()
         Console.WriteLine("Lays eggs")
      End Sub
   End Class
   Sub Main()
      Dim anAnimal As New Mammal()
      anAnimal.BirthMethod()
      Dim anotherAnimal As New Platypus()
      anotherAnimal.BirthMethod()
      Console.Read()
   End Sub

End Module
```

Generalization

A derived class that is built for generalization extends the behavior of the base class to create a more general object. This is somewhat the opposite of the specialization form of inheritance since the derived class is more general than the base class. Generalized derived classes usually override methods in the base class in order to make the base class's methods apply to more general circumstances.

As an example of generalization, let's look at classes that let us work with ranges of numbers. One range of numbers most people are familiar with are the grades you can score on a test. Typically, a legal grade is a value from 0 to 100. We can design a simple class that encapsulates this requirement:

```
Public Class Grade
   Private Const MIN_VALUE As Integer = 0
   Private Const MAX_VALUE As Integer = 100
   Private theData As Integer

   Public Sub New(ByVal g As Integer)
      If (g < MIN_VALUE Or g > MAX_VALUE) Then
         g = CInt(InputBox("Enter a valid grade: "))
      End If
      theData = g
   End Sub
   Public Sub LetterGrade()
     Select Case theData
        Case 90 To 100
           Console.WriteLine("A")
        Case 80 To 89
           Console.WriteLine("B")
        Case 70 To 79
           Console.WriteLine("C")
        Case 60 To 69
           Console.WriteLine("D")
        Case 0 To 59
           Console.WriteLine("F")
        Case Else
           Console.WriteLine("Invalid grade")
     End Select
   End Sub
End Class
```

This class works fine for test grades, but it is not general enough to use with any other type of numeric range. We can derive a class from Grade that works with many different types of numeric ranges, making the class more general than the Grade class. Here we'll show just the constructor method of this class:

```
Public Class IntNumericRange
   Inherits Grade
   Public Sub New(ByVal n As Integer)
      MyBase.New(n)
   End Sub
' Other class methods here
End Class
```

We only show a brief definition of this class because clearly we have the inverted the proper class hierarchy by deriving a more general class from a more specific class. A better idea is to create a general NumericRange class (even more general than IntNumericRange) and deriving classes for more specific class types, such as a test grade or a range of temperatures, for example.

Extension

When a derived class provides new functionality as compared to its base class, then the derived class is said to be an extension of the base class. This form differs from

generalization because an extended derived class will contain methods not found in its base class, rather than simply overriding methods in its base class.

An example of extension is deriving a class that works with sets from a class that works with just lists of objects. The following program demonstrates how extension works.

```
Option Strict On
Imports System
Module Module1

    Public Class List
        Protected store As New ArrayList()

        Public Overridable Sub Add(ByVal item As String)
            store.Add(item)
        End Sub

        ReadOnly Property Count() As Integer
            Get
                Return store.Count
            End Get
        End Property

        Public Overridable Sub Display()
            Dim obj As Object
            For Each obj In store
                Console.WriteLine(obj)
            Next
        End Sub
    End Class

    Public Class SortedList : Inherits List

        Public Overrides Sub Add(ByVal item As String)
            store.Add(item)
            store.Sort()
        End Sub

        Public Overrides Sub Display()
            Dim obj As Object
            For Each obj In store
                If Not (obj Is Nothing) Then
                    Console.WriteLine(obj)
                End If
            Next
        End Sub

    End Class
    Public Class SetList : Inherits SortedList

        Public Function Intersection(ByVal slist As SetList) As SetList
            Dim TempSet As New SetList()
            Dim obj As Object
            For Each obj In Me.store
                If (slist.store.Contains(obj)) Then
```

```
                    TempSet.Add(CStr(obj))
                End If
            Next
            Return TempSet
        End Function
    End Class
    Sub Main()
        Dim names1 As New SetList()
        Dim names2 As New SetList()
        Dim NamesIntersect As New SetList()
        names1.Add("Mike")
        names1.Add("Terri")
        names1.Add("Meredith")
        names1.Add("Allison")
        names1.Add("Mason")
        names2.Add("Mike")
        names2.Add("Allison")
        names1.Display()
        names2.Display()
        NamesIntersect = names1.Intersection(names2)
        Console.WriteLine("Intersection of names1 and names2")
        NamesIntersect.Display()
        Console.Write("Finished")
        Console.Read()
    End Sub

End Module
```

The SetList class provides one new method, Intersection, that is not part of the SortedList class at all. Other than this one method, though, along with other Set-specific methods you may add (such as Union, etc.), the functionality of the SetList class is the same as the SortedList class.

Limitation

A derived class takes the limitation form when the derived class limits the functionality of its base class. This occurs when methods in the base class are modified or rendered unusable by the derived class.

An example of this form is a class that inherits from the CollectionBase class in order to build a custom collection of another class type. The arraylist that is used in the class to store data is typed Object, meaning that any data can be stored in the collection. If we want a strongly-typed collection storing only the properly instantiated class objects, we can override methods in the CollectionBase class to ensure that only the proper objects are added to the collection.

In the example below, a NameCollection class is created to store objects of NameType type. The Add method from the CollectionBase class is modified so that only NameType objects can be added to the collection. An exception is flagged at design-time if a non-NameType object is passed to the Add method.

```
Public Class NameCollection
   Inherits CollectionBase
```

```
    Public Sub New()
        MyBase.New()
    End Sub

    Public Overloads Sub add(ByVal aName As NameType)
        list.Add(aName)
    End Sub

End Class
```

Variance

A derived class is subclassed for variance when the derived class and the base class have similar definitions but there is no clear hierarchical relationship that indicates the derived class should be lower in the hierarchy than the base class.

The canonical example of subclassing for variance is a set of classes designed for controlling computer pointing devices, such as a mouse and a tablet. Each of these devices will contain about the same methods, since each class performs essentially the same task. Yet it is not clear which class should be the base class and which class should be the derived class, so often an arbitrary choice is made.

Situations that lead to inheritance via variance should probably be redesigned by abstracting out as much common code as possible and putting it into an abstract base class. Each device can then inherit its behaviors from the abstract base class, leading to a clear hierarchical relationship between the subclasses.

The Advantages and Disadvantages of Inheritance

There is a long-going debate in the programming community over using inheritance versus using composition to develop hierarchies of classes. Many OOP experts believe inheritance is used in many situations where composition is more appropriate. We're not ready to join this discussion, since we haven't explored composition yet, but we can present the advantages and disadvantages of using inheritance.

Advantages of Inheritance

Code reuse – Inheriting the properties and methods of a base class means that you don't have to rewrite them in the derived class. In programs that are made up of unrelated functions and subroutines, it is easy to rewrite the same piece of code many times. Code reuse saves time, since the time you spent rewriting code can now be spent adding new functionality, and it saves on time spent doing debugging and maintenance. Inherited code has already been tested and debugged in the base class (or at least it should have been), so the code should be reasonably reliable for use in a derived class.

Code sharing – Once a class hierarchy is developed, the classes developed can be used in many different projects by many different programmers. We can consider this macro-level code sharing. At a micro level, on the other hand, classes that inherit from the same base class are obviously sharing the same code base. Object that are created from the derived classes all share the same code inherited from the base class.
Behavior consistency – Derived classes that inherit from a base class will contain code that behaves in the same way. Users of these classes can be assured that when they call a method they will know exactly what behavior to expect.
Software components – The use of inheritance allows programmers to build libraries of classes that can act like integrated circuits used in designing electronic devices. These components can be plugged right into a program without any customization, unless, of course, the programmers wants to customize. An excellent example of how derived classes can be used as components is the .NET Framework Library.
Rapid prototyping and development – Once we have a library of debugged software components, we can concentrate on developing the novel aspects of an application without worrying about the details handled by the components. This means we can develop systems more quickly, allowing the programmer to easily create system prototypes that can be shown to end users for their feedback. Once feedback on the prototype is obtained, the programmer can make changes to the system quickly and efficiently so that end users can see the new system and provide feedback, sending the programmer back to make more changes. This loop can occur several times before the final system is ready for release. Design and development in this manner was not nearly as easy before the development of object-oriented programming languages.
Polymorphism and high-level software design – Software is typically written from the bottom up, even though it is designed from the top down. There can be several "layers" of code before an application is complete. Typically, the code written at the lower levels of a program are more portable than the code written at a higher level of abstraction. For example, a function to retrieve data from a textbox can be reused in almost any place, while a function that calculates an employee's weekly paycheck is much less portable. Methods designed in base classes can be overridden so that their use can be customized to fit the particular task to be performed.
Information hiding – When data and methods are encapsulated in a class, the user of that class, be it a base class or a derived class, doesn't have to consider all of the implementation details of the class definition. The user only needs to understand the public interface to the class in order to use it properly. Encapsulation makes software designed in an object-oriented way easier to write, debug, and maintain and the complexity of the application is reduced as well.

Disadvantages of Inheritance

The primary disadvantages of inheritance have to do with performance issues that are common to all object-oriented programming languages. These efficiency issues are:

> Execution speed – Task-specific code runs faster than general-purpose code. Methods inherited from a base class are by definition more general than a new method defined in the derived class. Yet this issue is often a red herring. The slowdown in code execution may be small and can be balanced by a general increase in development time. Most experts suggest that developers use object-oriented principles in their coding to reduce the complexity of the system and, once the system is designed and coded, use a code profiler to see where the bottlenecks, if any, are located. Then the programmer can consider using optimization techniques to improve the performance of the code in question.
>
> Application size – As you've surely learned by now, applications developed using object-oriented techniques contain more lines of code than applications designed using a procedural approach. The size of an application isn't really an issue these days since memory is so inexpensive in relation to the costs of software development. In other words, designing applications in an object-oriented manner is less expensive overall than using older procedural approaches.
>
> Message-passing overhead – Code that implements message-passing is less efficient than code that calls subprograms. As Budd puts it, though, concern about this inefficiency, like concern over execution speed, is often "pennywise but pound-foolish." The gains in reduced program complexity, reduced development costs and reduced maintenance costs will probably outweigh the small gains in efficiency gained by writing procedural code. Also, there are ways to increase the efficiency of object-oriented software, as we discuss in this chapter and throughout the book.
>
> Application complexity – Overusing inheritance can create programs that are just as complex as the most subprogram-polluted procedural program ever developed. A complex hierarchy of derived classes can be very hard to understand, causing what Budd calls the "yo-yo" problem, where the developer is constantly moving his or her head up and down scanning the hierarchy chart.

Summary

Inheritance is one of the big three features in OOP, along with encapsulation and polymorphism. Inheritance is important because when we define code in a base class, that code can be automatically reused in a derived class. Another useful feature of inheritance is the ability of a derived class to be used in situations where the base class would normally be used. This is called substitution and it plays an important role in OOP.

The chapter discusses the different forms of inheritance commonly used in OOP and the chapter ends with a discussion of the advantages and disadvantages of using inheritance.

Exercises

1. Create a List class that stores data in an array. The class can allow duplicate items to be stored in the array. Write methods for adding an item to the list, deleting an item from the list, displaying the items in the list, and clearing the list. Develop a program to test your List class implementation.
2. Create a Set class that inherits from the List class. A Set is a list that doesn't allow duplicates, so modify the Add method accordingly. Also create methods for performing the union of two sets and the intersection of two sets.
3. Use the List class developed in Exercise 1 as the base class for a derived class named ListSorted. An instance of this class keeps the items it stores in sequential order (either numerical or string). Implement the Add method so that a new item added to the list is inserted in the proper place. Also implement a Delete and Display method.