Visual Basic 2008 Programming

Business Applications with a Design Perspective

Jeffrey J. Tsay

Table of Contents

Chapter 13

Object-Oriented Programming

This chapter continues the discussion of object-oriented programming, which includes three important features: encapsulation, inheritance, and polymorphism. Encapsulation was discussed in detail in Chapter 12, "Object-Based Programming." This chapter focuses on inheritance and polymorphism. Inheritance allows a new class to extend the functionality of an existing class without having to change the source code of the existing class. Polymorphism allows different classes to provide functionality with some invariant methods or properties but different implementations. These features greatly enhance the flexibility of coding and using classes, as covered in this chapter.

After completing this chapter, you should be able to:

- Explain the basic concepts and terms in inheritance and polymorphism,
- Design and develop code for the base class,
- Develop code for classes, derived from a base class, that are developed from scratch or available in the .NET framework, including classes pertaining to event arguments,
- Create and design forms derived from existing forms,
- Develop code for base classes and derived classes to implement polymorphism, and
- Develop code to implement interface-based polymorphism.

13.1 Inheritance

Inheritance allows the creation of a new class from an existing (base) class. The new class inherits all the properties, methods, and events from the *base class*. The new class is recognized as the *derived class*. In other languages, the base class is also called the *super class*, while the derived class is called *subclass*. You can add new code in the derived class to extend the functionality of the base class. To create a derived class, you use the *Inherits statement*. For example, suppose you have a base class called RealEstate from which you want to derive a new class called Residential. Your code for the Residential class will appear:

```
Public Class Residential
    Inherits RealEstate
    'Other statements
End Class
```

Inheritance Hierarchy

A class that does not explicitly inherit from any class actually inherits from the *System.Object class*. Because a class either inherits implicitly from the System.Object class or explicitly from another class, which in turn either inherits implicitly from System.Object or explicitly from yet another class, all classes can trace their heritage to the System.Object class; that is, all classes are derived from the System.Object base class.

A new class can inherit from a base class that is derived from yet another class, forming an *inheritance hierarchy*. There is no limit as to how tall this hierarchy can be; however, a tall inheritance hierarchy can cause maintenance problems. An update in a base class may affect countless derived classes, thus affecting many applications using these classes. (Imagine if the System.Object class is

changed drastically, what could happen to all classes.) A practical suggestion is to limit the hierarchy to about five or six layers.

Why Inheritance?

The essence of object-oriented programming is inheritance. It provides a way to organize the code from the most general to the most specific. For example, you can design your hierarchy of classes for the real estate application beginning with a set of properties and methods for the "RealEstate" class to a more specific classification of residential property versus commercial property. Residential can be further classified into single-household dwelling, multiple-household dwelling, and so on. As the hierarchy goes down, more specific properties and methods can be added to each class. Through inheritance, all the functionality of the base class is provided to the derived class without any additional code, so inheritance enhances code reusability.

An Inheritance Example

Consider a simplified payroll application for a restaurant that has two classes of employees: salaried, and tip earners. All employees have common properties such as social security number, address, and date of employment. The formula to compute the payroll tax withholding is also the same for all employees. Basically, the withholding is based on the employee's gross earnings, which may be different from the gross pay. In the case of the salaried employee, the gross pay is the same per pay period and is equal to the gross earnings. In the case of the tipped earner, however, the gross pay is computed by the hours worked times the wage rate. The gross earnings should include both the gross pay and tips, which are paid by the customer directly, not by the company. The net pay is the difference between gross pay (not gross earnings) and the withholding. The common and unique characteristics of each type of employee can be summarized as in the following table:

Common to all employees	**Unique to the salaried employee**	**Unique to the tipped earner**
SSN, name, address, date of employment	Gross pay = amount per period	Tips should be reported
Tax withholding = f(gross earnings)	Gross earnings = gross pay	Gross pay = hours worked × wage rate
Net pay = gross pay - withholding		Gross earnings = gross pay + tips

To code for the project you can create a class that contains the common characteristics of the all the employees and call the class **Employee**. The characteristics unique to each type of employee will be coded in separate classes. These classes can be called **SalariedPerson** and **TippedEarner**. Each should be derived from the Employee class so that it has the functionality of the Employee class.

The Employee Class

For simplicity, you will include only one property (SSN) in this class. Also, assume that the tax withholding is 15% of the gross earnings. Coding this class is no different from coding any class we have done so far. The code appears as follows:

```
Public Class Employee
    Private mSSN As Integer
    Public Property SSN() As Integer
        Get
```

```
            Return (mSSN)
        End Get
        Set(ByVal Value As Integer)
            mSSN = Value
        End Set
    End Property
    Function WithHolding(ByVal pGrossEarnings As Double) As Double
        'Assume 15% is the standard w/h
        Return (pGrossEarnings * 0.15)
    End Function
    Function NetPay(ByVal pGrossPay As Double, ByVal pWithHolding As Double) As
      Double
        Return (pGrossPay - pWithHolding)
    End Function
End Class
```

The Derived Classes

The Salaried class will be derived from the Employee class. The two methods used to compute the gross earnings and gross pay involve the straightforward assignment statements. The code should appear as follows and should be placed in a new class module within the same project.

```
Public Class SalariedPerson
    Inherits Employee
    Private mGrossPay As Double
    Private mGrossEarnings As Double

    Public ReadOnly Property GrossEarnings() As Double
        Get
            Return (mGrossEarnings)
        End Get
    End Property

    Public Function GrossPay(ByVal pSalary As Double) As Double
        mGrossPay = pSalary
        mGrossEarnings = mGrossPay
        Return (mGrossPay)
    End Function
End Class
```

Notice that there is an Inherits statement immediately following the Class definition. The statement in effect incorporates all the functionality of the Employee class in the SalariedPerson class. An object instantiated from this class automatically has the SSN property. In addition, it has the WithHolding and NetPay methods to compute the withholding and net pay for the salaried employee. Notice also that GrossEarnings is a read-only property, which should always be set equal to the gross pay for this class. If it is set up as both read and write, it can be accidentally set to a value different from the gross pay. Included also is a Private variable mGrossPay to retain the value of GrossPay in case it is needed for additional computations in this class.

The TippedEarner class has a Tips property. In addition, the gross pay is computed differently; however, it should still be derived from the Employee class so that it has the functionality of the Employee class. The code appears as follows:

```
Public Class TippedEarner
    Inherits Employee
    Dim mGrossEarnings As Double
    Dim mGrossPay As Double
```

```
    Dim mTips As Double

    Public ReadOnly Property GrossEarnings() As Double
        Get
            Return (mGrossEarnings)
        End Get
    End Property

    Public Property Tips() As Double
        Get
            Return (mTips)
        End Get
        Set(ByVal Value As Double)
            mTips = Value
            mGrossEarnings = mGrossPay + mTips
        End Set
    End Property

    Function GrossPay(ByVal HoursWorked As Double, ByVal WageRate As Double) As
      Double
        mGrossPay = HoursWorked * WageRate
        mGrossEarnings = mGrossPay + mTips
        Return (mGrossPay)
    End Function
End Class
```

Notice that GrossEarnings is a read-only property computed when the Tips property is set, or when the gross pay is computed. A Private copy of GrossPay is retained for other computations.

Using the Derived Classes

So, how can you use the derived classes? As mentioned, the derived classes have the functionality of their base class in addition to the additional features that are extended in the derived classes. The TippedEarner class now has the Tips and GrossEarnings properties as well as the GrossPay method on top of the Withholding and NetPay methods in the base Employee class. To test, place the following code in a button Click event procedure in a form:

```
    Dim Tipped As New TippedEarner()
    Dim I As Integer
    Dim TheGrossPay As Double
    Dim TheWithholding As Double
    Dim TheNetPay As Double

    Tipped.Tips = 1000
    TheGrossPay = Tipped.GrossPay(160, 5)
    TheWithholding = Tipped.Withholding(Tipped.GrossEarnings)
    TheNetPay = Tipped.NETPay(TheGrossPay, TheWithholding)
    MsgBox("The net pay is " & TheNetPay)
```

When you click the button, you should see the message box display 530 as the net pay. The computation is correct because the withholding is based on the total earnings of 1800 (tips plus the gross pay), while net pay is the difference between gross pay (800) and the withholding (15% of 1800 = 270).

You might have noticed an inconvenience in using the object. You must explicitly call the WithHolding and NetPay methods to obtain the withholding and net pay amounts. Ideally, the object should automatically compute these two values after the tips are reported and gross pay is computed. This issue will be revisited later in this section when *Protected members* are discussed.

Inheriting from VS 2008 Classes

The preceding discussion illustrates how to derive a new class from another class that is developed from scratch. You can create a new class from most of the VS 2008 classes in developing your VB projects; that is, you can actually extend the functionality of any control. As an example, the masked text box will be emulated by extending the text box. This emulated control will be named MaskEdit.

The added functionality of MaskEdit comes from its mask property. You can add the Mask property and the needed code to make the property function properly in the derived class. For simplicity, the mask will recognize only the # symbol as the mask for the numeric key. Other characters in the mask will be treated as placeholder literals. When the mask is set at run time, the placeholder literals will be displayed in the text box literally and cannot be changed by the user; however, all the positions with the # symbol will be displayed as blank spaces. Of course, these are the positions that the user can enter only numeric keys. When the user attempts to enter a non-numeric key, the routine will display the "numeric key only" message.

The initial code for the class module should appear as follows:

```
Public Class MaskEdit
    Inherits TextBox
    Private mMask As String
    Public Property Mask() As String
        Get
            Return (mMask)
        End Get
        Set(ByVal Value As String)
            <To be added>
        End Set
    End Property
End Class
```

You have coded the read-side of the Mask property to return its private copy. Now consider the write-side. Recall that when the mask is set, all the literals in the mask should be displayed, while the # symbols are displayed as blank spaces. It is easier to handle this with another Private variable (call it mDisplayText). This variable will be initially assigned the same value as the mask. All the # symbols in it are replaced with blank spaces, and the result is then displayed in the Text property inherited from the text box class. It turns out that displaying the text is not exactly straightforward. You can write a sub procedure (called ShowDispayText) to handle this. The Mask property's Set procedure appears as follows:

```
    Private mDisplayText As String 'Place this line with mMask declaration

    Set(ByVal Value As String)
        mMask = Value
        mDisplayText = Replace(mMask, "#", " ") 'Replace all # with blank space
        ShowDisplayText()
    End Set
```

The ShowDisplayText sub procedure can also be used to display the text after the user enters a key. Consider the KeyPress event procedure first. This procedure is similar to the key validation procedure shown in Chapter 10, "Special Topics in Data Entry." The code appears as follows:

```
Private Sub MaskEdit_KeyPress(ByVal sender As System.Object, ByVal e As
  System.Windows.Forms.KeyPressEventArgs)  Handles MyBase.KeyPress
    Dim KeyAscii As Integer
    Dim Pos As Integer
    Pos = SelectionStart + 1 'This is the position in which the key is entered
    KeyAscii = AscW(e.KeyChar)
    If KeyAscii < Keys.Space Then
```

```
        'Control key; let the system handle it
        Exit Sub
    End If
    If Len(mMask) = 0 Then
        'no mask; take anything
    ElseIf Pos > Len(mMask) Then
        ' Beyond the mask area; no key accepted
        e.Handled = True
    Else
        Select Case Mid(mMask, Pos, 1)
            Case "#"
                If KeyAscii>= Keys.D0 AndAlso KeyAscii <= Keys.D9 Then
                    ' Numeric key; accept and display
                   ShowDisplayText(Pos, KeyAscii)
                Else
                    ' non-numeric key; display an error message
                        MsgBox("Numeric key only")
                End If
                    e.Handled = True 'Don't process the key any further
            Case Else
        End Select
    End If
End Sub
```

Notice that you are coding in the MaskEdit class. The KeyPress event is inherited from the base class, the text box. To refer to an interface (method, property, or event) in the base class, qualify it with the *MyBase* keyword. The KeyPress event procedure header indicates that it handles the event in the MyBase.KeyPress text box class. Recall that when you code any event procedure for a form, the Handles clause always qualifies the event name with the MyBase keyword. On the other hand, the inherited properties and methods are generally referenced without any qualification; therefore, the text box's SelectionStart property is simply referenced as such in this event procedure.

The procedure first tests whether the key is a control key. If so, there is no need to handle it (exit the procedure and allow the system to handle the key). It then checks whether there is a mask. If there is no mask, nothing needs to be done; otherwise, if the current cursor position is beyond the mask, the key should be suppressed (no key should be entered). If a key is entered within the boundary of the mask, it is tested to see whether it is a numeric key. If so, it is displayed by the ShowDisplayText sub; otherwise, a "numeric key only" message is displayed. The key should not be further processed in either case, and therefore the Handled argument is set to True. In the first case, the key has already been displayed. In the other case, the key should be rejected.

Why does displaying a character in the text box take a special routine (ShowDisplayText)? The need stems from handling the cursor position after the key is displayed. Consider the simple mask ##-##-##. After the second digit is entered, the cursor should be moved to the fourth position, skipping the first "-" (which should never be overwritten). After a digit is displayed, the routine should look for the next # character to place to cursor. When there is no more # character, the cursor should be placed at the end of the text so that nothing else can be typed over. This need to place the cursor in the correct position also arises when the mask is initially set, although there is no particular digit to display but just the initial display text. The code for the ShowDisplayText sub appears as follows:

```
Private Sub ShowDisplayText(Optional ByVal Pos As Integer = 0, Optional ByVal
  KeyAscii As Integer = 0)
    Dim P As Integer
    If KeyAscii > 0 Then
        ' If there is a key, put it in its proper position
        Mid(mDisplayText, Pos, 1) = Chr(KeyAscii)
```

```
    End If
    ' Show the result in the text box
    Text = mDisplayText
    P = InStr(Pos + 1, mMask, "#") - 1
    If P >= 0 Then
        MyBase.Select(P, 0)
    Else
        MyBase.Select(Pos, 0)
    End If
End Sub
```

Notice that both of the parameters are optional. If the key code is passed, it is inserted to the specified position. The display text—mDisplayText, which holds what should appear in the text box—is assigned to the Text property so that it will appear in the text box; then the next "#" position is searched for. The text box's Select method is used to set the cursor position. Recall that the Select method takes up to two parameters. The first parameter specifies the position to place the cursor and the second specifies how many characters to highlight. A zero will indicate to highlight nothing. Notice the first position in the Select method is zero, while the first position for the InStr (and the Mid) function is 1; therefore, 1 is subtracted from what is returned from the InStr function.

Referencing the Class Itself and the Base Class

Notice also how the Select method is qualified. The method name conflicts with the reserved word for "Select Case;" therefore, it must be qualified by either *Me* or *MyBase*. Me refers to the current (derived) class, while MyBase refers to the base class. Because the method is inherited, either keyword will do. Again, notice that the Text property is used without being qualified because there is no name conflict. Suppose you want to create another property with the same name, Text, in this derived class. Are you allowed? Yes. In this case, you will be shadowing the base property, so you must declare it with a special keyword, Shadows as follows:

```
    Shadows Property Text () As String
```

To refer to Text in the text box, you code MyBase.Text; to refer to the Text property in this derived class, you code Me.Text.

This emulated masked edit class is used to illustrate several points: how classes can be extended in VS 2008; how inherited events, properties, and methods are referenced in the derived class; and how to declare and reference properties and methods that shadow members in the base class. Notice, however, that this derived MaskEdit class has not been completely coded. It handles only one kind of mask character—#. Also, no code is provided to handle the arrow keys and the Backspace key. The desired refinements are left to you as exercise 13-19 at the end of this chapter.

Creating Event Arguments

In Chapter 12, it was shown in the SortEngine example how events can be declared and raised in a class. There, the event was created with two arguments: Percent and TimeLeft; however, all events provided by the system have a standardized format: The first argument is the sender object and the second is an event argument class. MS recommends that events be structured in this format, and that the event argument class be inherited from the System.EventArgs class. To conform to this format, you can modify the project as follows:

- Create an event class (call it SortEngineEventArgs) that inherits from System.EventArgs and will include Percent and TimeLeft as additional properties. This class should be Public, and can be separate from the SortEngine class (does not have to be an inner class of SortEngine).
- Change the event declaration.

- Change the arguments in the RaiseEvent statement.

The SortEngineEventArgs class can be coded as follows:

```
Public Class SortEngineEventArgs
    Inherits System.EventArgs
    Dim mPercent As Integer
    Dim mTimeLeft As Double

    Sub New(ByVal pPercent As Integer, ByVal pTimeLeft As Double)
        mPercent = pPercent
        mTimeLeft = pTimeLeft
    End Sub

    Public ReadOnly Property Percent() As Integer
        Get
            Return (mPercent)
        End Get
    End Property
    Public ReadOnly Property TimeLeft() As Double
        Get
            Return (mTimeLeft)
        End Get
    End Property
End Class
```

The two arguments Percent and TimeLeft are included in the class as read-only properties. Their values are initialized in the New constructor, which takes Percent and Timeleft as its two required parameters.

In the SortEngine class itself, the PercentChanged event can then be declared as follows:

```
Public Event PercentChanged(ByVal sender As System.Object, ByVal e As
  SortEngineEventArgs)
```

This declaration now conforms to the MS recommendation. In the sorting routine itself, everything remains the same except for the RaiseEvent statement, which should be changed as follows:

```
RaiseEvent PercentChanged(Me, New SortEngineEventArgs(Percent, TimeLeft))
```

Notice that the object that raises the event is the SortEngine class itself; therefore, Me is the first argument. The second parameter expects a SortEngineEventArgs object, so one instance of this class is created with the New keyword. Percent and TimeLeft are passed to the constructor, which expects these two parameters.

After these changes, in the module that uses the SortEngine class, you should see that the event procedure header appears as follows (assuming Sorter is declared WithEvents as SortEngine):

```
Private Sub Sorter_PercentChanged(ByVal sender As System.Object, ByVal e As
  SortEngineEvent.SortEngineEventArgs) Handles Sorter.PercentChanged
```

You can then reference Percent and TimeLeft in the event handler by qualifying each with the e parameter as follows:

```
    e.Percent
    e.TimeLeft
```

Hiding Members from Classes Outside the Class Hierarchy

Consider the code in a base class. Suppose you would like for a method or property to be exposed to any class that inherits this base class, and yet the method or property should not be exposed to any other classes. Declaring such a member as Friend will not do because this member will still be visible to all other modules in the same project. Declaring it as Private, however, will make the member invisible to those derived classes. Solution? Declare the member as *Protected*. This access modifier does exactly

what our situation needs: It makes the base member visible to its derived classes, but not visible to any classes that are not in the inheritance hierarchy.

To illustrate the use of this access modifier, consider the payroll project again. The base class exposes two methods: WithHolding and NetPay. They are to be called from other modules; however, as the remarks at the end of the example indicate, a better design would be that withholdings and net pay are computed automatically in the base class and the results made available to the other modules as read-only Public properties. These methods should not be declared as Public, but if these methods are called by the derived classes, all derived classes will end up with a lot of duplicate code with each other. A better design will be for the derived classes to pass the required payroll data to the base class and have the base class perform the computations in this central location. In this way, the base class also provides more services to the derived classes, and renders more uniformity to the modules using these payroll classes. With these considerations, you will redesign the code in the base class as follows:

- Withholding and NetPay will be exposed as properties (rather than as methods in the original design).
- There should be a procedure for the derived classes to use to set the payroll data. This procedure will be named SetPayData and declared as Protected because it will be used exclusively by the derived classes.
- The methods to compute Withholding and NetPay will be renamed as ComputeWithholding and ComputeNetPay. These methods should be only used to compute on the data made available by the SetPayData method, and will be declared as Private.

Because gross pay is also passed to this class, the value can also be made as a read-only property in the base class so that each derived class will not have to set up a gross pay property. The modified base Employee class appears as follows:

```
Public Class Employee
    Private mGrossPay As Double
    Private mNetPay As Double
    Private mGrossEarnings As Double
    Private mTips As Double
    Private mWithholding As Double
    Private mSSN As Integer

    Protected Sub SetPayData(ByVal pGrossPay As Double, Optional ByVal pTips As
      Double = 0)
        mGrossPay = pGrossPay
        mTips = pTips
        mGrossEarnings = mGrossPay + mTips
        ComputeWithholding()
        ComputeNetPay()
    End Sub

    Private Sub ComputeWithholding()
        mWithholding = 0.15 * mGrossEarnings
    End Sub
    Private Sub ComputeNetPay()
        mNetPay = mGrossPay - mWithholding
    End Sub

    Public ReadOnly Property NetPay() As Double
        Get
            Return (mNetPay)
        End Get
```

```
    End Property

    Public ReadOnly Property Withholding() As Double
        Get
            Return (mWithholding)
        End Get
    End Property

    Public ReadOnly Property GrossPay() As Double
        Get
            Return (mGrossPay)
        End Get
    End Property

    Public ReadOnly Property GrossEarnings() As Double
        Get
            Return (mGrossEarnings)
        End Get
    End Property
End Class
```

The Protected SetPayData method allows the derived classes (only) to set gross pay and tips amounts and keeps a private copy of each. The method then computes the gross earnings and calls the Private subs: ComputeWithholding and ComputeNetPay to perform additional computations. In these subs, a private copy of the withholding and net pay amount is also retained to serve as the return value for the public read-only properties, Withholding and NetPay, respectively.

The derived classes will simply take the basic pay data such as hours worked, wage rates, and tips, compute the gross pay, and pass the pay data to the SetPayData sub in the base class. The Tipped class still needs to provide the Public Tips property. The modified Salaried class appears as follows:

```
Public Class SalariedPerson
    Inherits Employee
    Public Function ComputeGrossPay(ByVal pGrossPay As Double) As Double
        SetPayData(pGrossPay)
        Return (pGrossPay)
    End Function
End Class
```

Notice that the previous gross pay method is now renamed as ComputeGrossPay to avoid the name conflict with the GrossPay property in the base class. Using the Shadowing feature will make the GrossPay property in the base class unavailable to other modules that use this derived class. As you can see, all it needs to do is to pass the gross pay data through the SetPayData method to the Employee class. All the computations and properties are handled in the base class. The modified TippedEarner class appears as follows:

```
Public Class TippedEarner
    Inherits Employee
    Dim mGrossPay As Double
    Dim mTips As Double

    Public Function ComputeGrossPay(ByVal HoursWorked As Double, ByVal WageRate As
      Double) As Double
        mGrossPay = HoursWorked * WageRate
        SetPayData(mGrossPay, mTips)
        Return (mGrossPay)
    End Function

    Public Property Tips() As Double
```

```
        Get
            Return (mTips)
        End Get
        Set(ByVal Value As Double)
            mTips = Value
            SetPayData(mGrossPay, mTips)
        End Set
    End Property
End Class
```

The ComputeGrossPay method computes gross pay using the hours worked and wage rate parameters. A private copy of the gross pay is retained. The Tips property is a read-write property. When it is set, and when the gross pay is computed, both the gross pay and the tips (private copy of both) are passed to the base class through the SetPayData method. This is necessary because the order in which either value is given first in the module using this class is undeterminable and should not be dictated.

These modified classes are much easier to use. All you have to do is to create an instance and provide the basic pay data. The resulting gross pay, gross earnings, withholdings, and net pay are automatically computed without any method calls. The following code shows how you may use these classes in a procedure in another module, such as a form:

```
    Dim Salaried As New SalariedPerson()
    Dim Tipped As New TippedEarner()
    Salaried.ComputeGrossPay(1000) 'Set the salary
    'And you can see the net pay and withholding
     Console.WriteLine("Salaried gross pay is " & Salaried.GrossEarnings & _
      " and gross earning is " & Salaried.GrossEarnings)
    Console.WriteLine("Salaried has a W/H of " & Salaried.Withholding & _
      " and net pay of " & Salaried.NETPay)
    'Report tips
    Tipped.Tips = 500
    'Provide hours worked and wage rate
    Tipped.ComputeGrossPay(60, 4)
    'And you can see the gross earnings, withholdings, and net pay
     Console.WriteLine("Tipped gross pay is " & Tipped.GrossPay & _
       " and gross earnings of " & Tipped.GrossEarnings)
    Console.WriteLine("Tipped has a W/H of " & Tipped.Withholding & _
      " and net pay of " & Tipped.NETPay)
```

The MustInherit and NotIneritable Classes

The base class must be *inheritable* to allow inheritance. By default, classes are inheritable. If you have created a class that you believe should not be extended, you can mark it as *NotInheritable*. For example, the VS 2008 Math class is marked NotInheritable. On the other hand, some classes cannot be used alone, and must be inherited to implement certain invariant methods. These classes should be marked as *MustInherit*. The syntax to specify whether a class is NotInheritable or MustInherit is as follows:

```
[Public|Friend] [NotInheritable|MustInherit] Class Name
```

A MustInherit class is also recognized as an *abstract class*. It typically requires the derived class to implement certain invariant methods. These methods are specified in the base class but their implementation details are left to the derived classes. In the base class, these methods are specified with the *MustOverride* keyword. Methods marked as such are recognized as *abstract methods*. Abstract classes and abstract methods are typically used for polymorphism, which is the topic of Section 13.3.

13.2 Inheriting Designed Forms

When coding applications for an organization, you may encounter a situation where many forms appear similar to each other. To avoid repetitive coding and visual design, you can create a template and then copy and modify the template to create the needed new forms. In VB 2008, you can also achieve copying from the template by inheritance. This section briefly introduces this approach, which involves the following steps:

- Design a form template to be use as the base form.
- Build the project.
- Add a new class and inherits the base form. This new class becomes the derived form.
- Add additional visual elements and code to the derived form to make it a complete functional form.

You can then create and invoke the derived form in code similar to any other form.

The Bank Deposit Form: A Data Entry Form Example

To illustrate the use of inherited form at design time, consider the data entry forms for bank deposits. In most cases, the form should include fields such as transaction date, transaction number (automatically generated), the depositor number, and the deposit amount. For certificates of deposit (CD), at least two additional fields are required: maturity date and the rate of interest. The following discussion shows the steps of creating the base form with the basic entry fields, inheriting from this base form, and the additional steps to complete the design for the CD form:

1. Add a new (second) form to a new project, and name it **frmGeneralDeposit**; then design this form so that it appears as shown in Figure 13-1. The controls that can be involved in code are listed in the following table:

Control	Name	Use or comment
Masked text box	mskDate	Date of transaction
Text box	txtTransactionNo	Transaction number
Text box	txtDepositorNo	To identify the depositor
Text Box	txtAmount	Deposit amount
Button	btnSave	For the user to invoke data saving operation
Button	btnReturn	To return to the calling form

   Also place the following code in the code window:

```
Private Sub btnReturn_Click(ByVal sender As System.Object, ByVal e As
  System.EventArgs) Handles btnReturn.Click
    Me.Close()
End Sub
```

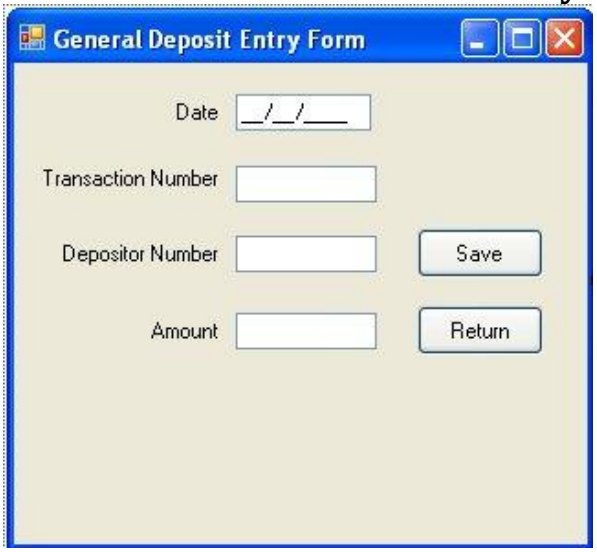
2. Press the F5 key to run and build the project. End the project.
3. Create a derived form (class) from this base form.
   a. Click the Project menu and then select Add Class (not form). The familiar Add New Item dialog box will appear.
   b. Enter frmCDDeposit as the class name, and click the Add button in the dialog box.
   c. In the frmCDDeposit code window, add the following code:

```
    Inherits frmGeneralDeposit
```

As soon as you enter the code, you should notice that the icon in the Solution Explorer for this class changes to a form icon (instead of the original class icon). You should also notice that this class actually has a form.

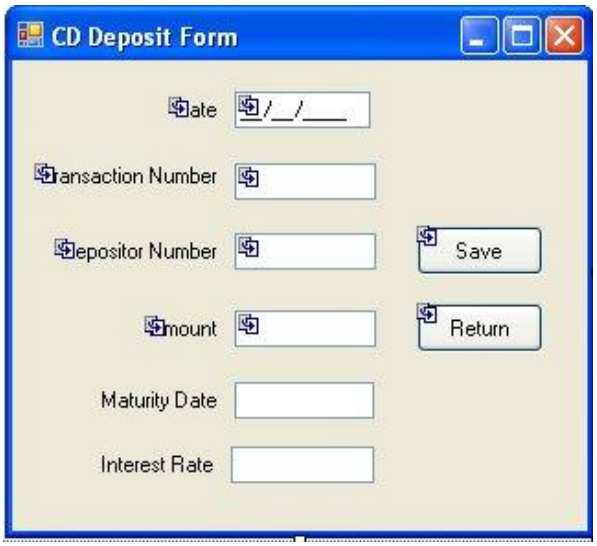
4. Design and edit the derived form. Double-click frmCDDeposit in the Solution Explorer so that the form appears. Add two labels, one masked text box and one text box on the form so that it will appear as shown in Figure 13.2. Name the text boxes mskMaturityDate and txtInterestRate.

Figure 13-1
Visual interface for the data entry base form



This form can be used to enter deposits for checking and money market account deposits. It can also be inherited and then modified for CD deposits.

Figure 13-2
The modified derived form



This form inherits from the one in Figure 13-1. Notice those inherited controls are marked as read-only. No change can be made to them. You can, however, add more controls to the form as well as modify the form itself. For example, the form's text has been changed to "CD Deposit Form."

Notice that all controls drawn on the base form are marked as read-only, meaning that their appearance or properties cannot be changed. If you find you need to change their tab order or any property, you will need to change them in the base form, not in the derived form. Notice also that

the designer automatically places some code in a Sub named InitializeComponent after you draw the additional controls on the derived form. These statements are needed to include these controls on the form at runtime. This Sub is not automatically executed; therefore, you need to also enter the following code in the code window of this derived form:

```
Sub New()
    InitializeComponent()
End Sub
```

The Sub New procedure is automatically executed when a new object is created from the class (frmCDDeposit). Additional explanations for the use of Sub New are given in Section 13.4.

To test this form, you can go back to the startup form, which is blank (in a real application, it should be the form containing the main menu). Add a button, and name it **btnShow**; then add the following code:

```
Private Sub btnShow_Click(ByVal sender As System.Object, ByVal e As
  System.EventArgs) Handles btnShow.Click
    frmCDDeposit.Show()
End Sub
```

Run the project, and click the Show button. The sales data entry form should appear. Verify the following:

- The title bar of the form shows "CD Deposit Form." This verifies that the modification you made in the derived form is recognized.
- All the controls in the derived form appear. This verifies that the form invoked is indeed the derived from and that the InializeComponent sub has been properly called. You may also want to comment out the statements for Sub New in the derived form to verify that those controls drawn on the derived form will not appear at runtime without invoking the InitializeComponent sub.
- Click the Return button in the CD Deposit form. The form should close. This verifies that the event and event handler in the base form are also recognized in the derived form (as all base and derived classes should behave). This also suggests that the base form can be used to implement all functionality common to its derived forms for efficient implementation.

13.3 Polymorphism

Polymorphism refers to the capability to define identically named methods or properties in multiple classes. Although these members have different implementations, the client code can use them at run time interchangeably without having to deal with the differences in class or implementation. Polymorphism can be inheritance-based or interface-based. *Inheritance-based polymorphism* involves defining the methods or properties in the base class and requiring the derived class to implement these members. *Interface-based polymorphism* involves defining the methods or properties in an interface definition and allowing these members to be implemented in various classes.

Inheritance-Based Polymorphism

Consider inheritance-based polymorphism. As discussed previously, this type of polymorphism is defined through abstract classes that contain abstract methods or properties. To illustrate, consider the case of general ledger accounting. There are two types of accounts: real accounts and nominal accounts. Real accounts have substance and represent the company's assets, liabilities, or owners' equity. Nominal accounts are revenues or expenses that track the changes to owners' equity. At end of each accounting period, the two types of accounts *close* differently. Net changes in the nominal account are closed to an account called *income summary*. In addition, the amounts in the account (balance and changes) are

cleared (set to zero). On the other hand, net changes in the real account are added to the beginning balance, which is carried forward to the next period as the beginning balance. While you can implement various details for all accounts the same way for both real and nominal accounts, you must have different implementations of the *Close* method depending on whether the account is real or nominal. In such a case, you can create a base Account class that provides common properties and methods for both types but defines the Close method in abstract. You can then create two Classes—RealAccount and NominalAccount—derived from the base class to implement the Close method.
The following provides additional implementation details about these classes:
The Account class should have the following properties and methods:

- Properties:
    - AccountNumber
    - AccountName
    - BeginningBalance—read only; the value is updated by the Close method explained later in this section
    - DebitTotal—representing all debit amounts posted to the account; a debit amount is represented with a positive value; read only, computed by the Post method explained later in this section
    - CreditTotal—representing all credit amounts posted to the account; a credit amount is represented with a negative value; read only, computed by the Post method explained later in this section
    - Balance—read only; representing the balance of the account; updated by the Post and Close methods explained later in this section
- Methods:
    - Post—takes an amount as parameter to post to the account; a positive value is added to the private copy of DebitTotal while a negative value is added to the private copy of CreditTotal; this method also accumulates balance for the account internally
    - Close—takes the income summary account as its parameter and requires different implementations by this class' derived class

The NominalAccount class should implement the Close method as follows:

- Its balance is posted to the income summary account, using the Post method
- All amount fields such as beginning balance, debit total, credit total, and balance are set to zero

The RealAccount class should implement the Close method as follows:

- Its beginning balance is set to the current balance, which has been updated by the Post method, to carry forward to the next accounting period
- Its debit total and credit total are set to zero

Coding the Account Class

As mentioned earlier in this subsection, this base class should be an abstract class; therefore its header must be marked as MustInherit. The class definition appears as follows:

```
Public MustInherit Class Account
```

In addition, its abstract Close method must be defined with the MustOverride keyword and should appear as follows:

```
MustOverride Sub Close(ByVal IncomeSummary As Account)
```

Finally, those internal variables associated with amounts such as beginning balance, debit total, credit total, and balance must be accessible to the derived classes and therefore should be declared as Protected, instead of the typical Private access. The complete code for this class appears as follows:

```
Public MustInherit Class Account
    Private mAccountNumber As Integer
    Private mAccountName As String
    'The following variables are declared as Protected
    'so that this class' derived classes can access
    Protected mBeginningBalance As Double
    Protected mDebitTotal As Double
    Protected mCreditTotal As Double
    Protected mBalance As Double

    Public Property AccountNumber() As Integer
        Get
            Return (mAccountNumber)
        End Get
        Set(ByVal Value As Integer)
            mAccountNumber = Value
        End Set
    End Property

    Public Property AccountName() As String
        Get
            Return (mAccountName)
        End Get
        Set(ByVal Value As String)
            mAccountName = Value
        End Set
    End Property

    Public ReadOnly Property BeginningBalance() As Double
        Get
            Return (mBeginningBalance)
        End Get
    End Property

    Public ReadOnly Property DebitTotal() As Double
        Get
            Return (mDebitTotal)
        End Get
    End Property

    Public ReadOnly Property CreditTotal() As Double
        Get
            Return (mCreditTotal)
        End Get
    End Property

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

    Public Function Post(ByVal Amount As Double) As Double
```

```
        If Amount >= 0 Then
            mDebitTotal += Amount
        Else
            mCreditTotal += Amount
        End If
        mBalance = mBeginningBalance + mDebitTotal + mCreditTotal
        Return (mBalance)
    End Function

    MustOverride Sub Close(ByVal IncomeSummary As Account)
End Class
```

The code for property definitions should be self-explanatory. Notice how the Post method is coded. If the parameter is a positive amount, it is added to DebitTotal; otherwise, it is added to CreditTotal. The method also computes (updates) the new balance, which is then returned to the caller. (Perhaps, you are wondering why not just combine the positive and negative amounts together. After all, the balance will be still the same, but the accountant believes that separating the debit total from the credit total enhances the audit trail.)

Implementing the Close Method for the RealAccount class

Recall that the Close method of this derived class takes its current balance (updated by the Post method) as its beginning balance and clears the amounts representing changes in the current period (debit total and credit total). The code appears as follows:

```
Public Class RealAccount
    Inherits Account
    Overrides Sub Close(ByVal IncomeSummary As Account)
        'Real account does not have to be closed to income summary
        'Just update the beginning balance
        mBeginningBalance = mBalance
        mDebitTotal = 0
        mCreditTotal = 0
    End Sub
End Class
```

Pay particular attention to the header for the Close method. The *Overrides keyword* must be specified because the method is defined as MustOverride in the base class. Notice that although the IncomeSummary account is passed to this method by definition, it is not used in the computation for the real account. Also notice how the Protected variables in the base class are referenced. They are referenced in exactly the same way as if they were declared in this class. Of course, they can also be qualified by the MyBase keyword.

Implementing the Close Method for the NominalAccount Class

Recall that the Close method of this derived class requires the current balance be closed (posted) to the income summary account. In addition, all amount fields should also be cleared. The code appears as follows:

```
Public Class NominalAccount
    Inherits Account
    Overrides Sub Close(ByVal IncomeSummary As Account)
        'Nominal account is closed by adding its balance to IncomeSummary
        'and clearing all amounts
        IncomeSummary.Post(mBalance)  'Post balance to income summary account
```

```
        'then set all pertinent amounts to zero
        mBeginningBalance = 0
        mDebitTotal = 0
        mCreditTotal = 0
        mBalance = 0
    End Sub
End Class
```

Using Polymorphism

As mentioned earlier in this section, polymorphism allows the client code to use the same methods or properties without regards to the differences in implementation between classes. In the preceding example, any object of the Account class (real or nominal) can be closed with the Close method at run time without encountering any problem, and the results will be correct. Before illustrating how polymorphism makes coding easier, it is necessary to introduce the CollectionBase class, which you need to use in the demonstration.

The CollectionBase Class

The *CollectionBase* class provides a way to create a collection of your own objects. It allows you by inheritance to create a collection of a particular type, such as the Account class in the preceding sub-section. A class derived from the CollectionBase must implement the Add and Remove methods, usually through the class' *List property*, which provides the Add and RemoveAt method. To allow access (reference) to a particular item in the collection, you should also implement the Item property.
In this application, a collection of accounts will be used to demonstrate the usefulness of polymorphism. You will create an AccountCollection class that can accept and return objects of the Account type for this purpose. The following code shows how this class is created by inheritance of the CollectionBase:

```
Public Class AccountCollection
    Inherits CollectionBase
    Default Public ReadOnly Property Item(ByVal Index As Integer) As Account
        Get
            Return (List.Item(Index))
        End Get
    End Property

    Sub Add(ByVal TheAccount As Account)
        List.Add(TheAccount)
    End Sub

    Sub Remove(ByVal Index As Integer)
        list.RemoveAt(Index)
    End Sub
End Class
```

As mentioned, the Add and Remove methods are implemented with the base class' *List.Add* and *List.RemoveAt* methods; the read-only Item property is implemented using the *List.Item property*. Pay close attention to the header of the Item property definition. The *Default keyword* marks that this property is the default for this class/object. The property takes an Index parameter——that indicates which element to return. A member can be marked as the default only when it requires a parameter. A default can be omitted from the reference as will be demonstrated later. Notice also that the Item property is declared As Account; therefore, items to be added and retrieved from the collection must be of the Account type.

Look It Up

For a walkthrough example of how to create a collection for a particular class, use the keyword Collection properties to search the search tab (not the index tab); then double-click the title "Walkthrough: Creating Your Own Collection Class" to read the page. It shows how the CollectionBase class can be used to create a collection class for the widget (defined in the example) class.

Testing the Classes

You are now ready to witness polymorphism in action in a form. For simplicity, assume that the general ledger system has four normal accounts: Cash, Payable, Sales, and Expense. The first two are real accounts, and the last two are nominal. In addition, you will have the IncomeSummary account (a real account) for closing purpose. You will create these accounts according to their types, assign their AccountName properties, and post some amounts to each account. You also create an Accounts object of the AccountCollection type to keep the first four accounts in this collection, which can be employed to demonstrate the use of the Close method. Assume you have a button named btnTest in the form. The code appears as follows:

```
Private Sub btnTest_Click(ByVal sender As System.Object, ByVal e As
  System.EventArgs) Handles btnTest.Click
    Dim Cash As New RealAccount()
    Dim Payable As New RealAccount()
    Dim Sales As New NominalAccount()
    Dim Expense As New NominalAccount()
    Dim IncomeSummary As New RealAccount()
    Dim Accounts As New AccountCollection()
    Dim I As Integer

    With Accounts
        .Add(Cash)
        .Add(Payable)
        .Add(Sales)
        .Add(Expense)
    End With

    Cash.AccountName = "Cash"
    Cash.Post(3000)
    Payable.AccountName = "A/P"
    Payable.Post(-1000)
    Sales.AccountName = "Sales revenue"
    Sales.Post(-9000)
    Expense.AccountName = "Various Expenses"
    Expense.Post(2000)
    IncomeSummary.AccountName = "Income Summary"

    ' Use the Close method regardless of the account type
    For I = 0 To Accounts.Count - 1
        Accounts(I).Close(IncomeSummary)
    Next I

    For I = 0 To Accounts.Count - 1
        Console.WriteLine(Accounts(I).AccountName & " " & _
      Accounts(I).BeginningBalance)
    Next I
```

```
    Console.WriteLine(IncomeSummary.AccountName & " " & _
      IncomeSummary.BeginningBalance)
    Console.WriteLine(IncomeSummary.AccountName & " " & IncomeSummary.DebitTotal)
    Console.WriteLine(IncomeSummary.AccountName & " " & IncomeSummary.CreditTotal)
    Console.WriteLine(IncomeSummary.AccountName & " " & IncomeSummary.Balance)
End Sub
```

In the code, the four accounts are added to the Accounts collection before each account's AccountName property is assigned and the amount is posted. The first For loop uses the Close method of each account in the Accounts collection to perform the closing. The second For loop writes out the resulting BeginningBalance of each account to verify the accuracy of the results. (You should see 3000, -1000, 0, and 0 because the first two accounts are real and the last two are nominal.) The last four statements display the effect of closing on the amounts in the IncomeSummary account. (You should see 0, 2000, -9000, -7000; indicating a 7000 credit balance.) Notice that the Accounts collection does not differentiate between real and nominal accounts, and uses the Close method indiscriminately. The results, however, are correct.

Have you noticed the effect of the Default keyword on this code? Without the default specification, you will have to reference each account in the Accounts collection as:

```
    Accounts.Item(I)
```

With the default specification, however, you are able to reference to each account as:

```
    Accounts(I)
```

Notice that the CollectionBase class is zero-based (the first element has an index value of zero).

Again, Why Inheritance-Based Polymorphism?

In the preceding example, if you do not use inheritance, you can still accomplish correct "Close" by the use of If blocks in the Account class: one block will handle the nominal account; the other will handle the real account. So, what is the advantage of using inheritance to deal with different type of closing? The answer lies in that many other programming problems are open-ended. At the time you are developing the base class, you may not be aware of all possible scenarios to incorporate in the If (or Select Case) blocks. As each new scenario is identified, you will have to go back and revise the class. On the other hand, using the inheritance-base polymorphism, there is no need to revise the base class. All you have to do is to develop another class derived from the base class to take care of the new scenario.

Interface-Based Polymorphism

In interface-based polymorphism, the interface defines the properties, methods, and/or events to be implemented similar to the definition of abstract members in a class. The interface, however, is not allowed to provide any implementation. (In contrast, recall that the base Account class in the previous example provides various implementations in addition to defining the abstract Close method.) The definitions in the interface represent a contract. Any class that implements the interface must implement every aspect of that interface exactly as defined. As a contract, after being published (the definition is made known and available to the software development team), the interface definition should never be changed. Any change to the definition will break the existing code that implements the particular interface.

Defining the Interface

The syntax to define the interface appears as follows:

```
[Public|Friend|Private|Protected] Interface Name
```

```
    'Interface definition statements
End Interface
```
Inside the Interface block, you can define any methods (subs or functions), properties, and or events by giving their header definitions. The following represents a valid interface definition:
```
Public Interface ITest
    Sub sub1(ByVal Parm1 As Integer)
    Function Fun(ByVal Gamer As String) As String
    ReadOnly Property Item(ByVal Index As Integer) As Object
    Event SomethingOccurred(ByVal sender As System.Object, ByVal e As EventArgs)
End Interface
```
You can also use the Inherits statements to inherit another interface (not class). In that case, the Inherits statement must be the first statement in the block. An interface can inherit more than one base interface. Note that inside the block, the only allowed keywords are Overloads, Default, ReadOnly, and WriteOnly. No other modifier, such as Public, Private, Protected, Shared, Static, Overrides, MustOverride, or Overridable is allowed. The interface itself, as defined in the first line, is Friend by default and can be explicitly declared Public, Friend, Private or Protected.

Placement of the Interface Definition

The interface can be defined in a class module or a standard module. When an interface has the Private or Protected modifier, it must be placed in a class; therefore, it must be in the class module in which the interface belongs. When an interface has no modifier or has the Public or Friend access modifier, it can be placed inside or outside of the class or module definition. That is, any of the following placements is allowed:

- Inside a class definition:
```
Public Class ClassName
    Public|Friend|Private|Protected] Interface InterfaceName
        'Other statements
    End Interface
    'Other statements
End Class
```
- Outside any class definition in class module:
```
Public Class ClassName
    'Other statements
End Class
[Public|Friend] Interface InterfaceName
    'Other statements
End Interface
```
- Inside a module definition:
```
Public Module ModuleName
    [Public|Friend|Private|Protected] Interface InterfaceName
   'Other statements
    End Interface
    'Other statements
End Module
```
- Outside any module definition in standard module:
```
Public Module ModuleName
    'Other statements
End Module
[Public|Friend] Interface InterfaceName
    'Other statements
End Interface
```

When the interface has either Public or Friend access, it is more appropriate to place the definition in a standard module because the interface cannot be instantiated and is meant to be implemented by a diverse group of classes. Placing the definition in a standard module makes it easily available to these classes, while placing it in a class module does not offer any obvious advantage.

Implementing an Interface

To implement an interface in a class, you use the *Implements keyword.* For example, to implement the ITest interface defined in the preceding example in a class named Class1, you will code:

```
Public Class Class1
    Implements ITest
    'Other statements
End Class
```

In addition, for each method, property, or event that you implement, you will also need to add an Implements clause at the end of the header to indicate which member you are implementing. For example, to implement Fun, the header of the method should appear:

```
Function Fun(ByVal Gamer As String) As String Implements ITest.Fun
```

Notice the Implements clause at the end of the header. The method to be implemented is expressed in the format *Interface.Method*.

Recall that everything defined in the interface must be implemented when you decide to implement an interface; therefore, in Class1, you will have to provide code to implement Sub1, Fun, Item, and SomethingOccurred after you code Implements ITest.

Tip

Although you can use a different name when implementing a method or property of an interface, it is advisable to stay with the same name. After all, the name should be considered an aspect of the interface definition.

An IMove Example

To illustrate how an interface is actually implemented, consider a simple example. In the versions of VB prior to VB.NET, each control has a Move method; however, in the newer versions, this method is replaced with the SetBounds method because Move is now an event in the control. Suppose, however, you still want to define a Move method as a standard interface that you can choose to implement for any control of interest. How do you proceed?

You can first define an interface as follows (place the code in a standard module):

```
Public Interface IMove
    Sub Move(ByVal Left As Integer, Optional ByVal Top As Integer = -1, Optional
      ByVal Width As Integer = -1, Optional ByVal Height As Integer = -1)
End Interface
```

Note the difference between the definition of Move and the SetBounds method. The SetBounds method requires all four parameters. On the other hand, the Move method requires only the first parameter although it takes up to four. The optional parameters will be replaced with their current values if omitted. Suppose you now want to implement the interface for the label and call the new label MyLabel. You can code the class as follows:

```
Public Class MyLabel
    Inherits Label
    Implements IMove
    Shadows Sub Move(ByVal Left As Integer, Optional ByVal Top As Integer = -1, _
```

```
    Optional ByVal Width As Integer = -1, Optional ByVal Height As Integer = -1) _
      Implements IMove.Move
        If Top = -1 Then
            Top = Me.Top
        End If
        If Width = -1 Then
            Width = Me.Width
        End If
        If Height = -1 Then
            Height = Me.Height
        End If
        Me.SetBounds(Left, Top, Width, Height)
    End Sub
End Class
```

Within the class definition, the Inherits statement must be the first statement. The Implements statement must immediately follow the Inherits statement. Other property definitions or procedures can then follow. (Note that before you completely code the implementation of the interface definition, the Implements statement will be underlined as if there were an error in the statement.) Notice that you place the *Shadows keyword* at the beginning of the Sub Move header because Move is an event in the Label class. The Shadows keyword allows you to define the identifier in a different way. In the code, if you need to refer to the Move event in the base Label class, you will need to qualify it with the *MyBase keyword* as: MyBase.Move.

The remaining code should be self-explanatory. Any omitted parameter is replaced with the original value as defined in the parameter lists. After these parameter values are ascertained, the SetBounds method is called to move the label to the new position.

Testing the IMove Implementation

How does the new MyLabel control work? You can test it on a new form. Declare and create a control of the MyLabel type, and make it appear on the form. To test the Move method, you will make it move horizontally to a random position when it is clicked. The test code appears as follows:

```
Dim WithEvents lblMover As New MyLabel()
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs)
  Handles MyBase.Load
    With lblMover
        .AutoSize = True
        .Text = "Test Move"
        .BorderStyle = System.Windows.Forms.BorderStyle.Fixed3D
        .Location = New Point(Me.Width \ 2, Me.Height \ 2)
    End With
    Me.Controls.Add(lblMover)
End Sub

Private Sub lblMover_Click(ByVal sender As System.Object, ByVal e As
  System.EventArgs) Handles lblMover.Click
    Dim Rand As New Random()
    Dim L As Integer
    L = Rand.Next(0, Me.Width - lblMover.Width)
    lblMover.Move(L)
End Sub
```

The new label is declared WithEvents so that it can recognize the Click event. It is created and associated with the lblMover variable. The form Load procedure defines the appearance of the new label, placing the label with its Left and Top aligned at the center of the form. Note that the control must

be added to the form's Controls collection using the collection's Add method for it to appear on the form.

The label's Click event procedure actually tests how the Move method works. The random number generator object, Random, is used to generate an integer number in the range of 0 and the difference between the form's width and the label's width. This value is passed to the new label's Move method, omitting the three optional parameters. When you run the project and then click the label, you should be able to verify that it moves to a random location at the same horizontal position. The formula to generate the random number ensures that the entire label appears on the form.

Why Interface-Based Polymorphism?

Perhaps, you wonder why go through all the troubles of defining the interface and then implementing it in classes. After all, if you just code the classes with the properties and methods defined in an interface without explicitly implementing the interface, these classes will still behave the same way as if the interface had been implemented. So, why is there a need for the interface, and why the implements? Keep in mind that classes can be developed by different individuals in an organization, and interfaces can be defined by other people. The published interface offers a way to standardize functionality of various methods and properties across classes regardless of the developers involved—even when the detailed implementations are different among the classes. Standardized interfaces make it easier to use the classes because developers, as users of the classes, will not be required to differentiate different implementations among various classes. Imagine the Move method is implemented in different controls by different developers, each with different order of the parameters such as height being the first parameter. The resulting confusion will most likely discourage most programmers from using the Move method of any control.

If there is already the inheritance-based polymorphism, however, is there truly a need for the interface-based polymorphism? Actually, each is suitable for a different situation. The inheritance-based polymorphism allows implementations in the base class, and is suitable for classes that are homogeneous and interrelated. The base class and the derived class should have an "is a" relationship—that is, the derived class "is a" base class such as a nominal account is a ledger account. The interface-based polymorphism is suitable for diverse classes that have no apparent relationships among themselves.

Multiple Inheritances

Can a class inherit from more than a base class? No. Such an inheritance is not allowed; however, a class is allowed to inherit from one base class and implements many interfaces. Also, an interface is allowed to inherit from multiple interfaces. One potential problem with the multiple-interface inheritance situation is the name conflict. Several interfaces may define elements that conflict with each other. The inheritance is still allowed in the situation. Some of the Explore and Discover exercises at the end of this chapter explore this issue.

13.4 Additional Topics

This section presents two sundry and yet important topics: garbage collection and user-defined exceptions. Understanding the garbage collection process enables you to manage system resources more efficiently; knowing how to create user-defined exceptions allows you to create exceptions that can be more meaningful to your program users.

Garbage Collection and the Finalizer

When an instance of a class is initialized, the Sub New method is automatically called . But what happens when the instance goes out of scope, or is set to Nothing? The system may not immediately reclaim the system resources used by the object, but wait until it determines that the resources have run low. The process of destroying those dead objects and reclaiming their associated resources is recognized as *garbage collection*. In this process, the object's Finalize method is called. Recall that all classes are based on the Object class. This class provides a Finalize method, so if your class does not provide a Finalize method, the method in the Object class will be called. If you want to perform any finishing touch when an instance of your class is being destroyed, be sure to use the following template:

```
Protected Overrides Sub Finalize()
    ' Place your finishing touch statements here
    MyBase.Finalize()
End Sub
```

The method should have the Protected access; that is, it should not be called from any client code. The method also overrides the same in the base class, and thus must call the base class' finalizer, using the MyBase qualifier, as shown in the code template. If the statement is not there, the Finalize method in the base class will not be executed. Resource leaks can result. Note also that calling the base class' Finalize method should be the last statement in the code because you want to finish all businesses in the current class before chaining execution of the finalizer up in the inheritance hierarchy. (Your code calls its base class' Finalize method, which in turns calls its base class' Finalize method and so forth.)

Because the Finalize method cannot be called by the client code, and the garbage collection process does not start immediately after an object is no longer accessible, you may find a need to provide a way for the client code to initiate destruction of the object explicitly. If your class is written and compiled as a component, the *Dispose method* is typically the method for this purpose.

Creating User Defined Exceptions

Chapter 12 showed you how to throw an exception that is already defined; however, you may encounter conditions in which those defined exceptions may not describe exactly the types of the exceptions that you want your class to throw. You may find it more desirable to throw an exception that more precisely indicates the nature of the problem. For example, in the FixedAsset example, you may want the system to show DepreMethodException instead of ArgumentOutOfRangeException so that the programmer can more easily identify the coding error. In such a case, you can define your own exception. To do so, you need to be aware of the exception hierarchy.

The base class for all exceptions is *System.Exception* from which *SystemException* and *ApplicationException* are derived. The exceptions found in the Exceptions option of the Debug menu are all derived from SystemException. Microsoft recommends that the user-defined exception be derived from ApplicationException. This exception class has three overloaded constructors defined as follows:

```
Sub New()
Sub New(Message As String)
Sub New(Message As String, InnerException As System.Exception)
```

In the third overloaded constructor, the *InnerException* refers to an exception that is caught in a Catch block and is passed to the caller, allowing the code to determine the chain of exceptions.
With this understanding, you can create a DepreMethodException class as follows:

```
Public Class DepreMethodException
    Inherits ApplicationException
    Sub New()
        MyBase.New("Depreciation method setting not recognizable")
    End Sub
```

```
    Sub New(ByVal Message As String)
        MyBase.New(Message)
    End Sub
    Sub New(ByVal Message As String, ByVal InnerException As System.Exception)
        MyBase.New(Message, InnerException)
    End Sub
End Class
```

The default constructor (the one with no parameter) will give the message, "Depreciation method setting not recognizable," so that the user will realize that the DepreMethod property is given a bad setting. The exception throwing statement in Section 12.3 (in the Set procedure for the DepreMethod property) can then be coded as:

```
    Throw New DepreMethodException()
```

Of course, you can also incorporate additional information, using the second overloaded constructor. For example, you can also include the Value parameter as follows:

```
    Throw New DepreMethodException(Value & " is not an allowed setting for
DepreMethod")
```

Summary

- The three most significant features of object-oriented programming are encapsulation, inheritance, and polymorphism.
- Inheritance allows the creation of a new class from an existing (base) class. The new class inherits all the properties, methods, and events from the base class without having to deal with the source code of the base class. The derived class is the sub class and the base class is the super class.
- All classes are either directly or indirectly derived from the System.Object class.
- A new class can be derived not only from a class created in a project, but also from those existing in the .NET framework. All VB controls and the form are inheritable.
- The Employee, SalariedPerson, and TippedEarner classes were used to show how the latter two classes can be derived from the first class.
- The MaskEdit class was used to show how the TextBox class can be inherited to provide the functionality for a newly created Mask property.
- If a member (property or method) of a derived class has the same name as a member of the base class, the member in the derived class must be declared with the Shadows keyword. The member in the derived class is referenced as Me.*MemberName* or just the member name, while the member in the base class is referenced (in the derived class) as MyBase.*MemberName*.
- It is recommended that all events contain two parameters: sender and event arguments. The event arguments parameter should be derived from the System.EventArgs class. The SortEngineEventArgs class was used to show how to create a class for the purpose of passing event arguments.
- To allow a member of the base class to be accessed by only its derived classes, declare the member with the Protected access modifier. The modified Employee, SalariedPerson, and TippedEarner classes showed how the declaration of the Protected access for a member can facilitate implementation of desired features in these classes.
- A form with controls and code can be inherited at design time. The derived form can be further added with other controls and code at design time. The general deposit form and CD

deposit form (class) were used to illustrate the steps to accomplish the inheritance and additional modification at design time.

- Polymorphism refers to the capability to define identically named methods or properties in multiple classes. Although these members have different implementations, the client code can use them at run time interchangeably without having to deal with the differences in class or implementation.
- There are two types of polymorphisms: inheritance-based and interface-based.
- Inheritance-based polymorphism involves defining the methods or properties in the base class and requiring the derived class to implement these members. The Account, RealAccount, and NominalAccount classes were used to illustrate inheritance-based polymorphism through the Close method.
- Interface-based polymorphism entails defining the methods or properties in an interface definition and allowing these members to be implemented in various classes. The IMove interface example was used to illustrate how to define and implement interfaces.
- Interface-based polymorphism ensures that classes implementing the interface provide standardized method (or property) calls, making these classes easier to use because of their uniformity.
- A class is not allowed to inherit from multiple base classes, but an interface is allowed to inherit from multiple interfaces.
- When an object is no longer accessible, the resources it used may not be immediately reclaimed by the system. Usually, the garbage collection process does not start until the system determines that the resources are running low. The garbage collector calls the Finalize method of each object before destroying and reclaiming resources used by the object.
- When creating your own exception, derive the new exception from the ApplicationException class.

Explore and Discover

13-1. Implicit Inheritance? The text claims that all objects are derived from the System.Object class. To see what this means, add a class module to a new project and then take the default file name **Class1**. You should see the following code in the class module:

```
Public Class Class1

End Class
```

Go back to the form. In the form Load event, type the following line:

```
    Dim TheClass As New Class1()
    TheClass.
```

When you key the period (.) after TheClass, do you see the IntelliSense display "GetType?" Complete the line (TheClass.GetType()). You have not coded any property or method in the class. If it did not inherit from any base class, the IntelliSense would have nothing to show. The GetType method is a method implemented in the System.Object class. Class1 must have implicitly inherited from that class.

13-2. Static or Instance Method? (continued from exercise 13-1) Add the following statement to the same event procedure:

```
    Class1.GetType()
```

Does the compiler accept the statement? If so, the method is a Shared (static) method. If not, the method is an Instance method. What do you find? Remove the statement after your test.

13-3. Access Modifier for the Class. Use the class module in exercise 13-2. Change the access modifier Public to Friend, Protected, Private, and Shared (one at a time). Which ones does the compiler accept or reject? (Protected, Private and Shared are not allowed.)

13-4. Access Modifier for the Inner Class. Use the class module in exercise 13-3. Keep the following code in the module:

```
Public Class Class1
    Class Class2

    End Class
End Class
```

Change the access modifier for Class2 to Public, Friend, Protected, Private, and Shared (one at a time). Which ones does the compiler accept or reject? (Only Shared is not allowed.)
Change the access modifier for Class1 to Friend; then change the access modifier for Class2 to Public, Friend, Private, and Shared (one at a time). Which ones does the compiler accept or reject? (Only Shared is not allowed.) When Class2 is declared with the Public access modifier, is it really accessible to "all?" Not really because Class2's outer class is a Friend access. Class2 cannot be accessible when Class1 is not accessible.

13-5. Name Conflict in Derived Class. Add a class module to a new project and then take the default name. Code the class as follows:

```
Public Class Class1
    Function Name1() As Integer
        Return(1)
    End Sub
End Class
```

Add another class in the same module (but outside of Class1) as follows:

```
Class Class2
    Inherits Class1
    Property Name1() As Integer
        Get
            Return(0)
        End Get
        Set(ByVal Value As Integer)

        End Set
    End Property
End Class
```

What does the compiler show? You should get a warning that Name1 conflicts with the function Name1 in the base class. To take care of the warning, qualify the property with the Shadows keyword as follows:

```
    Shadows Property Name1() As Integer
```

13-6. Referencing a Shadowed Member. (continued from exercise 13-5). Suppose you are coding for Class2 in the preceding exercise. How do you reference Name1 in Class1 and in Class2?
Add the following method to Class2.

```
Sub Show()
    MsgBox(MyBase.Name1)
    MsgBox(Me.Name1)
```

```
End Sub
```
The compiler accepts both, but what does each mean? Place the following code in the form's Click event:
```
    Dim TheClass As New Class2()
    TheClass.Show()
```
Run the project and then click the form. What do you see? If you see a 1, it comes from the base class (Class1). If you see a 0, it comes from the derived class (Class2).

13-7. Name Conflict Again. (continued from exercise 13-6) Change the code in Class2 as follows:
```
Class Class2
    Inherits Class1
    Overloads Function Name1() As Integer
        Return (2)
    End Function
    Sub Show()
    MsgBox(MyBase.Name1())
        MsgBox(Name1())
    End Sub
End Class
```
Notice the keyword to declare Name1. Do you see any error? You can also overload a member in the derived class. Run the project and then click the form. Do you see any difference?

13-8. Interface Access Declaration. Create a new project and add a new class module. In the class module inside the class definition, enter the following code:
```
Interface ITest

End Interface
```
Try to place the following access Modifiers before the Interface declaration: Public, Friend, Protected, Private, Shared. Which ones does the compiler accept or reject? (Shared is not allowed.)

Now move the interface definition out of the class definition, but still in the same module. Repeat the same experiment with the access modifiers. Which ones does the compiler accept or reject? (Only Friend and Public are accepted.)

13-9. Interface Access Declaration. (continued exercise from 13-8) Modify the code in the preceding exercise as follows:
```
Interface ITest
    Function Fun() As Integer
End Interface
```
Now try to place the following access Modifiers before the function definition: Public, Friend, Protected, Private, Shared. Which ones does the compiler accept or reject? (None are allowed.)

13-10. Interface Access Implementation. (continued from exercise 13-9) Add the following code to the class module in the preceding exercise:
```
Class Taker
    Implements ITest
    Function Fun() As Integer Implements ITest.Fun

    End Function
End Class
```
Now try to place the following access Modifiers before the function header in this class: Public, Friend, Protected, Private, Shared. Which ones does the compiler accept or reject? (Only Shared in not allowed.)

13-11. Multiple Inheritances. Can a class inherit from more than one class? Try the following code in a class module:

```
Class Class1
End Class
Class Class2
End Class
Class Class3
    Inherits Class1
    Inherits Class2
End Class
```

Does the compiler accept the second Inherits statement?

13-12. Multiple Inheritances. (continued from exercise 13-11) Can a class inherit from another class, and still implement more than one interface? Add the following code to the preceding module:

```
Interface ITest1
    Function Fun1() As Integer
End Interface
Interface ITest2
    Function Fun2() As Integer
End Interface
```

Modify the code for Class3 as follows:

```
Class Class3
    Inherits Class1
    Implements ITest1
    Implements ITest2
    Function Fun1() As Integer Implements ITest1.Fun1
    End Function
    Function Fun2() As Integer Implements ITest2.Fun2
    End Function
End Class
```

Do you see any compiler error after completing the code? You can implement as many interfaces in a class as you want.

13-13. Multiple Inheritances. (continued from exercise 13-12) Can an interface inherit from more than one interface? Add the following code to the preceding exercise.

```
Interface ITest3
    Inherits ITest1
    Inherits ITest2
End Interface
```

Do you see any error? An interface can inherit from many interfaces.

13-14. Name Conflict in Interface Inheritance. Add a class module to a new project, and retain the default name for the class module. Add the following code to the module:

```
Interface IConflict1
    Function Conf() As Integer
End Interface
Interface IConflict2
    ReadOnly Property Conf() As Integer
End Interface
Interface Iconflict
    Inherits IConflict1
    Inherits IConflict2
End Interface
```

Does the compiler accept the code? Notice that Conf is defined as a method in IConflict1 and as a read-only property in IConflict2. There is a name conflict, but the compiler still accepts.

How do you implement IConflict in a class? Try the following code in the same module:

```
Public Class Class1
    Implements Iconflict
    Function Conf1() As Integer Implements IConflict1.Conf
    End Function
    ReadOnly Property Conf2() As Integer Implements IConflict2.Conf
        Get
        End Get
    End Property
End Class
```

It works. Notice that you cannot have both the function and the property assigned the same name. Also notice how the Implements clause references the interface. It goes back to the base interface (IConflict1 or IConflict2). Try to change one of the interface names (to IConflict) in either of the Implements clause and see what happens.

Exercises

13-15. Bank Accounts. Create a base bank account that has a read-only Balance property and account number (read-write) property. It should also have a Deposit method that adds an amount (parameter of the Double type) to the balance, and a Withdraw method that subtracts an amount (parameter of the Double type) from the balance; then create two classes derived from this base class: SavingsAccount and CheckingAccount. The SavingsAccount should have the InterestRate (Double) property and the Update method, which assumes that the update is done monthly and adds 1/12 of the interest rate times the balance to the current balance. The CheckingAccount class has two properties: minimum compensation amount (Double), and monthly service fee (Double). The default minimum balance is $100,000. The class should have an Update method, which tests current balance against the minimum compensation amount. If the former is lower, the monthly service fee is subtracted from the current balance; otherwise, nothing is changed.

13-16. Bank Accounts: Inheritance Polymorphism. (continued from exercise 13-15) Revise the classes so that the Update method is an abstract method in the base bank account class, requiring the actual implementation in both the checking account and savings account class.

13-17. The BankAccountCollection Class: Inheritance from the CollectionBase Class. (continued from exercise 13-16) Create a BankAccountCollection class that inherits from the CollectionBase class. The derived class accepts only the bank account class into the collection (with the Add method), has a read-only Item property that takes a parameter of the Integer type, and returns the account as indexed by the parameter. The class should also implement a Remove method that removes an account in the position as specified by a parameter of the Integer type.

13-18. Depositor and the Bank Account Classes (nested classes). (continued from exercise 13-17) Create a Depositor class that has three properties: Depositor ID, Name, and Accounts. The Accounts property has an Item property and two methods, Add and Remove, as described in exercise 13-17. The BankAccount, CheckingAccount, SavingsAccount, and BankAccountCollection classes should all be inner classes of the Depositor.

13-19. The MaskEdit class. Refer to the MaskEdit class example in Section 13.1. Enhance the class so that it can handle the Backspace and Left arrow keys. In addition, it should also allow A and a as the mask character. The A mask requires that an uppercase letter be in that corresponding position; the a mask requires that either an upper- or lowercase letter be in that position.

13-20. The Inventory Classes. Create an inventory base class that has the ProductID and Description properties. It has two derived classes: SpecificItem and HomogeneousItem. The SpecificItem class has an ItemID, Cost, and SalesAmount properties. Its Valuation method tests whether the SalesAmount property is zero. If so, the method returns the Cost property value as its total cost; otherwise, it returns zero. The HomogeousItem class has standard cost and quantity on hand (read-only) properties. Its Valuation method returns the product (extension) of quantity on hand times the standard cost as its total cost. It also has a Purchase method that adds a quantity (parameter) to the private copy of the quantity on hand property. It has a Sell method that subtracts a quantity (parameter) from the private copy of the quantity on hand property.

13-21. The Inventory Classes: Inheritance Polymorphism. (continued from exercise 13-20) Revise the classes so that the Valuation method is an abstract method in the base inventory class, requiring the actual implementation in both the SpecificItem class and the HomogeneousItem class.

13-22. The InventoryCollection Class: Inheritance from the CollectionBase Class. (continued from exercise 13-21) Create a InventoryCollection class that inherits from the CollectionBase class. This derived class accepts only the inventory class into the collection (with the Add method), and has a read-only Item property that takes a parameter of the Integer type and then returns the account as indexed by the parameter (make this the default property). The class should also implement a Remove method that removes an inventory in the position as specified by a parameter of the Integer type.

13-23. A Student Class with an Inner Class. (Refer to exercise 12-13 in Chapter 12.) Modify the project so that the CourseID is replaced with the Course object, which has the CourseID, ClassRoom, and Date/Time of course offering such as Monday and Wednesday 10:00-11:20 A.M. Also, the Courses should now be a property that has the Item property and the Add and Remove methods similar to the Item property in the preceding exercise (13-22).

13-24. Implementing Event Arguments. (Refer to exercise 12-15 in Chapter 12.) Implement the event with the two parameters as recommended by Microsoft. Call the second parameter DepreMethodChangedArgs.

13-25. Implementing Event Arguments. (Refer to exercise 12-18 in Chapter 12.) Add an event that will fire when the percentage of sorting oepration is increased by one percentile. The event should have the two parameters as recommended by Microsoft. Name the second argument as **PercentChangedArgs**.

Projects

Note: Exercises 13-15 through 13-18 together can be considered as one project.

13-26. The Apartment Classes (Nested). Develop an Apartment class that has two properties: ApartmentNumber and Address. This class is inherited by two classes: EfficientSuite and DeluxeSuite.

The EfficientSuite class has two properties: Bedroom and Kitchen. Each of these properties has a Furnishing property. The Furnishing property has an Item property (default) and two methods: Add and Remove. The Add method adds a string (furniture description) to the item list, while the Remove method removes an item as specified by the index parameter from the item list. The Item property is read-only and takes a parameter of the Integer type. It returns a Furniture object at the position specified by the parameter. The Furniture object has four properties: AssetID, Description, DateAcquired, and Condition Index (a value ranging from 1 to 5; 1=Excellent, 5= to be junked). The DeluxeSuite class has two properties: Bedrooms and Kitchen. The DeluxeSuite's Bedrooms and Kitchen properties each have a Furnishing property that behaves the same way as the EfficientSuite's Furnishing property. The Bedrooms property has the Item property and two methods: Add and Remove that behave in exactly the same way as those described for the EfficientSuite's Furnishing property. (*Hint:* Write the Furnishing and BedRooms properties each respectively as a class inherited from the CollectionBase class.)