CHAPTER 3

Using Classes and Objects in VB

The original versions of Visual Basic (1.0 through 3.0) did not contain much in the way of object-oriented features, and many programmers' habits were formed by the features of these early versions. However, starting with Visual Basic 4.0, you could create Class modules as well as Form modules, and use them as objects. In this chapter we'll illustrate more of the advantages of using class modules. In the following chapter we'll extend these concepts for the more fully object-oriented VB.NET.

A Simple Temperature Conversion Program

Suppose we wanted to write a visual program to convert temperatures between the Celsius and Fahrenheit temperature scales. You may remember that water freezes at 0° on the Celsius scale and boils at 100°, whereas on the Fahrenheit scale, water freezes at 32° and boils at 212°. From these numbers you can quickly deduce the conversion formula that you may have forgotten.

The difference between freezing and boiling on one scale is 100° and on the other 180° or 100/180 or 5/9. The Fahrenheit scale is "offset" by 32, since water freezes at 32° on its scale. Thus,

$$C = (F - 32) * 5/9$$

and

$$F = 9/5 * C + 32$$

In our visual program, we'll allow the user to enter a temperature and select the scale to convert it, as we see in Figure 3-1.

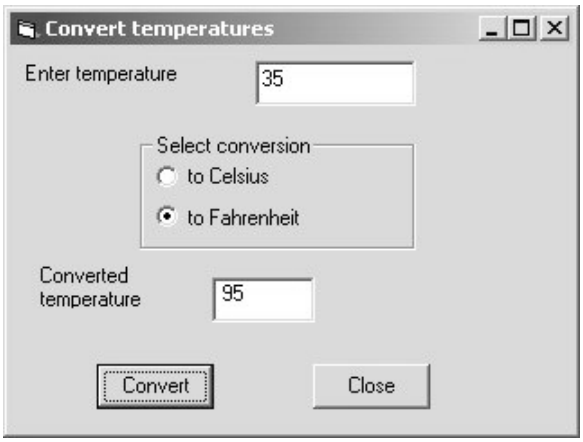

Figure 3-1 Converting 35° Celsius to 95° Fahrenheit with our visual interface

Using the very nice visual builder provided in VB, we can draw the user interface in a few seconds and simply implement routines to be called when the two buttons are pressed.

```
Private Sub btConvert_Click()
Dim enterTemp As Single, newTemp As Single

 enterTemp = Val(txTemperature.Text)

 If opFahr.Value Then
    newTemp = 9 * (enterTemp / 5) + 32
 Else
    newTemp = 5 * (enterTemp - 32) / 9
 End If

 lbNewtemp.Caption = Str$(newTemp)
End Sub
'------
Private Sub Closit_Click()
 End
End Sub
```

The preceding program is extremely straightforward and easy to understand and is typical of how many VB programs operate. However, it has some disadvantages that we might want to improve on.

The most significant problem is that the user interface and the data handling are combined in a single program module, rather than being handled separately. It is usually a good idea to keep the data manipulation and the interface manipulation separate so that changing interface logic doesn't impact the computation logic and vice versa.

Building a Temperature Class

As we noted in the previous chapter, a *class* in VB is a module that can contain both public and private functions and subroutines and can hold data values as well. It is logically the same as a Form, except that it has no visual aspects to it. These functions and subroutines in a class are frequently referred to collectively as *methods*.

Class modules are also like Basic Types or C structs that allow you to keep a set of data values in a single named place and fetch those values using get and set functions, which we then refer to as accessor methods.

You create a class module from the VB integrated development environment (IDE) using the menu item Project | Add class module. Then, you select the Properties window (using function key F4) and enter the module's name. In this example, we'll call the class module clsTemp.

What we want to do is to move all of the computation and conversion between temperature scales into this new clsTemp class module. One way to design this module is to rewrite the calling programs that will use the class module first. In the code sample below, we create an instance of the clsTemp class and use it to do whatever conversions are needed.

```
Private Sub btConvert_Click()
 Dim enterTemp As Single, newTemp As Single
 Dim clTemp As New clsTemp  'create class instance

 If opFahr.Value Then
    clTemp.setCels txTemperature
    lbNewtemp.Caption = Str$(clTemp.getFahr)
 Else
    clTemp.setFahr txTemperature
    lbNewtemp.Caption = Str$(clTemp.getCels)
 End If
```

Note that to create a working copy of a class (called an *instance*) you have to use the *new* keyword with the Dim statement.

```
Dim clTemp As New clsTemp    'create class instance
```

If you simply declare a variable without the New keyword,

```
Dim clTemp as clsTemp
```

you have created a pointer to a class instance but have not initialized an actual instance until you actually create one using New. You can set the value of the pointer you created using the Set keyword.

```
Set clTemp = New clsTemp    'create instance of clsTemp
```

In this program, we have two set methods—setCels and setFahr—and two get methods—getCels and getFahr.

These methods put values into the class and retrieve other values from the class. The actual class is just this.

```
Private temperature As Single

Public Sub setFahr(tx As String)
       temperature = 5 * (Val(tx) - 32) / 9
End Sub

Public Sub setCels(tx As String)
       temperature = Val(tx)
End Sub

Public Function getFahr() As Single
       getFahr = 9 * (temperature / 5) + 32
End Function

Public Function getCels() As Single
       getCels = temperature
End Function
```

Note that the temperature variable is declared as *private,* so it cannot be "seen" or accessed from outside the class. You can only put data into the class and get it back out using the four accessor methods. The main point to this code rearrangement is that the outer calling program does not have to know how the data are stored and how they are retrieved: that is only known inside the class. In this class we always store data in Celsius form and convert on the way in and out as needed. We could also do validity checks for legal strings on the way in, but since the Val function returns zeros and no error for illegal strings, we don't have to in this case.

The other important feature of the class is that it actually *holds data*. You can put data into it, and it will return it at any later time. This class only holds the one temperature value, but classes can contain quite complex sets of data values.

We could easily modify this class to get temperature values out in other scales without still ever requiring that the user of the class know anything about how the data are stored or how the conversions are performed.

Converting to Kelvin

Absolute zero on the Celsius scale is defined as –273.16° degrees. This is the coldest possible temperature, since it is the point at which all molecular motion stops. We can add a function

```
Public Function getKelvin() As Single
    getKelvin = temperature + 273.16
End Function
```

without any changes to the visual client at all. What would the setKelvin method look like?

Putting the Decisions into the Temperature Class

Now we are still making decisions within the user interface about which methods of the temperature class. It would be even better if all that complexity could disappear into the clsTemp class. It would be nice if we just could write our Conversion button click method as

```
Private Sub btConvert_Click()
Dim clTemp As New clsTemp

'put the entered value and conversion request
'into the class
clTemp.setEnterTemp txTemperature.Text, opFahr.Value

'and get out the requested conversion
lbNewtemp.Caption = clTemp.getTempString

End Sub
```

This removes the decision-making process to the temperature class and reduces the calling interface program to just two lines of code.

The class that handles all this becomes somewhat more complex, however, but then it keeps track of what data as been passed in and what conversion must be done.

```
Private temperature As Single   'always in Celsius
Private toFahr As Boolean       'conversion to F requested

Public Sub setEnterTemp(ByVal tx As String, _
       ByVal isCelsius As Boolean)
'convert to Celsius and save
 If Not isCelsius Then
  makeCel tx              'convert and save
  toFahr = False
 Else
  temperature = Val(tx)   'just save temperature
  toFahr = True
 End If
End Sub
'------
Private Sub makeCel(tx As String)
```

```
  temperature = 5 * (Val(tx) - 32) / 9
End Sub
```

Now, the isCelsius Boolean tells the class whether to convert and whether conversion is required on fetching the temperature value. The output routine is simply the following.

```
Public Function getTempString() As String
 getTempString = Str$(getTempVal)
End Function
'------
Public Function getTempVal() As Single
 Dim outTemp As Single
 If toFahr Then          'should we convert to F?
  outTemp = makeFahr    'yes
 Else
  outTemp = temperature 'no
 End If
 getTempVal = outTemp    'return temp value
End Function
'------
Private Function makeFahr() As Single
 Dim t As Single
'convert t to Fahrenheit
 t = 9 * (temperature / 5) + 32
 makeFahr = t
End Function
```

In this class we have both public and private methods. The public ones are callable from other modules, such as the user interface form module. The private ones, makeFahr and makeCel, are used internally and operate on the temperature variable.

Note that we now also have the opportunity to return the output temperature as either a string or a single floating point value and could thus vary the output format as needed.

Using Classes for Format and Value Conversion

It is convenient in many cases to have a method for converting between formats and representations of data. You can use a class to handle and hide the details of such conversions. For example, you might enter an elapsed time in minutes and seconds with or without the colon.

```
315.20
3:15.20
315.2
```

Since all styles are likely, you'd like a class to parse the legal possibilities and keep the data in a standard format within. Figure 3-2 shows how the entries "112" and "102.3" are parsed.

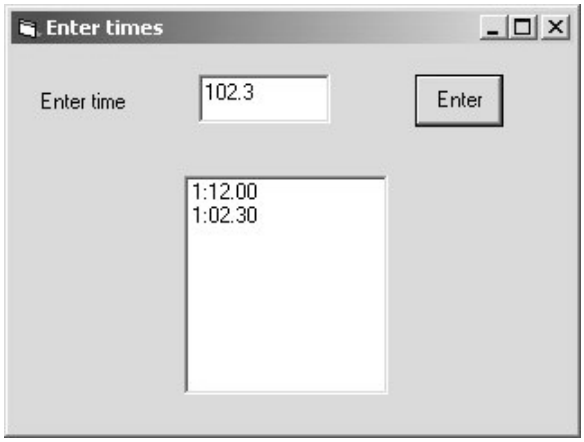

Figure 3-2 A simple parsing program that uses the Times class

The accessor functions for our Times class include the following.

```
setText (tx as String)
setSingle (t as Single)
getSingle as Single
getFormatted as String
getSeconds as Single
```

Parsing is quite simple and depends primarily on looking for a colon. If there is no colon, then values greater than 99 are treated as minutes.

```
Public Function setText(ByVal tx As String) As Boolean
 Dim i As Integer, mins As Long, secs As Single
  errflag = False
  i = InStr(tx, ":")
  If i > 0 Then
  mins = Val(Left$(tx, i - 1))
   secs = Val(Right$(tx, Len(tx) - i))
   If secs > 59.99 Then
     errflag = True
   End If
   t = mins * 100 + secs
 Else
  mins = Val(tx) \ 100
  secs = Val(tx) - (100 * mins)
  If secs > 59.99 Then
```

```
    errflag = True
    t = NT
  Else
   setSingle Val(tx)
  End If
 End If
 setText = errflag
End Function
```

Since illegal time values might also be entered, we test for cases like 89.22 and set an error flag.

Depending on the kind of time measurements these represent, you might also have some non-numeric entries such as NT for no time or in the case of athletic times, SC for scratch or DQ for disqualified. All of these are best managed inside the class. Thus, you never need to know what numeric representations of these values are used internally.

```
Private Const tmNT As Integer = 10000, tmDQ As Integer = 20000
Private Const tmSCRATCH As Integer = 30000
```

Some of these are processed in the code represented by Figure 3-3.

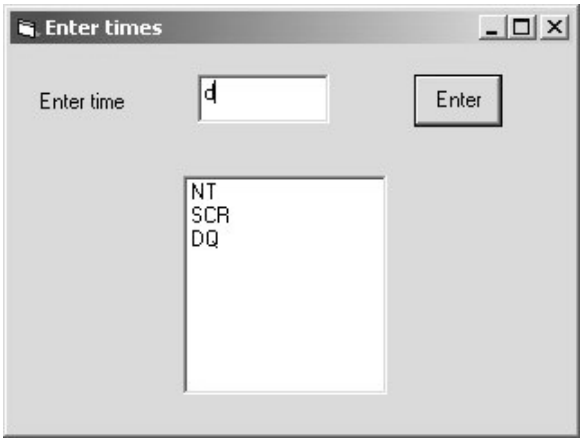

Figure 3-3 The time entry interface, showing the parsing of symbols for No Time, Scratch, and Disqualification

Handling Unreasonable Values

A class is also a good place to encapsulate error handling. For example, it might be that times greater than some threshold value are unlikely and might actually be times that were entered without a decimal point. If large times are unlikely, then a number such as 123473 could be assumed to be 12:34.73.

```
Public Sub setSingle(tv As Single)
t = tv
If tv > minVal And tv <> tmNT Then
  t = tv / 100
End If
End Sub
```

The cutoff value minVal may vary with the domain of times being considered and thus should be a variable. While classes do not have a Form_Load event like Forms do, they do have and initialize events where you can set up default values for variables.

```
Private Sub Class_Initialize()
       minVal = 10000
End Sub
```

To set up the Initialize event in the IDE, click on the left drop-down in the editor title bar so that Class is selected and select Initialize from the right drop-down as shown in Figure 3-4.

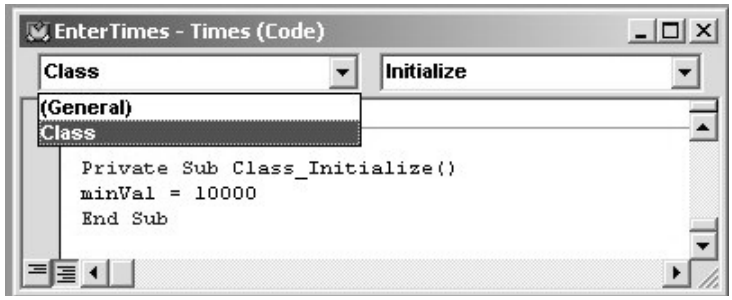

Figure 3-4 Selecting the Class Initialize method

A String Tokenizer Class

A number of languages provide a simple method for dividing strings into tokens separated by a specified character. While VB does not provide a class for this feature, we can write one quite easily using the little-known Split function. The goal of the Tokenizer class will be to pass in a string and obtain the successive string tokens back one at a time. For example, if we had the simple string

```
Now is the time
```

our tokenizer should return four tokens.

```
Now
is
```

```
the
time
```

The critical part of this class is that it holds the initial string and remembers which token is to be returned next.

We could write this class using the Instr function, or we could use the Split function, which approximates the Tokenizer but returns an array of substrings instead of having a class interface. The class we want to write will have a nextToken method that returns string tokens or a zero length string when we reach the end of the series of tokens.

The whole class is shown here.

```
'String tokenizer class
Private s As String, i As Integer
Private sep As String   'token separator
Private stokens() As String 'array of tokens
Public Sub init(ByVal st As String)
 s = st
 setSeparator " "
End Sub
Private Sub Class_Initialize()
 sep = " "   'default is a space separator
End Sub
Public Sub setSeparator(ByVal sp As String)
  sep = sp
  stokens = Split(s, sp)
  i = -1
End Sub
Public Function nextToken() As String
 Dim tok As String
 If i < UBound(stokens) Then
   i = i + 1
   tok = stokens(i)
 Else
   tok = ""
 End If
 nextToken = tok    'return token
End Function
```

The class is illustrated in use in Figure 3-5.

This is the code that uses the Tokenizer class.

```
Private Sub Tokenize_Click()
 Dim tok As New Tokenizer
 Dim s As String

 tok.init txString.Text  'set the string from the input
 lsTokens.Clear          'clear the list box
 s = tok.nextToken       'get a token
```

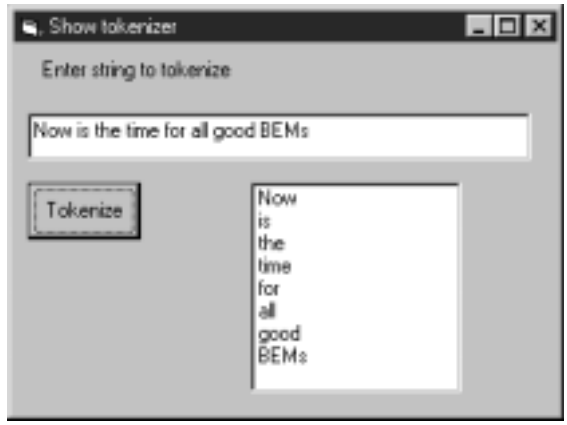

Figure 3-5 The tokenizer in use

```
 While Len(s) > 0         'as long as not of zero length
   lsTokens.AddItem s     'add into the list
   s = tok.nextToken      'and look for next token
 Wend
End Sub
```

Classes as Objects

The primary difference between ordinary procedural programming and object-oriented (OO) programming is the presence of classes. A class is just a module as we have just shown, that has both public and private methods and that can contain data. However, classes are also unique in that there can be any number of *instances* of a class, each containing different data. We frequently refer to these instances as objects. We'll see some examples of single and multiple instances following.

Suppose we have a file of results from a swimming event stored in a text data file. Such a file might look, in part, like this.

```
1 Emily Fenn                17  WRAT       4:59.54
2 Kathryn Miller            16  WYW        5:01.35
3 Melissa Sckolnik          17  WYW        5:01.58
4 Sarah Bowman              16  CDEV       5:02.44
5 Caitlin Klick             17  MBM        5:02.59
6 Caitlin Healey            16  MBM        5:03.62
```

The columns represent place, names, age, club, and time. If we wrote a program to display these swimmers and their times, we'd need to read in and parse this file. For each swimmer, we'd have a first and last name, an age, a club, and a

time. An efficient way to keep the data for each swimmer grouped together is to design a Swimmer class and create an instance for each swimmer.

Here is how we read the file and create these instances. As each instance is created, we add it into a Collection object.

```
Private swimmers As New Collection

Private Sub Form_Load()
Dim f As Integer, S As String
Dim sw As Swimmer
Dim i As Integer

f = FreeFile
'read in data file and create swimmer instances
Open App.Path & "\500free.txt" For Input As #f
While Not EOF(f)
  Line Input #f, S
  Set sw = New Swimmer  'create instances
  sw.init S             'load in data
  swimmers.Add sw       'add to collection
Wend
Close #f
'put names of swimmers in list box
For i = 1 To swimmers.Count
  Set sw = swimmers(i)
  lsSwimmers.AddItem sw.getName
Next i
End Sub
```

The Swimmer class itself parses each line of data from the file and stores it for retrieval using getXXX accessor functions.

```
Private frname As String, lname As String
Private club As String
Private age As Integer
Private tms As New Times
Private place As Integer
'------
Public Sub init(dataline As String)
 Dim tok As New Tokenizer

 tok.init dataline           'initilaize string tokenizer
 place = Val(tok.nextToken)  'get lane number
 frname = tok.nextToken      'get first name
 lname = tok.nextToken       'get last name
 age = Val(tok.nextToken)    'get age
 club = tok.nextToken        'get club
 tms.setText tok.nextToken   'get and parse time
End Sub
'------
```

```
Public Function getTime() As String
 getTime = tms.getFormatted
End Function
'------
Public Function getName() As String
 'combine first and last names and return together
 getName = frname & " " & lname
End Function
'------
Public Function getAge() As Integer
 getAge = age
End Function
'------
Public Function getClub() As String
 getClub = club
End Function
```

Class Containment

Each instance of the Swimmer class contains an instance of the Tokenizer that it uses to parse the input string and an instance of the Times class we wrote previously to parse the time and return it in formatted form to the calling program. Having a class contain other classes is a very common ploy in OO programming and is one of the main ways we can build up more complicated programs from rather simple components.

The program that displays these swimmers is shown in Figure 3-6.

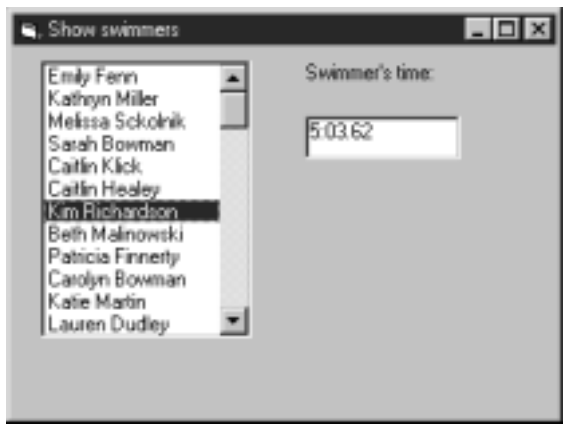

Figure 3-6 A list of swimmers and their times, using containment

When you click on any swimmer, her time is shown in the box on the right. The code for showing that time is extremely easy to write, since all the data are in the swimmer class.

```
Private Sub lsSwimmers_Click()
 Dim i As Integer
 Dim sw As Swimmer
 i = lsSwimmers.ListIndex        'get index of list
 If i >= 0 Then
   Set sw = swimmers(i)          'get that swimmer
   lbTime.Caption = sw.getTime   'display that time
 End If
End Sub
```

Class Initialization

As we showed previously, you can use the Class_Initialize event to set up default values for some class variables. However, if you want to set up some values that are specific for each instance (such as our swimmer's names and times), we need a standard way to do this. In other languages, classes have special methods called *constructors* that you can use to pass in useful data at the same time you create the instance. Since VB6 classes lack these methods, we introduce the convention of an *init* method that we'll use to pass in instance specific data.

In our preceding Swimmer class, note that we have an init method that in turn calls the init method of the Tokenizer class.

```
Public Sub init(dataline As String)
Dim tok As New Tokenizer

tok.init dataline           'initialize string tokenizer
```

Other languages, including VB7, also allow classes to have a series of constructors that each have different arguments. Since this is not a feature of VB6, we'll use various setXXX methods instead.

Classes and Properties

Classes in VB can have Property methods as well as public and private functions and subs. These correspond to the kinds of properties you associate with Forms, but they can store and fetch any kinds of values you care to use. For example, rather than having methods called getAge and setAge, you could have a single Age property that then corresponds to a Property Let and a Property Get method.

```
Property Get age() As Integer
 age = sAge  'return the current age
End Property
'------
Property Let age(ag As Integer)
 sAge = ag   'save a new age
End Property
```

To use these properties, you refer to the Let property on the left side of an equals sign and the Get property on the right side.

```
myAge = sw.Age       'Get this swimmer's age
sw.Age = 12          'Set a new age for this swimmer
```

Properties are somewhat vestigial, since they really applied more to Forms, but many programmers find them quite useful. They do not provide any features not already available using get and set methods, and both generate equally efficient code.

In the revised version of our SwimmerTimes display program, we convert all of the get and set methods to properties and then allow users to vary the times of each swimmer by typing in new ones. Here is the Swimmer class.

```
Option Explicit
Private frname As String, lname As String
Private sClub As String
Private sAge As Integer
Private tms As New Times
Private place As Integer
'------
Public Sub init(dataline As String)
Dim tok As New Tokenizer

 tok.init dataline           'initilaize string tokenizer
 place = Val(tok.nextToken)  'get lane number
 frname = tok.nextToken      'get first name
 lname = tok.nextToken       'get last name
 sAge = Val(tok.nextToken)   'get age
 sClub = tok.nextToken       'get club
 tms.setText tok.nextToken   'get and parse time
End Sub
'------
Property Get time() As String
 time = tms.getFormatted
End Property
'------
Property Let time(tx As String)
 tms.setText tx
End Property
'------
Property Get Name() As String
 'combine first and last names and return together
 Name = frname & " " & lname
End Property
'------
Property Get age() As Integer
 age = sAge  'return the current age
End Property
'------
```

```
Property Let age(ag As Integer)
 sAge = ag   'save a new age
End Property
'------
Property Get Club() As String
 Club = sClub
End Property
```

Then when the txTime text entry field loses focus, we can store a new time as follows.

```
Private Sub txTime_Change()
 Dim i As Integer
 Dim sw As Swimmer
 i = lsSwimmers.ListIndex        'get index of list
 If i >= 0 Then
  Set sw = swimmers(i)          'get that swimmer
  sw.time = txTime.Text  'store that time
 End If
End Sub
```

Another Interface Example—The Voltmeter

Suppose that you need to interface a digital voltmeter to your computer. We'll assume that the meter can connect to your serial port and that you send it a string command and get the measured voltage back as a string. We'll also assume that you can set various measurement ranges such as millivolts, volts, and tens of volts. The methods for accessing this voltmeter might look like this.

```
'The Voltmeter class
Public Sub setRange(ByVal maxVal As Single)
  'set maximum voltage to measure
End Sub
'------
Public Function getVoltage() As Single
 'get the voltage and convert it to a Single
End Function
```

The nice visual data-gathering program you then write for this voltmeter works fine, until you suddenly need to make another simultaneous set of measurements. You discover that the model of voltmeter that you wrote the program for is no longer available and that the new model has different commands. It might even have a different interface (IEEE-488 or USB, for instance).

This is an ideal time to think about program interfaces. The simple two-method interface we specified previously should work for any voltmeter, and the rest of the program should run without change. All you need to do is write a class for the new voltmeter that implements the same interface. Then your

data-gathering program only needs to be told which meter to use and it will run completely unchanged, as we show here.

```
Private Sub OK_Click()
 If opPe.Value Then
    Set vm = New PE2345
 Else
   Set vm = New HP1234
 End If
 vm.getVoltage
End Sub
```

Further, should your data needs expand so that there are still more meters, you can quickly write more classes that implement this same Voltmeter interface. This is the advantage of OO programming in a nutshell: Only the individual classes have detailed knowledge of how they work. The only external knowledge is contained in the interfaces.

A vbFile Class

File handling in VB is for the most part awkward and primitive for historical reasons. The statements for opening files have this form.

```
f = FreeFile
Open "file.txt" for Input as #f
```

And those for reading data from files have this form.

```
Input #f, s
Line Input #f, sLine
```

There is no simple statement for checking for the existence of a file, and the file rename and delete have counterintuitive names.

```
Exists = len(dir$(filename))>0          'file exists
Name file1 as file2                     'Rename file
Kill filename                           'Delete file
```

None of these statements are at all object oriented. There ought to be objects that encapsulate some of this awkwardness and keep the file handles suitably hidden.

VB6 introduced the Scripting.FileSystemObject as a way to handle files in a presumably more object-oriented way. However, these objects are not fully realized and a bit difficult to use. Thus, we might do well to create our own vbFile object with convenient methods. These methods could include the following.

```
Public Function OpenForRead(Filename As String) As Boolean
Public Function fEof() As Boolean
Public Function readLine() As String
Public Function readToken() As String
Public Sub closeFile()
Public Function exists() As Boolean
Public Function delete() As Boolean
Public Function OpenForWrite(fname As String) As Boolean
Public Sub writeText(s As String)
Public Sub writeLine(s As String)
Public Sub setFilename(fname As String)
Public Function getFilename() As String
```

A typical implementation of a few of these methods includes the following.

```
Public Function OpenForRead(Filename As String) As Boolean
 'open file for reading
 f = FreeFile                'get a free handle
 File_name = Filename        'save the filename

 On Error GoTo nofile       'trap errors
 Open Filename For Input As #f
    opened = True           'set true if open successful
oexit:
    OpenForRead = opened    'return to caller
Exit Function
'--error handling--
nofile:
  end_file = True           'set end of file flag
  errDesc = Err.Description 'save error messae
  opened = False            'no file open
  Resume oexit              'and resume
End Function
'------
Public Function fEof() As Boolean
 'return end of file
 If opened Then
  fEof = EOF(f)
 Else
  fEof = True    'if not opened then end file is true
 End If
End Function
'------
Public Function readLine() As String
 Dim s As String
 'read one line from a text file
 If opened Then
   Line Input #f, s
   readLine = s
 Else
   readLine = ""
 End If
End Function
```

With these useful methods, we can write a simple program to read a file and display it in a list box.

```
Dim fl As New vbFile
cDlg.ShowOpen   'use common dialog open

fl.OpenForRead cDlg.Filename
'read in up to end of file
sline = fl.readLine
While Not fl.fEof
  lsFiles.AddItem sline
  sline = fl.readLine
Wend
fl.closeFile
```

Now, the implementation of this vbFile object can change as VB evolves. However, by concealing the details, we can vary the implementation in the future. We'll see another implementation of this class when we discuss VB.NET.

Programming Style in Visual Basic

You can develop any of a number of readable programming styles for VB. The one we use here is partly influenced by Microsoft's Hungarian notation (named after its originator, Charles Simonyi) and partly on styles developed for Java.

We favor using names for VB controls such as buttons and list boxes that have prefixes that make their purpose clear, and we will use them whenever there is more than one of them on a single form.

Control	Prefix	Example
Buttons	bt	btCompute
List boxes	ls	lsSwimmers
Radio (option buttons)	op	opFSex
Combo boxes	cb	cbCountry
Menus	mnu	mnuFile
Text boxes	tx	TxTime

We will name classes in ways that describe their purpose and only precede them with clsXXX if there is any ambiguity. We will not generally create new names for labels, frames, and forms when they are never referred to directly in the code. Even though VB is case insensitive, we otherwise will begin class

names with capital letters and instances of classes with lowercase letters. We will also spell instances and classes with a mixture of lowercase and capital letters to make their purpose clearer.

```
swimmerTime
```

Summary

In this chapter, we've introduced VB classes and shown how they can contain public and private methods and can contain data. Each class can have many instances and each could contain different data values. Classes can also have Property methods for setting and fetching data. These Property methods provide a simpler syntax over the usual getXXX and setXX accessor methods but have no other substantial advantages.