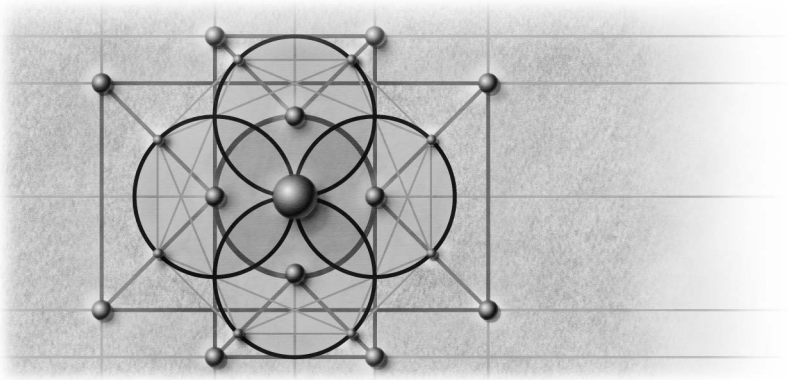

11

Resolving Issues with Language

The Basic programming language has been around for a long time in a variety of forms—GW-BASIC, QuickBasic, Visual Basic, and Visual Basic for Applications (VBA), to name just a few Microsoft PC-based varieties. Each product provides its own brand of the Basic programming language, supporting or not supporting certain types and language constructs. Each product has also made an attempt to clean up or simplify the language. Microsoft Visual Basic .NET offers yet another version of the Basic language, adding its own types and language constructs and omitting others.

In Visual Basic .NET you will not find *Type…End Type*, *GoSub…Return*, *While…Wend*, nonzero-based arrays, the *Currency* type, static subroutines, or fixed-length strings. You will, however, find new language elements such as *Structure…End Structure*, *Inherits*, *Overloads*, *Try…Catch*, *SyncLock*, the *Char* type, the *Decimal* type, and assignment shortcuts such as += and -= to increment or decrement a value. New statements such as *Inherits*, *Overloads*, *Try…Catch*, and *SyncLock* provide support for new features—inheritance, structured exception handling, and multithreading—not found in previous versions. Other changes such as the removal of *GoSub…Return*, the change from *Type…End Type* to *Structure…End Structure*, and the addition of the *Char* type are intended to modernize the language. Still other changes—removal of support for nonzero-based arrays and fixed-length strings—are the result of tradeoffs needed to make Visual Basic .NET work with other .NET languages.

This chapter discusses upgrade issues related to your code. In particular, it focuses on language elements in Visual Basic 6 that the Upgrade Wizard doesn't automatically upgrade (or upgrades only partially) for you.

Language Elements

Let's look first at language-related upgrade issues, setting aside for now issues related to data structures and user-defined types. We'll cover control-of-flow statements such as *While…Wend* and *GoSub…Return*, precompiler statements such as *#If…#End If*, and language statements such as *Abs*, *Open*, and *Close*.

#If…#End If Precompiler Statements

All conditional compilation statements are maintained after an upgrade. All code within all paths of an *#If…#End If* block will be found in the upgraded project, but only the code in the execution path—the path that evaluates to *True*—is upgraded. This happens because the Upgrade Wizard cannot guarantee that code in other execution paths is valid. For example, you can include any text you want in an *#If…#End If* block that evaluates to *False*, such as

```
#Const COMMENT = False

#If COMMENT Then
This is my invalid code. There is no comment. This will only confuse
the Upgrade Wizard if it tries to upgrade me.
#End If
```

You should make sure that you have turned on all the conditional compilation arguments in your Visual Basic 6 project for the code paths you want to upgrade. Also be sure that your code compiles and runs in Visual Basic 6. The wizard will not be able to upgrade invalid code contained in your project. This includes code contained within an *#If…#End If* block that evaluates to *True*.

Constants and Constant Expressions

If you define a constant (in Visual Basic 6) to represent a property on a Windows form or control, you may find that the code does not compile after you upgrade the project. For example, suppose that your Visual Basic 6 project contains the following code:

```
Const COLOR_RED = vbRed
```

The Upgrade Wizard will upgrade the code to the following:

```
'UPGRADE_NOTE: COLOR_RED was changed from a Constant to a Variable.
Dim COLOR_RED As System.Drawing.Color = System.Drawing.Color.Red
```

Why was the *Const* declaration changed to a *Dim*? The reason is that *System.Drawing.Color.Red*, although it looks like a constant, is not; it is a nonconstant expression. This means that, rather than being a constant value, such as

255 (the value of *vbRed*), it is a property that returns a *System.Drawing.Color* object. To see the declaration for *System.Drawing.Color.Red* within the upgraded Visual Basic .NET project, right-click Red in the Visual Basic .NET code window and choose Go To Definition. The Object Browser will display the definition of *System.Drawing.Color.Red*, as follows:

```
Public Shared ReadOnly Property Red As System.Drawing.Color
```

In some cases, you may encounter a compiler error in your upgraded code. For example, suppose that in Visual Basic 6 you create an enumerator containing values for red, green, and blue called *RGBColors*. You use the enumerator values in code to combine red and green to create a yellow background for your form, as follows:

```
Enum RGBColors
   Red = vbRed
   Green = vbGreen
   Blue = vbBlue
End Enum

Private Sub Form_Load()
   BackColor = RGBColors.Red + RGBColors.Green
End Sub
```

The resulting Visual Basic .NET code after upgrade will be as follows:

```
'UPGRADE_ISSUE: Declaration type not supported: Enum member
'of type Color.
Enum RGBColors
   Red = System.Drawing.Color.Red
   Green = System.Drawing.Color.Lime
   Blue = System.Drawing.Color.Blue
End Enum

Private Sub Form1_Load(ByVal eventSender As System.Object, _
                       ByVal eventArgs As System.EventArgs) _
                       Handles MyBase.Load
   BackColor = System.Drawing.ColorTranslator.FromOle(RGBColors.Red + _
   RGBColors.Green)
End Sub
```

Because the declarations for the color objects *Red*, *Lime*, and *Blue* are not constant, each line results in the compiler error "Constant expression is required."

There are a couple of ways that you can fix up the code to work. You can define your own constant values, or you can use nonconstant values directly, as described in the sections that follow.

Define Your Own Constant Values

The easiest way to fix your code is to use constant values in place of the *Red*, *Lime*, and *Blue* objects in the code just given. For example, you can declare constants for *vbRed*, *vbGreen*, and *vbBlue* so that you can define the enumerator members as these constant types, as was done in the Visual Basic 6 code.

```
Const vbRed As Integer = &HFF
Const vbGreen As Integer = &HFF00
Const vbBlue As Integer = &HFF0000

Enum RGBColors
   Red = vbRed
   Green = vbGreen
   Blue = vbBlue
End Enum

Private Sub Form1_Load(ByVal eventSender As System.Object, _
                       ByVal eventArgs As System.EventArgs) _
                       Handles MyBase.Load
   BackColor = System.Drawing.ColorTranslator.FromOle(RGBColors.Red + _
   RGBColors.Green)
End Sub
```

Use Nonconstant Values Directly

Another way to solve the problem is to replace any instances in which a member of *RGBColors*, such as *RGBColors.Red*, is used with the matching *System.Drawing.Color*, such as *System.Drawing.Color.Red*. When you make this change, a new question arises: How do you manage calculations that involve two objects? For example, in the example above, how do you add the values of two color objects together to arrive at a new color value? You can't directly add two objects together to get a result. If you can use a method or a function that will give you a meaningful numeric value for the object, you can use the numeric values in your calculation. Using a numeric-to-object conversion function allows you to turn the result of the numeric calculation back into an object representing the calculated value.

In the case of color objects, you can obtain a meaningful numeric (color) value by using the *ToOle* method of the *System.Drawing.ColorTranslator* class. The *ToOle* method takes a color object—such as *System.Drawing.Color.Red*—and obtains an RGB color value for it. *ToOle(System.Drawing.Color.Red)*, for example, will return the RGB value 255, or FF in hexadecimal. Not by coincidence, the *vbRed* constant in Visual Basic 6 has the same value. Once you have calculated the new color, you can convert the result back to a color object by using the *ColorTranslator.FromOle* method. The following is an example of

how you can change the original upgraded Visual Basic .NET code to use the *System.Drawing.Color* values directly:

```
'Include the Imports statement at the top of the Form file
Imports System.Drawing
…
Private Sub Form1_Load(ByVal sender As System.Object, _
                       ByVal e As System.EventArgs) Handles MyBase.Load
   BackColor = _
      ColorTranslator.FromOle(ColorTransltor.ToOle(Color.Red) _
      + ColorTranslator.ToOle(Color.Lime))

End Sub
```

It's up to you whether you want to define your own constants or use the nonconstant values directly in your code. If, in your original code, you are using the constant values in combination with other constant values—say, in a numeric calculation—it would make sense to define your own constant values and use the constants in your code. Otherwise, if you are making a simple assignment of a constant value to a property, it would make more sense to use the nonconstant value—in other words, the object—directly.

Control Flow

If you never knew that Visual Basic supported *GoSub…Return*, *On…GoTo*, and *On…GoSub*, and you don't use these statements in your Visual Basic 6 code, you're ahead of the game. Visual Basic .NET drops support for these three statements.

GoSub…Return

GoSub…Return is a holdover from earlier generations of the Basic language that did not support subroutines, *Sub* and *Function*. Visual Basic .NET does not support the *GoSub* statement. It does, however, support the *Return* statement, but its meaning is different from that in Visual Basic 6. *Return* is used to return from a subroutine. You can use the *Return* statement to return a value in a function.

If you are using the *GoSub* statement in your code, we recommend copying the code associated with the *GoSub* label to its own subroutine. Any local variables that the *GoSub* label handler uses should be passed as parameters to the subroutine that you create. The following code uses *GoSub* in two places within *Sub Main* to display a summary of the number of times an animal shows up in a sorted list of animals. It is a good example of when you would

use *GoSub* in a subroutine to perform a suboperation that reuses local variable values.

```
Sub Main()

   Dim Animals(2) As String
   Dim ctAnimal As Long
   Dim PrevAnimal As String
   Dim i As Long

   Animals(0) = "Alligator"
   Animals(1) = "Monkey"
   Animals(2) = "Monkey"

   For i = 0 To UBound(Animals)

      ' Detect break in sequence and show summary info
      If PrevAnimal <> "" And PrevAnimal <> Animals(i) Then
         GoSub ShowDetail
      End If

      ctAnimal = ctAnimal + 1
      PrevAnimal = Animals(i)

   Next

   ' Show summary info for last animal in list
   GoSub ShowDetail

   Exit Sub

ShowDetail:
   MsgBox PrevAnimal & " " & ctAnimal
   ctAnimal = 0
   Return
End Sub
```

The Upgrade Wizard upgrades the code as is, inserting UPGRADE_ISSUE comments for each occurrence of *GoSub* and UPGRADE_WARNING comments for each occurrence of *Return*. The comments tell you that *GoSub* is not supported and that *Return* has a new behavior in Visual Basic .NET.

You can replace the *GoSub* statements with calls to a subroutine called *ShowDetail*. You need to pass the local variables as parameters to the *ShowDetail* subroutine. In the previous example, you would pass *PrevAnimal* and *ctAnimal* to *ShowDetail*. You need to pass *ctAnimal ByRef* so that the value gets reset to 0 when execution returns from *ShowDetail*. The following code

demonstrates how you can use a function—*ShowDetail*—in place of a *GoSub* label of the same name (found in the code shown previously):

```
Public Sub Main()

   Dim Animals(2) As String
   Dim ctAnimal As Integer
   Dim PrevAnimal As String
   Dim i As Integer

   Animals(0) = "Alligator"
   Animals(1) = "Monkey"
   Animals(2) = "Monkey"

   For i = 0 To UBound(Animals)
      ' Detect break in sequence and show summary info
      If PrevAnimal <> "" And PrevAnimal <> Animals(i) Then
         ShowDetail(PrevAnimal, ctAnimal)
      End If

      ctAnimal = ctAnimal + 1
      PrevAnimal = Animals(i)
   Next

   ' Show summary info for last animal in list
   ShowDetail(PrevAnimal, ctAnimal)
End Sub

Sub ShowDetail(ByVal Animal As String, ByRef ctAnimal As Integer)
   MsgBox(Animal & " " & ctAnimal)
   ctAnimal = 0
End Sub
```

On...GoTo

On…GoTo is an outdated version of *Select Case*. *On…GoTo* evaluates a given numeric expression for a value between 1 and 255 and then jumps to the label given by the *n*th parameter after *GoTo*.

The following Visual Basic 6 example demonstrates the use of *On…GoTo*.

```
Dim Mode As Integer

Mode = 2   'WriteMode
On Mode GoTo ReadMode, WriteMode

MsgBox "Unexpected mode"
Exit Sub
```

(continued)

```
ReadMode:
   MsgBox "Read mode"
   Exit Sub

WriteMode:
   MsgBox "Write mode"
```

The Upgrade Wizard upgrades the code to use *Select Case* and *GoTo* as follows:

```
Dim Mode As Short

Mode = 2 'WriteMode
Select Case Mode
   Case Is < 0
      Error(5)
   Case 1
      GoTo ReadMode
   Case 2
      GoTo WriteMode
End Select

MsgBox("Unexpected mode")
Exit Sub

ReadMode:
MsgBox("Read mode")
Exit Sub

WriteMode:
MsgBox("Write mode")

End Sub
```

Although the wizard produces correct code, you will probably want to move the code contained under each label to each *Case* statement. This step will eliminate the need for *GoTo* and will also condense your Visual Basic .NET code as follows:

```
Dim Mode As Short

Mode = 2 'WriteMode
Select Case Mode
   Case 1
      MsgBox("Read mode")
   Case 2
      MsgBox("Write mode")
   Case Else
      MsgBox("Unexpected mode")
End Select
```

On...GoSub

On...GoSub works in much the same way as *On...GoTo*, except that when you jump to the label, execution can return back to the next statement after the *On...GoSub* statement when you call *Return*. The following example, in particular the *UpdateAccountBalance* subroutine, demonstrates the use of *On...GoSub* to jump to a transaction—deposit or withdrawal—depending on the value of the passed-in *Transaction* variable. The *Transaction* variable can either be 1 for deposit or 2 for withdrawal. If, for example, the value is 1, the *On...GoSub* statement jumps to the first label specified in the list—in this case *Deposit*:

```
Option Explicit
Private m_AccountBalance As Currency

Private Sub Main()

   Const Deposit = 1
   Const Withdrawal = 2

   m_AccountBalance = 500
   UpdateAccountBalance Deposit, 100

End Sub

Private Sub UpdateAccountBalance(ByVal Transaction As Integer, _
                                 ByVal Amount As Currency)

   On Transaction GoSub Deposit, Withdrawal
   MsgBox "New balance: " & m_AccountBalance
   Exit Sub

Deposit:
   m_AccountBalance = m_AccountBalance + Amount
   Return

Withdrawal:
   m_AccountBalance = m_AccountBalance - Amount
   Return

End Sub
```

As in the *On...GoTo* example given earlier, you can change your *On...GoSub* code to use a *Select Case* statement.

```
Select Case Transaction
   Case Deposit
      m_AccountBalance = m_AccountBalance + Amount
   Case Withdrawal
      m_AccountBalance = m_AccountBalance - Amount
End Select
MsgBox "New balance: " & m_AccountBalance
```

File Functions

Visual Basic 6 includes a number of language statements—such as *Open*, *Close*, *Put #*, *Get #*, and *Print #*—that allow you to read and write text and binary files. Visual Basic .NET provides a set of functions that are compatible with the Visual Basic 6 file statements. The Upgrade Wizard will upgrade your Visual Basic 6 file statements to the equivalent Visual Basic .NET functions. Table 11-1 illustrates those relationships.

Table 11-1 File Function Mapping from Visual Basic 6 to Visual Basic .NET

Visual Basic 6 Statement	Visual Basic .NET Function
ChDir	ChDir
ChDrive	ChDrive
Close	FileClose
FileCopy	FileCopy
Get	FileGet
Input #	Input
Kill	Kill
Line Input #	LineInput
Lock	Lock
MkDir	MkDir
Open	FileOpen
Print #	Print, PrintLine
Put	FilePut
Reset	Reset
RmDir	RmDir
SetAttr	SetAttr
Unlock	Unlock
Width #	FileWidth
Write #	Write, WriteLine

File Format Compatibility

When you upgrade an application from Visual Basic 6 to Visual Basic .NET, not only do you need to worry about getting your Visual Basic .NET application working in a compatible way, but you also need to be concerned about being able to read data that was written by a Visual Basic 6 application. Because Visual Basic .NET offers you a full set of compatible file functions, you can read files created by Visual Basic 6 applications. For example, you can read and write each of the file types that Visual Basic 6 supports—*Text*, *Binary*, and *Random*. In order to maintain compatibility, however, you need to be aware of some subtleties in the way that you read and write files.

Print*, *PrintLine*, *Write*, and *WriteLine The Visual Basic .NET *Print* and *PrintLine* functions are equivalent to the Visual Basic 6 *Print #* function. Which function you use depends on how *Print #* is used in your Visual Basic 6 application. If the text you are printing is terminated by a semicolon, use *Print*. If the text you are printing is not terminated by a semicolon, use *PrintLine*. The same sort of mapping applies to the Visual Basic 6 *Write #* function. For example, the following Visual Basic 6 code:

```
Print #1, "This is a full line of text"
Print #1, "This is a partial line of text";
Write #2, "This is a full line using write"
Write #2, "This is a partial line using write";
```

is equivalent to the following Visual Basic .NET code:

```
PrintLine(1, "This is a full line of text")
Print(1, "This is a partial line of text")
WriteLine(2, "This is a full line using write")
Write(2, "This is a partial line using write")
```

The Upgrade Wizard automatically upgrades your Visual Basic 6 *Print #* and *Write #* code to use the appropriate function, based on whether the text is terminated by a semicolon. It is when you start editing your upgraded Visual Basic .NET code or writing new code that you need to be aware that there are two separate functions that give you full-line versus partial-line file printing.

When Reading or Writing Variables, Size Matters If you are reading or writing information to a binary file, you need to pay attention to the variables you are using. For example, the byte size of *Integer* and *Long* variables is different between Visual Basic 6 and Visual Basic .NET. An *Integer* is 16 bits in Visual Basic 6 and 32 bits in Visual Basic .NET; a *Long* is 32 bits in Visual Basic 6 and 64 bits in Visual Basic .NET. If, in Visual Basic .NET, you are reading data from a binary file that was written as a Visual Basic 6 *Integer*, you will need to use a variable of type *Short* (16 bits) to read the data.

If you use the Upgrade Wizard to upgrade your application, this change will not be an issue because the wizard automatically maps all your variables that are type *Integer* to *Short* and maps type *Long* to *Integer*. However, when you edit the upgraded code or write new code, you need to be careful to use the correct data type—based on size, not name—when reading and writing binary files. This includes files that you open using Random mode.

Random-Access Files and Dynamic Arrays If you create a random-access binary file in Visual Basic 6, additional header information relating to the format of your data may be written to the file. This header information tells Visual Basic how much data to read back in when it reads the file. The information is written when you use a dynamic data type such as a dynamic array or a variable-length string. For example, header information associated with a dynamic array includes the count of the array dimensions, the size of the array, and the lower bound. Header information associated with a variable-length string includes the length of the string. If you use a fixed-length array or a fixed-length string, no header information is written.

Dynamic Array Refresher

Not sure what a dynamic array is? You are probably not alone. The difference between a dynamic array and a fixed-length array involves a subtle variation in syntax. If your declaration for an array does not specify the size, it is considered to be a dynamic array. For example, the following statement declares a dynamic array:

```
Dim MyDynamicArray() As Integer
```

Once the array is declared as a dynamic array, it is always considered a dynamic array, even after you redimension it to a known size, as in the following statement:

```
ReDim MyDynamicArray(100)
```

If you specify the size of the array as part of the declaration, it is considered to be a fixed-length array. For example,

```
Dim MyDynamicArray(100) As Integer
```

Once you have dimensioned a fixed-length array, you cannot use the *ReDim* statement to change its size.

The Upgrade Wizard does not handle situations that involve writing the contents of a dynamic array to a random-access file. Consider the following Visual Basic 6 code, which initializes the contents of a dynamic array, *OutputData*, with its own index values.

```
Dim OutputData() As Byte
ReDim OutputData(100)

Dim i As Integer

For i = 0 To UBound(OutputData)
   OutputData(i) = i
Next

Open Environ("TEMP") & "\ArrayData.Dat" For Random As #1 Len = 120
Put #1, , OutputData
Close #1
```

If we use the following Visual Basic 6 code to read in the contents of the array, everything works fine:

```
Dim OutputData() As Byte
Dim i As Integer

Open Environ("TEMP") & "\ArrayData.Dat" For Random As #1 Len = 120
Get #1, , OutputData
Close #1

For i = 0 To UBound(OutputData)
   If i <> OutputData(i) Then
      MsgBox "Error: Data element (" & i & ") is incorrect. " & _
         "Value=" & OutputData(i)
      Exit For
   End If
Next
```

If you upgrade this Visual Basic 6 code to Visual Basic .NET, the resulting code is as follows:

```
Dim OutputData() As Byte
Dim i As Short

FileOpen(1, Environ("TEMP") & "\ArrayData.Dat", _
   OpenMode.Random, , , 120)
'UPGRADE_WARNING: Get was upgraded to FileGet and has a new behavior.
FileGet(1, OutputData)
FileClose(1)

For i = 0 To UBound(OutputData)
```

(continued)

```
   If i <> OutputData(i) Then
      MsgBox("Error: Data element (" & i & ") is incorrect. " & _
         "Value=" & OutputData(i))
      Exit For
   End If
Next
```

If you run this upgraded code, the first problem you will encounter is a "Cannot determine array type because it is Nothing" exception on the line *FileGet(1, OutputData)*. This exception occurs for exactly the reason that it indicates: the *OutputData* array is uninitialized, so its value is *Nothing*. It is caused by a difference between Visual Basic 6 and Visual Basic .NET. In Visual Basic 6, although the array is not initialized, the Visual Basic *Get* statement can still determine the type of the array at run time. The type is needed so that the *Get* statement can redimension the array with the correct type and fill it with the data contained in the file.

To fix this problem, you need to redimension the array with at least one element so that it is initialized to a value other than *Nothing*. Include the following statement after the *Dim OutputData() As Byte* statement:

```
ReDim OutputData(0)
```

Run the code again, and you uncover another problem. A message box pops up telling you "Error: Data element (0) is incorrect. Value=1." This message occurs because the Visual Basic .NET *FileGet* statement assumes that the data it is reading was written from a fixed-length array, not a dynamic array. The value of 1 that leads to the error is actually a value contained within the header information for the dynamic array. It is the number of dimensions contained in the dynamic array, not the dynamic array data itself.

To fix this problem, you need to tell the *FileGet* statement that the data it is reading is coming from a dynamic array. You do this by specifying the optional *FileGet* parameter called *ArrayIsDynamic*. Change the following line in the Visual Basic .NET upgraded code:

```
FileGet(1, OutputData)
```

to

```
FileGet(1, OutputData, ArrayIsDynamic:=True)
```

Run the code again, and it will work as expected. The *FileGet* function will expand the *OutputData* array to have an upper bound of 100 and will read in the array contents, following the array header information, from the file.

Random-Access Files and Fixed-Length Strings A problem similar to the dynamic array problem discussed in the previous section will occur if you attempt to read a fixed-length string from a random-access file. Consider the following upgraded Visual Basic .NET code to read in a fixed-length string:

```
Dim OutputData As New VB6.FixedLengthString(9)
Dim i As Short

FileOpen(1, Environ("TEMP") & "\StringData.Dat", _
   OpenMode.Random, , , 15)

'UPGRADE_WARNING: Get was upgraded to FileGet and has a new behavior.
FileGet(1, OutputData.Value)
FileClose(1)

For i = 1 To Len(OutputData.Value)
   If CStr(i) <> Mid(OutputData.Value, i, 1) Then
      MsgBox("Error: Character (" & i & ") is incorrect.  Value=" & _
         Mid(OutputData.Value, i, 1))
      Exit For
   End If
Next
```

This situation is different from the dynamic array case in that the *FileGet* function assumes that the data contained within the file is a variable-length string, not a fixed-length string. This means that the *FileGet* function expects the file to contain a string header giving the length of the string. If you execute the code, you will encounter a "Bad record length" exception. The *FileGet* function is getting tripped up because it thinks the first 4 bytes of data is the string length when in fact it is the string data. The function attempts to use the first 4 bytes of string data as the length, comes up with a large number—larger than the record size—and throws an exception.

To address this problem, the *FileGet* function provides an optional parameter called *StringIsFixedLength*. Set the parameter to *True*, and the code will work as expected. To fix the Visual Basic .NET code, change the following line:

```
FileGet(1, OutputData.Value)
```

to

```
FileGet(1, OutputData.Value, StringIsFixedLength:=True)
```

Types and Type Operations

Visual Basic .NET offers most of the same types as Visual Basic 6. You will find the same basic types, such as *Byte*, *Integer*, *Single*, and *String*. You can also define the same types, such as *Enum*, user-defined types (called *Structure* in Visual Basic .NET), and *Class*. Visual Basic .NET also offers some new types that you can use—*Short*, *Char*, and *Decimal*. It also gives you a new *Interface* type that you can define or use. This section focuses on Visual Basic 6 types for which support has changed in Visual Basic .NET.

Object Replaces *Variant*

The *Variant* data type has existed in the Visual Basic language since Visual Basic 2. It is part of the Visual Basic franchise. If you do not specify a type for a variable, you can always rely on *Variant* being the default type. The beauty of the *Variant* data type is that you can assign any numeric, string, array, or object reference value to it. It can also be in various states of nothingness—missing, *Null*, or *Nothing*.

The same characteristic that makes the *Variant* data type easy to use is also the one that will get you into trouble. The compiler will not and cannot step in to save you if you assign variants containing incompatible types to each other. For example, if you attempt to assign a variant containing an object to a variant containing a simple type such as an *Integer*, you will encounter a run-time error.

Visual Basic .NET introduces the *Object* type. It is a combination of the *Object* and *Variant* types found in Visual Basic 6. Like the Visual Basic 6 *Object* type, a Visual Basic .NET object can be assigned to an object instance, such as a control, and you can make late-bound calls to the object. The *Variant*-like behavior of the *Object* type is that you can assign any type to it, ranging from *Integer*, *Array*, and *Structure* to class objects.

IsMissing Is Lost

One characteristic that the Visual Basic .NET *Object* type lacks compared to the Visual Basic 6 *Variant* type is a value that means *missing*. Other than being set to an actual value, the only other state a Visual Basic .NET *Object* type can be set to is *Nothing*. The following Visual Basic 6 code demonstrates how this can be a problem if you are testing for a missing optional parameter:

```
Private m_Info As String
Private m_Picture As StdPicture

Public Sub SetData(Optional ByVal Info As String, _
                   Optional ByVal Picture As Variant)
```

```
   If Not IsMissing(Info) Then
      m_Info = Info
   End If

   If Not IsMissing(Picture) Then
      Set m_Picture = Picture
   End If

End Sub
```

Because a Visual Basic .NET object cannot be set to a value representing "missing," the *IsMissing* function is not available in Visual Basic .NET. The closest approximation is *IsNothing*. The Upgrade Wizard will replace all of your calls to *IsMissing* with calls to *IsNothing*.

Visual Basic .NET requires that you specify a default value for all optional parameters to be used if no argument is passed in. The wizard will automatically initialize parameters to reasonable initial values depending on the type; a string is set to an empty string, a numeric is set to 0, and an object parameter is set to *Nothing*.

Let's suppose that in the code just given you wanted to pass in *Nothing* as the value for the *Picture* parameter. Your intent is to erase any current picture. Inspect the following upgraded code, and you will see that the logic cannot distinguish between a missing parameter and a parameter for which you explicitly pass in *Nothing*. The *Picture* parameter will always be set to *Nothing* in either case. If you explicitly pass in *Nothing* to erase the current picture, the *Not IsNothing(Picture)* test will fail and the picture member (*m_Picture*) will not be set to *Nothing*; in other words, the picture will not be erased.

```
Private m_Info As String
Private m_Picture As System.Drawing.Image

Public Sub SetData(Optional ByVal Info As String = "", _
                   Optional ByVal Picture As Object = Nothing)

   'UPGRADE_NOTE: IsMissing() was changed to IsNothing().
   If Not IsNothing(Info) Then
      m_Info = Info
   End If

   'UPGRADE_NOTE: IsMissing() was changed to IsNothing().
   If Not IsNothing(Picture) Then
      m_Picture = Picture
   End If
End Sub
```

How should you modify your code to cope with this situation? You need to reintroduce the concept of *IsMissing* into your code. One approach is to set the initialization value to a unique value that will not be passed in under normal circumstances. For example, in the previous code, you could set the parameter initialization values to "*<Missing>*". If you see that a parameter value is set to "*<Missing>*", you can assume that the caller did not specify the argument when making the call. The logic will be compatible with the way it was before. If the values are not specified, the current value is left intact, and if the caller passes in a value—including *Nothing* or an empty string—the current value will be replaced by the passed-in value. Applying "*<Missing>*" to the parameter initialization values and fixing up the code to test for "*<Missing>*" yields the following:

```
Private m_Info As String
Private m_Picture As System.Drawing.Image

Public Sub SetData(Optional ByVal Info As String = "<Missing>", _
                   Optional ByVal Picture As Object = "<Missing>")
   If Info <> "<Missing>" Then
      m_Info = Info
   End If

   If Picture <> "<Missing>" Then
      m_Picture = Picture
   End If
End Sub
```

Dealing with *Null*

As we discussed in the previous section, an *Object* can be in only one of two states—set to a value or to *Nothing*. A Visual Basic *Variant*, on the other hand, can contain other states, such as empty, *Null*, or missing. We can work around the absence of the missing state by using a special value that you create to mean "missing." When dealing with *Null*, you do not need to create your own value. Rather, the .NET Framework provides the *System.DBNull.Value* object as the equivalent to a Visual Basic 6 *Null* value.

Why is the name of the value *DBNull.Value*? Why not just *Null*? The reason is that the *DB* in *DBNull* is short for *database*. Since it is most common to deal with null values when reading values from or writing values to database fields, *DBNull.Value* is intended to be used when you want to set or check for null values in a database table. When you are setting an object variable to a field contained in an ADO.NET dataset or to a field in a traditional ADO recordset, if the field value is *Null*, the object variable will be set to *System.DBNull.Value* by convention.

Null Propagation

Many Visual Basic 6 functions—such as *Left*, *Mid*, *Chr*, and *Hex*—deal with *Variant* values containing *Null*. These functions return a *Variant* containing *Null*. Returning a *Null* value in response to a passed-in *Null* value is known as **null propagation**. The null-propagating functions all accept *Variant* values and return *Variant* values. A set of sibling functions in the Visual Basic 6 language accepts strings and returns strings. These functions—such as *Left$*, *Mid$*, *Chr$*, and *Hex$*—will result in an error if you attempt to pass in a value of *Null*.

By supporting null propagation, Visual Basic 6 allows you to write working code that does not need to check for *Null*. For example, if you have code that converts a customer name—taken from a field in a database—to lowercase, the code will work even if the customer name is not provided and is *Null*. The following Visual Basic 6 code will execute without error even though *vCustomerMiddleName* is *Null*.

```
Dim vCustomerMiddleName As Variant
vCustomerMiddleName = Null
vCustomerMiddleName = LCase(vCustomerMiddleName)
```

Visual Basic .NET does not support null propagation. This means that although the same Visual Basic 6 functions—*Left*, *Mid*, *Chr*, *Hex*, and *LCase*—are available in Visual Basic .NET, the Visual Basic .NET functions do not accept *Null*—*System.DBNull.Value* to be exact—as a legal value. The Upgrade Wizard produces the following code from the previous Visual Basic 6 example:

```
Dim vCustomerMiddleName As Object
vCustomerMiddleName = System.DBNull.Value
vCustomerMiddleName = LCase(vCustomerMiddleName)
```

This code leads to a run-time exception: "No accessible overloaded 'Strings.LCase' can be called without a narrowing conversion." This message is an overly technical, wordy way of saying that the *LCase* function does not accept *System.DBNull.Value* as a legal argument.

How can you fix this problem? The obvious fix is to insert a check before the call to every Visual Basic .NET function to make sure that the passed-in argument is not *System.DBNull.Value*. We could modify the code as follows:

```
Dim vCustomerMiddleName As Object
vCustomerMiddleName = System.DBNull.Value
If Not vCustomerMiddleName Is System.DBNull.Value Then
   vCustomerMiddleName = LCase(vCustomerMiddleName)
End If
```

This fix will work great if you are making a couple of calls to *LCase* in your code, but what if you are making hundreds or thousands of calls to the function? Adding checks for *System.DBNull.Value* before all calls to your Visual Basic .NET functions will be too time-consuming. Another solution is to declare your own *LCase* function that handles null propagation. For example, you can create the *LCase* function in a module—where it can be accessed from anywhere in your code—as follows:

```
Public Function LCase(ByVal obj As Object) As Object
   If Not obj Is System.DBNull.Value Then
      ' Value is not null so call Visual Basic .NET function
      obj = Microsoft.VisualBasic.LCase(obj)
   End If
   Return obj
End Function
```

If you place this function in a module named Module1, it is a simple matter to search for and replace all your calls to *LCase* with ones to *Module1.LCase*.

Arrays

Visual Basic .NET supports two array types—strongly typed and generic *System.Array*. Strongly typed arrays are the arrays you are accustomed to using in Visual Basic 6. Anytime you use the *Dim* statement to declare an array variable of a specific type, you are creating a strongly typed array. Visual Basic .NET—actually the .NET Framework—offers a new *System.Array* class. This class offers a set of methods so that you can define an array of any type and any number of dimensions at run time. An important difference between the two array types is that strongly typed arrays do not support nonzero lower bounds, whereas the *System.Array* class does.

Because Visual Basic .NET offers primary support for strongly typed arrays, it is not possible to declare a strongly typed array with a nonzero lower bound. Visual Basic .NET in turn does not support nonzero-bound array-related features such as array declarations including the *To* statement and *Option Base 1*. It does support *LBound*, but when *LBound* is used on a strongly typed array, it will always return 0.

The Upgrade Wizard will upgrade your arrays to strongly typed Visual Basic .NET arrays. If you are using *Option Base 1*, the wizard will issue an UPGRADE_WARNING message: "Lower bound of array <*ArrayName*> was changed from 1 to 0." The UPGRADE_WARNING is added for each array declaration found that does not specify a lower bound. If you are using a positive value for the array's lower bound, the wizard will remove the lower bound from the declaration and issue the same UPGRADE_WARNING. If you specify a negative value for the lower bound in your array declaration, the wizard will

leave the declaration as is and will include an UPGRADE_ISSUE comment: "Declaration type not supported: Array with lower bound less than zero." The following code samples demonstrate how the wizard upgrades your Visual Basic 6 array declarations. The Visual Basic 6 code

```
Option Base 1
…
Dim OptionOneBasedArray(10) As Integer
Dim PositiveLBoundArray(10 To 20) As Long
Dim NegativeLBoundArray(-10 To 10) As Variant
```

upgrades to the following Visual Basic .NET code:

```
'UPGRADE_WARNING: Lower bound of array OptionOneBasedArray
'was changed from 1 to 0.
Dim OptionOneBasedArray(10) As Short

'UPGRADE_WARNING: Lower bound of array PositiveLBoundArray was
'changed from 10 to 0.
Dim PositiveLBoundArray(20) As Integer

'UPGRADE_ISSUE: Declaration type not supported: Array with
'lower bound less than zero.
Dim NegativeLBoundArray(-10 To 10) As Object
```

If you are using positive lower bounds in your array declaration, the upgraded declaration will compile and run. However, if your code uses hard-coded references to the lower bound of the array, the array will have more elements than you need or will use. For example, suppose that you had code that accessed each element of the *PositiveLBoundArray*, as follows:

```
Dim i As Integer
For i = 10 To 20
   PositiveLBoundArray(i) = i
Next
```

This code will work without any problem. The only issue is that there are now 10 elements of the array, 0 through 9, that are not being used.

If, in Visual Basic 6, you declared your array with a negative lower bound, the wizard will leave the declaration as is. The declaration will result in a compiler error in Visual Basic .NET, however. You will need to change your array to use a 0 lower bound, and you may need to adjust your code to work with the new array bound. For example, you could change the *NegativeLBoundArray* declaration to use a 0 bound and the same number of elements as follows:

```
Dim NegativeLBoundArray(20) As Object
```

You would then need to update your code that accesses the elements of *NegativeLBoundArray*. Suppose, for example, that you have code such as the following:

```
Dim i As Integer
For i = LBound(NegativeLBoundArray) To UBound(NegativeLBoundArray)
   NegativeLBoundArray(i) = i
Next
```

The *For…Next* loop is acceptable because you are using *LBound* and *UBound*, and so the code automatically picks up the new lower and upper bounds for the array. The value that is being set into the array, however, is incorrect. The intent was to seed the array with values from –10 through 10, but the upgraded code is now seeding the array with values from 0 through 20.

To fix this problem, you could introduce an adjustment factor into your code and apply it to any code that is using hard-coded values to access an array element. The example above uses a calculated value *i* based on the array index. We could adjust *i* by subtracting 10. If we define our adjustment value in a constant, the updated code is

```
Dim i As Integer
Const NegativeLBoundArray_Adjustment = 10
For i = LBound(NegativeLBoundArray) To UBound(NegativeLBoundArray)
   NegativeLBoundArray(i) = i - NegativeLBoundArray_Adjustment
Next
```

With the adjustment in place, the code once again seeds the array with values from –10 through 10.

There is another way to solve this problem. If it's easier for you to use negative array bounds, or simply any nonzero lower bound, consider using the .NET Framework *System.Array* class. This class includes a method called *CreateInstance*, which allows you to create a generic *System.Array* with nonzero lower bounds. If we start with the original Visual Basic 6 array declarations given earlier:

```
Dim OptionOneBasedArray(10) As Integer
Dim PositiveLBoundArray(10 To 20) As Long
Dim NegativeLBoundArray(-10 To 10) As Variant
```

we can convert, in Visual Basic .NET, the declarations to *System.Array* and include calls to *CreateInstance* to initialize the type and dimensions of the arrays as follows:

```
Dim OptionOneBasedArray As System.Array  ' 1 to 10 As Short
Dim PositiveLBoundArray As System.Array  ' 10 to 20 As Integer
```

```
Dim NegativeLBoundArray As System.Array  ' -10 to 10 As Object

' Arrays to define the array length and LBound for each dimension
' Since we are only dealing with single dimension arrays, we only
' need one element
Dim ArrLens(0) As Integer
Dim LBounds(0) As Integer

ArrLens(0) = 10
LBounds(0) = 1
OptionOneBasedArray = System.Array.CreateInstance(GetType(Short), _
   ArrLens, LBounds)

ArrLens(0) = 11
LBounds(0) = 10
PositiveLBoundArray = System.Array.CreateInstance(GetType(Integer), _
   ArrLens, LBounds)

ArrLens(0) = 21
LBounds(0) = -10
NegativeLBoundArray = System.Array.CreateInstance(GetType(Object), _
   ArrLens, LBounds)

Dim i As Integer
For i = LBound(NegativeLBoundArray) To UBound(NegativeLBoundArray)
   NegativeLBoundArray(i) = i
Next
```

A caveat when using *System.Array* is that you cannot assign a *System.Array* variable containing a nonzero lower bound to a strongly typed array. For example, the following code will result in a "Specified cast is not valid" exception. The exception occurs on the assignment of the strongly typed array variable *StrongTypeArray* to the *System.Array* variable *NegativeLBoundArray*.

```
Dim StrongTypeArray() As Object

StrongTypeArray = NegativeLBoundArray
```

Structures

You will not find the Visual Basic 6 *Type…End Type*, commonly referred to as a user-defined type, in Visual Basic .NET. Instead you will find *Structure…End Structure*, a superset of *Type…End Type*. Within a structure, you can define members of any type. In addition, you can define methods and properties. You will find that a Visual Basic .NET structure works much like a class. One difference is that you cannot create instances of a structure, and all structure members implicitly include the *Shared* attribute. This means that a structure can

access shared members of the class or module where it is defined, but it cannot access instance members.

Member Initialization

A difference between Visual Basic 6 and Visual Basic .NET user-defined types is that Visual Basic 6 supports fixed-length array declarations within a user-defined type. In Visual Basic .NET, however, you cannot specify the size of the array contained within a user-defined type. To work around this limitation, you can define a constructor for your user-defined type in which you initialize the array to a fixed size. Consider the following Visual Basic 6 code:

```
Private Type MyType
   MyStringArray(10) As String
End Type

Sub Main()
   Dim mt As MyType
   MsgBox (mt.MyStringArray(0))
End Sub
```

The Upgrade Wizard upgrades this code to

```
Private Structure MyType
   <VBFixedArray(10)> Dim MyStringArray() As String

   'UPGRADE_TODO: "Initialize" must be called to initialize
   'instances of this structure.
   Public Sub Initialize()
      ReDim MyStringArray(10)
   End Sub
End Structure

Public Sub Main()
   'UPGRADE_WARNING: Arrays in structure mt may need to be
   'initialized before they can be used.
   Dim mt As MyType
   MsgBox(mt.MyStringArray(0))
End Sub
```

The wizard has done a few things here to create the equivalent Visual Basic .NET code. It has added the *VBFixedArray* attribute to the structure, specifying the size of the array. This attribute is useful if you are passing the structure to a Visual Basic file function such as *Get* or *Put*. The Visual Basic file function will use the attribute to determine whether it needs to read or write additional header information associated with the array. Another change the wizard has made is to insert a public method within the structure called *Initialize*. The code contained within the *Initialize* method includes *ReDim* calls to initialize fixed-length arrays contained within the structure. Finally, the wizard has

inserted some comments to let you know that you should include a call to the *Initialize* method before attempting to use the structure.

If you run the code as upgraded, you will encounter a *NullReferenceException*, "Object reference not set to an instance of an object," on the following line:

```
MsgBox(mt.MyStringArray(0))
```

The reason is that the structure member *MyStringArray* does not contain any array elements. To initialize the array with elements, you need to call the *Initialize* method on the structure. The best place to call *Initialize* is right after you declare a structure variable. In the example above, you should call *Initialize* right after the declaration of *mt*. After you apply this change, the updated code will be as follows:

```
Private Structure MyType
   <VBFixedArray(10)> Dim MyStringArray() As String

   Public Sub Initialize()
      ReDim MyStringArray(10)
   End Sub
End Structure

Public Sub Main()
   Dim mt As MyType
   mt.Initialize()
   MsgBox(mt.MyStringArray(0))
End Sub
```

LSet

The *LSet* statement can be used in Visual Basic 6 to assign the contents of one user-defined type to another. The assignment will work even if the two user-defined types do not contain the same fields. *LSet* works by copying the memory for the source user-defined type to the target user-defined type.

An example of where you might use *LSet* is to provide two different ways to view or access the same data. For example, suppose you want to store a version number in a 32-bit integer. The version number consists of 4 parts, each 1 byte in length—major, minor, revision, and build version numbers. To simplify creating the 32-bit integer, you can create a user-defined type containing each part of the version number. For the purpose of storing the version in a 32-bit integer, you can use *LSet* to assign a user-defined type containing a 32-bit integer to the user-defined type containing the version parts. The following Visual Basic 6 code demonstrates how this can be done.

```
Type VersionPartsType
   BuildVer As Byte
   RevisionVer As Byte
   MinorVer As Byte
   MajorVer As Byte
End Type

Type VersionType
   Version As Long
End Type

Sub Main()
   Dim vp As VersionPartsType
   Dim vt As VersionType

   vp.MajorVer = 7
   vp.MinorVer = 0
   vp.RevisionVer = 1
   vp.BuildVer = 2

   LSet vt = vp
   MsgBox "32-bit version number is " & Hex(vt.Version)
End Sub
```

Visual Basic .NET does not support the *LSet* statement to assign one type to another. The Upgrade Wizard will place the following comment before calls to *LSet* in the upgraded code.

```
'UPGRADE_ISSUE: LSet cannot assign one type to another.
```

To solve this problem, you can create a copy function that you use to copy each member of one user-defined type to another user-defined type. In this case, you will need to write a copy function that combines four members of *VersionPartsType* into a single member of *VersionType*. Here is an example of a copy function you can use to take the place of the *LSet* statement:

```
Sub CopyVersion(ByVal vp As VersionPartsType, _
                ByRef vt As VersionType)

   ' Calculate the 32-bit version number by placing each version part
   ' within each byte of the 32-bit integer. Major version goes into
   ' the top byte and build goes into the lowest byte.
   vt.Version = vp.MajorVer * 2 ^ 24 + vp.MinorVer * 2 ^ 16 + _
      vp.RevisionVer * 2 ^ 8 + vp.BuildVer
End Sub
```

> **Note** Note The *LSet* statement plays two roles: it left-aligns a string within a string, and it also sets a type to another type. The first role, left-aligning a string within a string, is supported in Visual Basic .NET.

Making Your Code Thread-Safe

Threading issues may affect Visual Basic 6 ActiveX DLLs or *UserControl*s that you upgrade to Visual Basic .NET, especially since the Visual Basic Upgrade Wizard does not make your code thread-safe when upgrading it. You will need to make manual changes to your code—as explained later in this section—to make it thread-safe.

Such threading issues are uncommon, since components created in Visual Basic 6 are thread-safe by nature. Thread safety is accomplished in Visual Basic 6 by letting only a single thread access a component. Even if multiple threads are attempting to gain access to the component, all requests are synchronized through the single accessor thread associated with the component.

Visual Basic .NET components (including *UserControls*), on the other hand, are not thread-safe. Multiple threads are allowed to access the component simultaneously. There is no synchronization of threads on behalf of a Visual Basic .NET component. As a result, code that is thread-safe in Visual Basic 6 becomes unsafe after being upgraded to Visual Basic .NET. This change will affect you if you are running your component in a multithreaded environment such as Internet Explorer, Internet Information Services, or COM+. If you are not sure whether your component will be run in a multithreaded environment, it is always best to err on the side of caution and ensure that your component is thread-safe.

When more than one thread is executing your code, each thread has access to the same shared data: member variables and global variables. Allowing more than one thread to manipulate shared data at the same time can lead to problems. For example, if two threads are attempting to increment a count of items in a collection simultaneously, the count may be end up being incremented by 1 and not 2. This will happen if both threads obtain the current count at about the same time and add 1. Neither thread will see what the other thread is doing. We call this a synchronization problem, since it involves a situation in which only one thread at a time should be allowed to perform the operation.

For example, a synchronization problem can occur when one thread caches a global or member variable value in a local variable and performs a calculation on that variable. Let's take a look at the following Visual Basic .NET code.

```
Public Class GreedyBank

   Private m_Balance As Decimal

   Public Sub New()
      m_Balance = 20
   End Sub

   Public Function Withdraw(ByVal Amount As Decimal) As String

      'Protect against negative balance
      If m_Balance >= Amount Then
         m_Balance = m_Balance - Amount
      Else
         Return "The amount you requested will overdraw " & _
            "your account by $" & Amount - m_Balance
      End If

   End Function

End Class
```

This example is pretty straightforward. A client application calls *Withdraw*. *Withdraw* takes the current customer balance and deducts the requested withdrawal amount. To ensure that the customer's account does not become overdrawn by the transaction, the function checks to make sure that the customer has enough money to cover the withdrawal amount.

This works great when only one thread at a time is executing the function. However, you can run into problems if two threads are executing the function simultaneously.

Let's suppose that the first thread has executed up to the following line:

```
m_Balance = m_Balance - Amount
```

The customer's balance checked out, so the thread is ready to deduct the *Amount*. Assume that the customer has just enough money ($20) to cover the withdrawal request of $20. Just before the first thread executes the above statement, a second thread comes along, requesting an amount of $20 and completing the following statement:

```
If m_Balance >= Amount Then
```

The second thread also sees that the bank balance, $20, is sufficient to cover the withdrawal request and proceeds to the next line:

```
m_Balance = m_Balance - Amount
```

At this point, the customer is rewarded with an account overdraft and the fees that accompany it. The first thread subtracts the withdrawal, leaving a balance of $0. The second thread, having successfully bypassed the sufficient-balance-to-cover-withdraw-amount check, then executes the same line, leaving a net balance of –$20. Here code that on the surface appears to be perfectly legitimate leads to unanticipated results—although the customer would need to submit two withdrawal requests simultaneously to encounter the problem.

Using a *SyncLock* Block

To fix this synchronization problem, you need to make sure that only one thread at a time has access to your shared member or global variables. In other words, you need to **synchronize** access to your data, letting only one thread at time execute certain blocks of code that manipulate shared data. You can do this by using the new Visual Basic .NET keyword *SyncLock*.

A *SyncLock* block ensures that only one thread at a time is executing code within the block. In the earlier code example, we can fix our negative balance problem by placing a *SyncLock* around the contents of the *Withdraw* function:

```
SyncLock GetType(GreedyBank)

   'Protect against negative balance
   If m_Balance >= Amount Then
      m_Balance = m_Balance - Amount
   Else
      Return "The amount you requested will overdraw " & _
         "your account by $" & Amount - m_Balance
   End If

End SyncLock
```

If there is any chance that your Visual Basic .NET component or *UserControl* will run in a multithreaded environment, we highly recommend using *SyncLock* blocks to protect your shared data. Taking the extra time to do this work up front can save you from grief down the road.

Windows API

The foundation upon which every Windows application is built is the Windows API. Visual Basic applications are no exception. Visual Basic, through the language and forms package, abstracts the Windows API to a set of easy-to-use statements, components, and controls. In many cases, you can build a Visual Basic application without needing to call a Windows API function directly.

However, because the Visual Basic language and forms package does not represent every Windows API function available, Visual Basic supports calling Windows API functions directly. It has done so since version 1, enabling you to add capabilities to your application that cannot be implemented in the Visual Basic language. Visual Basic .NET carries this capability forward by enabling you to declare and call Windows API functions in the same way as before.

This section focuses on the language changes that affect the way you create *Declare* statements for API functions. It also shows how you can use classes contained in the .NET Framework to complement or replace the Windows API calls you make in your application.

Type Changes

Two changes that affect almost every Windows API function declaration involve the numeric types *Integer* and *Long*. In Visual Basic 6, an *Integer* is 16 bits and a *Long* is 32 bits. In Visual Basic .NET, an *Integer* is 32 bits and a *Long* is 64 bits. Visual Basic .NET adds a new type called *Short*, which, in terms of size, is the replacement for the Visual Basic 6 *Integer* type.

In Visual Basic .NET, when you create a new *Declare* statement for a Windows API function, you need to be mindful of this difference. Any parameter type or user-defined type member that was formerly a *Long* needs to be declared as *Integer*; any member formerly declared as *Integer* needs to be declared as *Short*.

Take, for example, the following Visual Basic 6 *Declare* statement, which contains a mix of *Integer* and *Long* parameters:

```
Declare Function GlobalGetAtomName Lib "kernel32" _
   Alias "GlobalGetAtomNameA" (ByVal nAtom As Integer, _
   ByVal lpBuffer As String, ByVal nSize As Long) _
   As Long
```

The equivalent Visual Basic .NET declaration is as follows:

```
Declare Function GlobalGetAtomName Lib "kernel32" _
   Alias "GlobalGetAtomNameA" (ByVal nAtom As Short, _
   ByVal lpBuffer As String, ByVal nSize As Integer) _
   As Integer
```

The Upgrade Wizard will automatically change all the variable declarations in your code to use the right size. You need to worry about the size of a type only when you are creating new *Declare* statements or modifying existing statements.

As Any No Longer Supported

Because in some cases you need to pass either a string or a numeric value to a Windows API function, Visual Basic 6 allows you to declare parameter types *As Any*. This declaration allows you to pass an argument of any type, and Visual Basic will pass the correct information when the call is made. Although this gives you greater flexibility, no type checking is performed on the argument when you call the API function. If you pass an incompatible argument type, the application may crash.

Visual Basic .NET does not support declaring Windows API parameters *As Any*. Instead, it allows you to declare the same API function multiple times, using different types for the same parameter.

The Windows API function *SendMessage* is an example of an instance in which you would use *As Any* in the API function declaration. Depending on the Windows message you are sending, the parameter types required for the message will vary. The WM_SETTEXT message requires you to pass a string, whereas the WM_GETTEXTLENGTH message requires you to pass 0, a numeric value, for the last parameter. In order to handle both of these messages, you create the Visual Basic 6 *Declare* statement for *SendMessage* as follows:

```
Private Declare Function SendMessage Lib "user32" _
Alias "SendMessageA" _
   (ByVal hwnd As Long, _
    ByVal wMsg As Long, _
    ByVal wParam As Long, _
    ByVal lParam As Any) As Long
```

The last parameter has been declared *As Any*. The following code is an example of setting the caption of the current form and then retrieving the length of the caption set:

```
Dim TextLen As Long
Const WM_SETTEXT = &HC
Const WM_GETTEXTLENGTH = &HE

SendMessage Me.hwnd, WM_SETTEXT, 0, "My text"
TextLen = SendMessage(Me.hwnd, WM_GETTEXTLENGTH, 0, 0&)
```

To implement the equivalent functionality in Visual Basic .NET, you create multiple *Declare* statements for the *SendMessage* function. In the case of the *SendMessage* API, you create two *Declare* statements, using two different types for the last parameter—*String* and *Integer*—as follows:

```
Declare Function SendMessage Lib "user32" Alias "SendMessageA" _
   (ByVal hwnd As Integer, _
    ByVal wMsg As Integer, _
```

(continued)

```
    ByVal wParam As Integer, _
    ByVal lParam As String) As Integer

Declare Function SendMessage Lib "user32" Alias "SendMessageA" _
   (ByVal hwnd As Integer, _
    ByVal wMsg As Integer, _
    ByVal wParam As Integer, _
    ByVal lParam As Integer) As Integer
```

> **Note** You also need to change the other parameter types from *Long* to *Integer*, as discussed in the previous section.

Although having to create multiple *Declare* statements for the same function is more work, you benefit by getting both the flexibility of using the same function name and type checking when the call is made.

AddressOf Changes

Certain API functions, such as *EnumWindows*, require a pointer to a callback function. When you call the API function, Windows calls the callback function you provided. In the case of *EnumWindows*, Windows will call the function for each top-level window contained on the screen.

Visual Basic 6 allows you to declare Windows API functions that take callback function pointers by declaring the function pointer parameter as *Long*, to represent a 32-bit pointer. You call the API function passing the subroutine to serve as the callback function, qualified by the *AddressOf* keyword.

Visual Basic .NET still supports the *AddressOf* keyword, but instead of returning a 32-bit integer, it returns a delegate. A **delegate** is a new type in Visual Basic .NET that allows you to declare pointers to functions or to class members. This means that you can still create *Declare* statements for Windows API functions that take a callback function pointer as a parameter. The difference is that instead of declaring the parameter type as a 32-bit integer, you need to declare the function parameter as a delegate type.

The following Visual Basic 6 code demonstrates how to use *AddressOf* to pass a pointer to a callback function:

```
Declare Function EnumWindows Lib "USER32" _
   (ByVal EnumWindowsProc As Long, _
    ByVal lParam As Long) As Boolean

Declare Function GetWindowText Lib "USER32" Alias "GetWindowTextA" _
```

```
   (ByVal hwnd As Long, ByVal TextBuffer As String, _
    ByVal TextBufferLen As Long) As Long

Dim TopLevelWindows As New Collection

Sub Main()

   Dim WindowCaption As Variant

   ' Enumerate all top-level windows, EnumWindowsCallback will
   ' be called
   EnumWindows AddressOf EnumWindowsCallback, 0

   ' Echo list of captions to Immediate window
   For Each WindowCaption In TopLevelWindows
      Debug.Print WindowCaption
   Next

End Sub

Function EnumWindowsCallback(ByVal hwnd As Long, _
ByVal lParam As Long) _
As Boolean
   Dim TextBuffer As String
   Dim ActualLen As Long

   ' Get window caption
   TextBuffer = Space(255)
   ActualLen = GetWindowText(hwnd, TextBuffer, Len(TextBuffer))
   TextBuffer = Left(TextBuffer, ActualLen)

   ' Put window caption into list
   If TextBuffer <> "" Then
      TopLevelWindows.Add TextBuffer
   End If

   ' Return True to keep enumerating
   EnumWindowsCallback = True

End Function
```

When you use the Visual Basic .NET Upgrade Wizard to upgrade the above code, you get the following result:

```
Declare Function EnumWindows Lib "USER32" _
```

(continued)

```
   (ByVal EnumWindowsProc As Integer, ByVal lParam As Integer) _
      As Boolean

Declare Function GetWindowText Lib "USER32" Alias "GetWindowTextA" _
   (ByVal hwnd As Integer, ByVal TextBuffer As String, _
    ByVal TextBufferLen As Integer) As Integer

Dim TopLevelWindows As New Collection

Public Sub Main()

   Dim WindowCaption As Object

   'UPGRADE_WARNING: Add a delegate for AddressOf EnumWindowsCallback
   EnumWindows(AddressOf EnumWindowsCallback, 0)

   For Each WindowCaption In TopLevelWindows
      System.Diagnostics.Debug.WriteLine(WindowCaption)
   Next WindowCaption

End Sub

Function EnumWindowsCallback(ByVal hwnd As Integer, _
                             ByVal lParam As Integer) As Boolean

   Dim TextBuffer As String
   Dim ActualLen As Integer

   TextBuffer = Space(255)
   ActualLen = GetWindowText(hwnd, TextBuffer, Len(TextBuffer))
   TextBuffer = Left(TextBuffer, ActualLen)

   If TextBuffer <> "" Then
      TopLevelWindows.Add(TextBuffer)
   End If

   ' Return True to keep enumerating
   EnumWindowsCallback = True

End Function
```

The upgraded code contains a single compiler error: "'AddressOf' expression cannot be converted to 'Integer' because 'Integer' is not a delegate type." The error occurs on the following statement:

```
EnumWindows(AddressOf EnumWindowsCallback, 0)
```

To fix the error, you need to change the *Declare* declaration for *EnumWindows* to accept a delegate type for the *EnumWindowsProc* parameter instead of an *Integer* type. The easiest way to create a delegate declaration for the *EnumWindowsCallback* function is to perform the following steps:

1. Copy and paste the subroutine declaration in your code that is to serve as the callback function.
2. Insert the *Delegate* keyword at the beginning of the declaration.
3. Change the function name by appending the word *Delegate* to it. The delegate name must be unique.

In the previous example, we can create the delegate declaration by copying and pasting the following function signature to the section of code containing the *Declare* statements:

```
Function EnumWindowsCallback(ByVal hwnd As Integer, _
                             ByVal lParam As Integer) As Boolean
```

Insert the keyword *Delegate* at the beginning of the function declaration given previously, and change the name by appending *Delegate* to the original function name—*EnumWindowsCallback* becomes *EnumWindowsCallbackDelegate*. You end up with a delegate declaration for a function with the form—name and parameters—of *EnumWindowsCallback*:

```
Delegate Function EnumWindowsCallbackDelegate(ByVal hwnd As Integer, _
   ByVal lParam As Integer) _
   As Boolean
```

Now that the delegate declaration for the callback function is in place, you need to change the parameter type for the *EnumWindows Declare* statement from *Integer* to the delegate type *EnumWindowsCallbackDelegate*, as follows:

```
Declare Function EnumWindows Lib "USER32" _
   (ByVal EnumWindowsProc As EnumWindowsCallbackDelegate, _
    ByVal lParam As Integer) As Boolean
```

The code will compile and run without error, producing the same results as the original Visual Basic 6 code.

Passing User-Defined Types to API Functions

When you pass a Visual Basic 6 user-defined type to an API function, Visual Basic passes a pointer to the memory containing the user-defined type; the API function sees the members of the user-defined type in the same order as they were declared in Visual Basic. This is not the case for Visual Basic .NET, however. If you declare a user-defined type, the order of the members is not guaranteed to be the order in which they appear in the code. The .NET runtime may reorganize the members of a user-defined type in a way that is most efficient for the user-defined type to be passed to a function. To guarantee that the members are passed exactly as declared in the code, you need to use marshalling attributes.

Marshaling Attributes

A **marshaling attribute** is an attribute that you attach to a structure member or subroutine parameter declaration to specify how the member or parameter should be stored in memory and passed to the function. For example, if you are calling an API function named *MyFunction*, written in C or C++, that takes a VARIANT_BOOL—a 2-byte Boolean type—parameter, you can use the *MarshalAs* attribute to specify the parameter type that the API function expects. Normally, a Boolean parameter is passed using 4 bytes, but if you include *UnmanagedType.VariantBool* as a parameter to the *MarshalAs* attribute, the parameter is marshaled—in other words, passed—as a 2-byte VARIANT_BOOL argument. The following code demonstrates how to apply the *MarshalAs* attribute to your API function declarations:

```
Declare Sub MyFunction Lib "MyLibrary.dll" _
   (<MarshalAs(UnmanagedType.VariantBool)> ByVal MyBool As Boolean)
```

The *MarshalAs* attribute is contained in the System.Runtime.InteropServices namespace. In order to call *MarshalAs* without qualification, you need to add an *Imports* statement at the top of the module for System.Runtime.InteropServices, as follows:

```
Imports System.Runtime.InteropServices
```

For structures passed to API functions, declare the structure using the *StructLayout* attribute to ensure compatibility with Visual Basic 6. The *StructLayout* attribute takes a number of parameters, but the two most significant for ensuring compatibility are *LayoutKind* and *CharSet*. To specify that the structure members should be passed in the order in which they are declared, set *LayoutKind* to *LayoutKind.Sequential*. To ensure that string parameters are marshaled as ANSI strings, set the *CharSet* member to *CharSet.Ansi*.

Suppose you are calling an API function named *MyFunction* in a DLL named *MyLibrary*, which takes as a parameter a structure called *MyStructure*. *MyStructure* contains a string and a fixed-length string type, declared in Visual Basic 6 as follows:

```
Type MyStructure
   MyStringMember As String
   MyFixedLenStringMember As String * 32
End Type
```

An approximately equivalent Visual Basic .NET declaration is

```
Structure MyStructure
   Dim MyStringMember As String
   Dim MyFixedLenStringMember As String
End Structure
```

The above declaration is approximate because it is missing attributes to make it fully equivalent with the Visual Basic 6 declaration. It is missing attributes on the structure to specify layout and is also missing an attribute on *MyFixedLenStringMember* to mark it as a fixed-length string.

Because you cannot guarantee that the .NET runtime will pass the string members in the order declared—for example, the .NET runtime may reorganize the structure in memory so that *MyFixedLenStringMember* comes first—you need to add marshalling attributes to the structure as follows:

```
<StructLayout(LayoutKind.Sequential, CharSet:=CharSet.Ansi)> _
Structure MyStructure
   Dim MyStringMember As String
   Dim MyFixedLenStringMember As String
End Structure
```

Visual Basic .NET does not natively support fixed-length strings. However, when you pass a structure containing a string to an API function, you can change the string member to a character array and include a *MarshalAs*

attribute to tell the .NET runtime to pass a string of a fixed size. To do this, declare the structure member—*MyFixedLenStringMember*—as a *Char* array instead of a *String*. Include the *UnmanagedType.ByValArray* and *SizeConst* arguments to the *MarshalAs* attribute. If the structure is going to be used with any of the Visual Basic file functions—for example, if the structure serves as a record in a random access file—you need to add the *VBFixedString* attribute. You include or omit the *MarshalAs* and *VBFixedString* attributes depending on how the structure is used in your code. In this case, since we are passing the structure to a Windows API function, you need to add the *MarshalAs* attribute to the *MyFixedLenStringMember* structure member but omit the *VBFixedString* attribute, as follows:

```
<StructLayout(LayoutKind.Sequential, CharSet:=CharSet.Ansi)> _
Structure MyStructure
   Dim MyStringMember As String
   'Note: Change MyFixedLenStringMember to Char array.  You must
   '      pad this member to 32 characters before calling the API
   '      function.

   MarshalAs(UnmanagedType.ByValArray, SizeConst:=32) _
   Dim MyFixedLenStringMember() As Char
End Structure
```

In addition to the *ByValArray* and *SizeConst* attributes, the code contains a comment to warn you that the *ByValArray* attribute requires you to pass a string padded out to exactly the size specified by *SizeConst*. This means that if you want to pass 3 characters you must append 29 spaces to the end of the string to create a buffer 32 characters in length. You can use the *FixedLength-String* class provided in the compatibility library to create strings that are automatically padded out to the desired size. For example, if you want to set the *MyFixedLenStringMember* to a 32-character fixed-length string containing the characters "123," here's how you would do it:

```
Dim ms As MyStructure
ms.MyFixedLenStringMember = New VB6.FixedLengthString(32, "123")
```

With both the *StructLayout* and *MarshalAs* attributes in place, the structure will be passed in memory to an API function in the same way as the equivalent Visual Basic 6 *Type…End Type* structure is passed.

Note Note The Upgrade Wizard incorrectly upgrades fixed-length strings contained in structures. Instead of declaring the string as a *Char* array, the wizard declares the upgraded fixed-length string as a *String*. The wizard applies the *ByValTStr* attribute instead of the *ByValArray* attribute. The upgraded code will compile and run. However, there is a subtle problem that you need to be aware of: the last character in the fixed-length string buffer is set to *Null* when passed to an API function. If, for example, you are passing a 5-character fixed-length string (as part of a structure to an API function) containing "12345", only "1234" will be passed. The last character, "5", is replaced with a *Null* character. To fix this problem, you need to change the member type from *String* to *Char* array and change the *MarshalAs* attribute from *ByValTStr* to *ByValArray*. Review your code to make sure you are using the *FixedLengthString* class to create strings that you assign to the member variable. Keep in mind that if the structure is not passed to an API function you can remove the *MarshalAs* attribute, and you don't need to review your code in this case.

ObjPtr and *StrPtr* Not Supported

Visual Basic 6 includes helper functions that return a 32-bit address for an object (*ObjPtr*) or a string (*StrPtr*). Visual Basic .NET does not support these functions. Moreover, it does not allow you to obtain the memory address for any type.

In cases where you must have control over underlying memory, the .NET Framework provides a pointer type known as *System.IntPtr*. The *System.Runtime.InteropServices.Marshal* class contains a number of functions that you can use to obtain a pointer type for a given type. For example, the functions *AllocHGlobal*, *GetComInterfaceForObject*, and *OffSetOf* all return pointers to memory. You can also use the *GCHandle* structure to obtain a handle to an element contained in Garbage Collector–managed memory. In addition, you can obtain a pointer to the memory by having the Garbage Collector pin the memory to a single memory location and not move it.

The following Visual Basic 6 example calls *CopyMemory* to copy a source string to a destination string. It uses *StrPtr* to obtain the memory addresses of the source and destination strings.

```
Declare Sub CopyMemory Lib "kernel32" Alias "RtlMoveMemory" _
   (ByVal lpDest As Long, ByVal lpSource As Long, _
    ByVal cbCopy As Long)

Sub Main()
   Dim sourceString As String, destString As String

   sourceString = "Hello"
   destString = "      World"

   CopyMemory ByVal StrPtr(destString), _
              ByVal StrPtr(sourceString), 10

   MsgBox destString
End Sub
```

Because there is no direct equivalent for *StrPtr* in Visual Basic .NET, you must use a different approach. To obtain a memory address to a string in Visual Basic .NET, you call *GCHandle.Alloc* to associate a handle with the string. Calling *AddrOfPinnedObject* on the handle allows you to obtain an *IntPtr* to the string buffer. Finally, you can obtain the memory address by calling *IntPtr.ToInt32* to get the address value. It's more work than using *StrPtr*, but it achieves the same result. Here is the equivalent Visual Basic .NET code for the Visual Basic 6 code just given:

```
Declare Sub CopyMemory Lib "kernel32" Alias "RtlMoveMemory" _
   (ByVal lpDest As Integer, ByVal lpSource As Integer, _
    ByVal cbCopy As Integer)

Public Sub Main()
   Dim sourceString As String, destString As String

   sourceString = "Hello"
   destString = "      World"

   Dim sourceGCHandle As GCHandle = _
      GCHandle.Alloc(sourceString, GCHandleType.Pinned)
   Dim sourceAddress As Integer = _
      sourceGCHandle.AddrOfPinnedObject.ToInt32

   Dim destGCHandle As GCHandle = _
      GCHandle.Alloc(destString, GCHandleType.Pinned)
```

```
    Dim destAddress As Integer = _
       destGCHandle.AddrOfPinnedObject.ToInt32

    CopyMemory(destAddress, sourceAddress, 10)
    sourceGCHandle.Free()
    destGCHandle.Free()

    MsgBox(destString)
End Sub
```

Conclusion

Visual Basic .NET embodies the spirit of the Basic language by supporting most of the Visual Basic 6 language as is and by extending the language to embrace new features such as inheritance, structured exception handling, and free threading. As this chapter has demonstrated, alternate solutions exist for each language construct that has been retired or changed in meaning. You can address each language change by adding new attributes, by creating helper functions or overloaded functions that are equivalent to the Visual Basic 6 functionality, or by using the .NET Framework to achieve equivalent behavior. Ultimately, you can take your upgraded code and make it behave as it did before. Once you have your upgraded application working, you can update it to take advantage of the expansive set of features offered by the Visual Basic .NET language and the .NET Framework.