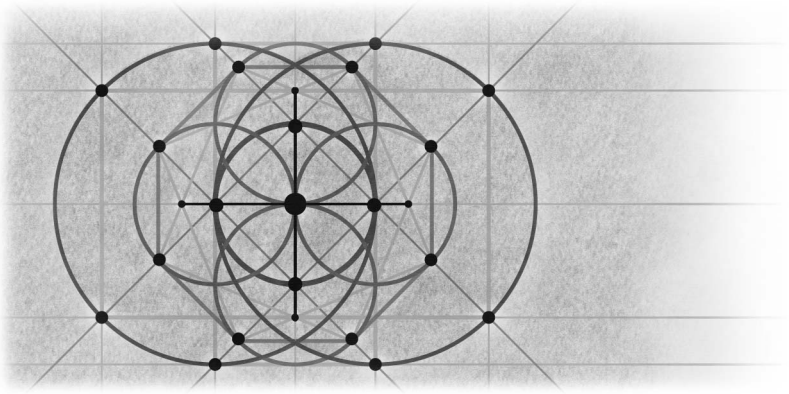


4

Preparing Your Project for the Upgrade to Visual Basic .NET

Now that you have decided to upgrade to Microsoft Visual Basic .NET, it's time to make the modifications necessary to smooth out the upgrade process. In doing so, you need to focus on issues that could get in the way of a straightforward upgrade. If you followed the guidelines in Chapter 3, you should already have a good idea of the potential problem areas in your application. This chapter will help you prepare for your upgrade project by identifying common coding practices that can cause upgrade issues, above and beyond the issues highlighted by the Upgrade Wizard's upgrade report.

Keep in mind that rather than addressing how to upgrade, this chapter focuses on identifying changes that you can make to existing applications to help the upgrade go more smoothly. There are two parts to this topic. First we'll discuss the use of up-to-date language and platform features. Then we'll look at good coding practices and common traps that Visual Basic developers fall into. Where necessary, examples are provided for clarity.

Why Change Anything?

The quality of your upgrade depends greatly on the quality of the application you start with. It is therefore crucial to take the time to make your application more compatible with the upgrade process. You'll need to replace obsolete language syntax and legacy (pre–Visual Basic 6) controls, modify your coding

style, and watch for common traps. These steps are the best way to ensure that your code maintains its current functionality throughout the upgrade. The following sections demonstrate what you can do to improve your application and make it more suitable to an upgrade.

Cleaning Up Legacy Code

Visual Basic 6 is almost a wonder of compatibility; it maintains code compatibility for many language features going all the way back to Visual Basic 1. These features include elements like *DefInt*, *VarPtr*, and *GoSub…Return* that still work, and work reasonably well. But just because they work does not mean that you should use them. These features and others are deprecated in Visual Basic 6, and they have been removed completely from Visual Basic .NET. This section covers the obsolete and deprecated Visual Basic features that you should remove from or replace in your Visual Basic 6 application before upgrading.

VarPtr, *DefInt*, and Other No-Shows

As we just mentioned, certain language elements are no longer supported in Visual Basic .NET. Some were eliminated because they lead to confusing and cryptic coding styles and are not appropriate for the kind of large-scale distributed application development that is becoming dominant in today's business environment. Others are just no longer meaningful in Visual Basic .NET. Take, for example, *VarPtr*. Pointers no longer have any meaning in the .NET world. In fact, pointers defeat the purpose of having a runtime environment like .NET in the first place. The runtime frees the developer from various tasks, and any Visual Basic language features that conflict with not only the runtime but also the goal of a scalable language for the enterprise have been eliminated.

These changes are intended to move the language (and the platform as a whole) forward in the most expeditious, and ultimately beneficial, way possible. The following Visual Basic language features are handled by the Upgrade Wizard but are not recommended for use in your applications:

- *DefInt*, *DefStr*, *DefObj*, *DefDbl*, *DefLng*, *DefBool*, *DefCur*, *DefSng*, *DefDec*, *DefByte*, *DefDate*, *DefVar*
- *GoTo* [*LineNum*]
- *Imp*, *Eqv*

The following features are simply not handled, and eliminating them is left as a task for the developer:

- *GoSub…Return*
- *LSet* (for user-defined types)
- *VarPtr*, *ObjPtr*, *StrPtr*
- *Null* and *Empty* nonzero-lower-bound arrays

Given these changes, where should you start? First you must search out areas in your application that use these language elements and either replace them, or leave them as they are and resolve the issues after you use the Upgrade Wizard. Both approaches are valid. It is often more efficient to fix a problem area before upgrading than to try to sort out the mess afterwards. There are times, however, when no alternatives exist in Visual Basic 6 and the most effective approach is to wait and solve the problem using Visual Basic .NET. This decision is ultimately one that you, the developer, need to make on a case-by-case basis. If you are unsure, it doesn't hurt to run the Upgrade Wizard just to see what it does, but that is a subject covered in a later chapter. (See Chapter 5)

DAO and RDO Data Binding

Over the years, Microsoft has introduced several data access technologies. Starting with Data Access Objects (DAO) and followed by Remote Data Objects (RDO), Microsoft started to bring data access to the masses with an accessible COM-based approach. The big breakthrough was the introduction of ActiveX Data Objects (ADO), with its flexible provider scheme. ADO was designed to provide a generic replacement for both DAO and RDO. It uses OLE DB to interface to various data providers, from comma-delimited text files to Microsoft Access databases to high-end database servers.

Visual Basic .NET introduces ADO.NET as the next generation of data access. Why does Microsoft continue to introduce new data access technologies? It reflects the fact that data access itself is evolving. Writing scalable Web sites is a different programming problem from querying a Microsoft Access database on the local machine. For this reason, ADO.NET's n-tier disconnected data access is very different from DAO data binding to an Access database.

You'll be pleased to know that Visual Basic .NET supports DAO, RDO, and ADO data access code. However, Visual Basic .NET does not support DAO or RDO data binding. What this means is that if your application has DAO or RDO data binding, you should either rewrite it as ADO data binding in Visual Basic 6 or rewrite it as ADO.NET data binding after upgrading the project to Visual Basic .NET.

Good Visual Basic 6 Coding Practices

Now that we have covered the deprecated features in Visual Basic 6, let's move on to identify the areas where programming practices can interfere with a smooth upgrade. You may need to adjust your code to make it more compatible with the Visual Basic Upgrade Wizard. Chapter 4 discusses how to use the Upgrade Wizard as a tool to identify and fix issues in your application. For now we will look at common problematic coding practices that you can correct prior to the actual upgrade. The Upgrade Wizard can identify most of these issues.

Variants and Variables

Visual Basic is an interesting language in that it has a default data type—the *Variant* type. What this means is that any variable, method parameter, or function return type that is not explicitly specified is considered a *Variant*. Although this default frees you from having to worry about types when coding—the *Variant* type can store all the intrinsic types as well as references to COM objects—using it is not the greatest programming practice.

One of the most significant impacts of relying on the *Variant* type is that you lose compile-time validation. Because Visual Basic cannot know what type you really wanted, it has to generate code to try to coerce your variable to the proper type at run time. It is not always successful, especially if the type you are using cannot be coerced to the expected data or object type.

Good programming practice aside, not explicitly declaring your variable types can generate a significant amount of upgrading work for the developer. This is unfortunate because given explicit types, the Upgrade Wizard can often upgrade variables and controls to their new Visual Basic .NET equivalents. If the meaning of your code is unclear, the Upgrade Wizard will not try to guess your intentions. It will simply leave the code as is and generate a warning. This leaves the task of upgrading the code exclusively to the developer. Why waste time, when the wizard will do the work for you?

Implicit vs. Explicit Variable Declaration

Look at the following code. Do you understand what is going on here?

```
Function MyFunction( var1, var2 )
   MyFunction = var1 + var2
End Function
```

Is this method doing string concatenation? Addition? If you can't be sure, the Upgrade Wizard certainly won't be. Testing the function shows that it is possible to use a variety of different inputs:

```
   Debug.Print MyFunction("This and", " that.")
   Debug.Print MyFunction(Now(), 1)
   Debug.Print MyFunction(4, 5)
```

The following results show in the Immediate window:

```
This and that.
7/23/2001 8:04:59 PM
 9
```

An alternative, and more proper, way to define the function would look like this:

```
Function MyFunction( var1 As String, var2 As String ) As String
   MyFunction = var1 & var2
End Function
```

or this:

```
Function MyFunction( var1 As Integer, var2 As Integer ) As Integer
   MyFunction = var1 + var2
End Function
```

With this explicit definition style, we can see exactly what the programmer intends to happen. We also get the benefits of compile-time validation: The compiler will generate an error if you pass an invalid type to the function (unless it is a *Variant*). In addition, the Upgrade Wizard will be able to generate the proper equivalent Visual Basic .NET code without a problem.

Abstraction

Abstraction is a fairly simple concept. The idea is to produce an implementation that separates the usage of an object, type, or constant from its underlying implementation. This concept is at the heart of every object-oriented programming environment. Abstraction insulates the developer from changes in the underlying implementation. When you violate this principle, you risk making your application more fragile and prone to errors. There are two common areas where Visual Basic developers get themselves into trouble: constants and underlying data types.

Constants and Underlying Values

Visual Basic 6 defines a whole host of constants. COM libraries make even more constants and enumerated types available through IntelliSense. The standard Visual Basic constants are really just pretty names that hide underlying integer values. Table 4-1 shows an excerpt from the Microsoft Developer Network (MSDN) documentation on mouse pointer constants.

Table 4-1 Mouse Pointer Constants

Constant	Underlying Value	Mouse Pointer
vbDefault	0	Default
vbArrow	1	Arrow
vbCrosshair	2	Cross
vbIbeam	3	I beam
vbIconPointer	4	Icon
vbSizePointer	5	Size
vbSizeNESW	6	Size NE, SW
vbSizeNS	7	Size N, S
vbSizeNWSE	8	Size NW, SE

You use these constants to change the mouse pointer, as in the following two examples:

```
Screen.MousePointer = vbIbeam
Screen.MousePointer = 3
```

These two examples achieve the same result. They are equivalent in that they both change the cursor to the I beam. Although they achieve the same result, the second line is hard to read. Unless you are intimately familiar with the *MousePointer* constants, it is hard to tell at a glance what result that line of code will actually produce. Using a named constant makes your code more readable and keeps the intent clear.

The Upgrade Wizard is unable to guess what you intended if you use an underlying value in place of a constant. When it encounters such a value, it leaves the code in place and inserts a special comment known as an upgrade warning (see Chapter 8 which you then have to resolve on your own. When constants are used properly, the Upgrade Wizard can upgrade code to the Visual Basic .NET equivalent (if it exists) with no warnings.

Use the Proper Constants A side issue to the notion of underlying values is that constant values can conflict. Remember that the Visual Basic standard constants are actually implemented with integers, and thus constants with unrelated function may have the same value. The problem is that you can often use constants interchangeably. While Visual Basic 6 doesn't really care—it has no notion of strict constant or enumerated type enforcement, and one integer is as good as another—the Upgrade Wizard will either change the constant definition to a possibly incompatible type in Visual Basic .NET or will not upgrade the offending line of code at all (leaving it in place with a warning).

The main lesson here is that when you use constants, make sure you are using them in the correct context. The following example illustrates this point:

```
Sub DefaultExample()
   Screen.MousePointer = vbIconPointer
   Screen.MousePointer = adCmdStoredProc
End Sub
```

Here *vbIconPointer* is defined as a *MousePointer* constant, and *adCmdStoredProc* is defined as an ADO *Command* object constant used for specifying stored procedures (totally unrelated to the *MousePointer* and not meaningful in this context). Both of these constants have a value of 4, and it is possible to use them interchangeably in Visual Basic 6. The Upgrade Wizard, however, will attempt to upgrade the constants to their .NET equivalent. Visual Basic .NET performs strict type checking for enumerated constants, and the upgraded code will not compile. Getting this right to start with will avoid this whole class of problems.

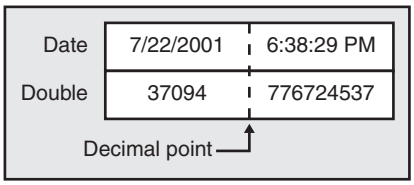
Underlying Data Types

Another problematic coding style involves the use of underlying (or implementation) data types instead of proper types. A case in point is the *Date* data type in Visual Basic 6. The *Date* data type is implemented as a *Double*. What this means is that you can coerce a *Date* to a *Double* and/or store the contents of a *Date* in a *Double*. Some developers may have used a *Double* directly instead of the *Date* data type. While this will work just fine in Visual Basic 6, it will cause problems for the Upgrade Wizard (and therefore for you) and will categorically not work in Visual Basic .NET. In the Microsoft .NET Framework resides a new

Date object, and the implementation is significantly different (and hidden from the developer). The new *Date* object supports much larger date ranges and higher-precision time storage than the Visual Basic 6 *Date* type. The only way to ensure that your code upgrades properly is to always use the *Date* data type and the date/time manipulation functions. Resist using the underlying *Double* value.

How to Get a Double Date

Date variables are stored as IEEE 64-bit (8-byte) floating-point numbers (which is the definition of a *Double*). The *Double* is capable of representing dates ranging from January 1, 100, to December 31, 9999, and times from 0:00:00 to 23:59:59. The following diagram shows how a *Date* is stored within said *Double*.



Try out the following example in Visual Basic 6:

```
Dim dt As Date
Dim dbl As Double

dbl = 37094.776724537
dt = dbl

Debug.Print dt
Debug.Print dbl
```

The Immediate window displays the following:

```
7/22/2001 6:38:29 PM
37094.776724537
```

You can adjust the date by adding or subtracting any *Integer* value to the double, and you can alter the time by adding or subtracting any *Decimal* value less than 1.

Taking advantage of implementation details rather than using the abstracted types defeats the purpose of abstraction in the first place. Abstraction is of significant use to any application because it allows you to change implementation details over time without requiring interface or coding changes. When you violate this rule of abstraction and take advantage of the underlying implementation, you risk having your application crumble to pieces when the implementation changes (as it is likely to do over time).

Early Binding vs. Late Binding vs. Soft Binding

Binding refers to the way that objects and methods are "bound" by the compiler. You can think of the different binding types in the context of variable declaration and usage. Early binding is the most explicit form of variable declaration and usage. Late binding is the least explicit form of variable declaration. Soft binding is somewhere in the middle. Confused yet? Read on.

Early Binding

Early binding refers to the use of strongly typed variables. **Strongly typed** means that each variable type is defined explicitly and that the *Variant* type is never used. Specifying the type of variables explicitly relieves Visual Basic of the task of second-guessing your work when you compile your application. It can distinguish the variable types and thus generate more optimized code (code that is faster and less resource intensive). It also enables the programmer to catch programming problems at compile time because the compiler will generate errors when nonexistent object methods are called or when parameter and return types aren't compatible.

Example of Early Binding The following code shows an example of the use of early binding. Notice that all of the function's parameters, the return type, and the local variables are explicitly typed. Nothing is explicitly or implicitly declared as *Variant*.

```
Function GetADORs(connStr As String, statement As String) _
   As ADODB.Recordset

   Dim conn As New ADODB.Connection
   conn.Open connStr
   Set GetADORs = conn.Execute(statement)
   conn.Close
End Function
```

Late Binding

Late binding occurs when the *Variant* type is used (either implicitly or explicitly). When you declare a variable as *Variant*, the compiler cannot know your exact intentions. The compiler then inserts additional logic to bind the method or property to the object during program execution. It becomes the responsibility of the runtime environment to catch any binding errors. Execution is less efficient because the runtime environment must resolve the variable types before it can perform an operation or method call. Take, for example, the following Visual Basic 6 method, which sets the *Caption* property value of a given label:

```
Sub SetLabel( lbl, value )
   lbl = value
End Sub
```

The compiler must insert code to evaluate whether *lbl* is an object and, if so, determine its default property. Furthermore, the runtime must check that the contents of *value* are valid for the default property of *lbl*. This step adds processing overhead to a single line of code. While the performance consequences are not obvious in a small application, large-scale enterprise or Web applications will notice the difference.

Late Binding and Upgrading In preparation for using the Upgrade Wizard, you should strongly type anything that you can. Review your code and explicitly specify function parameters and return types. Inspect every instance that uses a *Variant*, and ask yourself whether it is possible to use a strict data type instead. For example, consider the following function:

```
Public Function Test(frm) As Integer
   Dim lbl
   Set lbl = frm.Label1
   lbl.Caption = "This is a test"
   frm.Label2 = "Another Test"
   Test = True
End Function
```

This function implicitly takes a *Variant* parameter (intended to be a form) and explicitly returns an *Integer*. From the standpoint of good programming practice alone, it is important to ensure that your code clearly identifies the expected variable types. It becomes even more important with an application like the Visual Basic Upgrade Wizard. Look at what the Upgrade Wizard does with the code:

```
Public Function Test(ByRef frm As Object) As Short
   Dim lbl As Object
   lbl = frm.Label1
```

```
   lbl.Caption Label2 = "Another Test"
   'UPGRADE = "This is a test"
   frm. _WARNING: Boolean True is being converted into a numeric.
   Test = True
End Function
```

The following errors and warnings resulted:

- *lbl.Caption* should be *lbl.Text*.
- *frm.Label2* should be *frmLabel2.Text*.
- An upgrade warning occurred, indicating that the wizard was converting Boolean to numeric.

Part of the problem would have been avoided if *lbl* were explicitly declared as an instance of a Label control. Doing so would have ensured that the *Caption* property upgraded to *Text* in Visual Basic .NET.

Simple modifications to the code make the result of the Upgrade Wizard more predictable:

- Change *Test* to return *Boolean*.
- Change the definition of *lbl* to a variable of type *Label*.
- Explicitly type the *frm* parameter to be *Form1*.

The function looks like this after all of these changes:

```
Public Function Test(frm As Form1) As Boolean
   Dim lbl As Label
   Set lbl = frm.Label1
   lbl.Caption = "This is a test"
   frm.Label2 = "Another Test"
   Test = True
End Function
```

The upgraded function now looks like this:

```
Public Function Test(ByRef frm As Form1) As Boolean
   Dim lbl As System.Windows.Forms.Label
   lbl = frm.Label1
   lbl.Text = "This is a test"
   frm.Label2.Text = "Another Test"
   Test = True
End Function
```

The *Test* function has now upgraded cleanly with no errors.

The modifications to the original Visual Basic 6 code took very little work and also added a level of clarity to the original application. Granted, it is not always possible to make these modifications, but it is the ideal case.

Soft Binding

Soft binding is the last of the binding types and is often the most insidious. Imagine a situation involving a form (*MyForm*) with a label (*Label1*). Consider the following example, in which you pass the form and new text for your control:

```
Sub Form_Load()
   SetLabelText Me, "This is soft binding!"
End Sub

Sub SetLabelText( frm As Form, text As String )
   frm.Label1.Caption = text
End Sub
```

Notice that everything is strongly typed. There is certainly no obvious late binding going on, but something more subtle is happening. The parameter *frm* is declared as an object of type *Form*, while the form being passed to it is of type *MyForm*. The type *Form* does not have a property or a control called *Label1*. Visual Basic is implementing a form of late binding on a strongly typed variable. This is what we call **soft binding**. You have a couple of options for working around issues with soft binding:

```
Sub SetLabelText1( frm As MyForm, text As String )
   frm.Label1.Caption = text
End Sub
```

or this:

```
Sub SetLabelText2( frm As Form, text As String )
   Dim f as MyForm
   Set f  = frm
   frm.Label1.Caption = text
End Sub
```

SetLabelText1 is the preferred option because it will always prevent the wrong type of form from being passed to the function. *SetLabelText2*, while better than the original (and while it will upgrade properly), is not as robust because the assignment of *frm* to *f* could fail if the types are not compatible.

Watch Out for *Null* and *Empty*

Two keywords in Visual Basic 6, *Empty* and *Null*, could cause problems when you upgrade your application. *Empty* is typically used to indicate that a variable (including a *Variant*) has not been initialized. *Null* is specifically used to detect whether a *Variant* contains no valid data. If you ever test for either *Null* or *Empty* in your application, be sure to use the *IsNull* and *IsEmpty* functions, rather than the = operator. This is a minor change that should be easy to implement and will minimize upgrade headaches.

Another issue regarding *Null* involves the way in which various Visual Basic functions handle *Null* propagation. Take the following example:

```
Dim str As Variant
str = Null
str = Left(str, 4)
```

As you can see, a value of *Null* is passed to the *Left* function. The standard *Left* function in Visual Basic 6 is designed to propagate *Null* values. So the value of *str* after *Left* is called is *Null*. The problem is that in Visual Basic .NET, functions such as *Left* will not propagate *Null*. This disparity might introduce errors into your application without your realizing it. The *Left$* function provides a way around this problem:

```
Dim str As Variant
str = Null
str = Left$(str, 4)
```

Running this code will cause a run-time error, however (just as its Visual Basic .NET equivalent would). You can resolve the conflict (without actually testing for *Null*) by using the string concatenation operator:

```
Dim str As Variant
str = Null
str = Left$(str & "", 4)
```

Concatenation with an empty string will cause *Null* to be coerced to an empty string (but not a string that is *Empty*), and *Left$* will not cause a run-time error. Thus, you should use *Left$* (and all of the associated functions—*Right$*, *Mid$*, and so on) and ensure that your application behaves correctly before upgrading. If you do so, you should not have to worry about a difference in functionality because the behavior of functions such as *Left$* is equivalent to their Visual Basic .NET counterparts.

Implicit Object Instantiation

Many of you are probably familiar with the Visual Basic 6 *As New* syntax. It can be used to reduce coding by allowing the developer to write more compact code. Take, for instance, the following two examples:

```
' Example 1
   Dim c As ADODB.Connection
   Set c = New ADODB.Connection
   c.Open connStr
   Set c = Nothing
   c.Open connStr      ' Runtime error

' Example 2
   Dim c As New ADODB.Connection
   c.Open connStr
   Set c = Nothing
   c.Open connStr      ' No error
```

Example 2 shows how *As New* simplifies your code by taking care of the type declaration and object instantiation simultaneously. But the examples also demonstrate a behavioral difference between the two forms of variable declaration. The first example will produce an error; the second will not. What's going on here? Although declaring variables in this fashion can be considered equivalent, these examples are by no means equal. We were surprised to find out how *As New* had worked in the past. Looking at Example 2, you would think that the first line performs two tasks: creating the variable and instantiating the object. In fact, it does only the former. It turns out that Visual Basic 6 tries to be smart about creating the object. The code in Example 2 will not actually instantiate the object until the object is accessed, while Example 1 explicitly instantiates the object. Also in Example 1, when the connection object is destroyed by setting *c* to *Nothing*, the object is gone, and no object will be created in its place unless the application does so explicitly. In the second example, the connection object is created only when the *Open* method is called. The connection object is destroyed when *c* is set to *Nothing* but is created again (by the Visual Basic runtime) when *Open* is called the second time.

This issue is not a problem in Visual Basic 6. After all, even if you didn't know how it handles this situation, it would be safe to assume (as we did) that Visual Basic creates the object when told to. But it is important to note that Visual Basic .NET changes the behavior of *As New* to instantiate the object at the same time that the variable is defined. It also will not create new objects implicitly. If you set a reference to *Nothing*, you must create a new instance. Visual Basic .NET will not do it for you. This approach would seem to be the more logical (because it involves less uncertainty regarding object instantiation

and lifetimes), but it could have consequences for your code. Example 3 demonstrates the behavior of *As New* in Visual Basic .NET.

```
' Example 3
   Dim c As New ADODB.Connection()
   c.Open( connStr )
   c = Nothing
   c.Open( connStr )     ' Runtime Exception
```

If you are using this syntax and relying on behavior specific to Visual Basic 6, you need to change your approach. Everyone else can forget about this scenario and move on—move on, that is, to a discussion of another form of the *As New* syntax:

```
Dim forms(0) As New Form
forms(0).Title = "Form 1"
ReDim Preserve forms(2)
forms(1).Title = "Form 2"
forms(2).Title = "Form 3"
```

This is where matters get tricky. Notice that we need to create this array of controls only once. If we resize the array, each additional element acts like the first. Visual Basic 6 will create a new instance of *Form* when we access the *Title* property for any index of this array (provided, of course, that it falls within the bounds of the array). While this ability can be convenient, it can also cause problems. Imagine a situation in which a bug accidentally causes you to access another element (which might otherwise be empty). Visual Basic 6 will merrily create the object for you and at the same time hide a bug in your application. Depending on how your application is written, the bug may not be easily discovered.

To make a long story short, it is best not to define arrays in this way (especially in large-scale applications). Instead, you can use the following syntax to define the array in a way that is compatible with Visual Basic .NET:

```
Dim forms(0) As Form
Set forms(0) = New Form
forms(0).Title = "Form 1"
ReDim Preserve forms(2)
For i = 1 To 2
   Set forms(i) = New Form
   forms(i).Title = "Form " & i
Next
```

Let's look at the upgrade issue associated with this technique. Visual Basic .NET does support *As New*, but only for a single object declaration, in part because of one of the new features in Visual Basic .NET: construction. Every object in

Visual Basic .NET has a constructor of some sort. When you create an object, that constructor is called. (Standard object-oriented programming concepts work here.) COM objects in Visual Basic 6 did not support construction, and as a result the runtime could create objects without any real worry; you still needed to do the work of initializing the object (by opening a connection, creating a file, or whatever). In Visual Basic .NET, those kinds of assumptions cannot be made. Some objects require parameterized constructors, which further impedes the use of *As New* in the array case.

What all this means is that in Visual Basic .NET, you must explicitly declare your objects. *As New* thus becomes an explicit form of variable declaration and loses any implicit behaviors. If you want to create an array of objects, you must first create the array and then manually populate it with the objects you need. Although this does remove a shortcut from the toolbox of the Visual Basic programmer, it increases the clarity of your code. No object will exist until you create it, which makes for greater predictability all round.

Conclusion

As you have seen in this chapter, there are very definite and predictable steps that you can take to improve the chances that your application will upgrade smoothly. While some of these steps require work up front on your part, they will save you time and effort over the course of a complete upgrade. Abandoning deprecated features and improving code quality not only will ensure a smooth upgrade but will reduce the barrier between your application and other managed framework classes.

For additional information, you can consult the MSDN white paper "Preparing a Visual Basic 6.0 Application for Upgrading." You can find it by searching the Help files included with Visual Studio .NET.