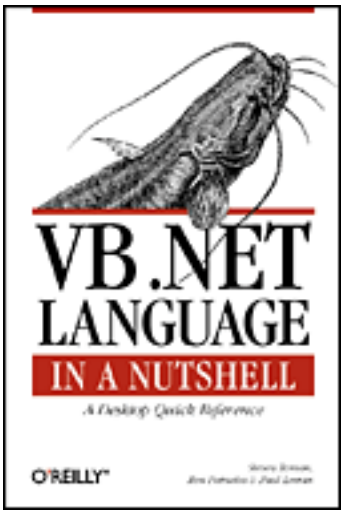

VB .NET Language in a Nutshell

By Steven Roman, Ron Petrusha & Paul Lomax
August 2001
0-596-00092-8, Order Number: 0928
654 pages, $34.95

Appendix A
What's New and Different in VB .NET

This appendix is for readers who are familiar with earlier versions of Visual Basic, specifically Version 6. In this appendix, we describe the basic changes to the VB language, both in syntax and in functionality. (Readers familiar only with Version 5 of Visual Basic will also benefit from this chapter, although we discuss only the changes since Version 6.)

We also touch upon other changes to VB, such as error handling and additional object-oriented programming support.

Language Changes for VB .NET

In this section, we outline the changes made to the Visual Basic language from Version 6 to Visual Basic .NET. These language changes were made to bring VB under the umbrella of the .NET Framework and allow a Common Language Runtime for all languages in Visual Studio .NET. In some sense, the changes made to the VB language were to bring the language component of VB (as opposed to the IDE component) more in line with the C# language (which is a derivative of C and C++).

Since we assume in this chapter that you are familiar with VB 6, we will not necessarily discuss how VB 6 handles a given language feature, unless the contrast is specifically helpful. You can assume that if a VB .NET language feature is described in this chapter, there has been a change in its behavior since VB 6.

Data Types

There have been fundamental changes to data types in VB .NET, which we outline in this section. The most important change is that all of the languages under the .NET umbrella (VB, C#, and Managed C++) now implement a subset of a *common set* of data types, defined in the .NET Framework's Base Class Library (BCL). We say *subset* because VB .NET does not implement all of these data types. In any case, each data type in the BCL is implemented either as a class or as a structure (which is similar to a class) and, as such, has members. The VB .NET data types are wrappers for the corresponding BCL data type. While this need not concern the VB programmer, it can be used in some cases to expose a bit more functionality from a data type. For more on data types, see Chapter 2.

Now let us consider the specifics.

Strings

As you may know, in VB 6, strings were implemented as a data type known as the BSTR. A BSTR is a pointer to a character array that is preceded by a 4-byte Long specifying the length of the array. In VB .NET, strings are implemented as objects of the String class, which is part of the .NET Framework's System namespace.

One consequence of this reimplementation of strings is that VB .NET does not have fixed-length strings, as does VB 6. Thus, the following code is illegal:

```
Dim Name As String * 30
```

Note, though, that strings in VB .NET are immutable. That is, although you do not have to declare a string's length in advance, once a value is assigned to a string, its length cannot change. If you change that string, the .NET Common Language Runtime actually gives you a reference to a new String object. (For more on this, see Chapter 2.)

Integer/Long data type changes

VB .NET defines the following signed-integer data types:

Short
: The 16-bit integer data type. It is the same as the Int16 data type in the Base Class Library.

Integer
: The 32-bit integer data type. It is the same as the Int32 data type in the Base Class

Library.

Long
: The 64-bit integer data type. It is the same as the Int64 data type in the Base Class Library.

Thus, with respect to the changes from VB 6 to VB .NET, we can say:

- The VB 6 Integer data type has become the VB .NET Short data type.
- The VB 6 Long data type has become the VB .NET Integer data type.

Variant data type

VB .NET does not support the Variant data type. The Object data type is VB .NET's *universal data type*, meaning that it can hold data of any other data type. According to the documentation, all of the functionality of the Variant data type is supplied by the Object data type.

We cannot resist the temptation to add that there are several penalties associated with using a universal data type, including poor performance and poor program readability. Thus, while VB .NET still provides this opportunity through the Object data type, its use is not recommended whenever it can be avoided.

The *VarType* function, which was used in VB 6 to determine the type of data stored in a variant variable (that is, the variant's data *subtype*), now reports the data subtype of the Object type instead. In addition, the *TypeName* function, which can be used to return a string that indicates the data type of a variable of type Object, is still supported.

Other data type changes

Here are some additional changes in data types:

- The `Deftype` statements (`DefBool`, `DefByte`, etc.), which were used to define the default data type for variables whose names began with particular letters of the alphabet, are not supported in VB .NET.
- The Currency data type is not supported in VB .NET. However, in VB .NET, the Decimal data type can handle more digits on both sides of the decimal point, and so it's a superior replacement. In VB .NET, Decimal is a strong data type; in VB 6, it was a Variant subtype, and a variable could be cast as a Decimal only by calling the *CDec* conversion function.

- In VB 6, a date is stored in a Double format using four bytes. In VB .NET, the Date data type is an 8-byte integer data type whose range of values is from January 1, 1 to December 31, 9999.

Variables and Their Declaration

The changes in variable declarations and related issues are described here.

Variable declaration

The syntax used to declare variables has changed for VB .NET, making it more flexible. Indeed, these are long awaited changes.

In VB .NET, when multiple variables are declared on the same line, if a variable is not declared with a type explicitly, then its type is that of the next variable with an explicit type declaration. Thus, in the line:

```
Dim x As Long, i, j, k As Integer, s As String
```

the variables *i*, *j*, and *k* have type Integer. (In VB 6, the variables *i* and *j* would have type Variant, and only the variable *k* would have type Integer.)

When declaring external procedures using the `Declare` statement, VB .NET does not support the `As Any` type declaration. All parameters must have a specific type declaration.

Variable initialization

VB .NET permits the initialization of variables in the same line as their declaration (at long last). Thus, we may write:

```
Dim x As Integer = 5
```

to declare an Integer variable and initialize its value to 5. Similarly, we can declare and initialize more than one variable on a single line:

```
Dim x As Integer = 6, y As Integer = 9
```

Variable scope changes

In VB 6, a variable that is declared *anywhere* in a procedure has *procedure scope*; that is,

the variable is visible to all code in the procedure.

In VB .NET, if a variable is defined inside a *code block* (a set of statements that is terminated by an `End...`, `Loop`, or `Next` statement), then the variable has *block-level scope*; that is, it is visible only within that block.

For example, consider the following VB .NET code:

```
Sub Test(  )
   If x <> 0 Then
      Dim rec As Integer
      rec = 1/x
   End If

   MsgBox CStr(rec)
End Sub
```

In this code, the variable *rec* is not recognized outside the block in which it is defined, so the final statement will produce an error.

It is important to note that the *lifetime* of a local variable is always that of the entire procedure, even if the variable's scope is block-level. This implies that if a block is entered more than once, a block-level variable will retain its value from the previous time the code block was executed.

Arrays and array declarations

VB 6 permitted you to define the lower bound of a specific array, as well as the default lower bound of arrays whose lower bound was not explicitly specified. In VB .NET, the lower bound of every array dimension is 0 and cannot be changed. The following examples show how to declare a one-dimensional array, with or without an explicit size and with or without initialization:

```
' Implicit constructor: No initial size and no initialization
Dim Days(  ) As Integer

' Explicit constructor: No initial size and no initialization
Dim Days(  ) As Integer = New Integer(  ) {}

' Implicit constructor: Initial size but no initialization
Dim Days(6) As Integer
```

```
' Explicit constructor: Initial size but no initialization
Dim Days(  ) As Integer = New Integer(6) {}

' Implicit constructor: Initial size implied by initialization
Dim Days(  ) As Integer = {1, 2, 3, 4, 5, 6, 7}

' Explicit constructor, Initial size and initialization
Dim Days(  ) As Integer = New Integer(6) {1, 2, 3, 4, 5, 6, 7}
```

Note that in the declaration:

```
Dim ArrayName(X) As ArrayType
```

the number X is the upper bound of the array. Thus, the array has size X+1.

Multidimensional arrays are declared similarly. For instance, the following example declares and initializes a two-dimensional array:

```
Dim X(,) As Integer = {{1, 2, 3}, {4, 5, 6}}
```

and the following code displays the contents of the array:

```
Debug.Write(X(0, 0))
Debug.Write(X(0, 1))
Debug.Writeline(X(0, 2))
Debug.Write(X(1, 0))
Debug.Write(X(1, 1))
Debug.Write(X(1, 2))

123
456
```

In VB .NET, all arrays are dynamic; there is no such thing as a fixed-size array. The declared size should be thought of simply as the initial size of the array, which is subject to change using the `ReDim` statement. Note, however, that the number of dimensions of an array cannot be changed.

Moreover, unlike VB 6, the `ReDim` statement cannot be used for array declaration, but only for array resizing. All arrays must be declared initially using a `Dim` (or equivalent) statement.

Structure/user-defined type declarations

In VB 6, a structure or user-defined type is declared using the `Type...End Type` structure.

In VB .NET, the `Type` statement is not supported. Structures are declared using the `Structure...End Structure` construct. Also, each member of the structure must be assigned an access modifier, which can be `Public`, `Protected`, `Friend`, `Protected Friend`, or `Private`. (The `Dim` keyword is equivalent to `Public` in this context.)

For instance, the VB 6 user-defined type:

```
Type RECT
   Left As Long
   Top As Long
   Right As Long
   Bottom As Long
End Type
```

is defined in VB .NET as:

```
Structure RECT
   Public Left As Long
   Public Top As Long
   Public Right As Long
   Public Bottom As Long
End Structure
```

Actually, the VB .NET `Structure` type is far more reaching than its VB 6 user-defined type predecessor. Indeed, structures have many properties in common with classes; for instance, structures can have members (properties and methods). We discuss structures in detail in Chapter 2.

Boolean and Bitwise Operators

`Eqv` and `Imp`, two infrequently used Boolean and bitwise operators that are present in VB6, have been removed from VB .NET.

In VB6, `Eqv` is the logical equivalence operator. As a Boolean operator, it returns `True` if both expressions are either `True` or `False`, but it returns `False` if one is `True` while the other is `False`. As a bitwise operator, it returns 1 if both bits are the same (that is, if both are 1 or both are 0), but it returns 0 if they are different. In VB .NET, `Eqv` can be replaced with the equals comparison operator for logical operations. However, for bitwise operations, you'll have to resort to a bit-by-bit comparison, as the following code fragment

shows:

```
Public Function BitwiseEqv(x1 As Byte, X2 As Byte) _
                           As Long

Dim b1, b2, bRet As Byte
Dim iCtr as Integer

For iCtr = 0 to len(x1) * 8 - 1
   b1 = x1 and 2^iCtr
   b2 = x2 and 2^iCtr
   if b1 = b2 then bRet += 2^iCtr
next

BitwiseEqv = bRet

End Function
```

In VB6, `Imp` is the logical implication operator. As a Boolean operator, it returns `True` unless its first expression is `True` while the second is `False`. As a bitwise operator, it returns 1 unless the bit in the first expression is 1 while the bit in the second expression is 0. In VB .NET, `Imp` can be replaced with a combination of the `Not` and `Or` operators for logical operations. For example, the code fragment:

```
bResult = (Not bFlag1) Or bFlag2
```

is equivalent to the VB6 statement:

```
bResult = bFlag1 Imp bFlag2
```

For bitwise operations, a bit-by-bit comparison is again necessary, as the following code fragment shows:

```
Public Function BitwiseImp(x1 As Byte, X2 As Byte) As Long

Dim b1, b2, bRet As Byte
Dim iCtr as Integer

For iCtr = 0 to len(x1)*8 - 1
   b1 = Not(x1) and 2^iCtr
   b2 = x2 and 2^iCtr
   if b1 Or b2 then
      bRet += 2^iCtr
```

```
   end If
next

BitwiseImp = bRet

End Function
```

Changes Related to Procedures

VB .NET features a number of changes to the way in which procedures are defined and called, most of which tend to make the language more streamlined and consistent.

Calling a procedure

In VB 6, parentheses are required around arguments when making function calls. When calling a subroutine, parentheses are required when using the `Call` statement and proscribed when not using the `Call` statement.

In VB .NET, parentheses are always required around a *nonempty* argument list in any procedure call--function or subroutine. (In subroutine calls, the `Call` statement is optional.) When calling a parameterless procedure, empty parentheses are optional.

Default Method of Passing Arguments

In VB 6, if the parameters to a function or subroutine were not explicitly prefaced with the ByVal or ByRef keywords, arguments were passed to that routine by reference, and modifications made to the argument in the function or subroutine were reflected in the variable's value once control returned to the calling routine. In VB .NET, on the other hand, if the ByRef or ByVal keyword is not used in a parameter, the argument is passed to the routine by value, and modifications made to the argument in the function or subroutine are discarded once control returns to the calling program.

Optional arguments

In VB 6, a procedure parameter can be declared as `Optional` without specifying a default value. For optional Variant parameters, the *IsMissing* function can be used to determine whether the parameter is present.

In VB .NET, an optional parameter must declare a default value, which is passed to the procedure if the calling program does not supply an argument for that parameter. The *IsMissing* function is not supported. The following example shows an optional parameter declaration:

```
Sub Calculate(Optional ByVal Switch As Boolean = False)
```

Return statement

In VB .NET, the `Return` statement is used to return control to the calling program from a function or subroutine. The `GoSub` statement is not supported. Note that the `Return` statement is used to return a value from a function.

The following function illustrates the `Return` statement:

```
Public Function Test(  ) As Integer
    If MsgBox("Return", MsgBoxStyle.YesNo) = MsgBoxResult.Yes Then
        Return 0
    Else
        MsgBox("Continue")
        Return 1
    End If
End Function
```

Passing property parameters in procedures

Consider passing a property to a procedure by reference, as in:

```
Sub ShrinkByHalf(ByRef lSize As Long)
   lSize = CLng(lSize/2)
End Sub

Call ShrinkByHalf(Text1.Height)
```

In VB 6, when passing the value of a property by reference, the property is *not* updated. In other words, passing a property by reference is equivalent to passing it by value. Hence, in the previous example, the property Text1.Height will not be changed.

In VB .NET, passing a property by reference does update the property, so in this case, the Text1.Height property will be changed. Note, however, that the value of the property is not changed immediately, but rather when the called procedure returns.

ParamArray parameters

In VB 6, if the `ParamArray` keyword is used on the last parameter of a procedure declaration, the parameter can accept an array of Variant parameters. In addition,

`ParamAarray` parameters are always passed by reference.

In VB .NET, `ParamArray` parameters are always passed by value, and the parameters in the array may be of any data type.

Miscellaneous Language Changes

VB .NET includes several miscellaneous changes that include the format of line numbers, the lack of support for the `GoTo` and `GoSub` statements, and the replacement of the `Wend` keyword by `End While`.

Line numbers

Visual Basic .NET requires that every line number be followed immediately by a colon (:). A statement can optionally follow the colon. In VB 6, line labels, which were used in particular for error handling by the `On Error GoTo` statement, had to be followed immediately by a colon, but line numbers did not.

On GoTo

The `On...GoSub` and `On...GoTo` constructions are not supported. However, VB .NET still supports the `On Error GoTo` statement.

While

The `While...Wend` construction loops through code while a specified condition is `True`. VB .NET retains that construction, but replaces the `Wend` keyword with the `End While` statement. The `Wend` keyword is not supported.

GoSub and Return statements

In VB .NET, the `GoSub` statement is not supported.

As remarked earlier, in VB .NET, the `Return` statement is used to return control to the *calling program* from a function or subroutine. The VB 6 `Exit Sub` and `Exit Function` statements continue to be supported in VB .NET; however, the advantage of the `Return` statement is that it allows you to specify the function's return value as an argument to the `Return` statement.

Changes to Programming Elements

VB .NET has removed support for several programming elements because the underlying .NET Framework class library and the Common Language Runtime (CLR) contain equivalent functionality. Here are the victims and their replacements. (We discuss the class library and CLR in Chapters and .)

Constants

The Microsoft.VisualBasic.Constants class in the Base Class Library defines a number of constants, such as the familiar `vbCrLf` constant, so these can be used as always. However, some constants, such as the color constants `vbRed` and `vbBlue`, are no longer directly supported. Indeed, the color constants are part of the System.Drawing namespace's `Color` structure, so they are accessed as follows:

```
Me.BackColor = System.Drawing.Color.BlanchedAlmond
```

In most cases, to access a particular constant that is not a field in the Microsoft.VisualBasic.Constants class, you must designate the enumeration (or structure) to which it belongs, along with the constant name. For example, the `vbYes` constant in VB 6 continues to exist as an intrinsic constant in VB .NET. However, it has a counterpart in the `MsgBoxResult` enumeration, which can be accessed as follows:

```
If MsgBoxResult.Yes = MsgBox("OK to proceed?", ...
```

For a list of all built-in constants and enums, see Appendix D.

String Functions

The *LSet* and *RSet* functions have been replaced by the *PadLeft* and *PadRight* methods of the System.String class. For instance, the following code pads the name "Donna" with spaces on the left to make the total string length equal to 15:

```
Dim sName As String = "Donna"
Msgbox(sName.PadLeft(15))
```

The *String* function has been removed from the language. In its place, we simply declare a string and initialize it, using syntax such as:

```
Dim str As New String("A"c, 5)
```

which will define a string containing five As. Note the use of the modifier `c` in `"A"c` to define a character (data type Char), as opposed to a String of length 1. This is discussed in

more detail in Chapter 2.

Emptiness

In VB 6, the `Empty` keyword indicates an uninitialized variable, and the `Null` keyword is used to indicate that a variable contains no valid data. VB .NET does not support either keyword, but uses the `Nothing` keyword in both of these cases.

According to the documentation: "`Null` is still a reserved word in Visual Basic .NET 7.0, even though it has no syntactical use. This helps avoid confusion with its former meanings." Whatever.

In addition, the *IsEmpty* function is not supported in VB .NET.

Graphical Functionality

The System.Drawing namespace contains classes that implement graphical methods. For instance, the Graphics class contains methods such as DrawEllipse and DrawLine. As a result, the VB 6 Circle and Line methods have been dropped.

Note that the VB 6 *PSet* and *Scale* methods are no longer supported and that there are no direct equivalents in the System.Drawing namespace.

Mathematical Functionality

Mathematical functions are implemented as members of the Math class of the System namespace. Thus, the VB 6 math functions, such as the trigonometric functions, have been dropped. Instead, we can use statements such as:

```
Math.Cos(1)
```

Note also that the *Round* function has been replaced by Round method of the System.Math class.

Diagnostics

The System.Diagonstics namespace provides classes related to programming diagnostics. Most notably, the VB 6 Debug object is gone, but its functionality is implemented in the System.Diagnostics.Debug class, which has methods such as Write, WriteLine (replacing Print), WriteIf, and WriteLineIf. (You won't believe it, but there is still no method to clear

the Output window!)

Miscellaneous

Here are a few additional changes to consider:

- The VB 6 *DoEvents* function has been replaced by the DoEvents method of the Application class of the System.Windows.Forms namespace.

- The VB 6 *IsNull* and *IsObject* functions have been replaced by the IsDBNull and IsReference methods of the Information class of the Microsoft.VisualBasic namespace. Since this namespace is implicitly loaded by VB as part of the project template when a project is created in Visual Studio, no `Imports` statement is required, and the members of its classes can be accessed without qualification.

- Several VB 6 functions have two versions: a String version and a Variant version. An example is provided by the *Trim$* and *Trim* functions. In VB .NET, these functions are replaced by a single overloaded function. Thus, for instance, we can call *Trim* using either a String or Object argument.

Obsolete Programming Elements

The following list shows some of the programming elements that have been removed from Visual Basic .NET:

`As Any`
: Required all parameters to have a declared data type.

Atn function
: Replaced by System.Math.Atan.

Calendar property
: Handled by classes in the System.Globalization namespace.

`Circle` statement
: Use methods in the System.Drawing namespace.

Currency data type
: Replaced by the Decimal data type.

Date function

Replaced by the Today property of the `DateTime` structure in the System namespace.

`Date` statement
: Replaced by the `Today` statement.

Debug.Assert method
: Replaced by the Assert method of the Debug class of the System.Diagonistics namespace.

Debug.Print method
: Replaced by the Write and WriteLine methods of the Debug class of the System.Diagonistics namespace.

`Deftype` statements
: Not supported.

DoEvents function
: Replaced by the DoEvents method of the Application class in System.Windows.Forms namespace.

`Empty` keyword
: Replaced by the `Nothing` keyword.

`Eqv` operator
: Use the equal sign.

`GoSub` statement
: Not supported.

`Imp` operator
: `A Imp B` is logically equivalent to `(Not A) Or B`.

Initialize event
: Replaced by the constructor method.

Instancing property
: Use the constructor to specify instancing.

IsEmpty function
: Not supported because the `Empty` keyword is not supported.

IsMissing function
: Not supported because every optional parameter must declare a default value.

IsNull function
: Not supported. The `Null` keyword is replaced by `Nothing`.

IsObject function
: Replaced by the *IsReference* function.

`Let` statement
: Not supported.

`Line` statement
: Use the DrawLine method of the Graphics class in the System.Drawing namespace.

`LSet` statement
: Use the PadLeft method of the String class in the System namespace.

`Null` keyword
: Use `Nothing`.

`On...GoSub` construction
: Not supported. No direct replacement.

`On...GoTo` construction
: Not supported. No direct replacement. `On Error...` is still supported, however.

`Option Base` statement
: Not supported. All arrays have lower bound equal to 0.

`Option Private Module` statement
: Use access modifiers in each individual `Module` statement.

`Property Get`, `Property Let`, and `Property Set` statements
: Replaced by a new unified syntax for defining properties.

PSet method
: Not supported. No direct replacement. See the System.Drawing namespace.

Round function
: Use the Round method of the Math class of the System namespace.

`RSet` statement
: Use the PadRight method of the String class in the System namespace.

Scale method
: Not supported. No direct replacement. See the System.Drawing namespace.

`Set` statement
: Not supported.

Sgn function
: Use Math.Sign.

Sqr function
: Use Math.Sqrt.

String function
: Use the String class constructor with parameters.

Terminate event
: Use the Destroy method.

Time function and statement
: Instead of the *Time* function, use the TimeOfDay method of the `DateTime` structure of the System namespace. Instead of the `Time` statement, use the `TimeOfDay` statement.

`Type` statement
: Use the `Structure` statement.

Variant data type
: Use the Object data type.

VarType function
: Use the *TypeName* function or the GetType method of the Object class.

`Wend` keyword
: Replaced by `End While`.

Structured Exception Handling

VB .NET has added a significant new technique for error handling. Along with the

traditional unstructured error handling through `On Error Goto` statements, VB .NET adds *structured exception handling*, using the `Try...Catch...Finally` syntax supported by other languages, such as C++. We discuss this in detail in Chapter 7.

Changes in Object-Orientation

As you may know, Visual Basic has implemented some features of object-oriented programming since Version 4. However, in terms of object-orientation, the step from Version 6 to VB .NET is very significant. Indeed, some people did not consider VB 6 (or earlier versions) to be a truly object-oriented programming language. Whatever your thoughts may have been on this matter, it seems clear that VB .NET is an object-oriented programming language by any reasonable definition of that term.

Here are the main changes in the direction of object-orientation. We discuss these issues in detail in Chapter 3.

Inheritance

VB .NET supports object-oriented inheritance (but not multiple inheritance). This means that a class can derive from another (base) class, thereby inheriting all of the properties, methods, and events of the base class. Since forms are also classes, inheritance applies to forms as well. This allows new forms to be created based on existing forms. We discuss inheritance in detail in Chapter 3.

Overloading

VB .NET supports a language feature known as *function overloading*. The idea is simple and yet quite useful. We can use the same name for different functions (or subroutines), as long as the functions can be distinguished by their *argument signature*. The argument signature of a function (or subroutine) is the sequence of data types of the arguments of the function. Thus, in order for two functions to have the same argument signature, they must have the same number of arguments, and the corresponding arguments must have the same data type. For example, the following declarations are legal in the same code module because they have different argument signatures:

```
Overloads Sub OpenFile(  )
    ' Ask user for file to open and open it
End Sub

Overloads Sub OpenFile(ByVal sFile As String)
    ' Open file sFile
End Sub
```

Object Creation

VB 6 supports a form of object creation called *implicit object creation*. If an object variable is declared using the `New` keyword:

```
Dim obj As New SomeClass
```

then the object is created the first time it is used in code. More specifically, the object variable is initially given the value `Nothing`, and then every time the variable is encountered during code execution, VB checks to see if the variable is `Nothing`. If so, the object is created at that time.

VB .NET does not support implicit object creation. If an object variable contains `Nothing` when it is encountered, it is left unchanged, and no object is created.

In VB .NET, we can create an object in the same statement as the object-variable declaration, as the following code shows:

```
Dim obj As SomeClass = New SomeClass
```

As a shorthand, we can also write:

```
Dim obj As New SomeClass
```

If the object's class constructor takes parameters, then they can be included, as in the following example:

```
Dim obj As SomeClass = New SomeClass(argument1, argument2,...)
```

As a shorthand, we can also write:

```
Dim obj As New SomeClass(argument1, argument2,...)
```

For details on class constructors, see Chapter 3.

Properties

There have been a few changes in how VB handles properties, particularly with respect to default properties and property declarations.

Default properties

As you know, you can use default properties in VB 6. For instance, if txt is a textbox control, then:

```
txt = "To be or not to be"
```

assigns the string "To be or not to be" to the default Text property of the textbox txt.

However, there is a price to pay for default properties: ambiguity. For example, if txt1 and txt2 are object variables referencing two TextBox controls, what does:

```
txt1 = txt2
```

mean? Are we equating the default properties or the object variables? In VB 6, this is interpreted as equating the default properties:

```
txt1.Text = txt2.Text
```

and we require the `Set` statement for object assignment:

```
Set txt1 = txt2
```

In VB .NET, default properties are not supported *unless* the property takes one or more parameters, in which case there is no ambiguity.

As Microsoft points out, default properties occur most commonly with collection classes. For example, in ActiveX Data Objects (ADO), the Fields collection of the Recordset object has a default Item property that returns a particular Field object. Thus, we can write:

```
rs.Fields.Item(1).Value
```

or, since the default Item property is parameterized:

```
rs.Fields(1).Value
```

Although we may not be used to thinking of this line as using default properties, it does.

Thus, in VB .NET, the line:

```
txt1 = txt2
```

is an *object* assignment. To equate the Text properties, we must write:

```
txt2.Text = txt1.Text
```

Since it is no longer needed, the `Set` keyword is not supported under VB .NET, nor is the companion `Let` keyword.

This settles the issue of equating object variables. For object variable *comparison*, however, we must use the `Is` operator, rather than the equal sign, as in:

```
If txt1 Is txt2 Then
```

or:

```
If Not (txt1 Is txt2) Then
```

Property declarations

In VB 6, properties are defined using `Property Let`, `Property Set`, and `Property Get` procedures. However, VB .NET uses a single property-declaration syntax of the form shown in the following example. Note also that there is no longer a need to distinguish between `Property Let` and `Property Set` because of the changes in default property support.

```
Property Salary(  ) As Decimal
    Get
        Salary = mdecSalary
    End Get
    Set
        mdecSalary = Value
    End Set
End Property
```

Note the use of the implicitly defined `Value` variable that holds the value being passed into the property procedure when it is being set.

Note also that VB .NET does not support `ByRef` property parameters. All property parameters are passed by value.

Back to: VB .NET Language in a Nutshell