VB6 vs VB.NET languages

This white paper includes a description of all the VB6 keywords, commands and objects that aren't available or that behave differently under VB.NET.

Contents

Unless otherwise stated, VB Migration Partner fully supports all the Visual Basic 6 features and keywords mentioned in this document. For more information, please read the manual and the knowledge base.

Language features

Integer data types

A VB6 Integer variable is a 16-bit signed integer, therefore it should be translated to **Short** (VB.NET) or **short** (C#); likewise, a VB6 Long variable is a 32-bit signed integer and should be translated to **Integer** (VB.NET) or **int** (C#). (A VB.NET or C# Long variable is a 64-bit signed integer.)

Currency data type

The Currency data type isn't directly supported by.NET; variables of this type should be converted to **Decimal** (VB.NET) or **decimal** (C#) Consider that the Decimal data type has greater precision and range than Currency, therefore you have no guarantee that math expressions deliver the same result they do in VB6. For example, in Currency operation might raise an overflow under VB6, but it would be evaluated correctly under .NET.

Variant data type

.NET doesn't support the Variant data type; all Variant members are translated to **Object** members. In many cases, however, the two types aren't equivalent. For example, an Object .NET variable can't hold the special Empty, Null, and Missing values.

Type declaration suffixes

A consequence of the fact that VB.NET redefines the meaning of Integer and Long keywords is that the meaning of type declaration suffixes has changed too. Now a variable whose name ends with "%" is a 32-bit integer; a variable whose name ends with "&" is a 64-bit integer; a variable whose name ends with "@" is a Decimal variable. However, it is recommended that you get rid of type declaration suffixes and convert them in standard and more readable As clauses.

Fixed-length strings

VB6 fields, local variables, and members of Type structures can be defined as fixed-length strings, as in:

```
Dim buffer As String * 256
```

An uninitialized fixed-length string initially contains only ASCII 0 characters; when you assign any value to it, the value is truncated to the maximum length, or padded with spaces to the right if shorted than the maximum length. VB.NET doesn't support fixed-length strings. The Microsoft.VisualBasic.Compatibility.dll assembly defines a **FixedLengthString** type which behaves like fixed-length strings, but there are significant differences with the original VB6 type.

VB Migration Partner maps fixed-length strings to the **VB6FixedString** type, which mimics VB6 behavior more closely:

```
Dim buffer As VB6FixedString(256)
```

Conversions between Date and Double types

VB6 allows you to use a Double variable whenever a Date value is expected, and vice versa. This is possible because a Date value is stored as a Double value whose integer portion represents the number of days elapsed since December 30, 1899 and whose fractional part represents the time portion of the date. When converting a piece of VB6 to VB.NET such implicit conversions can become explicit by calls to the **ToOADate** and **FromOADate** methods of the Date type:

```
Dim dat As Date, dbl As Double
…
dbl = dat.ToOADate()
dat = Date.FromOADate(dbl)
```

For readability's sake, VB Migration Partner generates calls to **DateToDouble6** and **DoubleToDate6** methods when the original VB6 code implicitly converts a Date value to Double, or vice versa.

Conversions between String and Byte arrays

VB6 supports implicit conversions from String to Byte arrays, and vice versa, as in this code snippet:

```
Dim s1 As String, s2 As String, b() As Byte
s1 = "abcde"
b = s1
s2 = b
```

VB.NET doesn't support such implicit conversions and requires explicit calls to methods of the **System.Text.Encoding** class:

```
Dim s1 As String, s2 As String, b() As Byte
```

```
s1 = "abcde"
b = Encoding.Unicode.GetBytes(s1)
s2 = Enconding.Unicode.GetString(b)
```

For uniformity and readability's sake, VB Migration Partner generates calls to **ByteArrayToString6** and **StringToByteArray6** methods when the original code implicitly converts a Byte array to a String, or vice versa.

Conversions from Boolean values

VB6 supports implicit conversions from Boolan values to other numeric data types. VB.NET requires that you use the appropriate conversion operator.

VB Migration Partner uses the **CByte** operator when converting to a Byte variable and **CShort** when converting to any other numeric data type.

VB.NET keywords

A few VB.NET keywords aren't reserved words under VB6 and can be used as member names. Examples are AddHandler, Handles, Shadows, and TimeSpan. When the name of a VB6 member matches a VB.NET keyword it must be enclosed between square brackets, as in

```
Dim [handles] As Integer
```

This is never a problem when converting to C#, which is case-sensitive and whose keywords are always lowercase

Block variables

If Dim keyword appears inside an If, For, For Each, Do, While, or Select block, then VB2005 limits the scope of the variable to the block itself whereas VB6 makes the variable visible to the entire method:

```
Sub Test(ByVal n As Integer)
    If n > 0 Then
        ' in VB.NET the x variable can be referenced only from by the code
        ' between the If and the Else keywords.
        Dim x As Integer
        …
    Else
        …
    End If
    …
End Sub
```

VB Migration Partner automatically moves the variable declaration outside the code block:

```
Sub Test(ByVal n As Short)
    Dim x As Short
    If n > 0 Then
        …
    Else
        …
    End If
    …
End Sub
```

Auto-instancing variables

VB6 variables declared with the "As New" clause are known as *auto-instancing* variables. The key property of these variables is lazy instantiation: the object referenced by the variable is created as soon as one of its members is referenced; if the variable is set to Nothing, the object is re-instantiated

the next time the variable is referenced. (A side-effect of this behavior is that testing such a variable against Nothing always returns False.) The .NET Framework has no notion of auto-instancing variables; in fact the following VB.NET statement

```
Dim w As New Widget
```

is just a shorthand for the following, more verbose, declaration:

```
Dim w As Widget = New Widget
```

or to the following C# statement

```
Widget w = New Widget();
```

where it is clear that the object is instantiated when it is declared. If you later set the variable to Nothing, no object is re-created when you reference the variable again.

By default, VB Migration Partner translates auto-instancing variables verbatim, therefore the original VB6 semantics is lost and runtime errors might occur in the converted program. However, it offers the ability to generate code that preserves the VB6 behavior and avoids subtle bugs or unexpected exceptions. You can enable this feature by means of the **AutoNew** pragma.

Auto-instancing arrays

In addition to individual auto-instancing variables, VB6 also supports auto-instancing arrays, as in the following statement:

```
Dim arr(10) As New Widget
```

Each element of such an array behaves like an auto-instancing variable. However, the “As New” clause is invalid for VB.NET arrays, therefore VB Migration Partner drops the “New” keyword and generate a regular array:

```
Dim arr(10) As Widget
```

It is up to the developer to correctly initialize all the elements in the array to avoid NullReference exceptions. However, if the application relies on the auto-instancing semantics, such a fix causes the VB.NET or C# application to behave differently from the original VB6 code. VB Migration Partner offers the ability to create a “true” auto-instancing array, by means of the **AutoNew** pragma.

Parameter default passing mechanism

Under VB6, method parameters are passed by-reference if the method parameter isn't explicitly declared with the ByVal keyword. Under VB.NET and C#, method parameters are passed by-value if the method parameter isn't explicitly declared with the ByRef keyword. Both VB Upgrade Wizard and VB Migration Partner correctly add an explicit ByRef keyword for parameters that don't specify a ByVal keyword.

Additionally, VB Migration Partner detects parameters that unnecessarily use ByRef and can optional convert them to ByVal parameters, if the **UseByVal** pragma is specified. (This pragma is implicitly applied when converting to C#).

Optional parameters

In VB6 you can include or omit the default value of an optional parameter; if you omit it, the default value for its type is assumed (0 for numeric types, "" for strings, Nothing for objects):

```
Sub Test(ByVal Optional x As Short, ByVal Optional y As String)
    ' ...
End Sub
```

VB.NET and C# require that the default value for a parameter be specified:

```
Sub Test(ByVal Optional x As Integer = 0, ByVal Optional y As String = "")
    ' ...
End Sub


void Test(int x = 0, string y = "")
{
}
```

ParamArray parameters

VB6 requires that ParamArray parameters be specified with an implicit ByRef keyword:

```
Sub Test(ParamArray arr() As Variant)
    ' ...
End Sub
```

By contrast, VB.NET requires that an explicit ByVal keyword be specified:

```
Sub Test(ByVal ParamArray arr() As Object)
    ' ...
End Sub
```

This detail makes the difference if the method modifies one of the elements of the parameter and some code relies on the fact that the argument is modified, as in this code:

```
Sub Increment(ParamArray arr() As Variant)
    Dim i As Integer
    For i = 0 To UBound(arr)
        arr(i) = arr(i) + 1
    Next
End Sub

Sub Main()
    Dim n As Variant
    n = 10
    Increment n
```

```
    Debug.Print n    ' => displays "11"
End Sub
```

After the migration to VB.NET, individual values passed to a ParamArray parameter are passed by value, therefore any change inside the method isn't propagated back to the caller.

```
Sub Increment(ByVal ParamArray arr() As Object)
    Dim i As Short
    For i = 0 To UBound(arr)
        arr(i) = arr(i) + 1
    Next
End Sub

Sub Main()
    Dim n As Object
    n = 10
    Increment(n)
    Debug.WriteLine(n)  ' => displays "10"
End Sub
```

VB Migration Partner copes with this issue in two ways. First, it makes you aware of the potential problem by emitting a warning if any element of a ParamArray vector is modified inside the method; second, it provides a pragma that allows you to generate an overload of the method that doesn't suffer from the issue.

Assignments between arrays

When you assign an array to another array variable under VB6, a *copy* of the source array is assigned to the target variable. If you later modify either the source or the target array, the other array isn't

affected. .NET arrays are reference types, therefore assignment between arrays are resolved internally by just assigning the target variable a *pointer* to the source array. If you later modify either array, the other array is modified.

```
Dim a(10) As Integer
Dim b() As Integer
b = a           ' copy the array
a(0) = 999      ' the modify the source array
MsgBox b(0)     ' displays '0' in VB6, '999' in VB.NET
```

VB Migration Partner solves this problem by invoking the destination array's **Clone** method:

```
targetArray = sourceArray.Clone()
```

Assignments between Structures

Both VB6's Type blocks and VB.NET **Structure** (or C# **struct**) blocks are value types; this implies that when you assign a Structure to a variable of same type, then a *copy* of the entire structure is assigned. If you later modify either the destination or the target Structure, then the other Structure isn't affected in any way. However, in .NET there's a caveat: if the Structure contains one or more arrays or fixed-length strings, then the target Structure shares a reference to these arrays or fixed-length strings. For example, consider the following code:

```
Structure TestUDT
    Public Names() As String
    Public Location As VB6FixedString
End Structure

Dim source As TestUDT
ReDim source.Names(10)
source.Location = "Italy"
' assignment to a variable of same type
```

```
Dim dest As TestUDT = source
' modify an array element in source UDT
source.Names(1) = "Code Architects"
' check that the value is modified also in the other UDT
Debug.WriteLine(dest.Names(1))          ' displays "Code Architects"
```

To have Structure assignments behave exactly as in VB6, if a Structure includes arrays or fixed-length strings then VB Migration Partner expands the Structure definition with a Clone method that returns a distinct copy of the Structure. All assignments between Structures of such types are then modified to call the Clone method:

```
' VB Migration Partner generates this code for UDT assignments
Dim dest As TestUDT = source.Clone()
```

Structure initialization

If you declare a VB6 Type variable, all the elements in the Type are correctly initialized; a VB.NET Structure can include neither a default constructor nor field initializers, therefore a Structure variable that has been just declared can have one or more uninitialized fields. For example, consider the following VB6 Type block:

```
Type TestUDT
    n As Integer
    s As String * 10
    a(10) As String
    w As New Widget
End Type
```

and now consider the corresponding VB.NET Structure, as generated by Microsoft Upgrade Wizard (and a competing tool that uses the UW's conversion engine):

```
Structure TestUDT
    Public n As Integer
```

```
    Public s As VBFixedString
    Public a() As String
    Public w As Widget

    Public Sub InitializeUDT()
        s = New VBFixedString(10)
        ReDim a(10)
        w = New Widget
    End Sub
End Structure
```

The InitializeUDT method is necessary because .NET Structures can't have constructors with zero arguments or field initializers. The Upgrade Wizard requires that you manually invoke the InitializeUDT method to ensure that a structure variable be correctly initialized before being used:

```
Dim udt As TestUDT
udt.InitializeUDT    ' you must insert this statement manually
```

VB Migration Partner frees you from the need to manually initialize the structure, because it generates a constructor with one (dummy) parameter and automatically invokes this constructor whenever the application defines a structure variable that requires this treatment:

```
Structure TestUDT
    Public n As Integer
    Public s As VBFixedString
    Public a() As String
    Public w As Widget
```

```
    Public Sub InitializeUDT()
        s = New VBFixedString(10)
        ReDim a(10)
        w = New Widget
    End Sub

    Public Sub New(ByVal dummy As Boolean)
        InitializeUDT()
    End Sub
End Structure

' this is the code generated when a TestUDT variable is declared
Dim udt As New TestUDT(True)
```

Method calls

VB.NET requires that the list of arguments passed to a Sub method be always enclosed in parenthesis; in VB6 only calls that return a value require that argument list be enclosed in parenthesis:

```
TestSub(12, "abc")
```

Late-bound method calls

VB.NET supports late-bound calls, but requires that the **Option Strict Off** directive be declared at the project-level or at the top of current file. VB Migration Partner declares Option Strict Off at the top of each file. After the migration process you should attempt to drop these statements where possible, adjusting the code in the file as necessary.

Fields passed by reference to a method

Consider the following VB6 code, inside the Widget class:

```
Public ID As String


Public Function GetString() As String
    Dim widget As New Widget
    widget.ID = "abcde"
    TestMethod widget.ID
    GetString = widget.ID
End Function


Sub Test(ByRef text As String)
    text = UCase(text)
    …
End Sub
```

Invoking the *GetString* method under VB6 delivers the result "abcde", which demonstrates that the ID field has been passed to Test method using by-value semantics even if the receiving *text* parameter is declared with the ByRef keyword. This behavior can be explained by knowing that a VB6 field is actually compiled as a Property Get/Let pair and therefore the *Test* method is actually receiving the result of the call to the "getter" block, not the actual field.

When this code is converted "as-is" to VB.NET, the ID field is uppercased on return from the Test method, which proves that VB.NET differs from VB6 in how fields are handled.

VB Migration Partner detects the potential bug and emits code that enforces the by-value semantics. The Upgrade Wizard and other migration tools convert the code as-is and don't emit a warning in this case.

Uninitialized local variables

If a local variable of reference type – that is, a String or Object variable – is declared and not immediately initialized, the VB.NET or C# compiler emits the following warning:

```
Variable 'varname' is used before it has been assigned a value.
A null reference exception could result at runtime.
```

To avoid this warning you should initialize the variable within the Dim statement:

```
Dim text As String = ""
Dim obj As Object = Nothing
```

VB Migration Partner automatically performs this initialization for you.

References to methods defined in modules

If code inside a VB6 form invokes a method defined in a BAS module and the base System.Windows.Forms.Form class exposes a public or protected method with same name, then a compilation error occurs (if the two methods have different syntax) or, worse, the program might not work as intended and possibly throw unexpected exceptions at runtime. For example, suppose that the following statements are located inside a VB6 form:

```
' both these methods are defined in Helpers.bas module
PerformLayout(False)
ProcessKeyDialog("x"c)
```

The *PerformLayout* method is exposed by the .NET System.Windows.Forms.Form class, but it has a different syntax and therefore it is marked under VB.NET as a compilation error. The *ProcessKeyDialog*

method is also exposed by the .NET Form class and it takes a character as an argument. Consequently, the form's *ProcessKeyDialog* method is invoked instead of the method defined in the Helpers.bas module, which surely causes a malfunctioning. In both cases, you can resolve the ambiguity by prefixing the method with the module's name (VB Migration Partner applies this fix):

```
Helpers.PerformLayout(False)
Helpers.ProcessKeyDialog("x")
```

VB Migration Partner detects method calls that are ambiguous and generates code that behaves as intended.

Event handlers

In VB6 a method that handles an event must follow the *object_eventname* naming convention. In VB.NET event handlers can have any name, provided that they are marked with an opportune **Handles** clause:

```
Private Sub NameClickHandler(ByVal sender As Object, ByVal e As EventArgs) _
    Handles txtName.Click
    ' handle click events originating from the txtName control
    ' ...
End Sub
```

Notice that the Handles clause for events raised by the form itself must reference the MyBase object:

```
Private Sub FormClickHandler(ByVal sender As Object, ByVal e As EventArgs) _
    Handles MyBase.Click
    ' handle click events originating from the current form
    ' ...
End Sub
```

Name collisions for Type…End Type blocks

In VB6 it is legal to have a private Type…End Type block with same name as a public class or as a Declare method defined elsewhere in the project. When the code is converted to VB.NET, the Structure that corresponds to the original Type must be renamed to avoid these name collisions.

Sub Main

When a Sub Main method is converted to VB.NET it should be decorated with an **STAThread** attribute:

```
<STAThread()> _
Public Sub Main()
    ' ...
End Sub
```

Member shadowing

A method, property, or event defined in a VB6 form might coincidentally have same name as a member exposed by the System.Windows.Forms.Form class, for example:

```
Sub PerformLayout(ByVal refresh As Boolean)
    ' ...
End Sub
```

When this code is converted and compiled under VB.NET a warning occurs, because the .NET Form class exposes a method named *PerformLayout*. To make the compiler happy you should add a Shadows keyword:

```
Shadows Sub PerformLayout(ByVal refresh As Boolean)
```

```
    ' ...
End Sub
```

VB Migration Partner automatically applies this fix when necessary.

Null propagation

VB6 Variant variables can hold the special Null value, but no corresponding value exists in VB.NET or C#. What makes matters worse is that several VB6 functions – namely Str, Hex, Oct, CurDir, Environ, Chr, LCase, UCase, Left, Mid, Right, Trim, RTrim, LTrim, and Space – support null propagation, as the following VB6 code demonstrates:

```
Dim var As Variant
var = Null
var = var + 10        ' var is assigned Null, no error is raised
var = Left(var, 1)    ' var is assigned Null, no error is raised
```

A Null value is neither True nor False, therefore when the test condition of an If block is evaluated as Null, then the Else blocks is always executed; prefixing the expression with the Not operator doesn't transform the expression into a non-Null value. You often need to manually fixing converted VB6 code that relies on the peculiar way in which VB6 deals with Null values.

A Null value is often the result of a read operation from a database field that contains the NULL value. When converted to VB.NET, the actual value stored in the variable is **DBNull.Value**, but the two values aren't equivalent. For example, an exception is thrown at runtime if the test condition in an If statement evaluates to DBNull.Value, whereas no runtime error occurs in VB6 if the test condition of an If statement evaluates to Null.

Enum member names

VB6 allows you to use virtually any character inside the name of an Enum member. If the name isn't a valid Visual Basic identifier you just need to enclose the name inside square brackets:

```
Public Enum Test
    [two words]          ' space
    [dollar$ symbol]     ' symbol and space
    [3D]                 ' leading digit
    [New]                ' language keyword
End Enum
```

VB.NET forbids Enum member's names that start with a digit or contain spaces or other symbols. VB.NET does support square brackets in Enum names, but they only allow to define names that match a language's keyword.

VB Migration Partner handles this situation by replacing invalid characters with underscores and using a leading underscore if the first character is a digit:

```
Public Enum Test
    two_words
    dollar__symbol]
    _3D
    [New]
End Enum
```

References to enum members

In VB6 the name of an enum member is considered as a global name. For example, you can reference members of the ColorConstants enum type with or without including the enum name:

```
txtName.BackColor = ColorContants.vbYellow
txtName.BackColor = vbYellow
```

In VB.NET and C# the name of the enum type can't be omitted, therefore only the first syntax form is valid.

Date variables in For…Next loops

VB6 supports Date variables as controlling variables in For…Next loops:

```
Dim d As Date
For d = Start To Start + 10
    ' …
Next
```

Date variables cannot be used as controlling variables in VB.NET or C# For blocks, therefore VB Migration Partner converts the above code by using an "alias" variable of type Double:

```
Dim d As Date
For d_Alias As Double = DateToDouble6(Start) To DateToDouble6(Start + 10)
    d = DoubleToDate6(d_Alias)
    ' …
Next
```

Multi-dimensional arrays in For Each…Next loops

There is a minor difference in how elements of a multi-dimensional array are accessed when the array appears in a For Each…Next loop:

```
Dim arr(10, 20) As Double
Dim v As Variant
For Each v In arr
    …
Next
```

Under VB6, elements are accessed in column-wise order – that is, first all the elements of first column, then all elements in second column, and so forth. Conversely, under VB.NET the elements are accessed in row-wise fashion – that is, first all the elements of the first row, then all the elements of second row, and so forth. If visiting order is significant, the same loop delivers different results after the migration to VB.NET.

VB Migration Partner emits a warning when a multi-dimensional array appears in a For Each…Next block. If you believe that preserving the original visiting order is important, you can insert a call to the **TransposeArray6** method (defined in VBMigrationPartner_Support module), which transposes array elements so that the migrated code works as the original one:

```
For Each v In TransposeArray6(arr)
    …
Next
```

File operations with UDTs

VB6 and VB.NET greatly differ in how UDTs - Structures in VB.NET parlance – are read from or written to files. Not only are structure elements stored to file in a different format, but the two languages also manage the End-of-File condition in a different way. In VB6 you can read a UDT even if the operation would move the file pointer beyond the current file's length; in VB.NET such an operation would cause an exception.

VB Migration Partner correctly handles both these problems, by generating code that invoke alternate file-handling methods, such as **FileGet6** and **FilePut6**.

Collections can't be modified inside For Each…Next loops

VB6 allows you to modify a collection inside a For Each…Next loop that iterates on the collection itself. For example, the following code works well and is indeed quite common in VB6:

```
Dim frm As Form
For Each frm In Forms
    Unload frm
Next
```

This code throws an exception under VB.NET, because unloading a form causes the Forms collection to change inside the loop. The simplest way to work around this problem is having the loop iterate on a copy of the collection, as in this example:

```
Dim values As New List(Of String)
…
For Each item As String In New List(Of String)(values)
    …
Next
```

VB Migration Partner doesn't generate this fix, because in the most general case it is impossible to automatically detect whether the collection is indirectly modified by any method call inside the loop.

DAO.DBEngine object

The DAO.DBEngine object is a global object, which means that VB6 application can reference its members without having to instantiate it first and that it isn't necessary to include the class name in the method call. In practice, this means that the following VB6 method calls to the OpenDatabase method are both valid and have the same effects:

```
Dim db1 As DAO.Database, db2 As DAO.Database
Set db1 = DBEngine.OpenDatabase("biblio.mdb")
Set db2 = OpenDatabase("northwind.mdb")
```

VB.NET doesn't support global objects, therefore the above statements must be converted as follows:

```
Dim dbeng As New DAO.DBEngine
Dim db1 As DAO.Database, db2 As DAO.Database
Set db1 = dbeng.OpenDatabase("biblio.mdb")
Set db2 = dbeng.OpenDatabase("northwind.mdb")
```

ByVal keyword in method calls

VB6 allows you to pass an argument to a by-reference parameter using by-value semantics, by prefixing the argument with the ByVal keyword:

```
' the 'address' variable is passed by value
CopyMethod ByVal address, arr(0), 1024
```

This calling syntax can be used only with Declare methods, and is especially useful with methods whose arguments are declared As Any, because you can't use the ByVal keyword in the declaration of "As Any" parameters. VB.NET supports neither the ByVal keyword in method call nor As Any parameters in Declare statements.

VB Migration Partner accounts for such ByVal keywords and generates the corresponding overload for the Declare method.

Declare statements pointing to Visual Basic runtime

Expert VB6 developers can invoke methods defined in VB6 runtime, for example to retrieve information about variables and arrays. Code that uses methods defined in the VB6 runtime can't be migrated to VB.NET, both because the MSVBVM60.DLL library isn't available and because.NET variables and arrays are stored differently from VB6 and therefore the methods wouldn't work anyway because.

VB Migration Partner flags Declare statements that point to VB6 runtime with an appropriate warning.

Resource files

VB6 resource files can't be used under VB.NET and should be converted separately. In addition to convert files to the .NET Framework format, VB.NET applications must be able to reference resources as **My.Resources.*Xxxx*** elements.

Image format tests

VB6 developers can test the type of an image by testing the picture's Type property against the values of the PictureTypeConstants enumerated type, as in:

```
If Picture1.Picture.Type = PictureTypeConstants.vbPicTypeEMetafile Then …
```

The .NET Image type doesn't expose the Type property, but has an equivalent property named **RawFormat**. Here's how VB Migration Partner translates previous code:

```
If Picture1.Picture.RawFormat is System.Drawing.Imaging.ImageFormat.Emf Then …
```

Remarks starting with three apostrophes

Many developers like to emphasize comments by creating lines of asterisks, dashes, or apostrophes:

```
'''''''''''''''''''''''
' Methods
'''''''''''''''''''''''
```

The problem in converting this code is that any remark that starts with three apostrophes is considered as an XML comment under VB.NET, therefore this code causes a compilation warning under VB.NET.

VB Migration Partner recognizes the problem and replaces the third apostrophe with a space:

```
'' '''''''''''''''''''''
' Methods
'' '''''''''''''''''''''
```

Keywords

#Const

Both VB6 and VB.NET support this language directive; however, VB6 supports and correctly evaluates the following functions: **Abs, Sgn, Len**, and **LenB**. VB.NET and C# don't support these functions in compile-time expressions. (In this case VB Migration Partner issues a warning.)

#If, #ElseIf, #Else, #End If

These compiler directives are supported under VB.NET and C#; however, both the Upgrade Wizard and VB Migration Partner convert only the code inside the #If, #ElseIf, or #Else section whose condition is evaluated as "true"; VB6 code in other sections isn't converted.

VB6 and VB.NET/C# languages evaluate #If and #ElseIf conditions, but differ for two minor details. First, the VB6 test expression can contain the following functions: **Abs, Sgn, Len**, and **LenB,** whereas VB.NET and C# don't support these functions in compile-time expressions. (In this case VB Migration Partner issues a warning.). Second, all string comparisons are carried out in case-insensitive mode under VB6 and in case-sensitive mode under VB.NET.

Abs

The Abs keyword isn't implemented in Microsoft.VisualBasic.dll assembly. You must replace it with a reference to the **Math.Abs** method (in System namespace)

```
result = Math.Abs(value)
```

AddressOf

VB.NET supports the AddressOf keyword, but only when the application defines a delegate class that can point to the target method. To understand what this means, say that you have the following VB6 code:

```
Private Declare Function SetWindowLong Lib "user32" Alias "SetWindowLongA" ( _
    ByVal hWnd As Long, ByVal ndx As Long, ByVal newValue As Long) As Long

Sub StartSubclassing(ByVal hWnd As Long)
    oldProcAddr = SetWindowLong(hWnd, GWL_WNDPROC, AddressOf WndProc)
End Sub

Function WndProc(ByVal hWnd As Long, ByVal uMsg As Long, _
    ByVal wParam As Long, ByVal lParam As Long) As Long
    ' …
End Function
```

To have this code to compile correctly, you must define a delegate class that can point to the WndProc method and use this delegate in the definition of the SetWindowLong API method:

```
Public Delegate Function SetWindowLong_CBK(ByVal hWnd As Integer, _
    ByVal uMsg As Integer, ByVal wParam As Integer, _
    ByVal lParam As Integer) As Integer

Private Declare Function SetWindowLong Lib "user32" Alias "SetWindowLongA" _
    (ByVal hWnd As Integer, ByVal ndx As Integer, _
    ByVal newValue As SetWindowLong_CBK)

Public Sub StartSubclassing(ByVal hWnd As Integer)
    oldProcAddr = SetWindowLong(hWnd, GWL_WNDPROC, AddressOf WndProc)
End Sub
```

Notice that this mechanism works because VB.NET allows you to omit the name of the delegate in the AddressOf expression. In fact, the complete – and arguably, more readable – version of the code inside the StartSubclassing method is as follows:

```
oldProcAddr = SetWindowLong(hWnd, GWL_WNDPROC, _
    New SetWindowLong_CBK(AddressOf WndProc))
```

VB Migration Partner correctly generates the delegate definition and creates the overload for the Windows API method that takes the function pointer as an argument.

AppActivate

The VB6 version of the AppActivate method takes a second (optional) *wait* argument; if this argument is True than the method waits until the external application receives the input focus. The VB.NET version of this method takes only one argument.

VB Migration Partner provides a special **AppActivate6** method that takes two arguments and behave like the VB6 method.

Array

VB.NET lacks the Array method, which VB6 developers can use to create a Variant array on the fly.

VB Migration Partner defines a replacement method named **Array6**, whose source code follows:

```
Public Function Array6(ByVal ParamArray args() As Object) As Object
    Return args
End Function
```

AscB

VB.NET doesn't support "byte-oriented" string methods.

VB Migration Partner provides the **AscB6** replacement method, which approximates the original VB6 method's behavior but isn't guaranteed to work well in all circumstances. This replacement method is marked as obsolete and methods invocations are flagged with a migration warning.

Atn

The Atn keyword isn't implemented in Microsoft.VisualBasic.dll assembly. You must replace it with a reference to the **Math.ATan** method (in System namespace)

```
result = Math.ATan(value)
```

Calendar

VB.NET doesn't support the Calendar property.

VB Migration Partner defines a dummy replacement method named **Calendar6** method that always returns zero and throws if a nonzero value is assigned.

CDate, IsDate

VB.NET version of CDate and IsDate methods is less forgiving than the corresponding VB6 method. Under VB6 these methods attempt to reverse the day-month numbers if they don't make up a valid date. For example, under VB6 both statements assign the #11/23/2008# date constant to the target variable:

```
dat = CDate("11/23/2008")    ' dd/mm/yyyy format is assumed
dat = CDate("23/11/2008")    ' mm/dd/yyyy format is assumed
```

Under VB.NET one of the two assignments fails. (Which assignment fails depends on locale settings.)

VB Migration Partner's library contains two methods, named **CDate6** and **IsDate6**, that mimics VB6 behavior and ensures that converted applications behave like the original ones.

ChrB

VB.NET doesn't support "byte-oriented" string methods.

VB Migration Partner provides the **ChrB6** replacement method, which approximates the original VB6 method's behavior but isn't guaranteed to work well in all circumstances. This replacement method is marked as obsolete and methods invocations are flagged with a migration warning.

Close#

The Close# keyword maps to the **FileClose** method in Microsoft.VisualBasic.dll. File number can't be preceded by the # symbol.

Const

VB6 supports implicit conversion between string and numeric values, VB.NET doesn't. For example, the following code is legal in VB6 but not in VB.NET:

```
Const ONE As String = "1"
Const TWO As Integer = ONE * 2  ' this is equal to 2 (numeric)
```

Current version of VB Migration Partner doesn't automatically fix this problem.

CreateObject

The CreateObject method defined in Microsoft.VisualBasic.dll fails to recognize classes that have been converted to VB.NET. To understand why this detail can impact on migrated applications, let's suppose that you have an ActiveX EXE project named *TestProject* and that exposes a public class named *Widget*. Being a public COM class, the application can instantiate the class by means of either the New operator or the CreateObject method. (CreateObject is often used when the class name is parameterized or when you want the new object to run in a different thread.) After the migration to VB.NET, the class isn't a COM class any longer and the canonic CreateObject function won't work.

VB Migration Partner fixes this issue by defining a special **CreateObject6** method that can instantiate both classes converted from VB6 and standard COM classes.

CStr, Str

Str and CStr methods support date arguments under VB6, but not under VB.NET.

CVar

VB.NET doesn't support the Variant data type, consequently the CVar function isn't supported and should be rendered as **CObj**.

CVErr

VB.NET doesn't support the CVErr method. The simplest way to transport information about an error is to return an Exception object.

Date, Date$

VB6 overloads the Date keyword, in that it is both the name of the Date type and the name of the function that returns today's date. In VB.NET the value of today's date should be obtained by invoking the **Today** property. Instead, references to the Date$ function should be translated as **DateString**.

Debug.Print

The Debug.Print method must be converted to either **Debug.Write** or **Debug.WriteLine**, depending on whether the original VB6 statement has a trailing semicolon.

Declare

VB.NET fully supports the Declare keyword, except the ability to define "As Any" parameters. When converting a VB6 application you should replace such parameters with a definite data type; if callers pass different data types to the Declare – for example, a Long and a String – you should provide different overloads of the Declare statement, so that no "As Any" parameters are necessary.

Another problem with Declare statements is that VB.NET doesn't support 32-bit integers used as callback addresses, as it happens with a few Windows API methods such as EnumFonts or EnumWindows. In this case you must declare a Delegate class with opportune syntax and change the parameter type so that it uses the delegate.

Finally, a minor problem you might face is that VB6 allows Declare statements containing two or more parameters with same name, but they are illegal under VB.NET.

VB Migration Partner handles these problems automatically: it generates all the necessary overloads for the Declare, defines one delegate class for each callback parameters, and adjusts parameter names if any duplicate exists.

DefBool, DefByte, DefCur, DefDate, DefDbl, DefInt, DefLng, DefObj, DefSng, DefStr, DefVar

These compiler directives aren't supported by VB.NET; VB Migration Partner assigns the correct data type to all variables that aren't declared or don't have an explicit As clause.

Dim

VB6 allows you to define a variable without the As clause, in which case the type of the variable is affected by the Def*Xxx* directive whose range corresponds to the first character of the variable's name. (If no Def*Xxx* directive is present, the Variant type is used by default.) VB.NET allows Dim keywords without the As clause only if Option Strict Off is used; the type of the variable is always Object. A different rule applies if the variable is part of a list, as in this statement:

```
Dim x, y, z As Double
```

In VB6 the *z* variable is of type Double, whereas *x* and *y* are affected by the current Def*Xxx* directive (or are Variants if no directive is found). Conversely, in VB.NET all three variables are of type Double.

Another difference between VB6 and VB.NET is that, if the Dim keyword appears inside an If, For, For Each, Do, While, or Select block, then the latter limits the scope of the variable to the block itself whereas VB6 makes the variable visible to the entire method:

```
Do While x > 0
    ' in VB.NET the k variable can't be accessed outside the Do…Loop block
    Dim k As Integer
```

```
        ...
    Loop
```

Both Upgrade Wizard and VB Migration Partner solve the problem by moving the Dim at the top of the block where it is defined.

Dim (arrays)

VB.NET doesn't support arrays whose lower index is nonzero. Also, VB.NET requires that the rank – that is, the number of dimensions of the array - be specified if the array is declared but not initialized:

```
Dim arr(,) As Integer ' a two-dimensional array of integers
```

Dir

The version of the Dir method found in Microsoft.VisualBasic.dll assembly differs from the VB6 version in an important detail: when the latter enumerates all the directories in a subfolder, it can return two spurious entries: "." (single dot) and ".." (double dot), which represent current directory and parent directory. The VB.NET version doesn't return these items. If the VB6 code being migrated assumes that these elements are always present, the converted VB.NET applications will behave errantly.

VB Migration Partner mimics the VB6 behavior, so that the migrated VB.NET works correctly even if the original VB6 application discards the first two results from Dir without checking whether they contain the "." and ".." entries.

DoEvents

Under VB6 the DoEvents method ensures that all pending Windows messages are processed, then it returns the number of open forms. VB.NET's version of this method is **Application.DoEvents**: it ensures that pending messages are processed but it returns no value to the caller.

VB Migration Partner provides a **DoEvents6** method that behaves like the VB6 counterpart.

End

VB.NET supports the End keyword, but it is recommended that you invoke the **Application.Exit** method instead of (or before) executing the End keyword. This is the VB.NET code that VB Migration Partner produces:

```
Application.Exit(): End
```

EndIf

VB6 supports the obsolete spelling "EndIf" (no embedded space); the VB6 code editor automatically expands this word into "End If", but the obsolete keyword might be encountered in applications whose source code is automatically generated. VB.NET doesn't support the "EndIf" spelling.

EOF#

The EOF# keyword maps to the EOF method in Microsoft.VisualBasic.dll. File number can't be preceded by the # symbol.

Eqv

The Eqv operator isn't supported under VB.NET. This is the equivalent expression to be used instead:

```
result = (CBool(op1) = CBool(op2))
```

Erase

Under VB6 an array can be either static or dynamic: static arrays are declared and created in the Dim statement (e.g. Dim arr(10) As String), whereas dynamic arrays are first declared using a Dim statement (with no indexes) and later instantiated by means of a ReDim statement. The two array types use different memory allocation mechanisms – memory for static arrays is allocated at compile

time, whereas dynamic arrays are always allocated at runtime – but from the developer's perspective the main difference is in the behavior of the Erase keyword.

When the Erase keyword is applied to a static array, the memory allocated to the array is cleared: all the array elements are reset to their default value (zero, null string, or Nothing) but the array bounds aren't modified. Conversely, when a dynamic array is erased, then the memory allocated to the array is released and any attempt to reference any element causes a runtime exception.

Under .NET all arrays are dynamic and the Erase keyword behave as with all VB6 dynamic arrays. This minor differences can cause a problem when migrating a piece of VB6 code that erases a static array and is then followed by a reference to one of its elements:

```
' this code works under VB6 but fails after migrating to VB.NET
Dim arr(10) As Double    ' a static array
' …
Erase arr
arr(0) = 123             ' exception under VB.NET
```

To work around this issue, VB Migration Partner converts the Erase statement into either **Erase6** or the **ClearArray6** method, for dynamic or static arrays respectively.

Error

VB.NET doesn't support the Error statement; it should be replaced by a call to the **Err.Raise** method.

FileAttr

VB6 supports a version of FileAttr with two arguments. If the second argument is 1 or omitted the function returns the mode used to open the file; if the second argument is 2, then the function raises an Error 5 under VB6 (it used to return the operating system file handle under 16-bit version of Visual Basic.) The VB.NET version of this method supports only one argument.

FileDateTime

The VB6 version of the FileDateTime method works with both files and directories, whereas the VB.NET version works only with directories.

VB Migration Partner provides a special **FileDateTime6** method that behaves like the VB6 one.

Format

The VB.NET version of the Format method differs from the VB6 version in many ways. First, it doesn't accept named formats – namely, General Number, Currency, Fixed, Standard, Percent, Yes/No, True/False, On/Off. Second, it doesn't support formatting of string values with the @, &, <, and > placeholders. Third, it doesn't support a few date/time formats. Fourth, it doesn't support the FirstDayOfWeek and FirstWeekOfYear optional arguments. There are other, minor differences too.

VB Migration Partner works around all these differences by exposing the special **Format6** method, which behaves more closely like the VB6 method.

Get#

The Get# keyword maps to the **FileGet** method defined in the Microsoft.VisualBasic.dll assembly. However, the FileGet and FilePut methods don't work in exactly the same manner with nonscalar values, therefore you can't exchange data files between VB6 and VB.NET if the file contains dynamic arrays, variant values, or structures.

VB Migration Partner maps the Get# keyword to the **FileGet6** method, which offers better compatibility with the original VB6 method.

GoSub

VB.NET doesn't support the GoSub keyword. You should move the code in the GoSub block to a distinct method, defining and passing as arguments the local variables that the GoSub block uses.

If you use the **ConvertGosubs** pragma, VB Migration Partner performs this refactoring action for you wherever it is possible to do it while preserving functional equivalence with the original code. When this approach isn't possible, VB Migration Partner manages to convert this keyword nevertheless, but delivers code that can't be easily maintained. We recommend that you apply the pragma and then manually modify the VB6 code in those cases when refactoring isn't possible

ImeStatus

VB.NET doesn't support the ImeStatus method.

VB Migration Partner defines a dummy replacement method named **ImeStatus6** method that always returns zero.

Imp

The Imp operator isn't supported under VB.NET. This is the equivalent expression to be used instead:

```
result = Not op1 Or op2
```

Implements

VB.NET supports the Implements keyword, but its argument must be and Interface type, not a class as in VB6. Also, the Implements keyword is also used to qualify methods and properties that implement a member of the interface:

```
Private Sub IAddin_Connect(ByVal root As Object) Implements IAddin.Connect
    ' ...
End Sub
```

Input#

The Input# keyword maps to the Input method in Microsoft.VisualBasic.dll. File number can't be preceded by the # symbol.

InputB

VB.NET doesn't support "byte-oriented" string methods.

VB Migration Partner provides the **InputB6** replacement method, which approximates the original VB6 method's behavior but isn't guaranteed to work well in all circumstances. This replacement method is marked as obsolete and methods invocations are flagged with a migration warning.

InputBox

The VB.NET version of the InputBox method works exactly like the VB6 version, except for two details. First, the VB6 version accepts individual CR characters (ASCII 13) as line separators in the message text, whereas the VB.NET version requires a CR-LF pair (ASCII 13 + ASCII 10). Second, the VB6 version takes an optional pair of coordinates and interprets them as twips, whereas the VB.NET version interprets them as pixels. Third, the VB.NET version causes a Deactivated event in the current form, and then an Activated event when the input box is closed and the focus regains the input focus.

To avoid these minor problems, VB Migration Partner defines the **InputBox6** method that works exactly like the VB6 method.

InstrB

VB.NET doesn't support "byte-oriented" string methods.

VB Migration Partner provides the **InstrB6** replacement method, which approximates the original VB6 method's behavior but isn't guaranteed to work well in all circumstances. This replacement method is marked as obsolete and methods invocations are flagged with a migration warning.

IsEmpty

VB.NET doesn't support the Empty value, therefore the IsEmpty function isn't supported. In most cases, the Empty value is converted to Nothing, therefore the IsEmpty function can map to the IsNothing method.

IsMissing

VB.NET doesn't support missing optional parameters, therefore the IsMissing function isn't supported. For example, the following VB6 code:

```
Sub Execute(Optional ByVal shipDate As Variant)
    If IsMissing(shipDate) Then shipDate = Now
        …
End Sub
```

should be modified – before or after the migration – so that the *shipDate* parameter has a well defined value if omitted. For example, you can use a value that is surely invalid for the application as the "missing" value:

```
Sub Execute(Optional ByVal shipDate As Date = #1/1/1900#)
    If shipDate = #1/1/1900# Then shipDate = Now
        …
End Sub
```

VB Migration Partner uses the special **IsMissing6** method to handle missing optional arguments.

IsNull

VB.NET doesn't support the Null value, therefore the IsNull function isn't supported. In most cases, the Null value is converted to either Nothing or DBNull.Value, therefore the IsNull function can map to the following test:

```
If value Is Nothing OrElse TypeOf value is DBNull Then …
```

Left

VB.NET supports Left and Right string functions. However, if the code runs inside a form or a user control, VB.NET interprets these names as references to the Left and Right properties of the Form and UserControl object itself, which causes a compilation error. You can avoid this error in two ways. First, you can explicitly reference the Microsoft.VisualBasic namespace, possibly with an Imports alias at the top of the file (this is the approach that VB Migration Partner uses):

```
Imports VB = Microsoft.VisualBasic
…
Function GetFirstLastChar(ByVal arg As String) As String
    Return VB.Left(arg, 1) & VB.Right(arg, 1)
End Function
```

Alternatively, you can use methods exposes by the System.String class, for example:

```
Function GetFirstLastChar(ByVal arg As String) As String
    Return arg.SubString(0, 1) & arg.SubString(arg.Length – 1)
End Function
```

LeftB

VB.NET doesn't support "byte-oriented" string methods.

VB Migration Partner provides the **LeftB6** replacement method, which approximates the original VB6 method's behavior but isn't guaranteed to work well in all circumstances. This replacement method is marked as obsolete and methods invocations are flagged with a migration warning.

Len

VB6's Len method works with strings and Type…End Type blocks; in the latter case it returns the number of bytes taken when the block is written to disk or passed to a Windows API method. The

VB.NET's Len method only works with strings; you should use the **Marshal.SizeOf** method when working with structures, even though you aren't guaranteed that you get the same value you'd receive in VB6.

VB Migration Partner's library defines a **Len6** method that behaves like the VB6 function.

LenB

VB.NET doesn't support "byte-oriented" string methods.

VB Migration Partner provides the **LenB6** replacement method, which approximates the original VB6 method's behavior but isn't guaranteed to work well in all circumstances. This replacement method is marked as obsolete and methods invocations are flagged with a migration warning.

Let, Set

VB.NET doesn't support neither the Let nor the Set keyword. Object assignments don't require the Set keyword because the lack of support for parameterless default members ensures that no ambiguity exists for the following statement:

```
Dim tb As TextBox, txt As String
tb = TextBox1           ' assign an object reference
txt = TextBox1.Text     ' (explicitly) assign the default member
```

Line Input#

The Line Input# keyword maps to the LineInput function in Microsoft.VisualBasic.dll. File number can't be preceded by the # symbol. Notice that, instead of taking the variable as an argument, the LineInput method returns the value read from file.

```
result = LineInput(1)
```

Load

You can't load a form in VB.NET. Creating a form instance has more or less the same effect.

LoadPicture

VB.NET doesn't support the LoadPicture method; it can be rendered by means of the **Image.FromFile** method. There are a few differences, though, because the Image.FromFile method doesn't support the size, colorDepth, x, and y (optional) arguments. Also, when an empty string is passed in the first argument, the LoadPicture VB6 method returns a null image.

To account for all these differences, VB Migration Partner defines a special method named **LoadPicture6** that behaves like the VB6 method.

LoadResBitmap, LoadResData, LoadResString

VB.NET doesn't support LoadResString, LoadResBitmap, and LoadResData methods.

VB Migration Partner converts VB6 resource files to.NET files and attempts to convert these methods into references to **My.Resources.Xxxx** items. (This conversion is possible only if the resource ID is a literal constant value.) In addition, automatic conversion of LoadResBitmap and LoadResData methods is tricky, because these methods take a second argument that defines the kind of resource (icon, bitmap, cursor); if this argument isn't a literal or an enumerated constant, VB Migration Partner falls back to **LoadResBitmap6** or **LoadResData6** methods in the support library.

LOC#

The LOC# keyword maps to the Loc method in Microsoft.VisualBasic.dll. File number can't be preceded by the # symbol.

Lock#

The Lock# keyword maps to the Lock method in Microsoft.VisualBasic.dll. File number can't be preceded by the # symbol and the To keyword isn't allowed.

LOF#

The LOF# keyword maps to the LOF method in Microsoft.VisualBasic.dll. File number can't be preceded by the # symbol.

LSet

VB.NET doesn't support the LSet keyword. You can use the **String.PadRight** method as a replacement in string assignments:

```
s1 = s2.PadRight(s1.Length, " "c)  ' this replaces LSet s1 = s2
```

VB.NET has no equivalent for the LSet keyword used to copy elements of different Type…End Type blocks (Structure blocks in VB.NET).

MidB

VB.NET doesn't support "byte-oriented" string methods.

VB Migration Partner provides the **MidB6** replacement method, which approximates the original VB6 method's behavior but isn't guaranteed to work well in all circumstances. This replacement method is marked as obsolete and methods invocations are flagged with a migration warning.

Mod

VB6's Mod operator converts its operands to integers and then returns the (integer) remainder of the division. By contrast, VB.NET's Mod operator doesn't perform any conversion: if the operands are floating-point numbers, the result is the remainder of the floating-point division. If the operands of the Mod operator are Single, Double, or Currency values, you should explicitly convert them to 32-bit integers before using the Mod operator under VB.NET:

```
result = CInt(op1) Mod CInt(op2)
```

VB Migration Partner performs this fix automatically, if necessary.

MsgBox

The VB.NET version of the MsgBox method works exactly like the VB6 version, except that the VB6 version accepts individual CR characters (ASCII 13) as line separators in the message text, whereas the VB.NET version requires a CR-LF pair (ASCII 13 + ASCII 10). Also, the VB.NET MsgBox method causes a Deactivated event to be fired in the form that loses the input focus, and an Activated event when the message box is closed and the form regains the input focus.

To avoid these minor problems, VB Migration Partner defines a **MsgBox6** method that works exactly like the VB6 method.

Name

VB.NET doesn't support the Name method; you should use the **Rename** method, defined in the Microsoft.VisualBasic.dll assembly.

Next

Under VB6, a single Next keyword can terminate two or more For loops, as in this example:

```
For i = 0 To 10
    For j = 0 To 20
```

```
        …
    Next j, i
```

VB.NET doesn't support this syntax and requires that each For loop be terminated by a distinct Next keyword.

ObjPtr

VB6 language includes three undocumented functions: VarPtr, StrPtr, and ObjPtr. These methods have no equivalent under VB.NET and can't be translated.

VB Migration Partner issues a warning when one of these methods is encountered.

On … GoSub

VB.NET doesn't support calculated On…GoSub statements. You should move the code in GoSubs block to distinct methods, defining and passing as arguments the local variables that each GoSub block uses.

VB Migration Partner converts this keyword but delivers code that can't be easily maintained, therefore it's recommended that you get rid of On…GoSub statements before migrating the project.

On … Goto

VB.NET doesn't support calculated On…GoTo statements. You can replace it with a Select Case whose Case blocks contain a GoTo statement, which increases the number of GoTos in the application and makes control flow hard to follow. It is recommended that original VB6 code be revised to get rid of On…GoTo statements before migrating the project.

Open#

The Open# keyword maps to the **FileOpen** method defined in the Microsoft.VisualBasic.dll assembly, which has a standard object-oriented syntax; keywords normally used inside an Open# statement – such as Input, Output, Random, Binary, Access, Shared, Read, Write, Lock – aren't supported.

Option Base

VB.NET doesn't support Option Base, because all arrays must have a zero lower index.

VB Migration Partner accounts for this directive and allows you to control the actual lower index by means of the **ArrayBounds** pragma.

Option Explicit

VB.NET supports this directive, but it requires an explicit On (or Off) argument:

```
Option Explicit On
```

Option Private

This directive is used only in Access VBA and has no effect in VB6. VB.NET doesn't support this directive and VB Migration Partner can safely ignore it.

Print, Print#

VB.NET doesn't support the Print method – that outputs to a form, a user control, or a PictureBox's surface – and partially supports the Print# method that outputs to file. The latter method maps to the **Write** or **WriteLine** method defined in Microsoft.VisualBasic.dll assembly, but the bytes actually emitted aren't necessarily the same as in the original VB6 code, therefore the converted VB.NET application might not be able to exchange data with existing VB6 applications.

VB Migration Partner works around this issue by defining the **FilePrint6** and **FilePrintLint6** methods, which behave like the original VB6 methods.

Property Get, Property Let, Property Set

VB.NET uses a different syntax for properties. The Property Get method maps to the Get block in VB.NET; the Property Let or Property Set methods map to the Set block:

```
Public Property Name() As String
    Get
        Return m_Name
    End Get
    Set(ByVal value As String)
        m_Name = value
    End Set
End Property
```

Also, VB6 supports ByRef parameters for the Property Let and Property Set blocks, but VB.NET doesn't.

Put#

The Put# keyword maps to the **FilePut** method defined in the Microsoft.VisualBasic.dll assembly. However, the FileGet and FilePut methods don't work in exactly the same manner with nonscalar values, therefore you can't exchange data files between VB6 and VB.NET if the file contains dynamic arrays, variant values, or structures.

VB Migration Partner maps the Put# keyword to the **FilePut6** method, which offers better compatibility with the VB6 method.

ReDim

In VB6 the ReDim keyword can both declare an array and create its elements; in VB.NET you need two separate statements: the Dim keyword declares the array (and optionally creates its elements), but the ReDim keyword can only create (or recreate) array elements. In other words, the following VB6 statement:

```
ReDim arr(10) As Integer  ' defines and creates an array
```

must be converted to the following VB.NET sequence:

```
Dim arr() As Integer        ' defines an array
…
ReDim arr(10)               ' creates the array and its elements
```

VB.NET doesn't support the "As" clause in ReDim keywords: the type of array elements is defined in the Dim statement. Like the Dim keyword, the ReDim keyword doesn't support lower indices other than zero.

Rem

VB6 supports multiline remarks, as in this example:

```
' first line _
second line _
third line
```

VB.NET doesn't support this syntax, therefore you must add an apostrophe at the beginning of the line:

```
' first line
' second line
' third line
```

Return

VB.NET doesn't support the GoSub keyword, therefore it doesn't need to support the Return keyword. However, VB.NET uses the Return keyword to return a value from a Function or from the Get block of a Property block.

VB Migration Partner correctly translates both the Gosub and Return keywords, but delivers code that can't be easily maintained, therefore it's recommended that you get rid of GoSub statements before migrating the project.

Right

VB.NET supports Left and Right string functions. However, if the code runs inside a form or a user control, VB.NET interprets these names as references to the Left and Right properties of the Form and UserControl object itself, which causes a compilation error. You can avoid this error in two ways. First, you can explicitly reference the Microsoft.VisualBasic namespace, possibly with an Imports alias at the top of the file (this is the approach that VB Migration Partner uses):

```
Imports VB = Microsoft.VisualBasic
…
Function GetFirstLastChar(ByVal arg As String) As String
    Return VB.Left(arg, 1) & VB.Right(arg, 1)
End Function
```

Alternatively, you can use methods exposes by the System.String class, for example:

```
Function GetFirstLastChar(ByVal arg As String) As String
    Return arg.SubString(0, 1) & arg.SubString(arg.Length – 1)
End Function
```

RightB

VB.NET doesn't support "byte-oriented" string methods.

VB Migration Partner provides the **RightB6** replacement method, which approximates the original VB6 method's behavior but isn't guaranteed to work well in all circumstances. This replacement method is marked as obsolete and methods invocations are flagged with a migration warning.

RSet

VB.NET doesn't support the RSet keyword. You can use the **String.PadLeft** method to replace the RSet in string assignments:

```
s1 = s2.PadLeft(s1.Length, " "c)  ' this replaces RSet s1 = s2
```

SavePicture

VB.NET doesn't support the SavePicture method, which can be rendered by means of the **Image.Save** method.

Seek#

Both the Seek# command and the Seek function map to the Seek method in Microsoft.VisualBasic.dll. (The version with one argument is the function, the version with two argument is the command.) File number can't be preceded by the # symbol.

Spc

The SPC keyword, used to insert spaces in a Print, Print#, and Debug.Print method, is supported by VB.NET only inside converted Print# statements (which map to Write and WriteLine statements).

Split

When its first argument is an empty string, the VB6 version of Split returns an "empty" string array, that is an array that has no items. (Such an array has LBound=0 and UBound=-1.) Conversely, the

VB.NET version of the Split method returns an array with one element (with zero index) set equal to the null string.

VB Migration Partner accounts for this difference and translates Split into the Split6 helper method, which behaves exactly like the VB6 method.

Sqr

The Sqr keyword isn't implemented in Microsoft.VisualBasic.dll assembly. You must replace it with a reference to the **Math.Sqrt** method (in System namespace)

```
result = Math.Sqrt(value)
```

Static

VB6 supports the Static keyword both at the variable-declaration level (in which case that variable is declared as static and preserves its value between calls to the method) and at the method-declaration level, in which case all the variables inside the method are treated as static variables. VB.NET supports the Static keyword only at the variable-declaration level. For example, the following VB6 code:

```
Static Sub Test()
    Dim x As Integer, y As Long
    …
End Sub
```

must be converted to VB.NET as follows:

```
Sub Test()
    Static x As Short
    Static y As Integer
    …
```

```
End Sub
```

Stop

VB.NET supports the Stop keyword, but it is good programming habit to replace it with a call to the **Debugger.Break** method. The reason: if a Stop keyword remains in product code, it crashes the application. Instead, the Debugger.Break method is ignored if the project is compiled in Release mode.

StrConv

The VB6 version of the StrConv method takes both strings and Byte array in its first argument, and can convert from ASCII to Unicode and back. The StrConv method defines in Microsoft.VisualBasic.dll assembly works only with strings and can't convert to/from Unicode.

VB Migration Partner provides the **StrConv6** method, which matches the VB6 behavior perfectly.

String, String$

VB6 overloads the String keyword, in that it is both the name of the String type and the name of a library method. In VB.NET the String and String$ methods should be rendered as the **StrDup** function.

StrPtr

VB6 language includes three undocumented functions: VarPtr, StrPtr, and ObjPtr. These methods have no equivalent under VB.NET and can't be translated.

VB Migration Partner issues a warning when one of these methods is encountered.

TAB

The TAB keyword, used to insert spaces in a Print, Print#, and Debug.Print method, is supported by VB.NET only inside converted Print# statements (which map to Write and WriteLine statements).

Time, Time$

The Time function should be translated as a reference to the **TimeOfDay** property. Instead, references to the Time$ function should be translated as **TimeString**.

Timer

VB6's version of the Timer functions returns a Single value; the Timer function defined in the Microsoft.VisualBasic.dll assembly returns a Double value.

To

The To keyword is supported inside Dim and ReDim statements; however, the lower indices of the array can only be zero under VB.NET, therefore in practice this keyword can be always removed. The Upgrade Wizard leaves the Dim or ReDim statement unchanged, therefore any nonzero lower index causes a compilation error.

VB Migration Partner is able to solve this problem if an opportune **ArrayBounds** pragma is used.

Type … End Type

Type…End Type blocks must be converted to VB.NET Structure blocks. However, if the Type contains initialized arrays, fixed-length strings, or auto-instancing (As New) object variables, the Structure requires to be initialized:

```
Structure
    Public ID As Integer
    Public Name As String       ' this was String * 30
```

```
    Public Notes() As String    ' this was Notes(10)
    Public Address As Location  ' this was Address As New Location

    Public Sub Initialize()
        Name = Space(30)
        ReDim Notes(10)
        Address = New Location()
    End Sub
End Structure
```

In addition to converting the Type block into a Structure, VB Migration Partner automatically initializes it. VB Migration Partner generates also the correct **System.Runtime.InteropServices.MarshalAs** attributes to ensure that string and array elements are marshaled correctly when the structure is passed as an argument to a Declare method.

TypeName

The VB.NET Typename function works like the original VB6 method, however you must pay attention to an important detail. The following VB6 code tests whether a value is a 16-bit integer

```
If TypeName(value) = "Integer" Then
```

The problem in migrating this code to VB6 is that *value* is now a Short variable, therefore the code should be migrated as:

```
If TypeName(value) = "Short" Then
```

A similar problem occurs with other data types that have been renamed in VB.NET, such as Long and Currency.

To avoid this problem, VB Migration Partner defines a special **TypeName6** method that returns the same string that would return under VB6.

TypeOf

VB6 TypeOf keyword doesn't perfectly corresponds to VB.NET keyword in many cases. For example, the following test always succeeds in VB.NET, because all data types inherit from System.Object:

```
If TypeOf value Is Object Then …
```

Instead, if the test for Object is meant to check that a value isn't scalar you must use this code:

```
If Not TypeOf value Is String AndAlso Not value.GetType().IsValueType Then …
```

Under VB.NET you can't use TypeOf with value types, therefore you need a different approach when testing the type of a Structure (e.g. a converted Type…End Type block):

```
If value.GetType() Is GetType(myudt) Then …
```

Unload

You can't unload a form in VB.NET. Invoking the form's Close method has more or less the same effect.

Unlock#

The Unlock# keyword maps to the Unlock method in Microsoft.VisualBasic.dll. File number can't be preceded by the # symbol and the To keyword isn't allowed.

UserControl

Under VB6 you can use the UserControl keyword inside a user control class to reference the current user control, as in:

```
UserControl.BackColor = vbRed
```

VB.NET doesn't recognize this keyword, thus you must replace it with a reference to the "Me" object:

```
Me.BackColor = Color.Red
```

VarPtr

VB6 language includes three undocumented functions: VarPtr, StrPtr, and ObjPtr. These methods have no equivalent under VB.NET and can't be translated.

VB Migration Partner issues a warning when one of these methods is encountered.

Wend

VB6 supports While…Wend loops, whereas VB.NET supports While…End While loops; therefore the Wend keyword must be translated as **End While**.

Width#

The Width# keyword maps to the **FileWidth** method defined in Microsoft.VisualBasic.dll assembly.

Write#

VB.NET partially supports the Write# method that outputs to file, which maps to the Write or WriteLine method defined in Microsoft.VisualBasic.dll assembly. However, the bytes actually emitted aren't necessarily the same as in the original VB6 code, therefore the converted VB.NET application might not be able to exchange data with existing VB6 applications.

VB Migration Partner translates this method to FileWrite6 and FileWriteLine6 methods, which behave like the original VB6 method.

Classes and ActiveX Components

Property procedures

A VB6 property is defined by means of its Property Get, Property Let, and Property Set procedures. These procedures are converted into a single **Property…End Property** VB.NET block, which must be marked with the **ReadOnly** or **WriteOnly** keywords if one of the blocks is omitted. During the

conversion, it is also necessary to account for different scopes of the Property Get block and the Property Let (or Set) block. For example, consider the following VB6 code:

```
Public Property Get ID() As Integer
    ID = m_ID
End Property

Public Property Get Name() As String
    Name = m_Name
End Property

Friend Property Let Name(ByVal newValue As String)
    m_Name = newValue
End Property
```

This is how the property must be translated to VB.NET:

```
Public ReadOnly Property ID() As Short
    Get
        Return m_ID
    End Get
End Property

Public Property Name() As String
    Get
        Return m_Name
    End Get
```

```
    Friend Set(ByVal newValue As String)
        m_Name = newValue
    End Set
End Property
```

Properties with both Let and Set procedures

A VB6 property of Variant type can appear in both a Property Let and a Property Set procedure. VB.NET's **Property...End Property** block supports only one "setter" block, which must merge code from both original blocks. In most cases, you can (and should) simplify the code that is generated by converting and merging the VB6 code verbatim. For example, given the following VB6 code:

```
Property Get Owner() As Variant
    If IsObject(m_Owner) Then
        Set Owner = m_Owner
    Else
        Owner = m_Owner
    End If
End Property

Property Let Owner(ByVal newValue As Variant)
    m_Owner = newValue
End Property

Property Set Owner(ByVal newValue As Variant)
    Set m_Owner = newValue
End Property
```

VB Migration Partner converts this code to VB.NET as follows:

```
    Public Property Owner() As Object
        Get
            Return m_Owner
        End Get
        Friend Set(ByVal newValue As Object)
            m_Owner = newValue
        End Set
    End Property
```

Also, notice that the original Property Let and Property Set procedures might have different visibility – Friend and Private, for example – therefore you have to choose the "broader" visibility (Friend, in this case) when you merge them into a single "setter" block.

Optional parameters in Property procedures

In VB6 it is legal to have a Property Get and a Property Let (or Set) block whose parameters differ for the Optional keyword, as in the following example:

```
Dim m_Value(10) As String

Property Get Value(ByVal index As Long) As String
    Value = m_Value(index)
End Property

Property Let Value(ByVal Optional index As Long, ByVal newValue As String)
    m_Value(index) = newValue
End Property
```

(Notice that this is the only case in which a non-Optional argument can follow an Optional parameter.) In VB.NET the "getter" and "setter" blocks of a Property share the same parameters, therefore they can't differ for the Optional keyword. In this case, VB Migration Partner uses the Optional keyword for the parameter:

```
Dim m_Value(10) As String

Property Value(ByVal Optional index As Integer = 0) As String
    Get
        Return m_Value(index)
    End Get
    Set (ByVal newValue As String)
        m_Value(index) = newValue
    End Set
End Property
```

An even more intricate case occurs when the Property Get and Property Set block differ for the default value of an optional parameter, as in:

```
Dim m_Value(10) As String

Property Get Value(ByVal Optional index As Long = 0) As String
    Value = m_Value(index)
End Property

Property Let Value(ByVal Optional index As Long = -1, ByVal newValue As String)
    m_Value(index) = newValue
```

```
End Property
```

While this syntax admittedly makes little sense, it is legal in VB6. However, there is no way to convert this syntax correctly to VB.NET, thus VB Migration Partner flags it with a migration warning.

Initialize event

VB.NET doesn't support the Initialize event in classes, forms, and user controls. Any action that needs to be performed when an instance of the class is created should be moved to the class's constructor.

Terminate event

VB.NET doesn't support the Terminate event in classes, forms, and user controls. The VB.NET element that is closest to the Terminate event is the **Finalize** method, but the two aren't equivalent. The problem is that VB.NET (and all .NET Framework languages, for that matter) doesn't support the so-called *deterministic finalization*, which means that .NET objects aren't destroyed when the last reference to them is set to Nothing. This difference causes unpredictable runtime errors after the migration, unless the developer is very careful in how objects are destroyed; in general, the amount of code that must be written to work around the problem isn't negligible.

VB Migration Partner can generate such code if the **AutoDispose** pragma is used.

Default properties (definitions)

In VB6 you can define a field, a property, or a method as the default member of a class. The most common cases of default members are properties exposed by controls, such as the Text property of the TextBox control or the Caption property of the Label control. VB.NET supports neither default fields nor default methods; only default properties are supported and, more important, only properties that have one or more arguments (e.g. the Item property of a Collection). Here's how you can define a default property in VB.NET:

```
Default Property Item(ByVal index As Integer) As String
    Get
        Return m_Items(index)
    End Get
    Set(ByVal value As String)
        m_Items(index) = value
    End Set
End Property
```

Default properties (references)

VB Migration Partner correctly resolves reference to default properties if the variable is strongly-typed. For example, consider the following VB6 method:

```
Sub UppercaseText(ByVal tb As TextBox)
    tb = UCase(tb)
End Sub
```

VB.NET doesn't support default parameterless properties, therefore you must explicitly reference the default property:

```
Sub UppercaseText(ByVal tb As TextBox)
    tb.Text = UCase(tb.Text)
End Sub
```

The actual problem with default parameterless properties becomes apparent when the variable is late-bound, as in this case:

```
Sub UppercaseText(ByVal ctrl As Object)
    ctrl = UCase(ctrl)
End Sub
```

In this case, VB Migration Partner can correctly resolve the default property at runtime if you enable the corresponding feature with the **DefaultMemberSupport** pragma.

Default functions

In VB6 you can define a method as the default member of a class, whereas VB.NET supports only default properties and only if the property takes one or more arguments. For this reason, you should turn the Function into a **Readonly Property** block and mark it with the **Default** keyword.

VB Migration Partner automatically does this replacement.

Default members and COM clients

If a VB6 class contains a default member and the class is exposed to COM clients, when you translate the class to VB.NET you should mark the default member – be it a field, a property, or a method – with a **System.Runtime.InteropServices.DispID** attribute, as in this example:

```
<System.Runtime.InteropServices.DispID(0)> _
Public Name As String
```

Member description

You can decorate a VB6 class or class member with a description; such a description appears when the class is explored by means of the VB6 Object Browser. If the class is a user control and the member is a property, the description appears also in the property grid at design time. To implement the same support in a VB.NET class you must convert VB6's Description attribute to the equivalent XML

comment (to display the description in the object browser) and to a **System.ComponentModel.Description** attribute:

```
''' <summary>
''' Name of the widget
''' </summary>
<System.ComponentModel.Description("Name of the widget")> _
Public Property Name() As String
    ' …
End Property
```

Classes and interfaces

VB6 has no notion of interfaces: you define an interface by authoring a VB6 class with one or more empty methods or properties, then use the class's name in an Implements clause at the top of another class elsewhere in the same project. (If the class that defines the interface is public then the class can implement the interface can reside in a different project.) In VB.NET you have to render interfaces with an explicit **Interface…End Interface** block.

In some rare cases, however, a VB6 class is used to define an interface and is also instantiated: in the converted VB.NET program such a class must be rendered as two distinct types: an interface and a concrete class. Consider the following VB6 class named IAddin:

```
Public Property Get Name() As String
    ' no code here
End Property

Public Sub Connect(ByVal app As Object)
    ' no code here
```

```
End Sub
```

By default, VB Migration Partner converts this code into an Interface block plus a Class block:

```
Interface IAddin
    ReadOnly Property Name()  As String
    Sub Connect(ByVal app As Object)
End Interface

Class IAddinClass
    Implements IAddin

    Public ReadOnly Property Name() As String Implements IAddin.Name
        Get
            ' no code here
        End Get
    End Property

    Public Sub Connect(ByVal app As Object) Implements IAddin.Connect
        ' no code here
    End Sub
End Class
```

You can use the **ClassRenderMode** pragma to tell VB Migration Partner that only the Interface block should be generated.

Fields inside interfaces

A VB6 interface – more precisely, a VB6 class that is used to define an interface – can include one or more public class-level fields. Such fields become part of the interface and must be accounted for by classes that implement the interface, typically by including a Property Get and Property Let (or Set) pair of procedures. VB.NET interfaces can't include fields, therefore the VB6 must be transformed into a property when the class is converted into an Interface…End Interface block. For example, consider the following VB6 class named IAddin:

```
Public Name As String


Public Sub Connect(ByVal app As Object)
    ' no code here
End Sub
```

VB Migration Partner converts it to VB.NET as follows:

```
Interface IAddin
    Property Name()  As String
    Sub Connect(ByVal app As Object)
End Interface

Class IAddinClass
    Implements IAddin

    Private Name_InnerField As String

    Public Property Name() As String Implements IAddin.Name
        Get
            Return Name_InnerField
```

```
        End Get
        Set(ByVal value As String)
            Name_InnerField = value
        End Set
    End Property

    Public Sub Connect(ByVal app As Object) Implements IAddin.Connect
        ' no code here
    End Sub
End Class
```

Collection classes

VB6 collection classes require that a property or method returns the class's enumerator object, which the client application can use to iterate over all the elements of the collection. (This object is implicitly requested and used when a For Each loop is encountered.) This method – which is usually named **NewEnum** and is usually hidden - must be marked with DispID attribute equal to -4. The enumerator object returned by the NewEnum method must implement the **IEnumVariant** interface. However, you can't implement such an interface with VB6, therefore VB6 collection classes typically return the enumerator object of an inner collection. The following code represents the minimal implementation of a VB6 collection class named Widgets:

```
' The private collection used to hold the real data
Private m_Widgets As New Collection

' Return the number of items the collection
Public Function Count() As Long
    Count = m_Widgets.Count
End Function
```

```
' Return a Widget item from the collection
' This item is marked with DispID=0 to make it the default member of the class
Public Function Item(index As Variant) As Widget
    Set Item = m_Widgets.Item(index)
End Function

' Implement support for enumeration (For Each)
' this member is marked with DispID=-4 and is usually hidden
Function NewEnum() As IUnknown
    ' delegate to the private collection
    Set NewEnum = m_Widgets.[_NewEnum]
End Function
```

VB.NET collection classes must implement the **IEnumerable** interface and are expected to return an enumerator object through the only method of this interface, **GetEnumerator**. In turn, VB.NET Enumerator objects must implement the **IEnumerator** interface and its MoveNext, Reset, and Current members.

```
Class Widgets

    ' The private collection used to hold the real data
    Private m_Widgets As New Collection

    ' Return the number of items the collection
    Public Function Count() As Integer
        Return m_Widgets.Count()
```

```
        End Function

        ' Return a Widget item from the collection
        Public Function Item(ByRef index As Object) As Widget
            Return m_Widgets.Item(index)
        End Function

        ' Implement support for enumeration (For Each)
        Public Function NewEnum() As Object
            Return m_Widgets.GetEnumerator()
        End Function

    End Class
```

VB Migration Partner correctly converts the NewEnum member into the **IEnumerable.GetEnumerator** method, even if NewEnum was originally defined as a property. As explained above, the NewEnum member returns the inner collection's enumerator, therefore the resulting VB.NET collection class never needs to implement the IEnumerator interface. Therefore, VB.NET collection classes converted from VB6 work exactly as expected.

Public COM classes

A public VB6 class defined in an ActiveX EXE or DLL project is visible to COM clients, which can instantiate the class by either a New keyword or the CreateObject method. After the conversion to VB.NET the class must be marked with a **ComVisible** attribute to make explicitly visible to existing COM clients, plus a **ProgID** attribute that contains the original name of the class:

```
<System.Runtime.InteropServices.ComVisible(True)> _
<System.Runtime.InteropServices.ProgID("SampleProject.Widget")> _
```

```
Public Class Widget
    ' ...
End Class
```

PublicNotCreatable classes

Public VB6 classes whose Instancing attribute is set to 1-PublicNotCreatable must be converted to public VB.NET classes whose constructor has Friend scope, so that the class can't be instantiated from outside the project where the class is defined.

```
Public Class Widgets
    Friend Sub New()
    End Sub

    …
End Class
```

SingleUse classes

An ActiveX EXE project can define one or more SingleUse and Global SingleUse public classes. SingleUse classes differ from the more common MultiUse classes in that a new instance of the ActiveX process is created any time a client requests an instance of the class. The .NET Framework doesn't support anything similar to SingleUse classes and it isn't easy to simulate this feature under VB.NET; moreover, having a distinct process for each instance of a class impedes scalability, therefore it is recommended that you revise the overall architecture so that the application doesn't depend on SingleUse behavior.

VB Migration Partner ignores the SingleUse attribute and converts SingleUse classes to regular COM-visible VB.NET classes.

Global classes

VB6 supports Global SingleUse and Global MultiUse classes inside ActiveX EXE and DLL projects. (ActiveX DLL projects can't contain SingleUse classes, though.) There is nothing like global classes in VB.NET, therefore all such classes are handles as regular classes, but the client application instantiates and uses a default instance for each global class, and use it to invoke methods and properties.

VB Migration Partner can convert the global class to a class that contains only Shared members; VB Migration Partner can also convert the global class to a Visual Basic module, if an opportune **ClassRenderMode** pragma is used.

DataEnvironment classes

VB.NET doesn't support DataEnvironment classes. VB Migration Partner converts DataEnvironment classes to special VB.NET classes that inherit from the VB6DataEnvironment base class, and correctly handles default instances; however, it doesn't converts advanced features such as grouping, relations, and hierarchical DataEnvironment classes.

PropertyPages

The .NET Framework and VB.NET don't support property pages.

VB Migration Partner converts VB6 property pages to .NET user controls; developers should then write the plumbing code to use and display the user control as appropriate.

UserDocuments

The .NET Framework and VB.NET don't support user documents.

VB Migration Partner converts VB6 user documents to .NET user controls; developers should then write the plumbing code to use and display the user control as appropriate.

Sub Main in ActiveX DLL projects

If an ActiveX DLL project contains a Sub Main method, the Main method is guaranteed to be executed before any class in the DLL is instantiated. VB6 developers can use this feature to read configuration files, open database connections, and so forth. Conversely, the Sub Main method is ignored inside a DLL authored in VB.NET, therefore code must be written to ensure that initialization chores be performed before any .NET object is created.

VB Migration Partner ensures that the Sub Main is executed before any class in the DLL is instantiated. This is achieved by adding a static constructor to all public classes in the DLL, as in this code:

```
Public Class Widget
    Shared Sub New()
        EnsureVB6ComponentInitialization()
    End Sub
    …
End Class
```

where the *EnsureVB6ComponentInitialization* method is a method that invokes the Sub Main method if *Widget* is the first class being instantiated.

MTS components

A public VB6 class defined in an ActiveX DLL project can be made a transactional MTS/COM+ component by setting its MTSTransactionMode attribute to a value other than 0-NotAnMTSObject. VB.NET classes don't support this attribute: instead, the VB.NET class must inherit from the **ServicedComponent** base class and be tagged with the **Transaction** attribute whose argument specifies the required transaction level:

```
Imports System.EnterpriceServices


<Transaction(TransactionIsolationLevel.Required)> _
Public Class MoneyTransfer
    Inherits ServicedComponent
    ' ...
End Class
```

ObjectControl interface

MTS/COM+ components authored in VB6 can implement the ObjectControl interface, which consists of the following three methods: Activate, Deactivate, CanBePooled. VB.NET components that run under COM+ must not implement the ObjectControl interface; instead, they must override the **Activate**, **Deactivate**, and **CanBePooled** methods that they inherit from the **System.EnterpriseServices.ServicedComponent** base class.

VB Migration Partner automatically converts ObjectControl methods into the corresponding VB.NET overrides.

IObjectConstruct interface

MTS/COM+ components authored in VB6 can grab the construction string defined in Component Services applet by implementing the IObjectConstruct interface, which consists of just one method, Construct. This method receives an object argument, whose ConstructString property returns the construction string:

```
Private Sub IObjectConstruct_Construct(Byval pCtorObj As Object)
    Dim connStr As String
    connStr = pCtorObj.ConstructString
    ' ...
```

```
End Sub
```

VB.NET components that run under COM+ must not implement the IObjectConstruct interface; instead, they must override the **Construct** method that they inherit from the **System.EnterpriseServices.ServicedComponent** base class; the only argument that this method receives is the construction string:

```
Protected Overrides Sub Construct(Byval connStr As String)
    ' ...
End Sub
```

Persistable classes

Public VB6 classes in ActiveX EXE and DLL projects can be made persistable, by setting their Persistable attribute to 1-Persistable. VB.NET doesn't support the Persistable attribute: a VB.NET class can be persisted to file - or passed by value to an assembly living in a different AppDomain – if the class is marked with the **<Serializable>** attribute.

VB Migration Partner converts persistable VB6 classes into VB.NET classes that are marked with the <Serializable> attribute and that implement the **ISerializable** interface.

InitProperties, WriteProperties, and ReadProperties events

Persistable VB6 classes can handle the InitProperties, WriteProperties, and ReadProperties events, which fire – respectively – when the class is instantiated, when the COM infrastructure needs to store the object's state somewhere, and when the object is asked to restore a previous state. These events aren't supported by the .NET Framework: a VB.NET classes requiring custom serialization must implement the **ISerializable** interface and therefore implement the **GetObjectData** method and the special constructor that this interface implies.

VB Migration Partner extracts the code from the InitProperties, WriteProperties, and ReadProperties events and uses it inside GetObjectData method and the special constructor implied by the ISerializable interface.

ADO data source and data consumer classes

VB6 allows you to create databinding-aware classes, none of which are supported by VB.NET. More precisely, in VB6 you can create

- ADO data source classes or user controls (by setting the DataSourceBehavior attribute to 1-vbDataSource); for example you might create a custom version of the AdoDC control and bind other controls to it.
- ADO data consumer classes or user controls, that can be bound to an AdoDC control, a DataEnvironment object, an ADO Recordset object, or an ADO data source object. Two different flavours of data consumer classes are supported: simplex-bound (DataBindingBehavior=1-vbSimpleBound) and complex-bound (DataBindingBehavior=2-vbComplexBound). For example, a textbox-like user control might be defined as a simple-bound consumer class, because it displays data taken from a single record exposed by the data source, whereas a grid-like user control might be defined as a complex-bound class, because it displays data from multiple records.

VB Migration Partner supports data source classes and simple-bound data consumer classes and user controls (but not complex-bound data consumer classes and user controls).

AddIn classes

VB6 addin classes aren't supported by VB.NET. Microsoft Visual Basic 6 and Microsoft Visual Studio 2005's object models are too different for this feature to be migrated automatically. (Manual translation isn't exactly easier either.)

VB Migration Partner doesn't support addin clases.

WebClass components

VB.NET support WebClass components. VB6 applications that used WebClass components should be converted to ASP.NET for better speed, more power, and easier maintenance.

VB Migration Partner doesn't support these components.

DHTML Page components

VB.NET doesn't support DHTML Page components.

VB Migration Partner drops these components when converting VB6 applications, and emits one migration warning for each DHTML Page component.

Built-in and External Objects

App

VB.NET doesn't directly support the App object. However, most of its properties can be mapped to members of the My.Application object, as indicated below:

Comments: My.Application.Info.Description

CompanyName: My.Application.Info.CompanyName

ExeName: My.Application.Info.AssemblyName

FileDescription: My.Application.Info.Title

LegalCopywright: My.Application.Info.Copywright

LegalTrademarks: My.Application.Info.Trademark

Major: My.Application.Info.Version.Major

Minor: My.Application.Info.Version.Minor

Path: My.Application.Info.DirectoryPath

ProductName: My.Application.Info.ProductName

Revision: My.Application.Info.Version.Build

Title: My.Application.Info.Title

The Title property is read-write in VB6 and readonly in VB.NET.

A few properties can be rendered by means of specific methods in the .NET Framework:

The **hInstance** VB6 property corresponds to the Process.GetCurrentProcess().Id method. The **ThreadID** VB6 property corresponds to AppDomain.GetCurrentThreadID() methods; however, notice that the GetCurrentThreadID method has been obsoleted in .NET 2.0 because it doesn't account for cases when the current application runs on a lightweight thread (a.k.a. fibers). More more info, see http://go.microsoft.com/fwlink/?linkid=14202.

A few App members are used to log application events - namely the **LogMode** and **LogPath** properties and the **StartLogging** and **LogEvent** methods. These members have no direct counterparts in the .NET Framework. The simplest way to implement logging to file or the Windows log is by means of the methods and properties of My.Application.Log object.

The **PrevInstance** property has no direct equivalent under the .NET Framework. However, VB.NET notifies you when the current application is the second instance of another (already running) application by firing the StartupNextInstance event. You could then use this event to initialize a global variable to True

```
Public App_PrevInstance = False

Private Sub MyApplication_StartupNextInstance(ByVal sender As Object, _
    ByVal e As Microsoft.VisualBasic.ApplicationServices.StartupNextInstanceEventArgs)
_
    Handles Me.StartupNextInstance
    ' remember that this application has a previous instance
    App_PrevInstance = True
End Sub
```

For VB.NET to fire the StartupNextInstance event, it is required that the application be marked as a single-instance application, in the Application tab of the My Project page.

A few properties have no meaning under the .NET Framework, therefore they can be treated as constant values (or just ignored) under VB.NET:

HelpFile: VB.NET doesn't offer an automatic mechanism for implementing help, therefore this property is meaningless under .NET and can be ignored.

NonModalAllowed: the .NET Framework never prevents modeless windows, therefore this property can be assumed to always return True under VB.NET.

OleRequestPending* and **OleServerBusy*** properties: .NET apps can never receive an OleRequestPending or an OleServerBusy error, therefore all these properties can be safely ignored under VB.NET.

RetainedProject: the .NET doesn't support the notion of retained projects, hence this property can be assumed to be always equal to False under VB.NET.

StartMode: the .NET Framework doesn't support ActiveX EXE application, therefore this property is always equal to 0-vbSModeStandalone for EXE projects or equal to 1-vsSModeAutomation for DLL projects.

TaskVisible: the .NET Framework doesn't offer a straightforward way to hide an application from the Task Manager, therefore this property should be considered to be equal to True and nonwritable under VB.NET.

UnattendedApp: .NET doesn't have a property that corresponds to this, therefore this property should be always considered as equal to False.

VB Migration Partner translates the App object to the **App6** object; all properties and methods are supported, including all those related to event logging. A few members behave slightly differently from VB6, for the reasons explained above.

VB Migration Partner ignores assignments to the Title property, or throws a runtime error if **VB6Config.ThrowOnUnsupportedMembers** property is set to True. Under VB.NET you can change the text displayed in the Task Manager for the current application by changing the main window's Text property.

Binding and BindingCollection

These two VB6 objects are defined in the Microsoft Data Binding Collection type library (MSBIND.DLL). The Upgrade Wizard converts them using the MBinding and MBindingCollection types in the Microsoft.VisualBasic.Compatibility.Data assembly.

VB Migration Partner converts these objects using the VB6Binding and VB6BindingCollection types, defined in VB Migration Partner's support library. No dependency from the Microsoft.VisualBasic.Compatibility.Data assembly is necessary.

Clipboard

The VB6 Clipboard object corresponds to the My.Computer.Clipboard objects. There is a one-to-one correspondence for most methods, except the **GetFormat** method corresponds to the ContainsData .NET method; the enumerated type used for the argument is different.

VB Migration Partner translates the Clipboard object to the **Clipboard6** object; all properties and methods are supported, including minor details and quirks.

Collection

VB.NET fully supports the Collection object (defined in Microsoft.VisualBasic namespace) and the Upgrade Wizard correctly converts references to this type.

You can often improve the performance of migrated code by replacing the Collection object with one of the types defined in the System.Collection namespace, for example the ArrayList or HashTable objects. If the collection holds items of same type, you can also use the strong-typed List(Of T) or Dictionary(Of T) types, defined in the System.Collection.Generic namespace.

The main issue with these .NET objects is that none of them supports all the features of the VB6 Collection, for example the ability to reference an item both using a string key and its positional index.

Like the Upgrade Wizard, VB Migration Partner migrates references to the Collection object using the .NET Microsoft.VisualBasic.Collection type. However, you can optionally map the Collection object to the more powerful **VB6Collection** type, which gives you the same high performance as the .NET native objects while supporting all the features of the VB6 object.

DataEnvironment

The .NET Framework doesn't support the DataEnvironment object or any other objects that resembles the DataEnvironment object. The Upgrade Wizard converts these classes by generating the code for a class that inherit from the BaseDataEnvironment type defined in the Microsoft.VisualBasic.Compatibility.Data assembly. However, the translation isn't always perfect. For example, the Name property isn't supported and the Upgrade Wizard fails to automatically converts default properties in some cases. For example, if a DataEnvironment class exposes a method named AuthorsByState which takes a string argument, then the following statement:

```
DataEnvironment1.AuthorsByState txtState   ' txtState is a TextBox
```

isn't migrated correctly because the Upgrade Wizard fails to append the default Text property to the txtState reference.

VB Migration Partner follows the same approach as the Upgrade Wizard, except the generated class inherits from the **VB6DataEnvironment** class defined in VB Migration Partner's support library. Unlike the Upgrade Wizard, all the DataEnvironment members are fully supported, as is the generation of default properties.

DataObject

The VB6 DataObject is used in drag-and-drop scenarios and holds that data taken from the source control and about to be dropped on the target control. VB6 and VB.NET implement drag-and-drop in completely different and incompatible ways, therefore the Upgrade Wizard doesn't convert this object.

VB Migration Partner fully supports OLE drag-and-drop properties, methods, and events, and maps the DataObject type to the **VB6DataObject** type defined in VB Migration Partner's support library. All DataObject members are supported, including the ability to store file names when dragging elements from Windows Explorer.

Dictionary

The Dictionary object is defined in the Scripting type library (SCRRUN.DLL) and is often used by VB6 developers as an high-speed alternative to the Collection object. The Upgrade Wizard doesn't convert this object and leaves a reference to the original COM type library.

If you want to get rid of all dependencies from COM objects you should convert this object using the System.Collections.HashTable object. Even better, if all items of the dictionary of same type, you can convert it to System.Collections.Generic.Dictionary(Of T) type and achieve better performance and type safety.

VB Migration Partner converts Dictionary objects using the **VB6Dictionary** type, defined in VB Migration Partner's support library. All members are fully supported.

FileSystemObject

The FileSystemObject type and its ancillary types - e.g. Drive, Folder, File, TextFile, etc. - are defined in the Scripting type library (SCRRUN.DLL) and are often used by VB6 developers to manipulate files and folders. The Upgrade Wizard doesn't convert this object and leaves a reference to the original COM type library.

If you want to get rid of all dependencies from COM objects you should convert this object using the types defined in the System.IO namespace, for example DriveInfo, DirectoryInfo, and FileInfo. However, there are many subtle differences between the original COM objects and their closest counterparts in the .NET Framework. Just to mention one, the File.Copy method works both in COM and .NET, however only the COM version can handle wildcards.

VB Migration Partner converts the FileSystemObject type and its ancillary objects by mapping them to types defined in VB Migration Partner's support library. All members are supported and all COM dependencies are removed, yet functional equivalence with the original code is fully preserved.

Forms

The VB6 Forms collection broadly corresponds to the OpenForms collection of the System.Windows.Forms.Application object. However, there is an important difference: the VB6 Form collection includes all loaded forms, whereas the VB.NET OpenForms collection includes only the forms that are currently visible. This difference implies that the number of items in the Forms and

OpenForms collection can be different, which makes quite hard to correctly translate the following VB6 code:

```
' unload all forms (and indirectly terminates the current program)
For n = Forms.Count - 1 To 0 Step -1
    Unload Forms(n)
Next
```

In fact, if you translate this code to VB.NET using the OpenForms collection, the forms that are loaded but not visible won't be unloaded, thus preventing the current application from terminating as expecting.

VB Migration Partner translates the Forms collection to the **Forms6** collection; as in VB6, the Forms6 collection contains all the loaded forms, be them visible or hidden.

LicenseInfo and Licenses

The .NET Framework supports a licensing mechanism that is completely different from the VB6 mechanism, hence VB.NET doesn't support the LicenseInfo object and the Licenses collection.

VB Migration Partner does support these objects and all its members. Converted VB.NET code therefore create a LicenseInfo object and add it to (or remove it from) the Licenses collection. However, the LicenseInfo object does nothing and its only purpose is to avoid compilation errors in migrated projects.

ObjectContext and SecurityProperty

The ObjectContext and SecurityProperty objects are the most important classes defined in the COM+ Services Type Library (COMSVCS.DLL) and are used by VB6 developers when building COM+ applications.

The Upgrade Wizard correctly migrates COM+ classes into .NET types that inherit from System.EnterpriseServices.ServicedComponent and that are marked with an appropriate Transaction attribute. References to the ObjectContext type are migrated as instances of the

System.EnterpriseServices.ContextUtils helper class, which exposes methods such as SetAbort, SetComplete, etc. Not all members are supported, though.

VB Migration Partner fully supports the ObjectContext and SecurityProperty objects, which are converted using types defined in VB Migration Partner's support library. All members are supported. Notice that only these two classes are converted into native .NET objects; if the application uses other classes in the COMSVCS type library, a reference to this COM library is added to the migrated VB.NET project.

Printer and Printers

The VB6 Printer object and the Printers collection don't directly correspond to any .NET Framework object. The Upgrade Wizard 2008 manages to do the conversion by mapping these objects to the Printer and Printers objects defined in the Visual Basic Power Pack library. However, the Printer object in the Visual Basic Power Pack library lacks a few members of the VB6 object, namely the **DrawMode, DeviceName, hDC, Port,** and **Zoom** properties

VB Migration Partner converts references to the Printer object and the Printers collection using the Printer6 and **Printers6** types defined in the support library. All members are supported and no dependency from the Visual Basic Power Pack library is introduced.

PropertyBag

VB.NET doesn't support the PropertyBag object. The Upgrade Wizard doesn't migrate statements that use this object. You can simulate this object by writing data into a MemoryStream objects and then use its GetBuffer method to read the stream contents as a Byte array. You can even serialize entire object trees, provided that all the objects in the tree are marked with the Serializable attribute.

VB Migration Partner fully supports the PropertyBag object and its ReadProperty, WriteProperty, and Contents members.

RegExp

The RegExp type and its ancillary objects - namely MatchCollection, Match, and SubMatches - are defined in the VBScript type library (VBSCRIPT.DLL) and are sometimes used by VB6 developers to

regular expressions. The Upgrade Wizard doesn't convert this object and leaves a reference to the original COM type library.

If you want to get rid of all dependencies from COM objects you should convert this object using the types defined in the System.Text.RegularExpressions namespace. However, the correspondence between the COM and the .NET types isn't perfect and you must be account for minor adjustments in code. For example, regular expressions options are expressed as properties in the COM version (e.g. MultiLine, IgnoreCase), whereas they are specified as method arguments in the .NET version.

VB Migration Partner fully supports the RegExp types and related objects, and maps them to fully managed classes defined in VB Migration Partner's support library. All members are supported and all COM dependencies are removed, yet functional equivalence with the original code is fully preserved.

Screen

The VB6 Screen object broadly corresponds to the System.Windows.Forms.Screen .NET object, however a few members must be mapped to different properties and methods of the .NET Framework.

The **TwipsPerPixelX** and **TwipsPerPixelY** properties can be translated using methods defined in the Microsoft.VisualBasic.Compatibility assembly:

```
twipsPerPixelX = Microsoft.VisualBasic.Compatibility.VB6.Support.PixelsToTwipsX(1)
twipsPerPixelY = Microsoft.VisualBasic.Compatibility.VB6.Support.PixelsToTwipsY(1)
```

The **Width** and **Height** properties can be rendered by means of the Bounds property of the Screen object, as in:

```
width = System.Windows.Forms.Screen.PrimaryScreen.Bounds.Width
height = System.Windows.Forms.Screen.PrimaryScreen.Bounds.Height
```

Notice that the Bounds object's properties return values in pixels, therefore you should multiply the result by TwipsPerPixelX or TwipsPerPixelY to get the number of twips.

The **Fonts** collection and the FontCount property can be approximately rendered under VB.NET by means of the Families member the System.Drawing.FontFamily object:

```
fonts = System.Drawing.FontFamily.Families
fontCount = System.Drawing.FontFamily.Families.Length
```

The **ActiveForm** property has no direct .NET counterpart. Under VB.NET you can find the current form by iterating over the OpenForms collection until you find the form that has the input focus:

```
For Each frm As Form In Application.OpenForms
    If frm.ContainsFocus Then activeForm = frm: Exit For
Next
```

If the application is an MDI application, you can retrieve the active MDI child form as follows:

```
activeForm = MdiForm.ActiveMdiChild
```

The **ActiveControl** property has no direct .NET counterpart. You can simulate the Screen.ActiveControl property by searching for the active form first (see above) and then querying the ActiveControl property of the active form.

The **MousePointer** and **MouseIcon** properties have no direct .NET counterpart. You can simulate the Screen.MousePointer property by searching for the active form first (see above) and then querying the Cursor property of the active form.

VB Migration Partner translates the Screen object to the **Screen6** object. All properties and methods are supported, with just one limitation: the MouseIcon property always returns Nothing; attempts to assign it a non-Nothing value are ignored or raise an exception if the **VB6Config.ThrowOnUnsupportedMembers** property is set to True.

StdDataFormat and StdDataFormats

The StdDataFormat object and the StdDataFormats collection are defined in the Microsoft Data Formatting Object Library (MSSTDFMT.DLL) and are used by VB6 developers in conjunction with data binding. The Upgrade Wizard maps these objects to types in the Microsoft.StdFormat assembly. This is basically the Primary Interop Assembly (PIA) of the original COM library, hence a degree of dependence from COM still exists in converted .NET applications.

VB Migration Partner converts these objects using the VB6DataFormat and VB6DataFormats types, defined in VB Migration Partner's support library. These types supports all the members of the VB6 objects and preserve functional equivalence and allow you to get rid of all COM dependencies at the same time.

VBControlExtender

Under VB6 this object is typically used in conjunction with the Controls.Add method, because it gives access to a number of Extender properties (e.g. Container, Enabled, Left, Top, HelpContextID, and others), methods (Move, SetFocus, ZOrders) and events (GotFocus, LostFocus, Validate. Another important feature of the VBControlExtender object is the ability to handle events in late-bound mode, thanks to its ObjectEvent event.

The Upgrade Wizard doesn't support the Controls.Add method and attempts to converts the VBControlExtender object using the System.Windows.Forms.AxHost type. However, the AxHost type lacks many of the members originally exposed by VBControlExtender, including anything comparable to ObjectEvent. Worse, the AxHost object can hold only references to ActiveX controls, therefore you can't use it when dynamically adding a standard .NET control.

VB Migration Partner fully supports the Controls.Add method, the VBControlExtender type and all its members, including the ObjectEvent event. A VBControlExtender object can be associated with both a standard .NET control or an ActiveX control.