Avoiding Naming Conflicts

{ewc HLP95EN.DLL,DYNALINK,"See Also":"vaconAvoidingNamingConflictsC"} {ewc HLP95EN.DLL,DYNALINK,"Specifics":"vaconAvoidingNamingConflictsS"}

A naming conflict occurs when you try to create or use an identifier that was previously defined. In some cases, naming conflicts generate errors such as "`Ambiguous name detected`" or "`Duplicate declaration in current scope`". Naming conflicts that go undetected can result in bugs in your code that produce erroneous results, especially if you do not explicitly declare all variables before first use.

You can avoid most naming conflicts by understanding the scoping characteristics of identifiers for data, objects, and procedures. Visual Basic has three scoping levels: procedure-level, private module-level, and public module-level.

A naming conflict can occur when an identifier:

- Is visible at more than one scoping level.
- Has two different meanings at the same level.

For example, procedures in separate modules can have the same name. Therefore, you can define a procedure named `MySub` in modules named `Mod1` and `Mod2`. No conflicts occur if each procedure is called only from other procedures in its own module. However, an error can occur if `MySub` is called from a third module, and no qualification is provided to distinguish between the two `MySub` procedures.

Most naming conflicts can be resolved by preceding each identifier with a qualifier that consists of the module name and, if necessary, a project name. For example:

```
YourProject.YourModule.YourSub MyProject.MyModule.MyVar
```

The preceding code calls the **Sub** procedure `YourSub` and passes `MyVar` as an argument. You can use any combination of qualifiers to differentiate identical identifiers.

Visual Basic matches each reference to an identifier with the "closest" declaration of a matching identifier. For example, if `MyID` is declared **Public** in two modules in a project (`Mod1` and `Mod2`), you can specify the `MyID` declared in `Mod2` without qualification from within `Mod2`, but you must qualify it as `Mod2.MyID` to specify it in `Mod1`. This is also true if `Mod2` is in a different but directly referenced project. However, if `Mod2` is in an indirectly referenced project, that is, a project referenced by the project you directly reference, references to the `Mod2` variable named `MyID` must always be qualified with the project name. If you reference `MyID` from a third, directly referenced module, the match is made with the first declaration encountered by searching:

- Directly referenced projects, in the order that they appear in the **References** dialog box of the **Tools** menu.
- The modules of each project. Note that there is no inherent order to the modules in the project.

You can't reuse names of host-application objects, for example, R1C1 in Microsoft Excel, at different scoping levels.

> **Tip** Typical errors caused by naming conflicts include ambiguous names, duplicate declarations, undeclared identifiers, and procedures that are not found. By beginning each module with an **Option Explicit** statement to force explicit declarations of variables before they are used, you can avoid some potential naming conflicts and identifier-related bugs.

Calling Procedures with the Same Name

{ewc HLP95EN.DLL,DYNALINK,"See Also":"vaconCallingProceduresWithTheSameNameC"} {ewc HLP95EN.DLL,DYNALINK,"Specifics":"vaconCallingProceduresWithTheSameNameS"}

You can call a procedure located in any module in the same project as the active module just as you would call a procedure in the active module. However, if two or more modules contain a procedure with the same name, you must specify a module name in the calling statement, as shown in the following example:

```
Sub Main()
   Module1.MyProcedure
End Sub
```

If you give the same name to two different procedures in two different projects, you must specify a project name when you call that procedure. For example, the following procedure calls the `Main` procedure in the `MyModule` module in the `MyProject.vbp` project.

```
Sub Main()
   [MyProject.vbp].[MyModule].Main
End Sub
```

> **Note** Different applications have different names for a project. For example, in Microsoft Access, a project is called a database (.mdb); in Microsoft Excel, it's called a workbook (.xls)

Tips for Calling Procedures

- If you rename a module or project, be sure to change the module or project name wherever it appears in calling statements; otherwise, Visual Basic will not be able to find the called procedure. You can use the **Replace** command on the **Edit** menu to find and replace text in a module.
- To avoid naming conflicts among referenced projects, give your procedures unique names so you can call a procedure without specifying a project or module.

Calling Property Procedures

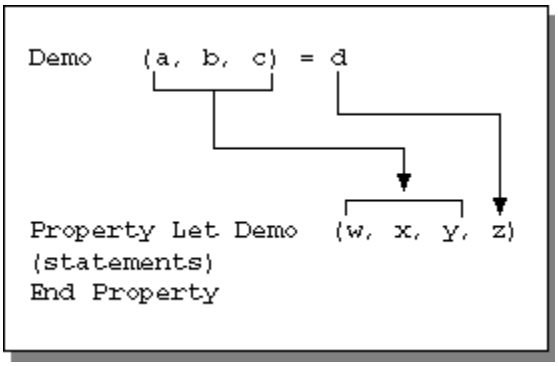
{ewc HLP95EN.DLL,DYNALINK,"See Also":"vaconCallingPropertyProceduresC"} {ewc HLP95EN.DLL,DYNALINK,"Specifics":"vaconCallingPropertyProceduresS"}

The following table lists the syntax for calling property procedures:

Property Procedure	Syntax
Property Let	[*object*.]*propname*(*arguments*)] = *argument*
Property Get	*varname* = [*object*.]*propname*(*arguments*)]
Property Set	**Set** [*object*.]*propname*[.(*arguments*)] = *varname*

When you call a **Property Let** or **Property Set** procedure, one argument always appears on the right side of the equal sign (**=**).

When you declare a **Property Let** or **Property Set** procedure with multiple arguments, Visual Basic passes the argument on the right side of the call to the last argument in the **Property Let** or **Property Set** declaration. For example, the following diagram shows how arguments in the **Property** procedure call relate to arguments in the **Property Let** declaration:



In practice, the only use for property procedures with multiple arguments is to create arrays of properties.

Calling Sub and Function Procedures

{ewc HLP95EN.DLL,DYNALINK,"See Also":"vaconCallingSubandFunctionProceduresC"} {ewc HLP95EN.DLL,DYNALINK,"Specifics":"vaconCallingSubandFunctionproceduresS"}

To call a **Sub** procedure from another procedure, type the name of the procedure and include values for any required arguments. The **Call** statement is not required, but if you use it, you must enclose any arguments in parentheses.

You can use a **Sub** procedure to organize other procedures so they are easier to understand and debug. In the following example, the **Sub** procedure `Main` calls the **Sub** procedure `MultiBeep`, passing the value 56 for its argument. After `MultiBeep` runs, control returns to `Main`, and `Main` calls the **Sub** procedure `Message`. `Message` displays a message box; when the user clicks **OK**, control returns to `Main`, and `Main` finishes.

```
Sub Main()
   MultiBeep 56
   Message
End Sub

Sub MultiBeep(numbeeps)
   For counter = 1 To numbeeps
      Beep
   Next counter
End Sub

Sub Message()
   MsgBox "Time to take a break!"
End Sub
```

Calling Sub Procedures with More than One Argument

The following example shows two ways to call a **Sub** procedure with more than one argument. The second time `HouseCalc` is called, parentheses are required around the arguments because the **Call** statement is used.

```
Sub Main()
   HouseCalc 99800, 43100
   Call HouseCalc(380950, 49500)
End Sub

Sub HouseCalc(price As Single, wage As Single)
   If 2.5 * wage <= 0.8 * price Then
      MsgBox "You cannot afford this house."
   Else
      MsgBox "This house is affordable."
   End If
End Sub
```

Using Parentheses when Calling Function Procedures

To use the return value of a function, assign the function to a variable and enclose the arguments in parentheses, as shown in the following example.

```
Answer3 = MsgBox("Are you happy with your salary?", 4, "Question 3")
```

If you're not interested in the return value of a function, you can call a function the same way you call a **Sub** procedure. Omit the parentheses, list the arguments, and do not assign the function to a variable, as shown in the following example.

```
MsgBox "Task Completed!", 0, "Task Box"
```

> **Caution** If you include parentheses in the preceding example, the statement causes a syntax error.

Passing Named Arguments

A statement in a **Sub** or **Function** procedure can pass values to called procedures using named arguments. You can list named arguments in any order. A named argument consists of the name of the argument followed by a colon and an equal sign (**:=**), and the value assigned to the argument.

The following example calls the **MsgBox** function using named arguments with no return value.

```
MsgBox Title:="Task Box", Prompt:="Task Completed!"
```

The following example calls the **MsgBox** function using named arguments. The return value is assigned to the variable `answer3`.

```
answer3 = MsgBox(Title:="Question 3", _
Prompt:="Are you happy with your salary?", Buttons:=4)
```

Creating Object Variables

{ewc HLP95EN.DLL,DYNALINK,"See Also":"vaconCreatingObjVarC"} {ewc HLP95EN.DLL,DYNALINK,"Specifics":"vaconCreatingObjVarS"}

You can treat an object variable exactly the same as the object to which it refers. You can set or return the properties of the object or use any of its methods.

To create an object variable:

1. Declare the object variable.
2. Assign the object variable to an object.

Declaring an Object Variable

Use the **Dim** statement or one of the other declaration statements (**Public**, **Private**, or **Static**) to declare an object variable. A variable that refers to an object must be a **Variant**, an **Object**, or a specific type of object. For example, the following declarations are valid:

```
' Declare MyObject as Variant data type.
Dim MyObject
' Declare MyObject as Object data type.
Dim MyObject As Object
' Declare MyObject as Font type.
Dim MyObject As Font
```

> **Note** If you use an object variable without declaring it first, the data type of the object variable is **Variant** by default.

You can declare an object variable with the **Object** data type when the specific object type is not known until the procedure runs. Use the **Object** data type to create a generic reference to any object.

If you know the specific object type, you should declare the object variable as that object type. For example, if the application contains a Sample object type, you can declare an object variable for that object using either of these statements:

```
Dim MyObject As Object  ' Declared as generic object.
Dim MyObject As Sample  ' Declared only as Sample object.
```

Declaring specific object types provides automatic type checking, faster code, and improved readability.

Assigning an Object Variable to an Object

Use the **Set** statement to assign an object to an object variable. You can assign an object expression or **Nothing**. For example, the following object variable assignments are valid:

```
Set MyObject = YourObject  ' Assign object reference.
Set MyObject = Nothing  ' Discontinue association.
```

You can combine declaring an object variable with assigning an object to it by using the **New** keyword with the **Set** statement. For example:

```
Set MyObject = New Object  ' Create and Assign
```

Setting an object variable equal to **Nothing** discontinues the association of the object variable with any specific object. This prevents you from accidentally changing the object by changing the variable. An object variable is always set to **Nothing** after closing the associated object so you can test whether or not the object variable points to a valid object. For example:

```
If Not MyObject Is Nothing Then
   ' Variable refers to valid object.
   . . .
End If
```

Of course, this test can never determine with absolute certainty whether or not a user has closed the application containing the object to which the object variable refers.

Referring to the Current Instance of an Object

Use the **Me** keyword to refer to the current instance of the object where the code is running. All procedures associated with the current object have access to the object referred to as **Me**. Using **Me** is particularly useful for passing information about the current instance of an object to a procedure in another module. For example, suppose you have the following procedure in a module:

```
Sub ChangeObjectColor(MyObjectName As Object)
   MyObjectName.BackColor = RGB(Rnd * 256, Rnd * 256, Rnd * 256)
End Sub
```

You can call the procedure and pass the current instance of the object as an argument using the following statement:

```
ChangeObjectColor Me
```

Creating Recursive Procedures

{ewc HLP95EN.DLL,DYNALINK,"See Also":"vaconCreatingRecursiveProceduresC"} {ewc HLP95EN.DLL,DYNALINK,"Specifics":"vaconCreatingRecursiveProceduresS"}

Procedures have a limited amount of space for variables. Each time a procedure calls itself, more of that space is used. A procedure that calls itself is a recursive procedure. A recursive procedure that continuously calls itself eventually causes an error. For example:

```
Function RunOut(Maximum)
   RunOut = RunOut(Maximum)
End Function
```

This error may be less obvious when two procedures call each other indefinitely, or when some condition that limits the recursion is never met. Recursion does have its uses. For example, the following procedure uses a recursive function to calculate factorials:

```
Function Factorial (N)
   If N <= 1 Then ' Reached end of recursive calls.
      Factorial = 1  ' (N = 0) so climb back out of calls.
   Else  ' Call Factorial again if N > 0.
      Factorial = Factorial(N - 1) * N
   End If
End Function
```

You should test your recursive procedure to make sure it does not call itself so many times that you run out of memory. If you get an error, make sure your procedure is not calling itself indefinitely. After that, try to conserve memory by:

- Eliminating unnecessary variables.
- Using data types other than **Variant**.
- Re-evaluating the logic of the procedure. You can often substitute nested loops for recursion.

Declaring Arrays

{ewc HLP95EN.DLL,DYNALINK,"See Also":"vaconDeclaringArraysC"} {ewc HLP95EN.DLL,DYNALINK,"Specifics":"vaconDeclaringArraysS"}

Arrays are declared the same way as other variables, using the **Dim**, **Static**, **Private**, or **Public** statements. The difference between scalar variables (those that aren't arrays) and array variables is that you generally must specify the size of the array. An array whose size is specified is a fixed-size array. An array whose size can be changed while a program is running is a dynamic array.

Whether an array is indexed from 0 or 1 depends on the setting of the **Option Base** statement. If **Option Base 1** is not specified, all array indexes begin at zero.

Declaring a Fixed Array

In the following line of code, a fixed-size array is declared as an **Integer** array having 11 rows and 11 columns:

```
Dim MyArray(10, 10) As Integer
```

The first argument represents the rows; the second argument represents the columns.

As with any other variable declaration, unless you specify a data type for the array, the data type of the elements in a declared array is **Variant**. Each numeric **Variant** element of the array uses 16 bytes. Each string **Variant** element uses 22 bytes. To write code that is as compact as possible, explicitly declare your arrays to be of a data type other than **Variant**. The following lines of code compare the size of several arrays:

```
' Integer array uses 22 bytes (11 elements * 2 bytes).
ReDim MyIntegerArray(10) As Integer

' Double-precision array uses 88 bytes (11 elements * 8 bytes).
ReDim MyDoubleArray(10) As Double

' Variant array uses at least 176 bytes (11 elements * 16 bytes).
ReDim MyVariantArray(10)

' Integer array uses 100 * 100 * 2 bytes (20,000 bytes).
ReDim MyIntegerArray (99, 99) As Integer

' Double-precision array uses 100 * 100 * 8 bytes (80,000 bytes).
ReDim MyDoubleArray (99, 99) As Double

' Variant array uses at least 160,000 bytes (100 * 100 * 16 bytes).
ReDim MyVariantArray(99, 99)
```

The maximum size of an array varies, based on your operating system and how much memory is available. Using an array that exceeds the amount of RAM available on your system is slower because the data must be read from and written to disk.

Declaring a Dynamic Array

By declaring a dynamic array, you can size the array while the code is running. Use a **Static**, **Dim**, **Private**, or **Public** statement to declare an array, leaving the parentheses empty, as shown in the following example.

```
Dim sngArray() As Single
```

> **Note** You can use the **ReDim** statement to declare an array implicitly within a procedure. Be careful not to misspell the name of the array when you use the **ReDim** statement. Even if the **Option Explicit** statement is included in the module, a second array will be created.

In a procedure within the array's scope, use the **ReDim** statement to change the number of dimensions, to define the number of elements, and to define the upper and lower bounds for each dimension. You can use the **ReDim** statement to change the dynamic array as often as necessary. However, each time you do this, the existing values in the array are lost. Use **ReDim Preserve** to expand an array while preserving existing values in the array. For example, the following statement enlarges the array `varArray` by 10 elements without losing the current values of the original elements.

```
ReDim Preserve varArray(UBound(varArray) + 10)
```

> **Note** When you use the **Preserve** keyword with a dynamic array, you can change only the upper bound of the last dimension, but you can't change the number of dimensions.

Declaring Constants

{ewc HLP95EN.DLL,DYNALINK,"See Also":"vaconDeclaringconstantsC"} {ewc HLP95EN.DLL,DYNALINK,"Specifics":"vaconDeclaringConstantsS"}

By declaring a constant, you can assign a meaningful name to a value. You use the **Const** statement to declare a constant and set its value. After a constant is declared, it cannot be modified or assigned a new value.

You can declare a constant within a procedure or at the top of a module, in the Declarations section. Module-level constants are private by default. To declare a public module-level constant, precede the **Const** statement with the **Public** keyword. You can explicitly declare a private constant by preceding the **Const** statement with the **Private** keyword to make it easier to read and interpret your code. For more information, see "Understanding Scope and Visibility" in Visual Basic Help.

The following example declares the **Public** constant `conAge` as an **Integer** and assigns it the value `34`.

```
Public Const conAge As Integer = 34
```

Constants can be declared as one of the following data types: **Boolean**, **Byte**, **Integer**, **Long**, **Currency**, **Single**, **Double**, **Date**, **String**, or **Variant**. Because you already know the value of a constant, you can specify the data type in a **Const** statement. For more information on data types, see "Data Type Summary" in Visual Basic Help.

You can declare several constants in one statement. To specify a data type, you must include the data type for each constant. In the following statement, the constants `conAge` and `conWage` are declared as **Integer**.

```
Const conAge As Integer = 34, conWage As Currency = 35000
```

Declaring Variables

{ewc HLP95EN.DLL,DYNALINK,"See Also":"vaconDeclaringvariablesC"} {ewc HLP95EN.DLL,DYNALINK,"Specifics":"vaconDeclaringVariablesS"}

When declaring variables, you usually use a **Dim** statement. A declaration statement can be placed within a procedure to create a procedure-level variable. Or it may be placed at the top of a module, in the Declarations section, to create a module-level variable.

The following example creates the variable `strName` and specifies the String data type.

```
Dim strName As String
```

If this statement appears within a procedure, the variable `strName` can be used only in that procedure. If the statement appears in the Declarations section of the module, the variable `strName` is available to all procedures within the module, but not to procedures in other modules in the project. To make this variable available to all procedures in the project, precede it with the **Public** statement, as in the following example:

```
Public strName As String
```

For information about naming your variables, see "Visual Basic Naming Rules" in Visual Basic Help.

Variables can be declared as one of the following data types: **Boolean**, **Byte**, **Integer**, **Long**, **Currency**, **Single**, **Double**, **Date**, **String** (for variable-length strings), **String *** *length* (for fixed-length strings), **Object**, or **Variant**. If you do not specify a data type, the **Variant** data type is assigned by default. You can also create a user-defined type using the **Type** statement. For more information on data types, see "Data Type Summary" in Visual Basic Help.

You can declare several variables in one statement. To specify a data type, you must include the data type for each variable. In the following statement, the variables `intX`, `intY`, and `intZ` are declared as type **Integer**.

```
Dim intX As Integer, intY As Integer, intZ As Integer
```

In the following statement, `intX` and `intY` are declared as type **Variant**; only `intZ` is declared as type **Integer**.

```
Dim intX, intY, intZ As Integer
```

You don't have to supply the variable's data type in the declaration statement. If you omit the data type, the variable will be of type **Variant**.

Using the Public Statement

You can use the **Public** statement to declare public module-level variables.

```
Public strName As String
```

Public variables can be used in any procedures in the project. If a public variable is declared in a standard module or a class module, it can also be used in any projects that reference the project where the public variable is declared.

Using the Private Statement

You can use the **Private** statement to declare private module-level variables.

```
Private MyName As String
```

Private variables can be used only by procedures in the same module.

> **Note** When used at the module level, the **Dim** statement is equivalent to the **Private** statement. You might want to use the **Private** statement to make your code easier to read and interpret.

Using the Static Statement

When you use the **Static** statement instead of a **Dim** statement, the declared variable will retain its value between calls.

Using the Option Explicit Statement

You can implicitly declare a variable in Visual Basic simply by using it in an assignment statement. All variables that are implicitly declared are of type **Variant**. Variables of type **Variant** require more memory resources than most other variables. Your application will be more efficient if you declare variables explicitly and with a specific data type. Explicitly declaring all variables reduces the incidence of naming-conflict errors and spelling mistakes.

If you don't want Visual Basic to make implicit declarations, you can place the **Option Explicit** statement in a module before any procedures. This statement requires you to explicitly declare all variables within the module. If a module includes the **Option Explicit** statement, a compile-time error will occur when Visual Basic encounters a variable name that has not been previously declared, or that has been spelled incorrectly.

You can set an option in your Visual Basic programming environment to automatically include the **Option Explicit** statement in all new modules. See your application's documentation for help on how to change Visual Basic environment options. Note that this option does not change existing code you have written.

> **Note** You must explicitly declare fixed arrays and dynamic arrays.

Declaring an Object Variable for Automation

When you use one application to control another application's objects, you should set a reference to the other application's type library. Once you set a reference, you can declare object variables according to their most specific type. For example, if you are in Microsoft Word when you set a reference to the Microsoft Excel type library, you can declare a variable of type **Worksheet** from within Microsoft Word to represent a Microsoft Excel **Worksheet** object.

If you are using another application to control Microsoft Access objects, in most cases, you can declare object variables according to their most specific type. You can also use the **New** keyword to create a new instance of an object automatically. However, you may have to indicate that it is a Microsoft Access object. For example, when you declare an object variable to represent a Microsoft Access form from within Microsoft Visual Basic, you must distinguish the Microsoft Access **Form** object from a Visual Basic **Form** object. Include the name of the type library in the variable declaration, as in the following example:

```
Dim frmOrders As New Access.Form
```

Some applications don't recognize individual Microsoft Access object types. Even if you set a reference to the Microsoft Access type library from these applications, you must declare all Microsoft Access object variables as type **Object**. Nor can you use the **New** keyword to create a new instance of the object. The following example shows how to declare a variable to represent an instance of the Microsoft Access **Application** object from an application that doesn't recognize Microsoft Access object types. The application then creates an instance of

the **Application** object.

```
Dim appAccess As Object
Set appAccess = CreateObject("Access.Application")
```

To determine which syntax an application supports, see the application's documentation.

Executing Code when Setting Properties

{ewc HLP95EN.DLL,DYNALINK,"See Also":"vaconExecutingCodeWhenSettingPropertiesC"} {ewc HLP95EN.DLL,DYNALINK,"Specifics":"vaconExecutingCodeWhenSettingPropertiesS"}

You can create **Property Let**, **Property Set**, and **Property Get** procedures that share the same name. By doing this, you can create a group of related procedures that work together. Once a name is used for a **Property** procedure, that name can't be used to name a **Sub** or **Function** procedure, a variable, or a user-defined type.

The **Property Let** statement allows you to create a procedure that sets the value of the property. One example might be a **Property** procedure that creates an inverted property for a bitmap on a form. This is the syntax used to call the **Property Let** procedure:

```
Form1.Inverted = True
```

The actual work of inverting a bitmap on the form is done within the **Property Let** procedure:

```
Private IsInverted As Boolean

Property Let Inverted(X As Boolean)
   IsInverted = X
   If IsInverted Then
      …
      (statements)
   Else
      (statements)
   End If
End Property
```

The form-level variable `IsInverted` stores the setting of your property. By declaring it **Private**, the user can only change it only using your **Property Let** procedure. Use a name that makes it easy to recognize that the variable is used for the property.

This **Property Get** procedure is used to return the current state of the `Inverted` property:

```
Property Get Inverted() As Boolean
   Inverted = IsInverted
End Property
```

Property procedures make it easy to execute code at the same time the value of a property is set. You can use property procedures to do the following processing:

- Before a property value is set to determine the value of the property.
- After a property value is set, based on the new value.

Looping Through Code

{ewc HLP95EN.DLL,DYNALINK,"See Also":"vaconLoopingThroughCodeC"} {ewc HLP95EN.DLL,DYNALINK,"Specifics":"vaconLoopingThroughCodeS"}

Using conditional statements and looping statements (also called control structures), you can write Visual Basic code that makes decisions and repeats actions. Another useful control structure, the **With** statement, lets you to run a series of statements without having to requalify an object.

Using Conditional Statements to Make Decisions

Conditional statements evaluate whether a condition is **True** or **False**, and then specify one or more statements to run, depending on the result. Usually, a condition is an expression that uses a comparison operator to compare one value or variable with another.

Choosing a Conditional Statement to Use

- If...Then...Else: Branching when a condition is **True** or **False**
- Select Case: Selecting a branch from a set of conditions

Using Loops to Repeat Code

Looping allows you to run a group of statements repeatedly. Some loops repeat statements until a condition is **False**; others repeat statements until a condition is **True**. There are also loops that repeat statements a specific number of times or for each object in a collection.

Choosing a Loop to Use

- Do...Loop: Looping while or until a condition is **True**
- For...Next: Using a counter to run statements a specified number of times
- For Each...Next: Repeating a group of statements for each object in a collection

Running Several Statements on the Same Object

In Visual Basic, usually you must specify an object before you can run one of its methods or change one of its properties. You can use the **With** statement to specify an object once for an entire series of statements.

- With: Running a series of statements on the same object

Making Faster For...Next Loops

{ewc HLP95EN.DLL,DYNALINK,"See Also":"vaconMakingFasterLoopsC"} {ewc HLP95EN.DLL,DYNALINK,"Specifics":"vaconMakingFasterLoopsS"}

Integers use less memory than the Variant data type and are slightly faster to update. However, this difference is only noticeable if you perform many thousands of operations. For example:

```
Dim CountFaster As Integer ' First case, use Integer.
For CountFaster = 0 to 32766
Next CountFaster

Dim CountSlower As Variant ' Second case, use Variant.
For CountSlower = 0 to 32766
Next CountSlower
```

The first case above takes slightly less time to run than the second case. However, if `CountFaster` exceeds 32,767, an error occurs. To fix this, you can change `CountFaster` to the Long data type, which accepts a wider range of integers. In general, the smaller the data type, the less time it takes to update. Variants are slightly slower than their equivalent data type.

Passing Arguments Efficiently

{ewc HLP95EN.DLL,DYNALINK,"See Also":"vaconPassingArgsC"} {ewc HLP95EN.DLL,DYNALINK,"Specifics":"vaconPassingArgsS"}

All arguments are passed to procedures by reference, unless you specify otherwise. This is efficient because all arguments passed by reference take the same amount of time to pass and the same amount of space (4 bytes) within a procedure regardless of the argument's data type.

You can pass an argument by value if you include the **ByVal** keyword in the procedure's declaration. Arguments passed by value consume from 2 – 16 bytes within the procedure, depending on the argument's data type. Larger data types take slightly longer to pass by value than smaller ones. Because of this, **String** and **Variant** data types generally should not be passed by value.

Passing an argument by value copies the original variable. Changes to the argument within the procedure aren't reflected back to the original variable. For example:

```
Function Factorial (ByVal MyVar As Integer)     ' Function declaration.
   MyVar = MyVar - 1
   If MyVar = 0 Then
      Factorial = 1
      Exit Function
   End If
   Factorial = Factorial(MyVar) * (MyVar + 1)
End Function

' Call Factorial with a variable S.
S = 5
Print Factorial(S)   ' Displays 120 (the factorial of 5)
Print S  ' Displays 5.
```

Without including **ByVal** in the function declaration, the preceding **Print** statements would display 1 and 0. This is because `MyVar` would then refer to variable `S`, which is reduced by 1 until it equals 0.

Because **ByVal** makes a copy of the argument, it allows you to pass a variant to the **Factorial** function above. You can't pass a variant by reference if the procedure that declares the argument is another data type.

Returning Strings from Functions

{ewc HLP95EN.DLL,DYNALINK,"See Also":"vaconReturningStringsC"} {ewc HLP95EN.DLL,DYNALINK,"Specifics":"vaconReturningStringsS"}

Some functions have two versions: one that returns a Variant data type and one that returns a String data type. The **Variant** versions are more convenient because variants handle conversions between different types of data automatically. They also allow Null to be propagated through an expression. The **String** versions are more efficient because they use less memory.

Consider using the **String** version when:

- Your program is very large and uses many variables.
- You write data directly to random-access files.

The following functions return values in a **String** variable when you append a dollar sign (**$**) to the function name. These functions have the same usage and syntax as their **Variant** equivalents without the dollar sign.

Chr$	ChrB$	*Command$
CurDir$	Date$	Dir$
Error$	Format$	Hex$
Input$	InputB$	LCase$
Left$	LeftB$	LTrim$
Mid$	MidB$	Oct$
Right$	RightB$	RTrim$
Space$	Str$	String$
Time$	Trim$	UCase$

* May not be available in all applications.

Understanding Conditional Compilation

{ewc HLP95EN.DLL,DYNALINK,"See Also":"vaconUnderstandingConditionalCompilationC"} {ewc HLP95EN.DLL,DYNALINK,"Specifics":"vaconUnderstandingConditionalCompilationS"}

You can use conditional compilation to run blocks of code selectively, for example, debugging statements comparing the speed of different approaches to the same programming task, or localizing an application for different languages.

You declare a conditional compiler constant in code with the **#Const** directive, and you denote blocks of code to be conditionally compiled with the **#If...Then...#Else** directive. The following example runs debug code or production code, based on the value of the `conDebug` variable.

```
' Declare public compilation constant in Declarations section.
#Const conDebug = 1

Sub SelectiveExecution()
   #If conDebug = 1 Then
      .           ' Run code with debugging statements.
      .
      .
   #Else
      .           ' Run normal code.
      .
      .
   #End If
End Sub
```

Understanding the Lifetime of Variables

{ewc HLP95EN.DLL,DYNALINK,"See Also":"vaconUnderstandingLifetimeC"} {ewc HLP95EN.DLL,DYNALINK,"Specifics":"vaconUnderstandingLifetimeS"}

The time during which a variable retains its value is known as its lifetime. The value of a variable may change over its lifetime, but it retains some value. When a variable loses scope, it no longer has a value.

When a procedure begins running, all variables are initialized. A numeric variable is initialized to zero, a variable-length string is initialized to a zero-length string (""), and a fixed-length string is filled with the character represented by the ASCII character code 0, or **Chr(**0**)**. Variant variables are initialized to Empty. Each element of a user-defined type variable is initialized as if it were a separate variable.

When you declare an object variable, space is reserved in memory, but its value is set to **Nothing** until you assign an object reference to it using the **Set** statement.

If the value of a variable isn't changed during the running of your code, it retains its initialized value until it loses scope.

A procedure-level variable declared with the **Dim** statement retains a value until the procedure is finished running. If the procedure calls other procedures, the variable retains its value while those procedures are running as well.

If a procedure-level variable is declared with the **Static** keyword, the variable retains its value as long as code is running in any module. When all code has finished running, the variable loses its scope and its value. Its lifetime is the same as a module-level variable.

A module-level variable differs from a static variable. In a standard module or a class module, it retains its value until you stop running your code. In a class module, it retains its value as long as an instance of the class exists. Module-level variables consume memory resources until you reset their values, so use them only when necessary.

If you include the **Static** keyword before a **Sub** or **Function** statement, the values of all the procedure-level variables in the procedure are preserved between calls.

Understanding Named and Optional Arguments

{ewc HLP95EN.DLL,DYNALINK,"See Also":"vaconUnderstandingNamedOptionalArgsC"} {ewc HLP95EN.DLL,DYNALINK,"Specifics":"vaconUnderstandingNamedOptionalArgsS"}

When you call a **Sub** or **Function** procedure, you can supply arguments positionally, in the order they appear in the procedure's definition, or you can supply the arguments by name without regard to position.

For example, the following **Sub** procedure takes three arguments:

```
Sub PassArgs(strName As String, intAge As Integer, dteBirth As Date)
   Debug.Print strName, intAge, dteBirth
End Sub
```

You can call this procedure by supplying its arguments in the correct position, each delimited by a comma, as shown in the following example:

```
PassArgs "Mary", 29, #2-21-69#
```

You can also call this procedure by supplying named arguments, delimiting each with a comma.

```
PassArgs intAge:=29, dteBirth:=#2/21/69#, strName:="Mary"
```

A named argument consists of an argument name followed by a colon and an equal sign (**:=**), followed by the argument value.

Named arguments are especially useful when you are calling a procedure that has optional arguments. If you use named arguments, you don't have to include commas to denote missing positional arguments. Using named arguments makes it easier to keep track of which arguments you passed and which you omitted.

Optional arguments are preceded by the **Optional** keyword in the procedure definition. You can also specify a default value for the optional argument in the procedure definition. For example:

```
Sub OptionalArgs(strState As String, Optional strCountry As String = "USA")
. . .
End Sub
```

When you call a procedure with an optional argument, you can choose whether or not to specify the optional argument. If you don't specify the optional argument, the default value, if any, is used. If no default value is specified, the argument is it would be for any variable of the specified type.

The following procedure includes two optional arguments, the `varRegion` and `varCountry` variables. The **IsMissing** function determines whether an optional Variant argument has been passed to the procedure.

```
Sub OptionalArgs(strState As String, Optional varRegion As Variant, _
Optional varCountry As Variant = "USA")
   If IsMissing(varRegion) And IsMissing(varCountry) Then
      Debug.Print strState
```

```
   ElseIf IsMissing(varCountry) Then
      Debug.Print strState, varRegion
   ElseIf IsMissing(varRegion) Then
      Debug.Print strState, varCountry
   Else
      Debug.Print strState, varRegion, varCountry
   End If
End Sub
```

You can call this procedure using named arguments as shown in the following examples.

```
OptionalArgs varCountry:="USA", strState:="MD"

OptionalArgs strState:= "MD", varRegion:=5
```

Understanding Objects, Properties, Methods, and Events

{ewc HLP95EN.DLL,DYNALINK,"See Also":"vaconUnderstandingobjectsC"} {ewc HLP95EN.DLL,DYNALINK,"Specifics":"vaconUnderstandingobjectsS"}

An object represents an element of an application, such as a worksheet, a cell, a chart, a form, or a report. In Visual Basic code, you must identify an object before you can apply one of the object's methods or change the value of one of its properties.

A collection is an object that contains several other objects, usually, but not always, of the same type. In Microsoft Excel, for example, the **Workbooks** object contains all the open **Workbook** objects. In Visual Basic, the **Forms** collection contains all the **Form** objects in an application.

Items in a collection can be identified by number or by name. For example, in the following procedure, `Workbooks(1)` identifies the first open **Workbook** object.

```
Sub CloseFirst()
   Workbooks(1).Close
End Sub
```

The following procedure uses a name specified as a string to identify a **Form** object.

```
Sub CloseForm()
   Forms("MyForm.frm").Close
End Sub
```

You can also manipulate an entire collection of objects if the objects share common methods. For example, the following procedure closes all open forms.

```
Sub CloseAll()
   Forms.Close
End Sub
```

A method is an action that an object can perform. For example, **Add** is a method of the **ComboBox** object, because it adds a new entry to a combo box.

The following procedure uses the **Add** method to add a new item to a **ComboBox**.

```
Sub AddEntry(newEntry as String)
   Combo1.Add newEntry
End Sub
```

A property is an attribute of an object that defines one of the object's characteristics, such as size, color, or screen location, or an aspect of its behavior, such as whether it is enabled or visible. To change the characteristics of an object, you change the values of its properties.

To set the value of a property, follow the reference to an object with a period, the property name, an equal sign (**=**), and the new property value. For example, the following procedure changes the caption of a Visual Basic form by setting the **Caption** property.

```
Sub ChangeName(newTitle)
```

```
   myForm.Caption = newTitle
End Sub
```

You can't set some properties. The Help topic for each property indicates whether you can set that property (read-write), only read the property (read-only), or only write the property (write-only).

You can retrieve information about an object by returning the value of one of its properties. The following procedure uses a message box to display the title that appears at the top of the currently active form.

```
Sub GetFormName()
   formName = Screen.ActiveForm.Caption
   MsgBox formName
End Sub
```

An event is an action recognized by an object, such as clicking the mouse or pressing a key, and for which you can write code to respond. Events can occur as a result of a user action or program code, or they can be triggered by the system.

Returning Objects

Every application has a way to return the objects it contains. However, they are not all the same, so you must refer to the Help topic for the object or collection you're using in the application to see how to return the object.

Understanding Automation

{ewc HLP95EN.DLL,DYNALINK,"See Also":"vaconUnderstandingOLEAutomationC"} {ewc HLP95EN.DLL,DYNALINK,"Specifics":"vaconUnderstandingOLEAutomationS"}

Automation (formerly OLE Automation) is a feature of the Component Object Model (COM), an industry-standard technology that applications use to expose their objects to development tools, macro languages, and other applications that support Automation. For example, a spreadsheet application may expose a worksheet, chart, cell, or range of cells — each as a different type of object. A word processor might expose objects such as an application, a document, a paragraph, a sentence, a bookmark, or a selection.

When an application supports Automation, the objects the application exposes can be accessed by Visual Basic. Use Visual Basic to manipulate these objects by invoking methods on the object or by getting and setting the object's properties. For example, you can create an Automation object named `MyObj` and write the following code to access the object:

```
MyObj.Insert "Hello, world."  ' Place text.
MyObj.Bold = True ' Format text.
If Mac = True  ' Check your platform constant
   MyObj.SaveAs  "HD:\WORDPROC\DOCS\TESTOBJ.DOC"' Save the object
(Macintosh).
Else
   MyObj.SaveAs  "C:\WORDPROC\DOCS\TESTOBJ.DOC" ' Save the object (Windows).
```

Use the following functions to access an Automation object:

Function	Description
CreateObject	Creates a new object of a specified type.
GetObject	Retrieves an object from a file.

For details on the properties and methods supported by an application, see the application documentation. The objects, functions, properties, and methods supported by an application are usually defined in the application's object library.

Understanding Parameter Arrays

{ewc HLP95EN.DLL,DYNALINK,"See Also":"vaconUnderstandingParamArraysC"} {ewc HLP95EN.DLL,DYNALINK,"Specifics":"vaconUnderstandingParamArraysS"}

A parameter array can be used to pass an array of arguments to a procedure. You don't have to know the number of elements in the array when you define the procedure.

You use the **ParamArray** keyword to denote a parameter array. The array must be declared as an array of type **Variant**, and it must be the last argument in the procedure definition.

The following example shows how you might define a procedure with a parameter array.

```
Sub AnyNumberArgs(strName As String, ParamArray intScores() As Variant)
   Dim intI As Integer

   Debug.Print strName; "    Scores"
   ' Use UBound function to determine upper limit of array.
   For intI = 0 To UBound(intScores())
      Debug.Print "          "; intScores(intI)
   Next intI
End Sub
```

The following examples show how you can call this procedure.

```
AnyNumberArgs "Jamie", 10, 26, 32, 15, 22, 24, 16

AnyNumberArgs "Kelly", "High", "Low", "Average", "High"
```

Understanding Scope and Visibility

{ewc HLP95EN.DLL,DYNALINK,"See Also":"vaconUnderstandingScopeC"} {ewc HLP95EN.DLL,DYNALINK,"Specifics":"vaconUnderstandingScopeS"}

Scope refers to the availability of a variable, constant, or procedure for use by another procedure. There are three scoping levels: procedure-level, private module-level, and public module-level.

You determine the scope of a variable when you declare it. It's a good idea to declare all variables explicitly to avoid naming-conflict errors between variables with different scopes.

Defining Procedure-Level Scope

A variable or constant defined within a procedure is not visible outside that procedure. Only the procedure that contains the variable declaration can use it. In the following example, the first procedure displays a message box that contains a string. The second procedure displays a blank message box because the variable `strMsg` is local to the first procedure.

```
Sub LocalVariable()
   Dim strMsg As String
   strMsg = "This variable can't be used outside this procedure."
   MsgBox strMsg
End Sub

Sub OutsideScope()
   MsgBox strMsg
End Sub
```

Defining Private Module-Level Scope

You can define module-level variables and constants in the Declarations section of a module. Module-level variables can be either public or private. Public variables are available to all procedures in all modules in a project; private variables are available only to procedures in that module. By default, variables declared with the **Dim** statement in the Declarations section are scoped as private. However, by preceding the variable with the **Private** keyword, the scope is obvious in your code.

In the following example, the string variable `strMsg` is available to any procedures defined in the module. When the second procedure is called, it displays the contents of the string variable `strMsg` in a dialog box.

```
' Add following to Declarations section of module.
Private strMsg As String

Sub InitializePrivateVariable()
   strMsg = "This variable can't be used outside this module."
End Sub

Sub UsePrivateVariable()
   MsgBox strMsg
End Sub
```

Note Public procedures in a standard module or class module are available to any referencing project. To limit the scope of all procedures in a module to the current project, add an **Option Private Module** statement to the Declarations section of the module. Public variables and procedures will still be available to other procedures in the current project, but not to referencing projects.

Defining Public Module-Level Scope

If you declare a module-level variable as public, it's available to all procedures in the project. In the following example, the string variable `strMsg` can be used by any procedure in any module in the project.

```
' Include in Declarations section of module.
Public strMsg As String
```

All procedures are public by default, except for event procedures. When Visual Basic creates an event procedure, the **Private** keyword is automatically inserted before the procedure declaration. For all other procedures, you must explicitly declare the procedure with the **Private** keyword if you do not want it to be public.

You can use public procedures, variables, and constants defined in standard modules or class modules from referencing projects. However, you must first set a reference to the project in which they are defined.

Public procedures, variables, and constants defined in other than standard or class modules, such as form modules or report modules, are not available to referencing projects, because these modules are private to the project in which they reside.

Understanding Variants

{ewc HLP95EN.DLL,DYNALINK,"See Also":"vaconUnderstandingVariantsC"} {ewc HLP95EN.DLL,DYNALINK,"Specifics":"vaconUnderstandingVariantsS"}

The **Variant** data type is automatically specified if you don't specify a data type when you declare a constant, variable, or argument. Variables declared as the **Variant** data type can contain string, date, time, Boolean, or numeric values, and can convert the values they contain automatically. Numeric **Variant** values require 16 bytes of memory (which is significant only in large procedures or complex modules) and they are slower to access than explicitly typed variables of any other type. You rarely use the **Variant** data type for a constant. String **Variant** values require 22 bytes of memory.

The following statements create **Variant** variables:

```
Dim myVar
Dim yourVar As Variant
theVar = "This is some text."
```

The last statement does not explicitly declare the variable `theVar`, but rather declares the variable implicitly, or automatically. Variables that are declared implicitly are specified as the **Variant** data type.

> **Tip** If you specify a data type for a variable or argument, and then use the wrong data type, a data type error will occur. To avoid data type errors, either use only implicit variables (the **Variant** data type) or explicitly declare all your variables and specify a data type. The latter method is preferred.

Understanding Visual Basic Syntax

{ewc HLP95EN.DLL,DYNALINK,"See Also":"vaconUnderstandingVBSyntaxC"} {ewc HLP95EN.DLL,DYNALINK,"Specifics":"vaconUnderstandingVBSyntaxS"}

The syntax in a Visual Basic Help topic for a method, statement, or function shows all the elements necessary to use the method, statement, or function correctly. The examples in this topic explain how to interpret the most common syntax elements.

Activate Method Syntax

object.**Activate**

In the **Activate** method syntax, the italic word "object" is a placeholder for information you supply — in this case, code that returns an object. Words that are bold should be typed exactly as they appear. For example, the following procedure activates the second window in the active document.

```
Sub MakeActive()
   Windows(2).Activate
End Sub
```

MsgBox Function Syntax

MsgBox(*prompt*[*, buttons*] [*, title*] [*, helpfile, context*]**)**

In the **MsgBox** function syntax, the bold italic words are named arguments of the function. Arguments enclosed in brackets are optional. (Do not type the brackets in your Visual Basic code.) For the **MsgBox** function, the only argument you must provide is the text for the prompt.

Arguments for functions and methods can be specified in code either by position or by name. To specify arguments by position, follow the order presented in the syntax, separating each argument with a comma, for example:

```
MsgBox "Your answer is correct!",0,"Answer Box"
```

To specify an argument by name, use the argument name followed by a colon and an equal sign (**:=**), and the argument's value. You can specify named arguments in any order, for example:

```
MsgBox Title:="Answer Box", Prompt:="Your answer is correct!"
```

The syntax for functions and some methods shows the arguments enclosed in parentheses. These functions and methods return values, so you must enclose the arguments in parentheses to assign the value to a variable. If you ignore the return value or if you don't pass arguments at all, don't include the parentheses. Methods that don't return values do not need their arguments enclosed in parentheses. These guidelines apply whether you're using positional arguments or named arguments.

In the following example, the return value from the **MsgBox** function is a number indicating the selected button that is stored in the variable `myVar`. Because the return value is used, parentheses are required. Another message box then displays the value of the variable.

```
Sub Question()
   myVar = MsgBox(Prompt:="I enjoy my job.", _
      Title:="Answer Box", Buttons:="4")
   MsgBox myVar
```

```
End Sub
```

Option Statement Syntax

Option Compare {**Binary** | **Text** | **Database**}

In the **Option Compare** statement syntax, the braces and vertical bar indicate a mandatory choice between three items. (Do not type the braces in the Visual Basic statement). For example, the following statement specifies that within the module, strings will be compared in a sort orderthat is not case-sensitive.

```
Option Compare Text
```

Dim Statement Syntax

Dim *varname*[**(**[*subscripts*]**)**] [**As** *type*] [**,** *varname*[**(**[*subscripts*]**)**] [**As** *type*]] **. . .**

In the **Dim** statement syntax, the word **Dim** is a required keyword. The only required element is *varname* (the variable name). For example, the following statement creates three variables: `myVar`, `nextVar`, and `thirdVar`. These are automatically declared as **Variant** variables.

```
Dim myVar, nextVar, thirdVar
```

The following example declares a variable as a **String**. Including a data type saves memory and can help you find errors in your code.

```
Dim myAnswer As String
```

To declare several variables in one statement, include the data type for each variable. Variables declared without a data type are automatically declared as **Variant**.

```
Dim x As Integer, y As Integer, z As Integer
```

In the following statement, `x` and `y` are assigned the **Variant** data type. Only `z` is assigned the **Integer** data type.

```
Dim x, y, z As Integer
```

If you are declaring an array variable, you must include parentheses. The subscripts are optional. The following statement dimensions a dynamic array, `myArray`.

```
Dim myArray()
```

Using Arrays

{ewc HLP95EN.DLL,DYNALINK,"See Also":"vaconUsingArraysC"} {ewc HLP95EN.DLL,DYNALINK,"Specifics":"vaconUsingArraysS"}

You can declare an array to work with a set of values of the same data type. An array is a single variable with many compartments to store values, while a typical variable has only one storage compartment in which it can store only one value. Refer to the array as a whole when you want to refer to all the values it holds, or you can refer to its individual elements.

For example, to store daily expenses for each day of the year, you can declare one array variable with 365 elements, rather than declaring 365 variables. Each element in an array contains one value. The following statement declares the array variable `curExpense` with 365 elements. By default, an array is indexed beginning with zero, so the upper bound of the array is 364 rather than 365.

```
Dim curExpense(364) As Currency
```

To set the value of an individual element, you specify the element's index. The following example assigns an initial value of 20 to each element in the array.

```
Sub FillArray()
   Dim curExpense(364) As Currency
   Dim intI As Integer
   For intI = 0 to 364
      curExpense(intI) = 20
   Next
End Sub
```

Changing the Lower Bound

You can use the **Option Base** statement at the top of a module to change the default index of the first element from 0 to 1. In the following example, the **Option Base** statement changes the index for the first element, and the **Dim** statement declares the array variable `curExpense` with 365 elements.

```
Option Base 1
Dim curExpense(365) As Currency
```

You can also explicitly set the lower bound of an array by using a **To** clause, as shown in the following example.

```
Dim curExpense(1 To 365) As Currency
Dim strWeekday(7 To 13) As String
```

Storing Variant Values in Arrays

There are two ways to create arrays of **Variant** values. One way is to declare an array of Variant data type, as shown in the following example:

```
Dim varData(3) As Variant
varData(0) = "Claudia Bendel"
varData(1) = "4242 Maple Blvd"
varData(2) = 38
```

```
varData(3) = Format("06-09-1952", "General Date")
```

The other way is to assign the array returned by the **Array** function to a **Variant** variable, as shown in the following example.

```
Dim varData As Variant
varData = Array("Ron Bendel", "4242 Maple Blvd", 38, _
Format("06-09-1952", "General Date"))
```

You identify the elements in an array of **Variant** values by index, no matter which technique you use to create the array. For example, the following statement can be added to either of the preceding examples.

```
MsgBox "Data for " & varData(0) & " has been recorded."
```

Using Multidimensional Arrays

In Visual Basic, you can declare arrays with up to 60 dimensions. For example, the following statement declares a 2-dimensional, 5-by-10 array.

```
Dim sngMulti(1 To 5, 1 To 10) As Single
```

If you think of the array as a matrix, the first argument represents the rows and the second argument represents the columns.

Use nested **For...Next** statements to process multidimensional arrays. The following procedure fills a two-dimensional array with **Single** values.

```
Sub FillArrayMulti()
   Dim intI As Integer, intJ As Integer
   Dim sngMulti(1 To 5, 1 To 10) As Single

   ' Fill array with values.
   For intI = 1 To 5
      For intJ = 1 To 10
         sngMulti(intI, intJ) = intI * intJ
         Debug.Print sngMulti(intI, intJ)
      Next intJ
   Next intI
End Sub
```

Using Constants

{ewc HLP95EN.DLL,DYNALINK,"See Also":"vaconUsingConstantsC"} {ewc HLP95EN.DLL,DYNALINK,"Specifics":"vaconUsingConstantsS"}

Your code might contain frequently occurring constant values, or might depend on certain numbers that are difficult to remember and have no obvious meaning. You can make your code easier to read and maintain using constants. A constant is a meaningful name that takes the place of a number or string that does not change. You can't modify a constant or assign a new value to it as you can a variable.

There are three types of constants:

Intrinsic constants or system-defined constants are provided by applications and controls. Other applications that provide object libraries, such as Microsoft Access, Microsoft Excel, Microsoft Project , and Microsoft Word also provide a list of constants you can use with their objects, methods, and properties. You can get a list of the constants provided for individual object libraries in the Object Browser.

Visual Basic constants are listed in the Visual Basic for Applications type library, and Data Access Object (DAO) library.

> **Note** Visual Basic continues to recognize constants in applications created in earlier versions of Visual Basic or Visual Basic for Applications. You can upgrade your constants to those listed in the **Object Browser**. Constants listed in the **Object Browser** don't have to be declared in your application.

- Symbolic or user-defined constants are declared using the **Const** statement.
- Conditional compiler constants are declared using the **#Const** statement.

In earlier versions of Visual Basic, constant names were usually capitalized with underscores. For example:

```
TILE_HORIZONTAL
```

Intrinsic constants are now qualified to avoid the confusion when constants with the same name exist in more than one object library, which may have different values assigned to them. There are two ways to qualify constant names:

- By prefix
- By library reference

Qualifying Constants by Prefix

The intrinsic constants supplied by all objects appear in a mixed-case format, with a 2-character prefix indicating the object library that defines the constant. Constants from the Visual Basic for Applications object library are prefaced with "vb" and constants from the Microsoft Excel object library are prefaced with "xl". The following examples illustrate how prefixes for custom controls vary, depending on the type library.

- **vbTileHorizontal**
- **xlDialogBorder**

Qualifying Constants by Library Reference

You can also qualify the reference to a constant by using the following syntax:

[*libname*.] [*modulename*.]*constname*

The syntax for qualifying constants has these parts:

Part	Description
libname	Optional. The name of the type library that defines the constant. For most custom controls (not available on the Macintosh), this is also the class name of the control. If you don't remember the class name of the control, position the mouse pointer over the control in the toolbox. The class name is displayed in the **ToolTip**.
modulename	Optional. The name of the module within the type library that defines the constant. You can find the name of the module by using the **Object Browser**.
constname	The name defined for the constant in the type library.

For example:

```
Threed.LeftJustify
```

Using Data Types Efficiently

{ewc HLP95EN.DLL,DYNALINK,"See Also":"vaconUsingDataC"} {ewc HLP95EN.DLL,DYNALINK,"Specifics":"vaconUsingDataS"}

Unless otherwise specified, undeclared variables are assigned the Variant data type. This data type makes it easy to write programs, but it is not always the most efficient data type to use.

You should consider using other data types if:

- Your program is very large and uses many variables.
- Your program must run as quickly as possible.
- You write data directly to random-access files.

In addition to **Variant**, supported data types include **Byte**, **Boolean**, **Integer**, **Long**, **Single**, **Double**, **Currency**, **Decimal**, **Date**, **Object**, and **String**. Use the **Dim** statement to declare a variable of a specific type, for example:

```
Dim X As Integer
```

This statement declares that a variable `X` is an integer — a whole number between –32,768 and 32,767. If you try to set `X` to a number outside that range, an error occurs. If you try to set `X` to a fraction, the number is rounded. For example:

```
X = 32768      ' Causes error.
X = 5.9     ' Sets x to 6.
```

Using Do...Loop Statements

{ewc HLP95EN.DLL,DYNALINK,"See Also":"vaconUsingDoLoopC"} {ewc HLP95EN.DLL,DYNALINK,"Specifics":"vaconUsingDoLoopS"}

You can use **Do...Loop** statements to run a block of statements an indefinite number of times. The statements are repeated either while a condition is **True** or until a condition becomes **True**.

Repeating Statements While a Condition is True

There are two ways to use the **While** keyword to check a condition in a **Do...Loop** statement. You can check the condition before you enter the loop , or you can check it after the loop has run at least once.

In the following `ChkFirstWhile` procedure, you check the condition before you enter the loop. If `myNum` is set to 9 instead of 20, the statements inside the loop will never run. In the `ChkLastWhile` procedure, the statements inside the loop run only once before the condition becomes **False**.

```
Sub ChkFirstWhile()
   counter = 0
   myNum = 20
   Do While myNum > 10
      myNum = myNum - 1
      counter = counter + 1
   Loop
   MsgBox "The loop made " & counter & " repetitions."
End Sub

Sub ChkLastWhile()
   counter = 0
   myNum = 9
   Do
      myNum = myNum - 1
      counter = counter + 1
   Loop While myNum > 10
   MsgBox "The loop made " & counter & " repetitions."
End Sub
```

Repeating Statements Until a Condition Becomes True

There are two ways to use the **Until** keyword to check a condition in a **Do...Loop** statement. You can check the condition before you enter the loop (as shown in the `ChkFirstUntil` procedure), or you can check it after the loop has run at least once (as shown in the `ChkLastUntil` procedure). Looping continues while the condition remains **False**.

```
Sub ChkFirstUntil()
   counter = 0
   myNum = 20
```

```
   Do Until myNum = 10
      myNum = myNum - 1
      counter = counter + 1
   Loop
   MsgBox "The loop made " & counter & " repetitions."
End Sub

Sub ChkLastUntil()
   counter = 0
   myNum = 1
   Do
      myNum = myNum + 1
      counter = counter + 1
   Loop Until myNum = 10
   MsgBox "The loop made " & counter & " repetitions."
End Sub
```

Exiting a Do...Loop Statement from Inside the Loop

You can exit a **Do...Loop** using the **Exit Do** statement. For example, to exit an endless loop, use the **Exit Do** statement in the **True** statement block of either an **If...Then...Else** statement or a **Select Case** statement. If the condition is **False**, the loop will run as usual.

In the following example, `myNum` is assigned a value that creates an endless loop. The **If...Then...Else** statement checks for this condition, and then exits, preventing endless looping.

```
Sub ExitExample()
   counter = 0
   myNum = 9
   Do Until myNum = 10
      myNum = myNum - 1
      counter = counter + 1
      If myNum < 10 Then Exit Do
   Loop
   MsgBox "The loop made " & counter & " repetitions."
End Sub
```

Note To stop an endless loop, press ESC or CTRL+BREAK.

Using For...Next Statements

{ewc HLP95EN.DLL,DYNALINK,"See Also":"vaconUsingForNextC"} {ewc HLP95EN.DLL,DYNALINK,"Specifics":"vaconUsingForNextS"}

You can use **For...Next** statements to repeat a block of statements a specific number of times. **For** loops use a counter variable whose value is increased or decreased with each repetition of the loop.

The following procedure makes the computer beep 50 times. The **For** statement specifies the counter variable `x` and its start and end values. The **Next** statement increments the counter variable by 1.

```
Sub Beeps()
   For x = 1 To 50
      Beep
   Next x
End Sub
```

Using the **Step** keyword, you can increase or decrease the counter variable by the value you specify. In the following example, the counter variable `j` is incremented by 2 each time the loop repeats. When the loop is finished, `total` is the sum of 2, 4, 6, 8, and 10.

```
Sub TwosTotal()
   For j = 2 To 10 Step 2
      total = total + j
   Next j
   MsgBox "The total is " & total
End Sub
```

To decrease the counter variable, use a negative **Step** value. To decrease the counter variable, you must specify an end value that is less than the start value. In the following example, the counter variable `myNum` is decreased by 2 each time the loop repeats. When the loop is finished, `total` is the sum of 16, 14, 12, 10, 8, 6, 4, and 2.

```
Sub NewTotal()
   For myNum = 16 To 2 Step -2
      total = total + myNum
   Next myNum
   MsgBox "The total is " & total
End Sub
```

> **Note** It's not necessary to include the counter variable name after the **Next** statement. In the preceding examples, the counter variable name was included for readability.

You can exit a **For...Next** statement before the counter reaches its end value by using the **Exit For** statement. For example, when an error occurs, use the **Exit For** statement in the **True** statement block of either an **If...Then...Else** statement or a **Select Case** statement that specifically checks for the error. If the error doesn't occur, then the **If...Then...Else** statement is **False**, and the loop will continue to run as expected.

Using For Each...Next Statements

{ewc HLP95EN.DLL,DYNALINK,"See Also":"vaconUsingForEachC"} {ewc HLP95EN.DLL,DYNALINK,"Specifics":"vaconUsingForEachS"}

For Each...Next statements repeat a block of statements for each object in a collection or each element in an array. Visual Basic automatically sets a variable each time the loop runs. For example, the following procedure closes all forms except the form containing the procedure that's running.

```
Sub CloseForms()
   For Each frm In Application.Forms
      If frm.Caption <> Screen. ActiveForm.Caption Then frm.Close
   Next
End Sub
```

The following code loops through each element in an array and sets the value of each to the value of the index variable I.

```
Dim TestArray(10) As Integer, I As Variant
For Each I In TestArray
   TestArray(I) = I
Next I
```

Looping Through a Range of Cells

Use a **For Each...Next** loop to loop through the cells in a range. The following procedure loops through the range A1:D10 on Sheet1 and sets any number whose absolute value is less than 0.01 to 0 (zero).

```
Sub RoundToZero()
   For Each myObject in myCollection
      If Abs(myObject.Value) < 0.01 Then myObject.Value = 0
   Next
End Sub
```

Exiting a For Each...Next Loop Before it is Finished

You can exit a **For Each...Next** loop using the **Exit For** statement. For example, when an error occurs, use the **Exit For** statement in the **True** statement block of either an **If...Then...Else** statement or a **Select Case** statement that specifically checks for the error. If the error does not occur, then the **If...Then...Else** statement is **False** and the loop continues to run as expected.

The following example tests for the first cell in the range A1:B5 that does not contain a number. If such a cell is found, a message is displayed and **Exit For** exits the loop.

```
Sub TestForNumbers()
   For Each myObject In MyCollection
      If IsNumeric(myObject.Value) = False Then
         MsgBox "Object contains a non-numeric value."
         Exit For
```

```
      End If
   Next c
End Sub
```

Using If...Then...Else Statements

{ewc HLP95EN.DLL,DYNALINK,"See Also":"vaconUsingIfThenC"} {ewc HLP95EN.DLL,DYNALINK,"Specifics":"vaconUsingIfThenS"}

You can use the **If...Then...Else** statement to run a specific statement or a block of statements, depending on the value of a condition. **If...Then...Else** statements can be nested to as many levels as you need. However, for readability, you may want to use a **Select Case** statement rather than multiple levels of nested **If...Then...Else** statements.

Running Statements if a Condition is True

To run only one statement when a condition is **True**, use the single-line syntax of the **If...Then...Else** statement. The following example shows the single-line syntax, omitting the **Else** keyword:

```
Sub FixDate()
   myDate = #2/13/95#
   If myDate < Now Then myDate = Now
End Sub
```

To run more than one line of code, you must use the multiple-line syntax. This syntax includes the **End If** statement, as shown in the following example:

```
Sub AlertUser(value as Long)
   If value = 0 Then
      AlertLabel.ForeColor = "Red"
      AlertLabel.Font.Bold = True
      AlertLabel.Font.Italic = True
   End If
End Sub
```

Running Certain Statements if a Condition is True and Running Others if It's False

Use an **If...Then...Else** statement to define two blocks of executable statements: one block runs if the condition is **True**, the other block runs if the condition is **False**.

```
Sub AlertUser(value as Long)
   If value = 0 Then
      AlertLabel.ForeColor = vbRed
      AlertLabel.Font.Bold = True
      AlertLabel.Font.Italic = True
   Else
      AlertLabel.Forecolor = vbBlack
      AlertLabel.Font.Bold = False
      AlertLabel.Font.Italic = False
   End If
End Sub
```

Testing a Second Condition if the First Condition is False

You can add **ElseIf** statements to an **If...Then...Else** statement to test a second condition if the first condition is **False**. For example, the following function procedure computes a bonus based on job classification. The statement following the **Else** statement runs if the conditions in all of the **If** and **ElseIf** statements are **False**.

```
Function Bonus(performance, salary)
   If performance = 1 Then
      Bonus = salary * 0.1
   ElseIf performance = 2 Then
      Bonus = salary * 0.09
   ElseIf performance = 3 Then
      Bonus = salary * 0.07
   Else
      Bonus = 0
   End If
End Function
```

Using Parentheses in Code

{ewc HLP95EN.DLL,DYNALINK,"See Also":"vaconUsingParensC"} {ewc HLP95EN.DLL,DYNALINK,"Specifics":"vaconUsingParensS"}

Sub procedures, built-in statements, and some methods don't return a value, so the arguments aren't enclosed in parentheses. For example:

```
MySub "stringArgument", integerArgument
```

Function procedures, built-in functions, and some methods do return a value, but you can ignore it. If you ignore the return value, don't include parentheses. Call the function just as you would call a **Sub** procedure. Omit the parentheses, list any arguments , and don't assign the function to a variable. For example:

```
MsgBox "Task Completed!", 0, "Task Box"
```

To use the return value of a function, enclose the arguments in parentheses, as shown in the following example.

```
Answer3 = MsgBox("Are you happy with your salary?", 4, "Question 3")
```

A statement in a **Sub** or **Function** procedure can pass values to a called procedure using named arguments. The guidelines for using parentheses apply, whether or not you use named arguments. When you use named arguments, you can list them in any order, and you can omit optional arguments. Named arguments are always followed by a colon and an equal sign (**:=**), and then the argument value.

The following example calls the **MsgBox** function using named arguments, but it ignores the return value:

```
MsgBox Title:="Task Box", Prompt:="Task Completed!"
```

The following example calls the **MsgBox** function using named arguments and assigns the return value to the variable `answer3`:

```
answer3 = MsgBox(Title:="Question 3", _
   Prompt:="Are you happy with your salary?", Buttons:=4)
```

Using Select Case Statements

{ewc HLP95EN.DLL,DYNALINK,"See Also":"vaconUsingSelectCaseC"} {ewc HLP95EN.DLL,DYNALINK,"Specifics":"vaconUsingSelectCaseS"}

Use the **Select Case** statement as an alternative to using **ElseIf** in **If...Then...Else** statements when comparing one expression to several different values. While **If...Then...Else** statements can evaluate a different expression for each **ElseIf** statement, the **Select Case** statement evaluates an expression only once, at the top of the control structure.

In the following example, the **Select Case** statement evaluates the `performance` argument that is passed to the procedure. Note that each **Case** statement can contain more than one value, a range of values, or a combination of values and comparison operators. The optional **Case Else** statement runs if the **Select Case** statement doesn't match a value in any of the **Case** statements.

```
Function Bonus(performance, salary)
   Select Case performance
      Case 1
         Bonus = salary * 0.1
      Case 2, 3
         Bonus = salary * 0.09
      Case 4 To 6
         Bonus = salary * 0.07
      Case Is > 8
         Bonus = 100
      Case Else
         Bonus = 0
   End Select
End Function
```

Using With Statements

{ewc HLP95EN.DLL,DYNALINK,"See Also":"vaconUsingWithC"} {ewc HLP95EN.DLL,DYNALINK,"Specifics":"vaconUsingWithS"}

The **With** statement lets you specify an object or user-defined type once for an entire series of statements. **With** statements make your procedures run faster and help you avoid repetitive typing.

The following example fills a range of cells with the number 30, applies bold formatting, and sets the interior color of the cells to yellow.

```
Sub FormatRange()
   With Worksheets("Sheet1").Range("A1:C10")
      .Value = 30
      .Font.Bold = True
      .Interior.Color = RGB(255, 255, 0)
   End With
End Sub
```

You can nest **With** statements for greater efficiency. The following example inserts a formula into cell A1, and then formats the font.

```
Sub MyInput()
   With Workbooks("Book1").Worksheets("Sheet1").Cells(1, 1)
      .Formula = "=SQRT(50)"
      With .Font
         .Name = "Arial"
         .Bold = True
         .Size = 8
      End With
   End With
End Sub
```

Visual Basic Naming Rules

{ewc HLP95EN.DLL,DYNALINK,"See Also":"vaconVBNamingRulesC"} {ewc HLP95EN.DLL,DYNALINK,"Specifics":"vaconVBNamingRulesS"}

Use the following rules when you name procedures, constants, variables, and arguments in a Visual Basic module:

- You must use a letter as the first character.
- You can't use a space, period (**.**), exclamation mark (**!**), or the characters **@**, **&**, **$, #** in the name.
- Name can't exceed 255 characters in length.
- Generally, you shouldn't use any names that are the same as the functions, statements, and methods in Visual Basic. You end up shadowing the same keywords in the language. To use an intrinsic language function, statement, or method that conflicts with an assigned name, you must explicitly identify it. Precede the intrinsic function, statement, or method name with the name of the associated type library. For example, if you have a variable called `Left`, you can only invoke the **Left** function using `VBA.Left`.
- You can't repeat names within the same level of scope. For example, you can't declare two variables named `age` within the same procedure. However, you can declare a private variable named `age` and a procedure-level variable named `age` within the same module.

  Note Visual Basic isn't case-sensitive, but it preserves the capitalization in the statement where the name is declared.

Working Across Applications

{ewc HLP95EN.DLL,DYNALINK,"See Also":"vaconWorkingAcrossAppsC"} {ewc HLP95EN.DLL,DYNALINK,"Specifics":"vaconWorkingAcrossAppsS"}

Visual Basic can create new objects and retrieve existing objects from many Microsoft applications. Other applications may also provide objects that you can create using Visual Basic. See the application's documentation for more information.

To create an new object or get an existing object from another application, use the **CreateObject** function or **GetObject** function:

```
' Start Microsoft Excel and create a new Worksheet object.
Set ExcelWorksheet = CreateObject("Excel.Sheet")

' Start Microsoft Excel and open an existing Worksheet object.
Set ExcelWorksheet = GetObject("SHEET1.XLS")

' Start Microsoft Word.
Set WordBasic = CreateObject("Word.Basic")
```

Most applications provide an **Exit** or **Quit** method that closes the application whether or not it is visible. For more information on the objects, methods, and properties an application provides, see the application's documentation.

Some applications allow you to use the **New** keyword to create an object of any class that exists in its type library. For example:

```
Dim X As New Field
```

In this case, `Field` is an example of a class in the data access type library. A new instance of a **Field** object is created using this syntax. Refer to the application's documentation for information about which object classes can be created in this way.

Writing a Function Procedure

{ewc HLP95EN.DLL,DYNALINK,"See Also":"vaconWritingFunctionsC"} {ewc HLP95EN.DLL,DYNALINK,"Specifics":"vaconWritingFunctionsS"}

A **Function** procedure is a series of Visual Basic statements enclosed by the **Function** and **End Function** statements. A **Function** procedure is similar to a **Sub** procedure, but a function can also return a value. A **Function** procedure can take arguments, such as constants, variables, or expressions that are passed to it by a calling procedure. If a **Function** procedure has no arguments, its **Function** statement must include an empty set of parentheses. A function returns a value by assigning a value to its name in one or more statements of the procedure.

In the following example, the **Celsius** function calculates degrees Celsius from degrees Fahrenheit. When the function is called from the **Main** procedure, a variable containing the argument value is passed to the function. The result of the calculation is returned to the calling procedure and displayed in a message box.

```
Sub Main()
   temp = Application.InputBox(Prompt:= _
      "Please enter the temperature in degrees F.", Type:=1)
   MsgBox "The temperature is " & Celsius(temp) & " degrees C."
End Sub

Function Celsius(fDegrees)
   Celsius = (fDegrees - 32) * 5 / 9
End Function
```

Writing a Property Procedure

{ewc HLP95EN.DLL,DYNALINK,"See Also":"vaconWritingPropertiesC"} {ewc HLP95EN.DLL,DYNALINK,"Specifics":"vaconWritingPropertiesS"}

A **Property** procedure is a series of Visual Basic statements that allow a programmer to create and manipulate custom properties.

- **Property** procedures can be used to create read-only properties for forms, standard modules, and class modules.
- **Property** procedures should be used instead of **Public** variables in code that must be executed when the property value is set.
- Unlike **Public** variables, **Property** procedures can have Help strings assigned to them in the Object Browser.

When you create a **Property** procedure, it becomes a property of the module containing the procedure. Visual Basic provides the following three types of **Property** procedures:

Procedure	Description
Property Let	A procedure that sets the value of a property.
Property Get	A procedure that returns the value a property.
Property Set	A procedure that sets a reference to an object.

The syntax for declaring a **Property** procedure is:

[**Public** | **Private**] [**Static**] **Property** {**Get** | **Let** | **Set**} *propertyname_* [(*arguments*)] [**As** *type*]

statements

End Property

Property procedures are usually used in pairs: **Property Let** with **Property Get** and **Property Set** with **Property Get**. Declaring a **Property Get** procedure alone is like declaring a read-only property. Using all three **Property** procedure types together is only useful for **Variant** variables, since only a **Variant** can contain either an object or other data type information. **Property Set** is intended for use with objects; **Property Let** isn't.

The required arguments in **Property** procedure declarations are shown in the following table:

Procedure	Declaration Syntax
Property Get	**Property Get** *propname*(1, …, *n*) **As** *type*
Property Let	**Property Let** *propname*(1, …,,,, *n*, *n*+1)
Property Set	**Property Set** *propname*(1, …, *n*, *n*+1)

The first argument through the next to last argument (1, …, *n*) must share the same names and data types in all **Property** procedures with the same name.

A **Property Get** procedure declaration takes one less argument than the related **Property Let** and **Property Set** declarations. The data type of the **Property Get** procedure must be the same as the data type as the data type of the last argument (*n*+1) in the related **Property Let** and **Property Set** declarations. For example, if you declare the following **Property Let** procedure, the **Property Get** declaration must use arguments with the same name and data type as the arguments in the **Property Let** procedure.

```
Property Let Names(intX As Integer, intY As Integer, varZ As Variant)
   ‘ Statement here.
End Property

Property Get Names(intX As Integer, intY As Integer) As Variant
   ‘ Statement here.
End Property
```

The data type of the final argument in a **Property Set** declaration must be either an object type or a **Variant**.

Writing a Sub Procedure

{ewc HLP95EN.DLL,DYNALINK,"See Also":"vaconWritingSubsC"} {ewc HLP95EN.DLL,DYNALINK,"Specifics":"vaconWritingSubsS"}

A **Sub** procedure is a series of Visual Basic statements enclosed by the **Sub** and **End Sub** statements that performs actions but doesn't return a value. A **Sub** procedure can take arguments, such as constants, variables, or expressions that are passed by a calling procedure. If a **Sub** procedure has no arguments, the **Sub** statement must include an empty set of parentheses.

The following **Sub** procedure has comments explaining each line.

```
' Declares a procedure named GetInfo
' This Sub procedure takes no arguments
Sub GetInfo()
' Declares a string variable named answer
Dim answer As String
' Assigns the return value of the InputBox function to answer
answer = InputBox(Prompt:="What is your name?")
   ' Conditional If...Then...Else statement
   If answer = Empty Then
      ' Calls the MsgBox function
      MsgBox Prompt:="You did not enter a name."
   Else
      ' MsgBox function concatenated with the variable answer
      MsgBox Prompt:="Your name is " & answer
   ' Ends the If...Then...Else statement
   End If
' Ends the Sub procedure
End Sub
```

Writing Assignment Statements

{ewc HLP95EN.DLL,DYNALINK,"See Also":"vaconWritingassignmentsC"} {ewc HLP95EN.DLL,DYNALINK,"Specifics":"vaconWritingassignmentsS"}

Assignment statements assign a value or expression to a variable or constant. Assignment statements always include an equal sign (**=**). The following example assigns the return value of the **InputBox** function to the variable `yourName`.

```
Sub Question()
   Dim yourName As String
   yourName = InputBox("What is your name?")
   MsgBox "Your name is " & yourName
End Sub
```

The **Let** statement is optional and is usually omitted. For example, the preceding assignment statement can be written:

```
Let yourName = InputBox("What is your name?").
```

The **Set** statement is used to assign an object to a variable that has been declared as an object. The **Set** keyword is required. In the following example, the **Set** statement assigns a range on `Sheet1` to the object variable `myCell`:

```
Sub ApplyFormat()
Dim myCell As Range
Set myCell = Worksheets("Sheet1").Range("A1")
   With myCell.Font
      .Bold = True
      .Italic = True
   End With
End Sub
```

Statements that set property values are also assignment statements. The following example sets the **Bold** property of the **Font** object for the active cell:

```
ActiveCell.Font.Bold = True
```

Writing Declaration Statements

{ewc HLP95EN.DLL,DYNALINK,"See Also":"vaconWritingDeclarationsC"} {ewc HLP95EN.DLL,DYNALINK,"Specifics":"vaconWritingdeclarationsS"}

You use declaration statements to name and define procedures, variables, arrays, and constants. When you declare a procedure, variable, or constant, you also define its scope, depending on where you place the declaration and what keywords you use to declare it.

The following example contains three declarations.

```
Sub ApplyFormat()
   Const limit As Integer = 33
   Dim myCell As Range
   ' More statements
End Sub
```

The **Sub** statement (with matching **End Sub** statement) declares a procedure named `ApplyFormat`. All the statements enclosed by the **Sub** and **End Sub** statements are executed whenever the `ApplyFormat` procedure is called or run.

- Writing a Sub Procedure

The **Const** statement declares the constant `limit`, specifying the **Integer** data type and a value of 33.

- Declaring Constants

The **Dim** statement declares the variable `myCell`. The data type is an object, in this case, a Microsoft Excel **Range** object. You can declare a variable to be any object that is exposed in the application you are using. **Dim** statements are one type of statement used to declare variables. Other keywords used in declarations are **ReDim**, **Static**, **Public**, **Private**, and **Const**.

- Declaring Variables

Writing Executable Statements

{ewc HLP95EN.DLL,DYNALINK,"See Also":"vaconWritingexecutablestatementsC"} {ewc HLP95EN.DLL,DYNALINK,"Specifics":"vaconWritingExecutableStatementsS"}

An executable statement initiates action. It can execute a method or function, and it can loop or branch through blocks of code. Executable statements often contain mathematical or conditional operators.

The following example uses a **For Each...Next** statement to iterate through each cell in a range named `MyRange` on `Sheet1` of an active Microsoft Excel workbook. The variable `c` is a cell in the collection of cells contained in `MyRange`.

```
Sub ApplyFormat()
Const limit As Integer = 33
For Each c In Worksheets("Sheet1").Range("MyRange").Cells
   If c.Value > limit Then
      With c.Font
         .Bold = True
         .Italic = True
      End With
   End If
Next c
MsgBox "All done!"
End Sub
```

The **If...Then...Else** statement in the example checks the value of the cell. If the value is greater than 33, the **With** statement sets the **Bold** and **Italic** properties of the **Font** object for that cell. **If...Then...Else** statements end with **End If**.

The **With** statement can save typing because the statements it contains are automatically executed on the object following the **With** keyword.

The **Next** statement calls the next cell in the collection of cells contained in `MyRange`.

The **MsgBox** function (which displays a built-in Visual Basic dialog box) displays a message indicating that the **Sub** procedure has finished running.

Writing Data to Files

{ewc HLP95EN.DLL,DYNALINK,"See Also":"vaconWritingDataC"} {ewc HLP95EN.DLL,DYNALINK,"Specifics":"vaconWritingDataS"}

When working with large amounts of data, it is often convenient to write data to or read data from a file. The **Open** statement lets you create and access files directly. **Open** provides three types of file access:

- Sequential access (**Input**, **Output**, and **Append** modes) is used for writing text files, such as error logs and reports.
- Random access (**Random** mode) is used to read and write data to a file without closing it. Random access files keep data in records, which makes it easy to locate information quickly.
- Binary access (**Binary** mode) is used to read or write to any byte position in a file, such as storing or displaying a bitmap image.

  Note The **Open** statement should not be used to open an application's own file types. For example, don't use **Open** to open a Word document, a Microsoft Excel spreadsheet, or a Microsoft Access database. Doing so will cause loss of file integrity and file corruption.

The following table shows the statements typically used when writing data to and reading data from files.

Access Type	Writing Data	Reading Data
Sequential	**Print #**, **Write #**	**Input #**
Random	**Put**	**Get**
Binary	**Put**	**Get**

Writing Visual Basic Statements

{ewc HLP95EN.DLL,DYNALINK,"See Also":"vaconWritingVBStatementsC"} {ewc HLP95EN.DLL,DYNALINK,"Specifics":"vaconWritingVBStatementsS"}

A statement in Visual Basic is a complete instruction. It can contain keywords, operators, variables, constants, and expressions. Each statement belongs to one of the following three categories:

- Declaration statements, which name a variable, constant, or procedure and can also specify a data type.

  Writing Declaration Statements

- Assignment statements, which assign a value or expression to a variable or constant.

  Writing Assignment Statements

- Executable statements, which initiate actions. These statements can execute a method or function, and they can loop or branch through blocks of code. Executable statements often contain mathematical or conditional operators.

  Writing Executable Statements

Continuing a Statement over Multiple Lines

A statement usually fits on one line, but you can continue a statement onto the next line using a line-continuation character. In the following example, the **MsgBox** executable statement is continued over three lines:

```
Sub DemoBox()  'This procedure declares a string variable,
      ' assigns it the value Claudia, and then displays
      ' a concatenated message.
   Dim myVar As String
   myVar = "John"
   MsgBox Prompt:="Hello " & myVar, _
      Title:="Greeting Box", _
      Buttons:=vbExclamation
End Sub
```

Adding Comments

Comments can explain a procedure or a particular instruction to anyone reading your code. Visual Basic ignores comments when it runs your procedures. Comment lines begin with an apostrophe (') or with **Rem** followed by a space, and can be added anywhere in a procedure. To add a comment to the same line as a statement, insert an apostrophe after the statement, followed by the comment. By default, comments are displayed as green text.

Checking Syntax Errors

If you press ENTER after typing a line of code and the line is displayed in red (an error message may display as well), you must find out what's wrong with your statement, and then correct it.

Using the Add-In Manager

{ewc HLP95EN.DLL,DYNALINK,"See Also":"vaconUsingAddInManager_C"} {ewc HLP95EN.DLL,DYNALINK,"Specifics":"vaconUsingAddInManager_S"}

Use the **Add-In Manager** dialog box to load or unload an add-in. If you close only the visible portions of an add-in—by double-clicking its system menu or by clicking its close button, for example—its forms disappear from the screen, but the add-in is still present in memory. The add-in object itself will always stay resident in memory until the add-in is disconnected through the **Add-In Manager** dialog box.