# Understanding Visual Basic syntax

The syntax in a Visual Basic Help topic for a method, function, or statement shows all the elements necessary to use the method, function, or statement 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.

VBCopy

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

## MsgBox function syntax

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

In the **MsgBox** function syntax, the 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:

VBCopy

```
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:

VBCopy

```
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 are 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.

VBCopy

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

# Option Compare 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 order that is not case-sensitive.

VBCopy

```
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.

VBCopy
```
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.

VBCopy
```
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**.

VBCopy
```
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.

VBCopy
```
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`.

VBCopy
```
Dim myArray()
```

# Using arrays

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 with 365 elements. By default, an array is indexed beginning with zero, so the upper bound of the array is 364 rather than 365.

VBCopy

```
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.

VBCopy

```
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 with 365 elements.

VBCopy

```
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.

VBCopy

```
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:

VBCopy

```
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.

VBCopy

```
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.

VBCopy

```
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.

VBCopy

```
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.

VBCopy

```
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

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 by 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.

## Types of constants

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, Excel, Project, and Word also provide a list of constants that 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 by using the **Const** statement.
- Conditional compiler constants are declared by using the **#Const** statement (directive).

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

VBCopy
```
TILE_HORIZONTAL
```

Intrinsic constants are now qualified to avoid 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:

VBCopy

```
Threed.LeftJustify
```

# Using data types efficiently

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:

VBCopy

```
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:

VBCopy

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

# Using Do...Loop statements

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**.

VBCopy

```
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**.

VBCopy

```
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** by 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.

VBCopy

```
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 Each...Next statements

**For Each...Next** statements repeat a block of statements for each object in a collectionor 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.

VBCopy

```
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.

VBCopy

```
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).

VBCopy

```
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 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 does not occur, 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.

VBCopy

```
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 a For Each...Next loop to iterate over a VBA class

**For Each...Next** loops don't only iterate over arrays and instances of the **Collection**object. **For Each...Next** loops can also iterate over a VBA class that you have written.

Following is an example demonstrating how you can do this.

1. Create a class module in the VBE (Visual Basic Editor), and rename it **CustomCollection**.[cc1]
2. Place the following code in the newly created module.

   VBCopy
   ```
   Private MyCollection As New Collection

   ' The Initialize event automatically gets triggered
   ' when instances of this class are created.
   ' It then triggers the execution of this procedure.
   Private Sub Class_Initialize()
       With MyCollection
           .Add "First Item"
           .Add "Second Item"
           .Add "Third Item"
       End With
   End Sub

   ' Property Get procedure for the setting up of
   ' this class so that it works with 'For Each...'
   ' constructs.
   Property Get NewEnum() As IUnknown
   ' Attribute NewEnum.VB_UserMemId = -4

   Set NewEnum = MyCollection.[_NewEnum]
   End Property
   ```

3. Export this module to a file and store it locally.[cc2]
4. After you export the module, open the exported file by using a text editor (Window's *Notepad* software should be sufficient). The file contents should look like the following.

   VBCopy
   ```
   VERSION 1.0 CLASS
   BEGIN
   MultiUse = -1  'True
   END
   Attribute VB_Name = "CustomCollection"
   Attribute VB_GlobalNameSpace = False
   Attribute VB_Creatable = False
   Attribute VB_PredeclaredId = False
   Attribute VB_Exposed = False
   Private MyCollection As New Collection
   ```

```
' The Initialize event automatically gets triggered
' when instances of this class are created.
' It then triggers the execution of this procedure.
Private Sub Class_Initialize()
    With MyCollection
        .Add "First Item"
        .Add "Second Item"
        .Add "Third Item"
    End With
End Sub

' Property Get procedure for the setting up of
' this class so that it works with 'For Each...'
' constructs.
Property Get NewEnum() As IUnknown
' Attribute NewEnum.VB_UserMemId = -4

Set NewEnum = MyCollection.[_NewEnum]
End Property
```

5. Using the text editor, remove the `'` character from the first line under the `Property Get NewEnum() As IUnknown` text in the file. Save the modified file.
6. Back in the VBE, remove the class that you created from your VBA project and don't choose to export it when prompted.[cc3]
7. Import the file that you removed the `'` character from back into the VBE.[cc4]
8. Run the following code to see that you can now iterate over your custom VBA class that you have written by using both the VBE and a text editor.

VBCopy

```
Dim Element
Dim MyCustomCollection As New CustomCollection
For Each Element In MyCustomCollection
MsgBox Element
Next
```

| Footnotes | Description |
|---|---|
| [cc1] | You can create a class module by choosing **Class Module** on the **Insert** menu. You can rename a class module by modifying its properties in the Propertieswindow. |
| [cc2] | You can activate the **Export File** dialog box by choosing **Export File** on the **File** menu. |

| Footnotes | Description |
| --- | --- |
| [cc3] | You can remove a class module from the VBE by choosing **Remove Item** on the **File** menu. |
| [cc4] | You can import an external class-module file by activating the **Import File**dialog box (choose **Import File** on the **File** menu). |

# Using For...Next statements

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 and its start and end values. The **Next** statement increments the counter variable by 1.

VBCopy

```
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.

VBCopy

```
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.

VBCopy

```
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, the **If...Then...Else** statement is **False**, and the loop will continue to run as expected.

# Using If...Then...Else statements

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.

VBCopy

```
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.

VBCopy

```
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**, and the other block runs if the condition is **False**.

VBCopy

```
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**.

VBCopy

```
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

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

VBCopy

```
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:

VBCopy

```
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.

VBCopy

```
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 by 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 by using named arguments, but it ignores the return value.

VBCopy

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

The following example calls the **MsgBox** function by using named arguments and assigns the return value to the variable.

VBCopy

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

# Using Select Case statements

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 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.

VBCopy

```
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 the Add-In Manager

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.

# Using With statements

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.

VBCopy

```
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.

VBCopy

```
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
```

# VarType constants

The following constants can be used anywhere in your code in place of the actual values.

| Constant | Value | Description |
|---|---|---|
| **vbEmpty** | 0 | Uninitialized (default) |
| **vbNull** | 1 | Contains no valid data |
| **vbInteger** | 2 | Integer |
| **vbLong** | 3 | Long integer |
| **vbSingle** | 4 | Single-precision floating-point number |
| **vbDouble** | 5 | Double-precision floating-point number |
| **vbCurrency** | 6 | Currency |
| **vbDate** | 7 | Date |
| **vbString** | 8 | String |
| **vbObject** | 9 | Object |
| **vbError** | 10 | Error |
| **vbBoolean** | 11 | Boolean |
| **vbVariant** | 12 | Variant (used only for arrays of variants) |
| **vbDataObject** | 13 | Data access object |

| Constant | Value | Description |
|---|---|---|
| **vbDecimal** | 14 | Decimal |
| **vbByte** | 17 | Byte |
| **vbLongLong** | 20 | LongLong integer (valid on 64-bit platforms only) |
| **vbUserDefinedType** | 36 | Variants that contain user-defined types |
| **vbArray** | 8192 | Array |

# Visual Basic naming rules

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 function, statement, method, and intrinsic constant names used in Visual Basic or by the host application. Otherwise 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 by 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

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

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

VBCopy

```
' 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 about 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:

VBCopy

```
Dim X As New Field
```

This case is an example of a class in the data access type library. A new instance of a **Field** object is created by 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

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 expressionsthat 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.

VBCopy

```
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

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 modulecontaining 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 as follows.

[ **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, because 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 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.

VBCopy

```
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

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.

VBCopy

```
' 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

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.

VBCopy

```
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.

VBCopy

```
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`.

VBCopy

```
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.

VBCopy

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

# Writing data to files

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 declaration statements

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.

VBCopy

```
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.

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

The **Dim** statement declares the `myCell` variable. 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 that 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**.

# Writing executable statements

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*.

VBCopy

```
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 Visual Basic statements

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

## Continue a statement over multiple lines

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

VBCopy

```
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
```

## Add 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.

## Check 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.