Visual Basic for Applications Reference

Visual Studio 6.0

AppActivate Statement

See Also Example Specifics

Activates an application window.

Syntax

AppActivate ***title***[, ***wait***]

The **AppActivate** statement syntax has these named arguments:

Part	Description
title	Required. String expression specifying the title in the title bar of the application window you want to activate. The task ID returned by the **Shell** function can be used in place of ***title*** to activate an application.
wait	Optional. Boolean value specifying whether the calling application has the focus before activating another. If **False** (default), the specified application is immediately activated, even if the calling application does not have the focus. If **True**, the calling application waits until it has the focus, then activates the specified application.

Remarks

The **AppActivate** statement changes the focus to the named application or window but does not affect whether it is maximized or minimized. Focus moves from the activated application window when the user takes some action to change the focus or close the window. Use the **Shell** function to start an application and set the window style.

In determining which application to activate, ***title*** is compared to the title string of each running application. If there is no exact match, any application whose title string begins with ***title*** is activated. If there is more than one instance of the application named by ***title***, one instance is arbitrarily activated.

Visual Basic for Applications Reference

AppActivate Statement Example

This example illustrates various uses of the **AppActivate** statement to activate an application window. The **Shell** statements assume the applications are in the paths specified.

```
Dim MyAppID, ReturnValue
AppActivate "Microsoft Word"   ' Activate Microsoft
   ' Word.

' AppActivate can also use the return value of the Shell function.
MyAppID = Shell("C:\WORD\WINWORD.EXE", 1)   ' Run Microsoft Word.
AppActivate MyAppID   ' Activate Microsoft
   ' Word.



' You can also use the return value of the Shell function.
ReturnValue = Shell("c:\EXCEL\EXCEL.EXE",1)   ' Run Microsoft Excel.
AppActivate ReturnValue   ' Activate Microsoft
   ' Excel.
```

Visual Basic for Applications Reference

Visual Studio 6.0

Beep Statement

See Also Example Specifics

Sounds a tone through the computer's speaker.

Syntax

Beep

Remarks

The frequency and duration of the beep depend on your hardware and system software, and vary among computers.

Visual Basic for Applications Reference

Beep Statement Example

This example uses the **Beep** statement to sound three consecutive tones through the computer's speaker.

```
Dim I
For I = 1 To 3   ' Loop 3 times.
   Beep   ' Sound a tone.
Next I
```

Visual Basic for Applications Reference

Visual Studio 6.0

Call Statement

See Also Example Specifics

Transfers control to a **Sub** procedure, **Function** procedure, or dynamic-link library (DLL) procedure.

Syntax

[**Call**] *name* [*argumentlist*]

The **Call** statement syntax has these parts:

Part	Description
Call	Optional; keyword. If specified, you must enclose *argumentlist* in parentheses. For example:
	`Call MyProc(0)`
name	Required. Name of the procedure to call.
argumentlist	Optional. Comma-delimited list of variables, arrays, or expressions to pass to the procedure. Components of *argumentlist* may include the keywords **ByVal** or **ByRef** to describe how the arguments are treated by the called procedure. However, **ByVal** and **ByRef** can be used with **Call** only when calling a DLL procedure.

Remarks

You are not required to use the **Call** keyword when calling a procedure. However, if you use the **Call** keyword to call a procedure that requires arguments, *argumentlist* must be enclosed in parentheses. If you omit the **Call** keyword, you also must omit the parentheses around *argumentlist*. If you use either **Call** syntax to call any intrinsic or user-defined function, the function's return value is discarded.

To pass a whole array to a procedure, use the array name followed by empty parentheses.

Visual Basic for Applications Reference

Call Statement Example

This example illustrates how the **Call** statement is used to transfer control to a **Sub** procedure, an intrinsic function, and a dynamic-link library (DLL) procedure.

```
' Call a Sub procedure.
Call PrintToDebugWindow("Hello World")
' The above statement causes control to be passed to the following
' Sub procedure.
Sub PrintToDebugWindow(AnyString)
   Debug.Print AnyString   ' Print to the Immediate window.
End Sub

' Call an intrinsic function. The return value of the function is
' discarded.
Call Shell(AppName, 1)   ' AppName contains the path of the
      ' executable file.

' Call a Microsoft Windows DLL procedure. The Declare statement must be
' Private in a Class Module, but not in a standard Module.
Private Declare Sub MessageBeep Lib "User" (ByVal N As Integer)
Sub CallMyDll()
   Call MessageBeep(0)   ' Call Windows DLL procedure.
   MessageBeep 0   ' Call again without Call keyword.
End Sub
```

Visual Basic for Applications Reference

Visual Studio 6.0

ChDir Statement

See Also Example Specifics

Changes the current directory or folder.

Syntax

ChDir *path*

The required *path* argument is a string expression that identifies which directory or folder becomes the new default directory or folder. The *path* may include the drive. If no drive is specified, **ChDir** changes the default directory or folder on the current drive.

Remarks

The **ChDir** statement changes the default directory but not the default drive. For example, if the default drive is C, the following statement changes the default directory on drive D, but C remains the default drive:

```
ChDir "D:\TMP"
```

Visual Basic for Applications Reference

ChDir Statement Example

This example uses the **ChDir** statement to change the current directory or folder.

```
' Change current directory or folder to "MYDIR".
ChDir "MYDIR"

' Assume "C:" is the current drive. The following statement changes
' the default directory on drive "D:". "C:" remains the current drive.
ChDir "D:\WINDOWS\SYSTEM"
```

Visual Basic for Applications Reference

Visual Studio 6.0

ChDrive Statement

See Also Example Specifics

Changes the current drive.

Syntax

ChDrive *drive*

The required *drive* argument is a string expression that specifies an existing drive. If you supply a zero-length string (""), the current drive doesn't change. If the *drive* argument is a multiple-character string, **ChDrive** uses only the first letter.

Visual Basic for Applications Reference

ChDrive Statement Example

This example uses the **ChDrive** statement to change the current drive.

```
ChDrive "D"   ' Make "D" the current drive.
```

Visual Basic for Applications Reference

Visual Studio 6.0

Close Statement

See Also Example Specifics

Concludes input/output (I/O) to a file opened using the **Open** statement.

Syntax

Close [*filenumberlist*]

The optional *filenumberlist* argument can be one or more file numbers using the following syntax, where *filenumber* is any valid file number:

[[**#**]*filenumber*] [**,** [**#**]*filenumber*] **. . .**

Remarks

If you omit *filenumberlist*, all active files opened by the **Open** statement are closed.

When you close files that were opened for **Output** or **Append**, the final buffer of output is written to the operating system buffer for that file. All buffer space associated with the closed file is released.

When the **Close** statement is executed, the association of a file with its file number ends.

Visual Basic for Applications Reference

Close Statement Example

This example uses the **Close** statement to close all three files opened for **Output**.

```
Dim I, FileName
For I = 1 To 3   ' Loop 3 times.
   FileName = "TEST" & I   ' Create file name.
   Open FileName For Output As #I   ' Open file.
   Print #I, "This is a test."   ' Write string to file.
Next I
Close   ' Close all 3 open files.
```

Visual Basic for Applications Reference

Visual Studio 6.0

Const Statement

See Also Example Specifics

Declares constants for use in place of literal values.

Syntax

[**Public** | **Private**] **Const** *constname* [**As** *type*] = *expression*

The **Const** statement syntax has these parts:

Part	Description
Public	Optional. Keyword used at module level to declare constants that are available to all procedures in all modules. Not allowed in procedures.
Private	Optional. Keyword used at module level to declare constants that are available only within the module where the declaration is made. Not allowed in procedures.
constname	Required. Name of the constant; follows standard variable naming conventions.
type	Optional. Data type of the constant; may be Byte, Boolean, Integer, Long, Currency, Single, Double, Decimal (not currently supported), Date, String, or Variant. Use a separate **As** *type* clause for each constant being declared.
expression	Required. Literal, other constant, or any combination that includes all arithmetic or logical operators except **Is**.

Remarks

Constants are private by default. Within procedures, constants are always private; their visibility can't be changed. In standard modules, the default visibility of module-level constants can be changed using the **Public** keyword. In class modules, however, constants can only be private and their visibility can't be changed using the **Public** keyword.

To combine several constant declarations on the same line, separate each constant assignment with a comma. When constant declarations are combined in this way, the **Public** or **Private** keyword, if used, applies to all of them.

You can't use variables, user-defined functions, or intrinsic Visual Basic functions (such as **Chr**) in expressions assigned to constants.

Note Constants can make your programs self-documenting and easy to modify. Unlike variables, constants can't be inadvertently changed while your program is running.

If you don't explicitly declare the constant type using **As** *type*, the constant has the data type that is most appropriate for *expression*.

Constants declared in a **Sub**, **Function**, or **Property** procedure are local to that procedure. A constant declared outside a procedure is defined throughout the module in which it is declared. You can use constants anywhere you can use an expression.

Visual Basic for Applications Reference

Const Statement Example

This example uses the **Const** statement to declare constants for use in place of literal values. **Public** constants are declared in the General section of a standard module, rather than a class module. **Private** constants are declared in the General section of any type of module.

```
' Constants are Private by default.
Const MyVar = 459

' Declare Public constant.
Public Const MyString = "HELP"

' Declare Private Integer constant.
Private Const MyInt As Integer = 5

' Declare multiple constants on same line.
Const MyStr = "Hello", MyDouble As Double = 3.4567
```

Visual Basic for Applications Reference

Visual Studio 6.0

Date Statement

See Also Example Specifics

Sets the current system date.

Syntax

Date = *date*

For systems running Microsoft Windows 95, the required *date* specification must be a date from January 1, 1980 through December 31, 2099. For systems running Microsoft Windows NT, *date* must be a date from January 1, 1980 through December 31, 2079.

Visual Basic for Applications Reference

Date Statement Example

This example uses the **Date** statement to set the computer system date. In the development environment, the date literal is displayed in short date format using the locale settings of your code.

```
Dim MyDate
MyDate = #February 12, 1985#   ' Assign a date.
Date = MyDate   ' Change system date.
```

Visual Basic for Applications Reference

Visual Studio 6.0

Declare Statement

See Also Example Specifics

Used at module level to declare references to external procedures in a dynamic-link library (DLL).

Syntax 1

[**Public** | **Private**] **Declare Sub** *name* **Lib** "*libname*" [**Alias** "*aliasname*"] [**(**[*arglist*]**)**]

Syntax 2

[**Public** | **Private**] **Declare Function** *name* **Lib** "*libname*" [**Alias** "*aliasname*"] [**(**[*arglist*]**)**] [**As** *type*]

The **Declare** statement syntax has these parts:

Part	Description
Public	Optional. Used to declare procedures that are available to all other procedures in all modules.
Private	Optional. Used to declare procedures that are available only within the module where the declaration is made.
Sub	Optional (either **Sub** or **Function** must appear). Indicates that the procedure doesn't return a value.
Function	Optional (either **Sub** or **Function** must appear). Indicates that the procedure returns a value that can be used in an expression.
name	Required. Any valid procedure name. Note that DLL entry points are case sensitive.
Lib	Required. Indicates that a DLL or code resource contains the procedure being declared. The **Lib** clause is required for all declarations.
libname	Required. Name of the DLL or code resource that contains the declared procedure.
Alias	Optional. Indicates that the procedure being called has another name in the DLL. This is useful when the external procedure name is the same as a keyword. You can also use **Alias** when a DLL procedure has the same name as a public variable, constant, or any other procedure in the same scope. **Alias** is also useful if any characters in the DLL procedure name aren't allowed by the DLL naming convention.
aliasname	Optional. Name of the procedure in the DLL or code resource. If the first character is not a number sign (**#**), *aliasname* is the name of the procedure's entry point in the DLL. If (**#**) is the first character, all characters that follow must indicate the ordinal number of the procedure's entry point.
arglist	Optional. List of variables representing arguments that are passed to the procedure when it is called.

type	Optional. Data type of the value returned by a **Function** procedure; may be Byte, Boolean, Integer, Long, Currency, Single, Double, Decimal (not currently supported), Date, String (variable length only), or Variant, a user-defined type, or an object type.

The *arglist* argument has the following syntax and parts:

[**Optional**] [**ByVal** | **ByRef**] [**ParamArray**] *varname*[**( )**] [**As** *type*]

Part	Description
Optional	Optional. Indicates that an argument is not required. If used, all subsequent arguments in *arglist* must also be optional and declared using the **Optional** keyword. **Optional** can't be used for any argument if **ParamArray** is used.
ByVal	Optional. Indicates that the argument is passed by value.
ByRef	Indicates that the argument is passed by reference. **ByRef** is the default in Visual Basic.
ParamArray	Optional. Used only as the last argument in *arglist* to indicate that the final argument is an **Optional** array of **Variant** elements. The **ParamArray** keyword allows you to provide an arbitrary number of arguments. The **ParamArray** keyword can't be used with **ByVal**, **ByRef**, or **Optional**.
varname	Required. Name of the variable representing the argument being passed to the procedure; follows standard variable naming conventions.
( )	Required for array variables. Indicates that *varname* is an array.
type	Optional. Data type of the argument passed to the procedure; may be **Byte**, **Boolean**, **Integer**, **Long**, **Currency**, **Single**, **Double**, **Decimal** (not currently supported), **Date**, **String** (variable length only), **Object**, **Variant**, a user-defined type, or an object type.

Remarks

For **Function** procedures, the data type of the procedure determines the data type it returns. You can use an **As** clause following *arglist* to specify the return type of the function. Within *arglist*, you can use an **As** clause to specify the data type of any of the arguments passed to the procedure. In addition to specifying any of the standard data types, you can specify **As Any** in *arglist* to inhibit type checking and allow any data type to be passed to the procedure.

Empty parentheses indicate that the **Sub** or **Function** procedure has no arguments and that Visual Basic should ensure that none are passed. In the following example, `First` takes no arguments. If you use arguments in a call to `First`, an error occurs:

```
Declare Sub First Lib "MyLib" ()
```

If you include an argument list, the number and type of arguments are checked each time the procedure is called. In the following example, `First` takes one **Long** argument:

```
Declare Sub First Lib "MyLib" (X As Long)
```

Note You can't have fixed-length strings in the argument list of a **Declare** statement; only variable-length strings can be passed to procedures. Fixed-length strings can appear as procedure arguments, but they are converted to variable-length strings before being passed.

Note The **vbNullString** constant is used when calling external procedures, where the external procedure requires a string whose value is zero. This is not the same thing as a zero-length string ("").

Visual Basic for Applications Reference

Declare Statement Example

This example shows how the **Declare** statement is used at the module level of a standard module to declare a reference to an external procedure in a dynamic-link library (DLL). You can place the **Declare** statements in class modules if the **Declare** statements are **Private**.

```
' In Microsoft Windows (16-bit):
Declare Sub MessageBeep Lib "User" (ByVal N As Integer)
' Assume SomeBeep is an alias for the procedure name.
Declare Sub MessageBeep Lib "User" Alias "SomeBeep"(ByVal N As Integer)
' Use an ordinal in the Alias clause to call GetWinFlags.
Declare Function GetWinFlags Lib "Kernel" Alias "#132"() As Long

' In 32-bit Microsoft Windows systems, specify the library USER32.DLL,
' rather than USER.DLL. You can use conditional compilation to write
' code that can run on either Win32 or Win16.
#If Win32 Then
   Declare Sub MessageBeep Lib "User32" (ByVal N As Long)
#Else
   Declare Sub MessageBeep Lib "User" (ByVal N As Integer)
#End If
```

Visual Basic for Applications Reference

Visual Studio 6.0

Deftype Statements

See Also Example Specifics

Used at module level to set the default data type for variables, arguments passed to procedures, and the return type for **Function** and **Property Get** procedures whose names start with the specified characters.

Syntax

DefBool *letterrange*[, *letterrange*] . . .

DefByte *letterrange*[, *letterrange*] . . .

DefInt *letterrange*[, *letterrange*] . . .

DefLng *letterrange*[, *letterrange*] . . .

DefCur *letterrange*[, *letterrange*] . . .

DefSng *letterrange*[, *letterrange*] . . .

DefDbl *letterrange*[, *letterrange*] . . .

DefDec *letterrange*[, *letterrange*] . . .

DefDate *letterrange*[, *letterrange*] . . .

DefStr *letterrange*[, *letterrange*] . . .

DefObj *letterrange*[, *letterrange*] . . .

DefVar *letterrange*[, *letterrange*] . . .

The required *letterrange* argument has the following syntax:

letter1[-*letter2*]

The *letter1* and *letter2* arguments specify the name range for which you can set a default data type. Each argument represents the first letter of the variable, argument, **Function** procedure, or **Property Get** procedure name and can be any letter of the alphabet. The case of letters in *letterrange* isn't significant.

Remarks

The statement name determines the data type:

Statement	Data Type

DefBool	Boolean
DefByte	Byte
DefInt	Integer
DefLng	Long
DefCur	Currency
DefSng	Single
DefDbl	Double
DefDec	Decimal (not currently supported)
DefDate	Date
DefStr	String
DefObj	Object
DefVar	Variant

For example, in the following program fragment, `Message` is a string variable:

```
DefStr A-Q
. . .
Message = "Out of stack space."
```

A **Def***type* statement affects only the module where it is used. For example, a **DefInt** statement in one module affects only the default data type of variables, arguments passed to procedures, and the return type for **Function** and **Property Get** procedures declared in that module; the default data type of variables, arguments, and return types in other modules is unaffected. If not explicitly declared with a **Def***type* statement, the default data type for all variables, all arguments, all **Function** procedures, and all **Property Get** procedures is **Variant**.

When you specify a letter range, it usually defines the data type for variables that begin with letters in the first 128 characters of the character set. However, when you specify the letter range AZ, you set the default to the specified data type for all variables, including variables that begin with international characters from the extended part of the character set (128255).

Once the range AZ has been specified, you can't further redefine any subranges of variables using **Def***type* statements. Once a range has been specified, if you include a previously defined letter in another **Def***type* statement, an error occurs. However, you can explicitly specify the data type of any variable, defined or not, using a **Dim** statement with an **As** *type* clause. For example, you can use the following code at module level to define a variable as a **Double** even though the default data type is **Integer**:

```
DefInt A-Z
Dim TaxRate As Double
```

Def*type* statements don't affect elements of user-defined types because the elements must be explicitly declared.

Visual Basic for Applications Reference

Deftype Statements Example

This example shows various uses of the **Def***type* statements to set default data types of variables and function procedures whose names start with specified characters. The default data type can be overridden only by explicit assignment using the **Dim** statement. **Def***type* statements can only be used at the module level (that is, not within procedures).

```
' Variable names beginning with A through K default to Integer.
DefInt A-K
' Variable names beginning with L through Z default to String.
DefStr L-Z
CalcVar = 4   ' Initialize Integer.
StringVar = "Hello there"   ' Initialize String.
AnyVar = "Hello"    ' Causes "Type mismatch" error.
Dim Calc As Double   ' Explicitly set the type to Double.
Calc = 2.3455   ' Assign a Double.

' Deftype statements also apply to function procedures.
CalcNum = ATestFunction(4)   ' Call user-defined function.
' ATestFunction function procedure definition.
Function ATestFunction(INumber)
   ATestFunction = INumber * 2   ' Return value is an integer.
End Function
```

Visual Basic for Applications Reference

Visual Studio 6.0

DeleteSetting Statement

See Also Example Specifics

Deletes a section or key setting from an application's entry in the Windows registry.

Syntax

DeleteSetting *appname*, *section*[, *key*]

The **DeleteSetting** statement syntax has these named arguments:

Part	Description
appname	Required. String expression containing the name of the application or project to which the section or key setting applies.
section	Required. String expression containing the name of the section where the key setting is being deleted. If only ***appname*** and ***section*** are provided, the specified section is deleted along with all related key settings.
key	Optional. String expression containing the name of the key setting being deleted.

Remarks

If all arguments are provided, the specified setting is deleted. A run-time error occurs if you attempt to use the **DeleteSetting** statement on a non-existent section or key setting.

Visual Basic for Applications Reference

DeleteSetting Statement Example

The following example first uses the **SaveSetting** statement to make entries in the Windows registry (or .ini file on 16-bit Windows platforms) for the MyApp application, and then uses the **DeleteSetting** statement to remove them. Because no ***key*** argument is specified, the whole section is deleted, including the section name and all its keys.

```
' Place some settings in the registry.
SaveSetting appname := "MyApp", section := "Startup", _
            key := "Top", setting := 75
SaveSetting "MyApp","Startup", "Left", 50
' Remove section and all its settings from registry.
DeleteSetting "MyApp", "Startup"
```

Visual Basic for Applications Reference

Visual Studio 6.0

Dim Statement

See Also Example Specifics

Declares variables and allocates storage space.

Syntax

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

The **Dim** statement syntax has these parts:

Part	Description
WithEvents	Optional. Keyword that specifies that *varname* is an object variable used to respond to events triggered by an ActiveX object. **WithEvents** is valid only in class modules. You can declare as many individual variables as you like using **WithEvents**, but you can't create arrays with **WithEvents**. You can't use **New** with **WithEvents**.
varname	Required. Name of the variable; follows standard variable naming conventions.
subscripts	Optional. Dimensions of an array variable; up to 60 multiple dimensions may be declared. The *subscripts* argument uses the following syntax: [*lower* **To**] *upper* [**,** [*lower* **To**] *upper*] . . . When not explicitly stated in *lower*, the lower bound of an array is controlled by the **Option Base** statement. The lower bound is zero if no **Option Base** statement is present.
New	Optional. Keyword that enables implicit creation of an object. If you use **New** when declaring the object variable, a new instance of the object is created on first reference to it, so you don't have to use the **Set** statement to assign the object reference. The **New** keyword can't be used to declare variables of any intrinsic data type, can't be used to declare instances of dependent objects, and cant be used with **WithEvents**.
type	Optional. Data type of the variable; may be Byte, Boolean, Integer, Long, Currency, Single, Double, Decimal (not currently supported), Date, String (for variable-length strings), **String** * *length* (for fixed-length strings), Object, Variant, a user-defined type, or an object type. Use a separate **As** *type* clause for each variable you declare.

Remarks

Variables declared with **Dim** at the module level are available to all procedures within the module. At the procedure level, variables are available only within the procedure.

Use the **Dim** statement at module or procedure level to declare the data type of a variable. For example, the following statement declares a variable as an **Integer**.

```
Dim NumberOfEmployees As Integer
```

Also use a **Dim** statement to declare the object type of a variable. The following declares a variable for a new instance of a worksheet.

```
Dim X As New Worksheet
```

If the **New** keyword is not used when declaring an object variable, the variable that refers to the object must be assigned an existing object using the **Set** statement before it can be used. Until it is assigned an object, the declared object variable has the special value **Nothing**, which indicates that it doesn't refer to any particular instance of an object.

You can also use the **Dim** statement with empty parentheses to declare a dynamic array. After declaring a dynamic array, use the **ReDim** statement within a procedure to define the number of dimensions and elements in the array. If you try to redeclare a dimension for an array variable whose size was explicitly specified in a **Private**, **Public**, or **Dim** statement, an error occurs.

If you don't specify a data type or object type, and there is no **Def***type* statement in the module, the variable is **Variant** by default.

When variables are initialized, a numeric variable is initialized to 0, a variable-length string is initialized to a zero-length string (""), and a fixed-length string is filled with zeros. **Variant** variables are initialized to Empty. Each element of a user-defined type variable is initialized as if it were a separate variable.

Note When you use the **Dim** statement in a procedure, you generally put the **Dim** statement at the beginning of the procedure.

Visual Basic for Applications Reference

Dim Statement Example

This example shows the **Dim** statement used to declare variables. It also shows the **Dim** statement used to declare arrays. The default lower bound for array subscripts is 0 and can be overridden at the module level using the **Option Base** statement.

```
' AnyValue and MyValue are declared as Variant by default with values
' set to Empty.
Dim AnyValue, MyValue

' Explicitly declare a variable of type Integer.
Dim Number As Integer

' Multiple declarations on a single line. AnotherVar is of type Variant
' because its type is omitted.
Dim AnotherVar, Choice As Boolean, BirthDate As Date

' DayArray is an array of Variants with 51 elements indexed, from
' 0 thru 50, assuming Option Base is set to 0 (default) for
' the current module.
Dim DayArray(50)

' Matrix is a two-dimensional array of integers.
Dim Matrix(3, 4) As Integer

' MyMatrix is a three-dimensional array of doubles with explicit
' bounds.
Dim MyMatrix(1 To 5,  4 To 9,  3 To 5) As Double

' BirthDay is an array of dates with indexes from 1 to 10.
Dim BirthDay(1 To 10) As Date

' MyArray is a dynamic array of variants.
Dim MyArray()
```

Visual Basic for Applications Reference

Visual Studio 6.0

Do...Loop Statement

See Also Example Specifics

Repeats a block of statements while a condition is **True** or until a condition becomes **True**.

Syntax

Do [{**While** | **Until**} *condition*]
[*statements*]
[**Exit Do**]
[*statements*]

Loop

Or, you can use this syntax:

Do
[*statements*]
[**Exit Do**]
[*statements*]

Loop [{**While** | **Until**} *condition*]

The **Do Loop** statement syntax has these parts:

Part	Description
condition	Optional. Numeric expression or string expression that is **True** or **False**. If *condition* is Null, *condition* is treated as **False**.
statements	One or more statements that are repeated while, or until, *condition* is **True**.

Remarks

Any number of **Exit Do** statements may be placed anywhere in the **DoLoop** as an alternate way to exit a **DoLoop**. **Exit Do** is often used after evaluating some condition, for example, **IfThen**, in which case the **Exit Do** statement transfers control to the statement immediately following the **Loop**.

When used within nested **DoLoop** statements, **Exit Do** transfers control to the loop that is one nested level above the loop where **Exit Do** occurs.

Do...Loop Statement Example

This content is no longer actively maintained. It is provided as is, for anyone who may still be using these technologies, with no warranties or claims of accuracy with regard to the most recent product version or service release.

This example sorts the data in the first column on Sheet1 and then deletes any rows that contain duplicate data.

```
Worksheets("Sheet1").Range("A1").Sort _
    key1:=Worksheets("Sheet1").Range("A1")
Set currentCell = Worksheets("Sheet1").Range("A1")
Do While Not IsEmpty(currentCell)
    Set nextCell = currentCell.Offset(1, 0)
    If nextCell.Value = currentCell.Value Then
        currentCell.EntireRow.Delete
    End If
    Set currentCell = nextCell
Loop
```

Visual Basic for Applications Reference

Visual Studio 6.0

End Statement

See Also Example Specifics

Ends a procedure or block.

Syntax

End

End Function

End If

End Property

End Select

End Sub

End Type

End With

The **End** statement syntax has these forms:

Statement	Description
End	Terminates execution immediately. Never required by itself but may be placed anywhere in a procedure to end code execution, close files opened with the **Open** statement and to clear variables.
End Function	Required to end a **Function** statement.
End If	Required to end a block **IfThenElse** statement.
End Property	Required to end a **Property Let**, **Property Get**, or **Property Set** procedure.
End Select	Required to end a **Select Case** statement.
End Sub	Required to end a **Sub** statement.
End Type	Required to end a user-defined type definition (**Type** statement).

End With	Required to end a **With** statement.

Remarks

When executed, the **End** statement resets all module-level variables and all static local variables in all modules. To preserve the value of these variables, use the **Stop** statement instead. You can then resume execution while preserving the value of those variables.

Note The **End** statement stops code execution abruptly, without invoking the Unload, QueryUnload, or Terminate event, or any other Visual Basic code. Code you have placed in the Unload, QueryUnload, and Terminate events of forms and class modules is not executed. Objects created from class modules are destroyed, files opened using the **Open** statement are closed, and memory used by your program is freed. Object references held by other programs are invalidated.

The **End** statement provides a way to force your program to halt. For normal termination of a Visual Basic program, you should unload all forms. Your program closes as soon as there are no other programs holding references to objects created from your public class modules and no code executing.

Visual Basic for Applications Reference

End Statement Example

This example uses the **End** Statement to end code execution if the user enters an invalid password.

```
Sub Form_Load
   Dim Password, Pword
   PassWord = "Swordfish"
   Pword = InputBox("Type in your password")
   If Pword <> PassWord Then
      MsgBox "Sorry, incorrect password"
      End
   End If
End Sub
```

Visual Basic for Applications Reference

Visual Studio 6.0

Enum Statement

See Also Example Specifics

Declares a type for an enumeration.

Syntax

[**Public** | **Private**] **Enum** *name*

membername [= *constantexpression*]

membername [= *constantexpression*]

. . .

End Enum

The **Enum** statement has these parts:

Part	Description
Public	Optional. Specifies that the **Enum** type is visible throughout the project. **Enum** types are **Public** by default.
Private	Optional. Specifies that the **Enum** type is visible only within the module in which it appears.
name	Required. The name of the **Enum** type. The *name* must be a valid Visual Basic identifier and is specified as the type when declaring variables or parameters of the **Enum** type.
membername	Required. A valid Visual Basic identifier specifying the name by which a constituent element of the **Enum** type will be known.
constantexpression	Optional. Value of the element (evaluates to a **Long**). If no *constantexpression* is specified, the value assigned is either zero (if it is the first *membername*), or 1 greater than the value of the immediately preceding *membername*.

Remarks

Enumeration variables are variables declared with an **Enum** type. Both variables and parameters can be declared with an **Enum** type. The elements of the **Enum** type are initialized to constant values within the **Enum** statement. The assigned values can't be modified at run time and can include both positive and negative numbers. For example:

```
Enum SecurityLevel
   IllegalEntry = -1
   SecurityLevel1 = 0
   SecurityLevel2 = 1
End Enum
```

An **Enum** statement can appear only at module level. Once the **Enum** type is defined, it can be used to declare variables, parameters, or procedures returning its type. You can't qualify an **Enum** type name with a module name. **Public Enum** types in a class module are not members of the class; however, they are written to the type library. **Enum** types defined in standard modules arent written to type libraries. **Public Enum** types of the same name can't be defined in both standard modules and class modules, since they share the same name space. When two **Enum** types in different type libraries have the same name, but different elements, a reference to a variable of the type depends on which type library has higher priority in the **References**.

You can't use an **Enum** type as the target in a **With** block.

Visual Basic for Applications Reference

Enum Statement Example

The following example shows the **Enum** statement used to define a collection of named constants. In this case, the constants are colors you might choose to design data entry forms for a database.

```
Public Enum InterfaceColors
   icMistyRose = &HE1E4FF&
   icSlateGray = &H908070&
   icDodgerBlue = &HFF901E&
   icDeepSkyBlue = &HFFBF00&
   icSpringGreen = &H7FFF00&
   icForestGreen = &H228B22&
   icGoldenrod = &H20A5DA&
   icFirebrick = &H2222B2&
End Enum
```

Visual Basic for Applications Reference

Visual Studio 6.0

Erase Statement

See Also Example Specifics

Reinitializes the elements of fixed-size arrays and releases dynamic-array storage space.

Syntax

Erase *arraylist*

The required *arraylist* argument is one or more comma-delimited array variables to be erased.

Remarks

Erase behaves differently depending on whether an array is fixed-size (ordinary) or dynamic. **Erase** recovers no memory for fixed-size arrays. **Erase** sets the elements of a fixed array as follows:

Type of Array	Effect of Erase on Fixed-Array Elements
Fixed numeric array	Sets each element to zero.
Fixed string array (variable length)	Sets each element to a zero-length string ("").
Fixed string array (fixed length)	Sets each element to zero.
Fixed Variant array	Sets each element to Empty.
Array of user-defined types	Sets each element as if it were a separate variable.
Array of objects	Sets each element to the special value **Nothing**.

Erase frees the memory used by dynamic arrays. Before your program can refer to the dynamic array again, it must redeclare the array variable's dimensions using a **ReDim** statement.

Visual Basic for Applications Reference

Erase Statement Example

This example uses the **Erase** statement to reinitialize the elements of fixed-size arrays and deallocate dynamic-array storage space.

```
' Declare array variables.
Dim NumArray(10) As Integer   ' Integer array.
Dim StrVarArray(10) As String   ' Variable-string array.
Dim StrFixArray(10) As String * 10   ' Fixed-string array.
Dim VarArray(10) As Variant   ' Variant array.
Dim DynamicArray() As Integer   ' Dynamic array.
ReDim DynamicArray(10)   ' Allocate storage space.
Erase NumArray   ' Each element set to 0.
Erase StrVarArray   ' Each element set to zero-length
   ' string ("").
Erase StrFixArray   ' Each element set to 0.
Erase VarArray   ' Each element set to Empty.
Erase DynamicArray   ' Free memory used by array.
```

This documentation is archived and is not being maintained.

Visual Basic for Applications Reference

Visual Studio 6.0

Error Statement

See Also Example Specifics

Simulates the occurrence of an error.

Syntax

Error *errornumber*

The required *errornumber* can be any valid error number.

Remarks

The **Error** statement is supported for backward compatibility. In new code, especially when creating objects, use the **Err** object's **Raise** method to generate run-time errors.

If *errornumber* is defined, the **Error** statement calls the error handler after the properties of **Err** object are assigned the following default values:

Property	Value
Number	Value specified as argument to **Error** statement. Can be any valid error number.
Source	Name of the current Visual Basic project.
Description	String expression corresponding to the return value of the **Error** function for the specified **Number**, if this string exists. If the string doesn't exist, **Description** contains a zero-length string ("").
HelpFile	The fully qualified drive, path, and file name of the appropriate Visual Basic Help file.
HelpContext	The appropriate Visual Basic Help file context ID for the error corresponding to the **Number** property.
LastDLLError	Zero.

If no error handler exists or if none is enabled, an error message is created and displayed from the **Err** object properties.

Note Not all Visual Basic host applications can create objects. See your host application's documentation to determine whether it can create classes and objects.

Visual Basic for Applications Reference

Error Statement Example

This example uses the **Error** statement to simulate error number 11.

```
On Error Resume Next   ' Defer error handling.
Error 11   ' Simulate the "Division by zero" error.
```

Visual Basic for Applications Reference

Visual Studio 6.0

Event Statement

See Also Example Specifics

Declares a user-defined event.

Syntax

[**Public**] **Event** *procedurename* [**(***arglist***)**]

The **Event** statement has these parts:

Part	Description
Public	Optional. Specifies that the **Event** visible throughout the project. **Events** types are **Public** by default. Note that events can only be raised in the module in which they are declared.
procedurename	Required. Name of the event; follows standard variable naming conventions.

The *arglist* argument has the following syntax and parts:

[**ByVal** | **ByRef**] *varname*[**( )**] [**As** *type*]

Part	Description
ByVal	Optional. Indicates that the argument is passed by value.
ByRef	Optional. Indicates that the argument is passed by reference. **ByRef** is the default in Visual Basic.
varname	Required. Name of the variable representing the argument being passed to the procedure; follows standard variable naming conventions.
type	Optional. Data type of the argument passed to the procedure; may be Byte, Boolean, Integer, Long, Currency, Single, Double, Decimal (not currently supported), Date, String (variable length only), Object, Variant, a user-defined type, or an object type.

Remarks

Once the event has been declared, use the **RaiseEvent** statement to fire the event. A syntax error occurs if an **Event** declaration appears in a standard module. An event can't be declared to return a value. A typical event might be declared

and raised as shown in the following fragments:

```
' Declare an event at module level of a class module

Event LogonCompleted (UserName as String)

Sub
   RaiseEvent LogonCompleted("AntoineJan")
End Sub
```

Note You can declare event arguments just as you do arguments of procedures, with the following exceptions: events cannot have named arguments, **Optional** arguments, or **ParamArray** arguments. Events do not have return values.

Visual Basic for Applications Reference

Event Statement Example

The following example uses events to count off seconds during a demonstration of the fastest 100 meter race. The code illustrates all of the event-related methods, properties, and statements, including the **Event** statement.

The class that raises an event is the event source, and the classes that implement the event are the sinks. An event source can have multiple sinks for the events it generates. When the class raises the event, that event is fired on every class that has elected to sink events for that instance of the object.

The example also uses a form (`Form1`) with a button (`Command1`), a label (`Label1`), and two text boxes (`Text1` and `Text2`). When you click the button, the first text box displays "From Now" and the second starts to count seconds. When the full time (9.84 seconds) has elapsed, the first text box displays "Until Now" and the second displays "9.84"

The code for `Form1` specifies the initial and terminal states of the form. It also contains the code executed when events are raised.

```
Option Explicit

Private WithEvents mText As TimerState

Private Sub Command1_Click()
Text1.Text = "From Now"
    Text1.Refresh
    Text2.Text = "0"
    Text2.Refresh
Call mText.TimerTask(9.84)
End Sub

Private Sub Form_Load()
    Command1.Caption = "Click to Start Timer"
    Text1.Text = ""
    Text2.Text = ""
    Label1.Caption = "The fastest 100 meter run took this long:"
    Set mText = New TimerState
    End Sub

Private Sub mText_ChangeText()
    Text1.Text = "Until Now"
    Text2.Text = "9.84"
End Sub

Private Sub mText_UpdateTime(ByVal dblJump As Double)
    Text2.Text = Str(Format(dblJump, "0"))
    DoEvents
End Sub
```

The remaining code is in a class module named TimerState. The **Event** statements declare the procedures initiated when events are raised.

```
Option Explicit
Public Event UpdateTime(ByVal dblJump As Double)
Public Event ChangeText()

Public Sub TimerTask(ByVal Duration As Double)
```

```
    Dim dblStart As Double
    Dim dblSecond As Double
    Dim dblSoFar As Double
    dblStart = Timer
    dblSoFar = dblStart

    Do While Timer < dblStart + Duration
        If Timer - dblSoFar >= 1 Then
            dblSoFar = dblSoFar + 1
            RaiseEvent UpdateTime(Timer - dblStart)
        End If
    Loop

    RaiseEvent ChangeText

End Sub
```

Visual Basic for Applications Reference

Visual Studio 6.0

Exit Statement

See Also Example Specifics

Exits a block of **DoLoop**, **For...Next**, **Function**, **Sub**, or **Property** code.

Syntax

Exit Do

Exit For

Exit Function

Exit Property

Exit Sub

The **Exit** statement syntax has these forms:

Statement	Description
Exit Do	Provides a way to exit a **Do...Loop** statement. It can be used only inside a **Do...Loop** statement. **Exit Do** transfers control to the statement following the **Loop** statement. When used within nested **Do...Loop** statements, **Exit Do** transfers control to the loop that is one nested level above the loop where **Exit Do** occurs.
Exit For	Provides a way to exit a **For** loop. It can be used only in a **For...Next** or **For Each...Next** loop. **Exit For** transfers control to the statement following the **Next** statement. When used within nested **For** loops, **Exit For** transfers control to the loop that is one nested level above the loop where **Exit For** occurs.
Exit Function	Immediately exits the **Function** procedure in which it appears. Execution continues with the statement following the statement that called the **Function**.
Exit Property	Immediately exits the **Property** procedure in which it appears. Execution continues with the statement following the statement that called the **Property** procedure.
Exit Sub	Immediately exits the **Sub** procedure in which it appears. Execution continues with the statement following the statement that called the **Sub** procedure.

Remarks

Do not confuse **Exit** statements with **End** statements. **Exit** does not define the end of a structure.

Visual Basic for Applications Reference

Exit Statement Example

This example uses the **Exit** statement to exit a **For...Next** loop, a **Do...Loop**, and a **Sub** procedure.

```
Sub ExitStatementDemo()
Dim I, MyNum
   Do            ' Set up infinite loop.
      For I = 1 To 1000   ' Loop 1000 times.
         MyNum = Int(Rnd * 1000)   ' Generate random numbers.
         Select Case MyNum   ' Evaluate random number.
            Case 7: Exit For   ' If 7, exit For...Next.
            Case 29: Exit Do   ' If 29, exit Do...Loop.
            Case 54: Exit Sub   ' If 54, exit Sub procedure.
         End Select
      Next I
   Loop
End Sub
```

Visual Basic for Applications Reference

Visual Studio 6.0

FileCopy Statement

See Also Example Specifics

Copies a file.

Syntax

FileCopy *source*, *destination*

The **FileCopy** statement syntax has these named arguments:

Part	Description
source	Required. String expression that specifies the name of the file to be copied. The ***source*** may include directory or folder, and drive.
destination	Required. String expression that specifies the target file name. The ***destination*** may include directory or folder, and drive.

Remarks

If you try to use the **FileCopy** statement on a currently open file, an error occurs.

Visual Basic for Applications Reference

FileCopy Statement Example

This example uses the **FileCopy** statement to copy one file to another. For purposes of this example, assume that SRCFILE is a file containing some data.

```
Dim SourceFile, DestinationFile
SourceFile = "SRCFILE"    ' Define source file name.
DestinationFile = "DESTFILE"    ' Define target file name.
FileCopy SourceFile, DestinationFile    ' Copy source to target.
```

Visual Basic for Applications Reference

Visual Studio 6.0

For Each...Next Statement

See Also Example Specifics

Repeats a group of statements for each element in an array or collection.

Syntax

For Each *element* **In** *group*
[*statements*]
[**Exit For**]
[*statements*]

Next [*element*]

The **For...Each...Next** statement syntax has these parts:

Part	Description
element	Required. Variable used to iterate through the elements of the collection or array. For collections, *element* can only be a Variant variable, a generic object variable, or any specific object variable. For arrays, *element* can only be a **Variant** variable.
group	Required. Name of an object collection or array (except an array of user-defined types).
statements	Optional. One or more statements that are executed on each item in *group*.

Remarks

The **For...Each** block is entered if there is at least one element in *group*. Once the loop has been entered, all the statements in the loop are executed for the first element in *group*. If there are more elements in *group*, the statements in the loop continue to execute for each element. When there are no more elements in *group*, the loop is exited and execution continues with the statement following the **Next** statement.

Any number of **Exit For** statements may be placed anywhere in the loop as an alternative way to exit. **Exit For** is often used after evaluating some condition, for example **IfThen**, and transfers control to the statement immediately following **Next**.

You can nest **For...Each...Next** loops by placing one **For...Each...Next** loop within another. However, each loop *element* must be unique.

Note If you omit *element* in a **Next** statement, execution continues as if *element* is included. If a **Next** statement is encountered before its corresponding **For** statement, an error occurs.

You can't use the **For...Each...Next** statement with an array of user-defined types because a **Variant** can't contain a user-defined type.

For Each...Next Statement Example

This example loops on cells A1:D10 on Sheet1. If one of the cells has a value less than 0.001, the code replaces the value with 0 (zero).

```
For Each c in Worksheets("Sheet1").Range("A1:D10")
    If c.Value < .001 Then
        c.Value = 0
    End If
Next c
```

This example loops on the range named "TestRange" and then displays the number of empty cells in the range.

```
numBlanks = 0
For Each c In Range("TestRange")
    If c.Value = "" Then
        numBlanks = numBlanks + 1
    End If
Next c
MsgBox "There are " & numBlanks & " empty cells in this range."
```

This example closes and saves changes to all workbooks except the one thats running the example.

```
For Each w In Workbooks
    If w.Name <> ThisWorkbook.Name Then
        w.Close savechanges:=True
    End If
Next w
```

This example deletes every worksheet in the active workbook without displaying the confirmation dialog box. There must be at least one other visible sheet in the workbook.

```
Application.DisplayAlerts = False
For Each w In Worksheets
    w.Delete
Next w
Application.DisplayAlerts = True
```

This example creates a new worksheet and then inserts into it a list of all the names in the active workbook, including their formulas in A1-style notation in the language of the user.

```
Set newSheet = ActiveWorkbook.Worksheets.Add
i = 1
For Each nm In ActiveWorkbook.Names
    newSheet.Cells(i, 1).Value = nm.NameLocal
    newSheet.Cells(i, 2).Value = "'" & nm.RefersToLocal
    i = i + 1
Next nm
```

Visual Basic for Applications Reference

Visual Studio 6.0

For...Next Statement

See Also Example Specifics

Repeats a group of statements a specified number of times.

Syntax

For *counter* = *start* **To** *end* [**Step** *step*]
[*statements*]
[**Exit For**]
[*statements*]

Next [*counter*]

The **ForNext** statement syntax has these parts:

Part	Description
counter	Required. Numeric variable used as a loop counter. The variable can't be a Boolean or an array element.
start	Required. Initial value of *counter*.
end	Required. Final value of *counter*.
step	Optional. Amount *counter* is changed each time through the loop. If not specified, *step* defaults to one.
statements	Optional. One or more statements between **For** and **Next** that are executed the specified number of times.

Remarks

The *step* argument can be either positive or negative. The value of the *step* argument determines loop processing as follows:

Value	Loop executes if
Positive or 0	*counter* <= *end*
Negative	*counter* >= *end*

After all statements in the loop have executed, *step* is added to *counter*. At this point, either the statements in the loop execute again (based on the same test that caused the loop to execute initially), or the loop is exited and execution continues with the statement following the **Next** statement.

Tip Changing the value of *counter* while inside a loop can make it more difficult to read and debug your code.

Any number of **Exit For** statements may be placed anywhere in the loop as an alternate way to exit. **Exit For** is often used after evaluating of some condition, for example **If...Then**, and transfers control to the statement immediately following **Next**.

You can nest **For...Next** loops by placing one **For...Next** loop within another. Give each loop a unique variable name as its *counter*. The following construction is correct:

```
For I = 1 To 10
   For J = 1 To 10
      For K = 1 To 10
         ...
      Next K
   Next J
Next I
```

Note If you omit *counter* in a **Next** statement, execution continues as if *counter* is included. If a **Next** statement is encountered before its corresponding **For** statement, an error occurs.

For...Next Statement Example

This content is no longer actively maintained. It is provided as is, for anyone who may still be using these technologies, with no warranties or claims of accuracy with regard to the most recent product version or service release.

This example displays the number of columns in the selection on Sheet1. The code also tests for a multiple-area selection; if one exists, the code loops on the areas of the selection.

```
Worksheets("Sheet1").Activate
areaCount = Selection.Areas.Count
If areaCount <= 1 Then
    MsgBox "The selection contains " & _
        Selection.Columns.Count & " columns."
Else
    For i = 1 To areaCount
        MsgBox "Area " & i & " of the selection contains " & _
            Selection.Areas(i).Columns.Count & " columns."
    Next i
End If
```

This example creates a new worksheet and then inserts a list of the active workbook's sheet names into the first column of the worksheet.

```
Set newSheet = Sheets.Add(Type:=xlWorksheet)
For i = 1 To Sheets.Count
    newSheet.Cells(i, 1).Value = Sheets(i).Name
Next i
```

This example selects every other item in list box one on Sheet1.

```
Dim items() As Boolean
Set lbox = Worksheets("Sheet1").ListBoxes(1)
ReDim items(1 To lbox.ListCount)
For i = 1 To lbox.ListCount
    If i Mod 2 = 1 Then
        items(i) = True
    Else
        items(i) = False
    End If
Next
lbox.MultiSelect = xlExtended
lbox.Selected = items
```

Visual Basic for Applications Reference

Visual Studio 6.0

Function Statement

See Also Example Specifics

Declares the name, arguments, and code that form the body of a **Function** procedure.

Syntax

[**Public** | **Private** | **Friend**] [**Static**] **Function** *name* [**(***arglist***)**] [**As** *type*]
[*statements*]
[*name* = *expression*]
[**Exit Function**]
[*statements*]
[*name* = *expression*]

End Function

The **Function** statement syntax has these parts:

Part	Description
Public	Optional. Indicates that the **Function** procedure is accessible to all other procedures in all modules. If used in a module that contains an **Option Private**, the procedure is not available outside the project.
Private	Optional. Indicates that the **Function** procedure is accessible only to other procedures in the module where it is declared.
Friend	Optional. Used only in a class module. Indicates that the **Function** procedure is visible throughout the project, but not visible to a controller of an instance of an object.
Static	Optional. Indicates that the **Function** procedure's local variables are preserved between calls. The **Static** attribute doesn't affect variables that are declared outside the **Function**, even if they are used in the procedure.
name	Required. Name of the **Function**; follows standard variable naming conventions.
arglist	Optional. List of variables representing arguments that are passed to the **Function** procedure when it is called. Multiple variables are separated by commas.
type	Optional. Data type of the value returned by the **Function** procedure; may be Byte, Boolean, Integer, Long, Currency, Single, Double, Decimal (not currently supported), Date, String, or (except fixed length), Object, Variant, or any user-defined type.
statements	Optional. Any group of statements to be executed within the **Function** procedure.

expression	Optional. Return value of the **Function**.

The *arglist* argument has the following syntax and parts:

[**Optional**] [**ByVal** | **ByRef**] [**ParamArray**] *varname*[**( )**] [**As** *type*] [= *defaultvalue*]

Part	Description
Optional	Optional. Indicates that an argument is not required. If used, all subsequent arguments in *arglist* must also be optional and declared using the **Optional** keyword. **Optional** can't be used for any argument if **ParamArray** is used.
ByVal	Optional. Indicates that the argument is passed by value.
ByRef	Optional. Indicates that the argument is passed by reference. **ByRef** is the default in Visual Basic.
ParamArray	Optional. Used only as the last argument in *arglist* to indicate that the final argument is an **Optional** array of **Variant** elements. The **ParamArray** keyword allows you to provide an arbitrary number of arguments. It may not be used with **ByVal**, **ByRef**, or **Optional**.
varname	Required. Name of the variable representing the argument; follows standard variable naming conventions.
type	Optional. Data type of the argument passed to the procedure; may be **Byte**, **Boolean**, **Integer**, **Long**, **Currency**, **Single**, **Double**, **Decimal** (not currently supported) **Date**, **String** (variable length only), **Object**, **Variant**, or a specific object type. If the parameter is not **Optional**, a user-defined type may also be specified.
defaultvalue	Optional. Any constant or constant expression. Valid for **Optional** parameters only. If the type is an **Object**, an explicit default value can only be **Nothing**.

Remarks

If not explicitly specified using **Public**, **Private**, or **Friend**, **Function** procedures are public by default. If **Static** isn't used, the value of local variables is not preserved between calls. The **Friend** keyword can only be used in class modules. However, **Friend** procedures can be accessed by procedures in any module of a project. A **Friend** procedure doesn't appear in the type library of its parent class, nor can a **Friend** procedure be late bound.

Caution **Function** procedures can be recursive; that is, they can call themselves to perform a given task. However, recursion can lead to stack overflow. The **Static** keyword usually isn't used with recursive **Function** procedures.

All executable code must be in procedures. You can't define a **Function** procedure inside another **Function**, **Sub**, or **Property** procedure.

The **Exit Function** statement causes an immediate exit from a **Function** procedure. Program execution continues with the statement following the statement that called the **Function** procedure. Any number of **Exit Function** statements can appear anywhere in a **Function** procedure.

Like a **Sub** procedure, a **Function** procedure is a separate procedure that can take arguments, perform a series of statements, and change the values of its arguments. However, unlike a **Sub** procedure, you can use a **Function** procedure on the right side of an expression in the same way you use any intrinsic function, such as **Sqr**, **Cos**, or **Chr**, when you want to use the value returned by the function.

You call a **Function** procedure using the function name, followed by the argument list in parentheses, in an expression. See the **Call** statement for specific information on how to call **Function** procedures.

To return a value from a function, assign the value to the function name. Any number of such assignments can appear anywhere within the procedure. If no value is assigned to *name*, the procedure returns a default value: a numeric function returns 0, a string function returns a zero-length string (""), and a **Variant** function returns Empty. A function that returns an object reference returns **Nothing** if no object reference is assigned to *name* (using **Set**) within the **Function**.

The following example shows how to assign a return value to a function named `BinarySearch`. In this case, **False** is assigned to the name to indicate that some value was not found.

```
Function BinarySearch(. . .) As Boolean
. . .
   ' Value not found. Return a value of False.
   If lower > upper Then
      BinarySearch = False
      Exit Function
   End If
. . .
End Function
```

Variables used in **Function** procedures fall into two categories: those that are explicitly declared within the procedure and those that are not. Variables that are explicitly declared in a procedure (using **Dim** or the equivalent) are always local to the procedure. Variables that are used but not explicitly declared in a procedure are also local unless they are explicitly declared at some higher level outside the procedure.

Caution A procedure can use a variable that is not explicitly declared in the procedure, but a naming conflict can occur if anything you defined at the module level has the same name. If your procedure refers to an undeclared variable that has the same name as another procedure, constant, or variable, it is assumed that your procedure refers to that module-level name. Explicitly declare variables to avoid this kind of conflict. You can use an **Option Explicit** statement to force explicit declaration of variables.

Caution Visual Basic may rearrange arithmetic expressions to increase internal efficiency. Avoid using a **Function** procedure in an arithmetic expression when the function changes the value of variables in the same expression.

Visual Basic for Applications Reference

Function Statement Example

This example uses the **Function** statement to declare the name, arguments, and code that form the body of a **Function** procedure. The last example uses hard-typed, initialized **Optional** arguments.

```
' The following user-defined function returns the square root of the
' argument passed to it.
Function CalculateSquareRoot(NumberArg As Double) As Double
   If NumberArg < 0 Then   ' Evaluate argument.
      Exit Function   ' Exit to calling procedure.
   Else
      CalculateSquareRoot = Sqr(NumberArg)   ' Return square root.
   End If
End Function
```

Using the **ParamArray** keyword enables a function to accept a variable number of arguments. In the following definition, `FirstArg` is passed by value.

```
Function CalcSum(ByVal FirstArg As Integer, ParamArray OtherArgs())
Dim ReturnValue
' If the function is invoked as follows:
ReturnValue = CalcSum(4, 3 ,2 ,1)
' Local variables are assigned the following values: FirstArg = 4,
' OtherArgs(1) = 3, OtherArgs(2) = 2, and so on, assuming default
' lower bound for arrays = 1.
```

Optional arguments can have default values and types other than **Variant**.

```
' If a function's arguments are defined as follows:
Function MyFunc(MyStr As String, Optional MyArg1 As _ Integer = 5, Optional MyArg2 = "Dolly")
Dim RetVal
' The function can be invoked as follows:
RetVal = MyFunc("Hello", 2, "World")   ' All 3 arguments supplied.
RetVal = MyFunc("Test", , 5)   ' Second argument omitted.
' Arguments one and three using named-arguments.
RetVal = MyFunc(MyStr:="Hello ", MyArg1:=7)
```

Visual Basic for Applications Reference

Visual Studio 6.0

Get Statement

See Also Example Specifics

Reads data from an open disk file into a variable.

Syntax

Get [**#**]*filenumber*, [*recnumber*], *varname*

The **Get** statement syntax has these parts:

Part	Description
filenumber	Required. Any valid file number.
recnumber	Optional. **Variant** (**Long**). Record number (**Random** mode files) or byte number (**Binary** mode files) at which reading begins.
varname	Required. Valid variable name into which data is read.

Remarks

Data read with **Get** is usually written to a file with **Put**.

The first record or byte in a file is at position 1, the second record or byte is at position 2, and so on. If you omit *recnumber*, the next record or byte following the last **Get** or **Put** statement (or pointed to by the last **Seek** function) is read. You must include delimiting commas, for example:

```
Get #4,,FileBuffer
```

For files opened in **Random** mode, the following rules apply:

- If the length of the data being read is less than the length specified in the **Len** clause of the **Open** statement, **Get** reads subsequent records on record-length boundaries. The space between the end of one record and the beginning of the next record is padded with the existing contents of the file buffer. Because the amount of padding data can't be determined with any certainty, it is generally a good idea to have the record length match the length of the data being read.

- If the variable being read into is a variable-length string, **Get** reads a 2-byte descriptor containing the string length and then reads the data that goes into the variable. Therefore, the record length specified by the **Len** clause in the **Open** statement must be at least 2 bytes greater than the actual length of the string.

- If the variable being read into is a Variant of numeric type, **Get** reads 2 bytes identifying the **VarType** of the **Variant** and then the data that goes into the variable. For example, when reading a **Variant** of **VarType** 3, **Get** reads 6 bytes: 2 bytes identifying the **Variant** as **VarType** 3 (**Long**) and 4 bytes containing the Long data. The record length specified by the **Len** clause in the **Open** statement must be at least 2 bytes greater than the actual number of bytes required to store the variable.
  Note You can use the **Get** statement to read a **Variant** array from disk, but you can't use **Get** to read a scalar **Variant** containing an array. You also can't use **Get** to read objects from disk.

- If the variable being read into is a **Variant** of **VarType** 8 (**String**), **Get** reads 2 bytes identifying the **VarType**, 2 bytes indicating the length of the string, and then reads the string data. The record length specified by the **Len** clause in the **Open** statement must be at least 4 bytes greater than the actual length of the string.

- If the variable being read into is a dynamic array, **Get** reads a descriptor whose length equals 2 plus 8 times the number of dimensions, that is, 2 + 8 * *NumberOfDimensions*. The record length specified by the **Len** clause in the **Open** statement must be greater than or equal to the sum of all the bytes required to read the array data and the array descriptor. For example, the following array declaration requires 118 bytes when the array is written to disk.

  ```
  Dim MyArray(1 To 5,1 To 10) As Integer
  ```

  The 118 bytes are distributed as follows: 18 bytes for the descriptor (2 + 8 * 2), and 100 bytes for the data (5 * 10 * 2).

- If the variable being read into is a fixed-size array, **Get** reads only the data. No descriptor is read.

- If the variable being read into is any other type of variable (not a variable-length string or a **Variant**), **Get** reads only the variable data. The record length specified by the **Len** clause in the **Open** statement must be greater than or equal to the length of the data being read.

- **Get** reads elements of user-defined types as if each were being read individually, except that there is no padding between elements. On disk, a dynamic array in a user-defined type (written with **Put**) is prefixed by a descriptor whose length equals 2 plus 8 times the number of dimensions, that is, 2 + 8 * *NumberOfDimensions*. The record length specified by the **Len** clause in the **Open** statement must be greater than or equal to the sum of all the bytes required to read the individual elements, including any arrays and their descriptors.

For files opened in **Binary** mode, all of the **Random** rules apply, except:

- The **Len** clause in the **Open** statement has no effect. **Get** reads all variables from disk contiguously; that is, with no padding between records.

- For any array other than an array in a user-defined type, **Get** reads only the data. No descriptor is read.

- **Get** reads variable-length strings that aren't elements of user-defined types without expecting the 2-byte length descriptor. The number of bytes read equals the number of characters already in the string. For example, the following statements read 10 bytes from file number 1:

  ```
  VarString = String(10," ")
  Get #1,,VarString
  ```

Visual Basic for Applications Reference

Get Statement Example

This example uses the **Get** statement to read data from a file into a variable. This example assumes that `TESTFILE` is a file containing five records of the user-defined type `Record`.

```
Type Record   ' Define user-defined type.
   ID As Integer
   Name As String * 20
End Type

Dim MyRecord As Record, Position   ' Declare variables.
' Open sample file for random access.
Open "TESTFILE" For Random As #1 Len = Len(MyRecord)
' Read the sample file using the Get statement.
Position = 3   ' Define record number.
Get #1, Position, MyRecord   ' Read third record.
Close #1   ' Close file.
```

Visual Basic for Applications Reference

Visual Studio 6.0

GoSub...Return Statement

See Also Example Specifics

Branches to and returns from a subroutine within a procedure.

Syntax

GoSub *line*
. . .
line
. . .
Return

The *line* argument can be any line label or line number.

Remarks

You can use **GoSub** and **Return** anywhere in a procedure, but **GoSub** and the corresponding **Return** statement must be in the same procedure. A subroutine can contain more than one **Return** statement, but the first **Return** statement encountered causes the flow of execution to branch back to the statement immediately following the most recently executed **GoSub** statement.

Note You can't enter or exit **Sub** procedures with **GoSub...Return**.

Tip Creating separate procedures that you can call may provide a more structured alternative to using **GoSub...Return**.

Visual Basic for Applications Reference

GoSub...Return Statement Example

This example uses **GoSub** to call a subroutine within a **Sub** procedure. The **Return** statement causes the execution to resume at the statement immediately following the **GoSub** statement. The **Exit Sub** statement is used to prevent control from accidentally flowing into the subroutine.

```
Sub GosubDemo()
Dim Num
' Solicit a number from the user.
   Num = InputBox("Enter a positive number to be divided by 2.")
' Only use routine if user enters a positive number.
   If Num > 0 Then GoSub MyRoutine
   Debug.Print Num
   Exit Sub   ' Use Exit to prevent an error.
MyRoutine:
   Num = Num/2   ' Perform the division.
   Return   ' Return control to statement.
End Sub   ' following the GoSub statement.
```

Visual Basic for Applications Reference

Visual Studio 6.0

GoTo Statement

See Also Example Specifics

Branches unconditionally to a specified line within a procedure.

Syntax

GoTo *line*

The required *line* argument can be any line label or line number.

Remarks

GoTo can branch only to lines within the procedure where it appears.

Note Too many **GoTo** statements can make code difficult to read and debug. Use structured control statements (**Do...Loop**, **For...Next**, **If...Then...Else**, **Select Case**) whenever possible.

Visual Basic for Applications Reference

GoTo Statement Example

This example uses the **GoTo** statement to branch to line labels within a procedure.

```
Sub GotoStatementDemo()
Dim Number, MyString
   Number = 1    ' Initialize variable.
   ' Evaluate Number and branch to appropriate label.
   If Number = 1 Then GoTo Line1 Else GoTo Line2

Line1:
   MyString = "Number equals 1"
   GoTo LastLine   ' Go to LastLine.
Line2:
   ' The following statement never gets executed.
   MyString = "Number equals 2"
LastLine:
   Debug.Print MyString   ' Print "Number equals 1" in
      ' the Immediate window.
End Sub
```