Visual Basic for Applications Reference

Visual Studio 6.0

If...Then...Else Statement

See Also Example Specifics

Conditionally executes a group of statements, depending on the value of an expression.

Syntax

If *condition* **Then** [*statements*] [**Else** *elsestatements*]

Or, you can use the block form syntax:

If *condition* **Then**
[*statements*]

[**ElseIf** *condition-n* **Then**
[*elseifstatements*] . . .

[**Else**
[*elsestatements*]]

End If

The **If...Then...Else** statement syntax has these parts:

Part	Description
condition	Required. One or more of the following two types of expressions:
	A numeric expression or string expression that evaluates to **True** or **False**. If *condition* is Null, *condition* is treated as **False**.
	An expression of the form **TypeOf** *objectname* **Is** *objecttype*. The *objectname* is any object reference and *objecttype* is any valid object type. The expression is **True** if *objectname* is of the object type specified by *objecttype*; otherwise it is **False**.
statements	Optional in block form; required in single-line form that has no **Else** clause. One or more statements separated by colons; executed if *condition* is **True**.
condition-n	Optional. Same as *condition*.
elseifstatements	Optional. One or more statements executed if associated *condition-n* is **True**.
elsestatements	Optional. One or more statements executed if no previous *condition* or *condition-n* expression is **True**.

Remarks

You can use the single-line form (first syntax) for short, simple tests. However, the block form (second syntax) provides more structure and flexibility than the single-line form and is usually easier to read, maintain, and debug.

Note With the single-line form, it is possible to have multiple statements executed as the result of an **If...Then** decision. All statements must be on the same line and separated by colons, as in the following statement:

```
If A > 10 Then A = A + 1 : B = B + A : C = C + B
```

A block form **If** statement must be the first statement on a line. The **Else**, **ElseIf**, and **End If** parts of the statement can have only a line number or line label preceding them. The block **If** must end with an **End If** statement.

To determine whether or not a statement is a block **If**, examine what follows the **Then** keyword. If anything other than a comment appears after **Then** on the same line, the statement is treated as a single-line **If** statement.

The **Else** and **ElseIf** clauses are both optional. You can have as many **ElseIf** clauses as you want in a block **If**, but none can appear after an **Else** clause. Block **If** statements can be nested; that is, contained within one another.

When executing a block **If** (second syntax), *condition* is tested. If *condition* is **True**, the statements following **Then** are executed. If *condition* is **False**, each **ElseIf** condition (if any) is evaluated in turn. When a **True** condition is found, the statements immediately following the associated **Then** are executed. If none of the **ElseIf** conditions are **True** (or if there are no **ElseIf** clauses), the statements following **Else** are executed. After executing the statements following **Then** or **Else**, execution continues with the statement following **End If**.

Tip **Select Case** may be more useful when evaluating a single expression that has several possible actions. However, the **TypeOf** *objectname* **Is** *objecttype* clause can't be used with the **Select Case** statement.

Note **TypeOf** cannot be used with hard data types such as Long, Integer, and so forth other than Object.

If...Then...Else 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 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 sets the standard font to Geneva (on the Macintosh) or Arial (in Windows).

```
If Application.OperatingSystem Like "*Macintosh*" Then
    Application.StandardFont = "Geneva"
Else
    Application.StandardFont = "Arial"
End If
```

Visual Basic for Applications Reference

Visual Studio 6.0

Implements Statement

See Also Example Specifics

Specifies an interface or class that will be implemented in the class module in which it appears.

Syntax

Implements [*InterfaceName* | *Class*]

The required *InterfaceName* or *Class* is the name of an interface or class in a type library whose methods will be implemented by the corresponding methods in the Visual Basic class.

Remarks

An interface is a collection of prototypes representing the members (methods and properties) the interface encapsulates; that is, it contains only the declarations for the member procedures. A class provides an implementation of all of the methods and properties of one or more interfaces. Classes provide the code used when each function is called by a controller of the class. All classes implement at least one interface, which is considered the default interface of the class. In Visual Basic, any member that isn't explicitly a member of an implemented interface is implicitly a member of the default interface.

When a Visual Basic class implements an interface, the Visual Basic class provides its own versions of all the **Public** procedures specified in the type library of the Interface. In addition to providing a mapping between the interface prototypes and your procedures, the **Implements** statement causes the class to accept COM QueryInterface calls for the specified interface ID.

Note Visual Basic does not implement derived classes or interfaces.

When you implement an interface or class, you must include all the **Public** procedures involved. A missing member in an implementation of an interface or class causes an error. If you don't place code in one of the procedures in a class you are implementing, you can raise the appropriate error (**Const** E_NOTIMPL = &H80004001) so a user of the implementation understands that a member is not implemented.

The **Implements** statement can't appear in a standard module.

Visual Basic for Applications Reference

Implements Statement Example

The following example shows how to use the **Implements** statement to make a set of declarations available to multiple classes. By sharing the declarations through the **Implements** statement, neither class has to make any declarations itself.

Assume there are two forms. The Selector form has two buttons, Customer Data and Supplier Data. To enter name and address information for a customer or a supplier, the user clicks the Customer button or the Supplier button on the Selector form, and then enters the name and address using the Data Entry form. The Data Entry form has two text fields, Name and Address.

The following code for the shared declarations is in a class called PersonalData:

```
Public Name As String
Public Address As String
```

The code supporting the customer data is in a class module called Customer:

```
Implements PersonalData
Private Property Get PersonalData_Address() As String
PersonalData_Address = "CustomerAddress"
End Property

Private Property Let PersonalData_Address(ByVal RHS As String)
'
End Property

Private Property Let PersonalData_Name(ByVal RHS As String)
'
End Property

Private Property Get PersonalData_Name() As String
PersonalData_Name = "CustomerName"
End Property
```

The code supporting the supplier data is in a class module called Supplier:

```
Implements PersonalData

Private Property Get PersonalData_Address() As String
PersonalData_Address = "SupplierAddress"
End Property

Private Property Let PersonalData_Address(ByVal RHS As String)
'
End Property

Private Property Let PersonalData_Name(ByVal RHS As String)
'
End Property

Private Property Get PersonalData_Name() As String
PersonalData_Name = "SupplierName"
End Property
```

The following code supports the Selector form:

```
Private cust As New Customer
Private sup As New Supplier

Private Sub Command1_Click()
Dim frm2 As New Form2
    Set frm2.PD = cust
    frm2.Show 1
End Sub

Private Sub Command2_Click()
Dim frm2 As New Form2
    Set frm2.PD = sup
    frm2.Show 1
End Sub
```

The following code supports the Data Entry form:

```
Private m_pd As PersonalData
Private Sub Form_Load()
    With m_pd
        Text1 = .Name
        Text2 = .Address
    End With
End Sub
Public Property Set PD(Data As PersonalData)
    Set m_pd = Data
End Property
```

Visual Basic for Applications Reference

Visual Studio 6.0

Input # Statement

See Also Example Specifics

Reads data from an open sequential file and assigns the data to variables.

Syntax

Input #*filenumber, varlist*

The **Input #** statement syntax has these parts:

Part	Description
filenumber	Required. Any valid file number.
varlist	Required. Comma-delimited list of variables that are assigned values read from the file can't be an array or object variable. However, variables that describe an element of an array or user-defined type may be used.

Remarks

Data read with **Input #** is usually written to a file with **Write #**. Use this statement only with files opened in **Input** or **Binary** mode.

When read, standard string or numeric data is assigned to variables without modification. The following table illustrates how other input data is treated:

Data	Value assigned to variable
Delimiting comma or blank line	Empty
#NULL#	Null
#TRUE# or #FALSE#	**True** or **False**
#yyyy-mm-dd hh:mm:ss#	The date and/or time represented by the expression
#ERROR *errornumber#*	*errornumber* (variable is a Variant tagged as an error)

Double quotation marks (" ") within input data are ignored.

Note You should not write strings that contain embedded quotation marks, for example, `"1,2""X"` for use with the **Input #** statement: **Input #** parses this string as two complete and separate strings.

Data items in a file must appear in the same order as the variables in *varlist* and match variables of the same data type. If a variable is numeric and the data is not numeric, a value of zero is assigned to the variable.

If you reach the end of the file while you are inputting a data item, the input is terminated and an error occurs.

Note To be able to correctly read data from a file into variables using **Input #**, use the **Write #** statement instead of the **Print #** statement to write the data to the files. Using **Write #** ensures each separate data field is properly delimited.

Visual Basic for Applications Reference

Input # Statement Example

This example uses the **Input #** statement to read data from a file into two variables. This example assumes that TESTFILE is a file with a few lines of data written to it using the **Write #** statement; that is, each line contains a string in quotations and a number separated by a comma, for example, ("Hello", 234).

```
Dim MyString, MyNumber
Open "TESTFILE" For Input As #1   ' Open file for input.
Do While Not EOF(1)   ' Loop until end of file.
   Input #1, MyString, MyNumber   ' Read data into two variables.
   Debug.Print MyString, MyNumber   ' Print data to the Immediate window.
Loop
Close #1   ' Close file.
```

Visual Basic for Applications Reference

Visual Studio 6.0

Kill Statement

See Also Example Specifics

Deletes files from a disk.

Syntax

Kill *pathname*

The required *pathname* argument is a string expression that specifies one or more file names to be deleted. The *pathname* may include the directory or folder, and the drive.

Remarks

In Microsoft Windows, **Kill** supports the use of multiple-character (*****) and single-character (**?**) wildcards to specify multiple files

Visual Basic for Applications Reference

Kill Statement Example

This example uses the **Kill** statement to delete a file from a disk.

```
' Assume TESTFILE is a file containing some data.
Kill "TestFile"   ' Delete file.

' Delete all *.TXT files in current directory.
Kill "*.TXT"
```

Visual Basic for Applications Reference

Visual Studio 6.0

Let Statement

See Also Example Specifics

Assigns the value of an expression to a variable or property.

Syntax

[**Let**] *varname* = *expression*

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

Part	Description
Let	Optional. Explicit use of the **Let** keyword is a matter of style, but it is usually omitted.
varname	Required. Name of the variable or property; follows standard variable naming conventions.
expression	Required. Value assigned to the variable or property.

Remarks

A value expression can be assigned to a variable or property only if it is of a data type that is compatible with the variable. You can't assign string expressions to numeric variables, and you can't assign numeric expressions to string variables. If you do, an error occurs at compile time.

Variant variables can be assigned either string or numeric expressions. However, the reverse is not always true. Any **Variant** except a Null can be assigned to a string variable, but only a **Variant** whose value can be interpreted as a number can be assigned to a numeric variable. Use the **IsNumeric** function to determine if the **Variant** can be converted to a number.

Caution Assigning an expression of one numeric type to a variable of a different numeric type coerces the value of the expression into the numeric type of the resulting variable.

Let statements can be used to assign one record variable to another only when both variables are of the same user-defined type. Use the **LSet** statement to assign record variables of different user-defined types. Use the **Set** statement to assign object references to variables.

Visual Basic for Applications Reference

Let Statement Example

This example assigns the values of expressions to variables using the explicit **Let** statement.

```
Dim MyStr, MyInt
' The following variable assignments use the Let statement.
Let MyStr = "Hello World"
Let MyInt = 5
```

The following are the same assignments without the **Let** statement.

```
Dim MyStr, MyInt
MyStr = "Hello World"
MyInt = 5
```

Visual Basic for Applications Reference

Visual Studio 6.0

Line Input # Statement

See Also Example Specifics

Reads a single line from an open sequential file and assigns it to a String variable.

Syntax

Line Input #*filenumber*, *varname*

The **Line Input #** statement syntax has these parts:

Part	Description
filenumber	Required. Any valid file number.
varname	Required. Valid Variant or **String** variable name.

Remarks

Data read with **Line Input #** is usually written from a file with **Print #**.

The **Line Input #** statement reads from a file one character at a time until it encounters a carriage return (**Chr(**13**)**) or carriage returnlinefeed (**Chr(**13**)** + **Chr(**10**)**) sequence. Carriage returnlinefeed sequences are skipped rather than appended to the character string.

Visual Basic for Applications Reference

Line Input # Statement Example

This example uses the **Line Input #** statement to read a line from a sequential file and assign it to a variable. This example assumes that `TESTFILE` is a text file with a few lines of sample data.

```
Dim TextLine
Open "TESTFILE" For Input As #1   ' Open file.
Do While Not EOF(1)   ' Loop until end of file.
   Line Input #1, TextLine   ' Read line into variable.
   Debug.Print TextLine   ' Print to the Immediate window.
Loop
Close #1   ' Close file.
```

Visual Basic Reference

Visual Studio 6.0

Load Statement

See Also Example

Loads a form or control into memory.

Syntax

Load *object*

The *object* placeholder is the name of a **Form** object, **MDIForm** object, or control array element to load.

Remarks

You don't need to use the **Load** statement with forms unless you want to load a form without displaying it. Any reference to a form (except in a **Set** or **If...TypeOf** statement) automatically loads it if it's not already loaded. For example, the **Show** method loads a form before displaying it. Once the form is loaded, its properties and controls can be altered by the application, whether or not the form is actually visible. Under some circumstances, you may want to load all your forms during initialization and display them later as they're needed.

When Visual Basic loads a **Form** object, it sets form properties to their initial values and then performs the Load event procedure. When an application starts, Visual Basic automatically loads and displays the application's startup form.

If you load a **Form** whose **MDIChild** property is set to **True** (in other words, the child form) before loading an **MDIForm**, the **MDIForm** is automatically loaded before the child form. MDI child forms cannot be hidden, and thus are immediately visible after the Form_Load event procedure ends.

The standard dialog boxes produced by Visual Basic functions such as **MsgBox** and **InputBox** do not need to be loaded, shown, or unloaded, but can simply be invoked directly.

Visual Basic Reference

Load Statement Example

This example uses the **Load** statement to load a **Form** object. To try this example, paste the code into the Declarations section of a **Form** object, and then run the example and click the **Form** object.

```
Private Sub Form_Click ()
   Dim Answer, Msg as String   ' Declare variable.
   Unload Form1   ' Unload form.
   Msg = "Form1 has been unloaded. Choose Yes to load and "
   Msg = Msg & "display the form. Choose No to load the form "
   Msg = Msg & "and leave it invisible."
   Answer = MsgBox(Msg, vbYesNo)   ' Get user response.
   If Answer = vbYes Then   ' Evaluate answer.
      Show   ' If Yes, show form.
   Else
      Load Form1   ' If No, just load it.
      Msg = "Form1 is now loaded. Choose OK to display it."
      MsgBox Msg   ' Display message.
      Show   ' Show form.
   End If
End Sub
```

Visual Basic for Applications Reference

Visual Studio 6.0

Lock, Unlock Statements

See Also Example Specifics

Controls access by other processes to all or part of a file opened using the **Open** statement.

Syntax

Lock [**#**]*filenumber*[, *recordrange*]
. . .

Unlock [**#**]*filenumber*[, *recordrange*]

The **Lock** and **Unlock** statement syntax has these parts:

Part	Description
filenumber	Required. Any valid file number.
recordrange	Optional. The range of records to lock or unlock.

Settings

The *recordrange* argument settings are:

recnumber | [*start*] **To** *end*

Setting	Description
recnumber	Record number (**Random** mode files) or byte number (**Binary** mode files) at which locking or unlocking begins.
start	Number of the first record or byte to lock or unlock.
end	Number of the last record or byte to lock or unlock.

Remarks

The **Lock** and **Unlock** statements are used in environments where several processes might need access to the same file.

Lock and **Unlock** statements are always used in pairs. The arguments to **Lock** and **Unlock** must match exactly.

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 specify just one record, then only that record is locked or unlocked. If you specify a range of records and omit a starting record (*start*), all records from the first record to the end of the range (*end*) are locked or unlocked. Using **Lock** without *recnumber* locks the entire file; using **Unlock** without *recnumber* unlocks the entire file.

If the file has been opened for sequential input or output, **Lock** and **Unlock** affect the entire file, regardless of the range specified by *start* and *end*.

Caution Be sure to remove all locks with an **Unlock** statement before closing a file or quitting your program. Failure to remove locks produces unpredictable results.

Visual Basic for Applications Reference

Lock, Unlock Statements Example

This example illustrates the use of the **Lock** and **Unlock** statements. While a record is being modified, access by other processes to the record is denied. 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, RecordNumber   ' Declare variables.
' Open sample file for random access.
Open "TESTFILE" For Random Shared As #1 Len = Len(MyRecord)
RecordNumber = 4   ' Define record number.
Lock #1, RecordNumber   ' Lock record.
Get #1, RecordNumber, MyRecord   ' Read record.
MyRecord.ID = 234   ' Modify record.
MyRecord.Name = "John Smith"
Put #1, RecordNumber, MyRecord   ' Write modified record.
Unlock #1, RecordNumber   ' Unlock current record.
Close #1   ' Close file.
```

Visual Basic for Applications Reference

Visual Studio 6.0

LSet Statement

See Also Example Specifics

Left aligns a string within a string variable, or copies a variable of one user-defined type to another variable of a different user-defined type.

Syntax

LSet *stringvar* = *string*

LSet *varname1* = *varname2*

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

Part	Description
stringvar	Required. Name of string variable.
string	Required. String expression to be left-aligned within *stringvar*.
varname1	Required. Variable name of the user-defined type being copied to.
varname2	Required. Variable name of the user-defined type being copied from.

Remarks

LSet replaces any leftover characters in *stringvar* with spaces.

If *string* is longer than *stringvar*, **LSet** places only the leftmost characters, up to the length of the *stringvar*, in *stringvar*.

Warning Using **LSet** to copy a variable of one user-defined type into a variable of a different user-defined type is not recommended. Copying data of one data type into space reserved for a different data type can cause unpredictable results.

When you copy a variable from one user-defined type to another, the binary data from one variable is copied into the memory space of the other, without regard for the data types specified for the elements.

Visual Basic for Applications Reference

LSet Statement Example

This example uses the **LSet** statement to left align a string within a string variable. Although **LSet** can also be used to copy a variable of one user-defined type to another variable of a different, but compatible, user-defined type, this practice is not recommended. Due to the varying implementations of data structures among platforms, such a use of **LSet** can't be guaranteed to be portable.

```
Dim MyString
MyString = "0123456789"   ' Initialize string.
Lset MyString = "<-Left"   ' MyString contains "<-Left    ".
```

Visual Basic for Applications Reference

Visual Studio 6.0

Mid Statement

See Also Example Specifics

Replaces a specified number of characters in a **Variant** (**String**) variable with characters from another string.

Syntax

Mid(*stringvar*, *start*[, *length*]) = *string*

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

Part	Description
stringvar	Required. Name of string variable to modify.
start	Required; **Variant** (**Long**). Character position in *stringvar* where the replacement of text begins.
length	Optional; **Variant** (**Long**). Number of characters to replace. If omitted, all of *string* is used.
string	Required. String expression that replaces part of *stringvar*.

Remarks

The number of characters replaced is always less than or equal to the number of characters in *stringvar*.

Note Use the **MidB** statement with byte data contained in a string. In the **MidB** statement, *start* specifies the byte position within *stringvar* where replacement begins and *length* specifies the numbers of bytes to replace.

Visual Basic for Applications Reference

Mid Statement Example

This example uses the **Mid** statement to replace a specified number of characters in a string variable with characters from another string.

```
Dim MyString
MyString = "The dog jumps"   ' Initialize string.
Mid(MyString, 5, 3) = "fox"   ' MyString = "The fox jumps".
Mid(MyString, 5) = "cow"   ' MyString = "The cow jumps".
Mid(MyString, 5) = "cow jumped over"   ' MyString = "The cow jumpe".
Mid(MyString, 5, 3) = "duck"   ' MyString = "The duc jumpe".
```

Visual Basic for Applications Reference

Visual Studio 6.0

MkDir Statement

See Also Example Specifics

Creates a new directory or folder.

Syntax

MkDir *path*

The required *path* argument is a string expression that identifies the directory or folder to be created. The *path* may include the drive. If no drive is specified, **MkDir** creates the new directory or folder on the current drive.

Visual Basic for Applications Reference

MkDir Statement Example

This example uses the **MkDir** statement to create a directory or folder. If the drive is not specified, the new directory or folder is created on the current drive.

```
MkDir "MYDIR"   ' Make new directory or folder.
```

Visual Basic for Applications Reference

Visual Studio 6.0

Name Statement

See Also Example Specifics

Renames a disk file, directory, or folder.

Syntax

Name *oldpathname* **As** *newpathname*

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

Part	Description
oldpathname	Required. String expression that specifies the existing file name and location may include directory or folder, and drive.
newpathname	Required. String expression that specifies the new file name and location may include directory or folder, and drive. The file name specified by *newpathname* can't already exist.

Remarks

The Name statement renames a file and moves it to a different directory or folder, if necessary. Name can move a file across drives, but it can only rename an existing directory or folder when both newpathname and oldpathname are located on the same drive. Name cannot create a new file, directory, or folder.

Using **Name** on an open file produces an error. You must close an open file before renaming it. **Name** arguments cannot include multiple-character (*****) and single-character (**?**) wildcards.

Visual Basic for Applications Reference

Name Statement Example

This example uses the **Name** statement to rename a file. For purposes of this example, assume that the directories or folders that are specified already exist.

```
Dim OldName, NewName
OldName = "OLDFILE": NewName = "NEWFILE"   ' Define file names.
Name OldName As NewName   ' Rename file.

OldName = "C:\MYDIR\OLDFILE": NewName = "C:\YOURDIR\NEWFILE"
Name OldName As NewName   ' Move and rename file.
```

Visual Basic for Applications Reference

Visual Studio 6.0

On Error Statement

See Also Example Specifics

Enables an error-handling routine and specifies the location of the routine within a procedure; can also be used to disable an error-handling routine.

Syntax

On Error GoTo *line*

On Error Resume Next

On Error GoTo 0

The **On Error** statement syntax can have any of the following forms:

Statement	Description
On Error GoTo *line*	Enables the error-handling routine that starts at *line* specified in the required *line* argument. The *line* argument is any line label or line number. If a run-time error occurs, control branches to *line*, making the error handler active. The specified *line* must be in the same procedure as the **On Error** statement; otherwise, a compile-time error occurs.
On Error Resume Next	Specifies that when a run-time error occurs, control goes to the statement immediately following the statement where the error occurred where execution continues. Use this form rather than **On Error GoTo** when accessing objects.
On Error GoTo 0	Disables any enabled error handler in the current procedure.

Remarks

If you don't use an **On Error** statement, any run-time error that occurs is fatal; that is, an error message is displayed and execution stops.

An "enabled" error handler is one that is turned on by an **On Error** statement; an "active" error handler is an enabled handler that is in the process of handling an error. If an error occurs while an error handler is active (between the occurrence of the error and a **Resume**, **Exit Sub**, **Exit Function**, or **Exit Property** statement), the current procedure's error handler can't handle the error. Control returns to the calling procedure. If the calling procedure has an enabled error handler, it is activated to handle the error. If the calling procedure's error handler is also active, control passes back through previous calling procedures until an enabled, but inactive, error handler is found. If no inactive, enabled error handler is found, the error is fatal at the point at which it actually occurred. Each time the error handler passes control back to a calling procedure, that

procedure becomes the current procedure. Once an error is handled by an error handler in any procedure, execution resumes in the current procedure at the point designated by the **Resume** statement.

Note An error-handling routine is not a **Sub** procedure or **Function** procedure. It is a section of code marked by a line label or line number.

Error-handling routines rely on the value in the **Number** property of the **Err** object to determine the cause of the error. The error-handling routine should test or save relevant property values in the **Err** object before any other error can occur or before a procedure that might cause an error is called. The property values in the **Err** object reflect only the most recent error. The error message associated with **Err.Number** is contained in **Err.Description**.

On Error Resume Next causes execution to continue with the statement immediately following the statement that caused the run-time error, or with the statement immediately following the most recent call out of the procedure containing the **On Error Resume Next** statement. This statement allows execution to continue despite a run-time error. You can place the error-handling routine where the error would occur, rather than transferring control to another location within the procedure. An **On Error Resume Next** statement becomes inactive when another procedure is called, so you should execute an **On Error Resume Next** statement in each called routine if you want inline error handling within that routine.

Note The **On Error Resume Next** construct may be preferable to **On Error GoTo** when handling errors generated during access to other objects. Checking **Err** after each interaction with an object removes ambiguity about which object was accessed by the code. You can be sure which object placed the error code in **Err.Number**, as well as which object originally generated the error (the object specified in **Err.Source**).

On Error GoTo 0 disables error handling in the current procedure. It doesn't specify line 0 as the start of the error-handling code, even if the procedure contains a line numbered 0. Without an **On Error GoTo 0** statement, an error handler is automatically disabled when a procedure is exited.

To prevent error-handling code from running when no error has occurred, place an **Exit Sub**, **Exit Function**, or **Exit Property** statement immediately before the error-handling routine, as in the following fragment:

```
Sub InitializeMatrix(Var1, Var2, Var3, Var4)
   On Error GoTo ErrorHandler
   . . .
   Exit Sub
ErrorHandler:
   . . .
   Resume Next
End Sub
```

Here, the error-handling code follows the **Exit Sub** statement and precedes the **End Sub** statement to separate it from the procedure flow. Error-handling code can be placed anywhere in a procedure.

Untrapped errors in objects are returned to the controlling application when the object is running as an executable file. Within the development environment, untrapped errors are only returned to the controlling application if the proper options are set. See your host application's documentation for a description of which options should be set during debugging, how to set them, and whether the host can create classes.

If you create an object that accesses other objects, you should try to handle errors passed back from them unhandled. If you cannot handle such errors, map the error code in **Err.Number** to one of your own errors, and then pass them back to the caller of your object. You should specify your error by adding your error code to the **vbObjectError** constant. For example, if your error code is 1052, assign it as follows:

```
Err.Number = vbObjectError + 1052
```

Note System errors during calls to Windows dynamic-link libraries (DLL) do not raise exceptions and cannot be trapped with Visual Basic error trapping. When calling DLL functions, you should check each return value for success or failure (according to the API specifications), and in the event of a failure, check the value in the **Err** object's **LastDLLError** property.

Visual Basic for Applications Reference

On Error Statement Example

This example first uses the **On Error GoTo** statement to specify the location of an error-handling routine within a procedure. In the example, an attempt to delete an open file generates error number 55. The error is handled in the error-handling routine, and control is then returned to the statement that caused the error. The **On Error GoTo 0** statement turns off error trapping. Then the **On Error Resume Next** statement is used to defer error trapping so that the context for the error generated by the next statement can be known for certain. Note that **Err.Clear** is used to clear the **Err** object's properties after the error is handled.

```
Sub OnErrorStatementDemo()
   On Error GoTo ErrorHandler   ' Enable error-handling routine.
   Open "TESTFILE" For Output As #1   ' Open file for output.
   Kill "TESTFILE"   ' Attempt to delete open
            ' file.
   On Error Goto 0   ' Turn off error trapping.
   On Error Resume Next   ' Defer error trapping.
   ObjectRef = GetObject("MyWord.Basic")   ' Try to start nonexistent
            ' object, then test for
'Check for likely Automation errors.
   If Err.Number = 440 Or Err.Number = 432 Then
      ' Tell user what happened. Then clear the Err object.
      Msg = "There was an error attempting to open the Automation object!"
      MsgBox Msg, , "Deferred Error Test"
      Err.Clear   ' Clear Err object fields
   End If
Exit Sub      ' Exit to avoid handler.
ErrorHandler:   ' Error-handling routine.
   Select Case Err.Number   ' Evaluate error number.
      Case 55   ' "File already open" error.
         Close #1   ' Close open file.
      Case Else
         ' Handle other situations here...
   End Select
   Resume   ' Resume execution at same line
            ' that caused the error.
End Sub
```

Visual Basic for Applications Reference

Visual Studio 6.0

On...GoSub, On...GoTo Statements

See Also Example Specifics

Branch to one of several specified lines, depending on the value of an expression.

Syntax

On *expression* **GoSub** *destinationlist*

On *expression* **GoTo** *destinationlist*

The **On...GoSub** and **On...GoTo** statement syntax has these parts:

Part	Description
expression	Required. Any numeric expression that evaluates to a whole number between 0 and 255, inclusive. If *expression* is any number other than a whole number, it is rounded before it is evaluated.
destinationlist	Required. List of line numbers or line labels separated by commas.

Remarks

The value of *expression* determines which line is branched to in *destinationlist*. If the value of *expression* is less than 1 or greater than the number of items in the list, one of the following results occurs:

If *expression* is	Then
Equal to 0	Control drops to the statement following **On...GoSub** or **On...GoTo**.
Greater than number of items in list	Control drops to the statement following **On...GoSub** or **On...GoTo**.
Negative	An error occurs.
Greater than 255	An error occurs.

You can mix line numbers and line labels in the same list. You can use as many line labels and line numbers as you like with **On...GoSub** and **On...GoTo**. However, if you use more labels or numbers than fit on a single line, you must use the line-continuation character to continue the logical line onto the next physical line.

Tip **Select Case** provides a more structured and flexible way to perform multiple branching.

Visual Basic for Applications Reference

On...GoSub, On...GoTo Statements Example

This example uses the **On...GoSub** and **On...GoTo** statements to branch to subroutines and line labels, respectively.

```
Sub OnGosubGotoDemo()
Dim Number, MyString
   Number = 2   ' Initialize variable.
   ' Branch to Sub2.
   On Number GoSub Sub1, Sub2   ' Execution resumes here after
      ' On...GoSub.
   On Number GoTo Line1, Line2   ' Branch to Line2.
   ' Execution does not resume here after On...GoTo.
   Exit Sub
Sub1:
   MyString = "In Sub1" : Return
Sub2:
   MyString = "In Sub2" : Return
Line1:
   MyString = "In Line1"
Line2:
   MyString = "In Line2"
End Sub
```

Visual Basic for Applications Reference

Visual Studio 6.0

Open Statement

See Also Example Specifics

Enables input/output (I/O) to a file.

Syntax

Open *pathname* **For** *mode* [**Access** *access*] [*lock*] **As** [**#**]*filenumber* [**Len**=*reclength*]

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

Part	Description
pathname	Required. String expression that specifies a file name may include directory or folder, and drive.
mode	Required. Keyword specifying the file mode: **Append**, **Binary**, **Input**, **Output**, or **Random**. If unspecified, the file is opened for **Random** access.
access	Optional. Keyword specifying the operations permitted on the open file: **Read**, **Write**, or **Read Write**.
lock	Optional. Keyword specifying the operations restricted on the open file by other processes: **Shared**, **Lock Read**, **Lock Write**, and **Lock Read Write**.
filenumber	Required. A valid file number in the range 1 to 511, inclusive. Use the **FreeFile** function to obtain the next available file number.
reclength	Optional. Number less than or equal to 32,767 (bytes). For files opened for random access, this value is the record length. For sequential files, this value is the number of characters buffered.

Remarks

You must open a file before any I/O operation can be performed on it. **Open** allocates a buffer for I/O to the file and determines the mode of access to use with the buffer.

> **Security Note** When writing to files, an application may need to create a file if the file to which it is trying to write does not exist. To do so, it needs permission for the directory in which the file is to be created. However, if the file specified by *FileName* does exist, the application only needs **Write** permission to the file itself. Wherever possible, it is more secure to create the file during deployment and only grant **Write** permission to that file, rather than to the entire directory. It is also more secure to write data to user directories than to the root directory or the Program Files directory.

If the file specified by *pathname* doesn't exist, it is created when a file is opened for **Append**, **Binary**, **Output**, or **Random** modes.

If the file is already opened by another process and the specified type of access is not allowed, the **Open** operation fails and an error occurs.

The **Len** clause is ignored if *mode* is **Binary**.

Important In **Binary**, **Input**, and **Random** modes, you can open a file using a different file number without first closing the file. In **Append** and **Output** modes, you must close a file before opening it with a different file number.

Visual Basic for Applications Reference

Open Statement Example

This example illustrates various uses of the **Open** statement to enable input and output to a file.

The following code opens the file TESTFILE in sequential-input mode.

```
Open "TESTFILE" For Input As #1
' Close before reopening in another mode.
Close #1
```

This example opens the file in Binary mode for writing operations only.

```
Open "TESTFILE" For Binary Access Write As #1
' Close before reopening in another mode.
Close #1
```

The following example opens the file in Random mode. The file contains 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   ' Declare variable.
Open "TESTFILE" For Random As #1 Len = Len(MyRecord)
' Close before reopening in another mode.
Close #1
```

This code example opens the file for sequential output; any process can read or write to file.

```
Open "TESTFILE" For Output Shared As #1
' Close before reopening in another mode.
Close #1
```

This code example opens the file in Binary mode for reading; other processes can't read file.

```
Open "TESTFILE" For Binary Access Read Lock Read As #1
```

Visual Basic for Applications Reference

Visual Studio 6.0

Option Base Statement

See Also Example Specifics

Used at module level to declare the default lower bound for array subscripts.

Syntax

Option Base {**0** | **1**}

Remarks

Because the default base is **0**, the **Option Base** statement is never required. If used, the statement must appear in a module before any procedures. **Option Base** can appear only once in a module and must precede array declarations that include dimensions.

Note The **To** clause in the **Dim**, **Private**, **Public**, **ReDim**, and **Static** statements provides a more flexible way to control the range of an array's subscripts. However, if you don't explicitly set the lower bound with a **To** clause, you can use **Option Base** to change the default lower bound to 1. The base of an array created with the the **ParamArray** keyword is zero; **Option Base** does not affect **ParamArray** (or the **Array** function, when qualified with the name of its type library, for example **VBA.Array**).

The **Option Base** statement only affects the lower bound of arrays in the module where the statement is located.

Visual Basic for Applications Reference

Option Base Statement Example

This example uses the **Option Base** statement to override the default base array subscript value of 0. The **LBound** function returns the smallest available subscript for the indicated dimension of an array. The **Option Base** statement is used at the module level only.

```
Option base 1   ' Set default array subscripts to 1.

Dim Lower
Dim MyArray(20), TwoDArray(3, 4)   ' Declare array variables.
Dim ZeroArray(0 To 5)   ' Override default base subscript.
' Use LBound function to test lower bounds of arrays.
Lower = LBound(MyArray)   ' Returns 1.
Lower = LBound(TwoDArray, 2)   ' Returns 1.
Lower = LBound(ZeroArray)   ' Returns 0.
```

Visual Basic for Applications Reference

Visual Studio 6.0

Option Compare Statement

See Also Example Specifics

Used at module level to declare the default comparison method to use when string data is compared.

Syntax

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

Remarks

If used, the **Option Compare** statement must appear in a module before any procedures.

The **Option Compare** statement specifies the string comparison method (**Binary**, **Text**, or **Database**) for a module. If a module doesn't include an **Option Compare** statement, the default text comparison method is **Binary**.

Option Compare Binary results in string comparisons based on a sort order derived from the internal binary representations of the characters. In Microsoft Windows, sort order is determined by the code page. A typical binary sort order is shown in the following example:

```
A < B < E < Z < a < b < e < z < < < < < <
```

Option Compare Text results in string comparisons based on a case-insensitive text sort order determined by your system's locale. When the same characters are sorted using **Option Compare Text**, the following text sort order is produced:

```
(A=a) < ( =) < (B=b) < (E=e) < (=) < (Z=z) < (=)
```

Option Compare Database can only be used within Microsoft Access. This results in string comparisons based on the sort order determined by the locale ID of the database where the string comparisons occur.

Visual Basic for Applications Reference

Option Compare Statement Example

This example uses the **Option Compare** statement to set the default string comparison method. The **Option Compare** statement is used at the module level only.

```
' Set the string comparison method to Binary.
Option compare Binary    ' That is, "AAA" is less than "aaa".
' Set the string comparison method to Text.
Option compare Text   ' That is, "AAA" is equal to "aaa".
```

Visual Basic for Applications Reference

Visual Studio 6.0

Option Explicit Statement

See Also Example Specifics

Used at module level to force explicit declaration of all variables in that module.

Syntax

Option Explicit

Remarks

If used, the **Option Explicit** statement must appear in a module before any procedures.

When **Option Explicit** appears in a module, you must explicitly declare all variables using the **Dim**, **Private**, **Public**, **ReDim**, or **Static** statements. If you attempt to use an undeclared variable name, an error occurs at compile time.

If you don't use the **Option Explicit** statement, all undeclared variables are of **Variant** type unless the default type is otherwise specified with a **Def***type* statement.

Note Use **Option Explicit** to avoid incorrectly typing the name of an existing variable or to avoid confusion in code where the scope of the variable is not clear.

Visual Basic for Applications Reference

Option Explicit Statement Example

This example uses the **Option Explicit** statement to force explicit declaration of all variables. Attempting to use an undeclared variable causes an error at compile time. The **Option Explicit** statement is used at the module level only.

```
Option explicit   ' Force explicit variable declaration.
Dim MyVar   ' Declare variable.
MyInt = 10   ' Undeclared variable generates error.
MyVar = 10   ' Declared variable does not generate error.
```

Visual Basic for Applications Reference

Visual Studio 6.0

Option Private Statement

See Also Example Specifics

When used in host applications that allow references across multiple projects, **Option Private Module** prevents a modules contents from being referenced outside its project. In host applications that dont permit such references, for example, standalone versions of Visual Basic, **Option Private** has no effect.

Syntax

Option Private Module

Remarks

If used, the **Option Private** statement must appear at module level, before any procedures.

When a module contains **Option Private Module**, the public parts, for example, variables, objects, and user-defined types declared at module level, are still available within the project containing the module, but they are not available to other applications or projects.

Note **Option Private** is only useful for host applications that support simultaneous loading of multiple projects and permit references between the loaded projects. For example, Microsoft Excel permits loading of multiple projects and **Option Private Module** can be used to restrict cross-project visibility. Although Visual Basic permits loading of multiple projects, references between projects are never permitted in Visual Basic.

Visual Basic for Applications Reference

Option Private Statement Example

This example demonstrates the **Option Private** statement, which is used at module level to indicate that the entire module is private. With **Option Private Module**, module-level parts not declared **Private** are available to other modules in the project, but not to other projects or applications.

```
Option private Module    ' Indicates that module is private.
```

Visual Basic for Applications Reference

Visual Studio 6.0

Print # Statement

See Also Example Specifics

Writes display-formatted data to a sequential file.

Syntax

Print #*filenumber*, [*outputlist*]

The **Print #** statement syntax has these parts:

Part	Description
filenumber	Required. Any valid file number.
outputlist	Optional. Expression or list of expressions to print.

Settings

The *outputlist* argument settings are:

[{**Spc(***n***)** | **Tab**[**(***n***)**]}] [*expression*] [*charpos*]

Setting	Description
Spc(*n***)**	Used to insert space characters in the output, where *n* is the number of space characters to insert.
Tab(*n***)**	Used to position the insertion point to an absolute column number, where *n* is the column number. Use **Tab** with no argument to position the insertion point at the beginning of the next print zone.
expression	Numeric expressions or string expressions to print.
charpos	Specifies the insertion point for the next character. Use a semicolon to position the insertion point immediately after the last character displayed. Use **Tab(***n***)** to position the insertion point to an absolute column number. Use **Tab** with no argument to position the insertion point at the beginning of the next print zone. If *charpos* is omitted, the next character is printed on the next line.

Remarks

Data written with **Print #** is usually read from a file with **Line Input #** or **Input**.

If you omit *outputlist* and include only a list separator after *filenumber*, a blank line is printed to the file. Multiple expressions can be separated with either a space or a semicolon. A space has the same effect as a semicolon.

For Boolean data, either `True` or `False` is printed. The **True** and **False** keywords are not translated, regardless of the locale.

Date data is written to the file using the standard short date format recognized by your system. When either the date or the time component is missing or zero, only the part provided gets written to the file.

Nothing is written to the file if *outputlist* data is Empty. However, if *outputlist* data is Null, **Null** is written to the file.

For **Error** data, the output appears as `Error errorcode`. The **Error** keyword is not translated regardless of the locale.

All data written to the file using **Print #** is internationally aware; that is, the data is properly formatted using the appropriate decimal separator.

Because **Print #** writes an image of the data to the file, you must delimit the data so it prints correctly. If you use **Tab** with no arguments to move the print position to the next print zone, **Print #** also writes the spaces between print fields to the file.

Note If, at some future time, you want to read the data from a file using the **Input #** statement, use the **Write #** statement instead of the **Print #** statement to write the data to the file. Using **Write #** ensures the integrity of each separate data field by properly delimiting it, so it can be read back in using **Input #**. Using **Write #** also ensures it can be correctly read in any locale.

Visual Basic for Applications Reference

Print # Statement Example

This example uses the **Print #** statement to write data to a file.

```
Open "TESTFILE" For Output As #1   ' Open file for output.
Print #1, "This is a test"   ' Print text to file.
Print #1,   ' Print blank line to file.
Print #1, "Zone 1"; Tab ; "Zone 2"   ' Print in two print zones.
Print #1, "Hello" ; " " ; "World"   ' Separate strings with space.
Print #1, Spc(5) ; "5 leading spaces "   ' Print five leading spaces.
Print #1, Tab(10) ; "Hello"   ' Print word at column 10.

' Assign Boolean, Date, Null and Error values.
Dim MyBool, MyDate, MyNull, MyError
MyBool = False : MyDate = #February 12, 1969# : MyNull = Null
MyError = CVErr(32767)
' True, False, Null, and Error are translated using locale settings of
' your system. Date literals are written using standard short date
' format.
Print #1, MyBool ; " is a Boolean value"
Print #1, MyDate ; " is a date"
Print #1, MyNull ; " is a null value"
Print #1, MyError ; " is an error value"
Close #1   ' Close file.
```

Visual Basic for Applications Reference

Visual Studio 6.0

Private Statement

See Also Example Specifics

Used at module level to declare private variables and allocate storage space.

Syntax

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

The **Private** 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 being defined.

Remarks

Private variables are available only to the module in which they are declared.

Use the **Private** statement to declare the data type of a variable. For example, the following statement declares a variable as an **Integer**:

```
Private NumberOfEmployees As Integer
```

You can also use a **Private** statement to declare the object type of a variable. The following statement declares a variable for a new instance of a worksheet.

```
Private X As New Worksheet
```

If the **New** keyword isn't 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's 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.

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.

You can also use the **Private** 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.

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 **Private** statement in a procedure, you generally put the **Private** statement at the beginning of the procedure.

Visual Basic for Applications Reference

Private Statement Example

This example shows the **Private** statement being used at the module level to declare variables as private; that is, they are available only to the module in which they are declared.

```
Private Number As Integer   ' Private Integer variable.
Private NameArray(1 To 5) As String   ' Private array variable.
' Multiple declarations, two Variants and one Integer, all Private.
Private MyVar, YourVar, ThisVar As Integer
```

Visual Basic for Applications Reference

Visual Studio 6.0

Property Get Statement

See Also Example Specifics

Declares the name, arguments, and code that form the body of a **Property** procedure, which gets the value of a property.

Syntax

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

End Property

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

Part	Description
Public	Optional. Indicates that the **Property Get** procedure is accessible to all other procedures in all modules. If used in a module that contains an **Option Private** statement, the procedure is not available outside the project.
Private	Optional. Indicates that the **Property Get** 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 **Property Get** procedure is visible throughout the project, but not visible to a controller of an instance of an object.
Static	Optional. Indicates that the **Property Get** procedure's local variables are preserved between calls. The **Static** attribute doesn't affect variables that are declared outside the **Property Get** procedure, even if they are used in the procedure.
name	Required. Name of the **Property Get** procedure; follows standard variable naming conventions, except that the name can be the same as a **Property Let** or **Property Set** procedure in the same module.
arglist	Optional. List of variables representing arguments that are passed to the **Property Get** procedure when it is called. Multiple arguments are separated by commas. The name and data type of each argument in a **Property Get** procedure must be the same as the corresponding argument in a **Property Let** procedure (if one exists).
type	Optional. Data type of the value returned by the **Property Get** procedure; may be Byte, Boolean, Integer, Long, Currency, Single, Double, Decimal (not currently supported), Date, String (except fixed length), Object,

	Variant, user-defined type, and Arrays. The return *type* of a **Property Get** procedure must be the same data type as the last (or sometimes the only) argument in a corresponding **Property Let** procedure (if one exists) that defines the value assigned to the property on the right side of an expression.
statements	Optional. Any group of statements to be executed within the body of the **Property Get** procedure.
expression	Optional. Value of the property returned by the procedure defined by the **Property Get** statement.

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.
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**, **Property** procedures are public by default. If **Static** is not 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.

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

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

Like a **Sub** and **Property Let** procedure, a **Property Get** 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** or **Property Let** procedure, you can

use a **Property Get** procedure on the right side of an expression in the same way you use a **Function** or a property name when you want to return the value of a property.

Visual Basic for Applications Reference

Property Get Statement Example

This example uses the **Property Get** statement to define a property procedure that gets the value of a property. The property identifies the current color of a pen as a string.

```
Dim CurrentColor As Integer
Const BLACK = 0, RED = 1, GREEN = 2, BLUE = 3

' Returns the current color of the pen as a string.
Property Get PenColor() As String
   Select Case CurrentColor
      Case RED
         PenColor = "Red"
      Case GREEN
         PenColor = "Green"
      Case BLUE
         PenColor = "Blue"
   End Select
End Property

' The following code gets the color of the pen
' calling the Property Get procedure.
ColorName = PenColor
```

Visual Basic for Applications Reference

Visual Studio 6.0

Property Let Statement

See Also Example Specifics

Declares the name, arguments, and code that form the body of a **Property Let** procedure, which assigns a value to a property.

Syntax

[**Public** | **Private** | **Friend**] [**Static**] **Property Let** *name* **(**[*arglist***,**] *value***)**
[*statements*]
[**Exit Property**]
[*statements*]

End Property

The **Property Let** statement syntax has these parts:

Part	Description
Public	Optional. Indicates that the **Property Let** procedure is accessible to all other procedures in all modules. If used in a module that contains an **Option Private** statement, the procedure is not available outside the project.
Private	Optional. Indicates that the **Property Let** 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 **Property Let** procedure is visible throughout the project, but not visible to a controller of an instance of an object.
Static	Optional. Indicates that the **Property Let** procedure's local variables are preserved between calls. The **Static** attribute doesn't affect variables that are declared outside the **Property Let** procedure, even if they are used in the procedure.
name	Required. Name of the **Property Let** procedure; follows standard variable naming conventions, except that the name can be the same as a **Property Get** or **Property Set** procedure in the same module.
arglist	Required. List of variables representing arguments that are passed to the **Property Let** procedure when it is called. Multiple arguments are separated by commas. The name and data type of each argument in a **Property Let** procedure must be the same as the corresponding argument in a **Property Get** procedure.
value	Required. Variable to contain the value to be assigned to the property. When the procedure is called, this argument appears on the right side of the calling expression. The data type of *value* must be the same as the return type of the corresponding **Property Get** procedure.

statements	Optional. Any group of statements to be executed within the **Property Let** procedure.

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. Note that it is not possible for the right side of a **Property Let** expression to be **Optional**.
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**.

Note Every **Property Let** statement must define at least one argument for the procedure it defines. That argument (or the last argument if there is more than one) contains the actual value to be assigned to the property when the procedure defined by the **Property Let** statement is invoked. That argument is referred to as *value* in the preceding syntax.

Remarks

If not explicitly specified using **Public**, **Private**, or **Friend**, **Property** 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.

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

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

Like a **Function** and **Property Get** procedure, a **Property Let** procedure is a separate procedure that can take arguments, perform a series of statements, and change the value of its arguments. However, unlike a **Function** and **Property Get** procedure, both of which return a value, you can only use a **Property Let** procedure on the left side of a property assignment expression or **Let** statement.

Visual Basic for Applications Reference

Property Let Statement Example

This example uses the **Property Let** statement to define a procedure that assigns a value to a property. The property identifies the pen color for a drawing package.

```
Dim CurrentColor As Integer
Const BLACK = 0, RED = 1, GREEN = 2, BLUE = 3

' Set the pen color property for a Drawing package.
' The module-level variable CurrentColor is set to
' a numeric value that identifies the color used for drawing.
Property Let PenColor(ColorName As String)
   Select Case ColorName   ' Check color name string.
      Case "Red"
         CurrentColor = RED   ' Assign value for Red.
      Case "Green"
         CurrentColor = GREEN   ' Assign value for Green.
      Case "Blue"
         CurrentColor = BLUE   ' Assign value for Blue.
      Case Else
         CurrentColor = BLACK   ' Assign default value.
   End Select
End Property

' The following code sets the PenColor property for a drawing package
' by calling the Property let procedure.

PenColor = "Red"
```

Visual Basic for Applications Reference

Visual Studio 6.0

Property Set Statement

See Also Example Specifics

Declares the name, arguments, and code that form the body of a **Property** procedure, which sets a reference to an object.

Syntax

[**Public** | **Private** | **Friend**] [**Static**] **Property Set** *name* **(**[*arglist***,**] *reference***)**
[*statements*]
[**Exit Property**]
[*statements*]

End Property

The **Property Set** statement syntax has these parts:

Part	Description
Optional	Optional. Indicates that the argument may or may not be supplied by the caller.
Public	Optional. Indicates that the **Property Set** procedure is accessible to all other procedures in all modules. If used in a module that contains an **Option Private** statement, the procedure is not available outside the project.
Private	Optional. Indicates that the **Property Set** 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 **Property Set** procedure is visible throughout the project, but not visible to a controller of an instance of an object.
Static	Optional. Indicates that the **Property Set** procedure's local variables are preserved between calls. The **Static** attribute doesn't affect variables that are declared outside the **Property Set** procedure, even if they are used in the procedure.
name	Required. Name of the **Property Set** procedure; follows standard variable naming conventions, except that the name can be the same as a **Property Get** or **Property Let** procedure in the same module.
arglist	Required. List of variables representing arguments that are passed to the **Property Set** procedure when it is called. Multiple arguments are separated by commas.
reference	Required. Variable containing the object reference used on the right side of the object reference assignment.
statements	Optional. Any group of statements to be executed within the body of the **Property** procedure.

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. Note that it is not possible for the right side of a **Property Set** expression to be **Optional**.
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**.

Note Every **Property Set** statement must define at least one argument for the procedure it defines. That argument (or the last argument if there is more than one) contains the actual object reference for the property when the procedure defined by the **Property Set** statement is invoked. It is referred to as *reference* in the preceding syntax. It can't be **Optional**.

Remarks

If not explicitly specified using **Public**, **Private**, or **Friend**, **Property** 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.

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

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

Like a **Function** and **Property Get** procedure, a **Property Set** procedure is a separate procedure that can take arguments, perform a series of statements, and change the value of its arguments. However, unlike a **Function** and **Property Get** procedure, both of which return a value, you can only use a **Property Set** procedure on the left side of an object reference assignment (**Set** statement).

Visual Basic for Applications Reference

Property Set Statement Example

This example uses the **Property Set** statement to define a property procedure that sets a reference to an object.

```
' The Pen property may be set to different Pen implementations.
Property Set Pen(P As Object)
   Set CurrentPen = P   ' Assign Pen to object.
End Property
```

Visual Basic for Applications Reference

Visual Studio 6.0

Public Statement

See Also Example Specifics

Used at module level to declare public variables and allocate storage space.

Syntax

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

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

Part	Description
WithEvents	Optional. Keyword specifying 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 can't 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 being defined.

Remarks

Variables declared using the **Public** statement are available to all procedures in all modules in all applications unless **Option Private Module** is in effect; in which case, the variables are public only within the project in which they reside.

Caution The **Public** statement can't be used in a class module to declare a fixed-length string variable.

Use the **Public** statement to declare the data type of a variable. For example, the following statement declares a variable as an **Integer**:

```
Public NumberOfEmployees As Integer
```

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

```
Public 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 **Public** 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.

Visual Basic for Applications Reference

Public Statement Example

This example uses the **Public** statement at the module level (General section) of a standard module to explicitly declare variables as public; that is, they are available to all procedures in all modules in all applications unless **Option Private Module** is in effect.

```
Public Number As Integer   ' Public Integer variable.
Public NameArray(1 To 5) As String   ' Public array variable.
' Multiple declarations, two Variants and one Integer, all Public.
Public MyVar, YourVar, ThisVar As Integer
```

Visual Basic for Applications Reference

Visual Studio 6.0

Put Statement

See Also Example Specifics

Writes data from a variable to a disk file.

Syntax

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

The **Put** 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 writing begins.
varname	Required. Name of variable containing data to be written to disk.

Remarks

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

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 after the last **Get** or **Put** statement or pointed to by the last **Seek** function is written. You must include delimiting commas, for example:

```
Put #4,,FileBuffer
```

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

- If the length of the data being written is less than the length specified in the **Len** clause of the **Open** statement, **Put** writes 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 written. If the length of the data being written is greater than the length specified in the **Len** clause of the **Open** statement, an error occurs.

- If the variable being written is a variable-length string, **Put** writes a 2-byte descriptor containing the string length and then the variable. 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 written is a Variant of a numeric type, **Put** writes 2 bytes identifying the **VarType** of the **Variant** and then writes the variable. For example, when writing a **Variant** of **VarType** 3, **Put** writes 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 **Put** statement to write a **Variant** array to disk, but you can't use **Put** to write a scalar **Variant** containing an array to disk. You also can't use **Put** to write objects to disk.

- If the variable being written is a **Variant** of **VarType** 8 (**String**), **Put** writes 2 bytes identifying the **VarType**, 2 bytes indicating the length of the string, and then writes 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 written is a dynamic array, **Put** writes 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 write 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 written is a fixed-size array, **Put** writes only the data. No descriptor is written to disk.

- If the variable being written is any other type of variable (not a variable-length string or a **Variant**), **Put** writes 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 written.

- **Put** writes elements of user-defined types as if each were written individually, except 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 write 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. **Put** writes all variables to disk contiguously; that is, with no padding between records.

- For any array other than an array in a user-defined type, **Put** writes only the data. No descriptor is written.

- **Put** writes variable-length strings that are not elements of user-defined types without the 2-byte length descriptor. The number of bytes written equals the number of characters in the string. For example, the following statements write 10 bytes to file number 1:

  ```
  VarString$ = String$(10," ")
  Put #1,,VarString$
  ```

Visual Basic for Applications Reference

Put Statement Example

This example uses the **Put** statement to write data to a file. Five records of the user-defined type Record are written to the file.

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

Dim MyRecord As Record, RecordNumber   ' Declare variables.
' Open file for random access.
Open "TESTFILE" For Random As #1 Len = Len(MyRecord)
For RecordNumber = 1 To 5   ' Loop 5 times.
   MyRecord.ID = RecordNumber   ' Define ID.
   MyRecord.Name = "My Name" & RecordNumber   ' Create a string.
   Put #1, RecordNumber, MyRecord   ' Write record to file.
Next RecordNumber
Close #1   ' Close file.
```