Feature Information

VBScript Features
VBA Features not in VBScript
VBScript Features not in VBA
Microsoft Scripting Run-Time Features
Alphabetic Keyword List

Abs Function
Addition Operator (+)
And Operator
Array Function
Asc Function
Assignment Operator (=)
Atn Function
Call Statement
CBool Function
CByte Function
CCur Function
CDate Function
CDbl Function
Chr Function
CInt Function
Class Object
Class Statement
Clear Method
CLng Function
Color Constants
Comparison Constants
Concatenation Operator (&)
Const Statement
Cos Function
CreateObject Function
CSng Function
CStr Function
Date and Time Constants
Date Format Constants
Date Function
DateAdd Function
DateDiff Function
DatePart Function
DateSerial Function
DateValue Function
Day Function
Description Property
Dictionary Object
Dim Statement
Division Operator (/)
Do...Loop Statement
Empty
Eqv Operator
Erase Statement
Err Object
Eval Function
Execute Method
Execute Statement
Exit Statement
Exp Function
Exponentiation Operator (^)
False
FileSystemObject Object
Filter Function
FirstIndex Property
Fix Function
For...Next Statement
For Each...Next Statement
FormatCurrency Function
FormatDateTime Function
FormatNumber Function
FormatPercent Function
Function Statement
GetObject Function
GetRef Function
Global Property
Hex Function
HelpContext Property
HelpFile Property
Hour Function
If...Then...Else Statement
IgnoreCase Property
Imp Operator
Initialize Event
InputBox Function
InStr Function
InStrRev Function
Int Function
Integer Division Operator (/)
Is Operator
IsArray Function
IsDate Function
IsEmpty Function
IsNull Function
IsNumeric Function
IsObject Function
Join Function
LBound Function
LCase Function
Left Function
Len Function
Length Property
LoadPicture Function
Log Function
LTrim Function
Match Object
Matches Collection
Mid Function
Minute Function
Miscellaneous Constants
Mod Operator
Month Function
MonthName Function
MsgBox Constants
MsgBox Function
Multiplication Operator (*)
Negation Operator (-)
Not Operator
Now Function
Nothing
Null
Number Property
Oct Function
On Error Statement
Operator Precedence
Option Explicit Statement
Or Operator
Pattern Property
Private Statement
PropertyGet Statement
PropertyLet Statement
PropertySet Statement
Public Statement
Raise Method
Randomize Statement
ReDim Statement
RegExp Object
Rem Statement
Replace Function
Replace Method
RGB Function
Right Function
Rnd Function
Round Function
RTrim Function
ScriptEngine Function
ScriptEngineBuildVersion Function
ScriptEngineMajorVersion Function
ScriptEngineMinorVersion Function
Second Function
Select Case Statement
Set Statement
Sgn Function
Sin Function
Source Property
Space Function
Split Function
Sqr Function
StrComp Function
String Constants
String Function
StrReverse Function
Sub Statement
Subtraction Operator (-)
Tan Function
Terminate Event
Test Method
Time Function
Timer Function
TimeSerial Function
TimeValue Function
Trim Function
Tristate Constants
True
TypeName Function
UBound Function
UCase Function
Value Property
VarType Constants
VarType Function
VBScript Constants
Weekday Function
WeekdayName Function
While...Wend Statement
With Statement
Xor Operator
Year Function
Constants

Color Constants
Comparison Constants
Date and Time Constants
Date Format Constants
Miscellaneous Constants
MsgBox Constants
String Constants
Tristate Constants
VarType Constants
VBScript Constants
VBScript Errors

Run-time Errors
Syntax Errors
Events

Initialize Event
Terminate Event
Abs Function
Array Function
Asc Function
Atn Function
CBool Function
CByte Function
CCur Function
CDate Function
CDbl Function
Chr Function
CInt Function
CLng Function
Cos Function
CreateObject Function
CSng Function
CStr Function
Date Function
DateAdd Function
DateDiff Function
DatePart Function
DateSerial Function
DateValue Function
Day Function
Eval Function
Exp Function
Filter Function
Fix Function
FormatCurrency Function
FormatDateTime Function
FormatNumber Function
FormatPercent Function
GetObject Function
GetRef Function
Hex Function
Hour Function
InputBox Function
InStr Function
InStrRev Function
Int Function
IsArray Function
IsDate Function
IsEmpty Function
IsNull Function
IsNumeric Function
IsObject Function
Join Function
LBound Function
LCase Function
Left Function
Len Function
LoadPicture Function
Log Function
LTrim Function
Mid Function
Minute Function
Month Function
MonthName Function
MsgBox Function
Now Function
Oct Function
Replace Function
RGB Function
Right Function
Rnd Function
Round Function
RTrim Function
ScriptEngine Function
ScriptEngineBuildVersion Function
ScriptEngineMajorVersion Function
ScriptEngineMinorVersion Function
Second Function
Sgn Function
Sin Function
Space Function
Split Function
Sqr Function
StrComp Function
String Function
StrReverse Function
Tan Function
Time Function
Timer Function
TimeSerial Function
TimeValue Function
Trim Function
TypeName Function
UBound Function
UCase Function
VarType Function
Weekday Function
WeekdayName Function
Year Function
Methods

Clear Method
Execute Method
Raise Method
Replace Method
Test Method
Objects

Class Object
Dictionary Object
Err Object
FileSystemObject Object
Match Object
Matches Collection
RegExp Object
Operators

Addition Operator (+)
And Operator
Assignment Operator (=)
Concatenation Operator (&)
Division Operator (/)
Eqv Operator
Exponentiation Operator (^)
Imp Operator
Integer Division Operator (\)
Is Operator
Mod Operator
Multiplication Operator (*)
Negation Operator (-)
Not Operator
Operator Precedence
Or Operator
Subtraction Operator (-)
Xor Operator
Properties

Description Property
FirstIndex Property
Global Property
HelpContext Property
HelpFile Property
IgnoreCase Property
Length Property
Number Property
Pattern Property
Source Property
Value Property
Call Statement
Class Statement
Const Statement
Dim Statement
Do...Loop Statement
Erase Statement
Execute Statement
Exit Statement
For...Next Statement
For Each...Next Statement
Function Statement
If...Then...Else Statement
On Error Statement
Option Explicit Statement
Private Statement
Property Get Statement
Property Let Statement
Property Set Statement
Public Statement
Randomize Statement
ReDim Statement
Rem Statement
Select Case Statement
Set Statement
Sub Statement
While...Wend Statement
With Statement
Controlling Program Execution

You can control the flow of your script with conditional statements and looping statements. Using conditional statements, you can write VBScript code that makes decisions and repeats actions. The following conditional statements are available in VBScript:

- **If...Then...Else** statement
- **Select Case** statement

Making Decisions Using If...Then...Else

The **If...Then...Else** statement is used to evaluate whether a condition is **True** or **False** and, depending on the result, to specify one or more statements to run. Usually the condition is an expression that uses a comparison operator to compare one value or variable with another. For information about comparison operators, see [Comparison Operators](#). **If...Then...Else** statements can be nested to as many levels as you need.

Running Statements if a Condition is True

To run only one statement when a condition is **True**, use the single-line syntax for the **If...Then...Else** statement. The following example shows the single-line syntax. Notice that this example omits the **Else** keyword.

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

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

```vbs
Sub AlertUser(value)
  If value = 0 Then
    AlertLabel.ForeColor = vbRed
    AlertLabel.Font.Bold = True
    AlertLabel.Font.Italic = True
  End If
End Sub
```

*Running Certain Statements if a Condition is True and Running Others if a Condition is False*

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

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

Deciding Between Several Alternatives

A variation on the if...then...else statement allows you to choose from several alternatives. Adding elseif clauses expands the functionality of the if...then...else statement so you can control program flow based on different possibilities. For example:

Sub ReportValue(value)
    If value = 0 Then
        MsgBox value
    ElseIf value = 1 Then
        MsgBox value
    ElseIf value = 2 then
        MsgBox value
    Else
        MsgBox "Value out of range!"
    End If
End Sub

You can add as many elseif clauses as you need to provide alternative choices. Extensive use of the elseif clauses often becomes cumbersome. A better way to choose between several alternatives is the select case statement.
Making Decisions with Select Case

The **Select Case** structure provides an alternative to **If...Then...ElseIf** for selectively executing one block of statements from among multiple blocks of statements. A **Select Case** statement provides capability similar to the **If...Then...Else statement**, but it makes code more efficient and readable.

A **Select Case** structure works with a single test expression that is evaluated once, at the top of the structure. The result of the expression is then compared with the values for each **Case** in the structure. If there is a match, the block of statements associated with that **Case** is executed:

```vbnet
Select Case Document.Form1.CardType.Options(SelectedIndex).Text
    Case "MasterCard"
        DisplayMCLogo
        ValidateMCAccount
    Case "Visa"
        DisplayVisaLogo
        ValidateVisaAccount
    Case "American Express"
        DisplayAMEXCOLogo
        ValidateAMEXCOAccount
    Case Else
        DisplayUnknownImage
        PromptAgain
End Select
```

Notice that the **Select Case** structure evaluates an expression once at the top of the structure. In contrast, the **If...Then...ElseIf** structure can evaluate a different expression for each **Elseif** statement. You can replace an
If...Then...ElseIf structure with a Select Case structure only if each ElseIf statement evaluates the same expression.
Using Loops to Repeat Code

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

The following looping statements are available in VBScript:

- **Do...Loop**: Loops while or until a condition is True.
- **While...Wend**: Loops while a condition is True.
- **For...Next**: Uses a counter to run statements a specified number of times.
- **For Each...Next**: Repeats a group of statements for each item in a collection or each element of an array.

Using Do Loops

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

Repeating Statements While a Condition is True

Use the **While** keyword to check a condition in a **Do...Loop** statement. You can check the condition before you enter the loop (as shown in the following ChkFirstWhile example), or you can check it after the loop has run at least once (as shown in the ChkLastWhile example). In the ChkFirstWhile procedure, if **myNum** is set to 9 instead of 20, the statements inside the loop will never run. In the ChkLastWhile procedure, the
statements inside the loop run only once because the condition is already False.

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

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

Repeating a Statement Until a Condition Becomes True

You can use the Until keyword in two ways to check a condition in a Do...Loop statement. You can check the condition before you enter the loop (as shown in the following ChkFirstUntil example), or you can check it after the loop has run at least once (as shown in the ChkLastUntil example). As long as the condition is False, the looping occurs.

Sub ChkFirstUntil()
    Dim counter, myNum
counter = 0
myNum = 20
Do Until myNum = 10
    myNum = myNum - 1
    counter = counter + 1
Loop
MsgBox "The loop made " & counter & " repetitions."
End Sub

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

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

You can exit a Do...Loop by using the Exit Do statement. Because you usually want to exit only in certain situations, such as to avoid an endless loop, you should use the Exit Do statement in the True statement block of an If...Then...Else statement. If the condition is False, the loop runs as usual.

In the following example, myNum is assigned a value that creates an endless loop. The If...Then...Else statement checks for this condition, preventing the endless repetition.

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

Using While...Wend

The While...Wend statement is provided in VBScript for those who are familiar with its usage. However, because of the lack of flexibility in While...Wend, it is recommended that you use Do...Loop instead.

Using For...Next

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

For example, the following procedure causes a procedure called MyProc to execute 50 times. The For statement specifies the counter variable x and its start and end values. The Next statement increments the counter variable by 1.

Sub DoMyProc50Times()
   Dim x
   For x = 1 To 50
      MyProc
   Next
End Sub

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

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

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

\begin{verbatim}
Sub NewTotal()
    Dim myNum, total
    For myNum = 16 To 2 Step -2
        total = total + myNum
    Next
    MsgBox "The total is " & total
End Sub
\end{verbatim}

You can exit any \textbf{For...Next} statement before the counter reaches its end value by using the \textbf{Exit For} statement. Because you usually want to exit only in certain situations, such as when an error occurs, you should use the \textbf{Exit For} statement in the \textbf{True} statement block of an \textbf{If...Then...Else} statement. If the condition is \textbf{False}, the loop runs as usual.

\textbf{Using For Each...Next}

A \textbf{For Each...Next} loop is similar to a \textbf{For...Next} loop. Instead of repeating the statements a specified number of times, a \textbf{For Each...Next} loop repeats a group of statements for each item in a collection of objects or for each element of an array. This is especially helpful if you don't know how many elements are in a collection.

In the following HTML code example, the contents of a \textbf{Dictionary} object is used to place text in several text boxes:
<HTML>
<HEAD><TITLE>Forms and Elements</TITLE></HEAD>
<SCRIPT LANGUAGE="VBScript">
<!--
Sub cmdChange_OnClick
    Dim d 'Create a variable
    Set d = CreateObject("Scripting.Dictionary")
    d.Add "0", "Athens" 'Add some keys and items
    d.Add "1", "Belgrade"
    d.Add "2", "Cairo"

    For Each I in d
        Document.frmForm.Elements(I).Value = D.Item(I)
    Next
End Sub
--></SCRIPT>
<BODY>
<CENTER>
<FORM NAME="frmForm"

    <Input Type = "Text"><p>
    <Input Type = "Text"><p>
    <Input Type = "Text"><p>
    <Input Type = "Text"><p>
    <Input Type = "Text"><p>
    <Input Type = "Button" NAME="cmdChange" VALUE="Click Here"
</FORM>
</CENTER>
</BODY>
</HTML>
Simple Validation

You can use Visual Basic Scripting Edition to do much of the form processing that you'd usually have to do on a server. You can also do things that just can't be done on the server.

Here's an example of simple client-side validation. The HTML code is for a text box and a button. If you use Microsoft® Internet Explorer to view the page produced by the following code, you'll see a small text box with a button next to it.

```
<HTML>
<HEAD><TITLE>Simple Validation</TITLE>
<SCRIPT LANGUAGE="VBScript">
<!--
Sub Button1_OnClick
   Dim TheForm
   Set TheForm = Document.ValidForm
   If IsNumeric(TheForm.Text1.Value) Then
      If TheForm.Text1.Value < 1 Or TheForm.Text1.Value > 10 Then
         MsgBox "Please enter a number between 1 and 10."
      Else
         MsgBox "Thank you."
      End If
   Else
      MsgBox "Please enter a valid number."
   End If
End Sub
-->
</SCRIPT>
</HEAD>
<body>
<form>

   <input type="text" name="Text1" value="">
   <input type="button" name="Button1" value="Submit" onClick="Button1_OnClick()">

</form>
</body>
</HTML>
```
MsgBox "Please enter a numeric value."
End If
End Sub
-->
</SCRIPT>
</HEAD>
<BODY>
<H3>Simple Validation</H3><HR>
<FORM NAME="ValidForm">
Enter a value between 1 and 10:
<INPUT NAME="Text1" TYPE="TEXT" SIZE="
<INPUT NAME="Button1" TYPE="BUTTON" V.
</FORM>
</BODY>
</HTML>

The difference between this text box and the examples on A Simple VBScript Page is that the Value property of the text box is used to check the entered value. To get the Value property, the code has to qualify the reference to the name of the text box.

You can always write out the full reference Document.ValidForm.Text1. However, where you have multiple references to form controls, you'll want to do what was done here. First declare a variable. Then use the Set statement to assign the form to the variable TheForm. A regular assignment statement, such as Dim, doesn't work here; you must use Set to preserve the reference to an object.

**Using Numeric Values**

Notice that the example directly tests the value against a number: it uses the IsNumeric function to make sure the string in the text box is a number. Although VBScript automatically converts strings and numbers, it's always
a good practice to test a user-entered value for its data subtype and to use conversion functions as necessary. When doing addition with text box values, convert the values explicitly to numbers because the plus sign (+) operator represents both addition and string concatenation. For example, if Text1 contains "1" and Text2 contains "2", you see the following results:

\[
\begin{align*}
A &= \text{Text1.Value} + \text{Text2.Value} \quad \text{'}\ A \ is \ "12" \\
A &= \text{CDbl(Text1.Value)} + \text{Text2.Value} \quad \text{'}\ A \ is \ 3
\end{align*}
\]

Validating and Passing Data Back to the Server

The simple validation example uses a plain button control. If a Submit control was used, the example would never see the data to check it—everything would go immediately to the server. Avoiding the Submit control lets you check the data, but it doesn't submit the data to the server. That requires an additional line of code:

```vbscript
<SCRIPT LANGUAGE="VBScript">
<!--
Sub Button1_OnClick
Dim TheForm
Set TheForm = Document.ValidForm
If IsNumeric(TheForm.Text1.Value) Then
    If TheForm.Text1.Value < 1 Or TheForm.Text1.Value > 10 Then
        MsgBox "Please enter a number between 1 and 10."
    Else
        MsgBox "Thank you."
    TheForm.Submit \quad \text{'}\ Data correct; send to server.
    End If
Else
    MsgBox "Please enter a numeric value."
End If
End Sub
-->
```
To send the data to the server, the code invokes the **Submit** method on the form object when the data is correct. From there, the server handles the data just as it otherwise would—except that the data is correct before it gets there. Find complete information about the **Submit** method and other methods in the Internet Explorer Scripting Object Model documentation, which can be found on the Microsoft® Web site (http://www.microsoft.com).

So far, you've seen only the standard HTML `<FORM>` objects. Internet Explorer also lets you exploit the full power of ActiveX® controls (formerly called OLE controls) and Java™ objects.
Using VBScript with Objects

Whether you use an ActiveX® control (formerly called an OLE control) or a Java™ object, Microsoft Visual Basic Scripting Edition and Microsoft® Internet Explorer handle it the same way. If you're using Internet Explorer and have installed the Label control, you can see the page produced by the following code.

You include an object using the <OBJECT> tags and set its initial property values using <PARAM> tags. If you're a Visual Basic programmer, you'll recognize that using the <PARAM> tags is just like setting initial properties for a control on a form. For example, the following set of <OBJECT> and <PARAM> tags adds the ActiveX Label control to a page:

```html
<OBJECT
    classid="clsid:99B42120-6EC7-11CF-A6C7-00AA00A47DD2"
    id=lblActiveLbl
    width=250
    height=250
    align=left
    hspace=20
    vspace=0
>
<PARAM NAME="Angle" VALUE="90">
```
You can get properties, set properties, and invoke methods just as with any of the form controls. The following code, for example, includes <FORM> controls you can use to manipulate two properties of the Label control:

```xml
<FORM NAME="LabelControls">
  <INPUT TYPE="TEXT" NAME="txtNewText" SIZE=25>
  <INPUT TYPE="BUTTON" NAME="cmdChangeIt" VALUE="Change Text">
  <INPUT TYPE="BUTTON" NAME="cmdRotate">
</FORM>
```

With the form defined, an event procedure for the cmdChangeIt button changes the label text:

```vbs
<SCRIPT LANGUAGE="VBScript">
  <!--
  Sub cmdChangeIt_onClick
      Dim TheForm
      Set TheForm = Document.LabelControls
      lblActiveLbl.Caption = TheForm.txtNewText.Value
  -->
</SCRIPT>
```
The code qualifies references to controls and values inside the forms just as in the Simple Validation example.

Several ActiveX controls are available for use with Internet Explorer. You can find complete information about the properties, methods, and events there, as well as the class identifiers (CLSID) for the controls on the Microsoft® Web site (http://www.microsoft.com). You can find more information about the <OBJECT> tag on the Internet Explorer 4.0 Author’s Guide and HTML Reference page.

Note Earlier releases of Internet Explorer required braces ({}), around the classid attribute and did not conform to the W3C specification. Using braces with the current release generates a "This page uses an outdated version of the <OBJECT> tag" message.
Welcome to the Scripting Run-Time Library Reference

These handy blocks of information will help you explore the many different parts of the Scripting Run-Time Library.

You'll find *all* the parts of the Scripting Run-Time Library listed alphabetically under the Alphabetic Keyword List. But if you want to examine just one category, say, objects, each language category has its own, more compact section.

How's it work? Click on one of the headings to the left to display a list of items contained in that category. From this list, select the topic that you want to view. Once you've opened that topic, you can easily link to other related sections.
So, go ahead and take a look! Study some statements, mull over the methods, or figure out a few functions. You'll see just how versatile the Scripting Run-Time Library can be!
Feature Information

Microsoft Scripting Run-Time Features
Alphabetic Keyword List

Add Method (Dictionary)
Add Method (Folders)
AtEndOfLine Property
AtEndOfStream Property
Attributes Property
AvailableSpace Property
BuildPath Method
Close Method
Column Property
CompareMode Property
Copy Method
CopyFile Method
CopyFolder Method
Count Property
CreateFolder Method
CreateTextFile Method
DateCreated Property
DateLastAccessed Property
DateLastModified Property
Delete Method
DeleteFile Method
DeleteFolder Method
Dictionary Object
Drive Object
Drive Property
Drive Type Constants
DriveExists Method
DriveLetter Property
Drives Collection
Drives Property
DriveType Property
Exists Method
FileExists Method
File Attribute Constants
File Input/Output Constants
File Object
Files Collection
Files Property
FileSystemObject Constants
FileSystemObject Object
FileSystem Property
Folder Object
Folders Collection
FolderExists Method
FreeSpace Property
GetAbsolutePathName Method
GetBaseName Method
GetDrive Method
GetDriveName Method
GetExtensionName Method
GetFile Method
GetFileName Method
GetFileVersion Method
GetFolder Method
GetParentFolderName Method
GetSpecialFolder Method
GetTempName Method
IsReady Property
IsRootFolder Property
Item Property
Items Method
Key Property
Keys Method
Line Property
Move Method
MoveFile Method
MoveFolder Method
Name Property
OpenAsTextStream Method
OpenTextFile Method
ParentFolder Property
Path Property
Read Method
ReadAll Method
ReadLine Method
Remove Method
RemoveAll Method
RootFolder Property
SerialNumber Property
ShareName Property
ShortName Property
ShortPath Property
Size Property
Skip Method
SkipLine Method
SpecialFolder Constants
Subfolders Property
TextStream Object
TotalSize Property
Type Property
VolumeName Property
Write Method
WriteBlankLines Method
WriteLine Method
Constants

DriveType Constants
File Attribute Constants
File Input/Output Constants
FileSystemObject Constants
SpecialFolder Constants
Add Method (Dictionary)
Add Method (Folders)
BuildPath Method
Close Method
Copy Method
CopyFile Method
CopyFolder Method
CreateFolder Method
CreateTextFile Method
Delete Method
DeleteFile Method
DeleteFolder Method
DriveExists Method
Exists Method
FileExists Method
FolderExists Method
GetAbsolutePathName Method
GetBaseName Method
GetDrive Method
GetDriveName Method
GetExtensionName Method
GetFile Method
GetFileName Method
GetFileVersion Method
GetFolder Method
GetParentFolderName Method
GetSpecialFolder Method
GetTempName Method
Items Method
Keys Method
Move Method
MoveFile Method
MoveFolder Method
OpenAsTextStream Method
OpenTextFile Method
Read Method
ReadAll Method
ReadLine Method
Remove Method
RemoveAll Method
Skip Method
SkipLine Method
Write Method
WriteBlankLines Method
WriteLine Method
Objects

Dictionary Object
Drive Object
Drives Collection
File Object
Files Collection
FileSystemObject Object
Folder Object
Folders Collection
TextStream Object
AtEndOfLine Property
AtEndOfStream Property
Attributes Property
AvailableSpace Property
Column Property
CompareMode Property
Count Property
DateCreated Property
DateLastAccessed Property
DateLastModified Property
Drive Property
DriveLetter Property
Drives Property
DriveType Property
Files Property
FileSystem Property
FreeSpace Property
IsReady Property
IsRootFolder Property
Item Property
Key Property
Line Property
Name Property
ParentFolder Property
Path Property
RootFolder Property
SerialNumber Property
ShareName Property
ShortName Property
ShortPath Property
Size Property
SubFolders Property
TotalSize Property
Type Property
VolumeName Property
The FileSystemObject Object Model

When writing scripts for Active Server Pages, the Windows Scripting Host, or other applications where scripting can be used, it's often important to add, move, change, create, or delete folders (directories) and files on the Web server. It may also be necessary to get information about and manipulate drives attached to the Web server.

Scripting allows you to process drives, folders, and files using the FileSystemObject (FSO) object model, which is explained in the following sections:

- [Introduction to the FileSystemObject and the Scripting Run-Time Library Reference](#)
- [FileSystemObject Objects](#)
- [Programming the FileSystemObject](#)
- [Working with Drives and Folders](#)
- [Working with Files](#)
- [FileSystemObject Sample Code](#)
The **FileSystemObject** (FSO) object model allows you to use the familiar `object.method` syntax with a rich set of properties, methods, and events to process folders and files.

Use this object-based tool with:

- HTML to create Web pages
- Windows Scripting Host to create batch files for Microsoft Windows
- Script Control to provide a scripting capability to applications developed in other languages

Because use of the FSO on the client side raises serious security issues about providing potentially unwelcome access to a client's local file system, this documentation assumes use of the FSO object model to create scripts executed by Internet Web pages on the server side. Since the server side is used, the Internet Explorer default security settings do not allow client-side use of the **FileSystemObject** object. Overriding those defaults could subject a local computer to unwelcome access to the file system, which could result in total destruction of the file system's integrity, causing loss of data, or worse.

The FSO object model gives your server-side applications the ability to create, alter, move, and delete folders, or to detect if particular folders exist, and if so, where. You can also find out information about
folders, such as their names, the date they were created or last modified, and so forth.

The FSO object model also makes it easy to process files. When processing files, the primary goal is to store data in a space- and resource-efficient, easy-to-access format. You need to be able to create files, insert and change the data, and output (read) the data. Since storing data in a database, such as Access or SQL Server, adds a significant amount of overhead to your application, storing your data in a binary or text file may be the most efficient solution. You may prefer not to have this overhead, or your data access requirements may not require all the extra features associated with a full-featured database.

The FSO object model, which is contained in the Scripting type library (Scrrun.dll), supports text file creation and manipulation through the TextStream object. Although it does not yet support the creation or manipulation of binary files, future support of binary files is planned.
The **FileSystemObject** (FSO) object model contains the following objects and **collections**.

<table>
<thead>
<tr>
<th>Object/Collection</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Drive</strong></td>
<td>Object. Contains methods and properties that allow you to gather information about a drive attached to the system, such as its share name and how much room is available. Note that a &quot;drive&quot; isn't necessarily a hard disk, but can be a CD-ROM drive, a RAM disk, and so forth. A drive doesn't need to be physically attached to the system; it can be also be logically connected through a network.</td>
</tr>
<tr>
<td><strong>Drives</strong></td>
<td>Collection. Provides a list of the drives attached to the system, either physically or logically. The <strong>Drives</strong> collection includes all drives, regardless of type. Removable-media drives need not have media inserted for them to appear in this collection.</td>
</tr>
<tr>
<td><strong>File</strong></td>
<td>Object. Contains methods and properties that allow you to create, delete, or move a file.</td>
</tr>
<tr>
<td><strong>FileSystemObject</strong></td>
<td>Main object. Contains methods and properties that allow you to create, delete, gain information about, and generally manipulate drives, folders, and files. Many of the methods associated with this object duplicate those in other FSO objects; they are provided for convenience.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>----------</td>
<td>-----------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>Files</strong></td>
<td>Also allows you to query the system for a file name, path, and various other properties.</td>
</tr>
<tr>
<td><strong>Folder</strong></td>
<td>Collection. Provides a list of all files contained within a folder. Also allows you to query the system for folder names, paths, and various other properties.</td>
</tr>
<tr>
<td><strong>Folders</strong></td>
<td>Collection. Provides a list of all the folders within a <strong>Folder</strong>.</td>
</tr>
<tr>
<td><strong>TextStream</strong></td>
<td>Object. Allows you to read and write text files.</td>
</tr>
</tbody>
</table>
Programming the FileSystemObject

To program with the **FileSystemObject** (FSO) object model:

- Use the **CreateObject** method to create a **FileSystemObject** object.
- Use the appropriate method on the newly created object.
- Access the object's properties.

The FSO object model is contained in the Scripting **type library**, which is located in the Scrrun.dll file. Therefore, you must have Scrrun.dll in the appropriate system directory on your Web server to use the FSO object model.

**Creating a FileSystemObject Object**

First, create a **FileSystemObject** object by using the **CreateObject** method. In VBScript, use the following code to create an instance of the **FileSystemObject**:

```
Dim fso
Set fso = CreateObject("Scripting.FileSystemObject")
```

This [sample code](#) demonstrates how to create an instance of the **FileSystemObject**.

In JScript, use this code to do the same thing:

```
var fso;
fsa = new ActiveXObject("Scripting.FileSystemObject");
```

In both of these examples, **Scripting** is the name of the type library and **FileSystemObject** is the name of the object that you want to create.
can create only one instance of the **FileSystemObject** object, regardless of how many times you try to create another.

**Using the Appropriate Method**

Second, use the appropriate method of the **FileSystemObject** object. For example, to create a new object, use either **CreateTextFile** or **CreateFolder** (the FSO object model doesn't support the creation or deletion of drives).

To delete objects, use the **DeleteFile** and **DeleteFolder** methods of the **FileSystemObject** object, or the **Delete** method of the **File** and **Folder** objects. You can also copy and move files and folders, by using the appropriate methods.

---

**Note** Some functionality in the **FileSystemObject** object model is redundant. For example, you can copy a file using either the **CopyFile** method of the **FileSystemObject** object, or you can use the **Copy** method of the **File** object. The methods work the same; both exist to offer programming flexibility.

---

**Accessing Existing Drives, Files, and Folders**

To gain access to an existing drive, file, or folder, use the appropriate "get" method of the **FileSystemObject** object:

- **GetDrive**
- **GetFolder**
- **GetFile**

To gain access to an existing file in VBScript:

```
Dim fso, f1
Set fso = CreateObject("Scripting.FileSystemObject")
Set f1 = fso.GetFile("c:\test.txt")
```
To do the same thing in JScript, use the following code:

```javascript
var fso, f1;
fso = new ActiveXObject("Scripting.FileSystemObject");
f1 = fso.GetFile("c:\test.txt");
```

Do not use the "get" methods for newly created objects, since the "create" functions already return a handle to that object. For example, if you create a new folder using the **CreateFolder** method, don't use the **GetFolder** method to access its properties, such as **Name**, **Path**, **Size**, and so forth. Just set a variable to the **CreateFolder** function to gain a handle to the newly created folder, then access its properties, methods, and events. To do this in VBScript, use the following code:

```vbscript
Sub CreateFolder
    Dim fso, fldr
    Set fso = CreateObject("Scripting.FileSystemObject")
    Set fldr = fso.CreateFolder("C:\MyTest")
    Response.Write "Created folder: " & fldr.Name
End Sub
```

To set a variable to the **CreateFolder** function in JScript, use this syntax:

```javascript
function CreateFolder()
{
    var fso, fldr;
    fso = new ActiveXObject("Scripting.FileSystemObject");
    fldr = fso.CreateFolder("C:\MyTest");
    Response.Write("Created folder: " + fldr.Name);
}
```

**Accessing the Object's Properties**

Once you have a handle to an object, you can access its properties. For example, to get the name of a particular folder, first create an instance of the object, then get a handle to it with the appropriate method (in this case, the
GetFolder method, since the folder already exists).

Use this code to get a handle to the GetFolder method in VBScript:

```vbnet
Set fldr = fso.GetFolder("c:\")
```

To do the same thing in JScript, use the following code:

```javascript
var fldr = fso.GetFolder("c:\");
```

Now that you have a handle to a Folder object, you can check its Name property. Use the following code to check this in VBScript:

```vbnet
Response.Write "Folder name is: " & fldr.Name
```

To check a Name property in JScript, use this syntax:

```javascript
Response.Write("Folder name is: " + fldr.Name);
```

To find out the last time a file was modified, use the following VBScript syntax:

```vbnet
Dim fso, f1
Set fso = CreateObject("Scripting.FileSystemObject")
' Get a File object to query.
Set f1 = fso.GetFile("c:\detlog.txt")
' Print information.
Response.Write "File last modified: " & f1.DateLastModified
```

To find out the same thing in JScript, use this code:

```javascript
var fso, f1;
fso = new ActiveXObject("Scripting.FileSystemObject");
// Get a File object to query.
f1 = fso.GetFile("c:\detlog.txt");
// Print information.
Response.Write("File last modified: " + f1.DateLastModified);
```
Working with Drives and Folders

With the **FileSystemObject** (FSO) object model, you can work with drives and folders programmatically just as you can in the Windows Explorer interactively. You can copy and move folders, get information about drives and folders, and so forth.

Getting Information About Drives

The **Drive** object allows you to gain information about the various drives attached to a system, either physically or over a network. Its properties allow you to obtain information about:

- The total size of the drive in bytes (**TotalSize** property)
- How much space is available on the drive in bytes (**AvailableSpace** or **FreeSpace** properties)
- What letter is assigned to the drive (**DriveLetter** property)
- What type of drive it is, such as removable, fixed, network, CD-ROM, or RAM disk (**DriveType** property)
- The drive's serial number (**SerialNumber** property)
- The type of file system the drive uses, such as FAT, FAT32, NTFS, and so forth (**FileSystem** property)
- Whether a drive is available for use (**IsReady** property)
- The name of the share and/or volume (**ShareName** and **VolumeName** properties)
- The path or root folder of the drive (**Path** and **RootFolder** properties)

View the [sample code](sample_code) to see how these properties are used in
Use the **Drive** object to gather information about a drive. You won't see a reference to an actual **Drive** object in the following code; instead, use the **GetDrive** method to get a reference to an existing **Drive** object (in this case, `drv`).

The following example demonstrates how to use the **Drive** object in VBScript:

```vbscript
Sub ShowDriveInfo(drvPath)
    Dim fso, drv, s
    Set fso = CreateObject("Scripting.FileSystemObject")
    Set drv = fso.GetDrive(fso.GetDriveName(drvPath))
    s = "Drive " & UCase(drvPath) & " - "
    s = s & drv.VolumeName & "<br>"
    s = s & "Total Space: " & FormatNumber(drv.TotalSize / 1024, 0) & " Kb" & "<br>"
    s = s & "Free Space: " & FormatNumber(drv.FreeSpace / 1024, 0) & " Kb" & "<br>"
    Response.Write s
End Sub
```

The following code illustrates the same functionality in JScript:

```javascript
function ShowDriveInfo1(drvPath)
{
    var fso, drv, s ="";
    fso = new ActiveXObject("Scripting.FileSystemObject");
    drv = fso.GetDrive(fso.GetDriveName(drvPath));
    s += "Drive " + drvPath.toUpperCase()+ " - ";
    s += drv.VolumeName + "<br>";
    s += "Total Space: " + drv.TotalSize / 1024;
    s += " Kb" + "<br>";
    s += "Free Space: " + drv.FreeSpace / 1024;
```
s += " Kb" + "<br>");
Response.Write(s);
}

**Working with Folders**

Common folder tasks and the methods for performing them are described in the following table.

<table>
<thead>
<tr>
<th>Task</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create a folder.</td>
<td>FileSystemObject.CreateFolder</td>
</tr>
<tr>
<td>Delete a folder.</td>
<td>Folder.Delete or FileSystemObject.DeleteFolder</td>
</tr>
<tr>
<td>Move a folder.</td>
<td>Folder.Move or FileSystemObject.MoveFolder</td>
</tr>
<tr>
<td>Copy a folder.</td>
<td>Folder.Copy or FileSystemObject.CopyFolder</td>
</tr>
<tr>
<td>Retrieve the name of a folder.</td>
<td>Folder.Name</td>
</tr>
<tr>
<td>Find out if a folder exists on a drive.</td>
<td>FileSystemObject.FolderExists</td>
</tr>
<tr>
<td>Get an instance of an existing Folder object.</td>
<td>FileSystemObject.GetFolder</td>
</tr>
<tr>
<td>Find out the name of a folder's parent folder.</td>
<td>FileSystemObject.GetParentFolderName</td>
</tr>
<tr>
<td>Find out the path of system folders.</td>
<td>FileSystemObject.GetSpecialFolder</td>
</tr>
</tbody>
</table>

View the [sample code](#) to see how many of these methods and properties are used in FileSystemObject.

The following example demonstrates how to use the Folder and FileSystemObject objects to manipulate folders and gain information about them in VBScript:
Sub ShowFolderInfo()
    Dim fso, fldr, s
    ' Get instance of FileSystemObject.
    Set fso = CreateObject("Scripting.FileSystemObject")
    ' Get Drive object.
    Set fldr = fso.GetFolder("c:")
    ' Print parent folder name.
    Response.Write "Parent folder name is: " & fldr & "<br>
    ' Print drive name.
    Response.Write "Contained on drive " & fldr.Drive & "<br>
    ' Print root file name.
    If fldr.IsRootFolder = True Then
        Response.Write "This is the root folder." & ""
    Else
        Response.Write "This folder isn't a root folder." & ""
    End If
    ' Create a new folder with the FileSystemObject object.
    fso.CreateFolder ("C:\Bogus")
    Response.Write "Created folder C:\Bogus" & "<br>
    ' Print the base name of the folder.
    Response.Write "Basename = " & fso.GetBaseName("c:\bogus")
    ' Delete the newly created folder.
    fso.DeleteFolder ("C:\Bogus")
    Response.Write "Deleted folder C:\Bogus" & "<br>"
End Sub

This example shows how to use the **Folder** and **FileSystemObject** objects in JScript:

```javascript
function ShowFolderInfo()
{
    var fso, fldr, s = "";
    // Get instance of FileSystemObject.
    fso = new ActiveXObject("Scripting.FileSystemObject");
```
// Get Drive object.
fldr = fso.GetFolder("c:");
// Print parent folder name.
Response.Write("Parent folder name is: " + fldr + "<br>");
// Print drive name.
Response.Write("Contained on drive " + fldr.Drive + "<br>");
// Print root file name.
if (fldr.IsRootFolder)
    Response.Write("This is the root folder.");
else
    Response.Write("This folder isn't a root folder.");
Response.Write("<br><br>");
// Create a new folder with the FileSystemObject object.
fso.CreateFolder("C:\\Bogus");
Response.Write("Created folder C:\\Bogus" + "<br>");
// Print the base name of the folder.
Response.Write("Basename = " + fso.GetBaseName("c:\\bogus")
// Delete the newly created folder.
fso.DeleteFolder("C:\\Bogus");
Response.Write("Deleted folder C:\\Bogus" + "<br>");
}
There are two major categories of file manipulation:

- Creating, adding, or removing data, and reading files
- Moving, copying, and deleting files

Creating Files

There are three ways to create an empty text file (sometimes referred to as a "text stream").

The first way is to use the `CreateTextFile` method. The following example demonstrates how to create a text file using this method in VBScript:

```vbnet
Dim fso, f1
Set fso = CreateObject("Scripting.FileSystemObject")
Set f1 = fso.CreateTextFile("c:\testfile.txt", True)
```

To use this method in JScript, use this code:

```javascript
var fso, f1;
fso = new ActiveXObject("Scripting.FileSystemObject");
f1 = fso.CreateTextFile("c:\testfile.txt", true);
```

View this sample code to see how the `CreateTextFile` method is used in `FileSystemObject`.

The second way to create a text file is to use the `OpenTextFile` method of the `FileSystemObject` object with the `ForWriting` flag set. In VBScript, the code looks like this example:

```vbnet
Dim fso, ts
Const ForWriting = 2
Set fso = CreateObject("Scripting.FileSystemObject")
Set ts = fso.OpenTextFile("c:\test.txt", ForWriting, True)
```
To create a text file using this method in JScript, use this code:

```javascript
var fso, ts;
var ForWriting = 2;
fso = new ActiveXObject("Scripting.FileSystemObject");
ts = fso.OpenTextFile("c:\test.txt", ForWriting, true);
```

A third way to create a text file is to use the **OpenAsTextStream** method with the **ForWriting** flag set. For this method, use the following code in VBScript:

```vbnet
Dim fso, f1, ts
Const ForWriting = 2
Set fso = CreateObject("Scripting.FileSystemObject")
fso.CreateTextFile ("c:\test1.txt")
Set f1 = fso.GetFile("c:\test1.txt")
Set ts = f1.OpenAsTextStream(ForWriting, True)
```

In JScript, use the code in the following example:

```javascript
var fso, f1, ts;
var ForWriting = 2;
fso = new ActiveXObject("Scripting.FileSystemObject");
fso.CreateTextFile ("c:\test1.txt");
f1 = fso.GetFile("c:\test1.txt");
ts = f1.OpenAsTextStream(ForWriting, true);
```

**Adding Data to the File**

Once the text file is created, add data to the file using the following three steps:

1. Open the text file.
2. Write the data.
3. Close the file.
To open an existing file, use either the OpenTextFile method of the FileSystemObject object or the OpenAsTextStream method of the File object.

To write data to the open text file, use the Write, WriteLine, or WriteBlankLines methods of the TextStream object, according to the tasks outlined in the following table.

<table>
<thead>
<tr>
<th>Task</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write data to an open text file without a trailing newline character.</td>
<td>Write</td>
</tr>
<tr>
<td>Write data to an open text file with a trailing newline character.</td>
<td>WriteLine</td>
</tr>
<tr>
<td>Write one or more blank lines to an open text file.</td>
<td>WriteBlankLines</td>
</tr>
</tbody>
</table>

View this sample code to see how the Write, WriteLine, and WriteBlankLines methods are used in FileSystemObject.

To close an open file, use the Close method of the TextStream object.

View this sample code to see how the Close method is used in FileSystemObject.

---

**Note** The newline character contains a character or characters (depending on the operating system) to advance the cursor to the beginning of the next line (carriage return/line feed). Be aware that the end of some strings may already have such nonprinting characters.

---

The following VBScript example demonstrates how to open a file, use all three write methods to add data to the file, and then close the file:

```vbnet
Sub CreateFile()
    Dim fso, tf
    Set fso = CreateObject("Scripting.FileSystemObject")
    Set tf = fso.CreateTextFile("c:\testfile.txt", True)
    ' Write a line with a newline character.
    tf.WriteLine("Testing 1, 2, 3.")
End Sub
```
' Write three newline characters to the file.
tf.WriteBlankLines(3)
' Write a line.
tf.Write ('This is a test.')
tf.Close
End Sub

This example demonstrates how to use the three methods in JScript:

function CreateFile()
{
  var fso, tf;
  fso = new ActiveXObject("Scripting.FileSystemObject");
  tf = fso.CreateTextFile("c:\testfile.txt", true);
  // Write a line with a newline character.
  tf.WriteLine("Testing 1, 2, 3.");
  // Write three newline characters to the file.
  tf.WriteBlankLines(3);
  // Write a line.
  tf.Write ('This is a test.');
  tf.Close();
}

Reading Files

To read data from a text file, use the **Read**, **ReadLine**, or **ReadAll** method of the **TextStream** object. The following table describes which method to use for various tasks.

<table>
<thead>
<tr>
<th>Task</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read a specified number of characters from a file.</td>
<td>Read</td>
</tr>
<tr>
<td>Read an entire line (up to, but not including, the newline character).</td>
<td>ReadLine</td>
</tr>
<tr>
<td>Read the entire contents of a text file.</td>
<td>ReadAll</td>
</tr>
</tbody>
</table>
View this sample code to see how the **ReadAll** and **ReadLine** methods are used in FileSystemObject.

If you use the **Read** or **ReadLine** method and want to skip to a particular portion of data, use the **Skip** or **SkipLine** method. The resulting text of the read methods is stored in a string which can be displayed in a control, parsed by string functions (such as **Left**, **Right**, and **Mid**), concatenated, and so forth.

The following VBScript example demonstrates how to open a file, write to it, and then read from it:

```vbscript
Sub ReadFiles
    Dim fso, f1, ts, s
    Const ForReading = 1
    Set fso = CreateObject("Scripting.FileSystemObject")
    Set f1 = fso.CreateTextFile("c:\testfile.txt", True)
    ' Write a line.
    Response.Write "Writing file <br>"
    f1.WriteLine "Hello World"
    f1.WriteBlankLines(1)
    f1.Close
    ' Read the contents of the file.
    Response.Write "Reading file <br>"
    Set ts = fso.OpenTextFile("c:\testfile.txt", ForReading)
    s = ts.ReadLine
    Response.Write "File contents = " & s & ""
    ts.Close
End Sub
```

This code demonstrates the same thing in JScript:

```javascript
function ReadFiles()
{
    var fso, f1, ts, s;
    var ForReading = 1;
    fso = new ActiveXObject("Scripting.FileSystemObject");
    f1 = fso.CreateTextFile("c:\testfile.txt", true);
    // Write a line.
    Response.Write "Writing file <br>"
    f1.WriteLine "Hello World"
    f1.WriteBlankLines(1)
    f1.Close
    // Read the contents of the file.
    Response.Write "Reading file <br>"
    ts = fso.OpenTextFile("c:\testfile.txt", ForReading)
    s = ts.ReadLine
    Response.Write "File contents = " & s & ""
    ts.Close
}
```
Response.Write("Writing file <br>");
f1.WriteLine("Hello World");
f1.WriteBlankLines(1);
f1.Close();

// Read the contents of the file.
Response.Write("Reading file <br>");

<!DOCTYPE html>

Moving, Copying, and Deleting Files

The FSO object model has two methods each for moving, copying, and deleting files, as described in the following table.

<table>
<thead>
<tr>
<th>Task</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Move a file</td>
<td>File.Move or FileSystemObject.MoveFile</td>
</tr>
<tr>
<td>Copy a file</td>
<td>File.Copy or FileSystemObject.CopyFile</td>
</tr>
<tr>
<td>Delete a file</td>
<td>File.Delete or FileSystemObject.DeleteFile</td>
</tr>
</tbody>
</table>

View this [sample code](#) to see two ways to delete a file in FileSystemObject.

The following VBScript example creates a text file in the root directory of drive C, writes some information to it, moves it to a directory called \tmp, makes a copy of it in a directory called \temp, then deletes the copies from both directories.

To run the following example, create directories named \tmp and \temp in the root directory of drive C:

```vbnet
Sub ManipFiles
    Dim fso, f1, f2, s
    Set fso = CreateObject("Scripting.FileSystemObject")
    Set f1 = fso.CreateTextFile("c:\testfile.txt", True)
    Response.Write "Writing file <br>
    ' Write a line.
```
f1.Write ("This is a test.")
' Close the file to writing.
f1.Close
Response.Write "Moving file to c:\tmp <br>"
' Get a handle to the file in root of C:\.
Set f2 = fso.GetFile("c:\testfile.txt")
' Move the file to \tmp directory.
f2.Move ("c:\tmp\testfile.txt")
Response.Write "Copying file to c:\temp <br>"
' Copy the file to \temp.
f2.Copy ("c:\temp\testfile.txt")
Response.Write "Deleting files <br>"
' Get handles to files' current location.
Set f2 = fso.GetFile("c:\tmp\testfile.txt")
Set f3 = fso.GetFile("c:\temp\testfile.txt")
' Delete the files.
f2.Delete
f3.Delete
Response.Write "All done!"
End Sub

This code shows the same thing in JScript:

function ManipFiles()
{
    var fso, f1, f2, s;
fso = new ActiveXObject("Scripting.FileSystemObject");
f1 = fso.CreateTextFile("c:\testfile.txt", true);
Response.Write("Writing file <br>");
// Write a line.
f1.Write("This is a test.");
// Close the file to writing.
f1.Close();
Response.Write("Moving file to c:tmp <br>");
// Get a handle to the file in root of C:.
f2 = fso.GetFile("c:\testfile.txt");
// Move the file to tmp directory.
f2.Move("c:tmp\testfile.txt");
Response.Write("Copying file to c:temp <br>");
// Copy the file to temp.
f2.Copy("c:temp\testfile.txt");
Response.Write("Deleting files <br>");
// Get handles to files' current location.
f2 = fso.GetFile("c:tmp\testfile.txt");
f3 = fso.GetFile("c:temp\testfile.txt");
// Delete the files.
f2.Delete();
f3.Delete();
Response.Write("All done! ");
}

The sample code described in this section provides a real-world example that demonstrates many of the features available in the FileSystemObject object model. This code shows how all the features of the object model work together, and how to use those features effectively in your own code.

Note that since this code is fairly generic, some additional code and a little tweaking are needed to make this code actually run on your machine. These changes are necessary because of the different ways input and output to the user is handled between Active Server Pages and the Windows Scripting Host.

To run this code on an Active Server Page, use the following steps:

2. Copy the following sample code into that file between the <BODY;>... </BODY> tags.
3. Enclose all the code within <%...%> tags.
4. Move the Option Explicit statement from its current position in the code to the very top of your HTML page, positioning it even before the opening <HTML> tag.
5. Place <%...%> tags around the Option Explicit statement to ensure that it's run on the server side.
6. Add the following code to the end of the sample code:

```vbscript
Sub Print(x)
    Response.Write "<PRE><FONT; FACE="""" Courier New"""" SIZE=""""1"""">"""
    Response.Write x
    Response.Write "</FONT></PRE>"
End Sub
```

The previous code adds a print procedure that will run on the server side,
but display results on the client side. To run this code on the Windows Scripting Host, add the following code to the end of the sample code:

```vbs
Sub Print(x)
    WScript.Echo x
End Sub
Main
```

The code is contained in the following section:

```
'''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
Dim TabStop
Dim NewLine

Const TestDrive = "C"
Const TestFilePath = "C:\Test"

' Constants returned by Drive.DriveType

Const DriveTypeRemovable = 1
Const DriveTypeFixed = 2
Const DriveTypeNetwork = 3
Const DriveTypeCDROM = 4
Const DriveTypeRAMDisk = 5

' Constants returned by File.Attributes

Const FileAttrNormal = 0
Const FileAttrReadOnly = 1
Const FileAttrHidden = 2
Const FileAttrSystem = 4
Const FileAttrVolume = 8
Const FileAttrDirectory = 16
Const FileAttrArchive = 32
Const FileAttrAlias = 64
Const FileAttrCompressed = 128

' Constants for opening files

Const OpenFileForReading = 1
Const OpenFileForWriting = 2
Const OpenFileForAppending = 8

' ShowDriveType
' Purpose:
' Generates a string describing the drive type of a given Drive object.
' Demonstrates the following
' - Drive.DriveType

-------------

Function ShowDriveType(Drive)

    Dim S

    Select Case Drive.DriveType
        Case DriveTypeRemovable
            S = "Removable"
        Case DriveTypeFixed
            S = "Fixed"
        Case DriveTypeNetwork
            S = "Network"
        Case DriveTypeCDROM
            S = "CD-ROM"
        Case DriveTypeRAMDisk
            S = "RAM Disk"
        Case Else
            S = "Unknown"
    End Select

    ShowDriveType = S

End Function

-------------

' ShowFileAttr

' Purpose:
' Generates a string describing the attributes of a file or folder.
' Demonstrates the following
' - File.Attributes
' - Folder.Attributes

-------------

Function ShowFileAttr(File) ' File can be a file or folder
Dim S
Dim Attr

Attr = File.Attributes

If Attr = 0 Then
    ShowFileAttr = "Normal"
    Exit Function
End If

If Attr And FileAttrDirectory Then S = S & "Directory"
If Attr And FileAttrReadOnly Then S = S & "Read-Only"
If Attr And FileAttrHidden Then S = S & "Hidden"
If Attr And FileAttrSystem Then S = S & "System"
If Attr And FileAttrVolume Then S = S & "Volume"
If Attr And FileAttrArchive Then S = S & "Archive"
If Attr And FileAttrAlias Then S = S & "Alias"
If Attr And FileAttrCompressed Then S = S & "Compressed"

ShowFileAttr = S

End Function

' GenerateDriveInformation
'
' Purpose:
'
' Generates a string describing the current state of the available drives.
'
' Demonstrates the following
'
' - FileSystemObject.Drives
' - Iterating the Drives collection
' - Drives.Count
' - Drive.AvailableSpace
' - Drive.DriveLetter
' - Drive.DriveType
' - Drive.FileSystem
' - Drive.FreeSpace
' - Drive.IsReady
' - Drive.Path
' - Drive.SerialNumber
' - Drive.ShareName
' - Drive_TotalSize
' - Drive.VolumeName
'
'..............................................................................
Function GenerateDriveInformation(FSO)

    Dim Drives
    Dim Drive
    Dim S

    Set Drives = FSO.Drives

    S = "Number of drives:" & TabStop & Drives.Count & NewLine & NewLine

    ' Construct 1st line of report.
    S = S & String(2, TabStop) & "Drive"
    S = S & String(3, TabStop) & "File"
    S = S & TabStop & "Total"
    S = S & TabStop & "Free"
    S = S & TabStop & "Available"
    S = S & TabStop & "Serial" & NewLine

    ' Construct 2nd line of report.
    S = S & "Letter"
    S = S & TabStop & "Path"
    S = S & TabStop & "Type"
    S = S & TabStop & "Ready?"
    S = S & TabStop & "Name"
    S = S & TabStop & "System"
    S = S & TabStop & "Space"
    S = S & TabStop & "Space"
    S = S & TabStop & "Space"
    S = S & TabStop & "Number" & NewLine

    ' Separator line.
    S = S & String(105, "-") & NewLine

    For Each Drive In Drives
        S = S & Drive.DriveLetter
        S = S & TabStop & Drive.Path
        S = S & TabStop & ShowDriveType(Drive)
        S = S & TabStop & Drive.IsReady

        If Drive.IsReady Then
            If DriveTypeNetwork = Drive.DriveType Then
                S = S & TabStop & Drive.ShareName
            Else
                S = S & TabStop & Drive.VolumeName
            End If
        End If

        S = S & TabStop & Drive.FileSystem
        S = S & TabStop & Drive.TotalSize
S = S & TabStop & Drive.FreeSpace
S = S & TabStop & Drive.AvailableSpace
S = S & TabStop & Hex(Drive.SerialNumber)
End If

S = S & NewLine

Next

GenerateDriveInformation = S

End Function

''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
'
' GenerateFileInformation
'

' Purpose:
',
' Generates a string describing the current state of a file.
',
' Demonstrates the following
',
' - File.Path
' - File.Name
' - File.Type
' - File.DateCreated
' - File.DateLastAccessed
' - File.DateLastModified
' - File.Size
',
''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''

Function GenerateFileInformation(File)

Dim S

S = NewLine & "Path:" & TabStop & File.Path
S = S & NewLine & "Name:" & TabStop & File.Name
S = S & NewLine & "Type:" & TabStop & File.Type
S = S & NewLine & "Attribs:" & TabStop & ShowFileAttr(File)
S = S & NewLine & "Created:" & TabStop & File.DateCreated
S = S & NewLine & "Modified:" & TabStop & File.DateLastModified

GenerateFileInformation = S

End Function
generateFolderInformation

' Purpose:

' Generates a string describing the current state of a folder.

' Demonstrates the following

' - Folder.Path
' - Folder.Name
' - Folder.DateCreated
' - Folder.DateLastAccessed
' - Folder.DateLastModified
' - Folder.Size

---------------------------------------------------------------

Function GenerateFolderInformation(Folder)

    Dim S

    S = "Path:" & TabStop & Folder.Path
    S = S & NewLine & "Name:" & TabStop & Folder.Name
    S = S & NewLine & "Attribs:" & TabStop & ShowFileAttr(Folder)
    S = S & NewLine & "Created:" & TabStop & Folder.DateCreated
    S = S & NewLine & "Accessed:" & TabStop & Folder.DateLastAccessed
    S = S & NewLine & "Modified:" & TabStop & Folder.DateLastModified
    S = S & NewLine & "Size:" & TabStop & Folder.Size & NewLine

    GenerateFolderInformation = S

End Function

---------------------------------------------------------------

generateAllFolderInformation

' Purpose:

' Generates a string describing the current state of a folder and all files and subfolders.

' Demonstrates the following

' - Folder.Path
' - Folder.SubFolders
Function GenerateAllFolderInformation(Folder)

Dim S
Dim SubFolders
Dim SubFolder
Dim Files
Dim File

S = "Folder:" & TabStop & Folder.Path & NewLine & NewLine

Set Files = Folder.Files

If 1 = Files.Count Then
    S = S & "There is 1 file" & NewLine
Else
    S = S & "There are " & Files.Count & " files" & NewLine
End If

If Files.Count <> 0 Then
    For Each File In Files
        S = S & GenerateFileInformation(File)
    Next
End If

Set SubFolders = Folder.SubFolders

If 1 = SubFolders.Count Then
    S = S & NewLine & "There is 1 sub folder" & NewLine & NewLine
Else
    S = S & NewLine & "There are " & SubFolders.Count & " sub folders" & NewLine & NewLin
End If

If SubFolders.Count <> 0 Then
    For Each SubFolder In SubFolders
        S = S & GenerateFolderInformation(SubFolder)
    Next

    S = S & NewLine

    For Each SubFolder In SubFolders
        S = S & GenerateAllFolderInformation(SubFolder)
    Next
End If

GenerateAllFolderInformation = S

End Function

'''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
'	GenerateTestInformation
'
'''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
Function GenerateTestInformation(FSO)
    Dim TestFolder
    Dim S

    If Not FSO.DriveExists(TestDrive) Then Exit Function
    If Not FSO.FolderExists(TestFilePath) Then Exit Function

    Set TestFolder = FSO.GetFolder(TestFilePath)
    GenerateTestInformation = GenerateAllFolderInformation(TestFolder)

End Function

'''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
'	DeleteTestDirectory
'
'''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
Function DeleteTestDirectory()
    ' Demonstrate the following
    ' - FileSystemObject.GetFolder
    ' - FileSystemObject.DeleteFile

Sub DeleteTestDirectory(FSO)

    Dim TestFolder
    Dim SubFolder
    Dim File

    ' Two ways to delete a file:
    FSO.DeleteFile(TestFilePath & "\Beatles\OctopusGarden.txt")
    Set File = FSO.GetFile(TestFilePath & "\Beatles\BathroomWindow.txt")
    File.Delete

    ' Two ways to delete a folder:
    FSO.DeleteFolder(TestFilePath & "\Beatles")
    FSO.DeleteFile(TestFilePath & "\ReadMe.txt")
    Set TestFolder = FSO.GetFolder(TestFilePath)
    TestFolder.Delete

End Sub

CreateLyrics

Purpose:

Builds a couple of text files in a folder.

Demonstrates the following

- FileSystemObject.CreateTextFile
- TextStream.WriteLine
- TextStream.Write
- TextStream.WriteBlankLines
- TextStream.Close
Sub CreateLyrics(Folder)

    Dim TextStream

    Set TextStream = Folder.CreateTextFile("OctopusGarden.txt")

    TextStream.Write("Octopus' Garden ") ' Note that this does not add a line feed to the file.
    TextStream.WriteLine("(by Ringo Starr)")
    TextStream.WriteLine("I'd like to be under the sea in an octopus' garden in the shade,"")
    TextStream.WriteLine("He'd let us in, knows where we've been -- in his octopus' garden in the shade")
    TextStream.WriteBlankLines(2)

    TextStream.Close

    Set TextStream = Folder.CreateTextFile("BathroomWindow.txt")
    TextStream.WriteLine("She Came In Through The Bathroom Window (by Lennon/McCartney)")
    TextStream.WriteLine("She came in through the bathroom window protected by a silver spoon")
    TextStream.WriteLine("But now she sucks her thumb and wanders by the banks of her own lagoon")
    TextStream.WriteBlankLines(2)
    TextStream.Close

End Sub

' GetLyrics
'
' Purpose:
'
' Displays the contents of the lyrics files.
'
' Demonstrates the following
'
' - FileSystemObject.OpenTextFile
' - FileSystemObject.GetFile
' - TextStream.ReadAll
' - TextStream.Close
' - File.OpenAsTextStream
' - TextStream.AtEndOfStream
' - TextStream.ReadLine
'

Function GetLyrics(FSO)

    Dim TextStream
Dim S
Dim File

' There are several ways to open a text file, and several ways to read the
' data out of a file. Here's two ways to do each:

Set TextStream = FSO.OpenTextFile(TestFilePath & "\Beatles\OctopusGarden.txt", OpenFileForReading)
S = TextStream.ReadAll & NewLine & NewLine
TextStream.Close

Set File = FSO.GetFile(TestFilePath & "\Beatles\BathroomWindow.txt")
Set TextStream = File.OpenAsTextStream(OpenFileForReading)
Do While Not TextStream.AtEndOfStream
   S = S & TextStream.ReadLine & NewLine
Loop
TextStream.Close

GetLyrics = S

End Function

'''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
Function BuildTestDirectory(FSO)

    Dim TestFolder
    Dim SubFolders
    Dim SubFolder
    Dim TextStream

    ' Bail out if (a) the drive does not exist, or if (b) the directory being built
    ' already exists.

    If Not FSO.DriveExists(TestDrive) Then
        BuildTestDirectory = False
        Exit Function
    End If

    If FSO.FolderExists(TestFilePath) Then
        BuildTestDirectory = False
        Exit Function
    End If

    Set TestFolder = FSO.CreateFolder(TestFilePath)

    Set TextStream = FSO.CreateTextFile(TestFilePath & "\ReadMe.txt")
    TextStream.WriteLine("My song lyrics collection")
    TextStream.Close

    Set SubFolders = TestFolder.SubFolders

    Set SubFolder = SubFolders.Add("Beatles")

    CreateLyrics SubFolder

    BuildTestDirectory = True

End Function

' The main routine
'
' First, it creates a test directory, along with some subfolders and files.
' Then, it dumps some information about the available disk drives and
' about the test directory, and then cleans everything up again.
'
Sub Main

    Dim FSO

    ' Set up global data.
    TabStop = Chr(9)
    NewLine = Chr(10)

    Set FSO = CreateObject("Scripting.FileSystemObject")

    If Not BuildTestDirectory(FSO) Then
        Print "Test directory already exists or cannot be created. Cannot continue."
        Exit Sub
    End If

    Print GenerateDriveInformation(FSO) & NewLine & NewLine
    Print GenerateTestInformation(FSO) & NewLine & NewLine
    Print GetLyrics(FSO) & NewLine & NewLine
    DeleteTestDirectory(FSO)

    End Sub
### VBScript Features

<table>
<thead>
<tr>
<th>Category</th>
<th>Keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array handling</td>
<td>Array, Dim, Private, Public, ReDim, isArray, Erase, LBound, UBound</td>
</tr>
<tr>
<td>Assignments</td>
<td>Set</td>
</tr>
<tr>
<td>Comments</td>
<td>Comments using * or Rem</td>
</tr>
<tr>
<td>Constants/Literals</td>
<td>Empty, Nothing, Null, True, False</td>
</tr>
<tr>
<td>Control flow</td>
<td>Do...Loop, For...Next, For Each...Next, If...Then...Else, Select Case, While...Wend, With</td>
</tr>
<tr>
<td>Conversions</td>
<td>Abs, Asc, AscB, AscW, Chr, ChrB, ChrW, CBool, CBvte, CCur, CDate, CDbl, CInt, CLng, CSng, CStr, DateSerial, DateValue, Hex, Oct, Fix, Int, Sgn, TimeSerial, TimeValue</td>
</tr>
<tr>
<td>Dates/Times</td>
<td>Date, Time, DateAdd, DateDiff, DatePart, DateSerial, DateValue, Day, Month, MonthName, Weekday, WeekdayName, Year, Hour, Minute, Second, Now, TimeSerial, TimeValue</td>
</tr>
<tr>
<td>Class Const</td>
<td>Class, Const</td>
</tr>
<tr>
<td>Declarations</td>
<td>Dim, Private, Public, ReDim Function, Sub Property Get, Property Let, Property Set</td>
</tr>
<tr>
<td>-------------------------</td>
<td>----------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Error Handling</td>
<td>On Error Err</td>
</tr>
<tr>
<td>Expressions</td>
<td>Eval Execute RegExp Replace Test</td>
</tr>
<tr>
<td>Formatting Strings</td>
<td>FormatCurrency FormatDateTime FormatNumber FormatPercent</td>
</tr>
<tr>
<td>Input/Output</td>
<td>InputBox LoadPicture MsgBox</td>
</tr>
<tr>
<td>Literals</td>
<td>Empty False Nothing Null True</td>
</tr>
<tr>
<td>Math</td>
<td>Atn, Cos, Sin, Tan Exp, Log, Sqr Randomize, Rnd</td>
</tr>
<tr>
<td>Miscellaneous</td>
<td>Eval Function Execute Statement RGB Function</td>
</tr>
<tr>
<td>Objects</td>
<td>CreateObject Err Object GetObject RegExp</td>
</tr>
<tr>
<td>Operators</td>
<td>Addition (+), Subtraction (-) Exponentiation (^) Modulus arithmetic (Mod)</td>
</tr>
<tr>
<td></td>
<td>Multiplication (*), Division (/) Integer Division ()</td>
</tr>
<tr>
<td></td>
<td>Negation (-) String concatenation (&amp;) Equality (=), Inequality (&lt;&gt; ) Less Than (&lt;)</td>
</tr>
<tr>
<td></td>
<td>Greater Than (&gt;), Greater Than or Equal To (&gt;=) Is And, Or, Xor Eqv, Imp</td>
</tr>
<tr>
<td>Options</td>
<td>Option Explicit Call Function, Sub</td>
</tr>
<tr>
<td>Procedures</td>
<td>Property Get, Property Let, Property Set</td>
</tr>
<tr>
<td>------------</td>
<td>----------------------------------------</td>
</tr>
<tr>
<td>Rounding</td>
<td>Abs, Int, Fix, Round, Sgn</td>
</tr>
<tr>
<td>Strings</td>
<td>Asc, AscB, AscW, Chr, ChrB, ChrW, Filter, InStr, InStrB, InStrRev, Len, LenB, LCase, UCase, Left, LeftB, Mid, MidB, Right, RightB, Replace, Space, Split, StrComp, String, StrReverse, LTrim, RTrim, Trim</td>
</tr>
<tr>
<td>Variants</td>
<td>IsArray, IsDate, IsEmpty, IsNull, IsNumeric, IsObject, TypeName, VarType</td>
</tr>
</tbody>
</table>
### Features not in VBScript

<table>
<thead>
<tr>
<th>Category</th>
<th>Omitted Feature/Keyword</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array Handling</td>
<td>Option Base&lt;br&gt;Declaring arrays with lower bound &lt;&gt; 0</td>
</tr>
<tr>
<td>Collection</td>
<td>Add, Count, Item, Remove&lt;br&gt;Access to collections using ! character (e.g., MyCollection!Foo)</td>
</tr>
<tr>
<td>Conditional Compilation</td>
<td>#Const&lt;br&gt;#If...Then...#Else</td>
</tr>
<tr>
<td>Control Flow</td>
<td>DoEvents&lt;br&gt;GoSub...Return, GoTo&lt;br&gt;OnError GoTo&lt;br&gt;On...GoSub, On...GoTo&lt;br&gt;Line numbers, Line labels</td>
</tr>
<tr>
<td>Conversion</td>
<td>CVar, CDate&lt;br&gt;Str, Val</td>
</tr>
<tr>
<td>Data Types</td>
<td>All intrinsic data types except Variant&lt;br&gt;Type...End Type</td>
</tr>
<tr>
<td>Date/Time</td>
<td>Date statement, Time statement</td>
</tr>
<tr>
<td>DDE</td>
<td>LinkExecute, LinkPoke, LinkRequest, LinkSend</td>
</tr>
</tbody>
</table>

Language Reference
<table>
<thead>
<tr>
<th>Debugging</th>
<th>End, Stop</th>
</tr>
</thead>
</table>
| Declaration | Declare (for declaring DLLs)  
Optional  
ParamArray  
Static |
| Error Handling | Erl  
Error  
Resume, Resume Next |
| File Input/Output | All traditional Basic file I/O |
| Financial | All financial functions |
| Object Manipulation | TypeOf |
| Objects | Clipboard  
Collection |
| Operators | Like |
| Options | DefType  
Option Base  
Option Compare  
Option Private Module |
| Select Case | Expressions containing \texttt{Is} keyword or any comparison operators  
Expressions containing a range of values using the \texttt{To} keyword. |
| Strings | Fixed-length strings  
LSet, RSet  
Mid Statement  
StrConv |
| Using Objects | Collection access using ! |
# VBScript Features not in Visual Basic for Applications

<table>
<thead>
<tr>
<th>Category</th>
<th>Feature/Keyword</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declarations</td>
<td>Class</td>
</tr>
<tr>
<td>Miscellaneous</td>
<td>Eval</td>
</tr>
<tr>
<td></td>
<td>Execute</td>
</tr>
<tr>
<td>Objects</td>
<td>RegExp</td>
</tr>
<tr>
<td>Script Engine Identification</td>
<td>ScriptEngine</td>
</tr>
<tr>
<td></td>
<td>ScriptEngineBuildVersion</td>
</tr>
<tr>
<td></td>
<td>ScriptEngineMajorVersion</td>
</tr>
<tr>
<td></td>
<td>ScriptEngineMinorVersion</td>
</tr>
</tbody>
</table>
# Microsoft Scripting Run-Time Library Reference Features

<table>
<thead>
<tr>
<th>Category</th>
<th>Feature/Keyword</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Collections</strong></td>
<td><a href="#">Drives</a>, <a href="#">Files</a>, <a href="#">Folders</a></td>
</tr>
<tr>
<td><strong>Data Storage</strong></td>
<td><a href="#">Dictionary</a></td>
</tr>
<tr>
<td><strong>Dictionary</strong></td>
<td><a href="#">Add</a>, <a href="#">Exists</a>, <a href="#">Items</a>, <a href="#">Keys</a>, <a href="#">Remove</a>, <a href="#">RemoveAll</a>, <a href="#">Count</a>, <a href="#">Item</a>, <a href="#">Key</a></td>
</tr>
<tr>
<td><strong>File System</strong></td>
<td><a href="#">Drive</a>, <a href="#">File</a>, <a href="#">FileSystemObject</a>, <a href="#">Folder</a>, <a href="#">TextStream</a></td>
</tr>
<tr>
<td></td>
<td><a href="#">BuildPath</a>, <a href="#">CopyFile</a>, <a href="#">CopyFolder</a>, <a href="#">CreateFolder</a>, <a href="#">CreateTextFile</a>, <a href="#">DeleteFile</a>, <a href="#">DeleteFolder</a>, <a href="#">DriveExists</a>, <a href="#">FileExists</a>, <a href="#">FolderExists</a>, <a href="#">GetAbsolutePathName</a>, <a href="#">GetBaseName</a></td>
</tr>
</tbody>
</table>
| FileSystemObject | GetDrive, GetDriveName  
GetFile, GetExtensionName GetFileName  
GetFolder, GetParentFolderName  
GetSpecialFolder  
GetTempName  
MoveFile, MoveFolder  
OpenTextFile  
Drives |
| Drive, Drives | AvailableSpace  
Count  
DriveLetter  
DriveType  
FileSystem  
FreeSpace  
IsReady  
Item  
RootFolder  
SerialNumber  
ShareName  
TotalSize  
VolumeName |
| File, Files Folder, Folders | Add  
Attributes  
Copy, Delete, Move  
Count  
OpenAsTextStream  
DateCreated, DateLastAccessed, DateLastModified  
Drive  
Item  
ParentFolder  
Name, Path  
ShortName, ShortPath  
Size |
| | Close  
Read, ReadAll, ReadLine |
| TextStream | Skip, SkipLine, Write, WriteBlankLines, WriteLine, AtEndOfLine, AtEndOfStream, Column, Line |
**Abs**

**Function**

[See Also]

---

**Description**

Returns the absolute value of a number.

**Syntax**

```
Abs(number)
```

The `number` argument can be any valid numeric expression. If `number` contains `Null`, `Null` is returned; if it is an uninitialized variable, zero is returned.

**Remarks**

The absolute value of a number is its unsigned magnitude. For example, `Abs(-1)` and `Abs(1)` both return 1.

The following example uses the **Abs** function to compute the absolute value of a number:

```
Dim MyNumber
MyNumber = Abs(50.3) ' Returns 50.3.
MyNumber = Abs(-50.3) ' Returns 50.3.
```
Operator

See Also

Description

Sums two numbers.

Syntax

\[ result = expression1 + expression2 \]

The + operator syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>Any numeric variable.</td>
</tr>
<tr>
<td>expression1</td>
<td>Any expression.</td>
</tr>
<tr>
<td>expression2</td>
<td>Any expression.</td>
</tr>
</tbody>
</table>

Remarks

Although you can also use the + operator to concatenate two character strings, you should use the & operator for concatenation to eliminate ambiguity and provide self-documenting code.

When you use the + operator, you may not be able to determine whether addition or string concatenation will occur.

The underlying subtype of the expressions determines the behavior of the + operator in the following way:

<table>
<thead>
<tr>
<th>If</th>
<th>Then</th>
</tr>
</thead>
<tbody>
<tr>
<td>Both expressions are numeric</td>
<td>Add.</td>
</tr>
<tr>
<td>Both expressions are strings</td>
<td>Concatenate.</td>
</tr>
<tr>
<td></td>
<td>Add.</td>
</tr>
</tbody>
</table>
One expression is numeric and the other is a string.

If one or both expressions are **Null** expressions, result is **Null**. If both expressions are **Empty**, result is an **Integer** subtype. However, if only one expression is **Empty**, the other expression is returned unchanged as **result**.
And Operator

See Also

Description

Performs a logical conjunction on two expressions.

Syntax

\[ result = expression1 \text{ And } expression2 \]

The \textbf{And} operator syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>Any numeric variable.</td>
</tr>
<tr>
<td>expression1</td>
<td>Any expression.</td>
</tr>
<tr>
<td>expression2</td>
<td>Any expression.</td>
</tr>
</tbody>
</table>

Remarks

If, and only if, both expressions evaluate to \textbf{True}, \text{result} is \textbf{True}. If either expression evaluates to \textbf{False}, \text{result} is \textbf{False}. The following table illustrates how \text{result} is determined:

<table>
<thead>
<tr>
<th>If \textit{expression1} is</th>
<th>And \textit{expression2} is</th>
<th>The \textit{result} is</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>True</td>
<td>\textbf{Null}</td>
<td>Null</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>\textbf{Null}</td>
<td>False</td>
</tr>
</tbody>
</table>
The **And** operator also performs a **bitwise comparison** of identically positioned bits in two **numeric expressions** and sets the corresponding bit in *result* according to the following table:

<table>
<thead>
<tr>
<th>If bit in <em>expression1</em> is</th>
<th>And bit in <em>expression2</em> is</th>
<th>The <em>result</em> is</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Array Function

Description

Returns a Variant containing an array.

Syntax

Array(arglist)

The required arglist argument is a comma-delimited list of values that are assigned to the elements of an array contained with the Variant. If no arguments are specified, an array of zero length is created.

Remarks

The notation used to refer to an element of an array consists of the variable name followed by parentheses containing an index number indicating the desired element. In the following example, the first statement creates a variable named A. The second statement assigns an array to variable A. The last statement assigns the value contained in the second array element to another variable.

Dim A
A = Array(10,20,30)
B = A(2) ' B is now 30.
Note A variable that is not declared as an array can still contain an array. Although a **Variant** variable containing an array is conceptually different from an array variable containing **Variant** elements, the array elements are accessed in the same way.
Function

See Also

Description

Returns the ANSI character code corresponding to the first letter in a string.

Syntax

Asc(string)

The string argument is any valid string expression. If the string contains no characters, a run-time error occurs.

Remarks

In the following example, Asc returns the ANSI character code of the first letter of each string:

Dim MyNumber
MyNumber = Asc("A")    ' Returns 65.
MyNumber = Asc("a")    ' Returns 97.
MyNumber = Asc("Apple") ' Returns 65.

Note The AscB function is used with byte data contained in a string. Instead of returning the character code for the first character, AscB returns the first byte. AscW is provided for 32-
bit platforms that use Unicode characters. It returns the Unicode (wide) character code, thereby avoiding the conversion from Unicode to ANSI.
Operator

See Also

Description

Assigns a value to a variable or property.

Syntax

\[ variable = value \]

The = operator syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>variable</td>
<td>Any variable or any writable property.</td>
</tr>
<tr>
<td>value</td>
<td>Any numeric or string literal, constant, or expression.</td>
</tr>
</tbody>
</table>

Remarks

The name on the left side of the equal sign can be a simple scalar variable or an element of an array. Properties on the left side of the equal sign can only be those properties that are writable at run time.
**Description**

Returns the arctangent of a number.

**Syntax**

\[ \text{Atn}(number) \]

The `number` argument can be any valid numeric expression.

**Remarks**

The **Atn** function takes the ratio of two sides of a right triangle (`number`) and returns the corresponding angle in radians. The ratio is the length of the side opposite the angle divided by the length of the side adjacent to the angle. The range of the result is \(-\pi/2\) to \(\pi/2\) radians.

To convert degrees to radians, multiply degrees by \(\pi/180\). To convert radians to degrees, multiply radians by \(180/\pi\).

The following example uses **Atn** to calculate the value of pi:

```
Dim pi
pi = 4 * Atn(1) ' Calculate the value of pi.
```

**Note** **Atn** is the inverse trigonometric function of **Tan**, which
takes an angle as its argument and returns the ratio of two sides of a right triangle. Do not confuse Atn with the cotangent, which is the simple inverse of a tangent (1/tangent).
Call Statement

Description

Transfers control to a Sub or Function procedure.

Syntax

\[
\text{[Call] } \text{name [argumentlist]}
\]

The Call statement syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Call</td>
<td>Optional keyword. If specified, you must enclose argumentlist in parentheses. For example: Call MyProc(0)</td>
</tr>
</tbody>
</table>

name Required. Name of the procedure to call. argumentlist Optional. Comma-delimited list of variables, arrays, or expressions to pass to the 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.
Call MyFunction("Hello World")

Function MyFunction(text)
    MsgBox text
End Function
CBool Function

Description

Returns an expression that has been converted to a Variant of subtype Boolean.

Syntax

CBool(expression)

The expression argument is any valid expression.

Remarks

If expression is zero, False is returned; otherwise, True is returned. If expression can't be interpreted as a numeric value, a run-time error occurs.

The following example uses the CBool function to convert an expression to a Boolean. If the expression evaluates to a nonzero value, CBool returns True; otherwise, it returns False.

Dim A, B, Check
A = 5: B = 5    ' Initialize variables.
Check = CBool(A = B)  ' Check contains True.
A = 0          ' Define variable.
Check = CBool(A)    ' Check contains False.
CByte

Function

See Also

Description

Returns an expression that has been converted to a Variant of subtype Byte.

Syntax

CByte(expression)

The expression argument is any valid expression.

Remarks

In general, you can document your code using the subtype conversion functions to show that the result of some operation should be expressed as a particular data type rather than the default data type. For example, use CByte to force byte arithmetic in cases where currency, single-precision, double-precision, or integer arithmetic normally would occur.

Use the CByte function to provide internationally aware conversions from any other data type to a Byte subtype. For example, different decimal separators are properly recognized depending on the locale setting of your system, as are different thousand separators.

If expression lies outside the acceptable range for the Byte subtype, an error occurs. The following example uses the CByte function to convert an expression to a byte:

Dim MyDouble, MyByte
MyDouble = 125.5678 ' MyDouble is a Double
MyByte = CByte(MyDouble) ' MyByte contains
**CCur**

**Function**

See Also

**Description**

Returns an expression that has been converted to a **Variant** of subtype **Currency**.

**Syntax**

```
CCur(expression)
```

The `expression` argument is any valid expression.

**Remarks**

In general, you can document your code using the subtype conversion functions to show that the result of some operation should be expressed as a particular data type rather than the default data type. For example, use **CCur** to force currency arithmetic in cases where integer arithmetic normally would occur.

You should use the **CCur** function to provide internationally aware conversions from any other data type to a **Currency** subtype. For example, different decimal separators and thousands separators are properly recognized depending on the **locale** setting of your system.

The following example uses the **CCur** function to convert an expression to a Currency:

```
Dim MyDouble, MyCurr
MyDouble = 543.214588 ' MyDouble is a Double
MyCurr = CCur(MyDouble * 2) ' Convert result to a Currency (1086.4292).
```
CDate Function

Description

Returns an expression that has been converted to a Variant of subtype Date.

Syntax

CDate(date)

The date argument is any valid date expression.

Remarks

Use the IsDate function to determine if date can be converted to a date or time. CDate recognizes date literals and time literals as well as some numbers that fall within the range of acceptable dates. When converting a number to a date, the whole number portion is converted to a date. Any fractional part of the number is converted to a time of day, starting at midnight.

CDate recognizes date formats according to the locale setting of your system. The correct order of day, month, and year may not be determined if it is provided in a format other than one of the recognized date settings. In addition, a long date format is not recognized if it also contains the day-of-the-week string.

The following example uses the CDate function to convert a string to a
In general, hard coding dates and times as strings (as shown in this example) is not recommended. Use date and time literals (such as #10/19/1962#, #4:45:23 PM#) instead.

```plaintext
MyDate = "October 19, 1962"  ' Define date.
MyShortDate = CDate(MyDate)  ' Convert to Dat
MyTime = "4:35:47 PM"        ' Define time.
MyShortTime = CDate(MyTime)  ' Convert to Dat
```
**CDbl**

Function

**See Also**

---

**Description**

Returns an expression that has been converted to a **Variant** of subtype **Double**.

**Syntax**

```vbscript
CDbl(expression)
```

The `expression` argument is any valid expression.

**Remarks**

In general, you can document your code using the subtype conversion functions to show that the result of some operation should be expressed as a particular data type rather than the default data type. For example, use **CDbl** or **CSng** to force double-precision or single-precision arithmetic in cases where currency or integer arithmetic normally would occur.

Use the **CDbl** function to provide internationally aware conversions from any other data type to a **Double** subtype. For example, different decimal separators and thousands separators are properly recognized depending on the **locale** setting of your system.

This example uses the **CDbl** function to convert an expression to a **Double**.

```vbscript
Dim MyCurr, MyDouble
```
MyCurr = CCur(234.456784)  ' MyCurr is a Currency (234.4567).
MyDouble = CDbl(MyCurr * 8.2 * 0.01)  ' Convert result to a Double (19.2254576).
**Chr Function**

**Description**

Returns the character associated with the specified ANSI character code.

**Syntax**

```
Chr(charcode)
```

The *charcode* argument is a number that identifies a character.

**Remarks**

Numbers from 0 to 31 are the same as standard, nonprintable ASCII codes. For example, `Chr(10)` returns a line feed character.

The following example uses the `Chr` function to return the character associated with the specified character code:

```
Dim MyChar
MyChar = Chr(65)  ' Returns A.
MyChar = Chr(97)  ' Returns a.
MyChar = Chr(62)  ' Returns >.
MyChar = Chr(37)  ' Returns %.
```
**Note** The `ChrB` function is used with byte data contained in a string. Instead of returning a character, which may be one or two bytes, `ChrB` always returns a single byte. `ChrW` is provided for 32-bit platforms that use Unicode characters. Its argument is a Unicode (wide) character code, thereby avoiding the conversion from ANSI to Unicode.
CInt Function

Description

Returns an expression that has been converted to a Variant of subtype Integer.

Syntax

\[
\text{CInt}(\text{expression})
\]

The expression argument is any valid expression.

Remarks

In general, you can document your code using the subtype conversion functions to show that the result of some operation should be expressed as a particular data type rather than the default data type. For example, use CInt or CLng to force integer arithmetic in cases where currency, single-precision, or double-precision arithmetic normally would occur.

Use the CInt function to provide internationally aware conversions from any other data type to an Integer subtype. For example, different decimal separators are properly recognized depending on the locale setting of your system, as are different thousand separators.

If expression lies outside the acceptable range for the Integer subtype, an error occurs.

The following example uses the CInt function to convert a value to an
Integer:

Dim MyDouble, MyInt
MyDouble = 2345.5678  ' MyDouble is a Double.
MyInt = CInt(MyDouble)  ' MyInt contains 2346.

---

**Note**  **CInt** differs from the **Fix** and **Int** functions, which truncate, rather than round, the fractional part of a number. When the fractional part is exactly 0.5, the **CInt** function always rounds it to the nearest even number. For example, 0.5 rounds to 0, and 1.5 rounds to 2.
**Class Object**

**Description**

The object created using the **Class** statement. Provides access to the events of the **class**.

**Remarks**

You cannot explicitly declare a **variable** to be of type Class. In the VBScript context, the term "class object" refers to any object defined using the VBScript **Class** statement.

Once you have created a class definition using the **Class** statement, you can create an instance of the class using the following form:

```
Dim X
Set X = New classname
```

Because VBScript is a late-bound language, you cannot do any of the following:

```
Dim X as New classname
```

or

```
Dim X
```
\[
X = \text{New } \textit{classname}
\]

or

\[
\text{Set } X = \text{New Scripting.FileSystemObject}
\]
Description

Declares the name of a class, as well as a definition of the variables, properties, and methods that comprise the class.

Syntax

```
Class name
    statements
End Class
```

The Class statement syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Required. Name of the Class; follows standard variable naming conventions.</td>
</tr>
<tr>
<td>statements</td>
<td>Required. One or more statements that define the variables, properties, and methods of the Class.</td>
</tr>
</tbody>
</table>

Remarks

Within a Class block, members are declared as either Private or Public using the appropriate declaration statements. Anything declared as Private is visible only within the Class block. Anything declared as Public is visible within the Class block, as well as by code outside the Class block. Anything not explicitly declared as either Private or Public is Public by default.
**Procedures** (either **Sub** or **Function**) declared **Public** within the class block become methods of the class. **Public** variables serve as properties of the class, as do properties explicitly declared using **Property Get**, **Property Let**, and **Property Set**. Default properties and methods for the class are specified in their declarations using the **Default** keyword. See the individual declaration statement topics for information on how this **keyword** is used.
Clear Method

Description

Clears all property settings of the Err object.

Syntax

object.Clear

The object is always the Err object.

Remarks

Use Clear to explicitly clear the Err object after an error has been handled. This is necessary, for example, when you use deferred error handling with On Error Resume Next. VBScript calls the Clear method automatically whenever any of the following statements is executed:

- On Error Resume Next
- Exit Sub
- Exit Function

The following example illustrates use of the Clear method:

On Error Resume Next
Err.Raise 6 ' Raise an overflow error.
MsgBox ("Error # " & CStr(Err.Number) & " " & Err.Description)
Err.Clear ' Clear the error.
Description

Returns an expression that has been converted to a **Variant** of subtype **Long**.

Syntax

```
CLng(expression)
```

The *expression* argument is any valid expression.

Remarks

In general, you can document your code using the subtype conversion functions to show that the result of some operation should be expressed as a particular data type rather than the default data type. For example, use **CInt** or **CLng** to force integer arithmetic in cases where currency, single-precision, or double-precision arithmetic normally would occur.

Use the **CLng** function to provide internationally aware conversions from any other data type to a **Long** subtype. For example, different decimal separators are properly recognized depending on the **locale** setting of your system, as are different thousand separators.

If *expression* lies outside the acceptable range for the **Long** subtype, an error occurs.

The following example uses the **CLng** function to convert a value to a
Long:

```
Dim MyVal1, MyVal2, MyLong1, MyLong2
MyVal1 = 25427.45: MyVal2 = 25427.55  ' MyVa
MyLong1 = CLng(MyVal1)                ' MyLong1
MyLong2 = CLng(MyVal2)                ' MyLong2
```

---

**Note**  
*CLng* differs from the *Fix* and *Int* functions, which truncate, rather than round, the fractional part of a number. When the fractional part is exactly 0.5, the *CLng* function always rounds it to the nearest even number. For example, 0.5 rounds to 0, and 1.5 rounds to 2.
Comparison Constants

Since these constants are built into VBScript, you don't have to define them before using them. Use them anywhere in your code to represent the values shown for each.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vbBinaryCompare</td>
<td>0</td>
<td>Perform a binary comparison.</td>
</tr>
<tr>
<td>vbTextCompare</td>
<td>1</td>
<td>Perform a textual comparison.</td>
</tr>
</tbody>
</table>
Operator

See Also

Description

Forces string concatenation of two expressions.

Syntax

result = expression1 & expression2

The & operator syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>Any variable.</td>
</tr>
<tr>
<td>expression1</td>
<td>Any expression.</td>
</tr>
<tr>
<td>expression2</td>
<td>Any expression.</td>
</tr>
</tbody>
</table>

Remarks

Whenever an expression is not a string, it is converted to a String subtype. If both expressions are Null, result is also Null. However, if only one expression is Null, that expression is treated as a zero-length string ("") when concatenated with the other expression. Any expression that is Empty is also treated as a zero-length string.
**Const Statement**

See Also

---

### Description

Declares constants for use in place of literal values.

### Syntax

```
[Public | Private] Const constname = expression
```

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

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Public</strong></td>
<td>Optional. Keyword used at [script level] to declare constants that are available to all procedures in all scripts. Not allowed in procedures.</td>
</tr>
<tr>
<td><strong>Private</strong></td>
<td>Optional. Keyword used at script level to declare constants that are available only within the script where the declaration is made. Not allowed in procedures.</td>
</tr>
<tr>
<td>constname</td>
<td>Required. Name of the constant; follows standard [variable] naming conventions.</td>
</tr>
<tr>
<td>expression</td>
<td>Required. Literal or other constant, or any combination that includes all arithmetic or logical operators except [Is].</td>
</tr>
</tbody>
</table>

### Remarks

Constants are public by default. Within procedures, constants are always private; their visibility can't be changed. Within a script, the default visibility of a script-level constant can be changed using the **Private**
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 VBScript functions (such as **Chr**) in constant declarations. By definition, they can't be constants. You also can't create a constant from any expression that involves an operator, that is, only simple constants are allowed. Constants declared in a **Sub** or **Function** procedure are local to that procedure. A constant declared outside a procedure is defined throughout the script in which it is declared. You can use constants anywhere you can use an expression. The following code illustrates the use of the **Const** statement:

```
Const MyVar = 459    ' Cons
Private Const MyString = "HELP" ' Decl.
Const MyStr = "Hello", MyNumber = 3.4567    '      
```

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

Description

Returns the cosine of an angle.

Syntax

`Cos(number)`

The `number` argument can be any valid numeric expression that expresses an angle in radians.

Remarks

The `Cos` function takes an angle and returns the ratio of two sides of a right triangle. The ratio is the length of the side adjacent to the angle divided by the length of the hypotenuse. The result lies in the range -1 to 1.

To convert degrees to radians, multiply degrees by π/180. To convert radians to degrees, multiply radians by 180/π.

The following example uses the `Cos` function to return the cosine of an angle:

```
Dim MyAngle, MySecant
MyAngle = 1.3  ' Define angle in radians.
MySecant = 1 / Cos(MyAngle)  ' Calculate secant.
```
CreateObject Function

See Also

Description

Creates and returns a reference to an Automation object.

Syntax

CreateObject(servername.typename [, location])

The CreateObject function syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>servername</td>
<td>Required. The name of the application providing the object.</td>
</tr>
<tr>
<td>typename</td>
<td>Required. The type or class of the object to create.</td>
</tr>
<tr>
<td>location</td>
<td>Optional. The name of the network server where the object is to be created.</td>
</tr>
<tr>
<td></td>
<td>This feature is available in version 5.1 or later.</td>
</tr>
</tbody>
</table>

Remarks

Automation servers provide at least one type of object. For example, a word-processing application may provide an application object, a document object, and a toolbar object.

To create an Automation object, assign the object returned by CreateObject to an object variable:

Dim ExcelSheet
Set ExcelSheet = CreateObject("Excel.Sheet")
This code starts the application that creates the object (in this case, a Microsoft Excel spreadsheet). Once an object is created, refer to it in code using the object variable you defined. As shown in the following example, you can access properties and methods of the new object using the object variable, ExcelSheet, and other Excel objects, including the Application object and the ActiveSheet.Cells collection:

' Make Excel visible through the Application object.
ExcelSheet.Application.Visible = True
' Place some text in the first cell of the sheet.
ExcelSheet.ActiveSheet.Cells(1,1).Value = "This is column A, row 1"
' Save the sheet.
ExcelSheet.SaveAs "C:\DOCS\TEST.XLS"
' Close Excel with the Quit method on the Application object.
ExcelSheet.Application.Quit
' Release the object variable.
Set ExcelSheet = Nothing

Creating an object on a remote server can only be accomplished when Internet security is turned off. You can create an object on a remote networked computer by passing the name of the computer to the servername argument of CreateObject. That name is the same as the machine name portion of a sharename. For a network share named "\myserver\public", the servername is "myserver". In addition, you can specify servername using DNS format or an IP address.

The following code returns the version number of an instance of Excel running on a remote network computer named "myserver":

Function GetVersion
    Dim XLApp
    Set XLApp = CreateObject("Excel.Application", "myserver")
    ' Code to access the version number...
End Function
GetVersion = XLApp.Version
End Function

An error occurs if the specified remote server does not exist or cannot be found.
CSng Function

See Also

Description

Returns an expression that has been converted to a Variant of subtype Single.

Syntax

CSng(expression)

The expression argument is any valid expression.

Remarks

In general, you can document your code using the data type conversion functions to show that the result of some operation should be expressed as a particular data type rather than the default data type. For example, use CDbl or CSng to force double-precision or single-precision arithmetic in cases where currency or integer arithmetic normally would occur.

Use the CSng function to provide internationally aware conversions from any other data type to a Single subtype. For example, different decimal separators are properly recognized depending on the locale setting of your system, as are different thousand separators.

If expression lies outside the acceptable range for the Single subtype, an error occurs.

The following example uses the CSng function to convert a value to a
Single:

Dim MyDouble1, MyDouble2, MySingle1, MySingle2
MyDouble1 = 75.3421115: MyDouble2 = 75.3421555
MySingle1 = CSng(MyDouble1) ' MySingle1 contains 75.34211.
MySingle2 = CSng(MyDouble2) '
CStr

Function

See Also

Description

Returns an expression that has been converted to a Variant of subtype String.

Syntax

CStr(expression)

The expression argument is any valid expression.

Remarks

In general, you can document your code using the data type conversion functions to show that the result of some operation should be expressed as a particular data type rather than the default data type. For example, use CStr to force the result to be expressed as a String.

You should use the CStr function instead of Str to provide internationally aware conversions from any other data type to a String subtype. For example, different decimal separators are properly recognized depending on the locale setting of your system.

The data in expression determines what is returned according to the following table:

<table>
<thead>
<tr>
<th>If</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td></td>
</tr>
</tbody>
</table>


### CStr returns

<table>
<thead>
<tr>
<th>expression is</th>
<th>CStr returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>A String containing <strong>True</strong> or <strong>False</strong>.</td>
</tr>
<tr>
<td>Date</td>
<td>A String containing a date in the short-date format of your system.</td>
</tr>
<tr>
<td>Null</td>
<td>A run-time error.</td>
</tr>
<tr>
<td>Empty</td>
<td>A zero-length <strong>String</strong> (&quot;&quot;).</td>
</tr>
<tr>
<td>Error</td>
<td>A String containing the word Error followed by the error number.</td>
</tr>
<tr>
<td>Other numeric</td>
<td>A String containing the number.</td>
</tr>
</tbody>
</table>

The following example uses the **CStr** function to convert a numeric value to a **String**:

```vbnet
Dim MyDouble, MyString
MyDouble = 437.324 ' MyDouble is a Double
MyString = CStr(MyDouble) ' MyString contains
```
Since these constants are built into VBScript, you don't have to define them before using them. Use them anywhere in your code to represent the values shown for each.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vbSunday</td>
<td>1</td>
<td>Sunday</td>
</tr>
<tr>
<td>vbMonday</td>
<td>2</td>
<td>Monday</td>
</tr>
<tr>
<td>vbTuesday</td>
<td>3</td>
<td>Tuesday</td>
</tr>
<tr>
<td>vbWednesday</td>
<td>4</td>
<td>Wednesday</td>
</tr>
<tr>
<td>vbThursday</td>
<td>5</td>
<td>Thursday</td>
</tr>
<tr>
<td>vbFriday</td>
<td>6</td>
<td>Friday</td>
</tr>
<tr>
<td>vbSaturday</td>
<td>7</td>
<td>Saturday</td>
</tr>
<tr>
<td>vbUseSystem</td>
<td>0</td>
<td>Use the date format contained in the regional settings for your computer.</td>
</tr>
<tr>
<td>vbUseSystemDayOfWeek</td>
<td>0</td>
<td>Use the day of the week specified in your system settings for the first day of the week.</td>
</tr>
<tr>
<td>vbFirstJan1</td>
<td>1</td>
<td>Use the week in which January 1 occurs (default).</td>
</tr>
<tr>
<td>vbFirstFourDays</td>
<td>2</td>
<td>Use the first week that has at least four days in the new year.</td>
</tr>
</tbody>
</table>
Use the first full week of the year.
Since these constants are built into VBScript, you don't have to define them before using them. Use them anywhere in your code to represent the values shown for each.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vbGeneralDate</td>
<td>0</td>
<td>Display a date and/or time. For real numbers, display a date and time. If there is no fractional part, display only a date. If there is no integer part, display time only. Date and time display is determined by your system settings.</td>
</tr>
<tr>
<td>vbLongDate</td>
<td>1</td>
<td>Display a date using the long date format specified in your computer's regional settings.</td>
</tr>
<tr>
<td>vbShortDate</td>
<td>2</td>
<td>Display a date using the short date format specified in your computer's regional settings.</td>
</tr>
<tr>
<td>vbLongTime</td>
<td>3</td>
<td>Display a time using the long time format specified in your computer's regional settings.</td>
</tr>
<tr>
<td>vbShortTime</td>
<td>4</td>
<td>Display a time using the short time format specified in your computer's regional settings.</td>
</tr>
</tbody>
</table>
Date Function

Description

Returns the current system date.

Syntax

    Date

Remarks

The following example uses the Date function to return the current system date:

    Dim MyDate
    MyDate = Date  ' MyDate contains the current system date.
DateAdd Function

Description

Returns a date to which a specified time interval has been added.

Syntax

**DateAdd**(interval, number, date)

The **DateAdd** function syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>interval</td>
<td>Required. <strong>String expression</strong> that is the interval you want to add. See Settings section for values.</td>
</tr>
<tr>
<td>number</td>
<td>Required. <strong>Numeric expression</strong> that is the number of interval you want to add. The numeric expression can either be positive, for dates in the future, or negative, for dates in the past.</td>
</tr>
<tr>
<td>date</td>
<td>Required. <strong>Variant</strong> or literal representing the date to which <em>interval</em> is added.</td>
</tr>
</tbody>
</table>

Settings

The *interval* argument can have the following values:

<table>
<thead>
<tr>
<th>Setting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>yyyy</td>
<td>Year</td>
</tr>
<tr>
<td>q</td>
<td>Quarter</td>
</tr>
<tr>
<td>m</td>
<td>Month</td>
</tr>
<tr>
<td>y</td>
<td>Day of year</td>
</tr>
<tr>
<td>d</td>
<td>Day</td>
</tr>
</tbody>
</table>
You can use the **DateAdd** function to add or subtract a specified time interval from a date. For example, you can use **DateAdd** to calculate a date 30 days from today or a time 45 minutes from now. To add days to *date*, you can use Day of Year ("y"), Day ("d"), or Weekday ("w").

The **DateAdd** function won't return an invalid date. The following example adds one month to January 31:

```
NewDate = DateAdd("m", 1, "31-Jan-95")
```

In this case, **DateAdd** returns 28-Feb-95, not 31-Feb-95. If *date* is 31-Jan-96, it returns 29-Feb-96 because 1996 is a leap year.

If the calculated date would precede the year 100, an error occurs.

If number isn't a **Long** value, it is rounded to the nearest whole number before being evaluated.
DateDiff Function

See Also

Description

Returns the number of intervals between two dates.

Syntax

DateDiff(interval, date1, date2 [,firstdayofweek[, firstweekofyear]])

The DateDiff function syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>interval</td>
<td>Required. String expression that is the interval you want to use to calculate the differences between date1 and date2. See Settings section for values.</td>
</tr>
<tr>
<td>date1, date2</td>
<td>Required. Date expressions. Two dates you want to use in the calculation.</td>
</tr>
<tr>
<td>firstdayofweek</td>
<td>Optional. Constant that specifies the day of the week. If not specified, Sunday is assumed. See Settings section for values.</td>
</tr>
<tr>
<td>firstweekofyear</td>
<td>Optional. Constant that specifies the first week of the year. If not specified, the first week is assumed to be the week in which January 1 occurs. See Settings section for values.</td>
</tr>
</tbody>
</table>

Settings

The interval argument can have the following values:

<table>
<thead>
<tr>
<th>Setting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>yyyy</td>
<td>Year</td>
</tr>
</tbody>
</table>
The `firstdayofweek` argument can have the following values:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>vbUseSystem</code></td>
<td>0</td>
<td>Use National Language Support (NLS) API setting.</td>
</tr>
<tr>
<td><code>vbSunday</code></td>
<td>1</td>
<td>Sunday (default)</td>
</tr>
<tr>
<td><code>vbMonday</code></td>
<td>2</td>
<td>Monday</td>
</tr>
<tr>
<td><code>vbTuesday</code></td>
<td>3</td>
<td>Tuesday</td>
</tr>
<tr>
<td><code>vbWednesday</code></td>
<td>4</td>
<td>Wednesday</td>
</tr>
<tr>
<td><code>vbThursday</code></td>
<td>5</td>
<td>Thursday</td>
</tr>
<tr>
<td><code>vbFriday</code></td>
<td>6</td>
<td>Friday</td>
</tr>
<tr>
<td><code>vbSaturday</code></td>
<td>7</td>
<td>Saturday</td>
</tr>
</tbody>
</table>

The `firstweekofyear` argument can have the following values:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>vbUseSystem</code></td>
<td>0</td>
<td>Use National Language Support (NLS) API setting.</td>
</tr>
<tr>
<td><code>vbFirstJan1</code></td>
<td>1</td>
<td>Start with the week in which January 1 occurs (default).</td>
</tr>
<tr>
<td><code>vbFirstFourDays</code></td>
<td>2</td>
<td>Start with the week that has at least four days in the new year.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Start with the first full week of the new year.</td>
</tr>
</tbody>
</table>
You can use the **DateDiff** function to determine how many specified time intervals exist between two dates. For example, you might use **DateDiff** to calculate the number of days between two dates, or the number of weeks between today and the end of the year.

To calculate the number of days between `date1` and `date2`, you can use either Day of year ("y") or Day ("d"). When `interval` is Weekday ("w"), **DateDiff** returns the number of weeks between the two dates. If `date1` falls on a Monday, **DateDiff** counts the number of Mondays until `date2`. It counts `date2` but not `date1`. If `interval` is Week ("ww"), however, the **DateDiff** function returns the number of calendar weeks between the two dates. It counts the number of Sundays between `date1` and `date2`. **DateDiff** counts `date2` if it falls on a Sunday; but it doesn't count `date1`, even if it does fall on a Sunday.

If `date1` refers to a later point in time than `date2`, the **DateDiff** function returns a negative number.

The `firstdayofweek` argument affects calculations that use the "w" and "ww" interval symbols.

If `date1` or `date2` is a date literal, the specified year becomes a permanent part of that date. However, if `date1` or `date2` is enclosed in quotation marks (" ") and you omit the year, the current year is inserted in your code each time the `date1` or `date2` expression is evaluated. This makes it possible to write code that can be used in different years.

When comparing December 31 to January 1 of the immediately succeeding year, **DateDiff** for Year ("yyyy") returns 1 even though only a day has elapsed.

The following example uses the **DateDiff** function to display the number of days between a given date and today:

```vba
Function DiffADate(theDate)
    DiffADate = "Days from today: " & DateDiff("d", Now, theDate 
End Function
```
DatePart Function

See Also

Description

Returns the specified part of a given date.

Syntax

**DatePart**(*interval, date[, firstdayofweek[, firstweekofyear]])

The **DatePart** function syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>interval</em></td>
<td>Required. <strong>String expression</strong> that is the interval of time you want to return. See Settings section for values.</td>
</tr>
<tr>
<td><em>date</em></td>
<td>Required. Date expression you want to evaluate.</td>
</tr>
<tr>
<td><em>firstdayofweek</em></td>
<td>Optional. Constant that specifies the day of the week. If not specified, Sunday is assumed. See Settings section for values.</td>
</tr>
<tr>
<td><em>firstweekofyear</em></td>
<td>Optional. Constant that specifies the first week of the year. If not specified, the first week is assumed to be the week in which January 1 occurs. See Settings section for values.</td>
</tr>
</tbody>
</table>

Settings

The *interval* argument can have the following values:

<table>
<thead>
<tr>
<th>Setting</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>yyyy</td>
<td>Year</td>
</tr>
<tr>
<td><em>q</em></td>
<td>Quarter</td>
</tr>
</tbody>
</table>
The `firstdayofweek` argument can have the following values:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>vbUseSystem</code></td>
<td>0</td>
<td>Use National Language Support (NLS) API setting.</td>
</tr>
<tr>
<td><code>vbSunday</code></td>
<td>1</td>
<td>Sunday (default)</td>
</tr>
<tr>
<td><code>vbMonday</code></td>
<td>2</td>
<td>Monday</td>
</tr>
<tr>
<td><code>vbTuesday</code></td>
<td>3</td>
<td>Tuesday</td>
</tr>
<tr>
<td><code>vbWednesday</code></td>
<td>4</td>
<td>Wednesday</td>
</tr>
<tr>
<td><code>vbThursday</code></td>
<td>5</td>
<td>Thursday</td>
</tr>
<tr>
<td><code>vbFriday</code></td>
<td>6</td>
<td>Friday</td>
</tr>
<tr>
<td><code>vbSaturday</code></td>
<td>7</td>
<td>Saturday</td>
</tr>
</tbody>
</table>

The `firstweekofyear` argument can have the following values:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>vbUseSystem</code></td>
<td>0</td>
<td>Use National Language Support (NLS) API setting.</td>
</tr>
<tr>
<td><code>vbFirstJan1</code></td>
<td>1</td>
<td>Start with the week in which January 1 occurs (default).</td>
</tr>
<tr>
<td><code>vbFirstFourDays</code></td>
<td>2</td>
<td>Start with the week that has at least four days in the new year.</td>
</tr>
<tr>
<td><code>vbFirstFullWeek</code></td>
<td>3</td>
<td>Start with the first full week of the new year.</td>
</tr>
</tbody>
</table>
Remarks

You can use the **DatePart** function to evaluate a date and return a specific interval of time. For example, you might use **DatePart** to calculate the day of the week or the current hour.

The *firstdayofweek* argument affects calculations that use the "w" and "ww" interval symbols.

If *date* is a date literal, the specified year becomes a permanent part of that date. However, if *date* is enclosed in quotation marks (" "), and you omit the year, the current year is inserted in your code each time the *date* expression is evaluated. This makes it possible to write code that can be used in different years.

This example takes a date and, using the **DatePart** function, displays the quarter of the year in which it occurs.

```vba
Function GetQuarter(TheDate)
    GetQuarter = DatePart("q", TheDate)
End Function
```
**DateSerial Function**

**See Also**

---

**Description**

Returns a **Variant** of subtype **Date** for a specified year, month, and day.

**Syntax**

\[ \text{DateSerial}(year, \text{month}, \text{day}) \]

The **DateSerial** function syntax has these arguments:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>year</td>
<td>Number between 100 and 9999, inclusive, or a numeric expression.</td>
</tr>
<tr>
<td>month</td>
<td>Any numeric expression.</td>
</tr>
<tr>
<td>day</td>
<td>Any numeric expression.</td>
</tr>
</tbody>
</table>

**Remarks**

To specify a date, such as December 31, 1991, the range of numbers for each **DateSerial** argument should be in the accepted range for the unit; that is, 1–31 for days and 1–12 for months. However, you can also specify relative dates for each argument using any numeric expression that represents some number of days, months, or years before or after a certain date.

The following example uses numeric expressions instead of absolute date
numbers. Here the **DateSerial** function returns a date that is the day before the first day (1 – 1) of two months before August (8 – 2) of 10 years before 1990 (1990 – 10); in other words, May 31, 1980.

```
Dim MyDate1, MyDate2
MyDate1 = DateSerial(1970, 1, 1) ' Returns January 1, 1970.
MyDate2 = DateSerial(1990 - 10, 8 - 2, 1 - 1) ' Returns May 31, 1980.
```

For the *year* argument, values between 0 and 99, inclusive, are interpreted as the years 1900–1999. For all other *year* arguments, use a complete four-digit year (for example, 1800).

When any argument exceeds the accepted range for that argument, it increments to the next larger unit as appropriate. For example, if you specify 35 days, it is evaluated as one month and some number of days, depending on where in the year it is applied. However, if any single argument is outside the range -32,768 to 32,767, or if the date specified by the three arguments, either directly or by expression, falls outside the acceptable range of dates, an error occurs.
DateValue Function

See Also

Description

Returns a Variant of subtype Date.

Syntax

```
DateValue(date)
```

The date argument is normally a string expression representing a date from January 1, 100 through December 31, 9999. However, date can also be any expression that can represent a date, a time, or both a date and time, in that range.

Remarks

If the date argument includes time information, DateValue doesn't return it. However, if date includes invalid time information (such as "89:98"), an error occurs.

If date is a string that includes only numbers separated by valid date separators, DateValue recognizes the order for month, day, and year according to the short date format you specified for your system. DateValue also recognizes unambiguous dates that contain month names, either in long or abbreviated form. For example, in addition to recognizing 12/30/1991 and 12/30/91, DateValue also recognizes December 30, 1991 and Dec 30, 1991.

If the year part of date is omitted, DateValue uses the current year from your computer's system date.

The following example uses the DateValue function to convert a string to a date. You can also use date literals to directly assign a date to a Variant variable, for example, MyDate = #9/11/63#.

```
Dim MyDate
MyDate = DateValue("September 11, 1963")  ' Re...
Day Function

See Also

Description

Returns a whole number between 1 and 31, inclusive, representing the day of the month.

Syntax

Day(date)

The date argument is any expression that can represent a date. If date contains Null, Null is returned.

The following example uses the Day function to obtain the day of the month from a specified date:

```vbscript
Dim MyDay
```
Description Property

See Also Applies to

Description

Returns or sets a descriptive string associated with an error.

Syntax

`object.Description [= stringexpression]`

The `Description` property syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>object</code></td>
<td>Always the <code>Err</code> object.</td>
</tr>
<tr>
<td><code>stringexpression</code></td>
<td>A <code>string expression</code> containing a description of the error.</td>
</tr>
</tbody>
</table>

Remarks

The `Description` property consists of a short description of the error. Use this property to alert the user to an error that you can't or don't want to handle. When generating a user-defined error, assign a short description of your error to this property. If `Description` isn't filled in, and the value of `Number` corresponds to a VBScript run-time error, the descriptive string associated with the error is returned.

On Error Resume Next

`Err.Raise 6 ' Raise an overflow error.
Dim Statement

**Description**

Declares variables and allocates storage space.

**Syntax**

```
Dim varname[[subscripts]][, varname[[subscripts]]] . . .
```

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

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>varname</td>
<td>Name of the variable; follows standard variable naming conventions.</td>
</tr>
<tr>
<td>subscripts</td>
<td>Dimensions of an array variable; up to 60 multiple dimensions may be declared. The <code>subscripts</code> argument uses the following syntax:</td>
</tr>
<tr>
<td></td>
<td><code>upperbound [,upperbound] . . .</code></td>
</tr>
<tr>
<td></td>
<td>The lower bound of an array is always zero.</td>
</tr>
</tbody>
</table>

**Remarks**

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

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 redefine a dimension for an array variable whose size was explicitly specified in a **Dim** statement, an error occurs.

---

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

---

The following examples illustrate the use of the **Dim** statement:

```vbnet
Dim Names(9) ' Declare an array with 10 elements.
Dim Names()  ' Declare a dynamic array.
Dim MyVar, MyNum ' Declare two variables.
```
Operator

Description

Divides two numbers and returns a floating-point result.

Syntax

\[ \text{result} = \text{number1}/\text{number2} \]

The / operator syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>Any numeric variable.</td>
</tr>
<tr>
<td>number1</td>
<td>Any numeric expression.</td>
</tr>
<tr>
<td>number2</td>
<td>Any numeric expression.</td>
</tr>
</tbody>
</table>

Remarks

If one or both expressions are Null expressions, result is Null. Any expression that is Empty is treated as 0.
Do...Loop Statement

Description

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 [{While | Until} condition]
```

Or, you can use this syntax:

```
Do
    [statements]
    [Exit Do]
    [statements]
Loop [{While | Until} condition]
```

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

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>condition</td>
<td><strong>Numeric</strong> or <strong>string expression</strong> that is <strong>True</strong> or <strong>False</strong>. If <strong>condition</strong> is <strong>Null</strong>, <strong>condition</strong> is treated as <strong>False</strong>.</td>
</tr>
<tr>
<td>statements</td>
<td>One or more statements that are repeated while or until <strong>condition</strong> is <strong>True</strong>.</td>
</tr>
</tbody>
</table>

Remarks

The **Exit Do** can only be used within a **Do...Loop** control structure to
provide an alternate way to exit a **Do...Loop**. Any number of **Exit Do** statements may be placed anywhere in the **Do...Loop**. Often used with the evaluation of some condition (for example, **If...Then**), **Exit Do** transfers control to the statement immediately following the **Loop**.

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

The following examples illustrate use of the **Do...Loop** statement:

```vbnet
Do Until DefResp = vbNo
    MyNum = Int (6 * Rnd + 1) ' Generate a random integer between
    DefResp = MsgBox (MyNum & " Do you want another number?
Loop

Dim Check, Counter
Check = True: Counter = 0 ' Initialize variables.
Do ' Outer loop.
    Do While Counter < 20 ' Inner loop.
        Counter = Counter + 1 ' Increment Counter.
        If Counter = 10 Then ' If condition is True...
            Check = False ' set value of flag to False.
            Exit Do ' Exit inner loop.
        End If
        Loop
    Loop Until Check = False ' Exit outer loop immediately.
```
The **Empty** keyword is used to indicate an uninitialized variable value. This is not the same thing as **Null**.
Eqv Operator

Description

Performs a logical equivalence on two expressions.

Syntax

\[
\text{result} = \text{expression1} \text{ Eqv expression2}
\]

The Eqv operator syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>Any numeric variable.</td>
</tr>
<tr>
<td>expression1</td>
<td>Any expression.</td>
</tr>
<tr>
<td>expression2</td>
<td>Any expression.</td>
</tr>
</tbody>
</table>

Remarks

If either expression is Null, result is also Null. When neither expression is Null, result is determined according to the following table:

<table>
<thead>
<tr>
<th>If expression1 is</th>
<th>And expression2 is</th>
<th>The result is</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>True</td>
</tr>
</tbody>
</table>

The Eqv operator performs a bitwise comparison of identically positioned bits in two numeric expressions and sets the corresponding bit in result according to the following table:

<table>
<thead>
<tr>
<th>And bit in expression2 is</th>
<th>The result is</th>
</tr>
</thead>
<tbody>
<tr>
<td>If bit in expression1 is</td>
<td></td>
</tr>
<tr>
<td>-------------------------</td>
<td>---</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
**Erase Statement**

**See Also**

**Description**

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

**Syntax**

```
Erase array
```

The *array* argument is the name of the array *variable* to be erased.

**Remarks**

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

<table>
<thead>
<tr>
<th>Type of array</th>
<th>Effect of Erase on fixed-array elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed numeric array</td>
<td>Sets each element to zero.</td>
</tr>
<tr>
<td>Fixed string array</td>
<td>Sets each element to zero-length (&quot;&quot;).</td>
</tr>
<tr>
<td>Array of objects</td>
<td>Sets each element to the special value <em>Nothing</em>.</td>
</tr>
</tbody>
</table>

*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.

The following example illustrates the use of the *Erase* statement:

```
Dim NumArray(9)
```
Dim DynamicArray()
ReDim DynamicArray(9) ' Allocate storage space.
Erase NumArray       ' Each element is reinitialized
Erase DynamicArray   ' Free memory used by array
Err Object

Description

Contains information about run-time errors. Accepts the Raise and Clear methods for generating and clearing run-time errors.

Remarks

The Err object is an intrinsic object with global scope — there is no need to create an instance of it in your code. The properties of the Err object are set by the generator of an error — Visual Basic, an Automation object, or the VBScript programmer.

The default property of the Err object is Number. Err.Number contains an integer and can be used by an Automation object to return an SCODE.

When a run-time error occurs, the properties of the Err object are filled with information that uniquely identifies the error and information that can be used to handle it. To generate a run-time error in your code, use the Raise method.

The Err object's properties are reset to zero or zero-length strings (""") after an On Error Resume Next statement. The Clear method can be used to explicitly reset Err.

The following example illustrates use of the Err object:

```
On Error Resume Next
Err.Raise 6 ' Raise an overflow error.
MsgBox ("Error # " & CStr(Err.Number) & " " & Err.Description)
Err.Clear ' Clear the error.
```
Eval Function

Description

Evaluates an expression and returns the result.

Syntax

[result = ]Eval(expression)

The Eval function syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>Optional. Variable to which return value assignment is made. If result is not specified, consider using the Execute statement instead.</td>
</tr>
<tr>
<td>expression</td>
<td>Required. String containing any legal VBScript expression.</td>
</tr>
</tbody>
</table>

Remarks

In VBScript, \(x = y\) can be interpreted two ways. The first is as an assignment statement, where the value of \(y\) is assigned to \(x\). The second interpretation is as an expression that tests if \(x\) and \(y\) have the same value. If they do, result is True; if they are not, result is False. The Eval method always uses the second interpretation, whereas the Execute statement always uses the first.

Note In Microsoft® JScript, no confusion exists between assignment and comparison, because the assignment operator (=) is different from the comparison operator (==).

The following example illustrates the use of the Eval function:
Sub GuessANumber
    Dim Guess, RndNum
    RndNum = Int((100) * Rnd(1) + 1)
    Guess = CInt(InputBox("Enter your guess:",",0))
    Do
        If Eval("Guess = RndNum") Then
            MsgBox "Congratulations! You guessed it!"
            Exit Sub
        Else
            Guess = CInt(InputBox("Sorry! Try again.",",0))
        End If
    Loop Until Guess = 0
End Sub
Execute Method

Description

Executes a regular expression search against a specified string.

Syntax

object.Execute(string)

The **Execute** method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>Required. Always the name of a RegExp object.</td>
</tr>
<tr>
<td>string</td>
<td>Required. The text string upon which the regular expression is executed.</td>
</tr>
</tbody>
</table>

Remarks

The actual pattern for the regular expression search is set using the **Pattern** property of the **RegExp** object.

The **Execute** method returns a **Matches** collection containing a **Match** object for each match found in *string*. **Execute** returns an empty **Matches** collection if no match is found.

The following code illustrates the use of the **Execute** method:

```vbscript
Function RegExpTest(patrn, strng)
    ' Code to illustrate the use of Execute method
End Function
```
Dim regEx, Match, Matches  ' Create variable.
Set regEx = New RegExp  ' Create a regular expression
regEx.Pattern = patrn  ' Set pattern.
regEx.IgnoreCase = True  ' Set case insensitivity.
regEx.Global = True  ' Set global applicability.
Set Matches = regEx.Execute(strng)  ' Execute search.
For Each Match in Matches  ' Iterate Matches collection.
  RetStr = RetStr & "Match found at position 
  RetStr = RetStr & Match.FirstIndex & ". Match Value is 
  RetStr = RetStr & Match.Value & "." & vbCRLF
Next
RegExpTest = RetStr
End Function

MsgBox(RegExpTest("is.", "IS1 is2 IS3 is4"))
Execute Statement

See Also

Description

Executes one or more specified statements.

Syntax

**Execute statement**

The required *statement* argument is a *string expression* containing one or more statements for execution. Include multiple statements in the *statement* argument, using colons or embedded line breaks to separate them.

Remarks

In VBScript, \( x = y \) can be interpreted two ways. The first is as an assignment statement, where the value of \( y \) is assigned to \( x \). The second interpretation is as an *expression* that tests if \( x \) and \( y \) have the same value. If they do, *result* is **True**; if they are not, *result* is **False**. The **Execute** statement always uses the first interpretation, whereas the **Eval** method always uses the second.

**Note** In Microsoft® JScript, no confusion exists between assignment and comparison, because the assignment operator (=) is different from the comparison operator (==).

The context in which the **Execute** statement is invoked determines what objects and *variables* are available to the code being run. In-scope objects and variables are available to code running in an **Execute** statement. However, it is important to understand that if you execute code that creates a **procedure**, that procedure does not inherit the **scope** of the procedure in which it occurred.

Like any procedure, the new procedure's scope is global, and it inherits everything in the global scope. Unlike any other procedure, its context is not global scope, so it can only be executed in the context of the procedure where the **Execute** statement occurred. However, if the same **Execute** statement is invoked outside of a procedure (i.e., in global scope), not only does it inherit everything in global scope, but it can also be called from anywhere, since its context is global. The following example illustrates this behavior:
Dim X ' Declare X in global scope.
X = "Global" ' Assign global X a value.
Sub Proc1 ' Declare procedure.
    Dim X ' Declare X in local scope.
    X = "Local" ' Assign local X a value.
    ' The Execute statement here creates a
    ' procedure that, when invoked, prints X.
    ' It print the global X because Proc2
    ' inherits everything in global scope.
    Execute "Sub Proc2: Print X: End Sub"
Print Eval("X") ' Print local X.
Proc2 ' Invoke Proc2 in Proc1's scope.
End Sub
Proc2 ' This line causes an error since
    ' Proc2 is unavailable outside Proc1.
Proc1 ' Invoke Proc1.
    Execute "Sub Proc2: Print X: End Sub"
Proc2 ' This invocation succeeds because Proc2
    ' is now available globally.

The following example shows how the Execute statement can be rewritten
so you don't have to enclose the entire procedure in the quotation marks:

    S = "Sub Proc2" & vbCrLf
    S = S & " Print X" & vbCrLf
    S = S & "End Sub"
    Execute S
Exit Statement

See Also

Description

Exits a block of **Do...Loop, For...Next, Function, or Sub** code.

Syntax

**Exit Do**

Exit For

Exit Function

Exit Property

Exit Sub

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

<table>
<thead>
<tr>
<th>Statement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Exit Do</strong></td>
<td>Provides a way to exit a <strong>Do...Loop</strong> statement. It can be used only inside a <strong>Do...Loop</strong> statement. <strong>Exit Do</strong> transfers control to the statement following the <strong>Loop</strong> statement. When used within nested <strong>Do...Loop</strong> statements, <strong>Exit Do</strong> transfers control to the loop that is one nested level above the loop where it occurs.</td>
</tr>
<tr>
<td><strong>Exit For</strong></td>
<td>Provides a way to exit a <strong>For</strong> loop. It can be used only in a <strong>For...Next</strong> or <strong>For Each...Next</strong> loop. <strong>Exit For</strong> transfers control to the statement following the <strong>Next</strong> statement. When used within nested <strong>For</strong> loops, <strong>Exit For</strong> transfers control to the loop that is one nested level above the loop where it occurs.</td>
</tr>
<tr>
<td></td>
<td>Immediately exits the <strong>Function</strong> procedure in which</td>
</tr>
</tbody>
</table>
Exit Function

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.

The following example illustrates the use of the Exit statement:

Sub RandomLoop
    Dim I, MyNum
    Do ' Set up infinite loop.
        For I = 1 To 1000 ' Loop 1000 times.
            MyNum = Int(Rnd * 100) ' Generate random number.
        Select Case MyNum ' Evaluate random number.
            Case 17: MsgBox "Case 17"
                Exit For ' If 17, exit For...Next.
            Case 29: MsgBox "Case 29"
                Exit Do ' If 29, exit Do...Loop.
            Case 54: MsgBox "Case 54"
                Exit Sub ' If 54, exit Sub procedure.
        End Select
    Next Loop
End Sub
Exp Function

See Also

Description

Returns e (the base of natural logarithms) raised to a power.

Syntax

\texttt{Exp(number)}

The \textit{number} argument can be any valid numeric expression.

Remarks

If the value of \textit{number} exceeds 709.782712893, an error occurs. The constant $e$ is approximately 2.718282.

\textbf{Note}  The \texttt{Exp} function complements the action of the \texttt{Log} function and is sometimes referred to as the antilogarithm.

The following example uses the \texttt{Exp} function to return $e$ raised to a power:

\begin{verbatim}
Dim MyAngle, MyHSin  ' Define angle in radians.
MyAngle = 1.3     ' Calculate hyperbolic sine.
MyHSin = (Exp(MyAngle) - Exp(-1 * MyAngle))
\end{verbatim}
Operator

See Also

Description

Raises a number to the power of an exponent.

Syntax

\[ result = number^{exponent} \]

The \(^\text{^}\) operator syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>Any numeric variable.</td>
</tr>
<tr>
<td>number</td>
<td>Any numeric expression.</td>
</tr>
<tr>
<td>exponent</td>
<td>Any numeric expression.</td>
</tr>
</tbody>
</table>

Remarks

\(\text{Number}\) can be negative only if \(\text{exponent}\) is an integer value. When more than one exponentiation is performed in a single expression, the \(^\text{^}\) operator is evaluated as it is encountered from left to right.

If either \(\text{number}\) or \(\text{exponent}\) is a Null expression, \(\text{result}\) is also Null.
See Also

Description

The **False** keyword has a value equal to 0.
**Filter**

**Function**

**See Also**

**Description**

Returns a zero-based `array` containing a subset of a string array based on a specified filter criteria.

**Syntax**

```
Filter(InputStrings, Value[, Include[, Compare]])
```

The `Filter` function syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>InputStrings</code></td>
<td>Required. One-dimensional array of strings to be searched.</td>
</tr>
<tr>
<td><code>Value</code></td>
<td>Required. String to search for.</td>
</tr>
<tr>
<td><code>Include</code></td>
<td>Optional. Boolean value indicating whether to return substrings that include or exclude <code>Value</code>. If <code>Include</code> is <code>True</code>, <code>Filter</code> returns the subset of the array that contains <code>Value</code> as a substring. If <code>Include</code> is <code>False</code>, <code>Filter</code> returns the subset of the array that does not contain <code>Value</code> as a substring.</td>
</tr>
<tr>
<td><code>Compare</code></td>
<td>Optional. Numeric value indicating the kind of string comparison to use. See Settings section for values.</td>
</tr>
</tbody>
</table>

**Settings**

The `Compare` argument can have the following values:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>vbBinaryCompare</th>
<th>0</th>
<th>Perform a binary comparison.</th>
</tr>
</thead>
<tbody>
<tr>
<td>vbTextCompare</td>
<td>1</td>
<td>Perform a textual comparison.</td>
</tr>
</tbody>
</table>

**Remarks**

If no matches of *Value* are found within *InputStrings*, **Filter** returns an empty array. An error occurs if *InputStrings* is **Null** or is not a one-dimensional array.

The array returned by the **Filter** function contains only enough elements to contain the number of matched items.

The following example uses the **Filter** function to return the array containing the search criteria "Mon":

```vbnet
Dim MyIndex
Dim MyArray (3)
MyArray(0) = "Sunday"
MyArray(1) = "Monday"
MyArray(2) = "Tuesday"
MyIndex = Filter(MyArray, "Mon") ' MyIndex(0) contains "Monday".
```
FirstIndex Property

Description

Returns the position in a search string where a match occurs.

Syntax

`object.FirstIndex`

The `object` argument is always a `Match` object.

Remarks

The `FirstIndex` property uses a zero-based offset from the beginning of the search string. In other words, the first character in the string is identified as character zero (0). The following code illustrates the use of the `FirstIndex` property:

```vbscript
Function RegExpTest(patrn, strng)
    Dim regEx, Match, Matches
    ' Create variable.
    Set regEx = New RegExp
    ' Create regular expression
    regEx.Pattern = patrn
    ' Set pattern.
    regEx.IgnoreCase = True
    ' Set case insensitivity.
    regEx.Global = True
    ' Set global applicability.
    Set Matches = regEx.Execute(strng)
    ' Execute search.
    For Each Match in Matches
        ' Iterate Matches collection.
        RetStr = RetStr & "Match " & I & " found at position "
        RetStr = RetStr & Match.FirstIndex & ". Match Value is "
        RetStr = RetStr & Match.Value & "." & vbCrLf
    Next Match
    ' Process Matches collection.
End Function
```
Next
    RegExpTest = RetStr
End Function

MsgBox(RegExpTest("is.", "IS1 is2 IS3 is4"))
Description

Returns the integer portion of a number.

Syntax

**Int**(number)

**Fix**(number)

The *number* argument can be any valid numeric expression. If *number* contains **Null**, **Null** is returned.

Remarks

Both **Int** and **Fix** remove the fractional part of *number* and return the resulting integer value.

The difference between **Int** and **Fix** is that if *number* is negative, **Int** returns the first negative integer less than or equal to *number*, whereas **Fix** returns the first negative integer greater than or equal to *number*. For example, **Int** converts -8.4 to -9, and **Fix** converts -8.4 to -8.

**Fix**(number) is equivalent to:

\[
\text{Sgn}(number) \times \text{Int}(\text{Abs}(number))
\]

The following examples illustrate how the **Int** and **Fix** functions return integer portions of numbers:

MyNumber = **Int**(99.8)   ' Returns 99.
MyNumber = **Fix**(99.2)   ' Returns 99.
MyNumber = **Int**(-99.8)  ' Returns -100.
MyNumber = Fix(-99.8)  ' Returns -99.
MyNumber = Int(-99.2)  ' Returns -100.
MyNumber = Fix(-99.2)  ' Returns -99.
For...Next Statement

Description

Repeats a group of statements a specified number of times.

Syntax

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

The `For...Next` statement syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>counter</td>
<td>Numeric variable used as a loop counter. The variable can't be an array element or an element of a user-defined type.</td>
</tr>
<tr>
<td>start</td>
<td>Initial value of <code>counter</code>.</td>
</tr>
<tr>
<td>end</td>
<td>Final value of <code>counter</code>.</td>
</tr>
<tr>
<td>step</td>
<td>Amount <code>counter</code> is changed each time through the loop. If not specified, <code>step</code> defaults to one.</td>
</tr>
<tr>
<td>statements</td>
<td>One or more statements between <code>For</code> and <code>Next</code> that are executed the specified number of times.</td>
</tr>
</tbody>
</table>

Remarks

The `step` argument can be either positive or negative. The value of the `step` argument determines loop processing as follows:
<table>
<thead>
<tr>
<th>Value</th>
<th>Loop executes if</th>
</tr>
</thead>
<tbody>
<tr>
<td>Positive or 0</td>
<td>counter &lt;= end</td>
</tr>
<tr>
<td>Negative</td>
<td>counter &gt;= end</td>
</tr>
</tbody>
</table>

Once the loop starts and 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.

Exit For can only be used within a For Each...Next or For...Next control structure to provide an alternate way to exit. Any number of Exit For statements may be placed anywhere in the loop. Exit For is often used with the evaluation 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
    Next
Next
```
For Each...Next Statement

See Also

Description

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:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>element</td>
<td>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 Automation object variable. For arrays, element can only be a Variant variable.</td>
</tr>
<tr>
<td>group</td>
<td>Name of an object collection or array.</td>
</tr>
<tr>
<td>statements</td>
<td>One or more statements that are executed on each item in group.</td>
</tr>
</tbody>
</table>

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*. As long as 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.

The **Exit For** can only be used within a **For Each...Next** or **For...Next** control structure to provide an alternate way to exit. Any number of **Exit For** statements may be placed anywhere in the loop. The **Exit For** is often used with the evaluation of some condition (for example, **If...Then**), 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 you had included it. If a **Next** statement is encountered before its corresponding **For** statement, an error occurs.

---

The following example illustrates use of the **For Each...Next** statement:

```vbscript
Function ShowFolderList(folderspec)
    Dim fso, f, f1, fc, s
    Set fso = CreateObject("Scripting.FileSystemObject")
    Set f = fso.GetFolder(folderspec)
    Set fc = f.Files
    For Each f1 in fc
        s = s & f1.name
        s = s & "<BR>"
    Next
    ShowFolderList = s
End Function
```
## FormatCurrency Function

### See Also

### Description

Returns an expression formatted as a currency value using the currency symbol defined in the system control panel.

### Syntax

```vbnet
FormatCurrency(Expression[, NumDigitsAfterDecimal [,IncludeLeadingDigit [,UseParensForNegativeNumbers [,GroupDigits]]]])
```

The `FormatCurrency` function syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expression</td>
<td>Required. Expression to be formatted.</td>
</tr>
<tr>
<td>NumDigitsAfterDecimal</td>
<td>Optional. Numeric value indicating how many places to the right of the decimal are displayed. Default value is -1, which indicates that the computer's regional settings are used.</td>
</tr>
<tr>
<td>IncludeLeadingDigit</td>
<td>Optional. Tristate constant that indicates whether or not a leading zero is displayed for fractional values. See Settings section for values.</td>
</tr>
<tr>
<td>UseParensForNegativeNumbers</td>
<td>Optional. Tristate constant that indicates whether or not to use parentheses for negative numbers.</td>
</tr>
<tr>
<td>GroupDigits</td>
<td>Optional. Tristate constant that indicates whether or not to use grouping for thousands.</td>
</tr>
</tbody>
</table>
UseParensForNegativeNumbers

Place negative values within parentheses. See Settings section for values.

GroupDigits

Optional. Tristate constant that indicates whether or not numbers are grouped using the group delimiter specified in the computer's regional settings. See Settings section for values.

Settings

The IncludeLeadingDigit, UseParensForNegativeNumbers, and GroupDigits arguments have the following settings:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TristateTrue</td>
<td>-1</td>
<td>True</td>
</tr>
<tr>
<td>TristateFalse</td>
<td>0</td>
<td>False</td>
</tr>
<tr>
<td>TristateUseDefault</td>
<td>-2</td>
<td>Use the setting from the computer's regional settings.</td>
</tr>
</tbody>
</table>

Remarks

When one or more optional arguments are omitted, values for omitted arguments are provided by the computer's regional settings. The position of the currency symbol relative to the currency value is determined by the system's regional settings.

Note All settings information comes from the Regional Settings Currency tab, except leading zero which comes from the Number tab.
The following example uses the **FormatCurrency** function to format the expression as a currency and assign it to MyCurrency:

```vbnet
Dim MyCurrency
MyCurrency = FormatCurrency(1000) ' MyCurrency
```
FormatDateTime Function

Description

Returns an expression formatted as a date or time.

Syntax

\textbf{FormatDateTime}(\text{Date[, NamedFormat]})

The \textbf{FormatDateTime} function syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{Date}</td>
<td>Required. Date expression to be formatted.</td>
</tr>
<tr>
<td>\text{NamedFormat}</td>
<td>Optional. Numeric value that indicates the date/time format used. If omitted, \text{vbGeneralDate} is used.</td>
</tr>
</tbody>
</table>

Settings

The \textit{NamedFormat} argument has the following settings:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{vbGeneralDate}</td>
<td>0</td>
<td>Display a date and/or time. If there is a date part, display it as a short date. If there is a time part, display it as a long time. If present, both parts are displayed.</td>
</tr>
<tr>
<td>\text{vbLongDate}</td>
<td>1</td>
<td>Display a date using the long date format specified in your computer's regional settings.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Display a date using the short date</td>
</tr>
<tr>
<td>vbShortDate</td>
<td>2</td>
<td>format specified in your computer's regional settings.</td>
</tr>
<tr>
<td>--------------</td>
<td>---</td>
<td>-------------------------------------------------------</td>
</tr>
<tr>
<td>vbLongTime</td>
<td>3</td>
<td>Display a time using the time format specified in your computer's regional settings.</td>
</tr>
<tr>
<td>vbShortTime</td>
<td>4</td>
<td>Display a time using the 24-hour format (hh:mm).</td>
</tr>
</tbody>
</table>

**Remarks**

The following example uses the **FormatDateTime** function to format the expression as a long date and assign it to MyDateTime:

```vbscript
Function GetCurrentDate
    ' FormatDateTime formats Date in long date.
    GetCurrentDate = FormatDateTime(Date, 1)
End Function
```
FormatNumber Function

See Also

Description

Returns an expression formatted as a number.

Syntax

FormatNumber(Expression [,NumDigitsAfterDecimal [,IncludeLeadingDigit [,UseParensForNegativeNumbers [,GroupDigits]]]])

The FormatNumber function syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expression</td>
<td>Required. Expression to be formatted.</td>
</tr>
<tr>
<td>NumDigitsAfterDecimal</td>
<td>Optional. Numeric value indicating how many places to the right of the decimal are displayed. Default value is -1, which indicates that the computer's regional settings are used.</td>
</tr>
<tr>
<td>IncludeLeadingDigit</td>
<td>Optional. Tristate constant that indicates whether or not a leading zero is displayed for fractional values. See Settings section for values.</td>
</tr>
<tr>
<td>UseParensForNegativeNumbers</td>
<td>Optional. Tristate constant that indicates whether or not to place negative values within</td>
</tr>
</tbody>
</table>
GroupDigits

Optional. Tristate constant that indicates whether or not numbers are grouped using the group delimiter specified in the control panel. See Settings section for values.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TristateTrue</td>
<td>-1</td>
<td>True</td>
</tr>
<tr>
<td>TristateFalse</td>
<td>0</td>
<td>False</td>
</tr>
<tr>
<td>TristateUseDefault</td>
<td>-2</td>
<td>Use the setting from the computer's regional settings.</td>
</tr>
</tbody>
</table>

Remarks

When one or more of the optional arguments are omitted, the values for omitted arguments are provided by the computer's regional settings.

Note  All settings information comes from the Regional Settings Number tab.

The following example uses the **FormatNumber** function to format a number to have four decimal places:

Function FormatNumberDemo
Dim MyAngle, MySecant, MyNumber

MyAngle = 1.3  ' Define angle in radians
MySecant = 1 / Cos(MyAngle)  ' Calculate secant

FormatNumberDemo = FormatNumber(MySecant)

End Function
# FormatPercent Function

## See Also

## Description

Returns an expression formatted as a percentage (multiplied by 100) with a trailing % character.

## Syntax

```plaintext
FormatPercent(Expression[,NumDigitsAfterDecimal [,IncludeLeadingDigit [,UseParensForNegativeNumbers [,GroupDigits]]]])
```

The `FormatPercent` function syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Expression</code></td>
<td>Required. Expression to be formatted.</td>
</tr>
<tr>
<td><code>NumDigitsAfterDecimal</code></td>
<td>Optional. Numeric value indicating how many places to the right of the decimal are displayed. Default value is -1, which indicates that the computer's regional settings are used.</td>
</tr>
<tr>
<td><code>IncludeLeadingDigit</code></td>
<td>Optional. Tristate constant that indicates whether or not a leading zero is displayed for fractional values. See Settings section for values.</td>
</tr>
<tr>
<td><code>UseParensForNegativeNumbers</code></td>
<td>Optional. Tristate constant that indicates whether or not to use parentheses for negative numbers.</td>
</tr>
<tr>
<td><code>GroupDigits</code></td>
<td>Optional. Tristate constant that indicates whether or not to use grouping characters.</td>
</tr>
</tbody>
</table>
**UseParensForNegativeNumbers**

Place negative values within parentheses. See Settings section for values.

**GroupDigits**

Optional. Tristate constant that indicates whether or not numbers are grouped using the group delimiter specified in the control panel. See Settings section for values.

## Settings

The `IncludeLeadingDigit`, `UseParensForNegativeNumbers`, and `GroupDigits` arguments have the following settings:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TristateTrue</td>
<td>-1</td>
<td>True</td>
</tr>
<tr>
<td>TristateFalse</td>
<td>0</td>
<td>False</td>
</tr>
<tr>
<td>TristateUseDefault</td>
<td>-2</td>
<td>Use the setting from the computer's regional settings.</td>
</tr>
</tbody>
</table>

## Remarks

When one or more optional arguments are omitted, the values for the omitted arguments are provided by the computer's regional settings.

**Note** All settings information comes from the Regional Settings Number tab.

The following example uses the `FormatPercent` function to format an expression as a percent:
Dim MyPercent
MyPercent = FormatPercent(2/32) ' MyPercent contains 6.25%.
Function Statement

See Also

Description

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

Syntax

```
[Public [Default] | Private] Function name [(arglist)]
    [statements]
    [name = expression]
    [Exit Function]
    [statements]
    [name = expression]
End Function
```

The Function statement syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Public</td>
<td>Indicates that the Function procedure is accessible to all other procedures in all scripts.</td>
</tr>
<tr>
<td>Default</td>
<td>Used only with the Public keyword in a Class block to indicate that the Function procedure is the default method for the class. An error occurs if more than one Default procedure is specified in a class.</td>
</tr>
<tr>
<td>Private</td>
<td>Indicates that the Function procedure is accessible only to other procedures in the script where it is declared or if the function is a member of a class, and that the Function procedure is accessible only to other procedures in that class.</td>
</tr>
<tr>
<td>name</td>
<td>Name of the Function; follows standard variable naming conventions.</td>
</tr>
<tr>
<td>arglist</td>
<td>List of variables representing arguments that are passed to the Function procedure when it is called. Multiple variables are separated by commas.</td>
</tr>
</tbody>
</table>
**Statements**

Any group of statements to be executed within the body of the **Function** procedure.

<table>
<thead>
<tr>
<th><strong>Expression</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Return value of the <strong>Function</strong>.</td>
</tr>
</tbody>
</table>

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

\[
[\text{ByVal} \mid \text{ByRef}] \ varname[()]
\]

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ByVal</td>
<td>Indicates that the <em>argument</em> is passed <em>by value</em>.</td>
</tr>
<tr>
<td>ByRef</td>
<td>Indicates that the argument is passed <em>by reference</em>.</td>
</tr>
<tr>
<td>varname</td>
<td>Name of the variable representing the argument; follows standard variable naming conventions.</td>
</tr>
</tbody>
</table>

**Remarks**

If not explicitly specified using either **Public** or **Private**, **Function** procedures are public by default, that is, they are visible to all other procedures in your script. The value of local variables in a **Function** is not preserved between calls to the *procedure*.

You can't define a **Function** procedure inside any other procedure (e.g. **Sub** or **Property Get**).

The **Exit Function** statement causes an immediate exit from a **Function** procedure. Program execution continues with the statement that follows 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 \texttt{Sqr}, \texttt{Cos}, or \texttt{Chr}, when you want to use the value returned by the function.

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

\textbf{Caution} Function procedures can be recursive, that is, they can call themselves to perform a given task. However, recursion can lead to stack overflow.

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 \textit{name}, the procedure returns a default value: a numeric function returns 0 and a string function returns a zero-length string (""). A function that returns an object reference returns \texttt{Nothing} if no object reference is assigned to \textit{name} (using \texttt{Set}) within the \textbf{Function}.

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

\begin{verbatim}
Function BinarySearch(...) 
  ...  
  ' Value not found. Return a value of False.
  If lower > upper Then
    BinarySearch = False  
    Exit Function 
  End If 
  ...  
End Function
\end{verbatim}
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 have defined at the **script 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 is referring to that script-level name. To avoid this kind of conflict, use an **Option Explicit** statement to force explicit declaration of variables.

**Caution** VBScript 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.
GetObject Function

See Also

Description

Returns a reference to an Automation object from a file.

Syntax

GetObject([pathname] [, class])

The GetObject function syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pathname</td>
<td>Optional; String. Full path and name of the file containing the object to retrieve. If pathname is omitted, class is required.</td>
</tr>
<tr>
<td>class</td>
<td>Optional; String. Class of the object.</td>
</tr>
</tbody>
</table>

The class argument uses the syntax appname.objectype and has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>appname</td>
<td>Required; String. Name of the application providing the object.</td>
</tr>
<tr>
<td>objectype</td>
<td>Required; String. Type or class of object to create.</td>
</tr>
</tbody>
</table>

Remarks

Use the GetObject function to access an Automation object from a file and assign the object to an object variable. Use the Set statement to assign the object returned by GetObject to the object variable. For example:
Dim CADObject
Set CADObject = GetObject("C:\CAD\SCHEMA.CAD")

When this code is executed, the application associated with the specified pathname is started and the object in the specified file is activated. If *pathname* is a zero-length string (""), *GetObject* returns a new object instance of the specified type. If the *pathname* argument is omitted, *GetObject* returns a currently active object of the specified type. If no object of the specified type exists, an error occurs.

Some applications allow you to activate part of a file. Add an exclamation point (!) to the end of the file name and follow it with a string that identifies the part of the file you want to activate. For information on how to create this string, see the documentation for the application that created the object.

For example, in a drawing application you might have multiple layers to a drawing stored in a file. You could use the following code to activate a layer within a drawing called SCHEMA.CAD:

    Set LayerObject = GetObject("C:\CAD\SCHEMA.

If you don't specify the object's class, Automation determines the application to start and the object to activate, based on the file name you provide. Some files, however, may support more than one class of object. For example, a drawing might support three different types of objects: an Application object, a Drawing object, and a Toolbar object, all of which are part of the same file. To specify which object in a file you want to activate, use the optional *class* argument. For example:

Dim MyObject
Set MyObject = GetObject("C:\DRAWINGS\SAMPLE.DRW", "FIGMENT.DRAWING")

In the preceding example, FIGMENT is the name of a drawing application and DRAWING is one of the object types it supports. Once an object is activated, you reference it in code using the object variable you defined. In the preceding example, you access properties and methods of the new
object using the object variable MyObject. For example:

```
MyObject.Line 9, 90
MyObject.InsertText 9, 100, "Hello, world."
MyObject.SaveAs "C:\DRAWINGS\SAMPLE.DRW"
```

**Note** Use the `GetObject` function when there is a current instance of the object or if you want to create the object with a file already loaded. If there is no current instance, and you don't want the object started with a file loaded, use the `CreateObject` function.

If an object has registered itself as a single-instance object, only one instance of the object is created, no matter how many times `CreateObject` is executed. With a single-instance object, `GetObject` always returns the same instance when called with the zero-length string ("") syntax, and it causes an error if the `pathname` argument is omitted.
GetRef Function

See Also

Description

Returns a reference to a procedure that can be bound to an event.

Syntax

Set object.eventname = GetRef(procname)

The GetRef function syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>Required. Name of the object with which event is associated.</td>
</tr>
<tr>
<td>event</td>
<td>Required. Name of the event to which the function is to be bound.</td>
</tr>
<tr>
<td>procname</td>
<td>Required. String containing the name of the Sub or Function procedure being associated with the event.</td>
</tr>
</tbody>
</table>

Remarks

The GetRef function allows you to connect a VBScript procedure (Function or Sub) to any available event on your DHTML (Dynamic HTML) pages. The DHTML object model provides information about what events are available for its various objects.

In other scripting and programming languages, the functionality provided by GetRef is referred to as a function pointer, that is, it points to the address of a procedure to be executed when the specified event occurs.

The following example illustrates the use of the GetRef function:
Function GetRefTest()
    Dim Splash
    Splash = "GetRefTest Version 1.0" & vbCrLf
    Splash = Splash & Chr(169) & " YourCompany 1999 "
    MsgBox Splash
End Function

Set Window.Onload = GetRef("GetRefTest")
Global Property

Description

Sets or returns a **Boolean** value that indicates if a pattern should match all occurrences in an entire search string or just the first one.

Syntax

```
object.Global [= True | False ]
```

The `object` argument is always a **RegExp** object. The value of the **Global** property is **True** if the search applies to the entire string, **False** if it does not. Default is **False**.

Remarks

The following code illustrates the use of the **Global** property (change the value assigned to **Global** property to see its effect):

```vbscript
Function RegExpTest(patrn, strng)
    Dim regEx, Match, Matches
    ' Create variable.
    Set regEx = New RegExp
    ' Create a regular expression.
    regEx.Pattern = patrn
    ' Set pattern.
    regEx.IgnoreCase = True
    ' Set case insensitivity.
    regEx.Global = True
    ' Set global applicability.
    Set Matches = regEx.Execute(strng)
    ' Execute search.
End Function
```
For Each Match in Matches   ' Iterate Matches collection.  
  RetStr = RetStr & "Match found at position "  
  RetStr = RetStr & Match.FirstIndex & ". Match Value is "  
  RetStr = RetStr & Match.Value & "." & vbCRLF  
Next  
  RegExpTest = RetStr  
End Function  
  
MsgBox(RegExpTest("is.", "IS1 is2 IS3 is4"))
Hex Function

See Also

Description

Returns a string representing the hexadecimal value of a number.

Syntax

**Hex(number)**

The *number* argument is any valid expression.

Remarks

If *number* is not already a whole number, it is rounded to the nearest whole number before being evaluated.

<table>
<thead>
<tr>
<th>If <em>number</em> is</th>
<th>Hex returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Null</td>
<td>Null.</td>
</tr>
<tr>
<td>Empty</td>
<td>Zero (0).</td>
</tr>
<tr>
<td>Any other number</td>
<td>Up to eight hexadecimal characters.</td>
</tr>
</tbody>
</table>

You can represent hexadecimal numbers directly by preceding numbers in the proper range with &H. For example, &H10 represents decimal 16 in hexadecimal notation.

The following example uses the **Hex** function to return the hexadecimal value of a number:

```
Dim MyHex
MyHex = Hex(5) ' Returns 5.
MyHex = Hex(10) ' Returns A.
MyHex = Hex(459) ' Returns 1CB.
```
HelpContext Property

See Also    Applies to

Description

Sets or returns a context ID for a topic in a Help File.

Syntax

```
object.HelpContext [= contextID]
```

The HelpContext property syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>Required. Always the Err object.</td>
</tr>
<tr>
<td>contextID</td>
<td>Optional. A valid identifier for a Help topic within the Help file.</td>
</tr>
</tbody>
</table>

Remarks

If a Help file is specified in HelpFile, the HelpContext property is used to automatically display the Help topic identified. If both HelpFile and HelpContext are empty, the value of the Number property is checked. If it corresponds to a VBScript run-time error value, then the VBScript Help context ID for the error is used. If the Number property doesn't correspond to a VBScript error, the contents screen for the VBScript Help file is displayed.

The following example illustrates use of the HelpContext property:

```
On Error Resume Next
Dim Msg
Err.Clear
```
Err.Raise 6 ' Generate "Overflow" error.
Err.Helpfile = "yourHelp.hlp"
Err.HelpContext = yourContextID
If Err.Number <> 0 Then
  Msg = "Press F1 or Help to see " & Err.Helpfile & " topic for" & " the following HelpContext: " & Err.HelpContext
End If
HelpFile Property

Description

Sets or returns a fully qualified path to a Help File.

Syntax

`object.HelpFile [= contextID]`

The HelpFile property syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>Required. Always the Err object.</td>
</tr>
<tr>
<td>contextID</td>
<td>Optional. Fully qualified path to the Help file.</td>
</tr>
</tbody>
</table>

Remarks

If a Help file is specified in HelpFile, it is automatically called when the user clicks the Help button (or presses the F1 key) in the error message dialog box. If the HelpContext property contains a valid context ID for the specified file, that topic is automatically displayed. If no HelpFile is specified, the VBScript Help file is displayed.

```vbnet
On Error Resume Next
Dim Msg
Err.Clear
Err.Raise 6  ' Generate "Overflow" error.
Err.Helpfile = "yourHelp.hlp"
Err.HelpContext = yourContextID
If Err.Number <> 0 Then
    Msg = "Press F1 or Help to see " & Err.Helpfile & " topic for" &
```
"the following HelpContext: " & Err.HelpContext
End If
Description

Returns a whole number between 0 and 23, inclusive, representing the hour of the day.

Syntax

**Hour**(time)

The *time* argument is any expression that can represent a time. If *time* contains *Null*, *Null* is returned.

The following example uses the **Hour** function to obtain the hour from the current time:

```
Dim MyTime, MyHour
MyTime = Now
MyHour = Hour(MyTime) ' MyHour contains the current hour.
```
If...Then...Else Statement

Description

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

Syntax

If condition Then statements [Else elseifstatements ]

Or, you can use the block form syntax:

If condition Then
    [statements]
    [ElseIf condition-n Then
        [elseifstatements]] . . .
    [Else
        [elseifstatements]]
End If

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

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>condition</td>
<td>One or more of the following two types of expressions:</td>
</tr>
<tr>
<td></td>
<td>A <strong>numeric</strong> or <strong>string expression</strong> that evaluates to <strong>True</strong> or <strong>False</strong>. If condition is <strong>Null</strong>, condition is treated as <strong>False</strong>.</td>
</tr>
<tr>
<td></td>
<td>An expression of the form <strong>TypeOf objectname Is objecttype</strong>. The <strong>objectname</strong> is any object reference and <strong>objecttype</strong> is any valid object type.</td>
</tr>
<tr>
<td></td>
<td>The expression is <strong>True</strong> if <strong>objectname</strong> is of the object type specified by <strong>objecttype</strong>; otherwise it is <strong>False</strong>.</td>
</tr>
</tbody>
</table>

**statements** One or more statements separated by colons; executed if condition is **True**. **condition-n** Same as condition. **elseifstatements** One or more statements executed if the associated condition-n is **True**. **elseifstatements** 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 syntax, it is possible to have multiple statements executed as the result of an `If...Then` decision, but they must all be on the same line and separated by colons, as in the following statement:

\[
\text{If } A > 10 \text{ Then } A = A + 1 : B = B + A : C = C + B
\]

---

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` (if any) is evaluated in turn. When a `True` condition is found, the statements following the associated `Then` are executed. If none of the `ElseIf` statements are `True` (or 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`.

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

What follows the `Then` keyword is examined to determine whether or not a statement is a block `If`. If anything other than a comment appears after `Then` on the same line, the statement is treated as a single-line `If` statement.

A block `If` statement must be the first statement on a line. The block `If` must end with an `End If` statement.
IgnoreCase Property

Description

Sets or returns a **Boolean** value that indicates if a pattern search is case-sensitive or not.

Syntax

```
object.IgnoreCase [= True | False ]
```

The `object` argument is always a **RegExp** object. The value of the **IgnoreCase** property is **False** if the search is case-sensitive, **True** if it is not. Default is **False**.

Remarks

The following code illustrates the use of the **IgnoreCase** property (change the value assigned to **IgnoreCase** property to see its effect):

```vbnet
Function RegExpTest(patrn, strng)
    Dim regEx, Match, Matches
    Set regEx = New RegExp
    regEx.Pattern = patrn
    regEx.IgnoreCase = True
    ' Set case insensitivity.
    regEx.Execute (strng)
    For Each Match In Matches
        WScript.Echo Match.Text
    Next
End Function
```
regEx.Global = True  ' Set global applicability.
Set Matches = regEx.Execute(strng)  ' Execute search.
For Each Match in Matches  ' Iterate Matches collection.
  RetStr = RetStr & "Match found at position "
  RetStr = RetStr & Match.FirstIndex & ". Match Value is "
  RetStr = RetStr & Match.Value & "," & vbCRLF
Next
RegExpTest = RetStr
End Function

MsgBox(RegExpTest("is.", "IS1 is2 IS3 is4"))
Imp Operator

See Also

Description

Performs a logical implication on two expressions.

Syntax

\[
\text{result} = \text{expression1} \textbf{Imp} \text{ expression2}
\]

The \textbf{Imp} operator syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>Any numeric variable.</td>
</tr>
<tr>
<td>expression1</td>
<td>Any expression</td>
</tr>
<tr>
<td>expression2</td>
<td>Any expression</td>
</tr>
</tbody>
</table>

Remarks

The following table illustrates how \textit{result} is determined:

<table>
<thead>
<tr>
<th>If \textit{expression1} is</th>
<th>And \textit{expression2} is</th>
<th>Then \textit{result} is</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>True</td>
<td>Null</td>
<td>Null</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>False</td>
<td>Null</td>
<td>True</td>
</tr>
<tr>
<td>Null</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>Null</td>
<td>False</td>
<td>Null</td>
</tr>
</tbody>
</table>
The **Imp** operator performs a [bitwise comparison](#) of identically positioned bits in two [numeric expressions](#) and sets the corresponding bit in *result* according to the following table:

<table>
<thead>
<tr>
<th>If bit in <em>expression1</em> is</th>
<th>And bit in <em>expression2</em> is</th>
<th>Then <em>result</em> is</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Initialize Event

Description

Occurs when an instance of the associated class is created.

Syntax

Private Sub Class_Initialize()
    statements
End Sub

The statements part consists of zero or more code statements to be run when the class is initialized.

Remarks

The following example illustrates the use of the Initialize event:

Class TestClass
    Private Sub Class_Initialize ' Setup Initialize event.
        MsgBox("TestClass started")
    End Sub
    Private Sub Class_Terminate ' Setup Terminate event.
        MsgBox("TestClass terminated")
    End Sub
End Class

Set X = New TestClass ' Create an instance of TestClass.
Set X = Nothing ' Destroy the instance.
### InputBox Function

**Description**

Displays a prompt in a dialog box, waits for the user to input text or click a button, and returns the contents of the text box.

**Syntax**

```plaintext
InputBox(prompt[, title][, default][, xpos][, ypos][, helpfile, context])
```

The `InputBox` function syntax has these arguments:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>prompt</code></td>
<td>String expression displayed as the message in the dialog box. The maximum length of <code>prompt</code> is approximately 1024 characters, depending on the width of the characters used. If <code>prompt</code> consists of more than one line, you can separate the lines using a carriage return character (<code>Chr(13)</code>), a linefeed character (<code>Chr(10)</code>), or carriage return–linefeed character combination (<code>Chr(13) &amp; Chr(10)</code>) between each line.</td>
</tr>
<tr>
<td><code>title</code></td>
<td>String expression displayed in the title bar of the dialog box. If you omit <code>title</code>, the application name is placed in the title bar.</td>
</tr>
<tr>
<td><code>default</code></td>
<td>String expression displayed in the text box as the default response if no other input is provided. If you omit <code>default</code>, the text box is displayed empty.</td>
</tr>
<tr>
<td><code>xpos</code></td>
<td>Numeric expression that specifies, in twips, the horizontal distance of the left edge of the dialog box from the left edge of the screen. If <code>xpos</code> is omitted, the dialog box is horizontally centered.</td>
</tr>
<tr>
<td><code>ypos</code></td>
<td>Numeric expression that specifies, in twips, the vertical</td>
</tr>
</tbody>
</table>
**ypos**
Distance of the upper edge of the dialog box from the top of the screen. If *ypos* is omitted, the dialog box is vertically positioned approximately one-third of the way down the screen.

**helpfile**
String expression that identifies the Help file to use to provide context-sensitive Help for the dialog box. If *helpfile* is provided, *context* must also be provided.

**context**
Numeric expression that identifies the Help context number assigned by the Help author to the appropriate Help topic. If *context* is provided, *helpfile* must also be provided.

**Remarks**

When both *helpfile* and *context* are supplied, a Help button is automatically added to the dialog box.

If the user clicks **OK** or presses **ENTER**, the **InputBox** function returns whatever is in the text box. If the user clicks **Cancel**, the function returns a zero-length string (""").

The following example uses the **InputBox** function to display an input box and assign the string to the variable *Input*:

```
Dim Input
Input = InputBox("Enter your name")
MsgBox ("You entered: " & Input)
```
InStr Function

See Also

Description

Returns the position of the first occurrence of one string within another.

Syntax

InStr([start, ]string1, string2[, compare])

The InStr function syntax has these arguments:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>Optional. Numeric expression that sets the starting position for each search. If omitted, search begins at the first character position. If start contains Null, an error occurs. The start argument is required if compare is specified.</td>
</tr>
<tr>
<td>string1</td>
<td>Required. String expression being searched.</td>
</tr>
<tr>
<td>string2</td>
<td>Required. String expression searched for.</td>
</tr>
<tr>
<td>compare</td>
<td>Optional. Numeric value indicating the kind of comparison to use when evaluating substrings. See Settings section for values. If omitted, a binary comparison is performed.</td>
</tr>
</tbody>
</table>

Settings

The compare argument can have the following values:
<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vbBinaryCompare</td>
<td>0</td>
<td>Perform a binary comparison.</td>
</tr>
<tr>
<td>vbTextCompare</td>
<td>1</td>
<td>Perform a textual comparison.</td>
</tr>
</tbody>
</table>

**Return Values**

The `InStr` function returns the following values:

<table>
<thead>
<tr>
<th>If</th>
<th><code>InStr</code> returns</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>string1</code> is zero-length</td>
<td>0</td>
</tr>
<tr>
<td><code>string1</code> is Null</td>
<td><code>Null</code></td>
</tr>
<tr>
<td><code>string2</code> is zero-length</td>
<td><code>start</code></td>
</tr>
<tr>
<td><code>string2</code> is Null</td>
<td><code>Null</code></td>
</tr>
<tr>
<td><code>string2</code> is not found</td>
<td>0</td>
</tr>
<tr>
<td><code>string2</code> is found within <code>string1</code></td>
<td>Position at which match is found</td>
</tr>
<tr>
<td><code>start</code> &gt; <code>Len(string2)</code></td>
<td>0</td>
</tr>
</tbody>
</table>

**Remarks**

The following examples use `InStr` to search a string:

```vba
Dim SearchString, SearchChar, MyPos
SearchString = "XXpXXpXXPXXP"
SearchChar = "P"

MyPos = Instr(4, SearchString, SearchChar, 1) ' position 4
MyPos = Instr(1, SearchString, SearchChar, 0) ' position 1
MyPos = Instr(SearchString, SearchChar)      ' search for "P" in entire string
```
MyPos = Instr(1, SearchString, "W")
' Returns 9

' (last argument is omitted).
' Returns 0

Note  The InStrB function is used with byte data contained in a string. Instead of returning the character position of the first occurrence of one string within another, InStrB returns the byte position.
Operator

**Description**

Divides two numbers and returns an integer result.

**Syntax**

\[ result = number1 \div number2 \]

The \ operator syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>Any numeric variable.</td>
</tr>
<tr>
<td>number1</td>
<td>Any numeric expression.</td>
</tr>
<tr>
<td>number2</td>
<td>Any numeric expression.</td>
</tr>
</tbody>
</table>

**Remarks**

Before division is performed, numeric expressions are rounded to **Byte**, **Integer**, or **Long** subtype expressions.

If any expression is **Null**, \( result \) is also **Null**. Any expression that is **Empty** is treated as 0.
Is Operator

Description

Compares two object reference variables.

Syntax

\[ result = object1 \ Is \ object2 \]

The Is operator syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>Any numeric variable.</td>
</tr>
<tr>
<td>object1</td>
<td>Any object name.</td>
</tr>
<tr>
<td>object2</td>
<td>Any object name.</td>
</tr>
</tbody>
</table>

Remarks

If \( object1 \) and \( object2 \) both refer to the same object, \( result \) is \textbf{True}; if they do not, \( result \) is \textbf{False}. Two variables can be made to refer to the same object in several ways.

In the following example, \( A \) has been set to refer to the same object as \( B \):

\[
\text{Set } A = B
\]

The following example makes \( A \) and \( B \) refer to the same object as \( C \):

\[
\text{Set } A = C
\]
Set \( B = C \)
**IsArray**

**Function**

See Also

**Description**

Returns a Boolean value indicating whether a variable is an array.

**Syntax**

\[ \text{IsArray}(\text{varname}) \]

The \textit{varname} argument can be any variable.

**Remarks**

\texttt{IsArray} returns True if the variable is an array; otherwise, it returns False. \texttt{IsArray} is especially useful with variants containing arrays.

The following example uses the \texttt{IsArray} function to test whether MyVariable is an array:

\begin{verbatim}
Dim MyVariable
Dim MyArray(3)
MyArray(0) = "Sunday"
MyArray(1) = "Monday"
MyArray(2) = "Tuesday"
MyVariable = IsArray(MyArray) ' MyVariable contains "True".
\end{verbatim}
IsDate Function

See Also

Description

Returns a Boolean value indicating whether an expression can be converted to a date.

Syntax

IsDate(expression)

The expression argument can be any date expression or string expression recognizable as a date or time.

Remarks

IsDate returns True if the expression is a date or can be converted to a valid date; otherwise, it returns False. In Microsoft Windows, the range of valid dates is January 1, 100 A.D. through December 31, 9999 A.D.; the ranges vary among operating systems.

The following example uses the IsDate function to determine whether an expression can be converted to a date:

Dim MyDate, YourDate, NoDate, MyCheck
MyDate = "October 19, 1962": YourDate = #10/19/
MyCheck = IsDate(MyDate) ' Returns True.
MyCheck = IsDate(YourDate) ' Returns True.
MyCheck = IsDate(NoDate) ' Returns False.
IsEmpty Function

See Also

Description

Returns a Boolean value indicating whether a variable has been initialized.

Syntax

IsEmpty(expression)

The expression argument can be any expression. However, because IsEmpty is used to determine if individual variables are initialized, the expression argument is most often a single variable name.

Remarks

IsEmpty returns True if the variable is uninitialized, or is explicitly set to Empty; otherwise, it returns False. False is always returned if expression contains more than one variable.

The following example uses the IsEmpty function to determine whether a variable has been initialized:

Dim MyVar, MyCheck
MyCheck = IsEmpty(MyVar) ' Returns True.
MyVar = Null ' Assign Null.
MyCheck = IsEmpty(MyVar) ' Returns False.
MyVar = Empty ' Assign Empty.
MyCheck = **IsEmpty** (MyVar)  ' Returns True.
IsNull Function

Description

Returns a Boolean value that indicates whether an expression contains no valid data (null).

Syntax

IsNull(expression)

The expression argument can be any expression.

Remarks

IsNull returns True if expression is Null, that is, it contains no valid data; otherwise, IsNull returns False. If expression consists of more than one variable, Null in any constituent variable causes True to be returned for the entire expression.

The Null value indicates that the variable contains no valid data. Null is not the same as Empty, which indicates that a variable has not yet been initialized. It is also not the same as a zero-length string (""), which is sometimes referred to as a null string.

Important Use the IsNull function to determine whether an expression contains a Null value. Expressions that you might expect to evaluate to True under some circumstances, such as If Var = Null and If Var <> Null, are always False. This is because any expression containing a Null is itself Null, and therefore, False.

The following example uses the IsNull function to determine whether a variable contains a Null:
Dim MyVar, MyCheck
MyCheck = IsNull(MyVar)  ' Returns False.
MyVar = Null            ' Assign Null.
MyCheck = IsNull(MyVar)  ' Returns True.
MyVar = Empty           ' Assign Empty.
MyCheck = IsNull(MyVar)  ' Returns False.
IsNumeric Function

Description

Returns a Boolean value indicating whether an expression can be evaluated as a number.

Syntax

IsNumeric(expression)

The expression argument can be any expression.

Remarks

IsNumeric returns True if the entire expression is recognized as a number; otherwise, it returns False. IsNumeric returns False if expression is a date expression.

The following example uses the IsNumeric function to determine whether a variable can be evaluated as a number:

```
Dim MyVar, MyCheck
MyVar = 53          ' Assign a value.
MyCheck = IsNumeric(MyVar)  ' Returns True.
MyVar = "459.95"     ' Assign a value.
MyCheck = IsNumeric(MyVar)  ' Returns True.
MyVar = "45 Help"   ' Assign a value.
MyCheck = IsNumeric(MyVar)  ' Returns False.
```
**IsObject Function**

**See Also**

**Description**

Returns a Boolean value indicating whether an expression references a valid [Automation object](#).

**Syntax**

```
IsObject(expression)
```

The `expression` argument can be any `expression`.

**Remarks**

`IsObject` returns **True** if `expression` is a variable of `Object` subtype or a user-defined object; otherwise, it returns **False**.

The following example uses the `IsObject` function to determine if an identifier represents an object variable:

```
Dim MyInt, MyCheck, MyObject
Set MyObject = Me
MyCheck = IsObject(MyObject)  ' Returns True.
MyCheck = IsObject(MyInt)     ' Returns False.
```
Join Function

Description

Returns a string created by joining a number of substrings contained in an array.

Syntax

\[ \text{Join}(list[, \text{delimiter}]) \]

The \text{Join} function syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>list</td>
<td>Required. One-dimensional array containing substrings to be joined.</td>
</tr>
<tr>
<td>delimiter</td>
<td>Optional. String character used to separate the substrings in the returned string. If omitted, the space character (&quot; &quot;) is used. If \text{delimiter} is a zero-length string, all items in the list are concatenated with no delimiters.</td>
</tr>
</tbody>
</table>

Remarks

The following example uses the \text{Join} function to join the substrings of \text{MyArray}:

\begin{verbatim}
Dim MyString
Dim MyArray(4)
MyArray(0) = "Mr."
\end{verbatim}
MyArray(1) = "John 
MyArray(2) = "Doe 
MyArray(3) = "III"
MyString = Join(MyArray) ' MyString contains "

**LBound**

**Function**

**See Also**

**Description**

Returns the smallest available subscript for the indicated dimension of an array.

**Syntax**

\[
\text{LBound}(arrayname[, dimension])
\]

The `LBound` function syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>arrayname</code></td>
<td>Name of the array variable; follows standard variable naming conventions.</td>
</tr>
<tr>
<td><code>dimension</code></td>
<td>Whole number indicating which dimension's lower bound is returned. Use 1 for the first dimension, 2 for the second, and so on. If <code>dimension</code> is omitted, 1 is assumed.</td>
</tr>
</tbody>
</table>

**Remarks**

The `LBound` function is used with the `UBound` function to determine the size of an array. Use the `UBound` function to find the upper limit of an array dimension.

The lower bound for any dimension is always 0.
LCase Function

See Also

Description

Returns a string that has been converted to lowercase.

Syntax

```
LCase(string)
```

The `string` argument is any valid string expression. If `string` contains `Null`, `Null` is returned.

Remarks

Only uppercase letters are converted to lowercase; all lowercase letters and nonletter characters remain unchanged.

The following example uses the `LCase` function to convert uppercase letters to lowercase:

```
Dim MyString
Dim LCaseString
MyString = "VBSCript"
LCaseString = LCase(MyString) ' LCaseString contains "vbscript"
```
Function

Description

Returns a specified number of characters from the left side of a string.

Syntax

**Left**(string, length)

The *Left* function syntax has these arguments:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td><em>String expression</em> from which the leftmost characters are returned. If <em>string</em> contains <em>Null, Null</em> is returned.</td>
</tr>
<tr>
<td>length</td>
<td><em>Numeric expression</em> indicating how many characters to return. If 0, a zero-length string(&quot;&quot;) is returned. If greater than or equal to the number of characters in <em>string</em>, the entire string is returned.</td>
</tr>
</tbody>
</table>

Remarks

To determine the number of characters in *string*, use the **Len** function.

The following example uses the *Left* function to return the first three characters of *MyString*:

```
Dim MyString, LeftString
MyString = "VBScript"
LeftString = Left(MyString, 3) ' LeftString contains "VBS"
```
**Note** The **LeftB** function is used with byte data contained in a string. Instead of specifying the number of characters to return, *length* specifies the number of bytes.
Function

See Also

Description

Returns the number of characters in a string or the number of bytes required to store a variable.

Syntax

```
Len(string | varname)
```

The `Len` function syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>Any valid string expression. If string contains Null, Null is returned.</td>
</tr>
<tr>
<td>varname</td>
<td>Any valid variable name. If varname contains Null, Null is returned.</td>
</tr>
</tbody>
</table>

Remarks

The following example uses the `Len` function to return the number of characters in a string:

```
Dim MyString
MyString = Len("VBSCRIPT") ' MyString contains 8.
```

**Note** The `LenB` function is used with byte data contained in a string. Instead of returning the number of characters in a string, `LenB` returns the number of bytes used to represent that string.
**Description**

Returns the length of a match found in a search string.

**Syntax**

```vbscript
object.Length
```

The `object` argument is always a `Match` object.

**Remarks**

The following code illustrates the use of the `Length` property:

```vbscript
Function RegExpTest(patrn, strng)
    Dim regEx, Match, Matches
    ' Create variable.
    Set regEx = New RegExp
    ' Create regular expression.
    regEx.Pattern = patrn
    ' Set pattern.
    regEx.IgnoreCase = True
    ' Set case insensitivity.
    regEx.Global = True
    ' Set global applicability.
    Set Matches = regEx.Execute(strng)
    ' Execute search.
    For Each Match in Matches
        ' Iterate Matches collection.
        RetStr = RetStr & "Match " & I & " found at position " & Match.FirstIndex & ". Match Length is " & Match.Length & " characters." & vbCrLf
    Next
End Function
```
Next
    RegExpTest = RetStr
End Function

MsgBox(RegExpTest("is.", "IS1 is2 IS3 is4"))
LoadPicture Function

Description

Returns a picture object. Available only on 32-bit platforms.

Syntax

LoadPicture(picturename)

The picturename argument is a string expression that indicates the name of the picture file to be loaded.

Remarks

Graphics formats recognized by LoadPicture include bitmap (.bmp) files, icon (.ico) files, run-length encoded (.rle) files, metafile (.wmf) files, enhanced metafiles (.emf), GIF (.gif) files, and JPEG (.jpg) files.
**Description**

Returns the natural logarithm of a number.

**Syntax**

```
Log(number)
```

The `number` argument can be any valid numeric expression greater than 0.

**Remarks**

The natural logarithm is the logarithm to the base $e$. The constant $e$ is approximately 2.718282.

You can calculate base-$n$ logarithms for any number $x$ by dividing the natural logarithm of $x$ by the natural logarithm of $n$ as follows:

```
Logn(x) = Log(x) / Log(n)
```

The following example illustrates a custom **Function** that calculates base-10 logarithms:

```
Function Log10(X)
    Log10 = Log(X) / Log(10)
End Function
```
LTrim, RTrim, and Trim Functions

Description

Returns a copy of a string without leading spaces (LTrim), trailing spaces (RTrim), or both leading and trailing spaces (Trim).

Syntax

LTrim(string)

RTrim(string)

Trim(string)

The string argument is any valid string expression. If string contains Null, Null is returned.

Remarks

The following example uses the LTrim, RTrim, and Trim functions to trim leading spaces, trailing spaces, and both leading and trailing spaces, respectively:

Dim MyVar
MyVar = LTrim(" vbscript ") ' MyVar contains "vbscript"
MyVar = `RTrim(" vbscript ")` ' MyVar contains "vbscript".
MyVar = `Trim(" vbscript ")` ' MyVar contains "vbscript".
**Match Object**

**Description**

Provides access to the read-only properties of a regular expression match.

**Remarks**

A **Match** object can be only created using the **Execute** method of the **RegExp** object, which actually returns a **collection** of **Match** objects. All **Match** object properties are read-only.

When a regular expression is executed, zero or more **Match** objects can result. Each **Match** object provides access to the string found by the regular expression, the length of the string, and an index to where the match was found.

The following code illustrates the use of the **Match** object:

```vbs
Function RegExpTest(pattern, string)
    Dim regEx, Match, Matches
    Set regEx = New RegExp
    regEx.Pattern = pattern
    regEx.IgnoreCase = True
    regEx.Global = True
    Set Matches = regEx.Execute(string)
    For Each Match in Matches
        ' Iterate Matches collection.
    Next
End Function
```
RetStr = RetStr & "Match " & I & " found at position 
RetStr = RetStr & Match.FirstIndex & ". Match Value is "
RetStr = RetStr & Match.Value & "." & vbCRLF
Next
RegExpTest = RetStr
End Function

MsgBox(RegExpTest("is.", "IS1 is2 IS3 is4"))
### Description

A Matches collection contains individual Match objects, and can be only created using the Execute method of the RegExp object. The Matches collection's one property is read-only, as are the individual Match object properties.

When a regular expression is executed, zero or more Match objects can result. Each Match object provides access to the string found by the regular expression, the length of the string, and an index to where the match was found.

The following code illustrates how to obtain a Matches collection from a regular expression search and how to iterate the collection:

```vbscript
Function RegExpTest(patrn, strng)
    Dim regEx, Match, Matches
    ' Create variable.
    Set regEx = New RegExp
    ' Create regular expression.
    regEx.Pattern = patrn
    ' Set pattern.
    regEx.IgnoreCase = True
    ' Set case insensitivity.
    regEx.Global = True
    ' Set global applicability.
    Matches = regEx.Execute(strng)
    ' Iterate the matches.
End Function
```

Set Matches = regEx.Execute(strng)  ' Execute search.
For Each Match in Matches  ' Iterate Matches collection.
    RetStr = RetStr & "Match found at position "
    RetStr = RetStr & Match.FirstIndex & ". Match Value is "
    RetStr = RetStr & Match.Value & "." & vbCRLF
    Next
RegExpTest = RetStr
End Function

MsgBox(RegExpTest("is.", "IS1 is2 IS3 is4"))
Mid Function

See Also

Description

Returns a specified number of characters from a string.

Syntax

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

The Mid function syntax has these arguments:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>String expression from which characters are returned. If string contains Null, Null is returned.</td>
</tr>
<tr>
<td>start</td>
<td>Character position in string at which the part to be taken begins. If start is greater than the number of characters in string, Mid returns a zero-length string (&quot;&quot;).</td>
</tr>
<tr>
<td>length</td>
<td>Number of characters to return. If omitted or if there are fewer than length characters in the text (including the character at start), all characters from the start position to the end of the string are returned.</td>
</tr>
</tbody>
</table>

Remarks

To determine the number of characters in string, use the Len function.

The following example uses the Mid function to return six characters, beginning with the fourth character, in a string:

Dim MyVar
MyVar = **Mid**("VB Script is fun!", 4, 6) ' MyVar cc

**Note** The **MidB** function is used with byte data contained in a string. Instead of specifying the number of characters, the arguments specify numbers of bytes.
Minute Function

See Also

Description

Returns a whole number between 0 and 59, inclusive, representing the minute of the hour.

Syntax

Minute(time)

The time argument is any expression that can represent a time. If time contains Null, Null is returned.

Remarks

The following example uses the Minute function to return the minute of the hour:

Dim MyVar
MyVar = Minute(Now)
Since this constant is built into VBScript, you don't have to define it before using it. Use it anywhere in your code to represent the values shown.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vbObjectError</td>
<td>-2147221504</td>
<td>User-defined error numbers should be greater than this value, for example, Err.Raise Number = vbObjectE</td>
</tr>
</tbody>
</table>
Description

Divides two numbers and returns only the remainder.

Syntax

\[
result = number1 \ Mod \ number2
\]

The **Mod** operator syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>Any numeric variable.</td>
</tr>
<tr>
<td>number1</td>
<td>Any numeric expression.</td>
</tr>
<tr>
<td>number2</td>
<td>Any numeric expression.</td>
</tr>
</tbody>
</table>

Remarks

The modulus, or remainder, operator divides `number1` by `number2` (rounding floating-point numbers to integers) and returns only the remainder as `result`. For example, in the following expression, A (which is `result`) equals 5.

\[
A = 19 \ Mod \ 6.7
\]

If any expression is **Null**, `result` is also **Null**. Any expression that is **Empty** is treated as 0.
Month Function

Description

Returns a whole number between 1 and 12, inclusive, representing the month of the year.

Syntax

Month(date)

The date argument is any expression that can represent a date. If date contains Null, Null is returned.

Remarks

The following example uses the Month function to return the current month:

Dim MyVar
MyVar = Month(Now) ' MyVar contains the number corresponding to the current month.
**MonthName Function**

**Description**

Returns a string indicating the specified month.

**Syntax**

```vbnet
MonthName(month[, abbreviate])
```

The `MonthName` function syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>month</code></td>
<td>Required. The numeric designation of the month. For example, January is 1, February is 2, and so on.</td>
</tr>
<tr>
<td><code>abbreviate</code></td>
<td>Optional. Boolean value that indicates if the month name is to be abbreviated. If omitted, the default is <code>False</code>, which means that the month name is not abbreviated.</td>
</tr>
</tbody>
</table>

**Remarks**

The following example uses the `MonthName` function to return an abbreviated month name for a date expression:

```vbnet
Dim MyVar
MyVar = MonthName(10, True) ' MyVar contains "Oct".
```
The following constants are used with the **MsgBox** function to identify what buttons and icons appear on a message box and which button is the default. In addition, the modality of the **MsgBox** can be specified. Since these constants are built into VBScript, you don't have to define them before using them. Use them anywhere in your code to represent the values shown for each.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vbOKOnly</td>
<td>0</td>
<td>Display <strong>OK</strong> button only.</td>
</tr>
<tr>
<td>vbOKCancel</td>
<td>1</td>
<td>Display <strong>OK</strong> and <strong>Cancel</strong> buttons.</td>
</tr>
<tr>
<td>vbAbortRetryIgnore</td>
<td>2</td>
<td>Display <strong>Abort</strong>, <strong>Retry</strong>, and <strong>Ignore</strong> buttons.</td>
</tr>
<tr>
<td>vbYesNoCancel</td>
<td>3</td>
<td>Display <strong>Yes</strong>, <strong>No</strong>, and <strong>Cancel</strong> buttons.</td>
</tr>
<tr>
<td>vbYesNo</td>
<td>4</td>
<td>Display <strong>Yes</strong> and <strong>No</strong> buttons.</td>
</tr>
<tr>
<td>vbRetryCancel</td>
<td>5</td>
<td>Display <strong>Retry</strong> and <strong>Cancel</strong> buttons.</td>
</tr>
<tr>
<td>vbCritical</td>
<td>16</td>
<td>Display <strong>Critical Message</strong> icon.</td>
</tr>
<tr>
<td>vbQuestion</td>
<td>32</td>
<td>Display <strong>Warning Query</strong> icon.</td>
</tr>
<tr>
<td>vbExclamation</td>
<td>48</td>
<td>Display <strong>Warning Message</strong> icon.</td>
</tr>
<tr>
<td>vbInformation</td>
<td>64</td>
<td>Display <strong>Information Message</strong> icon.</td>
</tr>
<tr>
<td>vbDefaultButton1</td>
<td>0</td>
<td>First button is the default.</td>
</tr>
<tr>
<td>vbDefaultButton2</td>
<td>256</td>
<td>Second button is the default.</td>
</tr>
<tr>
<td>vbDefaultButton3</td>
<td>512</td>
<td>Third button is the default.</td>
</tr>
<tr>
<td>vbDefaultButton4</td>
<td>768</td>
<td>Fourth button is the default.</td>
</tr>
</tbody>
</table>
Application modal. The user must respond to the message box before continuing work in the current application.

System modal. On Win16 systems, all applications are suspended until the user responds to the message box. On Win32 systems, this constant provides an application modal message box that always remains on top of any other programs you may have running.

The following constants are used with the `MsgBox` function to identify which button a user has selected. These constants are only available when your project has an explicit reference to the appropriate type library containing these constant definitions. For VBScript, you must explicitly declare these constants in your code.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vbOK</td>
<td>1</td>
<td>OK button was clicked.</td>
</tr>
<tr>
<td>vbCancel</td>
<td>2</td>
<td>Cancel button was clicked.</td>
</tr>
<tr>
<td>vbAbort</td>
<td>3</td>
<td>Abort button was clicked.</td>
</tr>
<tr>
<td>vbRetry</td>
<td>4</td>
<td>Retry button was clicked.</td>
</tr>
<tr>
<td>vbIgnore</td>
<td>5</td>
<td>Ignore button was clicked.</td>
</tr>
<tr>
<td>vbYes</td>
<td>6</td>
<td>Yes button was clicked.</td>
</tr>
<tr>
<td>vbNo</td>
<td>7</td>
<td>No button was clicked.</td>
</tr>
</tbody>
</table>
Function

See Also

Description

Displays a message in a dialog box, waits for the user to click a button, and returns a value indicating which button the user clicked.

Syntax

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

The `MsgBox` function syntax has these arguments:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>prompt</code></td>
<td>String expression displayed as the message in the dialog box. The maximum length of <code>prompt</code> is approximately 1024 characters, depending on the width of the characters used. If <code>prompt</code> consists of more than one line, you can separate the lines using a carriage return character (<code>Chr(13)</code>), a linefeed character (<code>Chr(10)</code>), or carriage return–linefeed character combination (<code>Chr(13) &amp; Chr(10)</code>) between each line.</td>
</tr>
<tr>
<td><code>buttons</code></td>
<td>Numeric expression that is the sum of values specifying the number and type of buttons to display, the icon style to use, the identity of the default button, and the modality of the message box. See Settings section for values. If omitted, the default value for <code>buttons</code> is 0.</td>
</tr>
<tr>
<td><code>title</code></td>
<td>String expression displayed in the title bar of the dialog box. If you omit <code>title</code>, the application name is placed in the title bar.</td>
</tr>
<tr>
<td><code>helpfile</code></td>
<td>String expression that identifies the Help file to use to</td>
</tr>
</tbody>
</table>
**helpfile**

Provide context-sensitive Help for the dialog box. If *helpfile* is provided, *context* must also be provided. Not available on 16-bit platforms.

**context**

Numeric expression that identifies the Help context number assigned by the Help author to the appropriate Help topic. If *context* is provided, *helpfile* must also be provided. Not available on 16-bit platforms.

### Settings

The *buttons* argument settings are:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vbOKOnly</td>
<td>0</td>
<td>Display <strong>OK</strong> button only.</td>
</tr>
<tr>
<td>vbOKCancel</td>
<td>1</td>
<td>Display <strong>OK</strong> and <strong>Cancel</strong> buttons.</td>
</tr>
<tr>
<td>vbAbortRetryIgnore</td>
<td>2</td>
<td>Display <strong>Abort</strong>, <strong>Retry</strong>, and <strong>Ignore</strong> buttons.</td>
</tr>
<tr>
<td>vbYesNoCancel</td>
<td>3</td>
<td>Display <strong>Yes</strong>, <strong>No</strong>, and <strong>Cancel</strong> buttons.</td>
</tr>
<tr>
<td>vbYesNo</td>
<td>4</td>
<td>Display <strong>Yes</strong> and <strong>No</strong> buttons.</td>
</tr>
<tr>
<td>vbRetryCancel</td>
<td>5</td>
<td>Display <strong>Retry</strong> and <strong>Cancel</strong> buttons.</td>
</tr>
<tr>
<td>vbCritical</td>
<td>16</td>
<td>Display <strong>Critical Message</strong> icon.</td>
</tr>
<tr>
<td>vbQuestion</td>
<td>32</td>
<td>Display <strong>Warning Query</strong> icon.</td>
</tr>
<tr>
<td>vbExclamation</td>
<td>48</td>
<td>Display <strong>Warning Message</strong> icon.</td>
</tr>
<tr>
<td>vbInformation</td>
<td>64</td>
<td>Display <strong>Information Message</strong> icon.</td>
</tr>
<tr>
<td>vbDefaultButton1</td>
<td>0</td>
<td>First button is default.</td>
</tr>
<tr>
<td>vbDefaultButton2</td>
<td>256</td>
<td>Second button is default.</td>
</tr>
<tr>
<td>vbDefaultButton3</td>
<td>512</td>
<td>Third button is default.</td>
</tr>
<tr>
<td>vbDefaultButton4</td>
<td>768</td>
<td>Fourth button is default.</td>
</tr>
</tbody>
</table>

Application modal; the user must respond to the message box before
The first group of values (0–5) describes the number and type of buttons displayed in the dialog box; the second group (16, 32, 48, 64) describes the icon style; the third group (0, 256, 512, 768) determines which button is the default; and the fourth group (0, 4096) determines the modality of the message box. When adding numbers to create a final value for the argument buttons, use only one number from each group.

Return Values

The MsgBox function has the following return values:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Button</th>
</tr>
</thead>
<tbody>
<tr>
<td>vbOK</td>
<td>1</td>
<td>OK</td>
</tr>
<tr>
<td>vbCancel</td>
<td>2</td>
<td>Cancel</td>
</tr>
<tr>
<td>vbAbort</td>
<td>3</td>
<td>Abort</td>
</tr>
<tr>
<td>vbRetry</td>
<td>4</td>
<td>Retry</td>
</tr>
<tr>
<td>vbIgnore</td>
<td>5</td>
<td>Ignore</td>
</tr>
<tr>
<td>vbYes</td>
<td>6</td>
<td>Yes</td>
</tr>
<tr>
<td>vbNo</td>
<td>7</td>
<td>No</td>
</tr>
</tbody>
</table>

Remarks

When both helpfile and context are provided, the user can press F1 to view the Help topic corresponding to the context.

If the dialog box displays a Cancel button, pressing the ESC key has the same effect as clicking Cancel. If the dialog box contains a Help button, context-sensitive Help is provided for the dialog box. However, no value is returned until one of the other buttons is clicked.

When the MsgBox function is used with Microsoft Internet Explorer, the title of any dialog presented always contains "VBScript:" to differentiate it from standard system dialogs.

The following example uses the MsgBox function to display a message box and return a value describing which button was clicked:
Dim MyVar
MyVar = MsgBox ("Hello World!", 65, "MsgBox Example") ' My'
' depending on which button is 
' clicked.
**Operator**

**Description**

Multiplies two numbers.

**Syntax**

\[ result = number1 * number2 \]

The * operator syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>Any numeric variable.</td>
</tr>
<tr>
<td>number1</td>
<td>Any numeric expression.</td>
</tr>
<tr>
<td>number2</td>
<td>Any numeric expression.</td>
</tr>
</tbody>
</table>

**Remarks**

If one or both expressions are Null expressions, result is Null. If an expression is Empty, it is treated as if it were 0.
Operator

Description

Finds the difference between two numbers or indicates the negative value of a numeric expression.

Syntax 1

\[ result = number1 - number2 \]

Syntax 2

\[-number\]

The - operator syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>Any numeric variable.</td>
</tr>
<tr>
<td>number</td>
<td>Any numeric expression.</td>
</tr>
<tr>
<td>number1</td>
<td>Any numeric expression.</td>
</tr>
<tr>
<td>number2</td>
<td>Any numeric expression.</td>
</tr>
</tbody>
</table>

Remarks

In Syntax 1, the - operator is the arithmetic subtraction operator used to find the difference between two numbers. In Syntax 2, the - operator is used as the unary negation operator to indicate the negative value of an expression.

If one or both expressions are Null expressions, result is Null. If an expression is Empty, it is treated as if it were 0.
Not Operator

Description

Performs logical negation on an expression.

Syntax

\[ \text{result} = \textbf{Not} \ \text{expression} \]

The \textbf{Not} operator syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>Any numeric variable.</td>
</tr>
<tr>
<td>expression</td>
<td>Any \textit{expression}.</td>
</tr>
</tbody>
</table>

Remarks

The following table illustrates how \text{result} is determined:

<table>
<thead>
<tr>
<th>If \textit{expression} is</th>
<th>Then \text{result} is</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>\textbf{Null}</td>
<td>\textbf{Null}</td>
</tr>
</tbody>
</table>

In addition, the \textbf{Not} operator inverts the bit values of any variable and sets the corresponding bit in \text{result} according to the following table:

<table>
<thead>
<tr>
<th>Bit in \textit{expression}</th>
<th>Bit in \text{result}</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
**Function**

**See Also**

**Description**

Returns the current date and time according to the setting of your computer's system date and time.

**Syntax**

```
Now
```

**Remarks**

The following example uses the `Now` function to return the current date and time:

```
Dim MyVar
MyVar = Now ' MyVar contains the current date and time.
```
The **Nothing** keyword in VBScript is used to disassociate an object variable from any actual object. Use the **Set** statement to assign **Nothing** to an object variable. For example:

```
Set MyObject = Nothing
```

Several object variables can refer to the same actual object. When **Nothing** is assigned to an object variable, that variable no longer refers to any actual object. When several object variables refer to the same object, memory and system resources associated with the object to which the variables refer are released only after all of them have been set to **Nothing**, either explicitly using **Set**, or implicitly after the last object variable set to **Nothing** goes out of **scope**.
See Also

Description

The **Null** keyword is used to indicate that a variable contains no valid data. This is not the same thing as **Empty**.
Number Property

See Also  Applies to

Description

Returns or sets a numeric value specifying an error. Number is the Err object's default property.

Syntax

```
object.Number [= errornumber]
```

The Number property syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>Always the Err object.</td>
</tr>
<tr>
<td>errornumber</td>
<td>An integer representing a VBScript error number or an SCODE error value.</td>
</tr>
</tbody>
</table>

Remarks

When returning a user-defined error from an Automation object, set Err.Number by adding the number you selected as an error code to the constant vbObjectError.

The following code illustrates the use of the Number property:

```
On Error Resume Next
Err.Raise vbObjectError + 1, "SomeObject" ' Raise Object Error #
MsgBox ("Error # " & CStr(Err.Number) & " " & Err.Description) ' Clear the error.
Err.Clear
```
Function

See Also

Description

Returns a string representing the octal value of a number.

Syntax

Oct(number)

The number argument is any valid expression.

Remarks

If number is not already a whole number, it is rounded to the nearest whole number before being evaluated.

<table>
<thead>
<tr>
<th>If number is</th>
<th>Oct returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Null</td>
<td>Null.</td>
</tr>
<tr>
<td>Empty</td>
<td>Zero (0).</td>
</tr>
<tr>
<td>Any other number</td>
<td>Up to 11 octal characters,</td>
</tr>
</tbody>
</table>

You can represent octal numbers directly by preceding numbers in the proper range with &O. For example, &O10 is the octal notation for decimal 8.

The following example uses the Oct function to return the octal value of a number:

Dim MyOct
MyOct = Oct(459)  ' Returns 713.
On Error Statement

See Also

Description

Enables or disables error-handling.

Syntax

On Error Resume Next  
On Error GoTo 0

Remarks

If you don't use an On Error Resume Next statement anywhere in your code, any run-time error that occurs can cause an error message to be displayed and code execution stopped. However, the exact behavior is determined by the host running the code. The host can sometimes opt to handle such errors differently. In some cases, the script debugger may be invoked at the point of the error. In still other cases, there may be no apparent indication that any error occurred because the host does not to notify the user. Again, this is purely a function of how the host handles any errors that occur.

Within any particular procedure, an error is not necessarily fatal as long as error-handling is enabled somewhere along the call stack. If local error-handling is not enabled in a procedure and an error occurs, control is passed back through the call stack until a procedure with error-handling enabled is found and the error is handled at that point. If no procedure in the call stack is found to have error-handling enabled, an error message is displayed at that point and execution stops or the host handles the error as appropriate.

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 allows execution to continue despite a run-time error. You can then build the error-handling routine inline 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. When a procedure is exited, the error-handling capability reverts to whatever error-handling was in place before entering the exited procedure.

Use **On Error GoTo 0** to disable error handling if you have previously enabled it using **On Error Resume Next**.

The following example illustrates use of the **On Error Resume Next** statement:

```vba
On Error Resume Next
Err.Raise 6 ' Raise an overflow error.
MsgBox "Error # " & CStr(Err.Number) & " " & Err.Description
Err.Clear ' Clear the error.
```
Operator Precedence

See Also

Description

When several operations occur in an expression, each part is evaluated and resolved in a predetermined order called operator precedence. Parentheses can be used to override the order of precedence and force some parts of an expression to be evaluated before other parts. Operations within parentheses are always performed before those outside. Within parentheses, however, normal operator precedence is maintained.

When expressions contain operators from more than one category, arithmetic operators are evaluated first, comparison operators are evaluated next, and logical operators are evaluated last. Comparison operators all have equal precedence; that is, they are evaluated in the left-to-right order in which they appear. Arithmetic and logical operators are evaluated in the following order of precedence:

<table>
<thead>
<tr>
<th>Arithmetic</th>
<th>Comparison</th>
<th>Logical</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exponentiation (^)</td>
<td>Equality (=)</td>
<td>Not</td>
</tr>
<tr>
<td>Negation (-)</td>
<td>Inequality (&lt;&gt;)</td>
<td>And</td>
</tr>
<tr>
<td>Multiplication and division (*, /)</td>
<td>Less than (&lt;)</td>
<td>Or</td>
</tr>
<tr>
<td>Integer division ()</td>
<td>Greater than (&gt;)</td>
<td>Xor</td>
</tr>
<tr>
<td>Modulus arithmetic (Mod)</td>
<td>Less than or equal to (&lt;=)</td>
<td>Eqv</td>
</tr>
<tr>
<td>Addition and subtraction (+, -)</td>
<td>Greater than or equal to (&gt;=)</td>
<td>Imp</td>
</tr>
<tr>
<td>String concatenation (&amp;)</td>
<td>Is</td>
<td>&amp;</td>
</tr>
</tbody>
</table>

When multiplication and division occur together in an expression, each operation is evaluated as it occurs from left to right. Likewise, when addition and subtraction occur together in an expression, each operation is evaluated in order of appearance from left to right.

The string concatenation operator (&) is not an arithmetic operator, but in precedence it does fall
after all arithmetic operators and before all comparison operators. The `is` operator is an object reference comparison operator. It does not compare objects or their values; it checks only to determine if two object references refer to the same object.
Option Explicit Statement

Description

Forces explicit declaration of all variables in a script.

Syntax

Option Explicit

Remarks

If used, the Option Explicit statement must appear in a script before any other statements.

When you use the Option Explicit statement, you must explicitly declare all variables using the Dim, Private, Public, or ReDim statements. If you attempt to use an undeclared variable name, an error occurs.

Tip  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.

The following example illustrates use of the Option Explicit statement:

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

Description

Performs a logical disjunction on two expressions.

Syntax

\[ \text{result} = \text{expression1 Or expression2} \]

The \text{Or} operator syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{result}</td>
<td>Any numeric variable.</td>
</tr>
<tr>
<td>\text{expression1}</td>
<td>Any \text{expression}.</td>
</tr>
<tr>
<td>\text{expression2}</td>
<td>Any expression.</td>
</tr>
</tbody>
</table>

Remarks

If either or both expressions evaluate to \textbf{True}, \textit{result} is \textbf{True}. The following table illustrates how \textit{result} is determined:

<table>
<thead>
<tr>
<th>If \text{expression1} is</th>
<th>And \text{expression2} is</th>
<th>Then \text{result} is</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>\textbf{Null}</td>
<td>True</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>\textbf{Null}</td>
<td>\textbf{Null}</td>
</tr>
<tr>
<td>\textbf{Null}</td>
<td>True</td>
<td>True</td>
</tr>
</tbody>
</table>
The **Or** operator also performs a *bitwise comparison* of identically positioned bits in two *numeric expressions* and sets the corresponding bit in *result* according to the following table:

<table>
<thead>
<tr>
<th>If bit in <em>expression1</em> is</th>
<th>And bit in <em>expression2</em> is</th>
<th>Then <em>result</em> is</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Pattern Property

Description

Sets or returns the regular expression pattern being searched for.

Syntax

```object.Pattern [ = "searchstring"]```

The Pattern property syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>Required. Always a RegExp object variable.</td>
</tr>
<tr>
<td>searchstring</td>
<td>Optional. Regular string expression being searched for. May include any of the regular expression characters defined in the table in the Settings section.</td>
</tr>
</tbody>
</table>

Settings

Special characters and sequences are used in writing patterns for regular expressions. The following table describes and gives an example of the characters and sequences that can be used.

<table>
<thead>
<tr>
<th>Character</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\</td>
<td>Marks the next character as either a special character or a literal. For example, &quot;n&quot; matches the character &quot;n&quot;. &quot;\n&quot; matches a newline character. The sequence &quot;\&quot; matches &quot;&quot; and &quot;(&quot; matches &quot;.&quot;.</td>
</tr>
<tr>
<td>^</td>
<td>Matches the beginning of input.</td>
</tr>
<tr>
<td>$</td>
<td>Matches the end of input.</td>
</tr>
<tr>
<td>*</td>
<td>Matches the preceding character zero or more times. For example, &quot;zo*&quot; matches either &quot;z&quot; or &quot;zoo&quot;.</td>
</tr>
</tbody>
</table>
+ Matches the preceding character one or more times. For example, "zo+" matches "zoo" but not "z".

? Matches the preceding character zero or one time. For example, "a?ve?" matches the "ve" in "never".

. Matches any single character except a newline character.

(pattern) Matches pattern and remembers the match. The matched substring can be retrieved from the resulting Matches collection, using Item [0]...[n]. To match parentheses characters ( ), use "(" or ")".

x|y Matches either x or y. For example, "z|food" matches "z" or "food". "(z|f)oo" matches "zoo" or "food".

{n} n is a nonnegative integer. Matches exactly n times. For example, "o{2}" does not match the "o" in "Bob," but matches the first two o's in "foooood".

{n,} n is a nonnegative integer. Matches at least n times. For example, "o{2,}" does not match the "o" in "Bob" and matches all the o's in "foooood." "o{1,}" is equivalent to "o+". "o{0,}" is equivalent to "o*".

{m,n} m and n are nonnegative integers. Matches at least n and at most m times. For example, "o{1,3}" matches the first three o's in "fooooood." "o{0,1}" is equivalent to "o?".

[xyz] A character set. Matches any one of the enclosed characters. For example, "[abc]" matches the "a" in "plain".

[^xyz] A negative character set. Matches any character not enclosed. For example, "[^abc]" matches the "p" in "plain".

[a-z] A range of characters. Matches any character in the specified range. For example, "[a-z]" matches any lowercase alphabetic character in the range "a" through "z".

[^m-z] A negative range characters. Matches any character not in the specified range. For example, "[^m-z]" matches any character not in the range "m" through "z".

\b Matches a word boundary, that is, the position between a word and a space. For example, "er\b" matches the "er" in "never" but not the "er" in "verb".

\B Matches a nonword boundary. "ea*r\B" matches the "ear" in "never early".

\d Matches a digit character. Equivalent to [0-9].

\D Matches a nondigit character. Equivalent to [^0-9].

\f Matches a form-feed character.

\n Matches a newline character.

\r Matches a carriage return character.

\s Matches any white space including space, tab, form-feed, etc. Equivalent to " [\f\n\r\t\v]".

\S Matches any nonwhite space character. Equivalent to "[^\f\n\r\t\v]".

\t Matches a tab character.

\w Matches a vertical tab character.

\w Matches any word character including underscore. Equivalent to "[A-Za-z0-9_]".

\W Matches any nonword character. Equivalent to "[^A-Za-z0-9_]".

\num Matches num, where num is a positive integer. A reference back to remembered matches. For example, "(\.)1" matches two consecutive identical characters.

\n Matches n, where n is an octal escape value. Octal escape values must be 1, 2, or 3 digits long. For example, "\11" and "\011" both match a tab character. "\0011" is the equivalent of "\011" & "1". Octal escape values must not exceed 256. If they do, only the first two digits comprise the expression. Allows ASCII codes to be used in regular expressions.

\n Matches n, where n is a hexadecimal escape value. Hexadecimal escape values must
be exactly two digits long. For example, "\x41" matches "A". "\x041" is equivalent to "\x04" & "1". Allows ASCII codes to be used in regular expressions.

Remarks

The following code illustrates the use of the **Pattern** property:

```vbscript
Function RegExpTest(patrn, strng)
    Dim regEx, Match, Matches ' Create variable.
    Set regEx = New RegExp ' Create a regular expression.
    regEx.Pattern = patrn ' Set pattern.
    regEx.IgnoreCase = True ' Set case insensitivity.
    regEx.Global = True ' Set global applicability.
    Set Matches = regEx.Execute(strng) ' Execute search.
    For Each Match in Matches ' Iterate Matches collection.
        RetStr = RetStr & "Match found at position "
        RetStr = RetStr & Match.FirstIndex & ". Match Value is "
        RetStr = RetStr & Match.Value & "." & vbCRLF
    Next
    RegExpTest = RetStr
End Function

MsgBox(RegExpTest("is.", "IS1 is2 IS3 is4"))
```
Private

Statement

Description

Declares private variables and allocates storage space. Declares, in a Class block, a private variable.

Syntax

```
Private varname[[[subscripts]]][, varname[[[subscripts]]]] . . .
```

The Private statement syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>varname</td>
<td>Name of the variable; follows standard variable naming conventions.</td>
</tr>
<tr>
<td>subscripts</td>
<td>Dimensions of an array variable; up to 60 multiple dimensions may be declared. The subscripts argument uses the following syntax:</td>
</tr>
<tr>
<td></td>
<td>upper [, upper] . . .</td>
</tr>
<tr>
<td></td>
<td>The lower bound of an array is always zero.</td>
</tr>
</tbody>
</table>

Remarks

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

A variable that refers to an 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 is initialized as **Empty**.

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.

---

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

---

The following example illustrates use of the **Private** statement:

```
Private MyNumber       ' Private Variant variable.
Private MyArray(9)    ' Private array variable.
                     ' Multiple Private declarations of Variant variables.
Private MyNumber, MyVar, YourNumber
```

**Property Get Statement**

### Description

Declares, in a **Class** block, the name, **arguments**, and code that form the body of a **Property** procedure that gets (returns) the value of a **property**.

### Syntax

```
[Public [Default] [Private] Property Get name [(arglist)]
  [statements]
  [[Set] name = expression]
  [Exit Property]
  [statements]
  [[Set] name = expression]
End Property
```

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

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Public</td>
<td>Indicates that the <strong>Property Get</strong> procedure is accessible to all other <strong>procedures</strong> in all scripts.</td>
</tr>
<tr>
<td>Default</td>
<td>Used only with the <strong>Public</strong> keyword to indicate that the property defined in the <strong>Property Get</strong> procedure is the default property for the <strong>class</strong>.</td>
</tr>
<tr>
<td>Private</td>
<td>Indicates that the <strong>Property Get</strong> procedure is accessible only to other procedures in the <strong>Class</strong> block where it's declared.</td>
</tr>
<tr>
<td>name</td>
<td>Name of the <strong>Property Get</strong> procedure; follows standard <strong>variable</strong> naming conventions, except that the name can be the same as a <strong>Property Let</strong> or <strong>Property Set</strong> procedure in the same <strong>Class</strong> block.</td>
</tr>
<tr>
<td>arglist</td>
<td>List of variables representing arguments that are passed to the <strong>Property Get</strong> procedure.</td>
</tr>
</tbody>
</table>
arglist | when it is called. Multiple arguments are separated by commas. The name of each argument in a \texttt{Property Get} procedure must be the same as the corresponding argument in a \texttt{Property Let} procedure (if one exists).

statements | Any group of statements to be executed within the body of the \texttt{Property Get} procedure.

Set | \texttt{Keyword} used when assigning an object as the return value of a \texttt{Property Get} procedure.

expression | Return value of the \texttt{Property Get} procedure.

**Remarks**

If not explicitly specified using either \texttt{Public} or \texttt{Private}, \texttt{Property Get} procedures are public by default, that is, they are visible to all other procedures in your script. The value of local variables in a \texttt{Property Get} procedure is not preserved between calls to the procedure.

You can't define a \texttt{Property Get} procedure inside any other procedure (e.g. \texttt{Function} or \texttt{Property Let}).

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

Like a \texttt{Sub} and \texttt{Property Let} procedure, a \texttt{Property Get} procedure is a separate procedure that can take arguments, perform a series of statements, and change the value of its arguments. However, unlike a \texttt{Sub} and \texttt{Property Let}, you can use a \texttt{Property Get} procedure on the right side of an \texttt{expression} in the same way you use a \texttt{Function} or property name when you want to return the value of a property.
Property Let Statement

Description

Declares, in a Class block, the name, arguments, and code that form the body of a Property procedure that assigns (sets) the value of a property.

Syntax

[Public | Private] Property Let name ([arglist,] value)
   [statements]
   [Exit Property]
   [statements]
End Property

The Property Let statement syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Public</td>
<td>Indicates that the Property Let procedure is accessible to all other procedures in all scripts.</td>
</tr>
<tr>
<td>Private</td>
<td>Indicates that the Property Let procedure is accessible only to other procedures in the Class block where it’s declared.</td>
</tr>
<tr>
<td>name</td>
<td>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 Class block.</td>
</tr>
<tr>
<td>arglist</td>
<td>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 of each argument in a Property Let procedure must be the same as the corresponding argument in a Property Get procedure. In addition, the Property Let procedure will always have one more argument than its corresponding Property Get procedure. That argument is the value being assigned to the property.</td>
</tr>
</tbody>
</table>
### Remarks

If not explicitly specified using either **Public** or **Private**, **Property Let** procedures are public by default, that is, they are visible to all other procedures in your script. The value of local variables in a **Property Let** procedure is not preserved between calls to the procedure.

You can't define a **Property Let** procedure inside any other procedure (e.g. **Function** or **Property Get**).

The **Exit Property** statement causes an immediate exit from a **Property Let** procedure. Program execution continues with the statement that follows 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.
**Property Set Statement**

**See Also**

**Description**

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

**Syntax**

```
[Public | Private] Property Set name([arglist,] reference)
  [statements]
  [Exit Property]
  [statements]
End Property
```

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

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Public</td>
<td>Indicates that the <strong>Property Set</strong> procedure is accessible to all other procedures in all scripts.</td>
</tr>
<tr>
<td>Private</td>
<td>Indicates that the <strong>Property Set</strong> procedure is accessible only to other procedures in the <strong>Class</strong> block where it's declared.</td>
</tr>
<tr>
<td>name</td>
<td>Name of the <strong>Property Set</strong> procedure; follows standard variable naming conventions, except that the name can be the same as a <strong>Property Get</strong> or <strong>Property Let</strong> procedure in the same <strong>Class</strong> block.</td>
</tr>
<tr>
<td>arglist</td>
<td>List of variables representing arguments that are passed to the <strong>Property Set</strong> procedure when it is called. Multiple arguments are separated by commas. In addition, the <strong>Property Set</strong> procedure will always have one more argument than its corresponding <strong>Property Get</strong> procedure. That argument is the object being assigned to the property.</td>
</tr>
<tr>
<td>reference</td>
<td>Variable containing the object reference used on the right side of the object reference assignment.</td>
</tr>
</tbody>
</table>
Any group of statements to be executed within the body of the **Property Set** procedure.

**Note** Version 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. That argument is referred to as **reference** in the preceding syntax.

**Remarks**

If not explicitly specified using either **Public** or **Private**, **Property Set** procedures are public by default, that is, they are visible to all other procedures in your script. The value of local variables in a **Property Set** procedure is not preserved between calls to the procedure.

You can't define a **Property Set** procedure inside any other procedure (e.g. **Function** or **Property Let**).

The **Exit Property** statement causes an immediate exit from a **Property Set** procedure. Program execution continues with the statement that follows 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).
Public Statement

Description

Declares public variables and allocates storage space. Declares, in a **Class** block, a private variable.

Syntax

```
Public varname[([subscripts])][, varname[([subscripts])]] . . .
```

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

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>varname</td>
<td>Name of the variable; follows standard variable naming conventions.</td>
</tr>
<tr>
<td>subscripts</td>
<td>Dimensions of an array variable; up to 60 multiple dimensions may be declared. The subscripts argument uses the following syntax:</td>
</tr>
<tr>
<td></td>
<td><code>upper [, upper] . . .</code></td>
</tr>
<tr>
<td></td>
<td>The lower bound of an array is always zero.</td>
</tr>
</tbody>
</table>

Remarks

**Public** statement variables are available to all procedures in all scripts.

A variable that refers to an 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 is initialized as **Empty**.

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.

The following example illustrates the use of the **Public** statement:

```vba
Public MyNumber           ' Public Variant variable.
Public MyArray(9)         ' Public array variable.
                          ' Multiple Public declarations of Variant variables.
Public MyNumber, MyVar, YourNumber
```
# Raise Method

**Description**

Generates a run-time error.

**Syntax**

```vbscript
object.Raise(number, source, description, helpfile, helpcontext)
```

The `Raise` method has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>Always the <code>Err</code> object.</td>
</tr>
<tr>
<td>number</td>
<td>A <code>Long</code> integer subtype that identifies the nature of the error. VBScript errors (both VBScript-defined and user-defined errors) are in the range 0–65535.</td>
</tr>
<tr>
<td>source</td>
<td>A string expression naming the object or application that originally generated the error. When setting this property for an Automation object, use the form <code>project.class</code>. If nothing is specified, the programmatic ID of the current VBScript project is used.</td>
</tr>
<tr>
<td>description</td>
<td>A string expression describing the error. If unspecified, the value in <code>number</code> is examined. If it can be mapped to a VBScript run-time error code, a string provided by VBScript is used as <code>description</code>. If there is no VBScript error corresponding to <code>number</code>, a generic error message is used.</td>
</tr>
<tr>
<td>helpfile</td>
<td>The fully qualified path to the Help file in which help on this error can be found. If unspecified,</td>
</tr>
<tr>
<td>helpfile</td>
<td>VBScript uses the fully qualified drive, path, and file name of the VBScript Help file.</td>
</tr>
<tr>
<td>---------------</td>
<td>--------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>helpcontext</td>
<td>The context ID identifying a topic within helpfile that provides help for the error. If omitted, the VBScript Help file context ID for the error corresponding to the number property is used, if it exists.</td>
</tr>
</tbody>
</table>

**Remarks**

All the arguments are optional except number. If you use **Raise**, however, without specifying some arguments, and the property settings of the **Err** object contain values that have not been cleared, those values become the values for your error.

When setting the **number** property to your own error code in an **Automation object**, you add your error code number to the constant **vbObjectError**. For example, to generate the error number 1050, assign **vbObjectError** + 1050 to the **number** property.

The following example illustrates use of the **Raise** method:

```vbscript
On Error Resume Next
Err.Raise 6 ' Raise an overflow error.
MsgBox ("Error # " & CStr(Err.Number) & " " & Err.Description)
Err.Clear  ' Clear the error.
```
Randomize Statement

Description

Initializes the random-number generator.

Syntax

Randomize [number]

The number argument can be any valid numeric expression.

Remarks

Randomize uses number to initialize the Rnd function's random-number generator, giving it a new seed value. If you omit number, the value returned by the system timer is used as the new seed value.

If Randomize is not used, the Rnd function (with no arguments) uses the same number as a seed the first time it is called, and thereafter uses the last generated number as a seed value.

Note  To repeat sequences of random numbers, call Rnd with a negative argument immediately before using Randomize with a numeric argument. Using Randomize with the same value for number does not repeat the previous sequence.

The following example illustrates use of the Randomize statement:

Dim MyValue, Response
Randomize ' Initialize random-numbe
Do Until Response = vbNo
  MyValue = Int((6 * Rnd) + 1)  ' Generate random value
  MsgBox MyValue
  Response = MsgBox("Roll again? ", vbYesNo)
Loop
ReDim

Statement

Description

Declares dynamic-array variables, and allocates or reallocates storage space at procedure level.

Syntax

ReDim [Preserve] varname(subscripts) [, varname(subscripts)] . . .

The ReDim statement syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preserve</td>
<td>Preserves the data in an existing array when you change the size of the last dimension.</td>
</tr>
<tr>
<td>varname</td>
<td>Name of the variable; follows standard variable naming conventions.</td>
</tr>
</tbody>
</table>
| subscripts | Dimensions of an array variable; up to 60 multiple dimensions may be declared. The subscripts argument uses the following syntax: <br>upper [,upper] . . .  
|          | The lower bound of an array is always zero.                                  |

Remarks

The ReDim statement is used to size or resize a dynamic array that has already been formally declared using a Private, Public, or Dim statement with empty parentheses (without dimension subscripts). You can use the ReDim statement repeatedly to change the number of elements and
dimensions in an array.

If you use the **Preserve** keyword, you can resize only the last array dimension, and you can't change the number of dimensions at all. For example, if your array has only one dimension, you can resize that dimension because it is the last and only dimension. However, if your array has two or more dimensions, you can change the size of only the last dimension and still preserve the contents of the array.

The following example shows how you can increase the size of the last dimension of a dynamic array without erasing any existing data contained in the array.

```
ReDim X(10, 10, 10)
...
ReDim Preserve X(10, 10, 15)
```

---

**Caution**  If you make an array smaller than it was originally, data in the eliminated elements is lost.

---

When variables are initialized, a numeric variable is initialized to 0 and a string variable is initialized to a zero-length string ("""). A variable that refers to an 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**.
Description

Provides simple regular expression support.

Remarks

The following code illustrates the use of the RegExp object:

```vbscript
Function RegExpTest(pattern, string)
    Dim regEx, Match, Matches
    ' Create variable.
    Set regEx = New RegExp
    ' Create a regular expression.
    regEx.Pattern = pattern
    ' Set pattern.
    regEx.IgnoreCase = True
    ' Set case insensitivity.
    regEx.Global = True
    ' Set global applicability.
    Set Matches = regEx.Execute(string)
    ' Execute search.
    For Each Match in Matches
        ' Iterate Matches collection.
        RetStr = RetStr & "Match found at position "
        RetStr = RetStr & Match.FirstIndex & ". Match Value is "
        RetStr = RetStr & Match.Value & "." & vbCrLf
    Next
    RegExpTest = RetStr
End Function

MsgBox(RegExpTest("is.", "IS1 is2 IS3 is4"))
```
Statement

Description

Includes explanatory remarks in a program.

Syntax

Rem comment

or

' comment

The comment argument is the text of any comment you want to include. After the Rem keyword, a space is required before comment.

Remarks

As shown in the syntax section, you can use an apostrophe (' ) instead of the Rem keyword. If the Rem keyword follows other statements on a line, it must be separated from the statements by a colon. However, when you use an apostrophe, the colon is not required after other statements.

The following example illustrates the use of the Rem statement:

Dim MyStr1, MyStr2
MyStr1 = "Hello" : Rem Comment after a statement separated by a
MyStr2 = "Goodbye" ' This is also a comment; no colon is neede
Rem Comment on a line with no code; no colon is needed.
**Replace**

**Function**

**See Also**

**Description**

Returns a string in which a specified substring has been replaced with another substring a specified number of times.

**Syntax**

\[
\text{Replace}(\text{expression, find, replacewith[, start[, count[, compare]]]})
\]

The *Replace* function syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>expression</em></td>
<td>Required. [String expression] containing substring to replace.</td>
</tr>
<tr>
<td><em>find</em></td>
<td>Required. Substring being searched for.</td>
</tr>
<tr>
<td><em>replacewith</em></td>
<td>Required. Replacement substring.</td>
</tr>
<tr>
<td><em>start</em></td>
<td>Optional. Position within <em>expression</em> where substring search is to begin. If omitted, 1 is assumed. Must be used in conjunction with <em>count</em>.</td>
</tr>
<tr>
<td><em>count</em></td>
<td>Optional. Number of substring substitutions to perform. If omitted, the default value is -1, which means make all possible substitutions. Must be used in conjunction with <em>start</em>.</td>
</tr>
<tr>
<td><em>compare</em></td>
<td>Optional. Numeric value indicating the kind of comparison to use when evaluating substrings. See Settings section for values. If omitted, the default value is 0, which means perform a binary comparison.</td>
</tr>
</tbody>
</table>
Settings

The compare argument can have the following values:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vbBinaryCompare</td>
<td>0</td>
<td>Perform a binary comparison.</td>
</tr>
<tr>
<td>vbTextCompare</td>
<td>1</td>
<td>Perform a textual comparison.</td>
</tr>
</tbody>
</table>

Return Values

**Replace** returns the following values:

<table>
<thead>
<tr>
<th>If</th>
<th>Replace returns</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>expression</em> is zero-length</td>
<td>Zero-length string (&quot;&quot;).</td>
</tr>
<tr>
<td><em>expression</em> is Null</td>
<td>An error.</td>
</tr>
<tr>
<td><em>find</em> is zero-length</td>
<td>Copy of <em>expression</em>.</td>
</tr>
<tr>
<td><em>replacewith</em> is zero-length</td>
<td>Copy of <em>expression</em> with all occurrences of <em>find</em> removed.</td>
</tr>
<tr>
<td><em>start</em> &gt; <em>Len(expression)</em></td>
<td>Zero-length string.</td>
</tr>
<tr>
<td><em>count</em> is 0</td>
<td>Copy of <em>expression</em>.</td>
</tr>
</tbody>
</table>

Remarks

The return value of the **Replace** function is a string, with substitutions made, that begins at the position specified by *start* and and concludes at the end of the *expression* string. It is not a copy of the original string from start to finish.

The following example uses the **Replace** function to return a string:

```vba
Dim MyString
MyString = Replace("XXpXXPXXp", "p", "Y") ' of the string. R
```
MyString = Replace("XXpXXPXXp", "p", "Y", ' Returns "YXX

Replace Method

**Description**

Replaces text found in a regular expression search.

**Syntax**

```object.Replace(string1, string2)```

The **Replace** method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>Required. Always the name of a <code>RegExp</code> object.</td>
</tr>
<tr>
<td>string1</td>
<td>Required. <code>String1</code> is the text string in which the text replacement is to occur.</td>
</tr>
<tr>
<td>string2</td>
<td>Required. <code>String2</code> is the replacement text string.</td>
</tr>
</tbody>
</table>

**Remarks**

The actual pattern for the text being replaced is set using the `Pattern` property of the `RegExp` object.

The **Replace** method returns a copy of `string1` with the text of `RegExp.Pattern` replaced with `string2`. If no match is found, a copy of `string1` is returned unchanged.

The following code illustrates use of the **Replace** method:

```vbnet
Function ReplaceTest(patrn, replStr)
```

Dim regEx, str1 ' Create variables.
str1 = "The quick brown fox jumped over the lazy dog."
Set regEx = New RegExp ' Create regular expression.
regEx.Pattern = patrn ' Set pattern.
regEx.IgnoreCase = True ' Make case insensitive.
ReplaceTest = regEx.Replace(str1, replStr) ' Make replacement.
End Function

MsgBox(ReplaceTest("fox", "cat")) ' Replace 'fox' with 'c

In addition, the Replace method can replace subexpressions in the pattern. The following call to the function shown in the previous example swaps each pair of words in the original string:

MsgBox(ReplaceText("(\S+)(\s+)(\S+)", "$3$2$1")) ' Swap pair
**RGB Function**

**Description**

Returns a whole number representing an RGB color value.

**Syntax**

`RGB(red, green, blue)`

The **RGB** function has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>red</td>
<td>Required. Number in the range 0-255 representing the red component of the color.</td>
</tr>
<tr>
<td>green</td>
<td>Required. Number in the range 0-255 representing the green component of the color.</td>
</tr>
<tr>
<td>blue</td>
<td>Required. Number in the range 0-255 representing the blue component of the color.</td>
</tr>
</tbody>
</table>

**Remarks**

Application methods and properties that accept a color specification expect that specification to be a number representing an RGB color value. An RGB color value specifies the relative intensity of red, green, and blue to cause a specific color to be displayed.

The low-order byte contains the value for red, the middle byte contains the value for green, and the high-order byte contains the value for blue.
For applications that require the byte order to be reversed, the following function will provide the same information with the bytes reversed:

```
Function RevRGB(red, green, blue)
    RevRGB = CLng(blue + (green * 256) + (red * 6
End Function
```

The value for any argument to RGB that exceeds 255 is assumed to be 255.
Right Function

See Also

Description

Returns a specified number of characters from the right side of a string.

Syntax

Right(string, length)

The Right function syntax has these arguments:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td><em>String expression</em> from which the rightmost characters are returned. If <em>string</em> contains <em>Null</em>, <em>Null</em> is returned.</td>
</tr>
<tr>
<td>length</td>
<td><em>Numeric expression</em> indicating how many characters to return. If 0, a zero-length string is returned. If greater than or equal to the number of characters in <em>string</em>, the entire string is returned.</td>
</tr>
</tbody>
</table>

Remarks

To determine the number of characters in *string*, use the **Len** function.

The following example uses the **Right** function to return a specified number of characters from the right side of a string:

```
Dim AnyString, MyStr
AnyString = "Hello World" ' Define string.
MyStr = Right(AnyString, 1) ' Returns "d".
MyStr = Right(AnyString, 6) ' Returns " World".
```
MyStr = Right(AnyString, 20) ' Returns "Hello W

Note  The RightB function is used with byte data contained in a string. Instead of specifying the number of characters to return, length specifies the number of bytes.
Function

See Also

Description

Returns a random number.

Syntax

```
Rnd[(number)]
```

The `number` argument can be any valid numeric expression.

Remarks

The `Rnd` function returns a value less than 1 but greater than or equal to 0. The value of `number` determines how `Rnd` generates a random number:

<table>
<thead>
<tr>
<th>If <code>number</code> is</th>
<th><code>Rnd</code> generates</th>
</tr>
</thead>
<tbody>
<tr>
<td>Less than zero</td>
<td>The same number every time, using <code>number</code> as the seed.</td>
</tr>
<tr>
<td>Greater than zero</td>
<td>The next random number in the sequence.</td>
</tr>
<tr>
<td>Equal to zero</td>
<td>The most recently generated number.</td>
</tr>
<tr>
<td>Not supplied</td>
<td>The next random number in the sequence.</td>
</tr>
</tbody>
</table>

For any given initial seed, the same number sequence is generated because each successive call to the `Rnd` function uses the previous number as a seed for the next number in the sequence.

Before calling `Rnd`, use the `Randomize` statement without an argument to initialize the random-number generator with a seed based on the system timer.

To produce random integers in a given range, use this formula:
Int((upperbound - lowerbound + 1) * Rnd + lowerb)

Here, upperbound is the highest number in the range, and lowerbound is the lowest number in the range.

---

**Note** To repeat sequences of random numbers, call Rnd with a negative argument immediately before using Randomize with a numeric argument. Using Randomize with the same value for number does not repeat the previous sequence.
Round Function

Description

Returns a number rounded to a specified number of decimal places.

Syntax

**Round(expression[, numdecimalplaces])**

The Round function syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>expression</td>
<td>Required. Numeric expression being rounded.</td>
</tr>
<tr>
<td>numdecimalplaces</td>
<td>Optional. Number indicating how many places to the right of the decimal are included in the rounding. If omitted, integers are returned by the Round function.</td>
</tr>
</tbody>
</table>

Remarks

The following example uses the Round function to round a number to two decimal places:

```
Dim MyVar, pi
pi = 3.14159
MyVar = Round(pi, 2) ' MyVar contains 3.14.
```
ScriptEngine Function

Description

Returns a string representing the scripting language in use.

Syntax

ScriptEngine

Return Values

The ScriptEngine function can return any of the following strings:

<table>
<thead>
<tr>
<th>String</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VBScript</td>
<td>Indicates that Microsoft® Visual Basic® Scripting Edition is the current scripting engine.</td>
</tr>
<tr>
<td>JScript</td>
<td>Indicates that Microsoft JScript® is the current scripting engine.</td>
</tr>
<tr>
<td>VBA</td>
<td>Indicates that Microsoft Visual Basic for Applications is the current scripting engine.</td>
</tr>
</tbody>
</table>

Remarks

The following example uses the ScriptEngine function to return a string describing the scripting language in use:

```
Function GetScriptEngineInfo
    Dim s
    s = ""
    ' Build string with necessary info.
```
s = ScriptEngine & " Version "
s = s & ScriptEngineMajorVersion & "."
s = s & ScriptEngineMinorVersion & "."
s = s & ScriptEngineBuildVersion
GetScriptEngineInfo = s       ' Return the results.
End Function
**ScriptEngineBuildVersion Function**

**Description**

Returns the build version number of the scripting engine in use.

**Syntax**

```plaintext
ScriptEngineBuildVersion
```

**Remarks**

The return value corresponds directly to the version information contained in the DLL for the scripting language in use.

The following example uses the `ScriptEngineBuildVersion` function to return the build version number of the scripting engine:

```plaintext
Function GetScriptEngineInfo
    Dim s
    s = "" ' Build string with necessary info.
    s = ScriptEngine & " Version 
    s = s & ScriptEngineMajorVersion & "."
    s = s & ScriptEngineMinorVersion & "."
    s = s & ScriptEngineBuildVersion
    GetScriptEngineInfo = s ' Return the results.
End Function
```
ScriptEngineMajorVersion Function

See Also

Description

Returns the major version number of the scripting engine in use.

Syntax

ScriptEngineMajorVersion

Remarks

The return value corresponds directly to the version information contained in the DLL for the scripting language in use.

The following example uses the ScriptEngineMajorVersion function to return the version number of the scripting engine:

Function GetScriptEngineInfo
    Dim s
    s = "" ' Build string with necessary info.
    s = ScriptEngine & " Version "
    s = s & ScriptEngineMajorVersion & "."
    s = s & ScriptEngineMinorVersion & "."
    s = s & ScriptEngineBuildVersion
    GetScriptEngineInfo = s ' Return the results.
End Function
ScriptEngineMinorVersion Function

See Also

Description

Returns the minor version number of the scripting engine in use.

Syntax

ScriptEngineMinorVersion

Remarks

The return value corresponds directly to the version information contained in the DLL for the scripting language in use.

The following example uses the ScriptEngineMinorVersion function to return the minor version number of the scripting engine:

Function GetScriptEngineInfo
    Dim s
    s = "" ' Build string with necessary info.
    s = ScriptEngine & " Version "
    s = s & ScriptEngineMajorVersion & "."
    s = s & ScriptEngineMinorVersion & "."
    s = s & ScriptEngineBuildVersion
    GetScriptEngineInfo = s ' Return the results.
End Function
Second Function

Description

Returns a whole number between 0 and 59, inclusive, representing the second of the minute.

Syntax

Second(time)

The time argument is any expression that can represent a time. If time contains Null, Null is returned.

Remarks

The following example uses the Second function to return the current second:

Dim MySec
MySec = Second(Now) ' MySec contains the number representing the current second.
Select Case Statement

See Also

Description

Executes one of several groups of statements, depending on the value of an expression.

Syntax

```
Select Case testexpression
  [Case expressionlist-n
    [statements-n]] . . .
  [Case Else expressionlist-n
    [elsestatements-n]]
End Select
```

The `Select Case` statement syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>testexpression</code></td>
<td>Any numeric or string expression.</td>
</tr>
<tr>
<td><code>expressionlist-n</code></td>
<td>Required if <code>Case</code> appears. Delimited list of one or more expressions.</td>
</tr>
<tr>
<td><code>statements-n</code></td>
<td>One or more statements executed if <code>testexpression</code> matches any part of <code>expressionlist-n</code>.</td>
</tr>
<tr>
<td><code>elsestatements-n</code></td>
<td>One or more statements executed if <code>testexpression</code> doesn't match any of the <code>Case</code> clauses.</td>
</tr>
</tbody>
</table>

Remarks
If `testexpression` matches any **Case expressionlist** expression, the statements following that **Case** clause are executed up to the next **Case** clause, or for the last clause, up to **End Select**. Control then passes to the statement following **End Select**. If `testexpression` matches an **expressionlist** expression in more than one **Case** clause, only the statements following the first match are executed.

The **Case Else** clause is used to indicate the **else** statements to be executed if no match is found between the `testexpression` and an **expressionlist** in any of the other **Case** selections. Although not required, it is a good idea to have a **Case Else** statement in your **Select Case** block to handle unforeseen `testexpression` values. If no **Case expressionlist** matches `testexpression` and there is no **Case Else** statement, execution continues at the statement following **End Select**.

**Select Case** statements can be nested. Each nested **Select Case** statement must have a matching **End Select** statement.

The following example illustrates the use of the **Select Case** statement:

```vba
Dim Color, MyVar
Sub ChangeBackgroundColor (Color)
    MyVar = lcase (Color)
    Select Case MyVar
        Case "red"    document.bgColor = "red"
        Case "green"  document.bgColor = "green"
        Case "blue"   document.bgColor = "blue"
        Case Else     MsgBox "pick another color"
    End Select
End Sub
```
Set Statement

See Also

Description

Assigns an object reference to a variable or property, or associates a procedure reference with an event.

Syntax 1

\[ \text{Set } \text{objectvar} = \{ \text{objectexpression} \mid \text{New classname} \mid \text{Nothing} \} \]

Syntax 2

\[ \text{Set } \text{object.eventname} = \text{GetRef(procname)} \]

The Set statement syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>objectvar</td>
<td>Required. Name of the variable or property; follows standard variable naming conventions.</td>
</tr>
<tr>
<td>objectexpression</td>
<td>Optional. Expression consisting of the name of an object, another declared variable of the same object type, or a function or method that returns an object of the same object type.</td>
</tr>
<tr>
<td>New</td>
<td>Keyword used to create a new instance of a class. If objectvar contained a reference to an object, that reference is released when the new one is assigned. The New keyword can only be used to create an instance of a <a href="#">class</a>.</td>
</tr>
<tr>
<td>classname</td>
<td>Optional. Name of the class being created. A class and its members are defined using the <a href="#">Class</a> statement.</td>
</tr>
<tr>
<td>Nothing</td>
<td>Optional. Discontinues association of objectvar with any specific object or class. Assigning objectvar to Nothing releases all the system and memory resources associated with the previously referenced object when no other variable refers to it.</td>
</tr>
<tr>
<td>object</td>
<td>Required. Name of the object with which event is associated.</td>
</tr>
</tbody>
</table>
**Remarks**

To be valid, `objectvar` must be an object type consistent with the object being assigned to it.

The **Dim**, **Private**, **Public**, or **ReDim** statements only declare a variable that refers to an object. No actual object is referred to until you use the **Set** statement to assign a specific object.

Generally, when you use **Set** to assign an object reference to a variable, no copy of the object is created for that variable. Instead, a reference to the object is created. More than one object variable can refer to the same object. Because these variables are references to (rather than copies of) the object, any change in the object is reflected in all variables that refer to it.

```vbscript
Function ShowFreeSpace(drvPath)
    Dim fso, d, s
    Set fso = CreateObject("Scripting.FileSystemObject")
    Set d = fso.GetDrive(fso.GetDriveName(drvPath))
    s = "Drive " & UCase(drvPath) & " - "
    s = s & d.VolumeName & "<BR>"
    s = s & "Free Space: " & FormatNumber(d.FreeSpace/1024, 0)
    s = s & " Kbytes"
    ShowFreeSpace = s
End Function
```

Using the **New** keyword allows you to concurrently create an instance of a class and assign it to an object reference variable. The variable to which the instance of the class is being assigned
must already have been declared with the **Dim** (or equivalent) statement.

Refer to the documentation for the **GetRef** function for information on using **Set** to associate a procedure with an event.
Description

Returns an integer indicating the sign of a number.

Syntax

\[ \text{Sgn(number)} \]

The number argument can be any valid numeric expression.

Return Values

The Sgn function has the following return values:

<table>
<thead>
<tr>
<th>If number is</th>
<th>Sgn returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greater than zero</td>
<td>1</td>
</tr>
<tr>
<td>Equal to zero</td>
<td>0</td>
</tr>
<tr>
<td>Less than zero</td>
<td>-1</td>
</tr>
</tbody>
</table>

Remarks

The sign of the number argument determines the return value of the Sgn function.

The following example uses the Sgn function to determine the sign of a number:

```
Dim MyVar1, MyVar2, MyVar3, MySign
MyVar1 = 12: MyVar2 = -2.4: MyVar3 = 0
MySign = Sgn(MyVar1) ' Returns 1.
```
MySign = Sgn(MyVar2)  ' Returns -1.
MySign = Sgn(MyVar3)  ' Returns 0.
**Sin Function**

**Description**

Returns the sine of an angle.

**Syntax**

```plaintext
Sin(number)
```

The `number` argument can be any valid numeric expression that expresses an angle in radians.

**Remarks**

The `Sin` function takes an angle and returns the ratio of two sides of a right triangle. The ratio is the length of the side opposite the angle divided by the length of the hypotenuse. The result lies in the range -1 to 1.

To convert degrees to radians, multiply degrees by $\pi$/180. To convert radians to degrees, multiply radians by 180/$\pi$.

The following example uses the `Sin` function to return the sine of an angle:

```plaintext
Dim MyAngle, MyCosecant
MyAngle = 1.3 ' Define angle in radians
MyCosecant = 1 / Sin(MyAngle) ' Calculate cosecant
```
Source Property

Description

Returns or sets the name of the object or application that originally generated the error.

Syntax

object.Source [= stringexpression]

The Source property syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>Always the Err object.</td>
</tr>
<tr>
<td>stringexpression</td>
<td>A string expression representing the application that generated the error.</td>
</tr>
</tbody>
</table>

Remarks

The Source property specifies a string expression that is usually the class name or programmatic ID of the object that caused the error. Use Source to provide your users with information when your code is unable to handle an error generated in an accessed object. For example, if you access Microsoft Excel and it generates a Division by zero error, Microsoft Excel sets Err.Number to its error code for that error and sets Source to Excel.Application. Note that if the error is generated in another object called by Microsoft Excel, Excel intercepts the error and sets Err.Number to its own code for Division by zero. However, it leaves the other Err object (including Source) as set by the object that generated the error.

Source always contains the name of the object that originally generated the error — your code can try to handle the error according to the error documentation of the object you accessed. If your error
handler fails, you can use the **Err** object information to describe the error to your user, using **Source** and the other **Err** to inform the user which object originally caused the error, its description of the error, and so forth.

When generating an error from code, **Source** is your application's programmatic ID.

The following code illustrates use of the **Source** property:

```vbscript
On Error Resume Next
Err.Raise 6 ' Raise an overflow error.
MsgBox ("Error # " & CStr(Err.Number) & " " & Err.Description):
Err.Clear ' Clear the error.
```
Space Function

Description

Returns a string consisting of the specified number of spaces.

Syntax

Space(number)

The number argument is the number of spaces you want in the string.

Remarks

The following example uses the Space function to return a string consisting of a specified number of spaces:

Dim MyString
MyString = Space(10) ' Returns a string with 10 spaces
MyString = "Hello" & Space(10) & "World" ' Insert 10 spaces between two strings.
Function

Description

Returns a zero-based, one-dimensional array containing a specified number of substrings.

Syntax

`Split(expression[, delimiter[, count[, compare]]])`

The `Split` function syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>expression</code></td>
<td>Required. String expression containing substrings and delimiters. If <code>expression</code> is a zero-length string, <code>Split</code> returns an empty array, that is, an array with no elements and no data.</td>
</tr>
<tr>
<td><code>delimiter</code></td>
<td>Optional. String character used to identify substring limits. If omitted, the space character (&quot; &quot;) is assumed to be the delimiter. If <code>delimiter</code> is a zero-length string, a single-element array containing the entire <code>expression</code> string is returned.</td>
</tr>
<tr>
<td><code>count</code></td>
<td>Optional. Number of substrings to be returned; -1 indicates that all substrings are returned.</td>
</tr>
<tr>
<td><code>compare</code></td>
<td>Optional. Numeric value indicating the kind of comparison to use when evaluating substrings. See Settings section for values.</td>
</tr>
</tbody>
</table>

Settings
The *compare* argument can have the following values:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vbBinaryCompare</td>
<td>0</td>
<td>Perform a binary comparison.</td>
</tr>
<tr>
<td>vbTextCompare</td>
<td>1</td>
<td>Perform a textual comparison.</td>
</tr>
</tbody>
</table>

Remarks

The following example uses the *Split* function to return an array from a string. The function performs a textual comparison of the delimiter, and returns all of the substrings.

```vbs
Dim MyString, MyArray, Msg
MyString = "VBScriptXisXfun!"
MyArray = Split(MyString, "x", -1, 1)
' MyArray(0) contains "VBScript".
' MyArray(1) contains "is".
' MyArray(2) contains "fun!".
Msg = MyArray(0) & " " & MyArray(1)
Msg = MsgBox Msg
```
Description

Returns the square root of a number.

Syntax

\texttt{Sqr(number)}

The \textit{number} argument can be any valid \textit{numeric expression} greater than or equal to 0.

Remarks

The following example uses the \texttt{Sqr} function to calculate the square root of a number:

\begin{verbatim}
Dim MySqr
MySqr = Sqr(4)   ' Returns 2.
MySqr = Sqr(23)  ' Returns 4.79583152331272.
MySqr = Sqr(0)   ' Returns 0.
MySqr = Sqr(-4)  ' Generates a run-time error.
\end{verbatim}
StrComp Function

Description

Returns a value indicating the result of a string comparison.

Syntax

\[
\text{StrComp}(\text{string1, string2[, compare]})
\]

The \text{StrComp} function syntax has these arguments:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string1</td>
<td>Required. Any valid string expression.</td>
</tr>
<tr>
<td>string2</td>
<td>Required. Any valid string expression.</td>
</tr>
<tr>
<td>compare</td>
<td>Optional. Numeric value indicating the kind of comparison to use when evaluating strings. If omitted, a binary comparison is performed. See Settings section for values.</td>
</tr>
</tbody>
</table>

Settings

The \text{compare} argument can have the following values:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vbBinaryCompare</td>
<td>0</td>
<td>Perform a binary comparison.</td>
</tr>
<tr>
<td>vbTextCompare</td>
<td>1</td>
<td>Perform a textual comparison.</td>
</tr>
</tbody>
</table>

Return Values
The **StrComp** function has the following return values:

<table>
<thead>
<tr>
<th>If</th>
<th>StrComp returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>string1 is less than string2</td>
<td>-1</td>
</tr>
<tr>
<td>string1 is equal to string2</td>
<td>0</td>
</tr>
<tr>
<td>string1 is greater than string2</td>
<td>1</td>
</tr>
<tr>
<td>string1 or string2 is <strong>Null</strong></td>
<td><strong>Null</strong></td>
</tr>
</tbody>
</table>

**Remarks**

The following example uses the **StrComp** function to return the results of a string comparison. If the third argument is 1, a textual comparison is performed; if the third argument is 0 or omitted, a binary comparison is performed.

```plaintext
Dim MyStr1, MyStr2, MyComp
MyStr1 = "ABCD": MyStr2 = "abcd"  ' Define 
MyComp = StrComp(MyStr1, MyStr2, 1)  ' Return 
MyComp = StrComp(MyStr1, MyStr2, 0)  ' Return 
MyComp = StrComp(MyStr2, MyStr1)    ' Return 
```
Since these constants are built into VBScript, you don't have to define them before using them. Use them anywhere in your code to represent the values shown for each.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vbCr</td>
<td>Chr(13)</td>
<td>Carriage return.</td>
</tr>
<tr>
<td>vbCrLf</td>
<td>Chr(13) &amp; Chr(10)</td>
<td>Carriage return–linefeed combination.</td>
</tr>
<tr>
<td>vbFormFeed</td>
<td>Chr(12)</td>
<td>Form feed; not useful in Microsoft Windows.</td>
</tr>
<tr>
<td>vbLf</td>
<td>Chr(10)</td>
<td>Line feed.</td>
</tr>
<tr>
<td>vbNewLine</td>
<td>Chr(13) &amp; Chr(10) or Chr(10)</td>
<td>Platform-specific newline character; whatever is appropriate for the platform.</td>
</tr>
<tr>
<td>vbNullChar</td>
<td>Chr(0)</td>
<td>Character having the value 0.</td>
</tr>
<tr>
<td>vbNullString</td>
<td>String having value 0</td>
<td>Not the same as a zero-length string (&quot;'&quot;); used for calling external procedures.</td>
</tr>
<tr>
<td>vbTab</td>
<td>Chr(9)</td>
<td>Horizontal tab.</td>
</tr>
<tr>
<td>vbVerticalTab</td>
<td>Chr(11)</td>
<td>Vertical tab; not useful in Microsoft Windows.</td>
</tr>
</tbody>
</table>
**String Function**

**Description**

Returns a repeating character string of the length specified.

**Syntax**

`String(number, character)`

The `String` function syntax has these arguments:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>number</code></td>
<td>Length of the returned string. If <code>number</code> contains <code>Null</code>, <code>Null</code> is returned.</td>
</tr>
<tr>
<td><code>character</code></td>
<td>Character code specifying the character or string expression whose first character is used to build the return string. If <code>character</code> contains <code>Null</code>, <code>Null</code> is returned.</td>
</tr>
</tbody>
</table>

**Remarks**

If you specify a number for `character` greater than 255, `String` converts the number to a valid character code using the formula:

\[ character \mod 256 \]

The following example uses the `String` function to return repeating character strings of the length specified:
Dim MyString
MyString = String(5, "*")    ' Returns "*****".
MyString = String(5, 42)     ' Returns "*****".
MyString = String(10, "ABC") ' Returns "AAAAA"
StrReverse Function

Description

Returns a string in which the character order of a specified string is reversed.

Syntax

StrReverse(string1)

The string1 argument is the string whose characters are to be reversed. If string1 is a zero-length string (""), a zero-length string is returned. If string1 is Null, an error occurs.

Remarks

The following example uses the StrReverse function to return a string in reverse order:

Dim MyStr
MyStr = StrReverse("VBScript") ' MyStr contains "tpircSBV".
Sub Statement

See Also

Description

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

Syntax

\[
\text{[Public [Default]| Private] Sub name [(arglist)]} \\
\quad \text{[statements]} \\
\quad \text{[Exit Sub]} \\
\quad \text{[statements]} \\
\text{End Sub}
\]

The Sub statement syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Public</td>
<td>Indicates that the Sub procedure is accessible to all other procedures in all scripts.</td>
</tr>
<tr>
<td>Default</td>
<td>Used only with the Public keyword in a Class block to indicate that the Sub procedure is the default method for the class. An error occurs if more than one Default procedure is specified in a class.</td>
</tr>
<tr>
<td>Private</td>
<td>Indicates that the Sub procedure is accessible only to other procedures in the script where it is declared.</td>
</tr>
<tr>
<td>name</td>
<td>Name of the Sub; follows standard variable naming conventions.</td>
</tr>
<tr>
<td>arglist</td>
<td>List of variables representing arguments that are passed to the Sub procedure when it is called. Multiple variables are separated by commas.</td>
</tr>
<tr>
<td>statements</td>
<td>Any group of statements to be executed within the body of the Sub procedure.</td>
</tr>
</tbody>
</table>

The arglist argument has the following syntax and parts:
[**ByVal** | **ByRef**] *varname*[()]

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ByVal</td>
<td>Indicates that the argument is passed by value.</td>
</tr>
<tr>
<td>ByRef</td>
<td>Indicates that the argument is passed by reference.</td>
</tr>
<tr>
<td><em>varname</em></td>
<td>Name of the variable representing the argument; follows standard variable naming conventions.</td>
</tr>
</tbody>
</table>

**Remarks**

If not explicitly specified using either **Public** or **Private**, **Sub** procedures are public by default, that is, they are visible to all other procedures in your script. The value of local variables in a **Sub** procedure is not preserved between calls to the **procedure**.

You can't define a **Sub** procedure inside any other procedure (e.g. **Function** or **Property Get**).

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

Like a **Function** procedure, a **Sub** 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** procedure, which returns a value, a **Sub** procedure can't be used in an **expression**.

You call a **Sub** procedure using the procedure name followed by the argument list. See the **Call** statement for specific information on how to call **Sub** procedures.

---

**Caution**  **Sub** procedures can be recursive, that is, they can call themselves to perform a given
task. However, recursion can lead to stack overflow.

Variables used in Sub 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 have defined at the script 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 is referring to that script-level name. To avoid this kind of conflict, use an Option Explicit statement to force explicit declaration of variables.
Tan Function

Description

Returns the tangent of an angle.

Syntax

Tan(number)

The number argument can be any valid numeric expression that expresses an angle in radians.

Remarks

Tan takes an angle and returns the ratio of two sides of a right triangle. The ratio is the length of the side opposite the angle divided by the length of the side adjacent to the angle.

To convert degrees to radians, multiply degrees by pi/180. To convert radians to degrees, multiply radians by 180/pi.

The following example uses the Tan function to return the tangent of an angle:

Dim MyAngle, MyCotangent
MyAngle = 1.3 ' Define angle in radians
MyCotangent = 1 / Tan(MyAngle) ' Calculate cot
Termination Event

Description

Occurs when an instance of the associated class is terminated.

Syntax

Private Sub Class_Terminate()
    statements
End Sub

The statements part consists of zero or more code statements to be run when the class is initialized.

Remarks

The following example illustrates the use of the Terminate event:

Class TestClass
    Private Sub Class_Initialize ' Setup Initialize event.
        MsgBox("TestClass started")
    End Sub
    Private Sub Class_Terminate ' Setup Terminate event.
        MsgBox("TestClass terminated")
    End Sub
End Class
Set X = New TestClass  ' Create an instance of TestClass.
Set X = Nothing        ' Destroy the instance.
Description

Executes a regular expression search against a specified string and returns a **Boolean** value that indicates if a pattern match was found.

Syntax

```
object.Test(string)
```

The **Execute** method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>Required. Always the name of a <strong>RegExp</strong> object.</td>
</tr>
<tr>
<td>string</td>
<td>Required. The text string upon which the regular expression is executed.</td>
</tr>
</tbody>
</table>

Remarks

The actual pattern for the regular expression search is set using the **Pattern** property of the **RegExp** object. The **RegExp.Global** property has no effect on the **Test** method.

The **Test** method returns **True** if a pattern match is found; **False** if no match is found.

The following code illustrates the use of the **Test** method:
Function RegExpTest(patrn, strng)
    Dim regEx, retVal ' Create variable.
    Set regEx = New RegExp ' Create regular expression.
    regEx.Pattern = patrn ' Set pattern.
    regEx.IgnoreCase = False ' Set case sensitivity.
    retVal = regEx.Test(strng) ' Execute the search test.
    If retVal Then
        RegExpTest = "One or more matches were found."
    Else
        RegExpTest = "No match was found."
    End If
End Function

MsgBox(RegExpTest("is.", "IS1 is2 IS3 is4"))
Time Function

Description

Returns a Variant of subtype Date indicating the current system time.

Syntax

Time

Remarks

The following example uses the Time function to return the current system time:

Dim MyTime
MyTime = Time   ' Return current system time.
Function

Description

Returns the number of seconds that have elapsed since 12:00 AM (midnight).

Syntax

Timer

Remarks

The following example uses the Timer function to determine the time it takes to iterate a For...Next loop N times:

Function TimeIt(N)
    Dim StartTime, EndTime
    StartTime = Timer
    For I = 1 To N
        Next
    EndTime = Timer
    TimeIt = EndTime - StartTime
End Function
TimeSerial Function

Description

Returns a Variant of subtype Date containing the time for a specific hour, minute, and second.

Syntax

TimeSerial(hour, minute, second)

The TimeSerial function syntax has these arguments:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hour</td>
<td>Number between 0 (12:00 A.M.) and 23 (11:00 P.M.), inclusive, or a numeric expression.</td>
</tr>
<tr>
<td>minute</td>
<td>Any numeric expression.</td>
</tr>
<tr>
<td>second</td>
<td>Any numeric expression.</td>
</tr>
</tbody>
</table>

Remarks

To specify a time, such as 11:59:59, the range of numbers for each TimeSerial argument should be in the accepted range for the unit; that is, 0–23 for hours and 0–59 for minutes and seconds. However, you can also specify relative times for each argument using any numeric expression that represents some number of hours, minutes, or seconds before or after a certain time.

The following example uses expressions instead of absolute time numbers. The TimeSerial function returns a time for 15 minutes before (-15) six hours before noon (12 - 6), or 5:45:00 A.M.

Dim MyTime1
MyTime1 = TimeSerial(12 - 6, -15, 0) ' Returns 5:45

When any argument exceeds the accepted range for that argument, it increments to the next larger unit as appropriate. For example, if you specify 75 minutes, it is evaluated as one hour and 15 minutes. However, if any single argument is outside the range -32,768 to 32,767, or if the time specified by the three arguments, either directly or by expression, causes the date to fall outside the acceptable range of dates, an error occurs.
**TimeValue Function**

**Description**

Returns a **Variant** of subtype **Date** containing the time.

**Syntax**

\[
\text{TimeValue}(\text{time})
\]

The `time` argument is usually a **string expression** representing a time from 00:00:00 (12:00:00 A.M.) to 23:59:59 (11:59:59 P.M.), inclusive. However, `time` can also be any expression that represents a time in that range. If `time` contains **Null**, **Null** is returned.

**Remarks**

You can enter valid times using a 12-hour or 24-hour clock. For example, "2:24PM" and "14:24" are both valid `time` arguments. If the `time` argument contains date information, **TimeValue** doesn't return the date information. However, if `time` includes invalid date information, an error occurs.

The following example uses the **TimeValue** function to convert a string to a time. You can also use **date literals** to directly assign a time to a **Variant** (for example, `MyTime = #4:35:17 PM#`).

```vbnet
Dim MyTime
MyTime = TimeValue("4:35:17 PM") ' MyTime contains 4:35:17 PM
```
See Also

Description

The **True** keyword has a value equal to -1.
**TypeName**

**Function**

**See Also**

**Description**

Returns a string that provides **Variant** subtype information about a variable.

**Syntax**

```
TypeName(varname)
```

The required `varname` argument can be any variable.

**Return Values**

The **TypeName** function has the following return values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte</td>
<td>Byte value</td>
</tr>
<tr>
<td>Integer</td>
<td>Integer value</td>
</tr>
<tr>
<td>Long</td>
<td>Long integer value</td>
</tr>
<tr>
<td>Single</td>
<td>Single-precision floating-point value</td>
</tr>
<tr>
<td>Double</td>
<td>Double-precision floating-point value</td>
</tr>
<tr>
<td>Currency</td>
<td>Currency value</td>
</tr>
<tr>
<td>Decimal</td>
<td>Decimal value</td>
</tr>
<tr>
<td>Date</td>
<td>Date or time value</td>
</tr>
<tr>
<td>String</td>
<td>Character string value</td>
</tr>
<tr>
<td>Boolean</td>
<td>Boolean value; <strong>True</strong> or <strong>False</strong></td>
</tr>
<tr>
<td>Empty</td>
<td>Uninitialized</td>
</tr>
<tr>
<td>Null</td>
<td>No valid data</td>
</tr>
<tr>
<td>object; type</td>
<td>Actual type name of an object</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------------------------</td>
</tr>
<tr>
<td>Object</td>
<td>Generic object</td>
</tr>
<tr>
<td>Unknown</td>
<td>Unknown object type</td>
</tr>
<tr>
<td>Nothing</td>
<td>Object variable that doesn't yet refer to an object instance</td>
</tr>
<tr>
<td>Error</td>
<td>Error</td>
</tr>
</tbody>
</table>

Remarks

The following example uses the **TypeName** function to return information about a variable:

```vbnet
Dim ArrayVar(4), MyType
NullVar = Null          ' Assign Null value.

MyType = TypeName("VBScript") ' Returns "String".
MyType = TypeName(4)       ' Returns "Integer".
MyType = TypeName(37.50)    ' Returns "Double".
MyType = TypeName(NullVar)  ' Returns "Null".
MyType = TypeName(ArrayVar) ' Returns "Variant()".
```
UBound Function

Description

Returns the largest available subscript for the indicated dimension of an array.

Syntax

UBound(arrayname[, dimension])

The UBound function syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>arrayname</td>
<td>Required. Name of the array variable; follows standard variable naming conventions.</td>
</tr>
<tr>
<td>dimension</td>
<td>Optional. Whole number indicating which dimension's upper bound is returned. Use 1 for the first dimension, 2 for the second, and so on. If dimension is omitted, 1 is assumed.</td>
</tr>
</tbody>
</table>

Remarks

The UBound function is used with the LBound function to determine the size of an array. Use the LBound function to find the lower limit of an array dimension.

The lower bound for any dimension is always 0. As a result, UBound returns the following values for an array with these dimensions:

Dim A(100,3,4)
<table>
<thead>
<tr>
<th>Statement</th>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>UBound(A, 1)</td>
<td>100</td>
</tr>
<tr>
<td>UBound(A, 2)</td>
<td>3</td>
</tr>
<tr>
<td>UBound(A, 3)</td>
<td>4</td>
</tr>
</tbody>
</table>
UCase Function

Description

Returns a string that has been converted to uppercase.

Syntax

UCase(string)

The string argument is any valid string expression. If string contains Null, Null is returned.

Remarks

Only lowercase letters are converted to uppercase; all uppercase letters and nonletter characters remain unchanged.

The following example uses the UCase function to return an uppercase version of a string:

Dim MyWord
MyWord = UCase("Hello World") ' Returns "HELLO WORLD"
Value Property

Description

Returns the value or text of a match found in a search string.

Syntax

object.Value

The object argument is always a Match object.

Remarks

The following code illustrates the use of the Value property:

```vbnet
Function RegExpTest(patrn, strng)
    Dim regEx, Match, Matches   ' Create variable.
    Set regEx = New RegExp   ' Create regular expression.
    regEx.Pattern = patrn   ' Set pattern.
    regEx.IgnoreCase = True   ' Set case insensitivity.
    regEx.Global = True   ' Set global applicability.
    Set Matches = regEx.Execute(strng)   ' Execute search.
    For Each Match in Matches   ' Iterate Matches collection.
        RetStr = RetStr & "Match " & I & " found at position "
        RetStr = RetStr & Match.FirstIndex & ". Match Value is "
        RetStr = RetStr & Match.Value & "." & vbCRLF
    Next Match
    WScript.Echo RetStr
End Function
```
Next
RegExpTest = RetStr
End Function

MsgBox(RegExpTest("is.", "IS1 is2 IS3 is4"))
These constants are only available when your project has an explicit reference to the appropriate type library containing these constant definitions. For VBScript, you must explicitly declare these constants in your code.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vbEmpty</td>
<td>0</td>
<td>Uninitialized (default)</td>
</tr>
<tr>
<td>vbNull</td>
<td>1</td>
<td>Contains no valid data</td>
</tr>
<tr>
<td>vbInteger</td>
<td>2</td>
<td>Integer subtype</td>
</tr>
<tr>
<td>vbLong</td>
<td>3</td>
<td>Long subtype</td>
</tr>
<tr>
<td>vbSingle</td>
<td>4</td>
<td>Single subtype</td>
</tr>
<tr>
<td>vbSingle</td>
<td>5</td>
<td>Double subtype</td>
</tr>
<tr>
<td>vbCurrency</td>
<td>6</td>
<td>Currency subtype</td>
</tr>
<tr>
<td>vbDate</td>
<td>7</td>
<td>Date subtype</td>
</tr>
<tr>
<td>vbString</td>
<td>8</td>
<td>String subtype</td>
</tr>
<tr>
<td>vbObject</td>
<td>9</td>
<td>Object</td>
</tr>
<tr>
<td>vbError</td>
<td>10</td>
<td>Error subtype</td>
</tr>
<tr>
<td>vbBoolean</td>
<td>11</td>
<td>Boolean subtype</td>
</tr>
<tr>
<td>vbVariant</td>
<td>12</td>
<td>Variant (used only for arrays of variants)</td>
</tr>
<tr>
<td>vbDataObject</td>
<td>13</td>
<td>Data access object</td>
</tr>
<tr>
<td>vbDecimal</td>
<td>14</td>
<td>Decimal subtype</td>
</tr>
<tr>
<td>vbByte</td>
<td>17</td>
<td>Byte subtype</td>
</tr>
<tr>
<td>vbArray</td>
<td>8192</td>
<td>Array</td>
</tr>
</tbody>
</table>
VarType Function

See Also

Description

Returns a value indicating the subtype of a variable.

Syntax

VarType(varname)

The varname argument can be any variable.

Return Values

The VarType function returns the following values:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vbEmpty</td>
<td>0</td>
<td>Empty (uninitialized)</td>
</tr>
<tr>
<td>vbNull</td>
<td>1</td>
<td>Null (no valid data)</td>
</tr>
<tr>
<td>vbInteger</td>
<td>2</td>
<td>Integer</td>
</tr>
<tr>
<td>vbLong</td>
<td>3</td>
<td>Long integer</td>
</tr>
<tr>
<td>vbSingle</td>
<td>4</td>
<td>Single-precision floating-point number</td>
</tr>
<tr>
<td>vbDouble</td>
<td>5</td>
<td>Double-precision floating-point number</td>
</tr>
<tr>
<td>vbCurrency</td>
<td>6</td>
<td>Currency</td>
</tr>
<tr>
<td>vbDate</td>
<td>7</td>
<td>Date</td>
</tr>
<tr>
<td>vbString</td>
<td>8</td>
<td>String</td>
</tr>
<tr>
<td>vbObject</td>
<td>9</td>
<td>Automation object</td>
</tr>
<tr>
<td>vbError</td>
<td>10</td>
<td>Error</td>
</tr>
<tr>
<td>vbBoolean</td>
<td>11</td>
<td>Boolean</td>
</tr>
<tr>
<td>vbVariant</td>
<td>12</td>
<td>Variant (used only with arrays of Variants)</td>
</tr>
<tr>
<td>vbDataObject</td>
<td>13</td>
<td>A data-access object</td>
</tr>
<tr>
<td>-------------</td>
<td>----</td>
<td>----------------------</td>
</tr>
<tr>
<td>vbByte</td>
<td>17</td>
<td>Byte</td>
</tr>
<tr>
<td>vbArray</td>
<td>8192</td>
<td>Array</td>
</tr>
</tbody>
</table>

**Note**  These *constants* are specified by VBScript. As a result, the names can be used anywhere in your code in place of the actual values.

**Remarks**

The **VarType** function never returns the value for Array by itself. It is always added to some other value to indicate an array of a particular type. The value for Variant is only returned when it has been added to the value for Array to indicate that the argument to the **VarType** function is an array. For example, the value returned for an array of integers is calculated as 2 + 8192, or 8194. If an object has a default *property*, **VarType**(object) returns the type of its default property.

The following example uses the **VarType** function to determine the subtype of a variable.

```
Dim MyCheck
MyCheck = VarType(300)  ' Returns 2.
MyCheck = VarType(#10/19/62#)  ' Returns 7.
MyCheck = VarType("VBScript")  ' Returns 8.
```
A number of useful constants you can use in your code are built into VBScript. Constants provide a convenient way to use specific values without actually having to remember the value itself. Using constants also makes your code more maintainable should the value of any constant ever change. Because these constants are already defined in VBScript, you don't need to explicitly declare them in your code. Simply use them in place of the values they represent.

Here are the various categories of constants provided in VBScript and a brief description of each:

- **Color Constants**
  Defines eight basic colors that can be used in scripting.

- **Date and Time Constants**
  Defines date and time constants used by various date and time functions.

- **Date Format Constants**
  Defines constants used to format dates and times.

- **Miscellaneous Constants**
  Defines constants that don't conveniently fit into any other category.

- **MsgBox Constants**
  Defines constants used in the **MsgBox** function to describe button visibility, labeling, behavior, and return values.

- **String Constants**
  Defines a variety of non-printable characters used in string manipulation.

- **Tristate Constants**
  Defines constants used with functions that format numbers.

- **VarType Constants**
  Defines the various Variant subtypes.
Weekday Function

See Also

Description

Returns a whole number representing the day of the week.

Syntax

**Weekday**(*date*, [*firstdayofweek]*)

The `Weekday` function syntax has these arguments:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>date</code></td>
<td>Any expression that can represent a <code>date</code>. If <code>date</code> contains <code>Null</code>, <code>Null</code> is returned.</td>
</tr>
<tr>
<td><code>firstdayofweek</code></td>
<td>A constant that specifies the first day of the week. If omitted, <code>vbSunday</code> is assumed.</td>
</tr>
</tbody>
</table>

Settings

The `firstdayofweek` argument has these settings:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>vbUseSystem</code></td>
<td>0</td>
<td>Use National Language Support (NLS) API setting.</td>
</tr>
<tr>
<td><code>vbSunday</code></td>
<td>1</td>
<td>Sunday</td>
</tr>
<tr>
<td><code>vbMonday</code></td>
<td>2</td>
<td>Monday</td>
</tr>
<tr>
<td><code>vbTuesday</code></td>
<td>3</td>
<td>Tuesday</td>
</tr>
<tr>
<td><code>vbWednesday</code></td>
<td>4</td>
<td>Wednesday</td>
</tr>
<tr>
<td><code>vbThursday</code></td>
<td>5</td>
<td>Thursday</td>
</tr>
<tr>
<td><code>vbFriday</code></td>
<td>6</td>
<td>Friday</td>
</tr>
<tr>
<td><code>vbSaturday</code></td>
<td>7</td>
<td>Saturday</td>
</tr>
</tbody>
</table>
Return Values

The **Weekday** function can return any of these values:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vbSunday</td>
<td>1</td>
<td>Sunday</td>
</tr>
<tr>
<td>vbMonday</td>
<td>2</td>
<td>Monday</td>
</tr>
<tr>
<td>vbTuesday</td>
<td>3</td>
<td>Tuesday</td>
</tr>
<tr>
<td>vbWednesday</td>
<td>4</td>
<td>Wednesday</td>
</tr>
<tr>
<td>vbThursday</td>
<td>5</td>
<td>Thursday</td>
</tr>
<tr>
<td>vbFriday</td>
<td>6</td>
<td>Friday</td>
</tr>
<tr>
<td>vbSaturday</td>
<td>7</td>
<td>Saturday</td>
</tr>
</tbody>
</table>

Remarks

The following example uses the **Weekday** function to obtain the day of the week from a specified date:

```vbscript
Dim MyDate, MyWeekDay
MyDate = #October 19, 1962#  ' Assign a date.
MyWeekDay = Weekday(MyDate)  ' MyWeekDay contains 6 bec
' MyDate represents a Friday.
```
While...Wend
Statement

See Also

Description

Executes a series of statements as long as a given condition is True.

Syntax

While condition
    Version [statements]
Wend

The While...Wend statement syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>condition</td>
<td>Numeric or string expression that evaluates to True or False. If condition is Null, condition is treated as False.</td>
</tr>
<tr>
<td>statements</td>
<td>One or more statements executed while condition is True.</td>
</tr>
</tbody>
</table>

Remarks

If condition is True, all statements in statements are executed until the Wend statement is encountered. Control then returns to the While statement and condition is again checked. If condition is still True, the process is repeated. If it is not True, execution resumes with the statement following the Wend statement.

While...Wend loops can be nested to any level. Each Wend matches the most recent While.
**Tip** The **Do...Loop** statement provides a more structured and flexible way to perform looping.

The following example illustrates use of the **While...Wend** statement:

```
Dim Counter
Counter = 0       ' Initialize variable.
While Counter < 20 ' Test value of Counter.
    Counter = Counter + 1  ' Increment Counter.
    Alert Counter
Wend              ' End While loop when Counter
```
With Statement

See Also

Description

Executes a series of statements on a single object.

Syntax

\[
\text{With } \text{object} \\
\text{ statements} \\
\text{End With}
\]

The With statement syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>Required. Name of an object or a function that returns an object.</td>
</tr>
<tr>
<td>statements</td>
<td>Required. One or more statements to be executed on object.</td>
</tr>
</tbody>
</table>

Remarks

The With statement allows you to perform a series of statements on a specified object without requalifying the name of the object. For example, to change a number of different properties on a single object, place the property assignment statements within the With control structure, referring to the object once instead of referring to it with each property assignment. The following example illustrates use of the With statement to assign values to several properties of the same object.
With MyLabel
  .Height = 2000
  .Width = 2000
  .Caption = "This is MyLabel"
End With

While property manipulation is an important aspect of `With` functionality, it is not the only use. Any legal code can be used within a `With` block.

**Note** Once a `With` block is entered, `object` can't be changed. As a result, you can't use a single `With` statement to affect a number of different objects.

You can nest `With` statements by placing one `With` block within another. However, because members of outer `With` blocks are masked within the inner `With` blocks, you must provide a fully qualified object reference in an inner `With` block to any member of an object in an outer `With` block.

**Important** Do not jump into or out of `With` blocks. If statements in a `With` block are executed, but either the `With` or `End With` statement is not executed, you may get errors or unpredictable behavior.
Xor Operator

See Also

Description

Performs a logical exclusion on two expressions.

Syntax

\[ \text{result} = \text{expression1} \text{ Xor expression2} \]

The Xor operator syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>Any numeric variable.</td>
</tr>
<tr>
<td>expression1</td>
<td>Any expression.</td>
</tr>
<tr>
<td>expression2</td>
<td>Any expression.</td>
</tr>
</tbody>
</table>

Remarks

If one, and only one, of the expressions evaluates to True, result is True. However, if either expression is Null, result is also Null. When neither expression is Null, result is determined according to the following table:

<table>
<thead>
<tr>
<th>If expression1 is</th>
<th>And expression2 is</th>
<th>Then result is</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
</tbody>
</table>

The Xor operator also performs a bitwise comparison of identically
positioned bits in two numeric expressions and sets the corresponding bit in result according to the following table:

<table>
<thead>
<tr>
<th>If bit in expression1 is</th>
<th>And bit in expression2 is</th>
<th>Then result is</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
**Year**

**Function**

**Description**

Returns a whole number representing the year.

**Syntax**

Year(date)

The *date* argument is any expression that can represent a date. If *date* contains **Null**, **Null** is returned.

**Remarks**

The following example uses the **Year** function to obtain the year from a specified date:

```vbnet
Dim MyDate, MyYear
MyDate = #October 19, 1962#  ' Assign a date.
MyYear = Year(MyDate)        ' MyYear contains 1962.
```
Since these constants are built into VBScript, you don't have to define them before using them. Use them anywhere in your code to represent the values shown for each.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vbBlack</td>
<td>&amp;h0000</td>
<td>Black</td>
</tr>
<tr>
<td>vbRed</td>
<td>&amp;hFF00</td>
<td>Red</td>
</tr>
<tr>
<td>vbGreen</td>
<td>&amp;h00FF</td>
<td>Green</td>
</tr>
<tr>
<td>vbYellow</td>
<td>&amp;hFFFF</td>
<td>Yellow</td>
</tr>
<tr>
<td>vbBlue</td>
<td>&amp;hFF00</td>
<td>Blue</td>
</tr>
<tr>
<td>vbMagenta</td>
<td>&amp;hFF00FF</td>
<td>Magenta</td>
</tr>
<tr>
<td>vbCyan</td>
<td>&amp;hFFFF00</td>
<td>Cyan</td>
</tr>
<tr>
<td>vbWhite</td>
<td>&amp;hFFFFFF</td>
<td>White</td>
</tr>
</tbody>
</table>
Tristate Constants

See Also

Since these constants are built into VBScript, you don't have to define them before using them. Use them anywhere in your code to represent the values shown for each.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vbUseDefault</td>
<td>-2</td>
<td>Use default from computer's regional settings.</td>
</tr>
<tr>
<td>vbTrue</td>
<td>-1</td>
<td>True</td>
</tr>
<tr>
<td>vbFalse</td>
<td>0</td>
<td>False</td>
</tr>
</tbody>
</table>
## VBScript Run-time Errors

### VBScript Syntax Errors

<table>
<thead>
<tr>
<th>Error Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>Invalid procedure call or argument</td>
</tr>
<tr>
<td>6</td>
<td>Overflow</td>
</tr>
<tr>
<td>7</td>
<td>Out of memory</td>
</tr>
<tr>
<td>9</td>
<td>Subscript out of range</td>
</tr>
<tr>
<td>10</td>
<td>This array is fixed or temporarily locked</td>
</tr>
<tr>
<td>11</td>
<td>Division by zero</td>
</tr>
<tr>
<td>13</td>
<td>Type mismatch</td>
</tr>
<tr>
<td>14</td>
<td>Out of string space</td>
</tr>
<tr>
<td>17</td>
<td>Can't perform requested operation</td>
</tr>
<tr>
<td>28</td>
<td>Out of stack space</td>
</tr>
<tr>
<td>35</td>
<td>Sub or Function not defined</td>
</tr>
<tr>
<td>48</td>
<td>Error in loading DLL</td>
</tr>
<tr>
<td>51</td>
<td>Internal error</td>
</tr>
<tr>
<td>52</td>
<td>Bad file name or number</td>
</tr>
<tr>
<td>53</td>
<td>File not found</td>
</tr>
<tr>
<td>54</td>
<td>Bad file mode</td>
</tr>
<tr>
<td>55</td>
<td>File already open</td>
</tr>
<tr>
<td>57</td>
<td>Device I/O error</td>
</tr>
<tr>
<td>58</td>
<td>File already exists</td>
</tr>
<tr>
<td>61</td>
<td>Disk full</td>
</tr>
<tr>
<td>62</td>
<td>Input past end of file</td>
</tr>
<tr>
<td>67</td>
<td>Too many files</td>
</tr>
<tr>
<td>68</td>
<td>Device unavailable</td>
</tr>
<tr>
<td>70</td>
<td>Permission denied</td>
</tr>
<tr>
<td>71</td>
<td>Disk not ready</td>
</tr>
<tr>
<td>74</td>
<td>Can't rename with different drive</td>
</tr>
<tr>
<td>75</td>
<td>Path/File access error</td>
</tr>
<tr>
<td>76</td>
<td>Path not found</td>
</tr>
<tr>
<td>91</td>
<td>Object variable not set</td>
</tr>
<tr>
<td>Code</td>
<td>Message</td>
</tr>
<tr>
<td>-------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td>92</td>
<td>For loop not initialized</td>
</tr>
<tr>
<td>94</td>
<td>Invalid use of Null</td>
</tr>
<tr>
<td>322</td>
<td>Can't create necessary temporary file</td>
</tr>
<tr>
<td>424</td>
<td>Object required</td>
</tr>
<tr>
<td>429</td>
<td>ActiveX component can't create object</td>
</tr>
<tr>
<td>430</td>
<td>Class doesn't support Automation</td>
</tr>
<tr>
<td>432</td>
<td>File name or class name not found during Automation operation</td>
</tr>
<tr>
<td>438</td>
<td>Object doesn't support this property or method</td>
</tr>
<tr>
<td>440</td>
<td>Automation error</td>
</tr>
<tr>
<td>445</td>
<td>Object doesn't support this action</td>
</tr>
<tr>
<td>446</td>
<td>Object doesn't support named arguments</td>
</tr>
<tr>
<td>447</td>
<td>Object doesn't support current locale setting</td>
</tr>
<tr>
<td>448</td>
<td>Named argument not found</td>
</tr>
<tr>
<td>449</td>
<td>Argument not optional</td>
</tr>
<tr>
<td>450</td>
<td>Wrong number of arguments or invalid property assignment</td>
</tr>
<tr>
<td>451</td>
<td>Object not a collection</td>
</tr>
<tr>
<td>453</td>
<td>Specified DLL function not found</td>
</tr>
<tr>
<td>455</td>
<td>Code resource lock error</td>
</tr>
<tr>
<td>458</td>
<td>Variable uses an Automation type not supported in VBScript</td>
</tr>
<tr>
<td>462</td>
<td>The remote server machine does not exist or is unavailable</td>
</tr>
<tr>
<td>481</td>
<td>Invalid picture</td>
</tr>
<tr>
<td>500</td>
<td>Variable is undefined</td>
</tr>
<tr>
<td>501</td>
<td>Illegal assignment</td>
</tr>
<tr>
<td>502</td>
<td>Object not safe for scripting</td>
</tr>
<tr>
<td>503</td>
<td>Object not safe for initializing</td>
</tr>
<tr>
<td>504</td>
<td>Object not safe for creating</td>
</tr>
<tr>
<td>505</td>
<td>Invalid or unqualified reference</td>
</tr>
<tr>
<td>506</td>
<td>Class not defined</td>
</tr>
<tr>
<td>507</td>
<td>An exception occurred</td>
</tr>
<tr>
<td>5016</td>
<td>Regular Expression object expected</td>
</tr>
<tr>
<td>5017</td>
<td>Syntax error in regular expression</td>
</tr>
<tr>
<td>5018</td>
<td>Unexpected quantifier</td>
</tr>
<tr>
<td>5019</td>
<td>Expected ']' in regular expression</td>
</tr>
<tr>
<td>5020</td>
<td>Expected ')' in regular expression</td>
</tr>
<tr>
<td>5021</td>
<td>Invalid range in character set</td>
</tr>
<tr>
<td>32811</td>
<td>Element not found</td>
</tr>
<tr>
<td>Error Number</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>---------------------------------</td>
</tr>
<tr>
<td>1001</td>
<td>Out of memory</td>
</tr>
<tr>
<td>1002</td>
<td>Syntax error</td>
</tr>
<tr>
<td>1003</td>
<td>Expected ':'</td>
</tr>
<tr>
<td>1005</td>
<td>Expected '('</td>
</tr>
<tr>
<td>1006</td>
<td>Expected ')'</td>
</tr>
<tr>
<td>1007</td>
<td>Expected ']'</td>
</tr>
<tr>
<td>1010</td>
<td>Expected identifier</td>
</tr>
<tr>
<td>1011</td>
<td>Expected '='</td>
</tr>
<tr>
<td>1012</td>
<td>Expected 'If'</td>
</tr>
<tr>
<td>1013</td>
<td>Expected 'To'</td>
</tr>
<tr>
<td>1014</td>
<td>Expected 'End'</td>
</tr>
<tr>
<td>1015</td>
<td>Expected 'Function'</td>
</tr>
<tr>
<td>1016</td>
<td>Expected 'Sub'</td>
</tr>
<tr>
<td>1017</td>
<td>Expected 'Then'</td>
</tr>
<tr>
<td>1018</td>
<td>Expected 'Wend'</td>
</tr>
<tr>
<td>1019</td>
<td>Expected 'Loop'</td>
</tr>
<tr>
<td>1020</td>
<td>Expected 'Next'</td>
</tr>
<tr>
<td>1021</td>
<td>Expected 'Case'</td>
</tr>
<tr>
<td>1022</td>
<td>Expected 'Select'</td>
</tr>
<tr>
<td>1023</td>
<td>Expected expression</td>
</tr>
<tr>
<td>1024</td>
<td>Expected statement</td>
</tr>
<tr>
<td>1025</td>
<td>Expected end of statement</td>
</tr>
<tr>
<td>1026</td>
<td>Expected integer constant</td>
</tr>
<tr>
<td>1027</td>
<td>Expected 'While' or 'Until'</td>
</tr>
<tr>
<td>1028</td>
<td>Expected 'While', 'Until' or end of statement</td>
</tr>
<tr>
<td>1029</td>
<td>Expected 'With'</td>
</tr>
<tr>
<td>1030</td>
<td>Identifier too long</td>
</tr>
<tr>
<td>1031</td>
<td>Invalid number</td>
</tr>
<tr>
<td>1032</td>
<td>Invalid character</td>
</tr>
<tr>
<td>Code</td>
<td>Error Description</td>
</tr>
<tr>
<td>-------</td>
<td>-----------------------------------------------------------------------</td>
</tr>
<tr>
<td>1033</td>
<td>Unterminated string constant</td>
</tr>
<tr>
<td>1034</td>
<td>Unterminated comment</td>
</tr>
<tr>
<td>1037</td>
<td>Invalid use of 'Me' keyword</td>
</tr>
<tr>
<td>1038</td>
<td>'loop' without 'do'</td>
</tr>
<tr>
<td>1039</td>
<td>Invalid 'exit' statement</td>
</tr>
<tr>
<td>1040</td>
<td>Invalid 'for' loop control variable</td>
</tr>
<tr>
<td>1041</td>
<td>Name redefined</td>
</tr>
<tr>
<td>1042</td>
<td>Must be first statement on the line</td>
</tr>
<tr>
<td>1043</td>
<td>Cannot assign to non-ByVal argument</td>
</tr>
<tr>
<td>1044</td>
<td>Cannot use parentheses when calling a Sub</td>
</tr>
<tr>
<td>1045</td>
<td>Expected literal constant</td>
</tr>
<tr>
<td>1046</td>
<td>Expected 'In'</td>
</tr>
<tr>
<td>1047</td>
<td>Expected 'Class'</td>
</tr>
<tr>
<td>1048</td>
<td>Must be defined inside a Class</td>
</tr>
<tr>
<td>1049</td>
<td>Expected Let or Set or Get in property declaration</td>
</tr>
<tr>
<td>1050</td>
<td>Expected 'Property'</td>
</tr>
<tr>
<td>1051</td>
<td>Number of arguments must be consistent across properties specification</td>
</tr>
<tr>
<td>1052</td>
<td>Cannot have multiple default property/method in a Class</td>
</tr>
<tr>
<td>1053</td>
<td>Class initialize or terminate do not have arguments</td>
</tr>
<tr>
<td>1054</td>
<td>Property set or let must have at least one argument</td>
</tr>
<tr>
<td>1055</td>
<td>Unexpected 'Next'</td>
</tr>
<tr>
<td>1056</td>
<td>'Default' can be specified only on 'Property' or 'Function' or 'Sub'</td>
</tr>
<tr>
<td>1057</td>
<td>'Default' specification must also specify 'Public'</td>
</tr>
<tr>
<td>1058</td>
<td>'Default' specification can only be on Property Get</td>
</tr>
</tbody>
</table>
Comparison Operators

Description

Used to compare expressions.

Syntax

\[ \text{result} = \text{expression1} \ \text{comparisonoperator} \ \text{expression2} \]

\[ \text{result} = \text{object1} \ \text{Is} \ \text{object2} \]

Comparison operators have these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>result</td>
<td>Any numeric variable.</td>
</tr>
<tr>
<td>expression</td>
<td>Any expression.</td>
</tr>
<tr>
<td>comparisonoperator</td>
<td>Any comparison operator.</td>
</tr>
<tr>
<td>object</td>
<td>Any object name.</td>
</tr>
</tbody>
</table>

Remarks

The Is operator has specific comparison functionality that differs from the operators in the following table. The following table contains a list of the comparison operators and the conditions that determine whether result is True, False, or Null:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>True if</th>
<th>False if</th>
<th>Null if</th>
</tr>
</thead>
</table>

Microsoft® Visual Basic® Scripting Edition
### Comparison Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example 1</th>
<th>Example 2</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;</code></td>
<td>Less than</td>
<td><code>expression1</code> &lt; <code>expression2</code></td>
<td><code>expression1</code> &gt;= <code>expression2</code></td>
<td><code>expression1 or expression2 = Null</code></td>
</tr>
<tr>
<td><code>&lt;=</code></td>
<td>Less than or equal to</td>
<td><code>expression1</code> &lt;= <code>expression2</code></td>
<td><code>expression1</code> &gt; <code>expression2</code></td>
<td><code>expression1 or expression2 = Null</code></td>
</tr>
<tr>
<td><code>&gt;</code></td>
<td>Greater than</td>
<td><code>expression1</code> &gt; <code>expression2</code></td>
<td><code>expression1</code> &lt;= <code>expression2</code></td>
<td><code>expression1 or expression2 = Null</code></td>
</tr>
<tr>
<td><code>&gt;=</code></td>
<td>Greater than or equal to</td>
<td><code>expression1</code> &gt;= <code>expression2</code></td>
<td><code>expression1</code> &lt; <code>expression2</code></td>
<td><code>expression1 or expression2 = Null</code></td>
</tr>
<tr>
<td><code>=</code></td>
<td>Equal to</td>
<td><code>expression1</code> = <code>expression2</code></td>
<td><code>expression1</code> &lt;&gt; <code>expression2</code></td>
<td><code>expression1 or expression2 = Null</code></td>
</tr>
<tr>
<td><code>&lt;&gt;</code></td>
<td>Not equal to</td>
<td><code>expression1</code> &lt;&gt; <code>expression2</code></td>
<td><code>expression1</code> = <code>expression2</code></td>
<td><code>expression1 or expression2 = Null</code></td>
</tr>
</tbody>
</table>

When comparing two expressions, you may not be able to easily determine whether the expressions are being compared as numbers or as strings.

The following table shows how expressions are compared or what results from the comparison, depending on the underlying subtype:

<table>
<thead>
<tr>
<th>If</th>
<th>Then</th>
</tr>
</thead>
<tbody>
<tr>
<td>Both expressions are numeric</td>
<td>Perform a numeric comparison.</td>
</tr>
<tr>
<td>Both expressions are strings</td>
<td>Perform a string comparison.</td>
</tr>
<tr>
<td>One expression is numeric and the other is a string</td>
<td>The numeric expression is less than the string expression.</td>
</tr>
<tr>
<td>One expression is Empty and the other is numeric</td>
<td>Perform a numeric comparison, using 0 as the Empty expression.</td>
</tr>
<tr>
<td>One expression is <strong>Empty</strong> and the other is a string</td>
<td>Perform a string comparison, using a zero-length string (&quot;&quot;) as the <strong>Empty</strong> expression.</td>
</tr>
<tr>
<td>-----------------------------------------------------</td>
<td>----------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Both expressions are <strong>Empty</strong></td>
<td>The expressions are equal.</td>
</tr>
</tbody>
</table>
Welcome to the VBScript Language Reference

These handy blocks of information will help you explore the many different parts of the Visual Basic Scripting language.

You'll find all the parts of the VBScript language listed alphabetically under the Alphabetic Keyword List. But if you want to examine just one category, say, objects, each language category has its own, more compact section.

How's it work? Click on one of the headings to the left to display a list of items contained in that category. From this list, select the topic that you want to view. Once you've opened that topic, you can easily link to other related sections.

So, go ahead and take a look! Study some statements, mull over the methods, or figure out
a few functions. You'll see just how versatile the VBScript language can be!
Add Method (Dictionary)

Description

Adds a key and item pair to a Dictionary object.

Syntax

object.Add key, item

The Add method has the following parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>Required. Always the name of a Dictionary object.</td>
</tr>
<tr>
<td>key</td>
<td>Required. The key associated with the item being added.</td>
</tr>
<tr>
<td>item</td>
<td>Required. The item associated with the key being added.</td>
</tr>
</tbody>
</table>

Remarks

An error occurs if the key already exists.

The following example illustrates the use of the Add method:

```vbnet
Dim d ' Create a variable.
Set d = CreateObject("Scripting.Dictionary")
d.Add "a", "Athens" ' Add some keys and items.
d.Add "b", "Belgrade"
d.Add "c", "Cairo"
```
Add Method (Folders)

Description

Adds a new `Folder` to a `Folders` collection.

Syntax

```vbs
object.Add(folderName)
```

The `Add` method has the following parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>object</code></td>
<td>Required. Always the name of a <code>Folders</code> collection.</td>
</tr>
<tr>
<td><code>folderName</code></td>
<td>Required. The name of the new <code>Folder</code> being added.</td>
</tr>
</tbody>
</table>

Remarks

The following example illustrates the use of the `Add` method to add a new folder:

```vbs
Sub AddNewFolder(path, folderName)
    Dim fso, f, fc, nf
    Set fso = CreateObject("Scripting.FileSystemObject")
    Set f = fso.GetFolder(path)
    Set fc = f.SubFolders
    If folderName <> "" Then
        Set nf = fc.Add(folderName)
    Else
        Set nf = fc.Add("New Folder")
End Sub
```
End If
End Sub

An error occurs if folderName already exists.
AtEndOfLine

Property

Description

Returns **True** if the file pointer immediately precedes the end-of-line marker in a **TextStream** file; **False** if it is not. Read-only.

Syntax

```
object.AtEndOfLine
```

The `object` is always the name of a **TextStream** object.

Remarks

The **AtEndOfLine** property applies only to **TextStream** files that are open for reading; otherwise, an error occurs.

The following code illustrates the use of the **AtEndOfLine** property:

```
Function ReadEntireFile(filespec)
    Const ForReading = 1
    Dim fso, theFile, retstring
    Set fso = CreateObject("Scripting.FileSystemObject")
    Set theFile = fso.OpenTextFile(filespec, ForReading, False)
    Do While theFile.AtEndOfLine <> True
        retstring = theFile.Read(1)
    Loop
```

See Also

Applies to

Version 2
theFile.Close
ReadEntireFile = retstring
End Function
AtEndOfStream Property

Description

Returns **True** if the file pointer is at the end of a **TextStream** file; **False** if it is not. Read-only.

Syntax

```
object.AtEndOfStream
```

The `object` is always the name of a **TextStream** object.

Remarks

The **AtEndOfStream** property applies only to **TextStream** files that are open for reading, otherwise, an error occurs.

The following code illustrates the use of the **AtEndOfStream** property:

```vbs
Function ReadEntireFile(filespec)
    Const ForReading = 1
    Dim fso, theFile, retstring
    Set fso = CreateObject("Scripting.FileSystemObject")
    Set theFile = fso.OpenTextFile(filespec, ForReading, False)
    Do While theFile.AtEndOfStream <> True
        retstring = theFile.ReadLine
    Loop
End Function
```
theFile.Close
ReadEntireFile = retstring
End Function
Attributes

Property

Description

Sets or returns the attributes of files or folders. Read/write or read-only, depending on the attribute.

Syntax

`object.Attributes [= newattributes]`

The `Attributes` property has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>object</code></td>
<td>Required. Always the name of a File or Folder object.</td>
</tr>
<tr>
<td><code>newattributes</code></td>
<td>Optional. If provided, <code>newattributes</code> is the new value for the attributes of the specified <code>object</code>.</td>
</tr>
</tbody>
</table>

Settings

The `newattributes` argument can have any of the following values or any logical combination of the following values:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>0</td>
<td>Normal file. No attributes are set.</td>
</tr>
<tr>
<td>ReadOnly</td>
<td>1</td>
<td>Read-only file. Attribute is read/write.</td>
</tr>
<tr>
<td>Hidden</td>
<td>2</td>
<td>Hidden file. Attribute is read/write.</td>
</tr>
<tr>
<td>System</td>
<td>4</td>
<td>System file. Attribute is read/write.</td>
</tr>
<tr>
<td>Directory</td>
<td>16</td>
<td>Folder or directory. Attribute is read-only.</td>
</tr>
<tr>
<td>Attribute</td>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>-------</td>
<td>-----------------------------------------------------------</td>
</tr>
<tr>
<td>Archive</td>
<td>32</td>
<td>File has changed since last backup. Attribute is read/write.</td>
</tr>
<tr>
<td>Alias</td>
<td>1024</td>
<td>Link or shortcut. Attribute is read-only.</td>
</tr>
<tr>
<td>Compressed</td>
<td>2048</td>
<td>Compressed file. Attribute is read-only.</td>
</tr>
</tbody>
</table>

Remarks

Attempts to change any of the read-only attributes (Alias, Compressed, or Directory) are ignored.

When setting attributes, it is generally a good idea to first read the current attributes, then change the individual attributes as desired, and finally write the attributes back.

The following code illustrates the use of the `Attributes` property with a file:

```vbs
Function ToggleArchiveBit(filespec)
    Dim fso, f
    Set fso = CreateObject("Scripting.FileSystemObject")
    Set f = fso.GetFile(filespec)
    If f.attributes and 32 Then
        f.attributes = f.attributes - 32
        ToggleArchiveBit = "Archive bit is cleared."
    Else
        f.attributes = f.attributes + 32
        ToggleArchiveBit = "Archive bit is set."
    End If
End Function
```
AvailableSpace Property

Description

Returns the amount of space available to a user on the specified drive or network share.

Syntax

object.AvailableSpace

The object is always a Drive object.

Remarks

The value returned by the AvailableSpace property is typically the same as that returned by the FreeSpace property. Differences may occur between the two for computer systems that support quotas.

The following code illustrates the use of the AvailableSpace property:

Function ShowAvailableSpace(drvPath)
    Dim fso, d, s
    Set fso = CreateObject("Scripting.FileSystemObject")
    Set d = fso.GetDrive(fso.GetDriveName(drvPath))
    s = "Drive " & UCase(drvPath) & " - "
    s = s & d.VolumeName & "<BR>"
    s = s & "Available Space: " & FormatNumber(d.AvailableSpace/1024,0) & " Kbytes"
    ShowAvailableSpace = s
End Function
BuildPath Method

Description

Appends a name to an existing path.

Syntax

\( \text{object}.\text{BuildPath}(\text{path}, \text{name}) \)

The BuildPath method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>Required. Always the name of a FileSystemObject.</td>
</tr>
<tr>
<td>path</td>
<td>Required. Existing path to which name is appended. Path can be absolute or relative and need not specify an existing folder.</td>
</tr>
<tr>
<td>name</td>
<td>Required. Name being appended to the existing path.</td>
</tr>
</tbody>
</table>

Remarks

The BuildPath method inserts an additional path separator between the existing path and the name, only if necessary.

The following example illustrates use of the BuildPath method:

```vbs
Function GetBuildPath(path)
    Dim fso, newpath
    Set fso = CreateObject("Scripting.FileSystemObject")
    newpath = fso.BuildPath(path, "Sub Folder")
    GetBuildPath = newpath
End Function
```
End Function
Description

Closes an open TextStream file.

Syntax

```
object.Close
```

The `object` is always the name of a TextStream object.

Remarks

The following example illustrates use of the Close method to close an open TextStream file:

```
Sub CreateAFile
  Dim fso, MyFile
  Set fso = CreateObject("Scripting.FileSystemObject")
  Set MyFile = fso.CreateTextFile("c:\testfile.txt", True)
  MyFile.WriteLine("This is a test.")
  MyFile.Close
End Sub
```
**Column Property**

**Description**

Read-only property that returns the column number of the current character position in a `TextStream` file.

**Syntax**

```
object.Column
```

The `object` is always the name of a `TextStream` object.

**Remarks**

After a newline character has been written, but before any other character is written, `Column` is equal to 1.

The following example illustrates use of the `Column` property:

```
Function GetColumn
    Const ForReading = 1, ForWriting = 2
    Dim fso, f, m
    Set fso = CreateObject("Scripting.FileSystemObject")
    Set f = fso.OpenTextFile("c:\testfile.txt", ForWriting, True)
    f.Write "Hello world!"
    f.Close
    Set f = fso.OpenTextFile("c:\testfile.txt", ForReading)
    m = f.ReadLine
```
GetColumn = f.Column
End Function
**CompareMode Property**

**Description**

Sets and returns the comparison mode for comparing string keys in a Dictionary object.

**Syntax**

```vbnet
object.CompareMode[ = compare]
```

The `CompareMode` property has the following parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>object</code></td>
<td>Required. Always the name of a Dictionary object.</td>
</tr>
<tr>
<td><code>compare</code></td>
<td>Optional. If provided, <code>compare</code> is a value representing the comparison mode used by functions such as <code>StrComp</code>.</td>
</tr>
</tbody>
</table>

**Settings**

The `compare` argument has the following settings:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>vbBinaryCompare</code></td>
<td>0</td>
<td>Perform a binary comparison.</td>
</tr>
<tr>
<td><code>vbTextCompare</code></td>
<td>1</td>
<td>Perform a textual comparison.</td>
</tr>
</tbody>
</table>

**Remarks**

Values greater than 2 can be used to refer to comparisons using specific
Locale IDs (LCID). An error occurs if you try to change the comparison mode of a Dictionary object that already contains data.

The CompareMode property uses the same values as the compare argument for the StrComp function.

The following example illustrates use of the CompareMode property:

Dim d
Set d = CreateObject("Scripting.Dictionary")
d.CompareMode = vbTextCompare
d.Add "a", "Athens" ' Add some keys and items
d.Add "b", "Belgrade"
d.Add "c", "Cairo"
d.Add "B", "Baltimore" ' Add method fails on this
                         ' letter b already exists in the Dictio
Copy Method

Description

Copies a specified file or folder from one location to another.

Syntax

\[ \text{object}.\text{Copy} \text{ destination}[, \text{overwrite}] \]

The Copy method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>Required. Always the name of a File or Folder object.</td>
</tr>
<tr>
<td>destination</td>
<td>Required. Destination where the file or folder is to be copied. Wildcard characters are not allowed.</td>
</tr>
<tr>
<td>overwrite</td>
<td>Optional. Boolean value that is True (default) if existing files or folders are to be overwritten; False if they are not.</td>
</tr>
</tbody>
</table>

Remarks

The results of the Copy method on a File or Folder are identical to operations performed using FileSystemObject.CopyFile or FileSystemObject.CopyFolder where the file or folder referred to by object is passed as an argument. You should note, however, that the alternative methods are capable of copying multiple files or folders.

The following example illustrates use of the Copy method:

```
Dim fso, MyFile
```
Set fso = CreateObject("Scripting.FileSystemObject")
Set MyFile = fso.CreateTextFile("c:\testfile.txt", True)
MyFile.WriteLine("This is a test.")
MyFile.Close
Set MyFile = fso.GetFile("c:\testfile.txt")
MyFile.Copy("c:\windows\desktop\test2.txt")
CopyFile Method

Description

Copies one or more files from one location to another.

Syntax

\texttt{object.CopyFile\ source,\ destination[,\ overwrite]}

The \texttt{CopyFile} method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>Required. The \texttt{object} is always the name of a \texttt{FileSystemObject}.</td>
</tr>
<tr>
<td>source</td>
<td>Required. Character string file specification, which can include wildcard characters, for one or more files to be copied.</td>
</tr>
<tr>
<td>destination</td>
<td>Required. Character string destination where the file or files from \texttt{source} are to be copied. Wildcard characters are not allowed.</td>
</tr>
<tr>
<td>overwrite</td>
<td>Optional. Boolean value that indicates if existing files are to be overwritten. If \texttt{True}, files are overwritten; if \texttt{False}, they are not. The default is \texttt{True}. Note that \texttt{CopyFile} will fail if \texttt{destination} has the read-only attribute set, regardless of the value of \texttt{overwrite}.</td>
</tr>
</tbody>
</table>

Remarks

Wildcard characters can only be used in the last path component of the \texttt{source} argument. For example, you can use:
FileSystemObject.CopyFile "c:\mydocuments\letters\*.doc", "c:\te

But you can't use:

FileSystemObject.CopyFile "c:\mydocuments\*\R1??97.xls", "c:\

If source contains wildcard characters or destination ends with a path separator (\), it is assumed that destination is an existing folder in which to copy matching files. Otherwise, destination is assumed to be the name of a file to create. In either case, three things can happen when an individual file is copied.

- If destination does not exist, source gets copied. This is the usual case.
- If destination is an existing file, an error occurs if overwrite is False. Otherwise, an attempt is made to copy source over the existing file.
- If destination is a directory, an error occurs.

An error also occurs if a source using wildcard characters doesn't match any files. The CopyFile method stops on the first error it encounters. No attempt is made to roll back or undo any changes made before an error occurs.
**CopyFolder Method**

**Description**

Recursively copies a folder from one location to another.

**Syntax**

```
object.CopyFolder source, destination[, overwrite]
```

The `CopyFolder` method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>Required. Always the name of a <code>FileSystemObject</code>.</td>
</tr>
<tr>
<td>source</td>
<td>Required. Character string folder specification, which can include wildcard characters, for one or more folders to be copied.</td>
</tr>
<tr>
<td>destination</td>
<td>Required. Character string destination where the folder and subfolders from <code>source</code> are to be copied. Wildcard characters are not allowed.</td>
</tr>
<tr>
<td>overwrite</td>
<td>Optional. Boolean value that indicates if existing folders are to be overwritten. If <code>True</code>, files are overwritten; if <code>False</code>, they are not. The default is <code>True</code>.</td>
</tr>
</tbody>
</table>

**Remarks**

Wildcard characters can only be used in the last path component of the `source` argument. For example, you can use:

```
FileSystemObject.CopyFolder "c:\mydocuments\letters\*", "c:\temp"
```
But you can't use:

    FileSystemObject.CopyFolder "c:\mydocuments\*\*", "c:\tempfolder"

If source contains wildcard characters or destination ends with a path separator (\), it is assumed that destination is an existing folder in which to copy matching folders and subfolders. Otherwise, destination is assumed to be the name of a folder to create. In either case, four things can happen when an individual folder is copied.

- If destination does not exist, the source folder and all its contents gets copied. This is the usual case.
- If destination is an existing file, an error occurs.
- If destination is a directory, an attempt is made to copy the folder and all its contents. If a file contained in source already exists in destination, an error occurs if overwrite is False. Otherwise, it will attempt to copy the file over the existing file.
- If destination is a read-only directory, an error occurs if an attempt is made to copy an existing read-only file into that directory and overwrite is False.

An error also occurs if a source using wildcard characters doesn't match any folders.

The CopyFolder method stops on the first error it encounters. No attempt is made to roll back any changes made before an error occurs.
Count Property

Description

Returns the number of items in a collection or Dictionary object. Read-only.

Syntax

object.Count

The object is always the name of one of the items in the Applies To list.

Remarks

The following code illustrates use of the Count property:

Function ShowKeys
    Dim a, d, i, s  ' Create some variables.
    Set d = CreateObject("Scripting.Dictionary")
    d.Add "a", "Athens"  ' Add some keys and items.
    d.Add "b", "Belgrade"
    d.Add "c", "Cairo"
    a = d.Keys  ' Get the keys.
    For i = 0 To d.Count - 1 ' Iterate the array.
        s = s & a(i) & "<BR>" ' Create return string.
    Next
    ShowKeys = s
End Function
CreateFolder Method

Description

Creates a folder.

Syntax

```
object.CreateFolder(foldername)
```

The CreateFolder method has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>Required. Always the name of a FileSystemObject.</td>
</tr>
<tr>
<td>foldername</td>
<td>Required. String expression that identifies the folder to create.</td>
</tr>
</tbody>
</table>

Remarks

An error occurs if the specified folder already exists.

The following example illustrates use of the CreateFolder method:

```vbs
Function CreateFolderDemo
    Dim fso, f
    Set fso = CreateObject("Scripting.FileSystemObject")
    Set f = fso.CreateFolder("c:\New Folder")
    CreateFolderDemo = f.Path
End Function
```
CreateTextFile Method

Description

Creates a specified file name and returns a TextStream object that can be used to read from or write to the file.

Syntax

```object.CreateTextFile(filename[, overwrite[, unicode]])```

The CreateTextFile method has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>Required. Always the name of a FileSystemObject or Folder object.</td>
</tr>
<tr>
<td>filename</td>
<td>Required. String expression that identifies the file to create.</td>
</tr>
<tr>
<td>overwrite</td>
<td>Optional. Boolean value that indicates if an existing file can be overwritten. The value is True if the file can be overwritten; False if it can't be overwritten. If omitted, existing files are not overwritten.</td>
</tr>
<tr>
<td>unicode</td>
<td>Optional. Boolean value that indicates whether the file is created as a Unicode or ASCII file. The value is True if the file is created as a Unicode file; False if it's created as an ASCII file. If omitted, an ASCII file is assumed.</td>
</tr>
</tbody>
</table>

Remarks

The following code illustrates how to use the CreateTextFile method to
create and open a text file:

Sub CreateAfile
    Dim fso, MyFile
    Set fso = CreateObject("Scripting.FileSystemObject")
    Set MyFile = fso.CreateTextFile("c:\testfile.txt", True)
    MyFile.WriteLine("This is a test.")
    MyFile.Close
End Sub

If the overwrite argument is **False**, or is not provided, for a **filename** that already exists, an error occurs.
**DateCreated**

**Property**

**Description**

Returns the date and time that the specified file or folder was created. Read-only.

**Syntax**

```
object.DateCreated
```

The `object` is always a File or Folder object.

**Remarks**

The following code illustrates the use of the **DateCreated** property with a file:

```
Function ShowFileInfo(filespec)
  Dim fso, f
  Set fso = CreateObject("Scripting.FileSystemObject")
  Set f = fso.GetFile(filespec)
  ShowFileInfo = "Created: " & f.DateCreated
End Function
```
DateLastAccessed Property

Description

Returns the date and time that the specified file or folder was last accessed. Read-only.

Syntax

`object.DateLastAccessed`

The `object` is always a `File` or `Folder` object.

Remarks

The following code illustrates the use of the `DateLastAccessed` property with a file:

```vbscript
Function ShowFileAccessInfo(filespec)
    Dim fso, f, s
    Set fso = CreateObject("Scripting.FileSystemObject")
    Set f = fso.GetFile(filespec)
    s = UCase(filespec) & "<BR>"
    s = s & "Created: " & f.DateCreated & "<BR>"
    s = s & "Last Accessed: " & f.DateLastAccessed & "<BR>"
    s = s & "Last Modified: " & f.DateLastModified
    ShowFileAccessInfo = s
End Function
```

Important This method depends on the underlying operating system for its behavior.
If the operating system does not support providing time information, none will be returned.
DateLastModified Property

Description

Returns the date and time that the specified file or folder was last modified. Read-only.

Syntax

```
object.DateLastModified
```

The object is always a File or Folder object.

Remarks

The following code illustrates the use of the DateLastModified property with a file:

```
Function ShowFileAccessInfo(filespec)
    Dim fso, f, s
    Set fso = CreateObject("Scripting.FileSystemObject")
    Set f = fso.GetFile(filespec)
    s = UCase(filespec) & "<BR>"
    s = s & "Created: " & f.DateCreated & "<BR>"
    s = s & "Last Accessed: " & f.DateLastAccessed & "<BR>"
    s = s & "Last Modified: " & f.DateLastModified
    ShowFileAccessInfo = s
End Function
```
Delete Method

Description

Deletes a specified file or folder.

Syntax

\textit{object}.Delete \textit{force}

The \textit{Delete} method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{object}</td>
<td>Required. Always the name of a \textit{File} or \textit{Folder} object.</td>
</tr>
<tr>
<td>\textit{force}</td>
<td>Optional. Boolean value that is \textbf{True} if files or folders with the read-only attribute set are to be deleted; \textbf{False} (default) if they are not.</td>
</tr>
</tbody>
</table>

Remarks

An error occurs if the specified file or folder does not exist. The \textit{Delete} method does not distinguish between folders that have contents and those that do not. The specified folder is deleted regardless of whether or not it has contents.

The results of the \textit{Delete} method on a \textit{File} or \textit{Folder} are identical to operations performed using \textit{FileSystemObject.DeleteFile} or \textit{FileSystemObject.DeleteFolder}.

The following example illustrates use of the \textit{Delete} method:

\begin{verbatim}
Dim fso, MyFile
Set fso = CreateObject("Scripting.FileSystemObject")
\end{verbatim}
Set MyFile = fso.CreateTextFile("c:\testfile.txt", True)
MyFile.WriteLine("This is a test.")
MyFile.Close
Set MyFile = fso.GetFile("c:\testfile.txt")
MyFile.Delete
DeleteFile Method

Description

Deletes a specified file.

Syntax

```
object.DeleteFile filespec[, force]
```

The **DeleteFile** method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>Required. Always the name of a <strong>FileSystemObject</strong>.</td>
</tr>
<tr>
<td>filespec</td>
<td>Required. The name of the file to delete. The <code>filespec</code></td>
</tr>
<tr>
<td></td>
<td>can contain wildcard characters in the last path component.</td>
</tr>
<tr>
<td>force</td>
<td>Optional. Boolean value that is <strong>True</strong> if files with the</td>
</tr>
<tr>
<td></td>
<td>read-only attribute set are to be deleted; <strong>False</strong> (default) if they are not.</td>
</tr>
</tbody>
</table>

Remarks

An error occurs if no matching files are found. The **DeleteFile** method stops on the first error it encounters. No attempt is made to roll back or undo any changes that were made before an error occurred.

The following example illustrates use of the **DeleteFile** method:

```VB
Sub DeleteAFile(filespec)
    Dim fso
    Set fso = CreateObject("Scripting.FileSystemObject")
    fso.DeleteFile filespec
End Sub
```
fso.DeleteFile(filespec)
End Sub
DeleteFolder Method

**Description**

Deletes a specified folder and its contents.

**Syntax**

```object.DeleteFolder folderspec[, force]```

The `DeleteFolder` method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>Required. Always the name of a <code>FileSystemObject</code>.</td>
</tr>
<tr>
<td>folderspec</td>
<td>Required. The name of the folder to delete. The <code>folderspec</code> can contain wildcard characters in the last path component.</td>
</tr>
<tr>
<td>force</td>
<td>Optional. Boolean value that is <code>True</code> if folders with the read-only attribute set are to be deleted; <code>False</code> (default) if they are not.</td>
</tr>
</tbody>
</table>

**Remarks**

The `DeleteFolder` method does not distinguish between folders that have contents and those that do not. The specified folder is deleted regardless of whether or not it has contents.

An error occurs if no matching folders are found. The `DeleteFolder` method stops on the first error it encounters. No attempt is made to roll back or undo any changes that were made before an error occurred.

The following example illustrates use of the `DeleteFolder` method:
Sub DeleteAFolder(filespec)
    Dim fso
    Set fso = CreateObject("Scripting.FileSystemObject")
    fso.DeleteFolder(filespec)
End Sub
**Dictionary Object**

**Description**

Object that stores data key, item pairs.

**Remarks**

A **Dictionary** object is the equivalent of a PERL associative array. Items, which can be any form of data, are stored in the array. Each item is associated with a unique key. The key is used to retrieve an individual item and is usually a integer or a string, but can be anything except an array.

The following code illustrates how to create a **Dictionary** object:

```vbscript
Dim d ' Create a variable.
Set d = CreateObject("Scripting.Dictionary")
d.Add "a", "Athens" ' Add some keys and items.
d.Add "b", "Belgrade"
d.Add "c", "Cairo"
...
```
**Drive Object**

**Description**

Provides access to the properties of a particular disk drive or network share.

**Remarks**

The following code illustrates the use of the **Drive** object to access drive properties:

```vbscript
Function ShowFreeSpace(drvPath)
    Dim fso, d, s
    Set fso = CreateObject("Scripting.FileSystemObject")
    Set d = fso.GetDrive(fso.GetDriveName(drvPath))
    s = "Drive " & UCase(drvPath) & " - "
    s = s & d.VolumeName & "<BR>"
    s = s & "Free Space: " & FormatNumber(d.FreeSpace/1024, 0) & " Kbytes"
    ShowFreeSpace = s
End Function
```
Drive Property

See Also  Applies To

Description

Returns the drive letter of the drive on which the specified file or folder resides. Read-only.

Syntax

```
object.Drive
```

The `object` is always a `File` or `Folder` object.

Remarks

The following code illustrates the use of the `Drive` property:

```
Function ShowFileAccessInfo(filespec)
    Dim fso, f, s
    Set fso = CreateObject("Scripting.FileSystemObject")
    Set f = fso.GetFile(filespec)
    s = f.Name & " on Drive " & UCase(f.Drive) & "<BR>"
    s = s & "Created: " & f.DateCreated & "<BR>"
    s = s & "Last Accessed: " & f.DateLastAccessed & "<BR>"
    s = s & "Last Modified: " & f.DateLastModified
    ShowFileAccessInfo = s
End Function
```
DriveExists Method

Description

Returns True if the specified drive exists; False if it does not.

Syntax

```
object.DriveExists(drivespec)
```

The DriveExists method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>Required. Always the name of a FileSystemObject.</td>
</tr>
<tr>
<td>drivespec</td>
<td>Required. A drive letter or a complete path specification.</td>
</tr>
</tbody>
</table>

Remarks

For drives with removable media, the DriveExists method returns True even if there are no media present. Use the IsReady property of the Drive object to determine if a drive is ready.

The following example illustrates use of the DriveExists method:

```
Function ReportDriveStatus(drv)
    Dim fso, msg
    Set fso = CreateObject("Scripting.FileSystemObject")
    If fso.DriveExists(drv) Then
        msg = ("Drive " & UCase(drv) & " exists.")
    Else
        msg = ("Drive " & UCase(drv) & " does not exist.")
    End If
    WScript.Echo msg
End Function
```
msg = ("Drive " & UCase(drv) & " doesn't exist.")
End If
ReportDriveStatus = msg
End Function
DriveLetter Property

See Also  Applies To

Description

Returns the drive letter of a physical local drive or a network share. Read-only.

Syntax

object.DriveLetter

The object is always a Drive object.

Remarks

The DriveLetter property returns a zero-length string ("") if the specified drive is not associated with a drive letter, for example, a network share that has not been mapped to a drive letter.

The following code illustrates the use of the DriveLetter property:

Function ShowDriveLetter(drvPath)
    Dim fso, d, s
    Set fso = CreateObject("Scripting.FileSystemObject")
    Set d = fso.GetDrive(fso.GetDriveName(drvPath))
    s = "Drive " & d.DriveLetter & ": - "
    s = s & d.VolumeName & "<BR>"
    s = s & "Free Space: " & FormatNumber(d.FreeSpace/1024, 0)
    s = s & " Kbytes"
    ShowDriveLetter = s
End Function
**Drives Collection**

**Description**

Read-only *collection* of all available drives.

**Remarks**

Removable-media drives need not have media inserted for them to appear in the *Drives* collection.

The following code illustrates how to get the *Drives* collection and iterate the collection using the `For Each...Next` statement:

```vbscript
Function ShowDriveList
    Dim fso, d, dc, s, n
    Set fso = CreateObject("Scripting.FileSystemObject")
    Set dc = fso.Drives
    For Each d in dc
        n = ""
        s = s & d.DriveLetter & " - "
        If d.DriveType = Remote Then
            n = d.ShareName
        ElseIf d.IsReady Then
            n = d.VolumeName
        End If
        s = s & n & "<BR>"
    Next
    ShowDriveList = s
End Function
```
**Drives Property**

**Description**

Returns a **Drives** collection consisting of all **Drive** objects available on the local machine.

**Syntax**

```vbscript
object.Drives
```

The `object` is always a **FileSystemObject**.

**Remarks**

Removable-media drives need not have media inserted for them to appear in the **Drives** collection.

You can iterate the members of the **Drives** collection using a **For Each...Next** construct as illustrated in the following code:

```vbscript
Function ShowDriveList
    Dim fso, d, dc, s, n
    Set fso = CreateObject("Scripting.FileSystemObject")
    Set dc = fso.Drives
    For Each d in dc
        n = ""
        s = s & d.DriveLetter & " - "
        If d.DriveType = 3 Then
            n = d.ShareName
        ElseIf d.IsReady Then
            n = d.VolumeName
        End If
        ' Additional processing...
    Next
End Function
```
End If
    s = s & n & "<BR>"
Next
ShowDriveList = s
End Function
These constants are only available when your project has an explicit reference to the appropriate type library containing these constant definitions. For VBScript, you must explicitly declare these constants in your code.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unknown</td>
<td>0</td>
<td>Drive type can't be determined.</td>
</tr>
<tr>
<td>Removable</td>
<td>1</td>
<td>Drive has removable media. This includes all floppy drives and many other varieties of storage devices.</td>
</tr>
<tr>
<td>Fixed</td>
<td>2</td>
<td>Drive has fixed (nonremovable) media. This includes all hard drives, including hard drives that are removable.</td>
</tr>
<tr>
<td>Remote</td>
<td>3</td>
<td>Network drives. This includes drives shared anywhere on a network.</td>
</tr>
<tr>
<td>CDROM</td>
<td>4</td>
<td>Drive is a CD-ROM. No distinction is made between read-only and read/write CD-ROM drives.</td>
</tr>
<tr>
<td>RAMDisk</td>
<td>5</td>
<td>Drive is a block of Random Access Memory (RAM) on the local computer that behaves like a disk drive.</td>
</tr>
</tbody>
</table>
DriveType Property

Description

Returns a value indicating the type of a specified drive.

Syntax

object.DriveType

The object is always a Drive object.

Remarks

The following code illustrates the use of the DriveType property:

Function ShowDriveType(drvpath)
    Dim fso, d, t
    Set fso = CreateObject("Scripting.FileSystemObject")
    Set d = fso.GetDrive(drvpath)
    Select Case d.DriveType
        Case 0: t = "Unknown"
        Case 1: t = "Removable"
        Case 2: t = "Fixed"
        Case 3: t = "Network"
        Case 4: t = "CD-ROM"
        Case 5: t = "RAM Disk"
    End Select
    ShowDriveType = "Drive " & d.DriveLetter & ": - " & t
End Function
**Exists Method**

### Description

Returns **True** if a specified key exists in the **Dictionary** object, **False** if it does not.

### Syntax

```
object.Exists(key)
```

The `Exists` method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>Required. Always the name of a <strong>Dictionary</strong> object.</td>
</tr>
<tr>
<td>key</td>
<td>Required. Key value being searched for in the <strong>Dictionary</strong> object.</td>
</tr>
</tbody>
</table>

### Remarks

The following example illustrates use of the `Exists` method:

```vbs
Function KeyExistsDemo
    Dim d, msg
    ' Create some variables.
    Set d = CreateObject("Scripting.Dictionary")
    d.Add "a", "Athens"  ' Add some keys and items.
    d.Add "b", "Belgrade"
    d.Add "c", "Cairo"
    If d.Exists("c") Then
        msg = "Specified key exists."
    Else
```

```vbs```
msg = "Specified key doesn't exist."
End If
KeyExistsDemo = msg
End Function
These constants are only available when your project has an explicit reference to the appropriate type library containing these constant definitions. For VBScript, you must explicitly declare these constants in your code.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>0</td>
<td>Normal file. No attributes are set.</td>
</tr>
<tr>
<td>ReadOnly</td>
<td>1</td>
<td>Read-only file.</td>
</tr>
<tr>
<td>Hidden</td>
<td>2</td>
<td>Hidden file.</td>
</tr>
<tr>
<td>System</td>
<td>4</td>
<td>System file.</td>
</tr>
<tr>
<td>Directory</td>
<td>16</td>
<td>Folder or directory.</td>
</tr>
<tr>
<td>Archive</td>
<td>32</td>
<td>File has changed since last backup.</td>
</tr>
<tr>
<td>Alias</td>
<td>1024</td>
<td>Link or shortcut.</td>
</tr>
<tr>
<td>Compressed</td>
<td>2048</td>
<td>Compressed file.</td>
</tr>
</tbody>
</table>
These constants are only available when your project has an explicit reference to the appropriate type library containing these constant definitions. For VBScript, you must explicitly declare these constants in your code.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ForReading</td>
<td>1</td>
<td>Open a file for reading only. You can't write to this file.</td>
</tr>
<tr>
<td>ForWriting</td>
<td>2</td>
<td>Open a file for writing. If a file with the same name exists, its previous contents are overwritten.</td>
</tr>
<tr>
<td>ForAppending</td>
<td>8</td>
<td>Open a file and write to the end of the file.</td>
</tr>
</tbody>
</table>
FileExists Method

Description

Returns **True** if a specified file exists; **False** if it does not.

Syntax

```
object.FileExists(filespec)
```

The `FileExists` method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>object</code></td>
<td>Required. Always the name of a <strong>FileSystemObject</strong>.</td>
</tr>
<tr>
<td><code>filespec</code></td>
<td>Required. The name of the file whose existence is to be determined. A complete path specification (either absolute or relative) must be provided if the file isn't expected to exist in the current folder.</td>
</tr>
</tbody>
</table>

Remarks

The following example illustrates use of the `FileExists` method:

```
Function ReportFileStatus(filespec)
    Dim fso, msg
    Set fso = CreateObject("Scripting.FileSystemObject")
    If (fso.FileExists(filespec)) Then
        msg = filespec & " exists."
    Else
        msg = filespec & " doesn't exist."
    End If
    ReportFileStatus = msg
End Function
```
End Function
**File Object**

**Description**

Provides access to all the properties of a file.

**Remarks**

The following code illustrates how to obtain a `File` object and how to view one of its properties.

```vbscript
Function ShowDateCreated(filespec)
    Dim fso, f
    Set fso = CreateObject("Scripting.FileSystemObject")
    Set f = fso.GetFile(filespec)
    ShowDateCreated = f.DateCreated
End Function
```
Files Collection

See Also  Properties  Methods

Description

Collection of all File objects within a folder.

Remarks

The following code illustrates how to get a Files collection and iterate the collection using the For Each...Next statement:

Function ShowFolderList(folderspec)
    Dim fso, f, f1, fc, s
    Set fso = CreateObject("Scripting.FileSystemObject")
    Set f = fso.GetFolder(folderspec)
    Set fc = f.Files
    For Each f1 in fc
        s = s & f1.name
        s = s & "<BR>"
    Next
    ShowFolderList = s
End Function
Files

Property

Description

Returns a Files collection consisting of all File objects contained in the specified folder, including those with hidden and system file attributes set.

Syntax

object.Files

The object is always a Folder object.

Remarks

The following code illustrates the use of the Files property:

Function ShowFileList(folderspec)
    Dim fso, f, f1, fc, s
    Set fso = CreateObject("Scripting.FileSystemObject")
    Set f = fso.GetFolder(folderspec)
    Set fc = f.Files
    For Each f1 in fc
        s = s & f1.name
        s = s & "<BR>"
    Next
    ShowFileList = s
End Function
FileSystemObject Object

Description

Provides access to a computer's file system.

Remarks

The following code illustrates how the FileSystemObject is used to return a TextStream object that can be read from or written to:

```vbnet
Dim fso, MyFile
Set fso = CreateObject("Scripting.FileSystemObject")
Set MyFile = fso.CreateTextFile("c:\testfile.txt", True)
MyFile.WriteLine("This is a test.")
MyFile.Close
```

In the preceding code, the CreateObject function returns the FileSystemObject (fso). The CreateTextFile method then creates the file as a TextStream object (a) and the WriteLine method writes a line of text to the created text file. The Close method flushes the buffer and closes the file.
See Also

FileSystemObject has a number of useful constants that you can use in your code. Constants provide a convenient way to use specific values without actually having to remember the value itself. They also make your code more maintainable, should the value of any constant ever change.

Depending on your scripting host, these constants may be already defined. If so, simply use the constants anywhere in your code in place of the values they represent. In cases where SCRRUN.DLL is not explicitly referenced by your scripting host, you'll have to define these constants in your code before you can use them. Examples of this case include Microsoft Internet Explorer and Microsoft Internet Information Services (IIS).

The following list describes the various categories of constants provided for the FileSystemObject, along with a brief description:

- **DriveType Constants**
  Defines the various drive types available on the host computer, such as Fixed, Removable, CD-ROM, etc.

- **File Attribute Constants**
  Defines various file attributes such as Hidden, Read-Only, etc.

- **File Input/Output Constants**
  Defines constants used with file input and output.

- **SpecialFolder Constants**
  Defines special folders available in your operating system.
Description

Returns the type of file system in use for the specified drive.

Syntax

\textit{object.FileSystem}

The \textit{object} is always a \texttt{Drive} object.

Remarks

Available return types include FAT, NTFS, and CDFS.

The following code illustrates the use of the \texttt{FileSystem} property:

\begin{verbatim}
Function ShowFileSystemType(drvspec)
    Dim fso,d
    Set fso = CreateObject("Scripting.FileSystemObject")
    Set d = fso.GetDrive(drvspec)
    ShowFileSystemType = d.FileSystem
End Function
\end{verbatim}
Folder Object

Description

Provides access to all the properties of a folder.

Remarks

The following code illustrates how to obtain a Folder object and how to return one of its properties:

```vbscript
Function ShowDateCreated(folderspec)
    Dim fso, f
    Set fso = CreateObject("Scripting.FileSystemObject")
    Set f = fso.GetFolder(folderspec)
    ShowDateCreated = f.DateCreated
End Function
```
**Folders Collection**

**See Also**  
Properties  
Methods

**Description**

Collection of all **Folder** objects contained within a **Folder** object.

**Remarks**

The following code illustrates how to get a **Folders** collection and how to iterate the collection using the **For Each...Next** statement:

```vbscript
Function ShowFolderList(folderspec)
    Dim fso, f, f1, fc, s
    Set fso = CreateObject("Scripting.FileSystemObject")
    Set f = fso.GetFolder(folderspec)
    Set fc = f.SubFolders
    For Each f1 in fc
        s = s & f1.name
        s = s & "<BR>"
    Next
    ShowFolderList = s
End Function
```
FolderExists Method

Description

Returns True if a specified folder exists; False if it does not.

Syntax

```
object.FolderExists(folderspec)
```

The FolderExists method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>Required. Always the name of a FileSystemObject.</td>
</tr>
<tr>
<td>folderspec</td>
<td>Required. The name of the folder whose existence is to be determined. A complete path specification (either absolute or relative) must be provided if the folder isn't expected to exist in the current folder.</td>
</tr>
</tbody>
</table>

Remarks

The following example illustrates use of the FolderExists method:

```
Function ReportFolderStatus(fldr)
    Dim fso, msg
    Set fso = CreateObject("Scripting.FileSystemObject")
    If (fso.FolderExists(fldr)) Then
        msg = fldr & " exists."
    Else
        msg = fldr & " doesn't exist."
End Function
```
End If
ReportFolderStatus = msg
End Function
Description

Returns the amount of free space available to a user on the specified drive or network share. Read-only.

Syntax

`object.FreeSpace`

The `object` is always a `Drive` object.

Remarks

The value returned by the `FreeSpace` property is typically the same as that returned by the `AvailableSpace` property. Differences may occur between the two for computer systems that support quotas.

The following code illustrates the use of the `FreeSpace` property:

```vbscript
Function ShowFreeSpace(drvPath)
    Dim fso, d, s
    Set fso = CreateObject("Scripting.FileSystemObject")
    Set d = fso.GetDrive(fso.GetDriveName(drvPath))
    s = "Drive " & UCase(drvPath) & " - "
    s = s & d.VolumeName & "<BR>"
    s = s & "Free Space: " & FormatNumber(d.FreeSpace/1024, 0) & "Kbytes"
    ShowFreeSpace = s
End Function
```
GetAbsolutePathName Method

See Also

Applies To

Description

Returns a complete and unambiguous path from a provided path specification.

Syntax

object.GetAbsolutePathName(pathspec)

The GetAbsolutePathName method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>Required. Always the name of a FileSystemObject.</td>
</tr>
<tr>
<td>pathspec</td>
<td>Required. Path specification to change to a complete and unambiguous path.</td>
</tr>
</tbody>
</table>

Remarks

A path is complete and unambiguous if it provides a complete reference from the root of the specified drive. A complete path can only end with a path separator character (\) if it specifies the root folder of a mapped drive.

Assuming the current directory is c:\mydocuments\reports, the following table illustrates the behavior of the GetAbsolutePathName method.

<table>
<thead>
<tr>
<th>pathspec</th>
<th>Returned path</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;c:&quot;</td>
<td>&quot;c:\mydocuments\reports&quot;</td>
</tr>
<tr>
<td>&quot;c:&quot;</td>
<td>&quot;c:\mydocuments&quot;</td>
</tr>
<tr>
<td>&quot;c:\&quot;</td>
<td>&quot;c:&quot;</td>
</tr>
<tr>
<td>&quot;c:<em>.</em>\may97&quot;</td>
<td>&quot;c:\mydocuments\reports*.\may97&quot;</td>
</tr>
<tr>
<td>&quot;region1&quot;</td>
<td>&quot;c:\mydocuments\reports\region1&quot;</td>
</tr>
<tr>
<td>&quot;c:....\mydocuments&quot;</td>
<td>&quot;c:\mydocuments&quot;</td>
</tr>
</tbody>
</table>
GetBaseName Method

Description

Returns a string containing the base name of the file (less any file extension), or folder in a provided path specification.

Syntax

\[ \text{object}.\text{GetBaseName(path)} \]

The \text{GetBaseName} method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>Required. Always the name of a FileSystemObject.</td>
</tr>
<tr>
<td>path</td>
<td>Required. The path specification for the file or folder whose base name is to be returned.</td>
</tr>
</tbody>
</table>

Remarks

The \text{GetBaseName} method returns a zero-length string ("") if no file or folder matches the \text{path} argument.

The following example illustrates use of the \text{GetBaseName} method:

```
Function GetTheBase(filespec)
    Dim fso
    Set fso = CreateObject("Scripting.FileSystemObject")
    GetTheBase = fso.GetBaseName(filespec)
End Function
```
**Note**  The **GetBaseName** method works only on the provided *path* string. It does not attempt to resolve the path, nor does it check for the existence of the specified path.
GetDrive Method

Description

Returns a Drive object corresponding to the drive in a specified path.

Syntax

object.GetDrive drivespec

The GetDrive method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>Required. Always the name of a FileSystemObject.</td>
</tr>
<tr>
<td>drivespec</td>
<td>Required. The drivespec argument can be a drive letter (c), a drive letter with a colon appended (c:), a drive letter with a colon and path separator appended (c:), or any network share specification (\computer2\share1).</td>
</tr>
</tbody>
</table>

Remarks

For network shares, a check is made to ensure that the share exists.

An error occurs if drivespec does not conform to one of the accepted forms or does not exist. To call the GetDrive method on a normal path string, use the following sequence to get a string that is suitable for use as drivespec:

```
DriveSpec = GetDriveName(GetAbsolutePathName(Path))
```

The following example illustrates use of the GetDrive method:

```
Function ShowFreeSpace(drvPath)
    Dim fso, d, s
    Set fso = CreateObject("Scripting.FileSystemObject")
```

```
Set d = fso.GetDrive(fso.GetDriveName(drvPath))
s = "Drive " & UCase(drvPath) & " - "
s = s & d.VolumeName & "<BR>"
s = s & "Free Space: " & FormatNumber(d.FreeSpace/1024, 0)
s = s & " Kbytes"
ShowFreeSpace = s
End Function
GetDriveName Method

Description

Returns a string containing the name of the drive for a specified path.

Syntax

object.GetDriveName(path)

The GetDriveName method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>Required. Always the name of a FileSystemObject.</td>
</tr>
<tr>
<td>path</td>
<td>Required. The path specification for the component whose drive name is to be returned.</td>
</tr>
</tbody>
</table>

Remarks

The GetDriveName method returns a zero-length string (""") if the drive can't be determined.

The following example illustrates use of the GetDriveName method:

```
Function GetAName(DriveSpec)
    Dim fso
    Set fso = CreateObject("Scripting.FileSystemObject")
    GetAName = fso.GetDriveName(Drivespec)
End Function
```
Note  The **GetDriveName** method works only on the provided *path* string. It does not attempt to resolve the path, nor does it check for the existence of the specified path.
GetExtensionName Method

Description

Returns a string containing the extension name for the last component in a path.

Syntax

```vbscript
object.GetExtensionName(path)
```

The `GetExtensionName` method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>object</code></td>
<td>Required. Always the name of a <code>FileSystemObject</code>.</td>
</tr>
<tr>
<td><code>path</code></td>
<td>Required. The path specification for the component whose extension name is to be returned.</td>
</tr>
</tbody>
</table>

Remarks

For network drives, the root directory (\) is considered to be a component.

The `GetExtensionName` method returns a zero-length string ("") if no component matches the `path` argument.

The following example illustrates use of the `GetExtensionName` method:

```vbscript
Function GetAnExtension(DriveSpec)
    Dim fso
    Set fso = CreateObject("Scripting.FileSystemObject")
    GetAnExtension = fso.GetExtensionName(DriveSpec)
End Function
```
End Function
GetFile Method

Description

Returns a File object corresponding to the file in a specified path.

Syntax

```object.GetFile(filespec)```

The `GetFile` method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>object</code></td>
<td>Required. Always the name of a FileSystemObject.</td>
</tr>
<tr>
<td><code>filespec</code></td>
<td>Required. The <code>filespec</code> is the path (absolute or relative) to a specific file.</td>
</tr>
</tbody>
</table>

Remarks

An error occurs if the specified file does not exist.

The following example illustrates use of the `GetFile` method:

```vbscript
Function ShowFileAccessInfo(filespec)
    Dim fso, f, s
    Set fso = CreateObject("Scripting.FileSystemObject")
    Set f = fso.GetFile(filespec)
    s = f.Path & "<br>"
    s = s & "Created: " & f.DateCreated & "<br>"
    s = s & "Last Access: " & f.DateLastAccessed & "<br>"
    s = s & "Last Modified: " & f.DateLastModified
    ShowFileAccessInfo = s
End Function
```
End Function
GetFileName Method

Description

Returns the last file name or folder of a specified path that is not part of the drive specification.

Syntax

```
object.GetFileName(pathspec)
```

The `GetFileName` method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>Required. Always the name of a <code>FileSystemObject</code>.</td>
</tr>
<tr>
<td>pathspec</td>
<td>Required. The path (absolute or relative) to a specific file.</td>
</tr>
</tbody>
</table>

Remarks

The `GetFileName` method returns a zero-length string (""”) if `pathspec` does not end with the named file or folder.

The following example illustrates use of the `GetFileName` method:

```
Function GetAName(DriveSpec)
    Dim fso
    Set fso = CreateObject("Scripting.FileSystemObject")
    GetAName = fso.GetFileName(DriveSpec)
End Function
```
Note The \texttt{GetFileName} method works only on the provided path string. It does not attempt to resolve the path, nor does it check for the existence of the specified path.
GetFileVersion Method

Description

Returns the version number of a specified file.

Syntax

```
object.GetFileVersion(pathspec)
```

The `GetVersion` method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>Required. Always the name of a <code>FileSystemObject</code>.</td>
</tr>
<tr>
<td>pathspec</td>
<td>Required. The path (absolute or relative) to a specific file.</td>
</tr>
</tbody>
</table>

Remarks

The `GetFileVersion` method returns a zero-length string (""') if `pathspec` does not end with the named file or if the file does not contain version information.

The following example illustrates use of the `GetFileVersion` method:

```
Function GetVersion(PathSpec)
    Dim fso, temp
    Set fso = CreateObject("Scripting.FileSystemObject")
    temp = fso.GetFileVersion(PathSpec)
    If Len(temp) Then
```

GetVersion = temp
Else
    GetVersion = "No version information available."
End If
End Function

---

**Note**  The GetFileVersion method works only on the provided path string. It does not attempt to resolve the path, nor does it check for the existence of the specified path.
GetFolder Method

Description

Returns a **Folder** object corresponding to the folder in a specified path.

Syntax

```
object.GetFolder(folderspec)
```

The **GetFolder** method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>Required. Always the name of a <strong>FileSystemObject</strong>.</td>
</tr>
<tr>
<td>folderspec</td>
<td>Required. The <strong>folderspec</strong> is the path (absolute or relative) to a specific folder.</td>
</tr>
</tbody>
</table>

Remarks

An error occurs if the specified folder does not exist.

The following example illustrates the use of the **GetFolder** method to return a folder object:

```
Sub AddNewFolder(path, folderName)
    Dim fso, f, fc, nf
    Set fso = CreateObject("Scripting.FileSystemObject")
    Set f = fso.GetFolder(path)
    Set fc = f.SubFolders
    If folderName <> "" Then
        Set nf = fc.Add(folderName)
```

Else
    Set nf = fc.Add("New Folder")
End If
End Sub
GetParentFolderName Method

Description

Returns a string containing the name of the parent folder of the last file or folder in a specified path.

Syntax

object.GetParentFolderName(path)

The GetParentFolderName method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>Required. Always the name of a FileSystemObject.</td>
</tr>
<tr>
<td>path</td>
<td>Required. The path specification for the file or folder whose parent folder name is to be returned.</td>
</tr>
</tbody>
</table>

Remarks

The GetParentFolderName method returns a zero-length string ("") if there is no parent folder for the file or folder specified in the path argument.

The following example illustrates use of the GetParentFolderName method:

```vbscript
Function GetTheParent(DriveSpec)
    Dim fso
    Set fso = CreateObject("Scripting.FileSystemObject")
    GetTheParent = fso.GetParentFolderName(DriveSpec)
End Function
```
Note  The **GetParentFolderName** method works only on the provided *path* string. It does not attempt to resolve the path, nor does it check for the existence of the specified path.
**GetSpecialFolder Method**

**Description**

Returns the special folder specified.

**Syntax**

`object.GetSpecialFolder(folderspec)`

The `GetSpecialFolder` method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>object</code></td>
<td>Required. Always the name of a <code>FileSystemObject</code>.</td>
</tr>
<tr>
<td><code>folderspec</code></td>
<td>Required. The name of the special folder to be returned. Can be any of the constants shown in the Settings section.</td>
</tr>
</tbody>
</table>

**Settings**

The `folderspec` argument can have any of the following values:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>WindowsFolder</code></td>
<td>0</td>
<td>The Windows folder contains files installed by the Windows operating system.</td>
</tr>
<tr>
<td><code>SystemFolder</code></td>
<td>1</td>
<td>The System folder contains libraries, fonts, and device drivers.</td>
</tr>
<tr>
<td><code>TemporaryFolder</code></td>
<td>2</td>
<td>The Temp folder is used to store temporary files. Its path is found in the TMP environment variable.</td>
</tr>
</tbody>
</table>
Remarks

The following example illustrates use of the GetSpecialFolder method:

```vba
Dim fso, tempfile
Set fso = CreateObject("Scripting.FileSystemObject")

Function CreateTempFile
    Dim tfolder, tname, tfile
    Const TemporaryFolder = 2
    Set tfolder = fso.GetSpecialFolder(TemporaryFolder)
    tname = fso.GetTempName
    Set tfile = tfolder.CreateTextFile(tname)
    Set CreateTempFile = tfile
End Function

Set tempfile = CreateTempFile
tempfile.WriteLine "Hello World"
tempfile.Close
```
GetTempName Method

Description

Returns a randomly generated temporary file or folder name that is useful for performing operations that require a temporary file or folder.

Syntax

`object.GetTempName`

The optional `object` is always the name of a `FileSystemObject`.

Remarks

The `GetTempName` method does not create a file. It provides only a temporary file name that can be used with `CreateTextFile` to create a file.

The following example illustrates use of the `GetTempName` method:

```vbscript
Dim fso, tempfile
Set fso = CreateObject("Scripting.FileSystemObject")
Function CreateTempFile
    Dim tfolder, tname, tfile
    Const TemporaryFolder = 2
    Set tfolder = fso.GetSpecialFolder(TemporaryFolder)
    tname = fso.GetTempName
    ' Use tname...
End Function
```

See Also

Applies To
Set tfile = tfolder.CreateTextFile(tname)
Set CreateTempFile = tfile
End Function

Set tempfile = CreateTempFile
tempfile.WriteLine "Hello World"
tempfile.Close
IsReady Property

Description

Returns **True** if the specified drive is ready; **False** if it is not.

Syntax

```
object.IsReady
```

The `object` is always a `Drive` object.

Remarks

For removable-media drives and CD-ROM drives, **IsReady** returns **True** only when the appropriate media is inserted and ready for access.

The following code illustrates the use of the **IsReady** property:

```
Function ShowDriveInfo(drvpath)
    Dim fso, d, s, t
    Set fso = CreateObject("Scripting.FileSystemObject")
    Set d = fso.GetDrive(drvpath)
    Select Case d.DriveType
        Case 0: t = "Unknown"
        Case 1: t = "Removable"
        Case 2: t = "Fixed"
        Case 3: t = "Network"
        Case 4: t = "CD-ROM"
        Case 5: t = "RAM Disk"
    End Select
    s = "Drive " & d.DriveLetter & ": - " & t
End Function
```
If d.IsReady Then
    s = s & "<BR>" & "Drive is Ready."
Else
    s = s & "<BR>" & "Drive is not Ready."
End If
ShowDriveInfo = s
End Function
IsRootFolder Property

Description

Returns **True** if the specified folder is the root folder; **False** if it is not.

Syntax

```
object.IsRootFolder
```

The `object` is always a `Folder` object.

Remarks

The following code illustrates the use of the `IsRootFolder` property:

```
Function DisplayLevelDepth(pathspec)
    Dim fso, f, n
    Set fso = CreateObject("Scripting.FileSystemObject")
    Set f = fso.GetFolder(pathspec)
    If f.IsRootFolder Then
        DisplayLevelDepth = "The specified folder is the root folder."
    Else
        Do Until f.IsRootFolder
            Set f = f.ParentFolder
            n = n + 1
        Loop
        DisplayLevelDepth = "The specified folder is nested " & n & " levels deep."
    End If
End Function
```
**Item**

**Property**

**See Also**

**Applies To**

**Description**

Sets or returns an *item* for a specified *key* in a **Dictionary** object. For collections, returns an *item* based on the specified *key*. Read/write.

**Syntax**

```vbnet
object.Item(key) [= newItem]
```

The **Item** property has the following parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>Required. Always the name of a collection or <strong>Dictionary</strong> object.</td>
</tr>
<tr>
<td>key</td>
<td>Required. <em>Key</em> associated with the <em>item</em> being retrieved or added.</td>
</tr>
<tr>
<td>newItem</td>
<td>Optional. Used for <strong>Dictionary</strong> object only; no application for collections. If provided, <em>newItem</em> is the new value associated with the specified <em>key</em>.</td>
</tr>
</tbody>
</table>

**Remarks**

If *key* is not found when changing an *item*, a new *key* is created with the specified *newItem*. If *key* is not found when attempting to return an existing item, a new *key* is created and its corresponding item is left empty.

The following example illustrates the use of the **Item** property:

**Function ItemDemo**
Dim d ' Create some variables.
Set d = CreateObject("Scripting.Dictionary")
d.Add "a", "Athens"  ' Add some keys and items
  
d.Add "b", "Belgrade"
d.Add "c", "Cairo"
  
  ItemDemo = d.Item("c") ' Get the item.
End Function
**Items Method**

**Description**

Returns an array containing all the items in a Dictionary object.

**Syntax**

`object.Items`

The `object` is always the name of a Dictionary object.

**Remarks**

The following code illustrates use of the Items method:

```vbnet
Function DicDemo
    Dim a, d, i, s  ' Create some variables.
    Set d = CreateObject("Scripting.Dictionary")
    d.Add "a", "Athens"  ' Add some keys and items.
    d.Add "b", "Belgrade"
    d.Add "c", "Cairo"
    a = d.Items  ' Get the items.
    For i = 0 To d.Count -1 ' Iterate the array.
        s = s & a(i) & "<BR>" ' Create return string.
    Next
    DicDemo = s
End Function
```
**Key Property**

**See Also**

**Applies to**

**Description**

Sets a *key* in a **Dictionary** object.

**Syntax**

```
object.Key(key) = newkey
```

The `Key` property has the following parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>object</code></td>
<td>Required. Always the name of a <strong>Dictionary</strong> object.</td>
</tr>
<tr>
<td><code>key</code></td>
<td>Required. <em>Key</em> value being changed.</td>
</tr>
<tr>
<td><code>newkey</code></td>
<td>Required. New value that replaces the specified <em>key</em>.</td>
</tr>
</tbody>
</table>

**Remarks**

If *key* is not found when changing a *key*, a **run-time error** will occur.

The following example illustrates the use of the `Key` property:

**Function DicDemo**

```vbscript
Function DicDemo
    Dim d                   ' Create some variables.
    Set d = CreateObject("Scripting.Dictionary")
    d.Add "a", "Athens"     ' Add some keys and items
    d.Add "b", "Belgrade"
    d.Add "c", "Cairo"
End Function
```
d.Key("c") = "d" ' Set key for "c" to "d".
DicDemo = d.Item("d") ' Return associate item.
End Function
**Keys Method**

**Description**

Returns an array containing all existing keys in a Dictionary object.

**Syntax**

```
object.Keys
```

The `object` is always the name of a Dictionary object.

**Remarks**

The following code illustrates use of the `Keys` method:

```vbnet
Function DicDemo
    Dim a, d, i        ' Create some variables.
    Set d = CreateObject("Scripting.Dictionary")
    d.Add "a", "Athens"    ' Add some keys and items.
    d.Add "b", "Belgrade"
    d.Add "c", "Cairo"
    a = d.Keys           ' Get the keys.
    For i = 0 To d.Count -1 ' Iterate the array.
        s = s & a(i) & "<BR>" ' Return results.
    Next
    DicDemo = s
End Function
```
**Line Property**

**Description**

Read-only property that returns the current line number in a `TextStream` file.

**Syntax**

```vbscript
object.Line
```

The `object` is always the name of a `TextStream` object.

**Remarks**

After a file is initially opened and before anything is written, `Line` is equal to 1.

The following example illustrates use of the `Line` property:

```vbscript
Function GetLine
    Const ForReading = 1, ForWriting = 2
    Dim fso, f, ra
    Set fso = CreateObject("Scripting.FileSystemObject")
    Set f = fso.OpenTextFile("c:\testfile.txt", ForWriting, True)
    f.Write "Hello world!" & vbCrLf & "VB Script is fun!" & vbCrL
    Set f = fso.OpenTextFile("c:\testfile.txt", ForReading)
    ra = f.ReadAll
    GetLine = f.Line
End Function
```
Move Method

Description

Moves a specified file or folder from one location to another.

Syntax

object.Move destination

The Move method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>Required. Always the name of a File or Folder object.</td>
</tr>
<tr>
<td>destination</td>
<td>Required. Destination where the file or folder is to be moved. Wildcard characters are not allowed.</td>
</tr>
</tbody>
</table>

Remarks

The results of the Move method on a File or Folder are identical to operations performed using FileSystemObject.MoveFile or FileSystemObject.MoveFolder. You should note, however, that the alternative methods are capable of moving multiple files or folders.

The following example illustrates use of the Move method:

```vbs
Dim fso, MyFile
Set fso = CreateObject("Scripting.FileSystemObject")
Set MyFile = fso.CreateTextFile("c:\testfile.txt", True)
MyFile.WriteLine("This is a test.")
```
MyFile.Close
Set MyFile = fso.GetFile("c:\testfile.txt")
MyFile.Move "c:\windows\desktop\"
MoveFile Method

See Also  Applies To

Description

Moves one or more files from one location to another.

Syntax

object.MoveFile source, destination

The MoveFile method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>Required. Always the name of a FileSystemObject.</td>
</tr>
<tr>
<td>source</td>
<td>Required. The path to the file or files to be moved. The source argument string can contain wildcard characters in the last path component only.</td>
</tr>
<tr>
<td>destination</td>
<td>Required. The path where the file or files are to be moved. The destination argument can't contain wildcard characters.</td>
</tr>
</tbody>
</table>

Remarks

If source contains wildcards or destination ends with a path separator (\), it is assumed that destination specifies an existing folder in which to move the matching files. Otherwise, destination is assumed to be the name of a destination file to create. In either case, three things can happen when an individual file is moved:

- If destination does not exist, the file gets moved. This is the usual case.
- If destination is an existing file, an error occurs.
- If destination is a directory, an error occurs.
An error also occurs if a wildcard character that is used in source doesn't match any files. The MoveFile method stops on the first error it encounters. No attempt is made to roll back any changes made before the error occurs.

The following example illustrates use of the MoveFile method:

```vba
Sub MoveAFile(Drivespec)
    Dim fso
    Set fso = CreateObject("Scripting.FileSystemObject")
    fso.MoveFile Drivespec, "c:\windows\desktop\"
End Sub
```

---

Important This method allows moving files between volumes only if supported by the operating system.
MoveFolder Method

Description

Moves one or more folders from one location to another.

Syntax

object.MoveFolder source, destination

The MoveFolder method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>Required. Always the name of a FileSystemObject.</td>
</tr>
<tr>
<td>source</td>
<td>Required. The path to the folder or folders to be moved. The source argument string can contain wildcard characters in the last path component only.</td>
</tr>
<tr>
<td>destination</td>
<td>Required. The path where the folder or folders are to be moved. The destination argument can't contain wildcard characters.</td>
</tr>
</tbody>
</table>

Remarks

If source contains wildcards or destination ends with a path separator (\), it is assumed that destination specifies an existing folder in which to move the matching files. Otherwise, destination is assumed to be the name of a destination folder to create. In either case, three things can happen when an individual folder is moved:

- If destination does not exist, the folder gets moved. This is the usual case.
- If *destination* is an existing file, an error occurs.
- If *destination* is a directory, an error occurs.

An error also occurs if a wildcard character that is used in *source* doesn't match any folders. The **MoveFolder** method stops on the first error it encounters. No attempt is made to roll back any changes made before the error occurs.

The following example illustrates use of the **MoveFolder** method:

```vba
Sub MoveAFolder(Drivespec)
    Dim fso
    Set fso = CreateObject("Scripting.FileSystemObject")
fso.MoveFolder Drivespec, "c:\windows\desktop\"
End Sub
```

**Important** This method allows moving folders between volumes only if supported by the operating system.
Name Property

See Also       Applies To

Description

Sets or returns the name of a specified file or folder. Read/write.

Syntax

object.Name [= newname]

The Name property has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>Required. Always the name of a File or Folder object.</td>
</tr>
<tr>
<td>newname</td>
<td>Optional. If provided, newname is the new name of the specified object.</td>
</tr>
</tbody>
</table>

Remarks

The following code illustrates the use of the Name property:

Function ShowFileAccessInfo(filespec)
    Dim fso, f, s
    Set fso = CreateObject("Scripting.FileSystemObject")
    Set f = fso.GetFile(filespec)
    s = f.Name & " on Drive " & UCase(f.Drive) & "<BR>"
    s = s & "Created: " & f.DateCreated & "<BR>"
    s = s & "Last Accessed: " & f.DateLastAccessed & "<BR>"
    s = s & "Last Modified: " & f.DateLastModified
    ShowFileAccessInfo = s
End Function
OpenAsTextStream Method

Description

Opens a specified file and returns a TextStream object that can be used to read from, write to, or append to the file.

Syntax

```
object.OpenAsTextStream([iomode, [format]])
```

The OpenAsTextStream method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>Required. Always the name of a File object.</td>
</tr>
<tr>
<td>iomode</td>
<td>Optional. Indicates input/output mode. Can be one of three constants: ForReading, ForWriting, or ForAppending.</td>
</tr>
<tr>
<td>format</td>
<td>Optional. One of three Tristate values used to indicate the format of the opened file. If omitted, the file is opened as ASCII.</td>
</tr>
</tbody>
</table>

Settings

The iomode argument can have any of the following settings:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ForReading</td>
<td>1</td>
<td>Open a file for reading only. You can't write to this file.</td>
</tr>
<tr>
<td>ForWriting</td>
<td>2</td>
<td>Open a file for writing. If a file with the same name exists, its previous contents</td>
</tr>
</tbody>
</table>
are overwritten.

| ForAppending | 8   | Open a file and write to the end of the file. |

The format argument can have any of the following settings:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TristateUseDefault</td>
<td>-2</td>
<td>Opens the file using the system default.</td>
</tr>
<tr>
<td>TristateTrue</td>
<td>-1</td>
<td>Opens the file as Unicode.</td>
</tr>
<tr>
<td>TristateFalse</td>
<td>0</td>
<td>Opens the file as ASCII.</td>
</tr>
</tbody>
</table>

Remarks

The **OpenAsTextStream** method provides the same functionality as the **OpenTextFile** method of the **FileSystemObject**. In addition, the **OpenAsTextStream** method can be used to write to a file.

The following code illustrates the use of the **OpenAsTextStream** method:

```vbs
Function TextStreamTest
    Const ForReading = 1, ForWriting = 2, ForAppending = 8
    Const TristateUseDefault = -2, TristateTrue = -1, TristateFalse = 0
    Dim fso, f, ts
    Set fso = CreateObject("Scripting.FileSystemObject")
    fso.CreateTextFile "test1.txt" ' Create a file
    Set f = fso.GetFile("test1.txt")
    Set ts = f.OpenAsTextStream(ForWriting, TristateUseDefault)
    ts.Write "Hello World"
    ts.Close
    Set ts = f.OpenAsTextStream(ForReading, TristateFalse)
End Function
```
TextStreamTest = ts.ReadLine
ts.Close
End Function
OpenTextFile Method

Description

Opens a specified file and returns a **TextStream** object that can be used to read from, write to, or append to the file.

Syntax

```vbs
object.OpenTextFile(filename[, iomode[, create[, format]]])
```

The **OpenTextFile** method has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>Required. Always the name of a <strong>FileSystemObject</strong>.</td>
</tr>
<tr>
<td>filename</td>
<td>Required. String expression that identifies the file to open.</td>
</tr>
<tr>
<td>iomode</td>
<td>Optional. Indicates input/output mode. Can be one of three constants: <strong>ForReading</strong>, <strong>ForWriting</strong>, or <strong>ForAppending</strong>.</td>
</tr>
<tr>
<td>create</td>
<td>Optional. Boolean value that indicates whether a new file can be created if the specified <strong>filename</strong> doesn't exist. The value is True if a new file is created; False if it isn't created. The default is False.</td>
</tr>
<tr>
<td>format</td>
<td>Optional. One of three Tristate values used to indicate the format of the opened file. If omitted, the file is opened as ASCII.</td>
</tr>
</tbody>
</table>

Settings

The **iomode** argument can have either of the following settings:
### Constant Value Description

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ForReading</td>
<td>1</td>
<td>Open a file for reading only. You can't write to this file.</td>
</tr>
<tr>
<td>ForWriting</td>
<td>2</td>
<td>Open a file for writing only. You can't read from this file.</td>
</tr>
<tr>
<td>ForAppending</td>
<td>8</td>
<td>Open a file and write to the end of the file.</td>
</tr>
</tbody>
</table>

The format argument can have any of the following settings:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TristateUseDefault</td>
<td>-2</td>
<td>Opens the file using the system default.</td>
</tr>
<tr>
<td>TristateTrue</td>
<td>-1</td>
<td>Opens the file as Unicode.</td>
</tr>
<tr>
<td>TristateFalse</td>
<td>0</td>
<td>Opens the file as ASCII.</td>
</tr>
</tbody>
</table>

**Remarks**

The following code illustrates the use of the `OpenTextFile` method to open a file for writing text:

```vba
Sub OpenTextFileTest
    Const ForReading = 1, ForWriting = 2, ForAppending = 8
    Dim fso, f
    Set fso = CreateObject("Scripting.FileSystemObject")
    Set f = fso.OpenTextFile("c:\testfile.txt", ForWriting, True)
    f.Write "Hello world!"
    f.Close
End Sub
```
ParentFolder Property

Description

Returns the folder object for the parent of the specified file or folder. Read-only.

Syntax

object.ParentFolder

The object is always a File or Folder object.

Remarks

The following code illustrates the use of the ParentFolder property with a file:

Function ShowFileAccessInfo(filespec)
    Dim fso, f, s
    Set fso = CreateObject("Scripting.FileSystemObject")
    Set f = fso.GetFile(filespec)
    s = UCase(f.Name) & " in " & UCase(f.ParentFolder) & "<BR>"
    s = s & "Created: " & f.DateCreated & "<BR>"
    s = s & "Last Accessed: " & f.DateLastAccessed & "<BR>"
    s = s & "Last Modified: " & f.DateLastModified
    ShowFileAccessInfo = s
End Function
Description

Returns the path for a specified file, folder, or drive.

Syntax

\texttt{object.Path}

The \texttt{object} is always a \texttt{File}, \texttt{Folder}, or \texttt{Drive} object.

Remarks

For drive letters, the root drive is not included. For example, the path for the C drive is C:, not C:.

The following code illustrates the use of the \texttt{Path} property with a \texttt{File} object:

\begin{verbatim}
Function ShowFileAccessInfo(filespec)
    Dim fso, d, f, s
    Set fso = CreateObject("Scripting.FileSystemObject")
    Set f = fso.GetFile(filespec)
    s = UCase(f.Path) & "<BR>"
    s = s & "Created: " & f.DateCreated & "<BR>"
    s = s & "Last Accessed: " & f.DateLastAccessed & "<BR>"
    s = s & "Last Modified: " & f.DateLastModified
    ShowFileAccessInfo = s
End Function
\end{verbatim}
Read Method

Description

Reads a specified number of characters from a TextStream file and returns the resulting string.

Syntax

```
object.Read(characters)
```

The Read method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>Required. Always the name of a TextStream object.</td>
</tr>
<tr>
<td>characters</td>
<td>Required. Number of characters you want to read from the file.</td>
</tr>
</tbody>
</table>

Remarks

The following example illustrates how to use the Read method to read five characters from a file and return the resulting string:

```
Function ReadTextFileTest
    Const ForReading = 1, ForWriting = 2, ForAppending = 8
    Dim fso, f, Msg
    Set fso = CreateObject("Scripting.FileSystemObject")
    Set f = fso.OpenTextFile("c:\testfile.txt", ForWriting)
    ' ... write to file...
    MsgBox fso.TextStream.Read(5)
End Function
```
f.Write "Hello world!"
Set f = fso.OpenTextFile("c:\testfile.txt", ForReading)
ReadTextFileTest = f.Read(5)
End Function
**ReadAll Method**

**Description**

Reads an entire **TextStream** file and returns the resulting string.

**Syntax**

```vbnet
object.ReadAll
```

The `object` is always the name of a **TextStream** object.

**Remarks**

For large files, using the **ReadAll** method wastes memory resources. Other techniques should be used to input a file, such as reading a file line by line.

```vbnet
Function ReadAllTextFile
    Const ForReading = 1, ForWriting = 2
    Dim fso, f
    Set fso = CreateObject("Scripting.FileSystemObject")
    Set f = fso.OpenTextFile("c:\testfile.txt", ForWriting, True)
    f.Write "Hello world!"
    Set f = fso.OpenTextFile("c:\testfile.txt", ForReading)
    ReadAllTextFile = f.ReadAll
End Function
```
ReadLine Method

See Also
Applies to

Description

Reads an entire line (up to, but not including, the newline character) from a TextStream file and returns the resulting string.

Syntax

```
object.ReadLine
```

The `object` argument is always the name of a TextStream object.

Remarks

The following example shows how to use the **ReadLine** method to read a line from a TextStream file and return the string:

```vbs
Function ReadLineTextFile
    Const ForReading = 1, ForWriting = 2
    Dim fso, MyFile
    Set fso = CreateObject("Scripting.FileSystemObject")
    Set MyFile = fso.OpenTextFile("c:\testfile.txt", ForWriting, True)
    MyFile.WriteLine "Hello world!"
    MyFile.WriteLine "The quick brown fox"
    MyFile.Close
    Set MyFile = fso.OpenTextFile("c:\testfile.txt", ForReading)
    ReadLineTextFile = MyFile.ReadLine ' Returns "Hello world!"
End Function
```
Remove Method

Description

Removes a key, item pair from a Dictionary object.

Syntax

\[ \text{object}.\text{Remove}(\text{key}) \]

The Remove method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>Required. Always the name of a Dictionary object.</td>
</tr>
<tr>
<td>key</td>
<td>Required. Key associated with the key, item pair you want to remove from the Dictionary object.</td>
</tr>
</tbody>
</table>

Remarks

An error occurs if the specified key, item pair does not exist.

The following code illustrates use of the Remove method:

```vbscript
Dim a, d
'd Create some variables.
Set d = CreateObject("Scripting.Dictionary")
d.Add "a", "Athens" ' Add some keys and items.
d.Add "b", "Belgrade"
d.Add "c", "Cairo"
...
```
d.Remove("b")       ' Remove second pair.
### Description

The **RemoveAll** method removes all key, item pairs from a **Dictionary** object.

### Syntax

```
object.RemoveAll
```

The `object` is always the name of a **Dictionary** object.

### Remarks

The following code illustrates use of the **RemoveAll** method:

```vbnet
Dim a, d, i
    ' Create some variables.
Set d = CreateObject("Scripting.Dictionary")
d.Add "a", "Athens"        ' Add some keys and items.
d.Add "b", "Belgrade"
d.Add "c", "Cairo"
...
a = d.RemoveAll            ' Clear the dictionary.
```
RootFolder Property

Description

Returns a **Folder** object representing the root folder of a specified drive. Read-only.

Syntax

```
object.RootFolder
```

The `object` is always a **Drive** object.

Remarks

All the files and folders contained on the drive can be accessed using the returned **Folder** object.

The following example illustrates the use of the **RootFolder** property:

```
Function ShowRootFolder(drvspec)
    Dim fso, f
    Set fso = CreateObject("Scripting.FileSystemObject")
    Set f = fso.GetDrive(drvspec)
    ShowRootFolder = f.RootFolder
End Function
```
SerialNumber Property

Description

Returns the decimal serial number used to uniquely identify a disk volume.

Syntax

```object.SerialNumber```

The `object` is always a `Drive` object.

Remarks

You can use the `SerialNumber` property to ensure that the correct disk is inserted in a drive with removable media.

The following code illustrates the use of the `SerialNumber` property:

```vbs
Function ShowDriveInfo(drvpath)
    Dim fso, d, s, t
    Set fso = CreateObject("Scripting.FileSystemObject")
    Set d = fso.GetDrive(fso.GetDriveName(fso.GetAbsolutePathName(drvpath)))
    Select Case d.DriveType
        Case 0: t = "Unknown"
        Case 1: t = "Removable"
        Case 2: t = "Fixed"
        Case 3: t = "Network"
        Case 4: t = "CD-ROM"
        Case 5: t = "RAM Disk"
    End Select
End Function
```
s = "Drive " & d.DriveLetter & ": - " & t
s = s & 
"<BR>" & 
"SN: " & 
 d.SerialNumber
ShowDriveInfo = s
End Function
ShareName Property

Description

Returns the network share name for a specified drive.

Syntax

\textit{object.ShareName}

The \textit{object} is always a \texttt{Drive} object.

Remarks

If \textit{object} is not a network drive, the \texttt{ShareName} property returns a zero-length string (""").

The following code illustrates the use of the \texttt{ShareName} property:

\begin{verbatim}
Function ShowDriveInfo(drvpath)
    Dim fso, d
    Set fso = CreateObject("Scripting.FileSystemObject")
    Set d = fso.GetDrive(fso.GetDriveName(fso.GetAbsolutePathName(drvpath)))
    ShowDriveInfo = "Drive " & d.DriveLetter & " - " & d.ShareName
End Function
\end{verbatim}
**ShortName Property**

**Description**

Returns the short name used by programs that require the earlier 8.3 naming convention.

**Syntax**

```vbnet
object.ShortName
```

The `object` is always a `File` or `Folder` object.

**Remarks**

The following code illustrates the use of the `ShortName` property with a `File` object:

```vbnet
Function ShowShortName(filespec)
    Dim fso, f, s
    Set fso = CreateObject("Scripting.FileSystemObject")
    Set f = fso.GetFile(filespec)
    s = "The short name for " & UCase(f.Name) & "<BR>"
    s = s & "is: " & f.ShortName
    ShowShortName = s
End Function
```
ShortPath Property

Description

Returns the short path used by programs that require the earlier 8.3 file naming convention.

Syntax

object.ShortPath

The object is always a File or Folder object.

Remarks

The following code illustrates the use of the ShortName property with a File object:

```
Function ShowShortPath(filespec)
    Dim fso, f, s
    Set fso = CreateObject("Scripting.FileSystemObject")
    Set f = fso.GetFile(filespec)
    s = "The short path for " & UCase(f.Name) & "<BR>"
    s = s & "is: " & f.ShortPath
    ShowShortPath = s
End Function
```
Property

**Description**

For files, returns the size, in bytes, of the specified file. For folders, returns the size, in bytes, of all files and subfolders contained in the folder.

**Syntax**

`object.Size`

The `object` is always a **File** or **Folder** object.

**Remarks**

The following code illustrates the use of the **Size** property with a **Folder** object:

```vbscript
Function ShowFolderSize(filespec)
    Dim fso, f, s
    Set fso = CreateObject("Scripting.FileSystemObject")
    Set f = fso.GetFolder(filespec)
    s = UCase(f.Name) & " uses " & f.Size & " bytes."
    ShowFolderSize = s
End Function
```
**Skip Method**

See Also  
Applies To

**Description**

Skips a specified number of characters when reading a TextStream file.

**Syntax**

```object.Skip(characters)```

The `Skip` method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>object</code></td>
<td>Required. Always the name of a TextStream object.</td>
</tr>
<tr>
<td><code>characters</code></td>
<td>Required. Number of characters to skip when reading a file.</td>
</tr>
</tbody>
</table>

**Remarks**

Skipped characters are discarded.

The following example uses the `Skip` method to skip the first six characters before reading from a text file:

```vbnet
Function SkipTextFile
    Const ForReading = 1, ForWriting = 2
    Dim fso, f
    Set fso = CreateObject("Scripting.FileSystemObject")
    Set f = fso.OpenTextFile("c:\testfile.txt", ForWriting)
    f.Write "Hello world!"
End Function
```
Set f = fso.OpenTextFile("c:\testfile.txt", ForReading)
f.Skip(6)
SkipTextFile = f.ReadLine
End Function
SkipLine Method

SkipLine Method

See Also

Applies to

Description

Skips the next line when reading a TextStream file.

Syntax

object.SkipLine

The object is always the name of a TextStream object.

Remarks

Skipping a line means reading and discarding all characters in a line up to and including the next newline character. An error occurs if the file is not open for reading.

The following example illustrates use of the SkipLine method:

Function SkipLineInFile
    Const ForReading = 1, ForWriting = 2
    Dim fso, f
    Set fso = CreateObject("Scripting.FileSystemObject")
    Set f = fso.OpenTextFile("c:\testfile.txt", ForWriting, True)
    f.Write "Hello world!" & vbCrLf & "VB Script is fun!"
    Set f = fso.OpenTextFile("c:\testfile.txt", ForReading)
    f.SkipLine
    SkipLineInFile = f.ReadLine
End Function
These constants are only available when your project has an explicit reference to the appropriate type library containing these constant definitions. For VBScript, you must explicitly declare these constants in your code.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WindowsFolder</td>
<td>0</td>
<td>The Windows folder contains files installed by the Windows operating system.</td>
</tr>
<tr>
<td>SystemFolder</td>
<td>1</td>
<td>The System folder contains libraries, fonts, and device drivers.</td>
</tr>
<tr>
<td>TemporaryFolder</td>
<td>2</td>
<td>The Temp folder is used to store temporary files. Its path is found in the TMP environment variable.</td>
</tr>
</tbody>
</table>
**SubFolders Property**

**Description**

Returns a **Folders** collection consisting of all folders contained in a specified folder, including those with Hidden and System file attributes set.

**Syntax**

```
object.SubFolders
```

The `object` is always a **Folder** object.

**Remarks**

The following code illustrates the use of the **SubFolders** property:

```vbs
Function ShowFolderList(folderspec)
    Dim fso, f, f1, s, sf
    Set fso = CreateObject("Scripting.FileSystemObject")
    Set f = fso.GetFolder(folderspec)
    Set sf = f.SubFolders
    For Each f1 in sf
        s = s & f1.name
        s = s & "<BR>"
    Next
    ShowFolderList = s
End Function
```
### TextStream Object

**See Also**

**Properties**

**Methods**

### Description

Facilitates sequential access to a file.

### Remarks

In the following code, `a` is the TextStream object returned by the `CreateTextFile` method on the FileSystemObject:

```vbs
Dim fso, MyFile
Set fso = CreateObject("Scripting.FileSystemObject")
Set MyFile = fso.CreateTextFile("c:\testfile.txt", True)
MyFile.WriteLine("This is a test.")
MyFile.Close
```

**WriteLine** and **Close** are two methods of the TextStream Object.
TotalSize Property

Description

Returns the total space, in bytes, of a drive or network share.

Syntax

object.TotalSize

The object is always a Drive object.

Remarks

The following code illustrates the use of the TotalSize property:

Function ShowSpaceInfo(drvpath)
    Dim fso, d, s
    Set fso = CreateObject("Scripting.FileSystemObject")
    Set d = fso.GetDrive(fso.GetDriveName(fso.GetAbsolutePathName(drvpath)))
    s = "Drive " & d.DriveLetter & ":"
    s = s & vbCrLf
    s = s & "Total Size: " & FormatNumber(d.TotalSize/1024, 0) & " Kbytes"
    s = s & vbCrLf
    s = s & "Available: " & FormatNumber(d.AvailableSpace/1024, 0) & " Kbytes"
    ShowSpaceInfo = s
End Function
Type Property

Description

Returns information about the type of a file or folder. For example, for files ending in .TXT, "Text Document" is returned.

Syntax

```
object.Type
```

The *object* is always a *File* or *Folder* object.

Remarks

The following code illustrates the use of the *Type* property to return a folder type. In this example, try providing the path of the Recycle Bin or other unique folder to the procedure.

```
Function ShowFolderType(filespec)
    Dim fso, f, s
    Set fso = CreateObject("Scripting.FileSystemObject")
    Set f = fso.GetFolder(filespec)
    s = UCase(f.Name) & " is a " & f.Type
    ShowFolderType = s
End Function
```
**VolumeName**

**Property**

**Description**

Sets or returns the volume name of the specified drive. Read/write.

**Syntax**

```object.VolumeName [ = newname]```

The `VolumeName` property has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>object</code></td>
<td>Required. Always the name of a <strong>Drive</strong> object.</td>
</tr>
<tr>
<td><code>newname</code></td>
<td>Optional. If provided, <code>newname</code> is the new name of the specified <code>object</code>.</td>
</tr>
</tbody>
</table>

**Remarks**

The following code illustrates the use of the `VolumeName` property:

```vbnet
Function ShowVolumeInfo(drvpath)
    Dim fso, d, s
    Set fso = CreateObject("Scripting.FileSystemObject")
    Set d = fso.GetDrive(fso.GetDriveName(fso.GetAbsolutePathName(drvpath)))
    s = "Drive " & d.DriveLetter & ": " & d.VolumeName
    ShowVolumeInfo = s
End Function
```
**Write Method**

**Description**

Writes a specified string to a **TextStream** file.

**Syntax**

```
object.Write(string)
```

The `Write` method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>Required. Always the name of a <strong>TextStream</strong> object.</td>
</tr>
<tr>
<td>string</td>
<td>Required. The text you want to write to the file.</td>
</tr>
</tbody>
</table>

**Remarks**

Specified strings are written to the file with no intervening spaces or characters between each string. Use the **WriteLine** method to write a newline character or a string that ends with a newline character.

The following example illustrates use of the `Write` method:

```
Function WriteToFile
    Const ForReading = 1, ForWriting = 2
    Dim fso, f
    Set fso = CreateObject("Scripting.FileSystemObject")
    Set f = fso.OpenTextFile("c:\testfile.txt", ForWriting)
    f.Write "Hello world!"
End Function
```
Set f = fso.OpenTextFile("c:\testfile.txt", ForReading)
WriteToFile =  f.ReadLine
End Function
WriteBlankLines Method

Description

Writes a specified number of newline characters to a TextStream file.

Syntax

\[
\textit{object}.\textbf{WriteBlankLines}(\textit{lines})
\]

The \textbf{WriteBlankLines} method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{object}</td>
<td>Required. Always the name of a \textbf{TextStream} object.</td>
</tr>
<tr>
<td>\textit{lines}</td>
<td>Required. Number of newline characters you want to write to the file.</td>
</tr>
</tbody>
</table>

Remarks

The following example illustrates use of the \textbf{WriteBlankLines} method:

```vbscript
Function WriteBlankLinesToFile
    Const ForReading = 1, ForWriting = 2
    Dim fso, f
    Set fso = CreateObject("Scripting.FileSystemObject")
    Set f = fso.OpenTextFile("c:\testfile.txt", ForWriting, True)
    f.WriteBlankLines 2
    f.WriteLine "Hello World!"
    Set f = fso.OpenTextFile("c:\testfile.txt", ForReading)
    WriteBlankLinesToFile = f.ReadAll
End Function
```
End Function
Description

Writes a specified string and newline character to a TextStream file.

Syntax

\[
\text{object}.\text{WriteLine}([\text{string}])
\]

The \text{WriteLine} method syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>Required. Always the name of a TextStream object.</td>
</tr>
<tr>
<td>string</td>
<td>Optional. The text you want to write to the file. If omitted, a newline character is written to the file.</td>
</tr>
</tbody>
</table>

Remarks

The following example illustrates use of the \text{WriteLine} method:

```vbs
Function WriteLineToFile
    Const ForReading = 1, ForWriting = 2
    Dim fso, f
    Set fso = CreateObject("Scripting.FileSystemObject")
    Set f = fso.OpenTextFile("c:\testfile.txt", ForWriting, True)
    f.WriteLine "Hello world!"
    f.WriteLine "VBScript is fun!"
    Set f = fso.OpenTextFile("c:\testfile.txt", ForReading)
    WriteLineToFile = f.ReadAll
End Function
```
Copyright

Microsoft® Visual Basic® Scripting Edition

Information in this document is subject to change without notice. The names of companies, products, people, characters, and/or data mentioned herein are fictitious and are in no way intended to represent any real individual, company, product, or event, unless otherwise noted. Complying with all applicable copyright laws is the responsibility of the user. No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without the express written permission of Microsoft Corporation.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

© 1991-1999 Microsoft Corporation. All rights reserved.

Microsoft, MS, MS-DOS, ActiveX, JScript, Microsoft Press, Visual Basic, Windows, Windows NT, Win32, and Win32s are either registered trademarks or trademarks of Microsoft Corporation in the U.S.A. and/or other countries.

Other product and company names mentioned herein may be the trademarks of their respective owners.
InStrRev Function

See Also

Description

Returns the position of an occurrence of one string within another, from the end of string.

Syntax

InStrRev(string1, string2[, start[, compare]])

The InStrRev function syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string1</td>
<td>Required. String expression being searched.</td>
</tr>
<tr>
<td>string2</td>
<td>Required. String expression being searched for.</td>
</tr>
<tr>
<td>start</td>
<td>Optional. Numeric expression that sets the starting position for each search. If omitted, -1 is used, which means that the search begins at the last character position. If start contains Null, an error occurs.</td>
</tr>
<tr>
<td>compare</td>
<td>Optional. Numeric value indicating the kind of comparison to use when evaluating substrings. If omitted, a binary comparison is performed. See Settings section for values.</td>
</tr>
</tbody>
</table>

Settings

The compare argument can have the following values:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vbBinaryCompare</td>
<td>0</td>
<td>Perform a binary comparison.</td>
</tr>
<tr>
<td>vbTextCompare</td>
<td>1</td>
<td>Perform a textual comparison.</td>
</tr>
</tbody>
</table>
Return Values

\textbf{InStrRev} returns the following values:

<table>
<thead>
<tr>
<th>If</th>
<th>InStrRev returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{string1} is zero-length</td>
<td>0</td>
</tr>
<tr>
<td>\textit{string1} is \textbf{Null}</td>
<td>\textbf{Null}</td>
</tr>
<tr>
<td>\textit{string2} is zero-length</td>
<td>\textit{start}</td>
</tr>
<tr>
<td>\textit{string2} is \textbf{Null}</td>
<td>\textbf{Null}</td>
</tr>
<tr>
<td>\textit{string2} is not found</td>
<td>0</td>
</tr>
<tr>
<td>\textit{string2} is found within \textit{string1}</td>
<td>Position at which match is found</td>
</tr>
<tr>
<td>start &gt; \textbf{Len}(\textit{string2})</td>
<td>0</td>
</tr>
</tbody>
</table>

Remarks

The following examples use the \textbf{InStrRev} function to search a string:

```
Dim SearchString, SearchChar, MyPos
SearchString = "XXpXXpXXPXXP"      ' String to s
SearchChar = "P"                   ' Search for "P".
MyPos = InstrRev(SearchString, SearchChar, 10, 0) ' A binary c
MyPos = InstrRev(SearchString, SearchChar, -1, 1) ' A textual c
    ' the last position. Returns
MyPos = InstrRev(SearchString, SearchChar, 8)   ' Comparison is |
    ' argument is omitted. Ret
```

\textbf{Note}  The syntax for the \textbf{InStrRev} function is not the same as the syntax for the \textbf{InStr} function.
Function

See Also

Sgn Function
ActiveX control
An object that you place on a form to enable or enhance a user's interaction with an application. ActiveX controls have events and can be incorporated into other controls. The controls have an .ocx file name extension.

ActiveX object
An object that is exposed to other applications or programming tools through Automation interfaces.

argument
A constant, variable, or expression passed to a procedure.

array
A set of sequentially indexed elements having the same type of data. Each element of an array has a unique identifying index number. Changes made to one element of an array do not affect the other elements.

ASCII Character Set
American Standard Code for Information Interchange (ASCII) 7-bit character set widely used to represent letters and symbols found on a standard U.S. keyboard. The ASCII character set is the same as the first 128 characters (0–127) in the ANSI character set.

Automation object
An object that is exposed to other applications or programming tools through Automation interfaces.

bitwise comparison
A bit-by-bit comparison of identically positioned bits in two numeric expressions.
**Boolean expression**
An expression that evaluates to either True or False.

---

**by reference**
A way of passing the address, rather than the value, of an argument to a procedure. This allows the procedure to access the actual variable. As a result, the variable's actual value can be changed by the procedure to which it is passed.

---

**by value**
A way of passing the value, rather than the address, of an argument to a procedure. This allows the procedure to access a copy of the variable. As a result, the variable's actual value can't be changed by the procedure to which it is passed.

---

**character code**
A number that represents a particular character in a set, such as the ASCII character set.

---

**class**
The formal definition of an object. The class acts as the template from which an instance of an object is created at run time. The class defines the properties of the object and the methods used to control the object's behavior.

---

**class module**
A module containing the definition of a class (its property and method definitions).

---

**collection**
An object that contains a set of related objects. An object's position in the collection can change whenever a change occurs in the collection; therefore, the position of any specific object in the collection may vary.

---

**comment**
Text added to code by a programmer that explains how the code works. In Visual Basic Scripting Edition, a comment line generally starts with an apostrophe ('), or you can use the keyword **Rem** followed by a space.
**comparison operator**

A character or symbol indicating a relationship between two or more values or expressions. These operators include less than (<), less than or equal to (<=), greater than (>), greater than or equal to (>=), not equal (<>), and equal (=).

Is is also a comparison operator, but it is used exclusively for determining if one object reference is the same as another.

---

**constant**

A named item that retains a constant value throughout the execution of a program. Constants can be used anywhere in your code in place of actual values. A constant can be a string or numeric literal, another constant, or any combination that includes arithmetic or logical operators except Is and exponentiation. For example:

```
Const A = "MyString"
```

---

**data ranges**

Each Variant subtype has a specific range of allowed values:

<table>
<thead>
<tr>
<th>Subtype</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte</td>
<td>0 to 255.</td>
</tr>
<tr>
<td>Boolean</td>
<td>True or False.</td>
</tr>
<tr>
<td>Integer</td>
<td>-32,768 to 32,767.</td>
</tr>
<tr>
<td>Long</td>
<td>-2,147,483,648 to 2,147,483,647.</td>
</tr>
<tr>
<td>Single</td>
<td>-3.402823E38 to -1.401298E-45 for negative values; 1.401298E-45 to 3.402823E38 for positive values.</td>
</tr>
<tr>
<td>Double</td>
<td>-1.79769313486232E308 to -4.94065645841247E-324 for negative values; 4.94065645841247E-324 to 1.79769313486232E308 for positive values.</td>
</tr>
<tr>
<td>Date</td>
<td>January 1, 100 to December 31, 9999, inclusive.</td>
</tr>
<tr>
<td>Object</td>
<td>Any Object reference.</td>
</tr>
</tbody>
</table>
Variable-length strings may range in length from 0 to approximately 2 billion characters.

Date Expression
Any expression that can be interpreted as a date. This includes any combination of date literals, numbers that look like dates, strings that look like dates, and dates returned from functions. A date expression is limited to numbers or strings, in any combination, that can represent a date from January 1, 100 through December 31, 9999.

Dates are stored as part of a real number. Values to the left of the decimal represent the date; values to the right of the decimal represent the time. Negative numbers represent dates prior to December 30, 1899.

date literal
Any sequence of characters with a valid format that is surrounded by number signs (#). Valid formats include the date format specified by the locale settings for your code or the universal date format. For example, #12/31/99# is the date literal that represents December 31, 1999, where English-U.S. is the locale setting for your application.

VBScript always interprets a date literal as US-ENGLISH if it is possible to do so. If a date literal cannot be interpreted as a date, an error occurs.

date separators
Characters used to separate the day, month, and year when date values are formatted.

Empty
A value that indicates that no beginning value has been assigned to a variable. Empty variables are 0 in a numeric context, or zero-length in a string context.

Error Number
A whole number in the range 0 to 65,535, inclusive, that corresponds to the Number property of the Err object. When combined with the Name property of the Err object, this number represents a particular error message.
A combination of keywords, operators, variables, and constants that yield a string, number, or object. An expression can perform a calculation, manipulate characters, or test data.

**intrinsic constant**
A constant provided by an application. Because you can't disable intrinsic constants, you can't create a user-defined constant with the same name.

**keyword**
A word or symbol recognized as part of the VBScript language; for example, a statement, function name, or operator.

**locale**
The set of information that corresponds to a given language and country/region. A locale affects the language of predefined programming terms and locale-specific settings. There are two contexts where locale information is important:

- The code locale affects the language of terms such as keywords and defines locale-specific settings such as the decimal and list separators, date formats, and character sorting order.
- The system locale affects the way locale-aware functionality behaves, for example, when you display numbers or convert strings to dates. You set the system locale using the Control Panel utilities provided by the operating system.

**Nothing**
The special value that indicates that an object variable is no longer associated with any actual object.

**Null**
A value indicating that a variable contains no valid data. **Null** is the result of:

- An explicit assignment of **Null** to a variable.
- Any operation between expressions that contain **Null**.

**numeric expression**
Any expression that can be evaluated as a number. Elements of the expression can include any combination of keywords, variables, constants, and operators that result in a number.
**object type**
A type of object exposed by an application, for example, Application, File, Range, and Sheet. Refer to the application's documentation (Microsoft Excel, Microsoft Project, Microsoft Word, and so on) for a complete listing of available objects.

---

**pi**
Pi is a mathematical constant equal to approximately 3.1415926535897932.

---

**Private**
Variables that are visible only to the script in which they are declared.

---

**procedure**
A named sequence of statements executed as a unit. For example, **Function** and **Sub** are types of procedures.

---

**procedure level**
Describes statements located within a **Function** or **Sub** procedure. Declarations are usually listed first, followed by assignments and other executable code. For example:

```vba
Sub MySub()    ' This statement declares a sub procedure block.
    Dim A    ' This statement starts the procedure block.
    A = "My variable"    ' Procedure-level code.
    Debug.Print A    ' Procedure-level code.
End Sub    ' This statement ends a sub procedure block.
```

Note that script-level code resides outside any procedure blocks.

---

**property**
A named attribute of an object. Properties define object characteristics such as size, color, and screen location, or the state of an object, such as enabled or disabled.
Public

Variables declared using the **Public** Statement are visible to all procedures in all modules in all applications.

---

**run time**

The time when code is running. During run time, you can't edit the code.

---

**run-time error**

An error that occurs when code is running. A run-time error results when a statement attempts an invalid operation.

---

**scope**

Defines the visibility of a variable, procedure, or object. For example, a variable declared as **Public** is visible to all procedures in all modules. Variables declared in procedures are visible only within the procedure and lose their value between calls.

---

**SCODE**

A long integer value that is used to pass detailed information to the caller of an interface member or API function. The status codes for OLE interfaces and APIs are defined in FACILITY_ITF.

---

**script level**

Any code outside a procedure is referred to as script-level code.

---

**seed**

An initial value used to generate pseudorandom numbers. For example, the **Randomize** statement creates a seed number used by the **Rnd** function to create unique pseudorandom number sequences.

---

**string comparison**

A comparison of two sequences of characters. Unless specified in the function making the comparison, all string comparisons are binary. In English, binary comparisons are case-sensitive; text comparisons are not.
string expression
Any expression that evaluates to a sequence of contiguous characters. Elements of a string expression can include a function that returns a string, a string literal, a string constant, or a string variable.

type library
A file or component within another file that contains standard descriptions of exposed objects, properties, and methods.

variable
A named storage location that can contain data that can be modified during program execution. Each variable has a name that uniquely identifies it within its level of scope.

Variable names:

- Must begin with an alphabetic character.
- Can't contain an embedded period or type-declaration character.
- Must be unique within the same scope.
- Must be no longer than 255 characters.
Operator

See Also

& Operator
- Operator
Arithmetic Operators
Concatenation Operators
Operator Precedence
Operator Summary
And Operator

See Also

Logical Operators
Not Operator
Operator Precedence
Operator Summary
Or Operator
Xor Operator
Array Function

See Also

Dim Statement
Asc

Function

See Also

Chr Function
Operator

See Also

Comparison Operators
Operator Precedence
Operator Summary
Set Statement
Atn Function

See Also

- Cos Function
- Derived Math Functions
- Sin Function
- Tan Function
CBool Function

See Also

CByte Function
CCur Function
CDate Function
CDbl Function
CInt Function
CLng Function
CSng Function
CStr Function
CByte Function

See Also

CBool Function
CCur Function
CDate Function
CDbl Function
CInt Function
CLng Function
CSng Function
CStr Function
CCur Function

See Also

CBool Function
CByte Function
CDate Function
CDbl Function
CInt Function
CLng Function
CSng Function
CStr Function
CDate Function

See Also

IsDate Function
**CDbl Function**

See Also

- CBool Function
- CByte Function
- CCur Function
- CDate Function
- CInt Function
- CLng Function
- CSng Function
- CStr Function
Chr

Function

See Also

Asc Function
CInt Function

See Also

CBool Function
CByte Function
CCur Function
CDate Function
CDbl Function
CLng Function
CSng Function
CStr Function
Int, Fix Functions
The following table lists the version of Microsoft Visual Basic Scripting Edition implemented by host applications.

<table>
<thead>
<tr>
<th>Host Application</th>
<th>VBScript Version</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1.0</td>
</tr>
<tr>
<td>Microsoft Internet Explorer 3.0</td>
<td></td>
</tr>
<tr>
<td>Microsoft Internet Information Server 3.0</td>
<td></td>
</tr>
<tr>
<td>Microsoft Internet Explorer 4.0</td>
<td></td>
</tr>
<tr>
<td>Microsoft Internet Information Server 4.0</td>
<td></td>
</tr>
<tr>
<td>Microsoft Windows Scripting Host 1.0</td>
<td></td>
</tr>
<tr>
<td>Microsoft Outlook 98</td>
<td></td>
</tr>
<tr>
<td>Microsoft Visual Studio 6.0</td>
<td></td>
</tr>
<tr>
<td>Microsoft Internet Explorer 5.0</td>
<td></td>
</tr>
<tr>
<td>Microsoft Internet Information Services 5.0</td>
<td></td>
</tr>
</tbody>
</table>

The following table lists VBScript language features and the version when first introduced.

<table>
<thead>
<tr>
<th>Language Element</th>
<th>Version First Introduced</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1.0</td>
</tr>
<tr>
<td>Abs Function</td>
<td>x</td>
</tr>
<tr>
<td>Addition Operator (+)</td>
<td>x</td>
</tr>
<tr>
<td>And Operator</td>
<td>x</td>
</tr>
<tr>
<td>Array Function</td>
<td></td>
</tr>
<tr>
<td>Asc Function</td>
<td>x</td>
</tr>
<tr>
<td>Term</td>
<td>x</td>
</tr>
<tr>
<td>-------------------------------------------</td>
<td>---</td>
</tr>
<tr>
<td>Assignment Operator (=)</td>
<td></td>
</tr>
<tr>
<td>Atn Function</td>
<td>x</td>
</tr>
<tr>
<td>Call Statement</td>
<td></td>
</tr>
<tr>
<td>CBool Function</td>
<td>x</td>
</tr>
<tr>
<td>CByte Function</td>
<td>x</td>
</tr>
<tr>
<td>CCur Function</td>
<td>x</td>
</tr>
<tr>
<td>CDate Function</td>
<td>x</td>
</tr>
<tr>
<td>CDbl Function</td>
<td>x</td>
</tr>
<tr>
<td>Chr Function</td>
<td>x</td>
</tr>
<tr>
<td>CInt Function</td>
<td>x</td>
</tr>
<tr>
<td>Class Object</td>
<td></td>
</tr>
<tr>
<td>Class Statement</td>
<td></td>
</tr>
<tr>
<td>Clear Method</td>
<td>x</td>
</tr>
<tr>
<td>CLng Function</td>
<td>x</td>
</tr>
<tr>
<td>Color Constants</td>
<td>x</td>
</tr>
<tr>
<td>Comparison Constants</td>
<td></td>
</tr>
<tr>
<td>Concatenation Operator (&amp;)</td>
<td>x</td>
</tr>
<tr>
<td>Const Statement</td>
<td>x</td>
</tr>
<tr>
<td>Cos Function</td>
<td>x</td>
</tr>
<tr>
<td>CreateObject Function</td>
<td>x</td>
</tr>
<tr>
<td>CSng Function</td>
<td>x</td>
</tr>
<tr>
<td>CStr Function</td>
<td>x</td>
</tr>
<tr>
<td>Date and Time Constants</td>
<td>x</td>
</tr>
<tr>
<td>Date Format Constants</td>
<td>x</td>
</tr>
<tr>
<td>Date Function</td>
<td>x</td>
</tr>
<tr>
<td>DateAdd Function</td>
<td>x</td>
</tr>
<tr>
<td>DateDiff Function</td>
<td>x</td>
</tr>
<tr>
<td>DatePart Function</td>
<td>x</td>
</tr>
<tr>
<td>DateSerial Function</td>
<td>x</td>
</tr>
<tr>
<td>DateValue Function</td>
<td>x</td>
</tr>
<tr>
<td>Day Function</td>
<td>x</td>
</tr>
<tr>
<td>----------------------------</td>
<td>---</td>
</tr>
<tr>
<td>Description Property</td>
<td>x</td>
</tr>
<tr>
<td>Dim Statement</td>
<td>x</td>
</tr>
<tr>
<td>Division Operator (/)</td>
<td>x</td>
</tr>
<tr>
<td>Do...Loop Statement</td>
<td>x</td>
</tr>
<tr>
<td>Empty</td>
<td>x</td>
</tr>
<tr>
<td>Eqv Operator</td>
<td>x</td>
</tr>
<tr>
<td>Erase Statement</td>
<td>x</td>
</tr>
<tr>
<td>Err Object</td>
<td>x</td>
</tr>
<tr>
<td>Eval Function</td>
<td>x</td>
</tr>
<tr>
<td>Execute Method</td>
<td>x</td>
</tr>
<tr>
<td>Execute Statement</td>
<td>x</td>
</tr>
<tr>
<td>ExecuteGlobal Statement</td>
<td>x</td>
</tr>
<tr>
<td>Exit Statement</td>
<td>x</td>
</tr>
<tr>
<td>Exp Function</td>
<td>x</td>
</tr>
<tr>
<td>Exponentiation Operator (^)</td>
<td>x</td>
</tr>
<tr>
<td>False</td>
<td>x</td>
</tr>
<tr>
<td>Filter Function</td>
<td>x</td>
</tr>
<tr>
<td>FirstIndex Property</td>
<td>x</td>
</tr>
<tr>
<td>Fix Function</td>
<td>x</td>
</tr>
<tr>
<td>For...Next Statement</td>
<td>x</td>
</tr>
<tr>
<td>For Each...Next Statement</td>
<td>x</td>
</tr>
<tr>
<td>FormatCurrency Function</td>
<td>x</td>
</tr>
<tr>
<td>FormatDateTime Function</td>
<td>x</td>
</tr>
<tr>
<td>FormatNumber Function</td>
<td>x</td>
</tr>
<tr>
<td>FormatPercent Function</td>
<td>x</td>
</tr>
<tr>
<td>Function Statement</td>
<td>x</td>
</tr>
<tr>
<td>GetLocale Function</td>
<td>x</td>
</tr>
<tr>
<td>GetObject Function</td>
<td>x</td>
</tr>
<tr>
<td>Function/Property</td>
<td></td>
</tr>
<tr>
<td>--------------------------</td>
<td>---</td>
</tr>
<tr>
<td>GetRef Function</td>
<td>x</td>
</tr>
<tr>
<td>Global Property</td>
<td>x</td>
</tr>
<tr>
<td>Hex Function</td>
<td>x</td>
</tr>
<tr>
<td>HelpContext Property</td>
<td>x</td>
</tr>
<tr>
<td>HelpFile Property</td>
<td>x</td>
</tr>
<tr>
<td>Hour Function</td>
<td>x</td>
</tr>
<tr>
<td>If...Then...Else Statement</td>
<td>x</td>
</tr>
<tr>
<td>IgnoreCase Property</td>
<td>x</td>
</tr>
<tr>
<td>Imp Operator</td>
<td>x</td>
</tr>
<tr>
<td>Initialize Event</td>
<td>x</td>
</tr>
<tr>
<td>InputBox Function</td>
<td>x</td>
</tr>
<tr>
<td>InStr Function</td>
<td>x</td>
</tr>
<tr>
<td>InStrRev Function</td>
<td>x</td>
</tr>
<tr>
<td>Int Function</td>
<td>x</td>
</tr>
<tr>
<td>Integer Division Operator</td>
<td>x</td>
</tr>
<tr>
<td>Is Operator</td>
<td>x</td>
</tr>
<tr>
<td>IsArray Function</td>
<td>x</td>
</tr>
<tr>
<td>IsDate Function</td>
<td>x</td>
</tr>
<tr>
<td>IsEmpty Function</td>
<td>x</td>
</tr>
<tr>
<td>IsNull Function</td>
<td>x</td>
</tr>
<tr>
<td>IsNumeric Function</td>
<td>x</td>
</tr>
<tr>
<td>IsObject Function</td>
<td>x</td>
</tr>
<tr>
<td>Join Function</td>
<td>x</td>
</tr>
<tr>
<td>LBound Function</td>
<td>x</td>
</tr>
<tr>
<td>LCase Function</td>
<td>x</td>
</tr>
<tr>
<td>Left Function</td>
<td>x</td>
</tr>
<tr>
<td>Len Function</td>
<td>x</td>
</tr>
<tr>
<td>Length Property</td>
<td>x</td>
</tr>
<tr>
<td>LoadPicture Function</td>
<td>x</td>
</tr>
<tr>
<td>Function/Property</td>
<td></td>
</tr>
<tr>
<td>--------------------------------------</td>
<td></td>
</tr>
<tr>
<td>Log Function</td>
<td></td>
</tr>
<tr>
<td>LTrim Function</td>
<td></td>
</tr>
<tr>
<td>Match Object</td>
<td></td>
</tr>
<tr>
<td>Matches Collection</td>
<td></td>
</tr>
<tr>
<td>Mid Function</td>
<td></td>
</tr>
<tr>
<td>Minute Function</td>
<td></td>
</tr>
<tr>
<td>Miscellaneous Constants</td>
<td></td>
</tr>
<tr>
<td>Mod Operator</td>
<td></td>
</tr>
<tr>
<td>Month Function</td>
<td></td>
</tr>
<tr>
<td>MonthName Function</td>
<td></td>
</tr>
<tr>
<td>MsgBox Constants</td>
<td></td>
</tr>
<tr>
<td>MsgBox Function</td>
<td></td>
</tr>
<tr>
<td>Multiplication Operator (*)</td>
<td></td>
</tr>
<tr>
<td>Negation Operator (-)</td>
<td></td>
</tr>
<tr>
<td>Not Operator</td>
<td></td>
</tr>
<tr>
<td>Now Function</td>
<td></td>
</tr>
<tr>
<td>Nothing</td>
<td></td>
</tr>
<tr>
<td>Null</td>
<td></td>
</tr>
<tr>
<td>Number Property</td>
<td></td>
</tr>
<tr>
<td>Oct Function</td>
<td></td>
</tr>
<tr>
<td>On Error Statement</td>
<td></td>
</tr>
<tr>
<td>Option Explicit Statement</td>
<td></td>
</tr>
<tr>
<td>Or Operator</td>
<td></td>
</tr>
<tr>
<td>Pattern Property</td>
<td></td>
</tr>
<tr>
<td>Private Statement</td>
<td></td>
</tr>
<tr>
<td>PropertyGet Statement</td>
<td></td>
</tr>
<tr>
<td>PropertyLet Statement</td>
<td></td>
</tr>
<tr>
<td>PropertySet Statement</td>
<td></td>
</tr>
<tr>
<td>Public Statement</td>
<td></td>
</tr>
<tr>
<td>Method/Statement</td>
<td>x</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>---</td>
</tr>
<tr>
<td>Raise Method</td>
<td></td>
</tr>
<tr>
<td>Randomize Statement</td>
<td></td>
</tr>
<tr>
<td>ReDim Statement</td>
<td></td>
</tr>
<tr>
<td>RegExp Object</td>
<td></td>
</tr>
<tr>
<td>Rem Statement</td>
<td></td>
</tr>
<tr>
<td>Replace Function</td>
<td></td>
</tr>
<tr>
<td>Replace Method</td>
<td></td>
</tr>
<tr>
<td>RGB Function</td>
<td></td>
</tr>
<tr>
<td>Right Function</td>
<td></td>
</tr>
<tr>
<td>Rnd Function</td>
<td></td>
</tr>
<tr>
<td>Round Function</td>
<td></td>
</tr>
<tr>
<td>RTrim Function</td>
<td></td>
</tr>
<tr>
<td>ScriptEngine Function</td>
<td></td>
</tr>
<tr>
<td>ScriptEngineBuildVersion Function</td>
<td></td>
</tr>
<tr>
<td>ScriptEngineMajorVersion Function</td>
<td></td>
</tr>
<tr>
<td>ScriptEngineMinorVersion Function</td>
<td>x</td>
</tr>
<tr>
<td>Second Function</td>
<td></td>
</tr>
<tr>
<td>Select Case Statement</td>
<td></td>
</tr>
<tr>
<td>Set Statement</td>
<td></td>
</tr>
<tr>
<td>SetLocale Function</td>
<td></td>
</tr>
<tr>
<td>Sgn Function</td>
<td></td>
</tr>
<tr>
<td>Sin Function</td>
<td></td>
</tr>
<tr>
<td>Source Property</td>
<td></td>
</tr>
<tr>
<td>Space Function</td>
<td></td>
</tr>
<tr>
<td>Split Function</td>
<td></td>
</tr>
<tr>
<td>Sqr Function</td>
<td></td>
</tr>
<tr>
<td>StrComp Function</td>
<td></td>
</tr>
<tr>
<td>String Constants</td>
<td>x</td>
</tr>
<tr>
<td>-----------------------</td>
<td>---</td>
</tr>
<tr>
<td>String Function</td>
<td>x</td>
</tr>
<tr>
<td>StrReverse Function</td>
<td>x</td>
</tr>
<tr>
<td>Sub Statement</td>
<td>x</td>
</tr>
<tr>
<td>Subtraction Operator (-)</td>
<td>x</td>
</tr>
<tr>
<td>Tan Function</td>
<td>x</td>
</tr>
<tr>
<td>Terminate Event</td>
<td>x</td>
</tr>
<tr>
<td>Test Method</td>
<td>x</td>
</tr>
<tr>
<td>Time Function</td>
<td>x</td>
</tr>
<tr>
<td>Timer Function</td>
<td>x</td>
</tr>
<tr>
<td>TimeSerial Function</td>
<td>x</td>
</tr>
<tr>
<td>TimeValue Function</td>
<td>x</td>
</tr>
<tr>
<td>Trim Function</td>
<td>x</td>
</tr>
<tr>
<td>Tristate Constants</td>
<td>x</td>
</tr>
<tr>
<td>True</td>
<td>x</td>
</tr>
<tr>
<td>TypeName Function</td>
<td>x</td>
</tr>
<tr>
<td>UBound Function</td>
<td>x</td>
</tr>
<tr>
<td>UCase Function</td>
<td>x</td>
</tr>
<tr>
<td>Value Property</td>
<td>x</td>
</tr>
<tr>
<td>VarType Constants</td>
<td>x</td>
</tr>
<tr>
<td>VarType Function</td>
<td>x</td>
</tr>
<tr>
<td>VBScript Constants</td>
<td>x</td>
</tr>
<tr>
<td>Weekday Function</td>
<td>x</td>
</tr>
<tr>
<td>WeekdayName Function</td>
<td>x</td>
</tr>
<tr>
<td>While...Wend Statement</td>
<td>x</td>
</tr>
<tr>
<td>With Statement</td>
<td>x</td>
</tr>
<tr>
<td>Xor Operator</td>
<td>x</td>
</tr>
<tr>
<td>Year Function</td>
<td>x</td>
</tr>
</tbody>
</table>
Class

Object

Events

Initialize Event
Terminate Event
Class

Statement

See Also

Dim Statement
Function Statement
Private Statement
Property Get Statement
Property Let Statement
Property Set Statement
Public Statement
Set Statement
Sub Statement
Clear Method

See Also

Description Property
Err Object
Number Property
On Error Statement
Raise Method
Source Property
Clear

Method

Applies To

Err Object
CLng Function

See Also

CBool Function
CByte Function
CCur Function
CDate Function
CDbl Function
CInt Function
CSng Function
CStr Function
Int, Fix Functions
Comparison Constants

See Also

Color Constants
Date and Time Constants
Date Format Constants
Miscellaneous Constants
MsgBox Constants
String Constants
Tristate Constants
VarType Constants
Operator

See Also

Concatenation Operators
Operator Precedence
Operator Summary
Const Statement

See Also

Dim Statement
Function Statement
Private Statement
Public Statement
Sub Statement
Microsoft® Visual Basic® Scripting Edition

**Cos**

Function

See Also

- Atn Function
- Derived Math Functions
- Sin Function
- Tan Function
CreateObject Function

See Also

GetObject Function
CSng

Function

See Also

CBool Function
CByte Function
CCur Function
CDate Function
CDb1 Function
CInt Function
CLng Function
CStr Function

Language Reference
CStr Function

See Also

CBool Function
CByte Function
CCur Function
CDate Function
CDb1 Function
CInt Function
CLng Function
CSng Function
Date and Time Constants

See Also

- Color Constants
- Comparison Constants
- Date Format Constants
- Miscellaneous Constants
- MsgBox Constants
- String Constants
- Tristate Constants
- VarType Constants
Date Format Constants
See Also

Color Constants
Comparison Constants
Date and Time Constants
Miscellaneous Constants
MsgBox Constants
String Constants
Tristate Constants
VarType Constants
Date

Function

See Also

CDate Function
Now Function
Time Function
DateAdd Function

See Also

DateDiff Function
DatePart Function
DateDiff Function

See Also

DateAdd Function
DatePart Function
DatePart Function

See Also

DateAdd Function
DateDiff Function
DateSerial Function

See Also

Date Function
DateValue Function
Day Function
Month Function
Now Function
TimeSerial Function
TimeValue Function
Weekday Function
Year Function
DateValue Function

See Also

CDate Function
DateSerial Function
Day Function
Month Function
Now Function
TimeSerial Function
TimeValue Function
Weekday Function
Year Function
Day Function

See Also

Date Function
Hour Function
Minute Function
Month Function
Now Function
Second Function
Weekday Function
Year Function
<table>
<thead>
<tr>
<th>Description</th>
<th>Property</th>
<th>Applies To</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td><strong>Err Object</strong></td>
</tr>
</tbody>
</table>
Dim

Statement

See Also

Private Statement
Public Statement
ReDim Statement
Set Statement
Operator

See Also

* Operator
\ Operator
Arithmetic Operators
Operator Precedence
Operator Summary
Do...Loop Statement

See Also

Exit Statement
For...Next Statement
While...Wend Statement
Eqv Operator

See Also

Imp Operator
Logical Operators
Operator Precedence
Operator Summary
Erase Statement

See Also

Dim Statement
Nothing
ReDim Statement
Err

Object

See Also

Error Messages
On Error Statement
Err

Object

Properties

- Description Property
- HelpContext Property
- HelpFile Property
- Number Property
- Source Property
Object

Methods

Clear Method
Raise Method
Eval Function

See Also

Execute Statement
Execute Method

See Also

Replace Method
Test Method
Execute Method
Applies To

RegExp Object
Execute
Statement
See Also

Eval Function
ExecuteGlobal Statement
Statement

See Also

Do...Loop Statement
For Each...Next Statement
For...Next Statement
Function Statement
Sub Statement
Exp Function

See Also

Derived Math Functions
Log Function
Operator

See Also

Arithmetic Operators
Operator Precedence
Operator Summary
Filter

Function

See Also

Replace Function
FirstIndex Property

See Also

Length Property
Value Property
FirstIndex
Property
Applies To

Match Object
**Int, Fix**

**Functions**

*See Also*

- **CInt Function**
- **Round Function**

[Language Reference](#)
For...Next Statement

See Also

Do...Loop Statement
Exit Statement
For Each...Next Statement
While...Wend Statement
For Each...Next Statement

See Also

Do...Loop Statement
Exit Statement
For...Next Statement
While...Wend Statement
FormatCurrency Function

See Also

FormatDateTime Function
FormatNumber Function
FormatPercent Function
FormatDateTime Function

See Also

FormatCurrency Function
FormatNumber Function
FormatPercent Function
FormatNumber Function

See Also

FormatCurrency Function
FormatDateTime Function
FormatPercent Function
FormatPercent Function

See Also

FormatCurrency Function
FormatDateTime Function
FormatNumber Function
Function Statement

See Also

Call Statement
Dim Statement
Exit Statement
Nothing
Set Statement
Sub Statement
GetObject Function

See Also

CreateObject Function
GetRef

Function

See Also

Function Statement
Set Statement
Sub Statement
Global Property

See Also

IgnoreCase Property
Pattern Property
Global

Property

Applies To

RegExp Object
Hex Function

See Also

Oct Function
HelpContext Property

See Also

Description Property
HelpFile Property
Number Property
Source Property
HelpContext Property
Applies To

Err Object
HelpFile Property

See Also

Description Property
HelpContext Property
Number Property
Source Property
HelpFile Property
Applies To

Err Object
Hour Function

See Also

Day Function
Minute Function
Now Function
Second Function
Time Function
IgnoreCase Property

See Also

Global Property
Pattern Property
IgnoreCase Property
Applies To

RegExp Object
Operator

See Also

Eqv Operator
Logical Operators
Operator Precedence
Operator Summary
Initialize Event

See Also

Class Object
Class Statement
Terminate Event
Initialize Event

Applies To

Class Object
InputBox Function

See Also

MsgBox Function
InStr Function

See Also

InStrRev Function
Operator

See Also

* Operator
/ Operator
Arithmetic Operators
Operator Precedence
Operator Summary
Is

Operator

See Also

Comparison Operators
Operator Precedence
Operator Summary
IsArray Function

See Also

IsDate Function
IsEmpty Function
IsNull Function
IsNumeric Function
IsObject Function
VarType Function
IsDate

**Function**

**See Also**

- CDate Function
- IsArray Function
- IsEmpty Function
- IsNull Function
- IsNumeric Function
- IsObject Function
- VarType Function
IsEmpty Function

See Also

IsArray Function
IsDate Function
IsNull Function
IsNumeric Function
IsObject Function
VarType Function
IsNull Function

See Also

IsArray Function
IsDate Function
IsEmpty Function
IsNumeric Function
IsObject Function
VarType Function
IsNumeric Function

See Also

IsArray Function
IsDate Function
IsEmpty Function
IsNull Function
IsObject Function
VarType Function
**IsObject Function**

See Also

- *IsArray Function*
- *IsDate Function*
- *IsEmpty Function*
- *IsNull Function*
- *IsNumeric Function*
- *Set Statement*
- *VarType Function*
Join Function

See Also

Split Function
LBound

Function

See Also

Dim Statement
ReDim Statement
UBound Function
LCase Function

See Also

UCase Function
Left Function

See Also

Len Function
Mid Function
Right Function
Len Function

See Also

InStr Function
Length

Property

See Also

FirstIndex Property
Value Property
Length

Property

Applies To

Match Object
Log Function

See Also

Derived Math Functions
Exp Function
LTrim, RTrim, and Trim Functions

See Also

Left Function
Right Function
Match Object

See Also

Matches Collection
RegExp Object
Object Properties

FirstIndex Property
Length Property
Value Property
 Matches

Collection

See Also

For Each...Next Statement
Match Object
RegExp Object
Microsoft® Visual Basic® Scripting Edition Matches

Collection Properties

Count Property
Item Property

Language Reference
Mid Function

See Also

Left Function
Len Function
LTrim, RTrim, and Trim Functions
Right Function
Minute Function

See Also

Day Function
Hour Function
Now Function
Second Function
Time Function
Miscellaneous Constants

See Also

Color Constants
Comparison Constants
Date and Time Constants
Date Format Constants
MsgBox Constants
String Constants
Tristate Constants
VarType Constants
Mod Operator

See Also

Arithmetic Operators
Operator Precedence
Operator Summary
Month Function

See Also

Date Function
Day Function
Now Function
Weekday Function
Year Function
MonthName Function

See Also

WeekdayName Function
Constants

See Also

Color Constants
Comparison Constants
Date and Time Constants
Date Format Constants
Miscellaneous Constants
String Constants
Tristate Constants
VarType Constants
MsgBox Function

See Also

InputBox Function
Operator

See Also

\ Operator
Arithmetic Operators
Operator Precedence
Operator Summary
Operator

See Also

+ Operator
Arithmetic Operators
Operator Precedence
Operator Summary

Language Reference
Not Operator

See Also

And Operator
Logical Operators
Operator Precedence
Operator Summary
Or Operator
Xor Operator
Function

See Also

Date Function
Day Function
Hour Function
Minute Function
Month Function
Second Function
Time Function
Weekday Function
Year Function
See Also

Dim Statement
Set Statement
Number Property

See Also

Description Property
HelpContext Property
HelpFile Property
Err Object
Source Property
Error Messages
### Property

**Number**

**Applies To**

- **Err Object**
Oct

Function

See Also

Hex Function
On Error Statement

See Also

Err Object
Exit Statement
Operator Precedence

See Also

- Is Operator
- Operator Summary
Or

Operator

See Also

And Operator
Logical Operators
Not Operator
Operator Precedence
Operator Summary
Xor Operator
Pattern Property

See Also

Global Property
IgnoreCase Property
Pattern

Property

Applies To

RegExp Object
Private Statement

See Also

Dim Statement
Public Statement
ReDim Statement
Set Statement
Property Get Statement

See Also

Class Statement
Dim Statement
Exit Statement
Function Statement
Private Statement
Property Let Statement
Property Set Statement
Public Statement
Set Statement
Sub Statement
Property Let Statement

See Also

Class Statement
Dim Statement
Exit Statement
Function Statement
Private Statement
Property Get Statement
Property Set Statement
Public Statement
Set Statement
Sub Statement
Property Set Statement

See Also

Class Statement
Dim Statement
Exit Statement
Function Statement
Private Statement
Property Get Statement
Property Let Statement
Public Statement
Set Statement
Sub Statement

Language Reference
Public

Statement

See Also

Dim Statement
Private Statement
ReDim Statement
Set Statement
Microsoft® Visual Basic® Scripting Edition

Raise

Method

See Also

Clear Method
Description Property
Err Object
Number Property
Source Property

Language Reference
Raise Method
Applies To
Err Object
Randomize Statement

See Also

Rnd Function
Timer Function
ReDim Statement

See Also

Dim Statement
Set Statement
See Also

- Match Object
- Matches Collection
Object
Methods

Execute Method
Replace Method
Test Method
RegExp Object

Properties

- Global Property
- IgnoreCase Property
- Pattern Property
Replace

Function

See Also

Filter Function
Replace Method
See Also

Execute Method
Test Method
Replace Method Applies To

RegExp Object
Right Function

See Also

Left Function
Len Function
Mid Function

Language Reference
Function

See Also

Randomize Statement

Language Reference
Round Function

See Also

Int, Fix Functions
ScriptEngine Function

See Also

- ScriptEngineBuildVersion Function
- ScriptEngineMajorVersion Function
- ScriptEngineMinorVersion Function

Language Reference
ScriptEngineBuildVersion Function

See Also

ScriptEngine Function
ScriptEngineMajorVersion Function
ScriptEngineMinorVersion Function

Language Reference
ScriptEngineMajorVersion Function

See Also

ScriptEngine Function
ScriptEngineBuildVersion Function
ScriptEngineMinorVersion Function
ScriptEngineMinorVersion Function

See Also

ScriptEngine Function
ScriptEngineBuildVersion Function
ScriptEngineMajorVersion Function
Second Function

See Also

Day Function
Hour Function
Minute Function
Now Function
Time Function
Select Case Statement

See Also

If...Then...Else Statement
Set Statement

See Also

= Operator
Dim Statement
GetRef Function
ReDim Statement
Function

See Also

Abs Function
Function

See Also

Atn Function
Cos Function
Derived Math Functions
Tan Function
See Also

Description Property
Err Object
HelpContext Property
HelpFile Property
Number Property
On Error Statement
### Property Applies To

<table>
<thead>
<tr>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Err Object</strong></td>
</tr>
</tbody>
</table>
Function

See Also

String Function
Split Function

See Also

Join Function
String Constants

See Also

- Color Constants
- Comparison Constants
- Date and Time Constants
- Date Format Constants
- Miscellaneous Constants
- MsgBox Constants
- Tristate Constants
- VarType Constants
String

Function

See Also

Space Function

Language Reference
Sub

Statement

See Also

Call Statement
Dim Statement
Exit Statement
Function Statement
Tan Function

See Also

- Atn Function
- Cos Function
- Derived Math Functions
- Sin Function
Terminate Event

See Also

Class Object
Class Statement
Initialize Event
Terminate Event

Applies To

Class Object
Test Method

See Also

Execute Method
Replace Method
Test

Method

Applies To

RegExp Object

Language Reference
Time

Function

See Also

Date Function
Timer Function

See Also

Randomize Statement
TimeSerial Function

See Also

DateSerial Function
DateValue Function
Hour Function
Minute Function
Now Function
Second Function
TimeValue Function
TimeValue Function

See Also

DateSerial Function
DateValue Function
Hour Function
Minute Function
Now Function
Second Function
TimeSerial Function
TypeName
Function

See Also

IsArray Function
IsDate Function
IsEmpty Function
IsNull Function
IsNumeric Function
IsObject Function
VarType Function
UBound Function

See Also

Dim Statement
LBound Function
ReDim Statement
UCase Function

See Also

LCase Function
Value

Property

See Also

FirstIndex Property
Length Property
Value

Property

Applies To

Match Object
VarType Constants

See Also

Color Constants
Comparison Constants
Date and Time Constants
Date Format Constants
Miscellaneous Constants
MsgBox Constants
String Constants
Tristate Constants
VarType Function

See Also

IsArray Function
IsDate Function
IsEmpty Function
IsNull Function
IsNumeric Function
IsObject Function
TypeName Function
VBScript Constants

See Also

FileSystemObject Constants
Weekday Function

See Also

Date Function
Day Function
Month Function
Now Function
Year Function
While...Wend Statement

See Also

Do...Loop Statement
With Statement

See Also

Set Statement
Xor Operator

See Also

And Operator
Logical Operators
Not Operator
Operator Precedence
Operator Summary
Or Operator
Year

Function

See Also

Date Function
Day Function
Month Function
Now Function
Weekday Function
Color Constants

See Also

Comparison Constants
Date and Time Constants
Date Format Constants
Miscellaneous Constants
MsgBox Constants
String Constants
Tristate Constants
VarType Constants
Tristate Constants

See Also

- Color Constants
- Comparison Constants
- Date and Time Constants
- Date Format Constants
- Miscellaneous Constants
- MsgBox Constants
- String Constants
- VarType Constants
Comparison Operators

See Also

= Operator
Is Operator
Operator Precedence
Operator Summary
**ExecuteGlobal Statement**

**See Also**

**Description**

Executes one or more specified statements in the global namespace of a script.

**Syntax**

```
ExecuteGlobal statement
```

The required `statement` argument is a `string expression` containing one or more statements for execution. Include multiple statements in the `statement` argument, using colons or embedded line breaks to separate them.

**Remarks**

In VBScript, `x = y` can be interpreted two ways. The first is as an assignment statement, where the value of `y` is assigned to `x`. The second interpretation is as an `expression` that tests if `x` and `y` have the same value. If they do, `result` is `True`; if they are not, `result` is `False`. The `ExecuteGlobal` statement always uses the first interpretation, whereas the `Eval` method always uses the second.

**Note** In Microsoft® JScript®, no confusion exists between assignment and comparison, because the assignment operator (=) is different from the `comparison operator(==)`.

All statements used with `ExecuteGlobal` are executed in the script's global namespace. This allows code to be added to the program so that any `procedure` can access it. For example, a VBScript `Class` statement can be executed at run time and functions can subsequently create new instances of the class.

Adding procedures and classes at runtime can be useful, but also introduces the possibility of overwriting existing global variables and functions at runtime. Because this can cause significant programming problems, care should be exercised when using the `ExecuteGlobal` statement. If you don't need access to a variable or function outside of a procedure, use the `Execute` statement which
will only affect the namespace of the calling function.

The following example illustrates the use of the **ExecuteGlobal** statement:

```vba
Dim X          ' Declare X in global scope.
X = "Global"   ' Assign global X a value.
Sub Proc1      ' Declare procedure.
    Dim X      ' Declare X in local scope.
    X = "Local" ' Assign local X a value.
    ' The Execute statement here creates a
    ' procedure that, when invoked, prints X.
    ' It print the global X because Proc2
    ' inherits everything in global scope.
    ExecuteGlobal "Sub Proc2: Print X: End Sub"
Print Eval("X")  ' Print local X.
Proc2           ' Invoke Proc2 in Global scope resulting
                ' in "Global" being printed.
End Sub
Proc2           ' This line causes an error since
                ' Proc2 is unavailable outside Proc1.
Proc1           ' Invoke Proc1.
    Execute "Sub Proc2: Print X: End Sub"
Proc2           ' This invocation succeeds because Proc2
                ' is now available globally.
```

The following example shows how the **ExecuteGlobal** statement can be
rewritten so you don't have to enclose the entire procedure in the quotation
marks:

```vba
S = "Sub Proc2" & vbCrLf
S = S & " Print X" & vbCrLf
S = S & "End Sub"
ExecuteGlobal S
```
GetLocale Function

Description

Returns the current locale ID value.

Syntax

GetLocale()

Remarks

A locale is a set of user preference information related to the user's language, country/region, and cultural conventions. The locale determines such things as keyboard layout, alphabetic sort order, as well as date, time, number, and currency formats.

The return value can be any of the 32-bit values shown in the Locale ID chart:

The following example illustrates the use of the GetLocale function. To use this code, paste the entire example between the <BODY> tags of a standard HTML page.

Enter Date in UK format: <input type="text" id="UKDate" size=""; Here's the US equivalent: <input type="text" id="USdate" size="2" <input type="button" value="Convert" id="button1"><p> Enter a price in German: &nbsp; <input type="text" id="GermanNumber" size=""; Here's the UK equivalent: <input type="text" id="USNumber" size="" <input type="button" value="Convert" id="button2"><p>

<script language="vbscript">
Dim currentLocale
' Get the current locale
currentLocale = GetLocale

Sub Button1_onclick
    Dim original
    original = SetLocale("en-gb")
    mydate = CDate(UKDate.value)
    ' IE always sets the locale to US English so use the
    ' currentLocale variable to set the locale to US English
    original = SetLocale(currentLocale)
    USDate.value = FormatDateTime(mydate,vbShortDate)
End Sub

Sub button2_onclick
    Dim original
    original = SetLocale("de")
    myvalue = CCur(GermanNumber.value)
    original = SetLocale("en-gb")
    USNumber.value = FormatCurrency(myvalue)
End Sub

</script>
<table>
<thead>
<tr>
<th>Locale Description</th>
<th>Short String</th>
<th>Hex Value</th>
<th>Decimal Value</th>
<th>Locale Description</th>
<th>Short String</th>
<th>Hex Value</th>
<th>Decimal Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Afrikaans</td>
<td>af</td>
<td>0x0436</td>
<td>1078</td>
<td>Hebrew</td>
<td>he</td>
<td>0x040D</td>
<td>1037</td>
</tr>
<tr>
<td>Albanian</td>
<td>sq</td>
<td>0x041C</td>
<td>1052</td>
<td>Hindi</td>
<td>hi</td>
<td>0x0439</td>
<td>1081</td>
</tr>
<tr>
<td>Arabic - U.A.E.</td>
<td>ar-ae</td>
<td>0x3801</td>
<td>14337</td>
<td>Hungarian</td>
<td>hu</td>
<td>0x040E</td>
<td>1038</td>
</tr>
<tr>
<td>Arabic - Bahrain</td>
<td>ar-bh</td>
<td>0x3C01</td>
<td>15361</td>
<td>Icelandic</td>
<td>is</td>
<td>0x040F</td>
<td>1039</td>
</tr>
<tr>
<td>Arabic - Algeria</td>
<td>ar-dz</td>
<td>0x1401</td>
<td>5121</td>
<td>Indonesian</td>
<td>in</td>
<td>0x0421</td>
<td>1057</td>
</tr>
<tr>
<td>-----------------</td>
<td>-------</td>
<td>-------</td>
<td>------</td>
<td>------------</td>
<td>---</td>
<td>-------</td>
<td>------</td>
</tr>
<tr>
<td>Arabic - Egypt</td>
<td>ar-eg</td>
<td>0x0C01</td>
<td>3073</td>
<td>Italian - Standard</td>
<td>it</td>
<td>0x0410</td>
<td>1040</td>
</tr>
<tr>
<td>Arabic - Iraq</td>
<td>ar-iq</td>
<td>0x0801</td>
<td>2049</td>
<td>Italian - Switzerland</td>
<td>it-ch</td>
<td>0x0810</td>
<td>2064</td>
</tr>
<tr>
<td>Arabic - Jordan</td>
<td>ar-jo</td>
<td>0x2C01</td>
<td>11265</td>
<td>Japanese</td>
<td>ja</td>
<td>0x0411</td>
<td>1041</td>
</tr>
<tr>
<td>Arabic - Kuwait</td>
<td>ar-kw</td>
<td>0x3401</td>
<td>13313</td>
<td>Korean</td>
<td>ko</td>
<td>0x0412</td>
<td>1042</td>
</tr>
<tr>
<td>Arabic - Lebanon</td>
<td>ar-lb</td>
<td>0x3001</td>
<td>12289</td>
<td>Latvian</td>
<td>lv</td>
<td>0x0426</td>
<td>1062</td>
</tr>
<tr>
<td>Arabic - Libya</td>
<td>ar-ly</td>
<td>0x1001</td>
<td>4097</td>
<td>Lithuanian</td>
<td>lt</td>
<td>0x0427</td>
<td>1063</td>
</tr>
<tr>
<td>Arabic - Morocco</td>
<td>ar-ma</td>
<td>0x1801</td>
<td>6145</td>
<td>Macedonian</td>
<td>mk</td>
<td>0x042F</td>
<td>1071</td>
</tr>
<tr>
<td>Arabic - Oman</td>
<td>ar-om</td>
<td>0x2001</td>
<td>8193</td>
<td>Malay - Malaysia</td>
<td>ms</td>
<td>0x043E</td>
<td>1086</td>
</tr>
<tr>
<td>Arabic - Qatar</td>
<td>ar-qa</td>
<td>0x4001</td>
<td>16385</td>
<td>Maltese</td>
<td>mt</td>
<td>0x043A</td>
<td>1082</td>
</tr>
<tr>
<td>-------------------</td>
<td>------</td>
<td>------------</td>
<td>-------------------</td>
<td>------</td>
<td>------------</td>
<td>-------------------</td>
<td>------</td>
</tr>
<tr>
<td>Arabic - Saudia Arabia</td>
<td>arsa</td>
<td>0x0401</td>
<td>1025</td>
<td>Norwegian - Bokmål</td>
<td>no</td>
<td>0x0414</td>
<td>1044</td>
</tr>
<tr>
<td>Arabic - Syria</td>
<td>arsy</td>
<td>0x2801</td>
<td>10241</td>
<td>Polish</td>
<td>pl</td>
<td>0x0415</td>
<td>1045</td>
</tr>
<tr>
<td>Arabic - Tunisia</td>
<td>artn</td>
<td>0x1C01</td>
<td>7169</td>
<td>Portuguese - Standard</td>
<td>pt</td>
<td>0x0816</td>
<td>2070</td>
</tr>
<tr>
<td>Arabic - Yemen</td>
<td>arye</td>
<td>0x2401</td>
<td>9217</td>
<td>Portuguese - Brazil</td>
<td>pt-br</td>
<td>0x0416</td>
<td>1046</td>
</tr>
<tr>
<td>Basque</td>
<td>eu</td>
<td>0x042D</td>
<td>1069</td>
<td>Raeto-Romance</td>
<td>rm</td>
<td>0x0417</td>
<td>1047</td>
</tr>
<tr>
<td>Belarusian</td>
<td>be</td>
<td>0x0423</td>
<td>1059</td>
<td>Romanian</td>
<td>ro</td>
<td>0x0418</td>
<td>1048</td>
</tr>
<tr>
<td>Bulgarian</td>
<td>bg</td>
<td>0x0402</td>
<td>1026</td>
<td>Romanian - Moldova</td>
<td>ro-mo</td>
<td>0x0818</td>
<td>2072</td>
</tr>
<tr>
<td>Catalan</td>
<td>ca</td>
<td>0x0403</td>
<td>1027</td>
<td>Russian</td>
<td>ru</td>
<td>0x0419</td>
<td>1049</td>
</tr>
<tr>
<td>Chinese</td>
<td>zh</td>
<td>0x0004</td>
<td>4</td>
<td>Russian - Moldova</td>
<td>ru-mo</td>
<td>0x0819</td>
<td>2073</td>
</tr>
<tr>
<td>Chinese - PRC</td>
<td>zhen</td>
<td>0x0804</td>
<td>2052</td>
<td>Serbian - Cyrillic</td>
<td>sr</td>
<td>0x0C1A</td>
<td>3098</td>
</tr>
<tr>
<td>Language</td>
<td>Code</td>
<td>ID</td>
<td>Name</td>
<td>Code</td>
<td>ID</td>
<td>Name</td>
<td>Code</td>
</tr>
<tr>
<td>----------</td>
<td>------</td>
<td>----</td>
<td>------</td>
<td>------</td>
<td>----</td>
<td>------</td>
<td>------</td>
</tr>
<tr>
<td>Chinese - Hong Kong</td>
<td>zh-hk</td>
<td>0x0C04</td>
<td>3076</td>
<td>Setsuana</td>
<td>tn</td>
<td>0x0432</td>
<td>1074</td>
</tr>
<tr>
<td>Chinese - Singapore</td>
<td>zh-sg</td>
<td>0x1004</td>
<td>4100</td>
<td>Slovenian</td>
<td>sl</td>
<td>0x0424</td>
<td>1060</td>
</tr>
<tr>
<td>Chinese - Taiwan</td>
<td>zh-tw</td>
<td>0x0404</td>
<td>1028</td>
<td>Slovak</td>
<td>sk</td>
<td>0x041B</td>
<td>1051</td>
</tr>
<tr>
<td>Croatian</td>
<td>hr</td>
<td>0x041A</td>
<td>1050</td>
<td>Sorbian</td>
<td>sb</td>
<td>0x042E</td>
<td>1070</td>
</tr>
<tr>
<td>Czech</td>
<td>cs</td>
<td>0x0405</td>
<td>1029</td>
<td>Spanish - Standard</td>
<td>es</td>
<td>0x040A</td>
<td>1034</td>
</tr>
<tr>
<td>Danish</td>
<td>da</td>
<td>0x0406</td>
<td>1030</td>
<td>Spanish - Argentina</td>
<td>es-ar</td>
<td>0x2C0A</td>
<td>11274</td>
</tr>
<tr>
<td>Dutch</td>
<td>nl</td>
<td>0x0413</td>
<td>1043</td>
<td>Spanish - Bolivia</td>
<td>es-bo</td>
<td>0x400A</td>
<td>16394</td>
</tr>
<tr>
<td>Dutch - Belgium</td>
<td>nl-be</td>
<td>0x0813</td>
<td>2067</td>
<td>Spanish - Chile</td>
<td>es-cl</td>
<td>0x340A</td>
<td>13322</td>
</tr>
<tr>
<td>English</td>
<td>en</td>
<td>0x0009</td>
<td>9</td>
<td>Spanish - Columbia</td>
<td>es-co</td>
<td>0x240A</td>
<td>9226</td>
</tr>
<tr>
<td>English - Australia</td>
<td>en-au</td>
<td>0x0C09</td>
<td>3081</td>
<td>Spanish - Costa Rica</td>
<td>es-cr</td>
<td>0x140A</td>
<td>5130</td>
</tr>
<tr>
<td>English - Belize</td>
<td>en-bz</td>
<td>0x2809</td>
<td>10249</td>
<td>Spanish - Dominican Republic</td>
<td>es-do</td>
<td>0x1C0A</td>
<td>7178</td>
</tr>
<tr>
<td>-----------------</td>
<td>-------</td>
<td>--------</td>
<td>-------</td>
<td>-----------------------------</td>
<td>-------</td>
<td>--------</td>
<td>------</td>
</tr>
<tr>
<td>English - Canada</td>
<td>en-ca</td>
<td>0x1009</td>
<td>4105</td>
<td>Spanish - Ecuador</td>
<td>es-eq</td>
<td>0x300A</td>
<td>12298</td>
</tr>
<tr>
<td>English - Ireland</td>
<td>en-ie</td>
<td>0x1809</td>
<td>6153</td>
<td>Spanish - Guatemala</td>
<td>es-gt</td>
<td>0x100A</td>
<td>4106</td>
</tr>
<tr>
<td>English - Jamaica</td>
<td>en-jm</td>
<td>0x2009</td>
<td>8201</td>
<td>Spanish - Honduras</td>
<td>es-hn</td>
<td>0x480A</td>
<td>18442</td>
</tr>
<tr>
<td>English - New Zealand</td>
<td>en-nz</td>
<td>0x1409</td>
<td>5129</td>
<td>Spanish - Mexico</td>
<td>es-mx</td>
<td>0x080A</td>
<td>2058</td>
</tr>
<tr>
<td>English - South Africa</td>
<td>en-za</td>
<td>0x1C09</td>
<td>7177</td>
<td>Spanish - Nicaragua</td>
<td>es-ni</td>
<td>0x4C0A</td>
<td>19466</td>
</tr>
<tr>
<td>English - Trinidad</td>
<td>en-tt</td>
<td>0x2C09</td>
<td>11273</td>
<td>Spanish - Panama</td>
<td>es-pa</td>
<td>0x180A</td>
<td>6154</td>
</tr>
<tr>
<td>English - United Kingdom</td>
<td>en-gb</td>
<td>0x0809</td>
<td>2057</td>
<td>Spanish - Peru</td>
<td>es-pe</td>
<td>0x280A</td>
<td>10250</td>
</tr>
<tr>
<td>English - United States</td>
<td>en-us</td>
<td>0x0409</td>
<td>1033</td>
<td>Spanish - Puerto Rico</td>
<td>es-pr</td>
<td>0x500A</td>
<td>20490</td>
</tr>
<tr>
<td>Language</td>
<td>Code</td>
<td>BCP</td>
<td>Region</td>
<td>Code</td>
<td>BCP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>------------------</td>
<td>------</td>
<td>-------</td>
<td>-----------</td>
<td>------</td>
<td>-------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Estonian</td>
<td>et</td>
<td>0x0425</td>
<td>1061</td>
<td>py</td>
<td>0x3C0A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Farsi</td>
<td>fa</td>
<td>0x0429</td>
<td>1065</td>
<td>es-sv</td>
<td>0x440A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Finnish</td>
<td>fi</td>
<td>0x040B</td>
<td>1035</td>
<td>es-uy</td>
<td>0x380A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Faeroese</td>
<td>fo</td>
<td>0x0438</td>
<td>1080</td>
<td>es-ve</td>
<td>0x200A</td>
<td></td>
<td></td>
</tr>
<tr>
<td>French - Standard</td>
<td>fr</td>
<td>0x040C</td>
<td>1036</td>
<td>sx</td>
<td>0x0430</td>
<td></td>
<td></td>
</tr>
<tr>
<td>French - Belgium</td>
<td>fr-be</td>
<td>0x080C</td>
<td>2060</td>
<td>sv</td>
<td>0x041D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>French - Canada</td>
<td>fr-ca</td>
<td>0x0C0C</td>
<td>3084</td>
<td>sv-fi</td>
<td>0x081D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>French - Luxembourg</td>
<td>fr-lu</td>
<td>0x140C</td>
<td>5132</td>
<td>th</td>
<td>0x041E</td>
<td></td>
<td></td>
</tr>
<tr>
<td>French - Switzerland</td>
<td>fr-ch</td>
<td>0x1C0C</td>
<td>4108</td>
<td>tr</td>
<td>0x041F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gaelic - Scotland</td>
<td>gd</td>
<td>0x03C0</td>
<td>1084</td>
<td>ts</td>
<td>0x0431</td>
<td></td>
<td></td>
</tr>
<tr>
<td>German -</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>------------------------</td>
<td>----------</td>
<td>------------</td>
<td>------------</td>
<td>----------</td>
<td>----------</td>
<td>------------</td>
<td>------------</td>
</tr>
<tr>
<td>German - Austrian</td>
<td>de-at</td>
<td>0x0407</td>
<td>1031</td>
<td>Ukranian</td>
<td>uk</td>
<td>0x0422</td>
<td>1058</td>
</tr>
<tr>
<td>German - Lichtenstein</td>
<td>de-li</td>
<td>0x1407</td>
<td>3079</td>
<td>Urdu - Pakistan</td>
<td>ur</td>
<td>0x0420</td>
<td>1056</td>
</tr>
<tr>
<td>German - Luxembourg</td>
<td>de-lu</td>
<td>0x1007</td>
<td>4103</td>
<td>Xhosa</td>
<td>xh</td>
<td>0x0434</td>
<td>1076</td>
</tr>
<tr>
<td>German - Switzerland</td>
<td>de-ch</td>
<td>0x0807</td>
<td>2055</td>
<td>Yiddish</td>
<td>yi</td>
<td>0x043D</td>
<td>1085</td>
</tr>
<tr>
<td>Greek</td>
<td>el</td>
<td>0x0408</td>
<td>1032</td>
<td>Zulu</td>
<td>zu</td>
<td>0x0435</td>
<td>1077</td>
</tr>
</tbody>
</table>
SetLocale Function

See Also

Description

Sets the global locale and returns the previous locale.

Syntax

SetLocale(lcid)

The lcid argument can be any valid 32-bit value or short string that uniquely identifies a geographical locale. Recognized values can be found in the Locale ID chart.

Remarks

If lcid is zero, the locale is set to match the current system setting.

A locale is a set of user preference information related to the user's language, country/region, and cultural conventions. The locale determines such things as keyboard layout, alphabetic sort order, as well as date, time, number, and currency formats.

The following example illustrates the use of the SetLocale function. To use this code, paste the entire example between the <BODY> tags of a standard HTML page.
Enter Date in UK format: <input type="text" id="UKDate" size="20">
Here's the US equivalent: <input type="text" id="USdate" size="20">

<input type="button" value="Convert" id="button1">

Enter a price in German: &nbsp; <input type="text" id="GermanNumber" size="20">
Here's the UK equivalent: <input type="text" id="USNumber" size="20">

<input type="button" value="Convert" id="button2">

<script language="vbscript">
Dim currentLocale
' Get the current locale
currentLocale = GetLocale

Sub Button1_onclick
    Dim original
    original = SetLocale("en-gb")
    mydate = CDate(UKDate.value)
    ' IE always sets the locale to US English so use th
    ' currentLocale variable to set the locale to US En
    original = SetLocale(currentLocale)
    USDate.value = FormatDateTime(mydate,vbShor
End Sub

Sub button2_onclick
    Dim original
original = SetLocale("de")
myvalue = CCur(GermanNumber.value)
original = SetLocale("en-gb")
USNumber.value = FormatCurrency(myvalue)
End Sub

</script>
Add Method (Dictionary)

See Also

Add Method (Folders)
Exists Method
Items Method
Keys Method
Remove Method
RemoveAll Method
Add Method (Dictionary) Applies To Dictionary Object
Add Method (Folders)

See Also

Add Method (Dictionary)
Add Method (Folders)
Applies To
-Folders Collection-
AtEndOfLine Property

See Also

AtEndOfStream Property
Column Property
Line Property
**AtEndOfLine Property**

Applies To

<table>
<thead>
<tr>
<th>TextStream Object</th>
<th></th>
<th></th>
</tr>
</thead>
</table>

[Scripting Run-Time Reference](#)
AtEndOfStream Property

See Also

AtEndOfLine Property
Column Property
Line Property
AtEndOfStream Property

Applies To

TextStream Object
Attributes

Property

See Also

DateCreated Property
DateLastAccessed Property
DateLastModified Property
Drive Property
Files Property
IsRootFolder Property
Name Property
ParentFolder Property
Path Property
ShortName Property
ShortPath Property
Size Property
SubFolders Property
Type Property
Attributes

Property

Applies To

- File Object
- Folder Object
AvailableSpace Property

See Also

DriveLetter Property
DriveType Property
FileSystem Property
FreeSpace Property
IsReady Property
Path Property
RootFolder Property
SerialNumber Property
ShareName Property
TotalSize Property
VolumeName Property
AvailableSpace

Property

Applies To

Drive Object
BuildPath Method

See Also

GetAbsolutePathName Method
GetBaseName Method
GetDriveName Method
GetExtensionName Method
GetFileName Method
GetParentFolderName Method
GetTempName Method
BuildPath Method

Applies To

FileSystemObject Object
Close Method

See Also

Read Method
ReadAll Method
ReadLine Method
Skip Method
SkipLine Method
Write Method
WriteLine Method
WriteBlankLines Method
Close

Method

Applies To

TextStream Object
Column

Property

See Also

AtEndOfLine Property
AtEndOfStream Property
Line Property
Property
Applies To

TextStream Object
CompareMode Property

See Also

Count Property
Item Property
Key Property
CompareMode Property
Applies To

Dictionary Object
Copy Method

See Also

CopyFile Method
CopyFolder Method
Delete Method
Move Method
OpenAsTextStream Method
Copy

Method

Applies To

File Object
Folder Object
CopyFile Method

See Also

- Copy Method
- CopyFolder Method
- CreateTextFile Method
- DeleteFile Method
- MoveFile Method

Scripting Run-Time Reference
CopyFile Method

Applies To

FileSystemObject Object
CopyFolder Method

See Also

CopyFile Method
Copy Method
CreateFolder Method
DeleteFolder Method
MoveFolder Method
CopyFolder Method
Applies To

FileSystemObject Object

Scripting Run-Time Reference
Count Property

See Also

- CompareMode Property
- Item Property
- Key Property
## Count Property

### Applies To

- Dictionary Object
- Drives Collection
- Files Collection
- Folders Collection
- Matches Collection
CreateFolder Method

See Also

CopyFolder Method
DeleteFolder Method
MoveFolder Method

Scripting Run-Time Reference
CreateFolder Method

Applies To

FileSystemObject Object
CreateTextFile Method

See Also

CreateFolder Method
OpenAsTextStream Method
OpenTextFile Method
CreateTextFile Method

Applies To

FileSystemObject Object
Folder Object
DateCreated Property

See Also

Attributes Property
DateLastAccessed Property
DateLastModified Property
Drive Property
Files Property
IsRootFolder Property
Name Property
ParentFolder Property
Path Property
ShortName Property
ShortPath Property
Size Property
SubFolders Property
Type Property
DateCreated Property
Applies To

File Object
Folder Object
DateLastAccessed Property

See Also

Attributes Property
DateCreated Property
DateLastModified Property
Drive Property
Files Property
IsRootFolder Property
Name Property
ParentFolder Property
Path Property
ShortName Property
ShortPath Property
Size Property
SubFolders Property
Type Property
DateLastAccessed Property
Applies To

File Object
Folder Object
DateLastModified

Property

Applies To

File Object
Folder Object
Delete Method

See Also

Copy Method
DeleteFile Method
DeleteFolder Method
Move Method
OpenAsTextStream Method
Delete

Method

Applies To

File Object
Folder Object
DeleteFile Method

See Also

CopyFile Method
CreateTextFile Method
Delete Method
DeleteFolder Method
MoveFile Method
DeleteFile Method

Applies To

FileSystemObject Object
DeleteFolder Method

See Also

CopyFolder Method
CreateFolder Method
Delete Method
DeleteFile Method
MoveFolder Method
**DeleteFolder Method**

Applies To

*FileSystemObject Object*
Dictionary Object

See Also

FileSystemObject Object
TextStream Object
Dictionary Object

Properties

- CompareMode Property
- Count Property
- Item Property
- Key Property
Dictionary Object

Methods

Add Method (Dictionary)
Exists Method
Items Method
Keys Method
Remove Method
RemoveAll Method
Drive Object

See Also

Drives Collection
File Object
Files Collection
Folder Object
Folders Collection
GetDrive Method
Drive Object Properties

AvailableSpace Property
DriveLetter Property
DriveType Property
FileSystem Property
FreeSpace Property
IsReady Property
Path Property
RootFolder Property
SerialNumber Property
ShareName Property
TotalSize Property
VolumeName Property
The **Drive** object has no methods.
Drive Property

See Also

Attributes Property
DateCreated Property
DateLastAccessed Property
DateLastModified Property
Files Property
IsRootFolder Property
Name Property
ParentFolder Property
Path Property
ShortName Property
ShortPath Property
Size Property
SubFolders Property
Type Property
Drive

Property

Applies To

File Object
Folder Object
DriveExists Method

See Also

Drive Object
Drives Collection
FileExists Method
FolderExists Method
GetDrive Method
GetDriveName Method
IsReady Property

Scripting Run-Time Reference
DriveExists  
Method  
Applies To  

FileSystemObject Object
DriveLetter Property

See Also

AvailableSpace Property
DriveType Property
FileSystem Property
FreeSpace Property
IsReady Property
Path Property
RootFolder Property
SerialNumber Property
ShareName Property
TotalSize Property
VolumeName Property
**DriveLetter Property**

**Applies To**

**Drive Object**
Drives Collection

See Also

Drive Object
Drives Property
File Object
Files Collection
Folder Object
Folders Collection
Drives

Collection Properties

Count Property
Item Property
Drives

Collection

Methods

The **Drives** collection has no methods.
Drives

Property

See Also

Drives Collection
Files Property
SubFolders Property
Drives

Property

Applies To

FileSystemObject Object
DriveType Constants

See Also

- Comparison Constants
- File Attribute Constants
- File Input/Output Constants
- SpecialFolder Constants
- Tristate Constants

Scripting Run-Time Reference
DriveType Property

See Also

AvailableSpace Property
DriveLetter Property
FileSystem Property
FreeSpace Property
IsReady Property
Path Property
RootFolder Property
SerialNumber Property
ShareName Property
TotalSize Property
VolumeName Property
# DriveType Property

**Applies To**

<table>
<thead>
<tr>
<th>Drive Object</th>
</tr>
</thead>
</table>
Exists Method

See Also

Add Method (Dictionary)  
Items Method  
Keys Method  
Remove Method  
RemoveAll Method
Exists

Method
Applies To

Dictionary Object
FileAttribute Constants

See Also

Comparison Constants
DriveType Constants
File Input/Output Constants
SpecialFolder Constants
Tristate Constants
Input/Output Constants

See Also

Comparison Constants
DriveType Constants
File Attribute Constants
SpecialFolder Constants
Tristate Constants

Scripting Run-Time Reference
FileExists Method

See Also

DriveExists Method
FolderExists Method
GetFile Method
GetFileName Method

Scripting Run-Time Reference
FileExists Method

Applies To

FileSystemObject Object
File Object

See Also

Drive Object
Drives Collection
Files Collection
Folder Object
Folders Collection
Object Properties

Attributes Property
DateCreated Property
DateLastAccessed Property
DateLastModified Property
Drive Property
Name Property
ParentFolder Property
Path Property
ShortName Property
ShortPath Property
Size Property
Type Property
File Object Methods

Copy Method
Delete Method
Move Method
OpenAsTextStream Method
Files

Collection

See Also

Drive Object
Drives Collection
File Object
Folder Object
Folders Collection

Scripting Run-Time Reference
Files

Collection Properties

Count Property
Item Property
Files Collection Methods

The **Files** collection has no methods.
Properties

See Also

Attributes Property
DateCreated Property
DateLastAccessed Property
DateLastModified Property
Drives Property
IsRootFolder Property
Name Property
ParentFolder Property
Path Property
ShortName Property
ShortPath Property
Size Property
SubFolders Property
Type Property
Property
Applies To

Folder Object
FileSystemObject Object

See Also

CreateObject Function
Dictionary Object
Drive Object
Drives Collection
File Object
FileSystem Property
Files Collection
Folder Object
Folders Collection
TextStream Object
FileSystemObject

Object

Properties

---

Drives Property
FileSystemObject
Object
Methods

BuildPath Method
CopyFile Method
CopyFolder Method
CreateFolder Method
CreateTextFile Method
DeleteFile Method
DeleteFolder Method
DriveExists Method
FileExists Method
FolderExists Method
GetAbsolutePathName Method
GetBaseName Method
GetDrive Method
GetDriveName Method
GetExtensionName Method
GetFile Method
GetFileName Method
GetFolder Method
GetParentFolderName Method
GetSpecialFolder Method
GetTempName Method
MoveFile Method
MoveFolder Method
OpenTextFile Method
FileSystemObject Constants

See Also

VBScript Constants
**FileSystem Property**

See Also

- `AvailableSpace Property`
- `DriveLetter Property`
- `DriveType Property`
- `FileSystemObject Object`
- `FreeSpace Property`
- `IsReady Property`
- `Path Property`
- `RootFolder Property`
- `SerialNumber Property`
- `ShareName Property`
- `TotalSize Property`
- `VolumeName Property`
**FileSystem Property**

Applies To

Drive Object
Folder Object

See Also

Drive Object
Drives Collection
File Object
Files Collection
Folders Collection
Folder
Object Properties

Attributes Property
DateCreated Property
DateLastAccessed Property
DateLastModified Property
Drive Property
Files Property
IsRootFolder Property
Name Property
ParentFolder Property
Path Property
ShortName Property
ShortPath Property
Size Property
SubFolders Property
Type Property
Folder

Object Methods

Copy Method
Delete Method
Move Method
CreateTextFile Method
Folders Collection

See Also

Drive Object
Drives Collection
File Object
Files Collection
Folder Object
SubFolders Property
Folders

Collection

Properties

Count Property
Item Property
Folders

Collection Methods

Add Method
FolderExists Method

See Also

DriveExists Method
FileExists Method
GetFolder Method
GetParentFolderName Method
<table>
<thead>
<tr>
<th>FolderExists Method</th>
<th>Applies To</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>FileSystemObject Object</td>
</tr>
</tbody>
</table>
FreeSpace Property

See Also

AvailableSpace Property
DriveLetter Property
DriveType Property
FileSystem Property
IsReady Property
Path Property
RootFolder Property
SerialNumber Property
ShareName Property
TotalSize Property
VolumeName Property
FreeSpace Property
Applies To

Drive Object

Scripting Run-Time Reference
GetAbsolutePathName Method

See Also

GetBaseName Method
GetDrive Method
GetDriveName Method
GetExtensionName Method
GetFile Method
GetFileName Method
GetFileVersion Method
GetFolder Method
GetParentFolderName Method
GetSpecialFolder Method
GetTempName Method
GetAbsolutePathName
Method
Applies To

FileSystemObject Object
GetBaseName Method

See Also

GetAbsolutePathName Method
GetDrive Method
GetDriveName Method
GetExtensionName Method
GetFile Method
GetFileName Method
FileVersion Method
GetFolder Method
GetParentFolderName Method
GetSpecialFolder Method
GetTempName Method
GetBaseName

Method

Applies To

FileSystemObject Object
GetDrive Method

See Also

GetAbsolutePathName Method
GetBaseName Method
GetDriveName Method
GetExtensionName Method
GetFile Method
GetFileName Method
GetFileVersion Method
GetFolder Method
GetParentFolderName Method
GetSpecialFolder Method
GetTempName Method
GetDrive Method

Applies To

FileSystemObject Object
GetDriveName Method

See Also

GetAbsolutePathName Method
GetBaseName Method
GetDrive Method
GetExtensionName Method
GetFile Method
GetFileName Method
GetFileVersion Method
GetFolder Method
GetParentFolderName Method
GetSpecialFolder Method
GetTempName Method
GetDriveName

Method

Applies To

FileSystemObject Object
GetExtensionName Method

See Also

GetAbsolutePathName Method
GetBaseName Method
GetDrive Method
GetDriveName Method
GetFile Method
GetFileName Method
GetFileVersion Method
GetFolder Method
GetParentFolderName Method
GetSpecialFolder Method
GetTempName Method
GetExtensionName Method
Applies To

FileSystemObject Object
GetFile Method

See Also

GetAbsolutePathName Method
GetBaseName Method
GetDrive Method
GetDriveName Method
GetExtensionName Method
GetFileName Method
GetFileVersion Method
GetFolder Method
GetParentFolderName Method
GetSpecialFolder Method
GetTempName Method
GetFile

Method

Applies To

FileSystemObject Object
GetFileName Method

See Also

GetAbsolutePathName Method
GetBaseName Method
GetDrive Method
GetDriveName Method
GetExtensionName Method
GetFile Method
GetFileVersion Method
GetFolder Method
GetParentFolderName Method
GetSpecialFolder Method
GetTempName Method
**GetFileName Method**

Applies To

<table>
<thead>
<tr>
<th>FileSystemObject Object</th>
</tr>
</thead>
</table>
GetFileVersion Method

See Also

GetAbsolutePathName Method
GetBaseName Method
GetDrive Method
GetDriveName Method
GetExtensionName Method
GetFileName Method
GetFolder Method
GetParentFolderName Method
GetSpecialFolder Method
GetTempName Method
### GetFileVersion Method

#### Applies To

<table>
<thead>
<tr>
<th>FileSystemObject Object</th>
</tr>
</thead>
</table>
GetFolder Method

See Also

GetAbsolutePathName Method
GetBaseName Method
GetDrive Method
GetDriveName Method
GetExtensionName Method
GetFile Method
GetFileName Method
GetFileVersion Method
GetParentFolderName Method
GetSpecialFolder Method
GetTempName Method
GetFolder Method

Applies To

FileSystemObject Object
GetParentFolderName Method

See Also

GetAbsolutePathName Method
GetBaseName Method
GetDrive Method
GetDriveName Method
GetExtensionName Method
GetFile Method
GetFileName Method
GetFileVersion Method
GetFolder Method
GetSpecialFolder Method
GetTempName Method
GetParentFolderName Method
Applies To

FileSystemObject Object
GetSpecialFolder Method

See Also

GetAbsolutePathName Method
GetBaseName Method
GetDrive Method
GetDriveName Method
GetExtensionName Method
GetFileName Method
GetFileVersion Method
GetFolder Method
GetParentFolderName Method
GetTempName Method
**GetSpecialFolder Method**

**Applies To**

FileSystemObject Object
GetTempName Method

See Also

GetAbsolutePathName Method
GetBaseName Method
GetDrive Method
GetDriveName Method
GetExtensionName Method
GetFile Method
GetFileName Method
GetFileVersion Method
GetFolder Method
GetParentFolderName Method
GetSpecialFolder Method
GetTempName Method
Applies To

FileSystemObject Object
Microsoft® Visual Basic® Scripting Edition

IsReady

Property

See Also

AvailableSpace Property
DriveLetter Property
DriveType Property
FileSystem Property
FreeSpace Property
Path Property
RootFolder Property
SerialNumber Property
ShareName Property
TotalSize Property
VolumeName Property
IsReady

Property

Applies To

Drive Object
IsRootFolder

Property

Applies To

Folder Object
Item

Property

See Also

CompareMode Property
Count Property
Key Property
## Item

### Property

Applies To

- Dictionary Object
- Drives Collection
- Files Collection
- Folders Collection
- Matches Collection
Items

Method

See Also

Add Method (Dictionary)
Exists Method
Keys Method
Remove Method
RemoveAll Method
Items

Method
Applies To

Dictionary Object
Key

Property

See Also

CompareMode Property
Count Property
Item Property
<table>
<thead>
<tr>
<th>Property</th>
<th>Applies To</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Dictionary Object</td>
</tr>
</tbody>
</table>
array
A set of sequentially indexed elements having the same type of data. Each element of an array has a unique identifying index number. Changes made to one element of an array do not affect the other elements.

collection
An object that contains a set of related objects. An object's position in the collection can change whenever a change occurs in the collection; therefore, the position of any specific object in the collection may vary.

run-time error
An error that occurs when code is running. A run-time error results when a statement attempts an invalid operation.

string expression
Any expression that evaluates to a sequence of contiguous characters. Elements of a string expression can include a function that returns a string, a string literal, a string constant, or a string variable.

type library
A file or component within another file that contains standard descriptions of exposed objects, properties, and methods.
Keys

Method

See Also

- Add Method (Dictionary)
- Exists Method
- Items Method
- Remove Method
- RemoveAll Method

Scripting Run-Time Reference
**Keys**

**Method**

**Applies To**

*Dictionary Object*
Line Property

See Also

AtEndOfStream Property
AtEndOfLine Property
Column Property
Property
Applies To

TextStream Object
Move Method

See Also

Copy Method
Delete Method
MoveFile Method
MoveFolder Method
OpenAsTextStream Method
Move Method
Applies To

File Object
Folder Object
MoveFile Method

See Also

CopyFile Method
DeleteFile Method
GetFile Method
GetFileName Method
Move Method
MoveFolder Method
OpenTextFile Method
MoveFile Method

Applies To

FileSystemObject Object
MoveFolder Method

See Also

CopyFolder Method
CreateFolder Method
DeleteFolder Method
GetFolder Method
GetParentFolderName Method
Move Method
MoveFile Method
MoveFolder Method
Applies To

FileSystemObject Object
Name

Property

See Also

Attributes Property
DateCreated Property
DateLastAccessed Property
DateLastModified Property
Drive Property
Files Property
IsRootFolder Property
ParentFolder Property
Path Property
ShortName Property
ShortPath Property
Size Property
SubFolders Property
Type Property
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microsoft® Visual Basic® Scripting Edition</td>
<td></td>
</tr>
</tbody>
</table>

**Property**

**Applies To**

- File Object
- Folder Object
OpenAsTextStream Method

See Also

Copy Method
CreateTextFile Method
Delete Method
Move Method
OpenTextFile Method
OpenAsTextStream

Method

Applies To

File Object
OpenTextFile Method

See Also

OpenAsTextStream Method
CreateTextFile Method
OpenTextFile Method

Applies To

FileSystemObject Object
ParentFolder Property

See Also

Attributes Property
DateCreated Property
DateLastAccessed Property
DateLastModified Property
Drive Property
Files Property
IsRootFolder Property
Name Property
Path Property
ShortName Property
ShortPath Property
Size Property
SubFolders Property
Type Property
ParentFolder Property
Applies To

File Object
Folder Object
Path Property

See Also

Attributes Property
AvailableSpace Property
DateCreated Property
DateLastAccessed Property
DateLastModified Property
Drive Property
DriveLetter Property
DriveType Property
Files Property
FileSystem Property
FreeSpace Property
IsReady Property
IsRootFolder Property
Name Property
ParentFolder Property
RootFolder Property
SerialNumber Property
ShareName Property
ShortName Property
ShortPath Property
Size Property
SubFolders Property
TotalSize Property
Type Property
VolumeName Property
<table>
<thead>
<tr>
<th>Path Property</th>
<th>Applies To</th>
</tr>
</thead>
<tbody>
<tr>
<td>Drive Object</td>
<td></td>
</tr>
<tr>
<td>File Object</td>
<td></td>
</tr>
<tr>
<td>Folder Object</td>
<td></td>
</tr>
</tbody>
</table>
Read Method

See Also

Close Method
ReadAll Method
ReadLine Method
Skip Method
SkipLine Method
Write Method
WriteLine Method
WriteBlankLines Method
Read Method
Applies To

TextStream Object
**ReadAll Method**

See Also

- Close Method
- Read Method
- ReadLine Method
- Skip Method
- SkipLine Method
- Write Method
- WriteLine Method
- WriteBlankLines Method
ReadAll Method

Applies To

TextStream Object
ReadLine Method

See Also

Close Method
Read Method
ReadAll Method
Skip Method
SkipLine Method
Write Method
WriteLine Method
WriteBlankLines Method
ReadLine Method

Applies To

TextStream Object
Remove Method

See Also

Add Method (Dictionary)
Exists Method
Items Method
Keys Method
RemoveAll Method
Remove Method Applies To

Dictionary Object
RemoveAll Method

See Also

Add Method (Dictionary)
Exists Method
Items Method
Keys Method
Remove Method
RemoveAll Method

Applies To

Dictionary Object
RootFolder Property

See Also

AvailableSpace Property
DriveLetter Property
DriveType Property
FileSystem Property
FreeSpace Property
IsReady Property
Path Property
SerialNumber Property
ShareName Property
TotalSize Property
VolumeName Property
RootFolder
Property
Applies To

Drive Object
SerialNumber

Property

See Also

AvailableSpace Property
DriveLetter Property
DriveType Property
FileSystem Property
FreeSpace Property
IsReady Property
Path Property
RootFolder Property
ShareName Property
TotalSize Property
VolumeName Property
SerialNumber

Property

Applies To

Drive Object
ShareName Property

See Also

AvailableSpace Property
DriveLetter Property
DriveType Property
FileSystem Property
FreeSpace Property
IsReady Property
Path Property
RootFolder Property
SerialNumber Property
TotalSize Property
VolumeName Property
ShareName

Property

Applies To

Drive Object
ShortName Property

See Also

Attributes Property
DateCreated Property
DateLastAccessed Property
DateLastModified Property
Drive Property
Files Property
IsRootFolder Property
Name Property
ParentFolder Property
Path Property
ShortPath Property
Size Property
SubFolders Property
Type Property
# ShortName Property

**Applies To**

- File Object
- Folder Object
ShortPath Property

See Also

Attributes Property
DateCreated Property
DateLastAccessed Property
DateLastModified Property
Drive Property
Files Property
IsRootFolder Property
Name Property
ParentFolder Property
Path Property
ShortName Property
Size Property
SubFolders Property
Type Property
ShortPath Property
Applies To

File Object
Folder Object
Size

Property

See Also

Attributes Property
DateCreated Property
DateLastAccessed Property
DateLastModified Property
Drive Property
Files Property
IsRootFolder Property
Name Property
ParentFolder Property
Path Property
ShortName Property
ShortPath Property
SubFolders Property
Type Property
Size

Property

Applies To

- File Object
- Folder Object
Method

See Also

Close Method
Read Method
ReadAll Method
ReadLine Method
SkipLine Method
Write Method
WriteLine Method
WriteBlankLines Method
Method
Applies To

TextStream Object
SkipLine Method

See Also

Close Method
Read Method
ReadAll Method
ReadLine Method
Skip Method
Write Method
WriteLine Method
WriteBlankLines Method
SkipLine Method

Applies To

TextStream Object
SpecialFolder Constants

See Also

Comparison Constants
DriveType Constants
FileAttribute Constants
File Input/Output Constants
Tristate Constants
SubFolders Property

See Also

Attributes Property
DateCreated Property
DateLastAccessed Property
DateLastModified Property
Drive Property
Files Property
IsRootFolder Property
Name Property
ParentFolder Property
Path Property
ShortName Property
ShortPath Property
Size Property
Type Property
SubFolders Property
Applies To

Folder Object
TextStream Object

See Also

Dictionary Object
FileSystemObject Object
TextStream Object

Properties

AtEndOfLine Property
AtEndOfStream Property
Column Property
Line Property
TextStream Object Methods

Close Method
Read Method
ReadAll Method
ReadLine Method
Skip Method
SkipLine Method
Write Method
WriteLine Method
WriteBlankLines Method
TotalSize Property

See Also

AvailableSpace Property
DriveLetter Property
DriveType Property
FileSystem Property
FreeSpace Property
IsReady Property
Path Property
RootFolder Property
SerialNumber Property
ShareName Property
VolumeName Property
TotalSize Property

Applies To

Drive Object
Type

Property

See Also

- Attributes Property
- DateCreated Property
- DateLastAccessed Property
- DateLastModified Property
- Drive Property
- Files Property
- IsRootFolder Property
- Name Property
- ParentFolder Property
- Path Property
- ShortName Property
- ShortPath Property
- Size Property
- SubFolders Property
<table>
<thead>
<tr>
<th>Property</th>
<th>Applies To</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td></td>
</tr>
<tr>
<td>File Object</td>
<td></td>
</tr>
<tr>
<td>Folder Object</td>
<td></td>
</tr>
</tbody>
</table>
VolumeName
Property

See Also

AvailableSpace Property
DriveLetter Property
DriveType Property
FileSystem Property
FreeSpace Property
IsReady Property
Path Property
RootFolder Property
SerialNumber Property
ShareName Property
TotalSize Property
<table>
<thead>
<tr>
<th>VolumeName Property</th>
<th>Applies To</th>
</tr>
</thead>
<tbody>
<tr>
<td>Drive Object</td>
<td></td>
</tr>
</tbody>
</table>
Write Method

See Also

Close Method
Read Method
ReadAll Method
ReadLine Method
Skip Method
SkipLine Method
WriteLine Method
WriteBlankLines Method
Write Method Applies To TextStream Object
WriteBlankLines Method

See Also

Close Method
Read Method
ReadAll Method
ReadLine Method
Skip Method
SkipLine Method
Write Method
WriteLine Method
WriteBlankLines Method
Applies To

TextStream Object
WriteLine Method

See Also

Close Method
Read Method
ReadAll Method
ReadLine Method
Skip Method
SkipLine Method
Write Method
WriteBlankLines Method
WriteLine Method

Applies To

TextStream Object
InStrRev Function

See Also

InStr Function
Arithmetic Operators

^ Operator
* Operator
/ Operator
\ Operator
Mod Operator
+ Operator
- Operator
Concatenation Operators
Concatenation Operators

& Operator
+ Operator
Operator Summary

See Also

Arithmetic Operators

Operators used to perform mathematical calculations.

Assignment Operator

Operator used to assign a value to a property or variable.

Comparison Operators

Operators used to perform comparisons.

Concatenation Operators

Operators used to combine strings.

Logical Operators

Operators used to perform logical operations.
Logical Operators

And Operator
Not Operator
Or Operator
Xor Operator
Derived Math Functions

Description

The following nonintrinsic math functions can be derived from the intrinsic math functions:

<table>
<thead>
<tr>
<th>Function</th>
<th>Derived equivalents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Secant</td>
<td>Sec(X) = 1 / Cos(X)</td>
</tr>
<tr>
<td>Cosecant</td>
<td>Cosec(X) = 1 / Sin(X)</td>
</tr>
<tr>
<td>Cotangent</td>
<td>Cotan(X) = 1 / Tan(X)</td>
</tr>
<tr>
<td>Inverse Sine</td>
<td>Arcsin(X) = Atn(X / Sqr(-X * X + 1))</td>
</tr>
<tr>
<td>Inverse Cosine</td>
<td>Arccos(X) = Atn(-X / Sqr(-X * X + 1)) + 2 * Atn(1)</td>
</tr>
<tr>
<td>Inverse Secant</td>
<td>Arcsec(X) = Atn(X / Sqr(X * X - 1)) + Sgn((X) -1) * (2 * Atn(1))</td>
</tr>
<tr>
<td>Inverse Cosecant</td>
<td>Arccosec(X) = Atn(X / Sqr(X * X - 1)) + (Sgn(X) - 1) * (2 * Atn(1))</td>
</tr>
<tr>
<td>Inverse Cotangent</td>
<td>Arccotan(X) = Atn(X) + 2 * Atn(1)</td>
</tr>
<tr>
<td>Hyperbolic Sine</td>
<td>HSin(X) = (Exp(X) - Exp(-X)) / 2</td>
</tr>
<tr>
<td>Hyperbolic Cosine</td>
<td>HCos(X) = (Exp(X) + Exp(-X)) / 2</td>
</tr>
<tr>
<td>Hyperbolic Tangent</td>
<td>HTan(X) = (Exp(X) - Exp(-X)) / (Exp(X) + Exp(-X))</td>
</tr>
<tr>
<td>Hyperbolic Secant</td>
<td>HSec(X) = 2 / (Exp(X) + Exp(-X))</td>
</tr>
<tr>
<td>Hyperbolic Cosecant</td>
<td>HCosec(X) = 2 / (Exp(X) - Exp(-X))</td>
</tr>
<tr>
<td>Hyperbolic</td>
<td>HCotan(X) = (Exp(X) + Exp(-X)) / (Exp(X))</td>
</tr>
<tr>
<td>Cotangent</td>
<td>- Exp(-X)</td>
</tr>
<tr>
<td>-----------------</td>
<td>--------------------------------</td>
</tr>
<tr>
<td>Inverse Hyperbolic Sine</td>
<td>HArcsin(X) = Log(X + Sqr(X * X + 1))</td>
</tr>
<tr>
<td>Inverse Hyperbolic Cosine</td>
<td>HArcos(X) = Log(X + Sqr(X * X - 1))</td>
</tr>
<tr>
<td>Inverse Hyperbolic Tangent</td>
<td>HArcctan(X) = Log((1 + X) / (1 - X)) / 2</td>
</tr>
<tr>
<td>Inverse Hyperbolic Secant</td>
<td>HArcsec(X) = Log((Sqr(-X * X + 1) + 1) / X)</td>
</tr>
<tr>
<td>Inverse Hyperbolic Cosecant</td>
<td>HArcsec(X) = Log((Sgn(X) * Sqr(X * X + 1) + 1) / X)</td>
</tr>
<tr>
<td>Inverse Hyperbolic Cotangent</td>
<td>HArcctan(X) = Log((X + 1) / (X - 1)) / 2</td>
</tr>
<tr>
<td>Logarithm to base N</td>
<td>LogN(X) = Log(X) / Log(N)</td>
</tr>
</tbody>
</table>
# VBScript Error Messages

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>Invalid procedure call or argument</td>
</tr>
<tr>
<td>6</td>
<td>Overflow</td>
</tr>
<tr>
<td>7</td>
<td>Out of memory</td>
</tr>
<tr>
<td>9</td>
<td>Subscript out of range</td>
</tr>
<tr>
<td>10</td>
<td>Array fixed or temporarily locked</td>
</tr>
<tr>
<td>11</td>
<td>Division by zero</td>
</tr>
<tr>
<td>13</td>
<td>Type mismatch</td>
</tr>
<tr>
<td>14</td>
<td>Out of string space</td>
</tr>
<tr>
<td>28</td>
<td>Out of stack space</td>
</tr>
<tr>
<td>35</td>
<td><strong>Sub</strong> or <strong>Function</strong> not defined</td>
</tr>
<tr>
<td>48</td>
<td>Error in loading DLL</td>
</tr>
<tr>
<td>51</td>
<td>Internal error</td>
</tr>
<tr>
<td>53</td>
<td>File not found</td>
</tr>
<tr>
<td>57</td>
<td>Device I/O error</td>
</tr>
<tr>
<td>58</td>
<td>File already exists</td>
</tr>
<tr>
<td>61</td>
<td>Disk full</td>
</tr>
<tr>
<td>67</td>
<td>Too many files</td>
</tr>
<tr>
<td>70</td>
<td>Permission denied</td>
</tr>
<tr>
<td>75</td>
<td>Path/File access error</td>
</tr>
<tr>
<td>76</td>
<td>Path not found</td>
</tr>
<tr>
<td>91</td>
<td>Object variable or <strong>With</strong> block variable not set</td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>-------------------------------------------------------</td>
</tr>
<tr>
<td>92</td>
<td><strong>For</strong> loop not initialized</td>
</tr>
<tr>
<td>94</td>
<td>Invalid use of <strong>Null</strong></td>
</tr>
<tr>
<td>322</td>
<td>Can't create necessary temporary file</td>
</tr>
<tr>
<td>424</td>
<td>Object required</td>
</tr>
<tr>
<td>429</td>
<td>ActiveX component can't create object</td>
</tr>
<tr>
<td>430</td>
<td>Class doesn't support Automation</td>
</tr>
<tr>
<td>432</td>
<td>File name or class name not found during Automation operation</td>
</tr>
<tr>
<td>438</td>
<td>Object doesn't support this property or method</td>
</tr>
<tr>
<td>440</td>
<td>Automation error</td>
</tr>
<tr>
<td>445</td>
<td>Object doesn't support this action</td>
</tr>
<tr>
<td>446</td>
<td>Object doesn't support named arguments</td>
</tr>
<tr>
<td>447</td>
<td>Object doesn't support current locale setting</td>
</tr>
<tr>
<td>448</td>
<td>Named argument not found</td>
</tr>
<tr>
<td>449</td>
<td>Argument not optional</td>
</tr>
<tr>
<td>450</td>
<td>Wrong number of arguments or invalid property assignment</td>
</tr>
<tr>
<td>451</td>
<td>Object not a collection</td>
</tr>
<tr>
<td>453</td>
<td>Specified DLL function not found</td>
</tr>
<tr>
<td>455</td>
<td>Code resource lock error</td>
</tr>
<tr>
<td>457</td>
<td>This key already associated with an element of this collection</td>
</tr>
<tr>
<td>458</td>
<td>Variable uses an Automation type not supported in VBScript</td>
</tr>
<tr>
<td>500</td>
<td>Variable is undefined</td>
</tr>
<tr>
<td>501</td>
<td>Illegal assignment</td>
</tr>
<tr>
<td>502</td>
<td>Object not safe for scripting</td>
</tr>
<tr>
<td>503</td>
<td>Object not safe for initializing</td>
</tr>
<tr>
<td>1001</td>
<td>Out of memory</td>
</tr>
<tr>
<td>1002</td>
<td>Syntax error</td>
</tr>
<tr>
<td>1003</td>
<td>Expected ':'</td>
</tr>
<tr>
<td>Line</td>
<td>Error Description</td>
</tr>
<tr>
<td>------</td>
<td>--------------------------------------</td>
</tr>
<tr>
<td>1004</td>
<td>Expected ';'</td>
</tr>
<tr>
<td>1005</td>
<td>Expected '('</td>
</tr>
<tr>
<td>1006</td>
<td>Expected ')'</td>
</tr>
<tr>
<td>1007</td>
<td>Expected ']'</td>
</tr>
<tr>
<td>1008</td>
<td>Expected '{'</td>
</tr>
<tr>
<td>1009</td>
<td>Expected '}'}</td>
</tr>
<tr>
<td>1010</td>
<td>Expected identifier</td>
</tr>
<tr>
<td>1011</td>
<td>Expected '='</td>
</tr>
<tr>
<td>1012</td>
<td>Expected 'If'</td>
</tr>
<tr>
<td>1013</td>
<td>Expected 'To'</td>
</tr>
<tr>
<td>1014</td>
<td>Expected 'End'</td>
</tr>
<tr>
<td>1015</td>
<td>Expected 'Function'</td>
</tr>
<tr>
<td>1016</td>
<td>Expected 'Sub'</td>
</tr>
<tr>
<td>1017</td>
<td>Expected 'Then'</td>
</tr>
<tr>
<td>1018</td>
<td>Expected 'Wend'</td>
</tr>
<tr>
<td>1019</td>
<td>Expected 'Loop'</td>
</tr>
<tr>
<td>1020</td>
<td>Expected 'Next'</td>
</tr>
<tr>
<td>1021</td>
<td>Expected 'Case'</td>
</tr>
<tr>
<td>1022</td>
<td>Expected 'Select'</td>
</tr>
<tr>
<td>1023</td>
<td>Expected expression</td>
</tr>
<tr>
<td>1024</td>
<td>Expected statement</td>
</tr>
<tr>
<td>1025</td>
<td>Expected end of statement</td>
</tr>
<tr>
<td>1026</td>
<td>Expected integer constant</td>
</tr>
<tr>
<td>1027</td>
<td>Expected 'While' or 'Until'</td>
</tr>
<tr>
<td>1028</td>
<td>Expected 'While', 'Until', or end of statement</td>
</tr>
<tr>
<td>1029</td>
<td>Too many locals or arguments</td>
</tr>
<tr>
<td>1030</td>
<td>Identifier too long</td>
</tr>
<tr>
<td>1031</td>
<td>Invalid number</td>
</tr>
<tr>
<td>1032</td>
<td>Invalid character</td>
</tr>
<tr>
<td></td>
<td>Description</td>
</tr>
<tr>
<td>---</td>
<td>-------------------------------------</td>
</tr>
<tr>
<td>1033</td>
<td>Unterminated string constant</td>
</tr>
<tr>
<td>1034</td>
<td>Unterminated comment</td>
</tr>
<tr>
<td>1035</td>
<td>Nested comment</td>
</tr>
<tr>
<td>1037</td>
<td>Invalid use of 'Me' keyword</td>
</tr>
<tr>
<td>1038</td>
<td>'Loop' without 'Do'</td>
</tr>
<tr>
<td>1039</td>
<td>Invalid 'Exit' statement</td>
</tr>
<tr>
<td>1040</td>
<td>Invalid 'For' loop control variable</td>
</tr>
<tr>
<td>1041</td>
<td>Name redefined</td>
</tr>
<tr>
<td>1042</td>
<td>Must be first statement on the line</td>
</tr>
<tr>
<td>1043</td>
<td>Can't assign to non-ByVal argument</td>
</tr>
<tr>
<td>1044</td>
<td>Can't use parens when calling a Sub</td>
</tr>
<tr>
<td>1045</td>
<td>Expected literal constant</td>
</tr>
<tr>
<td>1046</td>
<td>Expected 'In'</td>
</tr>
<tr>
<td>32766</td>
<td>True</td>
</tr>
<tr>
<td>32767</td>
<td>False</td>
</tr>
<tr>
<td>32811</td>
<td>Element not found</td>
</tr>
</tbody>
</table>
WeekdayName Function

Description

Returns a string indicating the specified day of the week.

Syntax

WeekdayName(weekday, abbreviate, firstdayofweek)

The WeekdayName function syntax has these parts:

<table>
<thead>
<tr>
<th>Part</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>weekday</td>
<td>Required. The numeric designation for the day of the week. Numeric value of each day depends on setting of the firstdayofweek setting.</td>
</tr>
<tr>
<td>abbreviate</td>
<td>Optional. Boolean value that indicates if the weekday name is to be abbreviated. If omitted, the default is False, which means that the weekday name is not abbreviated.</td>
</tr>
<tr>
<td>firstdayofweek</td>
<td>Optional. Numeric value indicating the first day of the week. See Settings section for values.</td>
</tr>
</tbody>
</table>

Settings

The firstdayofweek argument can have the following values:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vbUseSystem</td>
<td>0</td>
<td>Use National Language Support (NLS) API setting.</td>
</tr>
<tr>
<td>vbSunday</td>
<td>1</td>
<td>Sunday (default)</td>
</tr>
</tbody>
</table>
The following example uses the **WeekDayName** function to return the specified day:

```
Dim MyDate
MyDate = WeekDayName(6, True) ' MyDate contains Fri.
```
ExecuteGlobal Statement

See Also

Eval Function
Execute Statement
GetLocale Function

See Also

SetLocale Function
Locale ID (LCID) Chart
Locale

ID (LCID) Chart

See Also

GetLocale Function
SetLocale Function
SetLocale Function

See Also

GetLocale Function
Locale ID (LCID) Chart
Operator Summary

See Also

Operator Precedence
Assignment Operator

= Operator
Derived Math Functions

See Also

- Atn Function
- Cos Function
- Exp Function
- Log Function
- Sin Function
- Sqr Function
- Tan Function
WeekdayName Function

See Also

MonthName Function