Quick Reference Library

VBScript Quick Reference

Welcome to the **DevGuru VBScript Quick Reference** guide. This is an extremely comprehensive 247 page reference source that explains and gives examples of code, plus the resultant output, for all of the constants, functions, methods, properties, objects, operators, and statements that define this language.

VBScript, or by its full name, the Microsoft Visual Basic Scripting Edition language, is a simplified version of the Visual Basic and Visual Basic for Applications family of programming languages. It is also considered to be closely related to the BASIC programming language.

VBScript is a scripting language. or more precisely a "scripting environment", which can enhance HTML Web pages by making them active, as compared to a simple static display. Specifically, VBScript was created by Microsoft to use either as a client-side scripting language for the Microsoft Internet Explorer (versions 3.0 and later) or as a server-side scripting language with the Microsoft Internet Information Server (versions 3.0 and later). A primary advantage for using the server-side approach is that the VBScript is processed by the server before it is transmitted to the client. Therefore, the client only receives an HTML page and we do not have to concern ourselves as to whether the browser can interpret the VBScript. In contrast, by using the client-side approach, you purposely transfer the work load to the browser in order to reduce the work load of the server. Unfortunately, older or non-Microsoft browsers may not be able to correctly interpret and display the transmitted file. In addition to this, the source code is exposed to the browser user. On the brighter side, a client-side program can produce a more-responsive application, since user input can be processed on the client machine, and not sent back to the server for processing.

The true importance of VBScript is that it is the default language of Active Server Pages (ASP).

ASP is an exciting technology from Microsoft that is of significant value to developers. ASP extends standard HTML by adding built-in objects and server-side scripting, and by allowing access to databases and other server-side ActiveX components. All of this means that it is now even easier than ever to make your Web pages as dynamic and enticing as you desire.

For many Web-application developers, VBScript may very well be the most important programming language.

VBScript Version 5.0 was released in 1999. Certainly, the most important new feature of Version 5.0 is the ability to use the **Class** statement to create your own class objects. Other new features of interest include the **Timer** function, the **With** statement, and regular expression searching using the **RegExp** and **Match** objects.

Recently, Microsoft renamed VBScript Version 5.0 to Version 5.5 to signify that it is part of the Windows Script Version 5.5 package.

INDEX

STATEMENTS

PROPERTIES

AtEndOfLine
AtEndofStream
Attributes
- File
- Folder

Column
Count
- Dictionary
- Drives Collection
- Files Collection
- Folders Collection
- Matches Collection

FirstIndex
Description
DriveType
FirstIndex
HelpContext
HelpFile

Item
- Dictionary
- Drives Collection
- Files Collection
- Folders Collection
- Matches Collection

Global
IgnoreCase
Key
Length
Line
Number
Pattern
Path
- Drives Object
- Files Object
- Folders Object

Source
Value

OPERATORS

And
Eqv
Imp
Is
Mod
Not
Or
Xor
= (equal)
< (less-than)
> (greater-than)
<= (less-than or equal)
>= (greater-than or equal)
<> (not equal)
+ (plus)
& (concatenate)
- (minus)
/ (divide)
\ (integer divide)
* (multiply)
^ (exponentiate)<

OBJECTS

- Class
- Dictionary
- Drive
- Drives Collection
- Err
- File
- Files Collection
- FileSystemObject
- Folder
- Folders Collection
- Match
- Matches Collection
- RegExp
- TextStream

METHODS

- Add
  - Dictionary
  - FoldersCollection
- BuildPath
- Close
- Copy
  - File
  - Folder
- CopyFolder
- CreateFolder
- CreateTextFile
  - FileSystemObject
  - Folder
- Delete
  - File
  - Folder
- DeleteFile
- DeleteFolder
- DriveExists
- Execute
- Exists
- FileExists
- FolderExists
- GetAbsolutePathName
- GetBaseName
- GetDrive
- GetDriveName
- GetExtensionName
- GetFile
- GetFileName
- GetFolder
- GetParentFolderName
- GetSpecialFolder
- GetTempName
- Items
- Keys
- Move
  - File
  - Folder
- MoveFile
- MoveFolder
- OpenAsTextStream
- OpenTextFile
- Raise
- Remove
- RemoveAll
- Read
- ReadAll
- ReadLine
- Replace
- SkipLine
- Write
- WriteBlankLines
- WriteLine

FUNCTIONS

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

EVENT HANDLER

- Initialize
- Terminate

CONSTANTS

DatePart
DateSerial
DateTime
DateValue
Day
Hex
Hour
If . . . Then
Imp
Initialize
Private
Property Get
Property Let
Property Set
Public
Write
WriteBlankLines
WriteLine
Xor
Year

STATEMENT: '

Implemented in version 1.0

'

The ' statement allows you to insert comments into your code.

Code:

```
<% ' This is one of two ways to comment in VBScript %>
<% Rem This is the other way %>
```

OPERATOR: -

Implemented in version 1.0

-

The **-** operator has two uses. It is used for subtracting numbers. It also is used to indicated the sign of a number (i.e., -4.5832).

Code:

```
<% total = 5.678 - 0.234 - 23.1 %>
```

Output:

```
-17.656
```

OPERATOR: &

Implemented in version 1.0

&

The **&** operator is the preferred operator for concatenating strings. However, the **+** operator may also be used for concatenation.

Code:

```
<% cheese=" is made of blue cheese." %>
<% sentence="The moon" & cheese & " Mice like blue cheese." %>
```

Output:

```
The moon is made of blue cheese. Mice like blue cheese.
```

OPERATOR: *

Implemented in version 1.0

*****

The ***** operator is used to multiply numbers.

Code:

```
<% answer = 2 * 3.14 * .71 %>
```

Output:

```
4.4588
```

OPERATOR: /

Implemented in version 1.0

/

The **/** operator divides two numbers and returns a floating-point number.

Code:
```
<% result = 25.0 / 4.975 %>
```

Output:
```
5.0251256281407
```

Code:
```
<% result = 25 / 5 %>
```

Output:
```
5
```

OPERATOR: \

Implemented in version 1.0

****

The **** operator divides two numbers and returns an integer (fixed-point). Each number can be either floating-point or fixed-point.

The answer is truncated to the integer portion. This is equivalent to rounding towards zero for both negative and positive numbers.

Code:
```
<% result = 25 \ 4 %>
```

Output:
6

Code:
```
<% result = 35.4 \ 6.01 %>
```

Output:
5

Code:
```
<% result = 25.12345 \ 4 %>
```

Output:
6

OPERATOR: ^

Implemented in version 1.0

^

The **^** operator, which is sometimes called "hat", is used to raise a number to a power.

Code:
```
<% power = 2^2 %>
```

Output:
```
4
```

Code:
```
<% power = 4.819^1.753 %>
```

Output:
```
15.7479305850686
```

OPERATOR: +

Implemented in version 1.0

The **+** operator has two uses. It is used to add numbers together. It also may be used to concatenate (combine) strings together.

Note it is usually recommended that only the **&** operator be used for string concatenation.

Addition:

Code:
```
<% total = 5.678 + 0.234 + 23.1 %>
```

Output:
```
29.012
```

Concatenation:

Code:
```
<% cheese=" is made of blue cheese." %>
<% sentence="The moon" + cheese + " Mice like blue cheese." %>
```

Output:
```
The moon is made of blue cheese. Mice like blue cheese.
```

OPERATOR: <

Implemented in version 1.0

<

The **<** operator determines if the first expression is less-than the second expression.

Code:

```
<% 123 < 456 %>
<% "a" < "A" %>
```

Output:

```
True
False
```

OPERATOR: <=

Implemented in version 1.0

<=

The **<=** operator determines if the first expression is less-than-or-equal the second expression.

Code:

```
<% 123 <= 456 %>
<% "a" <= "A" %>
```

Output:

```
True
False
```

OPERATOR: =

Implemented in version 1.0

=

The operator is used to assign a value to a variable, or to compare strings and numbers, or to print out the value of a variable.

When used for assignment, a single variable on the left side of the **=** operator is given the value determined by one or more variables and/or expressions on the right side.

Code:
```
<% mynumber = 5.0 + 7.9*(444.999 / 456.9) %>
<% mystring = "The moon is 2160 miles in diameter." %>
<% myvar = xxx + yyy + zzz %>
<% myarray(40) = 12345 %>
```

The **=** operator can also compare two strings or two numbers and see if they are the same. For strings, the comparison is case sensitive.

Code:
```
<% ="Apple"="Apple" %>
<% ="aPPle"="ApplE" %>

<% =123=123 %>
<% =5.67=98.7 %>
```

Output:
```
True
False

True
False
```

When the **=** operator is used to print out the value of a variable, the statement must be enclosed by its own set of <% %>. There can blank spaces between the **=** operator and the variable.

Code:
```
<% myfish = "Neon tetra" %>
<% =myfish %>
<% = myfish %>
```

Output:
```
Neon tetra
Neon tetra
```

OPERATOR: >

Implemented in version 1.0

>

The **>** operator determines if the first expression is greater-than the second expression.

Code:

```
<% 123 > 456 %>
<% "a" > "A" %>
```

Output:

```
False
True
```

OPERATOR: >=

Implemented in version 1.0

>=

The **>=** operator determines if the first expression is greater-than-or-equal the second expression.

Code:

```
<% 123 >= 456 %>
<% "a" >= "A" %>
```

Output:

```
False
True
```

OPERATOR: <>

Implemented in version 1.0

<>

The **<>** operator is called the not equal or inequality operator.

Code:

```
<%
If 128 <> 777 Then
   Response.Write("Not equal")
End If
%>
```

Output:

Not equal

FUNCTION: Abs()

Implemented in version 1.0

Abs(Number)

The **Abs** function returns the absolute value for a number.

Negative numbers become positive. Positive numbers remain positive.

Code:
```
<% =Abs(-127.89) %>
```

Output:
127.89

Code:
```
<% =Abs(127.89) %>
```

Output:
127.89

OPERATOR: And

Implemented in version 1.0

And

The **And** operator is used to perform a logical conjunction on two expressions, where the expressions are **Null**, or are of **Boolean** subtype and have a value of **True** or **False**.

The **And** operator can also be used a "bitwise operator" to make a bit-by-bit comparison of two integers. If both bits in the comparison are 1, then a 1 is returned. Otherwise, a 0 is returned.

When using the **And** to compare **Boolean** expressions, the order of the expressions is not important.

Code:
```
<% =True And True %>
<% =True And False %>
<% =False And True %>
<% =False And False %>
<% =True And Null %>
<% =Null And True %>
<% =False And Null %>
<% =Null And False %>
<% =Null And Null %>
```

Output:
```
True
False
False
False
(Null output)
(Null output)
False
False
(Null output)
```

Code:
```
<% anyexpression = True %>
<% someexpression = False %>
<% =anyexpression And someexpression %>
```

Output:
```
False
```

In this example, the **And** performs a bitwise comparison on the 1 (in binary 001) and the 2 (in binary 010), and returns a 0 (in binary 000).

Code:
```
<%
Expression1 = 1
Expression2 = 2
Result = Expression1 And Expression2
Response.Write "Result = " & Result
```

```
%>
```

Output:

```
Result = 0
```

FUNCTION: Array( )

Implemented in version 2.0

Array(List)

The **Array** function is used to create a static one-dimension array. You cannot declare a dynamic array using the **Array** function.

Note that the first element in an array is always labeled zero, for example, myarray(0).

The **List** argument is a listing of values that will become the elements of the array.

Code:
```
<% myarray = array("A", "B", "C", "D") %>
<% =myarray(0) %>
<% =myarray(1) %>
<% =myarray(2) %>
<% =myarray(3) %>
```

Output:
```
A
B
C
D
```

Code:
```
<% myarray = array(111, 222, 333, 444, 555) %>
<% =myarray(0) %>
<% =myarray(1) %>
<% =myarray(2) %>
<% =myarray(3) %>
<% =myarray(4) %>
```

Output:
```
111
222
333
444
555
```

A dynamic array is declared using the Dim and the ReDim statements. First, you use the **Dim** statement to declare the dynamic array by using empty parenthesis. Then, at a later point in your program, you use the **ReDim** statement to declare the number of elements. In fact, you can re-declare a dynamic array as many times as you desire.

Code:
```
<%
Dim SomeArray()
...
ReDim SomeArray(22)
...
```

```
ReDim SomeArray(500)
%>
```

Arrays can have up to sixty dimensions. If you wish to create a multiple-dimension array, you need to declare it using the Dim statement. For example, the following code creates a three dimensional array. The first dimension has 23 elements, the second dimension has 15 elements and the third dimension has 201 elements. Therefore, there are a total of 23x15x201 = 69345 elements in this array.

The number of dimensions that you can create is limited by the available memory. If you exceed the available memory while declaring an array, you will get an error message.

Code:

```
<% Dim ThreeDimensionArray(22, 14, 200) %>
```

STATEMENT: Dim

Implemented in version 1.0

Dim

The **Dim** statement allows you to explicitly declare one or more new variables and to allocate storage (memory) space.

While you do not have to use **Dim** to create new variables in VBScript, the wise programmer prefers to use **Dim**. In fact, many programmer purposely include the Option Explicit statement in all of their VBScript programs which mandates that all variables be explicitly declared.

Code:
```
<% Dim myvariable %>
<% Dim OneVar, TwoVar, ThreeVar, FourVar, MoreVar, EvenMoreVar %>
```

Dim can also be used to create static (fixed) and dynamic arrays.

A static array has the number of elements declared by the **Dim** statement. However, you must remember that the elements in an array are numbered starting at zero. Consider the following **Dim** declaration. It creates a static array containing six elements that are numbered 0, 1, 2, 3, 4 and 5.

Code:
```
<% Dim SixElementArray(5) %>
```

A dynamic array is declared using empty parentheses. At a later point in your program, you can use the ReDim statement to declare the number of dimensions and elements. In fact, you can redeclare a dynamic array as many times as you desire.

Code:
```
<%
Dim SomeArray()
...
ReDim SomeArray(22)
...
ReDim SomeArray(500)
%>
```

Arrays can also have up to sixty dimensions. For example, the following code creates a three dimensional array. The first dimension has 23 elements, the second dimension has 15 elements and the third dimension has 201 elements. Therefore, there are a total of 23x15x201 = 69345 elements in this array.

The number of dimensions that you can create is limited by the available memory. If you exceed the available memory while declaring an array, you will get an error message.

Code:
```
<% Dim ThreeDimensionArray(22, 14, 200) %>
```

STATEMENT: Option Explicit

Implemented in version 1.0

Option Explicit

The **Option Explicit** statement forces the explicit declaration of all variables using the **Dim, Private, Public,** or **ReDim** statements.

In a long program, this statement prevents the accidental reuse of the name of a previously declared variable. Also, if you mistype a declared variable's name or try to use an undeclared variable, an error message is generated.

Note that the **Option Explicit** statement must be placed at the top of the code before any other VBScript commands or any HTML code.

Code:

```
<% Option Explicit %>
< HTML >
< HEAD >
< TITLE > EXAMPLE < /TITLE >
< /HEAD >
< BODY >
<% Dim myvariable = 123.456  yourvar = 0 %>
< /BODY >
< /HTML >
```

STATEMENT: ReDim

ReDim

The **ReDim** statement allows you to formally declare, and to later redeclare as many times as you need, the size (and hence the memory space allocation) for a dynamic array that was originally declared using either a **Dim, Private** or **Public** statement.

The first time you use **ReDim** to declare an array, you can create either a single or a multiple dimension array. However, after you have set the number of the dimensions, you cannot later go back and change the number of the dimensions. Also, once you have created a multi-dimension array, you can only redeclare the size of the last element.

If you make the array bigger, you can use the keyword **Preserve** to protect all of the existing elements. If you make the array smaller, you will lose part of the array elements even if you use **Preserve**. Do not use **Preserve** the first time that you **ReDim** an array since it will prohibit setting multiple dimensions.

In this example, we declare myarray() to have a single dimensions and then resize the element. When we resize from 999 down to 9, we lose the data in the other 990 elements.

Code:

```
<% Dim myarray() %>
<% ReDim myarray(5) %>
<% ReDim Preserve myarray(999) %>
<% ReDim Preserve myarray(9) %>
```

In this example, we declare myarray() to have three dimensions and then resize the last element. When we resize the last element from 999 down to 9, we lose the data in the other 990 elements.

Code:

```
<% Dim myarray() %>
<% ReDim myarray(20, 10, 99) %>
<% ReDim Preserve myarray(20, 10, 999) %>
<% ReDim Preserve myarray(20, 10, 9) %>
```

FUNCTION: Asc()

Implemented in version 1.0

Asc(String)

The **Asc** function returns the ANSI character code value for the first character in a string.

In these two examples, note that the ANSI value is returned only for the "a".

Code:
```
<% =Asc("abcde fghij klmno pqrst uvwxyz") %>
```

Output:
97

Code:
```
<% =Asc("a") %>
```

Output:
97

Property: TextStream.AtEndOfLine

Implemented in version 2.0

object.AtEndOfLine

The AtEndOfLine property returns a Boolean value. If the file pointer is positioned immediately before the file's end-of-line marker, the value is **True**. Otherwise, the value is **False**. The **TextStream** object must be open for reading or an error will occur when retreiving this property this method.

The following code uses the **AtEndOfLine** property to get the last character in the first line of text in a file.

Code:

```
<%
dim filesys, text, readfile, contents
set filesys = CreateObject("Scripting.FileSystemObject")
Set text = filesys.CreateTextFile("c:\somefile2.txt")
text.Write "Find the last character in the text file"
text.close
set readfile = filesys.OpenTextFile("c:\somefile2.txt", 1, false)
do while readfile.AtEndOfLine <> true
    contents = readfile.Read(1)
loop
readfile.close
Response.Write "The last character in the text file is '" & contents & "'."
%>
```

Output:

```
"The last character in the text file is 'e'."
```

Property: TextStream.AtEndOfStream

Implemented in version 2.0

object.AtEndOfStream

The **AtEndOfStream** property returns a Boolean value. If the file pointer is positioned at the end of the file, the value is **True**. Otherwise, the value is **False**. The **TextStream** object must be open for reading or an error will occur when using this method.

The following code uses the **AtEndOfStream** property to get the last character in text file.

Code:

```
<%
dim filesys, text, readfile, contents
set filesys = CreateObject("Scripting.FileSystemObject")
Set text = filesys.CreateTextFile("c:\somefile2.txt")
text.Write "Find the last character in the text file"
text.close
set readfile = filesys.OpenTextFile("c:\somefile2.txt", 1, false)
do while readfile.AtEndOfStream <> true
    contents = readfile.Read(1)
loop
readfile.close
Response.Write "The last character in the text file is '" & contents & "'."
%>
```

Output:

```
"The last character in the text file is 'e'."
```

FUNCTION: Atn( )

Implemented in version 1.0

Atn(Number)

The **Atn** function returns the arctangent for a number.

Code:
```
<% =Atn(45.0) %>
```

Output:
1.54857776146818

You can also use a negative number.

Code:
```
<% =Atn(-45.0) %>
```

Output:
-1.54857776146818

METHOD: FileSystemObject.BuildPath

Implemented in version 2.0

[newfullpath =]object.BuildPath(path, name)

This method is used to append a name onto an existing path. The format of the new path is 'existingpath\name'.

Code:

```
<%
dim filesys, newfullpath
set filesys=CreateObject("Scripting.FileSystemObject")
newfullpath = filesys.BuildPath(c:\inetpub\wwwroot, "images")
%>
```

Output:

```
"c:\inetpub\wwwroot\images"
```

STATEMENT: Call

Implemented in version 1.0

Call

The **Call** statement can be used for calling a function or subroutine.

Note that the use of **Call** is optional. In other words, you can call a function or subroutine by simply stating the function's or subroutine's name. If you use **Call** and if there are one or more arguments to be passed you must enclose all of the arguments in parentheses. If you do not use **Call**, do not use the parentheses.

If you use **Call** to call a function, the return value of the function is discarded.

Code:

```
<% Call afunction %>

<% afunction %>

<% Call myfunction(myvar1, myvar2, mystring, myarray) %>

<% myfunction myvar1, myvar2, mystring, myarray %>

<% Call anysubroutine(anum, astring) %>

<% anothersubroutine %>
```

FUNCTION: CBool( )

Implemented in version 1.0

You can determine the expression subtype by using the function **VarType( )**.

CBool(Number)

The **CBool** function converts any number to the variant of subtype **Boolean**.

The output for **Boolean** is either true or false. If the conversion is successful, the output will be true. If the conversion fails, the output will be false or an error message.

Code:
```
<% anynumber=7.77 %>
<% =CBool(anynumber) %>
```

Output:
True

Code:
```
<% notnumber=abc %>
<% =CBool(notnumber) %>
```

Output:
False

FUNCTION: CByte( )

Implemented in version 1.0

You can determine the expression subtype by using the function **VarType( )**.

CByte(Number)

The **CByte** function converts any number between 0 and 255 to the variant of subtype **Byte**.

Code:
```
<% anynumber=(9.876) %>
<% =CByte(anynumber) %>
```

Output:
10

Code:
```
<% anynumber=(255) %>
<% =CByte(anynumber) %>
```

Output:
255

FUNCTION: CCur( )

Implemented in version 1.0

You can determine the expression subtype by using the function **VarType( )**.

CCur(Number)

The **CCur** function converts any number or numeric string to the variant of subtype **Currency**.

Converts to currency values ranging from
-922,337,203,685,477.5808 to 922,337,203,685,477.5807

Code:
```
<% anynumber=(12345) %>
<% =CCur(any_number) %>
```

Output:
12345

Code:
```
<% any_numeric_string=("98765") %>
<% =CCur(any_numeric_string) %>
```

Output:
98765

This function rounds off to 4 decimal places.

Code:
```
<% anynumber=(55555.123456) %>
<% =CCur(anynumber) %>
```

Output:
55555.1235

FUNCTION: CDate( )

Implemented in version 1.0

You can determine the expression subtype by using the function **VarType( )**.

CDate(Date)

The **CDate** function converts any valid date and time expression to the variant of subtype **Date**.

The default date output will be of the format: **/**/** The default time output will be of the format: **:**:** *M Converted date values can range from January 1, 100 to December 31, 9999

Code:
```
<% anydate = "June 26, 1943" %>
<% =CDate(anydate) %>
```

Output:
6/26/43

Code:
```
<% anydate = #6/26/43# %>
<% =CDate(anydate) %>
```

Output:
6/26/43

Code:
```
<% anytime="2:23:59 PM" %>
<% =CDate(anytime) %>
```

Output:
2:23:59 PM

FUNCTION: CDbl( )

Implemented in version 1.0

You can determine the expression subtype by using the function **VarType( )**.

CDbl(Number)

The **CDbl** function converts any number to the variant of subtype **Double**.

Converted values can range from
-1.79769313486232E308 to -4.94065645841247E-324 for negative values and
4.94065645841247E-324 to 1.79769313486232E308 for positive values.

Code:

```
<% anynumber=1234.5 %>
<% =CDbl(anynumber) %>
```

Output:

```
1234.5
```

FUNCTION: Chr()

Implemented in version 1.0

Chr(ANSIvalue)

The **Chr** function converts an ANSI character code value to a character.

Code:
```
<% =Chr(98) %>
```

Output:
b

FUNCTION: CInt( )

Implemented in version 1.0

You can determine the expression subtype by using the function **VarType( )**.

CInt(Number)

The **CInt** function converts any number to the variant of subtype **Integer**.

Converts to values ranging from -32,768 to 32,767 The number is rounded off.

Code:

```
<% anynumber=1234.567 %>
<% =CInt(anynumber) %>
```

Output:

1235

FUNCTION: CLng( )

Implemented in version 1.0

You can determine the expression subtype by using the function **VarType( )**.

CLng(Number)

The **CLng** function converts any number to the variant of subtype **Long**, with floating-point numbers rounded.

Converts to integers ranging from -2,147,483,648 to 2,147,483,647

Code:

```
<% anynumber=1234.6 %>
<% =CLng(anynumber) %>
```

Output:

```
1235
```

OBJECT: Class

Implemented in version 5.0

The term, **Class**, refers to any object that you create using the **Class ... End Class** statement. This ability to create your own object and to perform operations upon it using any valid VBScript code, represents a significant expansion of the versatility of the VBScript language.

Note that **Class** is a keyword and you may not use it as a class name.

After you have used the **Class** statement to declare a class name, then, as demonstrated below, you may declare an instance of that **Class** (i.e., object) using **Set** and **New**.

Code:

```
<%
Dim YourObjectVariable
Set YourObjectVariable = New YourClassName
%>
```

STATEMENT: Class

Implemented in version 5.0

Class ... End Class

The **Class** statement block is used to create a **Class** object. You can only create (name) one **Class** object with each **Class** statement. This ability to create your own **Class** is a significant expansion of the usefulness of the VBScript language.

Within the block of the **Class** statement you can declare the members of the class, which are variables, methods, and properties. Methods of the class are implemented by defining a **Sub** or **Function** procedure, while properties are defined through the use of **Property Get**, **Property Let**, and **Property Set** statements. Any member of a class may be declared as either **Public** or **Private**, with a **Public** declaration being the default state. **Private** members of a class are only accessable by other members of the same class, while **Public** members are accessable by anything, inside or outside of the scope of the class.

The **Class** statement must always end with an **End Class**.

Example:

```
<%
Class DevGuruProducts

  ' Creating a private property using Get, Let, Set
  Private mstrName
  ' Get
  Public Property Get CustomerName()
    CustomerName = mstrName
  End Property
  ' Let
  Public Property Let CustomerName(strName)
    mstrName = strName
  End Property
  ' Set
  Public Property Set Guru(objGuru)
    Private mobjGuru
    Set mobjGuru = objGuru
  End Property

  ' Creating a private method using a function
  Private Function DevGuruProductName(intProduct)
    Select Case intProduct
    Case 1
      DevGuruProductName = "dgCharge"
    Case 2
      DevGuruProductName = "dgList"
    Case 3
      DevGuruProductName = "dgReport"
  End Function

End Class
%>
```

CONSTANTS: COLOR

Implemented in version 2.0

Color Constants

You may use either the CONSTANT (left column) or the VALUE (center column) in your code and get the same results.

CONSTANT	VALUE	COLOR
VBBlack	&h00	Black
VBRed	&hFF	Red
VBGreen	&hFF00	Green
VBYellow	&hFFFF	Yellow
VBBlue	&hFF0000	Blue
VBMagenta	&hFF00FF	Magenta
VBCyan	&hFFFF00	Cyan
VBWhite	&hFFFFFF	White

Property: TextStream.Column

Implemented in version 2.0

object.Column

The **Column** property returns the current position of the file pointer within the current line. Column 1 is the first character in each line.

The follwing code uses the **Column** property to get the column number of the last character in a text file.

Code:

```
<%
dim filesys, text, readfile, contents
set filesys = CreateObject("Scripting.FileSystemObject")
Set text = filesys.CreateTextFile("c:\somefile2.txt")
text.Write "Find the last character in the text file"
text.close
set readfile = filesys.OpenTextFile("c:\somefile2.txt", 1, false)
do while readfile.AtEndOfStream <> true
   contents = readfile.Read(1)
   col = readfile.Column -1
loop
readfile.close
Response.Write "The last character in the text file is '" & contents & "' and its_
                        column number is '" & col & "'."
%>
```

Output:

"The last character in the text file is 'e' and its column number is '40'."

Property: Dictionary.CompareMode

Implemented in version 2.0

object.CompareMode[= comparison_mode]

The **CompareMode** property is used to set and return the key's string comparison mode which determines how keys are matched while looking up or searching. Keys can be treated as case-sensitive or case-insensitive.

To do the comparison, you use the Comparison Constants. You may use either the CONSTANT (left column) or the VALUE (center column) in your code and get the same results.

CONSTANT	VALUE	DESCRIPTION
VBBinaryCompare	0	binary Comparison
VBTextCompare	1	Text Comparison
VBDataBaseCompare	2	Compare information inside database

In the following example the Add method fails on the last line because the key "b" already exists. If **CompareMode** were set to VBBinaryCompare a new key "B" (upper case) with a value of "Bentley" will be added to the dictionary.

Code:

```
<%
Dim d
Set d = CreateObject("Scripting.Dictionary")
d.CompareMode = VBTextCompare
d.Add "a", "Alvis"
d.Add "b", "Buick"
d.Add "c", "Cadillac"
d.Add "B", "Bentley"   'fails here.
%>
```

Output:

```
"Alvis"
"Buick"
"Cadillac"
```

CONSTANTS: COMPARISON

Implemented in version 2.0

Comparison Constants

You may use either the CONSTANT (left column) or the VALUE (center column) in your code and get the same results.

CONSTANT	VALUE	DESCRIPTION
VBBinaryCompare	0	Binary comparison
VBTextCompare	1	Text Comparison
VBDataBaseCompare	2	Compare information inside database

STATEMENT: Const

Implemented in version 1.0

Const

The **Const** statement allows you to declare your own constants.

Once you declare a constant, you cannot change the value of that constant. If you try to change the value, you will get an error message (illegal assignment).

Code:

```
<% Const myconstant = 71.348 %>
Our current interest rate is <% =myconstant %> % per year.
```

Output:

```
Our current interest rate is 71.348 % per year.
```

You can declare several constants at the same time.

Code:

```
<% Const stringconstant = "Please send money!"  someconstant = .277  tigre = "Jaguar" %>
```

METHOD: FileSystemObject.CopyFile

Implemented in version 2.0

object.CopyFile source, destination [, overwrite]

This method lets us copy one or more files from one location (the source) to another (destination). Wildcards can be used within the **source** string, providing it is the last component in the path, to enable the copying of multiple files, but cannot be used in the **destination** string. Note that if the **source** does contain wildcards, it is automatically assumed that the **destination** is an existing folder and any matching files are copied to it.

The **overwrite** parameter is a Boolean. **True** allows the overwriting of existing files in the destination, providing that the permissions on the destination allow it (if the destination is set as read-only, the **CopyFile** method will fail). An overwrite setting of **False** prevents existing files in the destination from being overwritten.

It is recommended that you use the **FileExists** method when copying or moving a file - if a **source** file doesn't exist you'll get an error.

Code:

```
<%
dim filesys
set filesys=CreateObject("Scripting.FileSystemObject")
If filesys.FileExists("c:\sourcefolder\anyfile.html") Then
  filesys.CopyFile "c:\sourcefolder\anyfile.html", "c:\destfolder\"
End If
%>
```

METHOD: FileSystemObject.CopyFolder

Implemented in version 2.0

object.CopyFolder source, destination [, overwrite]

This method lets us copy one or more folders, and all contents, from one location (the source) to another (destination). Wildcards can be used within the **source** string, providing it is the last component in the path, to enable the copying of multiple files, but cannot be used in the **destination** string. Note that if the **source** does contain wildcards, it is automatically assumed that the **destination** is an existing folder and any matching folders are copied to it.

The **overwrite** parameter is a Boolean. **True** allows the overwriting of existing files in the destination, providing that the permissions on the destination allow it (if the destination is set as read-only, the **CopyFolder** method will fail). An overwrite setting of **False** prevents existing files in the destination from being overwritten.

It is recommended that you use the **FolderExists** method when copying or moving a folder - if a **source** folder doesn't exist you'll get an error.

Code:

```
<%
dim filesys
set filesys=CreateObject("Scripting.FileSystemObject")
If filesys.FolderExists("c:\sourcefolder\website") Then
  filesys.CopyFolder "c:\sourcefolder\website", "c:\destfolder\"
End If
%>
```

FUNCTION: Cos( )

Implemented in version 1.0

Cos(Number)

The **Cos** function returns the cosine for a number (angle).

Code:
```
<% =Cos(45.0) %>
```

Output:
0.52532198881773

You can also use a negative number (angle).

Code:
```
<% =Cos(-45.0) %>
```

Output:
0.52532198881773

OBJECT: Dictionary

Implemented in version 2.0

The **Dictionary** object stores name/value pairs (referred to as the key and item respectively) in an array. The key is a unique identifier for the corresponding item and cannot be used for any other item in the same **Dictionary** object.

The following code creates a **Dictionary** object called "cars", adds some key/item pairs, retrieves the item value for the key 'b' using the **Item** property and then outputs the resulting string to the browser.

Code:

```
<%
Dim cars
Set cars = CreateObject("Scripting.Dictionary")
cars.Add "a", "Alvis"
cars.Add "b", "Buick"
cars.Add "c", "Cadillac"
Response.Write "The value corresponding to the key 'b' is " & cars.Item("b")
%>
```

Output:

"The value corresponding to the key 'b' is Buick"

PROPERTIES

CompareMode Property

The **CompareMode** property is used to set and return the key's string comparison mode which determines how keys are matched while looking up or searching.

Syntax: **object.CompareMode[= comparison_mode]**

Count Property

The **Count** property is used to determine the number of key/item pairs in the **Dictionary** object.

Syntax: **object.Count**

Item Property

The **Item** property allows us to retreive the value of an item in the collection designated by the specified **key** argument and also to set that value by using **itemvalue**.

Syntax: **object.Item(key) [= itemvalue]**

Key Property

The **Key** property lets us change the key value of an existing key/item pair.

Syntax: **object.Key(keyvalue) = newkeyvalue**

METHODS

Add Method

The **Add** method is used to add a new key/item pair to a **Dictionary** object.

Syntax: **object. Addkeyvalue, itemvalue**

Exists Method

The **Exists** method is used to determine whether a key already exists in the specified **Dictionary**. Returns **True** if it does and **False** otherwise.

Syntax: **object.Exists(keyvalue)**

Items Method

The **Items** method is used to retreive all of the items in a particular **Dictionary** object and store them in an array.

Syntax: **[arrayname =]object.Items**

Keys Method

The **Keys** method is used to retreive all of the keys in a particular **Dictionary** object and store them in an array.

Syntax: **[arrayname =]object.Keys**

Remove Method

The **Remove** method is used to remove a single key/item pair from the specified **Dictionary** object.

Syntax: **object. Remove(keyvalue)**

RemoveAll Method

The **RemoveAll** method is used to remove all the key/item pairs from the specified **Dictionary** object.

Syntax: **object.RemoveAll**

Property: Dictionary.Item

Implemented in version 2.0

The **Item** property sets or returns the value of an item for the specified key and Dictionary.

Keyvalue is the value of the key associated with the item being returned or set and **itemvalue** is an optional value which sets the value of the item associated with that key.

If the specified key is not found when attempting to change an item, a new key/item pair is added to the dictionary. If the key is not found while trying to return an existing item, a new key is added and the item value is left empty.

Code:

```
<%
Set d = CreateObject("Scripting.Dictionary")
d.Add "a", "Alvis" 'Add some keys and items.
d.Add "b", "Buick"
d.Add "c", "Cadillac"
d.Item("c") = "Corvette"
Response.Write "Value associated with key 'c' has changed to " d.Item("c")
%>
```

Output:

"Value associated with key 'c' has changed to Corvette"

Property: Dictionary.Key

Implemented in version 2.0

object. Keys

The **Key** property lets us change the key value of an existing key/item pair.

Keyvalue is the current key value and **newkeyvalue** is the value you want to change it to.

If the specified key is not found when attempting to change it, a runtime error will occur

Code:

```
<%
Set d = CreateObject("Scripting.Dictionary")
d.Add "a", "Alvis" 'Add some keys and items.
d.Add "b", "Buick"
d.Add "c", "Corvette"
d.Key("c") = "d"
Response.Write "The key associated with the item " & d.Item("d") & " has been changed to 'd'."
%>
```

Output:

```
The key associated with the item Corvette has been changed to 'd'.
```

METHOD: Dictionary.Add

Implemented in version 2.0

object.Add keyvalue, itemvalue

The **Add** method is used to add a new key/item pair to a **Dictionary** object. Be aware that an error will occur if the key already exists.

The following example adds several key/item pairs to a **Dictionary** object called 'guitars'.

Code:

```
<%
dim guitars
set guitars=CreateObject("Scripting.Dictionary")
guitars.Add "e", "Epiphone"
guitars.Add "f", "Fender"
guitars.Add "g", "Gibson"
guitars.Add "h", "Harmony"
%>
```

Output:

```
"Epiphone"
"Fender"
"Gibson"
"Harmony"
```

METHOD: Dictionary.Exists

Implemented in version 2.0

object.Exists (keyvalue)

The **Exists** method is used to determine whether a key already exists in the specified **Dictionary**. Returns **True** if it does and **False** otherwise.

The following code uses the **Exists** method to see if the key with the value of 'g' exists in the **Dictionary** object and, if it does, changes its corresponding item value.

Code:

```
<%
dim guitars
set guitars=CreateObject("Scripting.Dictionary")
guitars.Add "e", "Epiphone"
guitars.Add "f", "Fender"
guitars.Add "g", "Gibson"
guitars.Add "h", "Harmony"
If guitars.Exists("g") Then
   guitars.Item("g") = "Guild"
End If
%>
```

Output:

```
"Epiphone"
"Fender"
"Guild"
"Harmony"
```

METHOD: Dictionary.Items

Implemented in version 2.0

[arrayname =] object. Items

The **Items** method can be used to retreive all the items in a **Dictionary** object and place them in an array. We can then iterate through the items using a **For**....**Next** loop.

The following code demonstrates this, using the **Dictionary** object's **Count** property to determine the amount of key/item pairs.

Code:

```
<%
dim guitars, arrayItems, i
set guitars=CreateObject("Scripting.Dictionary")
guitars.Add "e", "Epiphone"
guitars.Add "f", "Fender"
guitars.Add "g", "Gibson"
guitars.Add "h", "Harmony"
arrayItems = guitars.Items
for i = 0 to guitars.Count -1
    Response.Write "<br>" & arrayItems(i)
next
%>
```

Output:

```
"Epiphone"
"Fender"
"Gibson"
"Harmony"
```

METHOD: Dictionary.Keys

Implemented in version 2.0

[arrayname =] object. Keys

The **Keys** method can be used to retreive all the keys in a **Dictionary** object and place them in an array. We can then iterate through the keys using a **For**....**Next** loop.

The following code demonstrates this, using the **Dictionary** object's **Count** property to determine the amount of key/item pairs.

Code:

```
<%
dim guitars, arrayKeys, i
set guitars=CreateObject("Scripting.Dictionary")
guitars.Add "e", "Epiphone"
guitars.Add "f", "Fender"
guitars.Add "g", "Gibson"
guitars.Add "h", "Harmony"
arrayItems = guitars.Keys
for i = 0 to guitars.Count -1
    Response.Write "<br>Key = " & arrayItems(i) & "Item = "  & _
guitars.Item(arrayKeys(i))
next
%>
```

Output:

```
"Key = e Item = Epiphone"
"Key = f  Item = Fender"
"Key = g  Item = Gibson"
"Key = h  Item = Harmony"
```

METHOD: Dictionary.Remove

Implemented in version 2.0

object.Remove (keyvalue)

The **Remove** method is used to remove a single key/item pair from the specified **Dictionary** object.

Code:

```
<%
dim guitars
set guitars=CreateObject("Scripting.Dictionary")
guitars.Add "e", "Epiphone"
guitars.Add "f", "Fender"
guitars.Add "g", "Gibson"
guitars.Add "h", "Harmony"
guitars.Remove("g")
%>
```

Output:

```
"Epiphone"
"Fender"
"Harmony"
```

METHOD: Dictionary.RemoveAll

Implemented in version 2.0

object.RemoveAll

The **RemoveAll** method is used to remove all key/item pairs from the specified **Dictionary** object.

Code:

```
<%
dim guitars
set guitars=CreateObject("Scripting.Dictionary")
guitars.Add "e", "Epiphone"
guitars.Add "f", "Fender"
guitars.Add "g", "Gibson"
guitars.Add "h", "Harmony"
guitars.RemoveAll
guitars.Add "i", "Ibanez"
%>
```

Output:

```
"Ibanez"
```

OBJECT: Drives Collection

Implemented in version 3.0

The **Drives** collection is the collection of all the disk drives available on the system. This collection is obtained through the **Drives** property of the **FileSystemObject** object.

The following code gets the **Drives** collection, creates a list of available drives and displays the results in the browser.

Code:

```
<%
Dim filesys, drv, drvcoll, drvlist, vol
Set filesys = CreateObject("Scripting.FileSystemObject")
Set drvcoll = filesys.Drives
For Each drv in drvcoll
  drvlist = drvlist & drv.DriveLetter
  If drv.IsReady Then
     vol = drv.VolumeName
  End If
  drvlist = drvlist & vol & ", "
Next
Response.Write "Drives available on this system are " & drvlist
%>
```

Output:

"Drives available on this system are A, C, D, E, "

PROPERTIES

Count Property

Returns an integer that tells us how many **Drive** objects there are in the collection (the number of local and remote drives available).

Syntax: **object.Count**

Item Property

The **Item** property allows us to retreive the value of an item in the collection designated by the specified **key** argument and also to set that value by using **itemvalue**.

Syntax: **object.Item(key) [= itemvalue]**

Property: Drives Collection.Item

Implemented in version 3.0

object.Item (drive)

This property returns an **Item** from the collection relating to a specified key (drive name).

The following code shows how to retrieve an item from a **Drives** collection.

```
<%
Code:
Dim filesys, drvcoll, label
Set filesys = CreateObject("Scripting.FileSystemObject")
Set drvcoll = filesys.Drives
label = drvcoll.Item("c")
Response.Write "The volume label for drive c is '" & it & "'."
%>
```

Output:
"The volume label for drive c is 'C:'."

OBJECT: Files Collection

Implemented in version 3.0

The **Files** collection contains a set of **File** objects and is usually stored as a property of another object, such as a **Folder** object.

The following code demonstrates how to get a **Files** collection and list the contents in the browser.

Code:

```
<%
Dim filesys, demofolder, fil, filecoll, filist
Set filesys = CreateObject("Scripting.FileSystemObject")
Set demofolder = filesys.GetFolder("foldername")
Set filecoll = demofolder.Files
For Each fil in filecoll
   filist = filist & fil.name
   filist = filist & "<BR>"
Next
Response.Write filist
%>
```

PROPERTIES

Count Property
Returns an integer that tells us how many **File** objects there are in the collection.

Syntax: **object.Count**

Item Property
The **Item** property allows us to retreive the value of an item in the collection designated by the specified **key** argument and also to set that value by using **itemvalue**.

Syntax: **object.Item(key) [= itemvalue]**

Property: Files Collection.Item

Implemented in version 3.0

object.Item ("filename")

This property returns an **Item** from the collection relating to a specified key (file name).

The following code shows how to retrieve an item from a **Files** collection.

```
<%
Code:
Dim filesys, demofolder, filecoll, selectfile
Set filesys = CreateObject("Scripting.FileSystemObject")
Set demofolder = filesys.GetFolder("foldername")
Set filecoll = demofolder.Files
selectfile = filecoll.Item("filename")
%>
```

OBJECT: Folders Collection

Implemented in version 2.0

When using an instance of the **Folder** object, its **SubFolder** property returns a **Folders** collection consisting of all the subfolders (**Folder** objects) in that folder.

The following code illustrates how to get a **Folders** collection and display its contents in the browser.

Code:

```
<%
Dim filesys, demofolder, subfol, folcoll, folist
Set filesys = CreateObject("Scripting.FileSystemObject")
Set demofolder = filesys.GetFolder(folderspec)
Set folcoll = demofolder.SubFolders
For Each subfol in folcoll
   folist = folist & subfol.Name
   folist = folist & "<BR>"
Next
Response.Write folist
%>
```

PROPERTIES

Count Property
Returns an integer that tells us how many **Folder** objects there are in the collection.

Syntax: **object.Count**

Item Property
The **Item** property allows us to retreive the value of an item in the collection designated by the specified **key** argument and also to set that value by using **itemvalue**.

Syntax: **object.Item(key) [= itemvalue]**

METHODS

Add Method
This method is used to add a new **Folder** to a **Folders** collection.

Syntax: **object.Add("foldername")**

Property: Folders Collection.Item

Implemented in version 2.0

object.Item ("foldername")

This property returns an **Item** from the collection relating to a specified key (foldername).

The following code shows how to retrieve an item from a **Folders** collection.

```
<%
Code:
Dim filesys, demofolder, folcoll, selectfolder
Set filesys = CreateObject("Scripting.FileSystemObject")
Set demofolder = filesys.GetFolder("foldername")
Set folcoll = demofolder.SubFolders
selectfolder = folcoll.Item("foldername")
%>
```

METHOD: Folders Collection.Add

Implemented in version 2.0

object.Add ("foldername")

This method is used to add a new **Folder** to a **Folders** collection. Note that, if the **("foldername")** already exists, you will get an error.

```
<%
Code:
Dim filesys, demofolder, folcoll
Set filesys = CreateObject("Scripting.FileSystemObject")
Set demofolder = filesys.GetFolder("foldername")
Set folcoll = demofolder.SubFolders
folcoll.Add("foldername")
%>
```

OBJECT: Matches Collection

Implemented in version 5.0

The **Matches Collection** is a collection of objects that contains the results of a search and match operation that uses a regular expression.

Simply put, a regular expression is a string pattern that you can compare against all or a portion of another string. However, in all fairness, be warned that regular expressions can get very complicated.

The **RegExp** object can be used to search for and match string patterns in another string. A **Match** object is created each time the **RegExp** object finds a match. Since, zero or more matches could be made, the **RegExp** object actually return a collection of **Match** objects which is refered to as a **Matches** collection.

The following code is a simplier, working version of a program published by Microsoft. Note how the **For Each ... Next** statement is used to loop through the **Matches** collection.

Code:

```
<%
'this sub finds the matches
Sub RegExpTest(strMatchPattern, strPhrase)
   'create variables
   Dim objRegEx, Match, Matches, StrReturnStr
   'create instance of RegExp object
   Set objRegEx = New RegExp

   'find all matches
   objRegEx.Global = True
   'set case insensitive
   objRegEx.IgnoreCase = True
   'set the pattern
   objRegEx.Pattern = strMatchPattern

   'create the collection of matches
   Set Matches = objRegEx.Execute(strPhrase)

   'print out all matches
   For Each Match in Matches
      strReturnStr = "Match found at position "
      strReturnStr = strReturnStr & Match.FirstIndex & ". Match Value is '"
      strReturnStr = strReturnStr & Match.value & "'." & "<BR>" & VBCrLf
      'print
      Response.Write(strReturnStr)
  :  Next
End Sub

'call the subroutine
RegExpTest "is.", "Is1 is2 Is3 is4"
%>
```

Output:

```
Match found at position 0. Match Value is 'Is1'.
```

Match found at position 4. Match Value is 'is2'.
Match found at position 8. Match Value is 'Is3'.
Match found at position 12. Match Value is 'is4'.

PROPERTIES

Count Property

Returns an integer that tells us how many **Match** objects there are in the **Matches Collection**.

Syntax: **object.Count**

Item Property

The **Item** property allows us to retreive the value of an item in the collection designated by the specified **key** argument and also to set that value by using **itemvalue**.

Syntax: **object.Item(key) [= itemvalue]**

METHOD: FileSystemObject.CreateFolder

Implemented in version 2.0

object.CreateFolderfoldername

This method allows us to create a folder with the specified **foldername**.

If a folder already exists with the same name as you are trying to create, you will get an error. The **FolderExists** method can be used to check this before creating you new folder.

(Note that the "c:\DevGuru" folder must exist before you can add the "\myfolder" folder.)

Code:

```
<%
dim filesys, newfolder, newfolderpath
newfolderpath = "c:\DevGuru\myfolder"
set filesys=CreateObject("Scripting.FileSystemObject")
If Not filesys.FolderExists(newfolderpath) Then
   Set newfolder = filesys.CreateFolder(newfolderpath)
   Response.Write("A new folder has been created at: " & newfolderpath)
End If
%>
```

Output:

"A new folder has been created at: c:\DevGuru\myfolder"

FUNCTION: CreateObject( )

CreateObject(ServerName.TypeName, RemoteServerName)

The **CreateObject** function is used to create an object of the type specified in the argument.

The **Set** statement assigns the object reference to a variable or property. The keyword **Nothing** is used to unassign the object reference from the variable or property. Good programming techniques require that you unassign all objects before you exit the program.

There one mandatory and one optional argument.

ServerName.TypeName

ServerName is the name of the application that provides the object. **TypeName** is the type (class) of object to be created.

RemoteServerName

The optional **RemoteServerName** argument was added in verson 5.0 and is used to create an object on a remote server. It is the name of the remote server. To use this argument, the internet security must be turned off.

In this first example, we create an instance of Microsoft Word.

Code:

```
<%
Set objWrd = CreateObject("Word.Application")
...
' Place any code you desire here
...
Set objWrd = Nothing
%>
```

In this second example, we create an object on a remote site.

Code:

```
%lt%
Set objFarAway = CreateObject("Word.Application", "FarAwayServerName")
...
' Place any code you desire here
...
Set objFarAway = Nothing
%>
```

FUNCTION: CSng( )

Implemented in version 1.0

You can determine the expression subtype by using the function **VarType( )**.

CSng(Number)

The **CSng** function converts any number to the variant of subtype **Single**.

Converts to values ranging from
-3.402823E38 to -1.401298E-45 for negative values and 1.401298E-45 to 3.402823E38 for positive values.

Code:

```
<% anynumber=5678.123 %>
<% =CSng(anynumber) %>
```

Output:
5678.123

FUNCTION: CStr( )

Implemented in version 1.0

You can determine the expression subtype by using the function **VarType( )**.

CStr(Expression)

The **CStr** function converts an expression into a variant of subtype **String**.

If the expression is **Boolean**, the output is a string containing either true or false. If the expression is a **Date**, the output is a string using the short-date format. If the expression is empty, the output is a zero length string. If the expression is an error, the output is a string with the word error followed by the error number. If the expression is numeric, the output string contains the number. If the expression is **Null** (does not exist), the output is a runtime error.

Code:

```
<% anyexpression=5678 %>
<% =CStr(anyexpression) %>
```

Output:

```
5678
```

FUNCTION: Date

Implemented in version 1.0

Date

The **Date** function returns the current date as determined by your computer system.

Historically, a two-digit year (MM/DD/YY) was returned. However, courtesy of the millennium, a four-digit year is now returned (MM/DD/YYYY). The same applies to the **DateValue** function.

Code:
```
<% =Date %>
```

Output:
```
2/16/2001
```

FUNCTION: DateAdd( )

Implemented in version 2.0

DateAdd(Interval, Number, Date)

The **DateAdd** function adds a time interval to any date.

There are three mandatory arguments.

Interval

The **Interval** argument defines the the type of time interval you wish to add onto the date.

You must place the setting inside double quotes.

Only the following settings may be used.

SETTING	DESCRIPTION
YYYY	Year
Q	Quarter
M	Month
Y	Day Of Year
D	Day
W	WeekDay
WW	Week Of Year
H	Hour
N	Minute
S	Second

Number

The **Number** argument is a multiplier for the Interval argument (i.e., how many days, weeks, months, etc.). If this is a positive number, you will go forward in time. A negative number will go back in time.

Date

The **Date** argument is a date or time.

If you wish to use the current date or time, simply use the **Date** or the **Now** functions.

Code:

```
<% =Date %>
<% =DateAdd("WW", 6, Date) %>
```

Output:

```
3/16/99
4/27/99
```

Code:

```
<% =Now %>
<% =DateAdd("S", 30, Now) %>
```

Output:

```
3/16/99 12:14:00 PM
3/16/99 12:14:30 PM
```

Code:

```
<% ="1/1/2001" %>
<% =DateAdd("YYYY", 1000, "1/1/2001") %>
```

Output:

```
1/1/2001
1/1/3001
```

FUNCTION: DateDiff( )

Implemented in version 2.0

DateDiff(Interval, Date1, Date2, FirstDayofWeek, FirstWeekofYear)

The **DateDiff** function calculates the amount of time between two different dates.

There are three mandatory arguments.

Interval

The **Interval** argument defines the the type of time interval you wish to use to calculate the time difference.

Only the following settings can be used. You must place the setting inside a pair of double quotes.

SETTING	DESCRIPTION
YYYY	Year
Q	Quarter
M	Month
Y	Day Of Year
D	Day
W	WeekDay
WW	Week Of Year
H	Hour
N	Minute
S	Second

Date1

The **Date1** argument is the first date.

Date2

The **Date2** argument is the second date.

Code:
```
<%=DateDiff("d", Date, "1/1/2000") %>
```

Output:

The order of the two dates determines if the difference is represented as a positive or negative value.

Code:
```
<% olddate = #6/26/43# %>
<% nowdate = Now %>
<% =DateDiff("s", nowdate, olddate) %>
```

Output:
-3130748108

There are two optional arguments.

FirstDayofWeek

The **FirstDayofWeek** argument must only use the constants or values defined below in the Date And Time CONSTANTS.

CONSTANT	VALUE	DESCRIPTION
VBSunday	1	Sunday
VBMonday	2	Monday
VBTuesday	3	Tuesday
VBWednesday	4	Wednesday
VBThursday	5	Thursday
VBFriday	6	Friday
VBSaturday	7	Saturday
VBFirstJan1	1	Week of January 1
VBFirstFourDays	2	First week of the year that has at least four days
VBFirstFullWeek	3	First full week of the year
VBUseSystem	0	Use the date format of the computer's regionsl settings
VBUseSystemDayOfWeek	0	Use the first full day of the week as defined by the system settings

In this example, Tuesday is defined to be the first day of the week. The output is how many weeks, which are now defined to start on Tuesday, are left between the current date and 1/1/2001.

Code:
```
<% =DateDiff("w", Date, "1/1/2001", 3) %>
```

Output:
93

Code:
<% =DateDiff("w", Date, "1/1/2001", VBTUESDAY) %>

Output:
93

FirstWeekofYear

The **FirstWeekofYear** argument must only use the constants or values defined in the Date And Time CONSTANTS which are listed above.

Code:
<% =DateDiff("ww", Date, "1/1/2001", 1, VBFIRSTFOURDAYS)%>

Output:
94

CONSTANTS: Date Format

Implemented in version 2.0

Date Format Constants

You may use either the CONSTANT (left column) or the VALUE (center column) in your code and get the same results.

CONSTANT	VALUE	DESCRIPTION
VBGeneralDate	0	Display the date and time using system settings
VBLongDate	1	Display the date in long date format June 26, 1943
VBShortDate	2	Display the date in short date format 6/26/43
VBLongTime	3	Display the time in long time format 3:48:01 PM
VBShortTime	4	Display the time in short time format (24 hour clock) 15:48

METHOD: FileSystemObject.DeleteFile

Implemented in version 2.0

object.DeleteFile file [, force]

This method is used to delete a specified file or files (using wildcards). The optional **force** parameter returns a Boolean value - **True** allows files with read-only attributes to be deleted, while **False** (default) does not. Trying to delete a read-only file without setting **force** to **True** will return an error.

Note that an error also occurs if you try to delete a file that doesn't exist.

Code:

```
<%
dim filesys
Set filesys = CreateObject("Scripting.FileSystemObject")
filesys.CreateTextFile "c:\somefile.txt", True
If filesys.FileExists("c:\somefile.txt") Then
  filesys.DeleteFile "c:\somefile.txt"
  Response.Write("File deleted")
End If
%>
```

Output:

```
"File deleted"
```

METHOD: FileSystemObject.DeleteFolder

Implemented in version 2.0

object.DeleteFolder folder [, force]

This method is used to delete a specified folder or folders (using wildcards), including all of its subfolders and files. The optional **force** parameter returns a Boolean value - **True** allows folders with read-only attributes to be deleted, while **False** (default) does not.

Note that an error also occurs if you try to delete a folder that doesn't exist.

Code:

```
<%
dim filesys
Set filesys = CreateObject("Scripting.FileSystemObject")
If filesysFolderExists("c:\DevGuru\website\") Then
   filesysDeleteFolder "c:\DevGuru\website\"
   Response.Write("Folder deleted")
End If
%>
```

Output:

"Folder deleted"

Property: Err.Description

Implemented in version 1.0

object. Description [= string]

This property returns or sets a string containing a brief textual description of an error. The optional **string** argument lets you define the description of the error in your code. Otherwise, providing the generated error number relates to a VBScript runtime error, the **Description** value is automatically filled with a default description.

Code:

```
<%
On Error Resume Next
Err.Raise 48
Err.Description = "My error"
Response.Write "Error type is '" & Err.Description & "'."
%>
```

Output:

```
"Error type is 'My error'."
```

STATEMENT: Do

Implemented in version 1.0

Do . . . Loop

The **Do** statement repeats a block of code **Until** a condition becomes true or **While** a condition is satisfied.

You must end all **Do** statements with **Loop** or you will get an error message. You can place **Do** loops inside of other conditional statements. The **While** and the **Until** condition may be placed after the **Do** or the **Loop**.

Code:

```
<%
number = 1
Do Until number = 5
  number = number + 1
Loop
%>

<%
number = 1
Do
  number = number + 1
Loop Until number = 5
%>

<%
number = 1
Do While number < 5
  number = number + 1
Loop
%>

<%
number = 1
Do
  number = number + 1
Loop While number < 5
%>
```

OBJECT: Drive

Implemented in version 3.0

The **Drive** object provides access to the various properties of the local or remote disk drive.

The following code uses the GetDrive method of the **FileSystemObject** object to get the **Drive** object for drive "c".

Code:
```
<%
Set filesys = CreateObject("Scripting.FileSystemObject")
Set drv = filesys.GetDrive("c")
%>
```

PROPERTIES

AvailableSpace Property
Returns the amount of space available on the specified local or remote disk drive.

Syntax: **object.AvailableSpace**

DriveLetter Property
Returns the dirve letter of the specified local or remote disk drive. Read only.

Syntax: **object.DriveLetter**

DriveType Property
Returns an integer indicating the type of the drive.

Syntax: **object.DriveType**

FileSystem Property
This property returns the file system type that is in use on the specified drive.

Syntax: **object.FileSystem**

FreeSpace Property
Returns the amount of free space available to a user on the specified local or remote drive.

Syntax: **object.FreeSpace**

IsReady Property
This property is a Boolean whose value is **True** if the specified drive is available for use and **False** otherwise.

Syntax: **object.IsReady**

Path Property
Returns the path for a specified file, folder or drive.

Syntax: **object.Path**

RootFolder Property
Returns a **Folder** object that represents the root folder of the specified drive.

Syntax: **object.RootFolder**

SerialNumberProperty
Returns the decimal serial number for the specified drive. This number can be used to uniquely identify a disk volume.

Syntax: **object.SerialNumber**

ShareName Property
Returns the network name in Universal Naming Convention (UNC) for the remote disk drive. Used only when working with a remote drive (**DriveType** property is 3).

Syntax: **object.ShareName**

TotalSize Property
Returns the total space, in bytes, of the specified drive.

Syntax: **object.TotalSize**

VolumeName Property
Sets or returns the volume name of the specified drive.

Syntax: **object. VolumeName [= newname]**

Property: Drive.DriveType

Implemented in version 3.0

object.DriveType

The **DriveType** property returns an integer whose value corresponds to one of the **DriveType Constants** listed below. Note that only an integer is returned; to use the names below they must be defined in your code.

0 = Unknown
1 = Removable
2 = Fixed
3 = Network
4 = CD-ROM
5 = RAM Disk

Code:

```
<%
dim filesys
set filesys = CreateObject("Scripting.FileSystemObject")
Set drv = filesys.GetDrive("c")
select case drv.DriveType
   case 0: drtype = "Unknown"
   case 1: drtype = "Removable"
   case 2: drtype = "Fixed"
   case 3: drtype = "Network"
   case 4: drtype = "CD-ROM"
   case 5: drtype = "RAM Disk"
end select
Response.Write ("The specified drive is a " & drtype & " type disk.")
%>
```

Output:

```
The specified drive is a Fixed type disk.
```

METHOD: FileSystemObject.DriveExists

Implemented in version 2.0

object.DriveExists(drive)

This method lets us check if a specified drive exists. It returns a Boolean value - **True** if the drive does exist and **False** if it doesn't. Note that with removable media drives the **DriveExists** method always returns a **True** value. When accessing this type of drive, use the **IsReady** method to determine the drive's status.

Code:

```
<%
dim filesys
Set filesys = CreateObject("Scripting.FileSystemObject")
If filesysDriveExists(drive) Then
  Response.Write("The specified drive does exist.")
End If
%>
```

Output:

"The specified drive does exist"

KEYWORDS: Empty, False, New, Nothing, Null, True

Implemented in version 1.0

Empty

The **Empty** keyword is the value that a variable is automatically assigned when it is declared using the **Dim** statement. It is important to understand that this means that a declared variable does have a value, **Empty**, even if you have purposely not assigned a value in your code.

You can test whether a variable is empty using the **IsEmpty** function. Also, **Empty** is not the same as **Null** or **Nothing**.

False

The **False** keyword has the value of 0. You can use that value as a basis for testing in conditional statements.

New

The **New** keyword is used in conjunction with **Set** to create an instance of a **Class** or a **RegExp**. First, using **Dim**, you must declare the name of the variable that the instance will be assigned. Then, you can create the instance.

Code:
```
<%
Dim SearchPattern
Set SearchPattern = New RegExp
%>
```

Nothing

The **Nothing** keyword is used to disassociate an object variable from an object. If several object variables refer to the same object, all must be set to **Nothing** before system resources are released.

You must assign **Nothing** using the **Set** keyword.

Code:
```
<% Set YourObject = Nothing %>
```

Null

The **Null** keyword indicates that a variable contains no valid data. You can assign a variable the value of **Null** and use the **IsNull()** function to test whether a variable is **Null**. If you print the variable out, the output should be **Null**.

Also, **Null** is not the same as **Empty** or **Nothing**.

True

The **True** keyword has the value of -1. You can use that value as a basis for testing in conditional statements.

OPERATORS: Eqv

Implemented in version 1.0

Eqv

The **Eqv** operator is used to perform a logical comparison on two exressions (i.e., are the two expressions identical), where the expressions are **Null**, or are of **Boolean** subtype and have a value of **True** or **False**.

The **Eqv** operator can also be used a "bitwise operator" to make a bit-by-bit comparison of two integers. If both bits in the comparison are the same (both are 0's or 1's), then a 1 is returned. Otherwise, a 0 is returned.

The order of the expressions in the comparison is not important.

Code:
```
<% =True Eqv True %>
<% =True Eqv False %>
<% =False Eqv True %>
<% =False Eqv False %>
```

Output:
```
True
False
False
True
```

Code:
```
<% Expression1 = True %>
<% Expression2 = False %>
<% =Expression1 Eqv Expression2 %>
```

Output:
```
False
```

STATEMENT: Erase

Implemented in version 1.0

Erase

The **Erase** statement is used to empty arrays.

If the array is fixed (static), this statement removes the values for all of the elements. If the fixed array is a string array, all of the string elements are reinitialized to "". If the fixed array is a numeric array, all of the numeric elements are reinitialized to 0. So the memory remains allocated for the elements. The array continues to exist with the same size (in the example 3), but the elements are essentially zeroed out.

Fixed array:

Code:
```
<% Dim myarray(3) %>
<% myarray(0) = 111 %>
<% myarray(1) = 222 %>
<% myarray(2) = 333 %>
<% Erase myarray %>
```

If the array is dynamic, the erase statement frees the memory allocated to the dynamic array and the array is destroyed. If you try to access an element in the erased array, you will get an error message (array out of bounds) since the elements do not exist in memory any more. However, you can reinitialize the array by using **ReDim**. This ability to free memory is very useful.

Dynamic array:

Code:
```
<% Dim anarray() %>
<% ReDim anarray(3) %>
<% anarray(0) = "First string." %>
<% anarray(1) = "Second string." %>
<% anarray(2) = "Third string." %>
<% Erase anarray %>
```

OBJECT: Err

Implemented in version 1.0

The **Err** object holds information about the last runtime error that occured. It is not nessecary to create an instance of this object; it is intrinsic to VBScript. Its default property is **Number**, which contains an integer representing a VBScript error number or an ActiveX control Status Code (SCODE) number. This value is automatically generated when an error occurs and is reset to zero (no error) after an **On Error Resume Next** statement or after using the **Clear** method.

The following code checks the value of the **Number** property and, if it contains a value other than zero, displays the details in the browser.

Code:

```
<%
dim numerr, abouterr
On Error Resume Next
Err.Raise 6
numerr = Err.number
abouterr = Err.description
If numerr <> 0 Then
  Response.Write "An Error has occured! Error number " & numerr & " of_
                  the type '" & abouterr & "'."
End If
%>
```

Output:

"An Error has occured! Error number 6 of the type 'Overflow'."

PROPERTIES

Description Property

This property returns or sets a string containing a brief textual description of an error.

Syntax: **object.Description [= string]**

HelpContext Property

This property is used to set or return a context ID for a Help topic specified with the **HelpFile** property.

Syntax: **object.HelpContext [= contextID]**

HelpFile Property

This property is used to get or define the path to a Help file.

Syntax: **object.HelpFile [= contextID]**

Number Property

This property is used to retrieve or set the value that relates to a specific runtime error.

Syntax: **object.Number [= errnumber]**

Source Property

This property lets us determine the object or application that caused an error, and returns a string that contains its **Class** name or programmatic ID.

Syntax: **object.Source [= string]**

METHODS

Clear Method

This method is used to clear the settings of the Error object.

Syntax: **object.Clear**

Raise Method

This method is used to generate a VBScript runtime error.

Syntax: **object. Raise (number[, source, description, helpfile, helpcontext])**

Property: Err.HelpContext

Implemented in version 1.0

object.HelpContext [= contextID]

This property is used to set or return a context ID for a Help topic specified with the **HelpFile** property. If no specific Help file is defined in your code then one of two things can happen. Firstly, VBScript checks to see if the **Number** value of the **Err** object relates to a known runtime error. If it does then the VBScript Help context ID for that error is used. However, if the **Number** property value is not recognised, then VBScript will display the Help contents screen.

Code:

```
<%
On Error Resume Next
Err.Raise 10
Err.HelpFile = "userhelp.hlp"
Err.HelpContext = usercontextID
If Err.Number <> 0 Then
   MsgBox "Press F1 for help", , "Error: " & Err.Description, Err.Helpfile, _
            Err.HelpContext
End If
%>
```

Property: Err.HelpFile

Implemented in version 1.0

object. HelpFile [= contextID]

This property is used to get or define the path to a Help file. Once set, this Help file is available to the user when pressing the 'Help' button in the Message box. If the **HelpContext** property is set then the Help file that the context Id points to will be displayed and, if no **HelpFile** is specified, the VBScript Help contents screen is used.

Code:

```
<%
On Error Resume Next
Err.Raise 10
Err.HelpFile = "userhelp.hlp"
Err.HelpContext = usercontextID
If Err.Number <> 0 Then
   MsgBox "Press F1 for help", , "Error: " & Err.Description, Err.Helpfile, _
            Err.HelpContext
End If
%>
```

Property: Err.Number

Implemented in version 1.0

object.Number [= errnumber]

This property, which is the default property of the **Err** object, is used to retrieve or set the value that relates to a specific runtime error or SCODE. This value is automatically generated when an error occurs and is reset to zero (no error) after an **On Error Resume Next** statement or after using the **Clear** method.

Code:

```
<%
On Error Resume Next
Err.Raise 10
Err.HelpFile = "userhelp.hlp"
Err.HelpContext = usercontextID
If Err.Number <> 0 Then
   MsgBox "Press F1 for help", , "Error: " & Err.Description, Err.Helpfile, _
            Err.HelpContext
End If
%>
```

Property: Err.Source

Implemented in version 1.0

object. Source [= string]

This property lets us determine the object or application that caused an error, and returns a string that contains its **Class** name or programmatic ID. Also used when generating errors in your code to identify your application as the source, using the optional **[= string]** argument, as below.

Code:

```
<%
On Error Resume Next
Err.Raise 8    'raise a user-defined error
Err.Description = "My Error"
Err.Source = "MyAppName"
Response.Write "An Error of the type '" & Err.Description & "' has been_
                caused by '" & Err.Source & "'."
%>
```

Output:

"An Error of the type 'My Error' has caused by 'MyAppName'."

METHOD: Err.Raise

Implemented in version 1.0

object.Raise(number[, source, description, helpfile, helpcontext])

This method is used to generate a VBScript runtime error. The arguments, with the exception of **number**, are all optional. Note that if no arguments are supplied, any error values that have not been cleared from the Err object will automatically be inherited by the new instance of the object.

Code:

```
<%
On Error Resume Next
Err.Raise 8    'raise a user-defined error
Err.Description = "My Error"
Err.Source = "MyAppName"
Response.Write "An Error of the type '" & Err.Description & "' has been_
                caused by '" & Err.Source & "'."
%>
```

Output:

"An Error of the type 'My Error' has caused by 'MyAppName'."

FUNCTION: Eval()

Implemented in version 5.0

Eval(Expression)

The **Eval** function takes a single argument, evaluates it as a VBScript expression, and returns the result of this evaluation.

If the expression is of the form a = b, it is treated as an equality comparison. If the comparison is true, then **True** is returned. Otherwise, **False** is returned.

There is a statement, **Execute**, which is similar in operation to the **Eval** function. **Execute** differs in that it interprets a string expression as one or a series of statments to be executed, and in the fact that it does not return a value.

Code:
```
<%
ThisVar = 5.556
AnotherVar = 5.556
%>
<% =Eval("ThisVar = AnotherVar") %>
```

Output:
```
true
```

Code:
```
<% MyVar = Eval("CInt(12345.6789)") %>
<% =MyVar %>
```

Output:
```
12345
```

METHOD: object.Execute

Implemented in version 5.0

object.Execute (TargetString)

The **Execute** method is used with a **RegExp** object variable to look for a search string pattern (also known as a regular expression) inside a target string.

There is one mandatory argument, the target string to be searched. The search string pattern (regular expression) is declared using the **Pattern** property.

Each time a match is made (i.e., the search pattern is found inside the target string), a **Match** object is created and added to a **Matches** collection. Note that a match does not have to occur. Therefore the **Execute** method can return a **Matches** collection that is empty, or that contains one or more, objects.

Code:

```
<%
'this sub finds the matches
Sub RegExpTest(strMatchPattern, strPhrase)
   'create variables
   Dim objRegEx, Match, Matches, StrReturnStr
   'create instance of RegExp object
   Set objRegEx = New RegExp

   'set the pattern
   objRegEx.Pattern = strMatchPattern

   'create the collection of matches
   Set Matches = objRegEx.Execute(strPhrase)

   'print out all matches
   For Each Match in Matches
      strReturnStr = "Match found at position "
      strReturnStr = strReturnStr & Match.FirstIndex & ". Match Value is `"
      strReturnStr = strReturnStr & Match.value & "'."
      'print
      Response.Write(strReturnStr & "<BR>")
   Next
End Sub

'call the subroutine
RegExpTest "is.", "Is1 is2 Is3 is4"
%>
```

Output:

```
Match found at position 4. Match Value is `is2'.
```

STATEMENT: Execute

Execute (String)

The **Execute** statement takes a single string argument, interprets it as a VBScript statement or sequence of statements, and executes these statements in the current context. No value is returned. If you wish to evaluate an expression (which returns a value) use the **Eval** function instead.

To pass multiple statements to the **Execute** function you should use colons or line-breaks as separators.

Unlike the **Eval** function, **Execute** interprets all "=" operators as assignments rather than comparisons.

Code:

```
<%
Execute("Call MySub 4, strArg2")
%>
```

STATEMENT: Exit

Implemented in version 1.0

Exit

The **Exit** statement allows you to exit from inside a block of code such as a conditional statement (**Do ...Loop, For ... Next, For Each ... Next**) or a procedure (**Function, Sub**) before it would normally be completed.

This allows you to exit from inside a **Do Until ... Loop**.

Code:
```
<% Exit Do %>
```

This allows you to exit from inside a **Do While ... Loop**.

Code:
```
<% Exit Do %>
```

This allows you to exit from inside a **For ... Next**.

Code:
```
<% Exit For %>
```

This allows you to exit from inside a **For Each ... Next**.

Code:
```
<% Exit For %>
```

This allows you to exit from inside a **Function**.

Code:
```
<% Exit Function %>
```

This allows you to exit from inside a **Sub**.

Code:
```
<% Exit Sub %>
```

STATEMENT: Exit Property

Implemented in version 5.0

Exit Property

The **Exit Property** statement is used to immediately exit from a **Property Get**, **Property Let**, or a **Property Set** statement block.

It cannot be used in any other procedure.

Code:

```
<%
Property Get PieType Month
    If Month = "October" Then
        PieType = "pumpkin"
        Exit Property
    End If
    PieType = "apple"
End Property
%>
```

FUNCTION: Exp( )

Implemented in version 1.0

Exp(Number)

The **Exp** function raises e to the power of a number.

There is a companion function **Log** for the reverse operation.

Code:
```
<% =Exp(3.269) %>
```

Output:
26.2850411552082

You can also use a negative number.

Code:
```
<% =Exp(-3.269) %>
```

Output:
0.038044452511799

OBJECT: File

Implemented in version 3.0

The **File** object allows you access and manipulate the various properties of a file.

The following code uses the GetFile method of the **FileSystemObject** object to create a **File** object and view one of its properties.

Code:
```
<%
Dim filesys, demofile, createdate
Set filesys = CreateObject("Scripting.FileSystemObject")
Set demofile = filesys.GetFile("filename")
createdate = demofile.DateCreated
%>
```

PROPERTIES

Attributes Property
This property allows us to get or change the various attributes of a file.

Syntax: **object.Atributes [= newattributes]**

DateCreated Property
This property gets the date and time that the file was created.

Syntax: **object.DateCreated**

DateLastAccessed Property
Gets the date and time that the file was last accessed.

Syntax: **object.DateLastAccessed**

DateLastModified Property
This property returns the date and time that the file was last modified.

Syntax: **object.DateLastModified**

Drive Property
Returns the drive letter of the drive where the file is located.

Syntax: **object.Drive**

Name Property
Lets us get or change the name of the specified file.

Syntax: **object.Name [= newname]**

ParentFolder Property
This property gets the **Folder** object for the parent relating to the specified file.

Syntax: **object.ParentFolder**

Path Property
This property returns a file's path.

Syntax: **object.Path**

ShortName Property
Returns the short version of a filename (using the 8.3 convention).
e.g. Employees.html is truncated to Employ~1.htm

Syntax: **object.ShortName**

ShortPath Property
Returns the short version of the file path (this is the path with any folder and file names truncated as above).

Syntax: **object.ShortPath**

Size Property
Returns the size of a file in bytes.

Syntax: **object.Size**

TypeProperty
Returns a string containing the file type description.
e.g. For files ending in .TXT, "Text Document" is returned.

Syntax: **object.Type**

METHODS

Copy Method
This method copies the selected file to the specified destination.

Syntax: **object.Copy destination[, overwrite]**

Delete Method
The method used to delete the file relating to the specified **File** object.

Syntax: **object.Delete [force]**

Move Method
This method is used to move the file relating to the specified **File** object to a new destination.

Syntax: **object.Move destination**

OpenAsTextStream Method
This method opens a specified file and returns an instance of a **TextStream** object that can then be manipulated - read from, written or appended to.

Syntax: **object.OpenAsTextStream([iomode [, format]])**

Property: File.Attributes

Implemented in version 3.0

object.Attributes [= newattributes]

This property allows us to get or change the various attributes of a file. The available attribute values are listed below.

Name	Value	Description	Read/Write attribute
Normal	0	Normal file	Read/write
ReadOnly	1	Read-only file	Read only
Hidden	2	Hidden file	Read/write
System	4	System file	Read/write
Volume	8	Disk drive volume label	Read only
Directory	16	Folder or directory	Read-only
Archive	32	File has changed since last backup	Read/write
Alias	64	Link or shortcut	Read-only
Compressed	2048	Compressed file	Read-only

The following code shows how to check if a file is read/write or read-only. As you can see, logical operators are used to get or change the various attributes.

Code:

```
<%
dim filesys, text, readfile
set filesys = CreateObject("Scripting.FileSystemObject")
Set text = filesys.CreateTextFile("c:\somefile2.txt")
text.Write "A simple test to find if this file is writeable."
text.close
set readfile = filesys.GetFile("c:\somefile2.txt")
If Not readfile.Attributes And 1 Then
  Response.Write "The file is Read/Write."
Else
  Response.Write "The file is Read-only."
End If
%>
```

Output:

"The file is Read/Write."

METHOD: File.Copy

Implemented in version 3.0

object.Copy destination[, overwrite]

This method makes a copy of the selected file and saves it to the specified destination. The **overwrite** parameter is a Boolean value - **True** allows existing files with the same name as **destination** to be overwritten while **False** (the default) does not.

Note that after copying a file to a new location the **File** object still relates to the original file.

Code:

```
<%
dim filesys, demofile
set filesys = CreateObject ("Scripting.FileSystemObject")
set demofile = filesys.CreateTextFile ("c:\somefile.txt", true)
set demofile = filesys.GetFile("c:\somefile.txt")
demofile.Copy("c:\projects\someotherfile.txt")
%>
```

METHOD: File.Delete

Implemented in version 3.0

object.Delete [force]

The method used to delete the file relating to the specified **File** object. The **force** parameter is a Boolean value - **True** allows the deletion of read-only files while **False** (the default) does not.

Note that trying to delete a file that doesn't exist will cause an error.

Code:

```
<%
dim filesys, demofile
set filesys = CreateObject ("Scripting.FileSystemObject")
set demofile = filesys.CreateTextFile ("c:\somefile.txt", true)
set demofile = filesys.GetFile("c:\somefile.txt")
demofile.Delete
%>
```

METHOD: File.Move

Implemented in version 3.0

object.Move destination

This method is used to move the file relating to the specified **File** object to a new destination.

It is recommended that you use the **FileExists** method of the **FileSystemObject** object to determine if a file with the same name already exists in the destination location. If it does, and you attempt the **Move**, you'll receive an error.

Code:

```
<%
dim filesys, demofile
set filesys = CreateObject ("Scripting.FileSystemObject")
set demofile = filesys.CreateTextFile ("c:\somefile.txt", true)
set demofile = filesys.GetFile("c:\somefile.txt")
demofile.Move ("c:\projects\test\")
%>
```

METHOD: File.OpenAsTextStream

Implemented in version 3.0

object.OpenAsTextStream([iomode [, format]])

This method opens a specified file and returns an instance of a **TextStream** object that can then be manipulated - read from, written or appended to.

This method has two optional parameters. The first, **iomode**, has the following available values:

Constant	Value	Description
ForReading	1	File is opened for reading only
ForWriting	2	File is opened for writing and the contents of any existing file with the same name are overwritten
ForAppending	8	Opens the file and writes to the end of any existing text.

The second optional parameter, **format**, can have any of the following values:

Constant	Value	Description
TristateUseDefault	-2	Uses the system default file format
TristateTrue	-1	Opens the file using the Unicode format
TristateFalse	0	Opens the file in AscII format

Code:

```
<%
dim filesys, demofile, txtstream
set filesys = CreateObject ("Scripting.FileSystemObject")
set demofile = filesys.CreateTextFile ("c:\somefile.txt", true)
set demofile = filesys.GetFile("c:\somefile.txt")
set txtstream = demofile.OpenAsTextStream (2, -2)
txtstream.Write "This will overwrite any text already in the file."
txtstream.Close
%>
```

CONSTANTS: FILE ATTRIBUTE

Implemented in version 3.0

File Attribute Constants

These are used with the **File.Attributes** property

Name	Value	Description	Read/Write attribute
Normal	0	Normal file	Read/write
ReadOnly	1	Read-only file	Read only
Hidden	2	Hidden file	Read/write
System	4	System file	Read
Volume	8	Disk drive volume label	Read only
Directory	16	Folder or directory	Read-only
Archive	32	File has changed since last backup	Read/write
Alias	64	Link or shortcut	Read-only
Compressed	2048	Compressed file	Read-only

METHOD: FileSystemObject.FileExists

Implemented in version 2.0

object.FileExists(file)

This method lets us check if a specified file exists. It returns a Boolean value - **True** if the file does exist and **False** if it doesn't. Note that if the file that you are checking for isn't in the current directory, you must supply the complete path.

In the following example the **FileExists** method is used to check for a specific file before deleting it.

Code:

```
<%
dim filesys
Set filesys = CreateObject("Scripting.FileSystemObject")
filesys.CreateTextFile "c:\somefile.txt", True
If filesys.FileExists("c:\somefile.txt") Then
  filesys.DeleteFile "c:\somefile.txt"
  Response.Write("File deleted")
End If
%>
```

Output:

"File deleted"

CONSTANTS: File Input/Output

Implemented in version 3.0

File Input/Output Constants

You may use either the CONSTANT (left column) or the VALUE (center column) in your code and get the same results.

CONSTANT	VALUE	DESCRIPTION
ForReading	1	Opens a file for reading only
ForWriting	2	Opens a file for writing. If the file already exists, the contents are overwritten.
ForAppending	8	Opens a file and starts writing at the end (appends). Contents are not overwritten.

OBJECT: FileSystemObject

Implemented in version 2.0

The **FileSystemObject** is used to gain access to a computer's file system. It can create new files and access existing ones.

The following code uses the **CreateTextFile** method of the **FileSystemObject** object to create a text file (c:\somefile.txt) and then writes some text to it.

Code:

```
<%
dim filesys, filetxt, getname, path
Set filesys = CreateObject("Scripting.FileSystemObject")
Set filetxt = filesys.CreateTextFile("c:\somefile.txt", True)
path = filesys.GetAbsolutePathName("c:\somefile.txt")
getname = filesys.GetFileName(path)
filetxt.WriteLine("Your text goes here.")
filetxt.Close
If filesys.FileExists(path) Then
  Response.Write ("Your file, '" & getname & "', has been created.")
End If
%>
```

Output:

"Your file, 'somefile.txt', has been created."

PROPERTIES

Drives Property

Returns a **Drives collection** consisting of all the **Drive** objects on a computer.

Syntax: **[drvcollection =] object.Drives**

METHODS

BuildPath Method

This method is used to append a name onto an existing path.

[newfullpath =]object.BuildPath(path, name)

CopyFile Method

This method allows us to copy one or more files from one location (the source) to another (destination).

Syntax: **object.CopyFile source, destination [, overwrite]**

CopyFolder Method

Copies one or more folders and all contents, including files and subfolders, from one location to another.

Syntax: **object.CopyFolder source, destination, [, overwrite]**

CreateFolder Method

This method allows us to create a folder.

Syntax: **object.CreateFolderfoldername**

CreateTextFile Method
Creates a text file and returns a **TextStreamObject** that can then be used to write to and read from the file.

Syntax: **object.CreateTextFile filename [, overwrite[, unicode]]**

DeleteFile Method
This method deletes a specified file or files (using wilcards).

Syntax: **object.DeleteFile file [, force]**

DeleteFolder Method
This method deletes a specified folder, including all files and subfolders.

Syntax: **object.DeleteFolder folder [, force]**

DriveExists Method
This method lets us check if a specified drive exists. It returns **True** if the drive does exist and **False** if it doesn't.

Syntax: **object.DriveExists(drive)**

FileExists Method
Lets us check whether a specified file exists. Returns **True** if the file does exist and **False** otherwise.

Syntax: **object.FileExists(file)**

FolderExists Method
Allows us to check if a specified folder exists. Returns **True** if the folder does exist and **False** if it doesn't.

Syntax: **object.FolderExists(folder)**

GetAbsolutePathName Method
This method gets the complete path from the root of the drive for the specified path string.

Syntax: **object.GetAbsolutePathName(path)**

GetBaseName Method
This method gets the base name of the file or folder in a specified path.

Syntax: **object.GetBaseName(path)**

GetDrive Method
This method returns a **Drive** object corresponding to the drive in a supplied path.

Syntax: **object.GetDrive(drive)**

GetDriveName Method
This method gets a string containing the name of the drive in a supplied path.

Syntax: **object.GetDriveName(path)**

GetExtensionName Method
Used to return a string containing the extension name of the last component in a supplied path.

Syntax: **object.GetExtensionName(path)**

GetFile Method
Returns the **File** object for the specified file name.

Syntax: **object.GetFile(filename)**

GetFileName Method
This method is used to return the name of the last file or folder of the supplied path.

Syntax: **object.GetFileName(path)**

GetFolder Method
This method returns a **Folder** object fo the folder specified in the folder parameter.

Syntax: **object.GetFolder(folder)**

GetParentFolderName Method
Returns a string containing the name of the parent folder of the last file or folder in a specified path.

Syntax: **object.GetParentFolderName(path)**

GetSpecialFolder Method
Returns the path to one of the special folders - \Windows, \System or \TMP.

Syntax: **object.GetSpecialFolder(folder)**

GetTempName Method
This method is used to generate a random filename for a temporary file..

Syntax: **object.GetTempName**

MoveFile Method
Moves one or more files from one location to another.

Syntax: **object.MoveFile source, destination**

MoveFolder Method
Moves one or more folders from one location to another.

Syntax: **object.MoveFolder source, destination**

OpenTextFile Method
Opens the file specified in the filename parameter and returns an instance of the **TextStreamObject** for that file.

Syntax: **object.OpenTextFile(filename [, iomode[, create[, format]]])**

METHOD: FileSystemObject.CreateTextFile

Implemented in version 2.0

object.CreateTextFile filename [, overwrite[, unicode]]

This method is used to create a text file and returns a **TextStreamObject** that can then be used to write to and read from the file.

The optional **overwrite** parameter returns a Boolean value - **True** (the default) permits overwriting of existing files while **False** does not. The other optional parameter, **unicode**, is also a Boolean. In this case, **True** creates a Unicode file and **False** (the default) creates an AscII file.

Code:

```
<%
dim filesys, filetxt, getname, path
Set filesys = CreateObject("Scripting.FileSystemObject")
Set filetxt = filesys.CreateTextFile("c:\somefile.txt", True)
path = filesys.GetAbsolutePathName("c:\somefile.txt")
getname = filesys.GetFileName(path)
filetxt.WriteLine("Your text goes here.")
filetxt.Close
If filesys.FileExists(path) Then
  Response.Write ("Your file, '" & getname & "', has been created.")
End If
%>
```

Output:

```
"Your file, 'somefile.txt', has been created."
```

METHOD: FileSystemObject.FolderExists

Implemented in version 2.0

object.FolderExists(folder)

This method allows us check if a specified folder exists. It returns a Boolean value - **True** if the folder does exist and **False** if it doesn't. Note that if the folder that you are checking for isn't a subfolder of the current folder, you must supply the complete path.

In the following example the **FolderExists** method is used before creating a new folder to check that it doesn't already exist.

Code:

```
<%
dim filesys, newfolder
set filesys=CreateObject("Scripting.FileSystemObject")
If  Not filesys.FolderExists("c:\DevGuru\website\") Then
   newfolder = filesys.CreateFolder "c:\DevGuru\website\"
   Response.Write("A new folder '" & newfolder & "' has been created")
End If
%>
```

Output:

"A new folder 'c:\DevGuru\website\' has been created."

METHOD: FileSystemObject.GetAbsolutePathName

Implemented in version 2.0

object.GetAbsolutePathName(path)

This method returns the complete path from the root of the drive for the specified path string.

The following examples assume that the current directory is c:\DevGuru\website
Code:

```
<%
dim filesys, pathstring
set filesys=CreateObject("Scripting.FileSystemObject")
pathstring = filesys.GetAbsolutePathName("c:") %>
```

Output:
"c:\DevGuru\website"

Code:

```
<%
dim filesys, pathstring
set filesys=CreateObject("Scripting.FileSystemObject")
pathstring = filesys.GetAbsolutePathName("VBScript.html")
%>
```

Output:
"c:\DevGuru\website\VBScript.html"

METHOD: FileSystemObject.GetBaseName

Implemented in version 2.0

object.GetBaseName(path)

This method allows us to get the base name of a file or folder in the specified path. The returned string contains only the base name of the file, without any extension.

Note that this method doesn't check the existance or the validity of the supplied path.

Code:

```
<%
dim filesys, a
Set filesys = CreateObject("Scripting.FileSystemObject")
filesys.CreateTextFile "c:\DevGuru\website\VBScript.txt", True
a = filesys.GetBaseName("c:\DevGuru\website\VBScript.txt")
Response.Write (a)
%>
```

Output:

"VBScript"

METHOD: FileSystemObject.GetDrive

Implemented in version 2.0

object.GetDrive(drivename)

This method returns the drive object that is specified by the **drivename** argument.

The **drivename** argument must be in an accepted format:
A drive letter such as, c
A drive letter with a colon such as, c:
A drive letter with a colon and a path such as, c:\hisfolder\herfile

You also may use the **GetDriveName** method to obtain the **drivename** argument.

If the drive does not exist or the format is not correct, the **GetDrive** method will return an error.

Code:

```
<%
dim filesys, name, DriveObject
Set filesys = CreateObject("Scripting.FileSystemObject")
name = filesys.GetDriveName("c:\DevGuru\website\VBScript.txt")
Set DriveObject = filesys.GetDrive("c:\DevGuru\website\VBScript.txt")
Response.Write(name)
%>
```

Output:

```
"c:"
```

METHOD: FileSystemObject.GetDriveName

Implemented in version 2.0

object.GetDriveName(path)

This method returns a string that contains the drive name in the specified path. If the drive cannot be determined the **GetDriveName** method will return an empty string.

Note that this method doesn't check the existance or the validity of the supplied path.

Code:

```
<%
dim filesys, a
Set filesys = CreateObject("Scripting.FileSystemObject")
a = filesys.GetDriveName("c:\DevGuru\website\VBScript.txt")
Response.Write (a)
%>
```

Output:

```
"c:"
```

METHOD: FileSystemObject.GetExtensionName

Implemented in version 2.0

object.GetExtensionName(path)

Used to return a string containing the extension name of the last component in a supplied path. If no component matches the supplied path string the **GetExtensionName** method will return an empty string.

Code:

```
<%
dim filesys, a
Set filesys = CreateObject("Scripting.FileSystemObject")
filesys.CreateTextFile "c:\DevGuru\website\VBScript.txt", True
a = filesys.GetExtensionName("c:\DevGuru\website\VBScript.txt")
Response.Write (a)
%>
```

Output:

```
"txt"
```

METHOD: FileSystemObject.GetFile

Implemented in version 2.0

object.GetFile (filename)

This method is used to get the **File** object for the path that you specify. You can then use the new variable containing the file object to access its various methods and properties. The following code demonstrates this by returning the date the file was created and displaying the details in the browser.

Code:

```
<%
dim filesys, filetxt, f
Set filesys = CreateObject("Scripting.FileSystemObject")
Set filetxt = filesys.CreateTextFile("c:\somefile.txt", true)
Set f = filesys.GetFile("C:\somefile.txt")
Response.Write ("Your file was created on " & f.DateCreated)
%>
```

Output:

"Your file was created on 5/21/99 2:07:21 PM"

METHOD: FileSystemObject.GetFileName

Implemented in version 2.0

object.GetFileName(path)

This method is used to return the name of the last file or folder of the supplied path. If the supplied path string doesn't end with the specified file or folder the **GetFileName** method will return an empty string.

Note that this method doesn't check the existance or the validity of the supplied path.

Code:

```
<%
dim filesys, a
Set filesys = CreateObject("Scripting.FileSystemObject")
filesys.CreateTextFile "c:\DevGuru\website\VBScript.txt", True
a = filesysGetFileName("c:\DevGuru\website\VBScript.txt")
Response.Write (a)
%>
```

Output:

```
"txt"
```

METHOD: FileSystemObject.GetFolder

Implemented in version 2.0

object.GetFolder (foldername)

This method is used to get the **Folder** object for the path that you specify. You can than use the new variable containing the folder object to access its various methods and properties. The following code demonstrates this by returning the date the folder was created and displaying the details in the browser.

Code:

```
<%
dim filesys, f
Set filesys = CreateObject("Scripting.FileSystemObject")
Set f = filesys.GetFolder("C:\TestFolder\")
Response.Write ("Your folder was created on " & f.DateCreated)
%>
```

Output:

"Your folder was created on (date created goes here)."

METHOD: FileSystemObject.GetParentFolderName

Implemented in version 2.0

object.GetParentFolderName(path)

Used to return a string containing the parent folder name of the last component in a supplied path. If no component matches the supplied path string the **GetParentFolderName** method will return an empty string.

Note that this method doesn't check the existance or the validity of the supplied path.

Code:

```
<%
dim filesys, a
Set filesys = CreateObject("Scripting.FileSystemObject")
a = filesys.GetParentFolderName("c:\DevGuru\website\VBScript.txt")
Response.Write (a)
%>
```

Output:

```
"website\"
```

METHOD: FileSystemObject.GetSpecialFolder

Implemented in version 2.0

object.GetSpecialFolder (foldername)

This method is used to get the path for one of Windows' special folders. These folders are:

0 - WindowsFolder, containing the files installed by the operating system.

1 - SystemFolder, containing fonts, libraries and device drivers required by the operating system.

2 - TemporaryFolder, used to store temporary (.TMP) files.

Code:

```
<%
dim filesys, f
Set filesys = CreateObject("Scripting.FileSystemObject")
Set f = filesys.GetSpecialFolder(1)
Response.Write ("The path to your System folder is '" & f & "'." )
%>
```

Output:

"The path to your System folder is 'C:\WINNT\system32'."

METHOD: FileSystemObject.GetTempName

Implemented in version 2.0

object.GetTempName

This method is used to generate a random filename to use for a temporary file. It does not create a temporary file, but is used in conjunction with the **CreateTextFile** method when a temporary file is required.

Code:

```
<%
Dim filesys, tempname, tempfolder, tempfile
Set filesys = CreateObject("Scripting.FileSystemObject")
Set tempfolder = filesys.GetSpecialFolder(2)
tempname = filesys.GetTempName
Set tempfile = tempfolder.CreateTextFile(tempname)
Response.Write ("The temporary file, '" & tempfile & "', has been created")
%>
```

Output:

"The temporary file, 'rad80F30.tmp', has been created."

METHOD: FileSystemObject.MoveFile

Implemented in version 2.0

object.MoveFile source, destination

This method lets us move one or more files from one location (the source) to another (destination). Wildcards can be used within the **source** string, providing it is the last component in the path, to enable the moving of multiple files, but cannot be used in the **destination** string. Note that if the **source** does contain wildcards, or if the **destination** ends with a back-slash (path separator), it is automatically assumed that the **destination** is an existing folder and any matching files are moved to it.

It is recommended that you use the **FileExists** method when moving a file - if a **source** file doesn't exist you'll get an error. An error also occurs if the **destination** is a directory or an existing file.

Code:

```
<%
dim filesys
set filesys=CreateObject("Scripting.FileSystemObject")
If filesys.FileExists("c:\sourcefolder\anyfile.html") Then
  filesys.MoveFile "c:\sourcefolder\anyfile.html", "c:\destfolder\"
End If
%>
```

METHOD: FileSystemObject.MoveFolder

Implemented in version 2.0

object.MoveFolder source, destination

This method lets us move one or more folders from one location (the source) to another (destination). Wildcards can be used within the **source** string, providing it is the last component in the path, to enable the moving of multiple folders, but cannot be used in the **destination** string. Note that if the **source** does contain wildcards, or if the **destination** ends with a back-slash (path separator), it is automatically assumed that the **destination** is an existing folder and any matching folders are moved to it.

It is recommended that you use the **FolderExists** method when moving a folder - if a **source** folder doesn't exist you'll get an error. An error also occurs if the **destination** is a directory or an existing file.

Code:

```
<%
dim filesys
set filesys=CreateObject("Scripting.FileSystemObject")
If filesys.FolderExists("c:\sourcefolder\test\") Then
  filesys.MoveFolder "c:\sourcefolder\test\", "c:\destfolder\"
End If
%>
```

METHOD: FileSystemObject.OpenTextFile

Implemented in version 2.0

object.OpenTextFile (filename [, iomode[, create[, format]]])

This method is used to open a text file and returns a **TextStreamObject** that can then be used to write to, append to, and read from the file.

The optional **iomode** argument can have one of the following **Constants** as its value:

CONSTANT	VALUE	DESCRIPTION
ForReading	1	Opens a file for reading only
ForWriting	2	Opens a file for writing. If the file already exists, the contents are overwritten.
ForAppending	8	Opens a file and starts writing at the end (appends). Contents are not overwritten.

The optional **create** argument can be either **True**, which will create the specified file if it does not exist, or **False**, which won't.

The optional **format** argument uses one of the following **Tristate** values to specify in which format the file is opened. If not set, this defaults to **TristateFalse**, and the file will be opened in ASCII format.

CONSTANT	VALUE	DESCRIPTION
TristateTrue	-1	Opens the file as Unicode
TristateFalse	0	Opens the file as ASCII
TristateUseDefault	-2	Use default system setting

The following example will open the file, "c:\somefile.txt" (or create it if it does not exist), and append the specified text to it.

Code:

```
<%
dim filesys, filetxt
Const ForReading = 1, ForWriting = 2, ForAppending = 8
Set filesys = CreateObject("Scripting.FileSystemObject")
Set filetxt = filesys.OpenTextFile("c:\somefile.txt", ForAppending, True)
filetxt.WriteLine("Your text goes here.")
filetxt.Close
%>
```

FUNCTION: Filter( )

Implemented in version 1.0

Filter(String, Substring, Include,Compare)

The **Filter** function searches the elements of a zero-based array, to match a pattern of one or more characters, and creates a new array, either with or without the elements containing the matched pattern.

You can create a zero-based string array by using the **Split** function. The **Join** function is used to reassemble the string after applying the **Filter** function.

There are two mandatory arguments.

String

The **String** argument is the name of a zero-based string array.

Substring

The **Substring** argument is the pattern of one or more characters that are searched for in the array.

Code:
```
<% myarray = Split("How now purple cow?") %>
<% myfilterarray = Filter(myarray, "ow") %>
<% =Join(myfilterarray) %>
```

Output:
How now cow?

There are two optional arguments.

Include

The optional **Include** argument must only be **True** or **False**. If **True**, the returned array will only consist of the values that contain the search pattern. If **False**, the returned array will only consist of the values that do not contain the search pattern.

Code:
```
<% myarray = Split("How now purple cow?") %>
<% myfilterarray = Filter(myarray, "ow", True) %>
<% =Join(myfilterarray) %>
```

Output:
How now cow?

Code:
```
<% myarray = Split("How now purple cow?") %>
<% myfilterarray = Filter(myarray, "ow", False) %>
<% =Join(myfilterarray) %>
```

Output:
purple

Compare

The optional **Compare** argument must only use either the constant or value of the COMPARISON CONSTANTS.

CONSTANT	VALUE	DESCRIPTION
VBBinaryCompare	0	Binary comparison
VBTextCompare	1	Text Comparison
VBDataBaseCompare	2	Compare information inside database

In the example, by using VBBinaryCompare, or 0, for the **Compare** argument, all upper/lower case differences are obeyed in the search.

Code:
```
<% myarray = Split("How now purple cow?") %>
<% myfilterarray = Filter(myarray, "OW", True, 0) %>
<% =Join(myfilterarray) %>
```

Output:
(No output, because no match)

In the example, by using VBTextCompare, or 1, for the **Compare** argument, all upper/lower case differences are ignored in the search.

Code:
```
<% myarray = Split("How now purple cow?") %>
<% myfilterarray = Filter(myarray, "OW", True, VBTextCompare) %>
<% =Join(myfilterarray) %>
```

Output:
How now cow?

FUNCTION: Fix( )

Implemented in version 1.0

Fix(Number)

The **Fix** function converts a decimal number (floating-point) to an integer number (fix-point).

There is a companion function **Int** that also converts to integers. There is one major difference between **Int** and **Fix**. **Int** rounds negative numbers down. **Fix** rounds negative numbers up.

Code:
```
<% =Fix(123.456) %>
```

Output:
123

Positive numbers are not rounded up. The decimal point and all digits to the right are effectively chopped off.

Code:
```
<% =Fix(123.899) %>
```

Output:
123

Negative numbers can also be converted. Negative numbers are rounded up (towards more positive). The decimal point and all digits to the right are effectively chopped off.

Code:
```
<% =Fix(-123.899) %>
```

Output:
-123

OBJECT: Folder

Implemented in version 3.0

The **Folder** object allows you access and manipulate the various properties of a folder.

The following code uses the GetFolder method of the **FileSystemObject** object to obtain a **Folder** object and view one of its properties.

Code:

```
<%
Dim filesys, demofolder, createdate
Set filesys = CreateObject("Scripting.FileSystemObject")
Set demofolder = filesys.GetFolder("foldername")
createdate = demofolder.DateCreated
%>
```

PROPERTIES

Attributes Property

This property allows us to get or change the various attributes of a file.

Syntax: **object.Atributes [= newattributes]**

DateCreated Property

This property gets the date and time that the folder was created.

Syntax: **object.DateCreated**

DateLastAccessed Property

Gets the date and time that the folder was last accessed.

Syntax: **object.DateLastAccessed**

DateLastModified Property

This property gets the date and time that the folder was last modified.

Syntax: **object.DateLastModified**

Drive Property

Returns the drive letter of the drive where the folder is located.

Syntax: **object.Drive**

Files Property

Returns a **Files** collection consisting of all the **File** objects in the specified folder.

Syntax: **object.Files**

IsRootFolder Property

Returns a Boolean value: **True** if the folder is the root folder, and **False** otherwise.

Syntax: **object.IsRootFolder**

Name Property

This property lets us get or change the name of the specified folder.

Syntax: **object.Name [= newname]**

ParentFolder Property
This property gets the parent **Folder** object relating to the specified folder.

Syntax: **object.ParentFolder**

Path Property
This property returns a folder's path.

Syntax: **object.Path**

ShortName Property
Returns the short version of a folder name (using the 8.3 convention). e.g. the folder 'testingshortname' is truncated to 'testin~1'.

Syntax: **object.ShortName**

ShortPath Property
Returns the short version of the folder path (this is the path with any folder and file names truncated as above).

Syntax: **object.ShortPath**

Size Property
Returns the size of the specified folder and its contents (in bytes).

Syntax: **object.Size**

SubFolders Property
This property returns a **Folders** collection that consists of all the folders in the specified folder.

Syntax: **object.SubFolders**

Type Property
Returns a string containing the folder type description.

Syntax: **object.Type**

METHODS

Copy Method
Copies the specified folder from one location to another.

Syntax: **object.Copy destination[, overwrite]**

CreateTextFile Method
This method is used to create a text file and returns a **TextStreamObject** that can then be used to write to and read from the file.

Syntax: **object.CreateTextFile(filename,[, overwrite[, unicode]]))**

Delete Method
Deletes the specified folder.

Syntax: **object.Delete [force]**

Move Method

Moves the specified folder from one location to another.

Syntax: **object.Move destination**

Property: Folder.Attributes

Implemented in version 2.0

object.Attributes [= newattributes]

This property allows us to get or change the various attributes of a folder. The available attribute values are listed below.

Name	Value	Description	Read/Write attribute
Normal	0	Normal file	Read/write
ReadOnly	1	Read-only file	Read only
Hidden	2	Hidden file	Read/write
System	4	System file	Read/write
Volume	8	Disk drive volume label	Read only
Directory	16	Folder or directory	Read-only
Archive	32	File has changed since last backup	Read/write
Alias	64	Link or shortcut	Read-only
Compressed	2048	Compressed file	Read-only

The following code shows how to check if a folder is read/write or read-only. As you can see, logical operators are used to get or change the various attributes.

Code:

```
<%
dim filesys, demofolder
set filesys = CreateObject("Scripting.FileSystemObject")
set demofolder = filesys.GetFolder("c:\Projects\")
If Not demofolder.Attributes And 1 Then
   Response.Write "The folder is Read/Write."
Else
   Response.Write "The folder is Read-only."
End If
%>
```

Output:

"The folder is Read/Write."

METHOD: Folder.Copy

Implemented in version 2.0

object.Copy destination[, overwrite]

This method makes a copy of the selected folder and saves it to the specified destination. The **overwrite** parameter is a Boolean value - **True** allows existing folders with the same name as **destination** to be overwritten while **False** (the default) does not.

Note that after copying a folder to a new location the **Folder** object still relates to the original folder.

Code:

```
<%
dim filesys, demofolder
set filesys = CreateObject ("Scripting.FileSystemObject")
set demofolder = filesys.GetFolder("c:\projects\")
demofolder.Copy("c:\work\")
%>
```

METHOD: Folder.CreateTextFile

Implemented in version 2.0

object.CreateTextFile filename [, overwrite[, unicode]]

This method is used to create a text file and returns a **TextStreamObject** that can then be used to write to and read from the file.

The optional **overwrite** parameter returns a Boolean value - **True** (the default) permits overwriting of existing files while **False** does not. The other optional parameter, **unicode**, is also a Boolean. In this case, **True** creates a Unicode file and **False** (the default) creates an AscII file.

This example creates a text file called, somefile.txt, with the following path: c:\projects\demofolder\somefile.txt

Code:

```
<%
dim filesys, demofolder, filetxt
Set filesys = CreateObject("Scripting.FileSystemObject")
Set demofolder = filesys.GetFolder("c:\projects\")
Set filetxt = demofolder.CreateTextFile("somefile.txt", True)
filetxt.WriteLine("Your text goes here.")
filetxt.Close
%>
```

METHOD: Folder.Delete

Implemented in version 2.0

object.Delete [force]

The method used to delete the folder (and any contents) relating to the specified **Folder** object. The **force** parameter is a Boolean value - **True** allows the deletion of read-only folders while **False** (the default) does not.

Note that trying to delete a folder that doesn't exist will cause an error.

Code:

```
<%
dim filesys, demofolder
set filesys = CreateObject ("Scripting.FileSystemObject")
set demofolder = filesys.GetFolder("c:\projects\")
demofolder.Delete
%>
```

METHOD: Folder.Move

Implemented in version 2.0

object.Move destination

This method is used to move the folder relating to the specified **Folder** object to a new destination.

It is recommended that you use the **FolderExists** method of the **FileSystemObject** object to determine if a folder with the same name already exists in the destination location. If it does, and you attempt the **Move**, you'll receive an error.

Code:

```
<%
dim filesys, demofolder
set filesys = CreateObject ("Scripting.FileSystemObject")
set demofolder = filesys.GetFolder("c:\projects\")
demofolder.Move ("c:\work\")
%>
```

STATEMENT: For Each

Implemented in version 2.0

For Each . . . Next

The **For Each** conditional statement repeats a block of code for each element of an array or a collection of data.

You can use **Exit For** statements to exit out of a **For Each** loop. You can place **For Each** statements inside of other conditional statements. You must end all **For Each** statements with **Next** or you will get an error message.

In the example, the variable i will assume the value of each element in the array, one at a time, in order, from the first element in the array up to the last element actually being used.

Code:

```
<%
For Each i in myarray
   Rem You can place all of the code you desire inside a For Each loop
Next
%>
```

STATEMENT: For

Implemented in version 1.0

For . . . Next

The **For** conditional statement repeats a block of code a specified number of times.

You must end all **For** statements with **Next** or you will get an error message. You can place **For** statements inside of other conditional statements. You can use **Exit For** statements to exit out of a **For** loop. The keyword **Step** allows you to loop through a **For** statement in any size of increment.

Code:

```
<%
For i = 1 To 100
  Rem You can place all of the code you desire inside a For loop
Next
%>

<%
For k = 100 To 1 Step -1
  Rem You can place all of the code you desire inside a For loop
Next
%>

<%
For mynum = 200 To 200000 Step 200
  Rem You can place all of the code you desire inside a For loop
Next
%>
```

FUNCTION: FormatCurrency( )

Implemented in version 2.0

FormatCurrency(Expression, NumDigitsAfterDecimal, IncludeLeadingDigit, UseParensForNegativeNumbers, GroupDigit)

The **FormatCurrency** function return a formatted currency value for the numeric expression.

There is one mandatory argument.

Expression

The **Expression** argument is the number to be converted to a currency format.

Code:
```
<% =FormatCurrency(31567) %>
```

Output:
$31,567.00

Note that this function rounds off values.

Code:
```
<% =FormatCurrency(31567.8977) %>
```

Output:
$31,567.90

There are 4 optional arguments.

NumDigitsAfterDecimal
The optional **NumDigitsAfterDecimal** argument allows you to choose the number of digits after the decimal.

Code:
```
<% =FormatCurrency(31567, 2) %>
```

Output:
$31,567.00

IncludeLeadingDigit

The optional **IncludeLeadingDigit** argument includes the leading zero.

You must only use the constant or value from the Tristate CONSTANTS for this argument.

CONSTANT	VALUE	DESCRIPTION
TristateTrue	-1	True, will use options
TristateFalse	0	False, will not use options

TristateUseDefault	-2	Use default setting

Code:
```
<% =FormatCurrency(.77, 2, -1) %>
```

Output:
$0.77

UseParensForNegativeNumbers

The optional **UseParensForNegativeNumber** argument replaces a negative sign with parentheses around the number.

You must only use the constant or value from the Tristate CONSTANTS for this argument.

CONSTANT	VALUE	DESCRIPTION
TristateTrue	-1	True, will use options
TristateFalse	0	False, will not use options
TristateUseDefault	-2	Use default setting

Code:
```
<% =FormatCurrency(-31567, 2, 0, -1) %>
```

Output:
($31,567.00)

GroupDigit

The optional **GroupDigit** argument allows the use of the options specified in the Currency tab in the Regional Settings Properties in the Control Panel.

You must only use the constant or value from the Tristate CONSTANTS for this argument.

CONSTANT	VALUE	DESCRIPTION
TristateTrue	-1	True, will use options
TristateFalse	0	False, will not use options
TristateUseDefault	-2	Use default setting

Code:
```
<% =FormatCurrency(31567, 2, 0, -1, -1) %>
```

Output:
$31,567.00

FUNCTION: FormatDateTime( )

Implemented in version 2.0

FormatDateTime (Date, DateFormat)

The **FormatDateTime** function formats dates and times. The output format is: MM/DD/YYYY

There is one mandatory argument.

Date

The **Date** argument is any valid date expression.

Code:
```
<% =FormatDateTime("6/26/43") %>
<% =FormatDateTime("15:34") %>
```

Output:
6/26/1943
3:34:00 PM

There is one optional argument.

DateFormat

The optional **DateFormat** argument must use the constant or value from the Date Format CONSTANTS.

CONSTANT	VALUE	DESCRIPTION
VBGeneralDate	0	Display the date and time using system settings
VBLongDate	1	Display the date in long date format June 26, 1943
VBShortDate	2	Display the date in short date format 6/26/43
VBLongTime	3	Display the time in long time format 3:48:01 PM
VBShortTime	4	Display the time in short time format (24 hour clock) 15:48

Code:
```
<% =FormatDateTime("6/26/1943", 1) %>
<% =FormatDateTime("3:34:00 PM", 4) %>
```

Output:
Saturday, June 26, 1943
15:34

FUNCTION: FormatNumber( )

Implemented in version 2.0

FormatNumber(Expression, NumDigitsAfterDecimal, IncludeLeadingDigit, UseParensForNegativeNumbers, GroupDigit)

The **FormatNumber** function return a formatted number value for the numeric expression.

There is one mandatory argument.

Expression

The **Expression** argument is the number to be converted to a formatted number.

Code:
```
<% =FormatNumber(12345) %>
```

Output:
12,345.00

Note that this function rounds off values.

Code:
```
<% =FormatNumber(12345.67899) %>
```

Output:
12,345.68

There are 4 optional arguments.

NumDigitsAfterDecimal

The optional **NumDigitsAfterDecimal** argument allows you to choose the number of digits after the decimal.

Code:
```
<% =FormatNumber(12345.67899, 4) %>
```

Output:
12,345.6790

IncludeLeadingDigit

The optional **IncludeLeadingDigit** argument includes the leading zero.

You must only use the constant or value from the Tristate CONSTANTS for this argument.

CONSTANT	VALUE	DESCRIPTION
TristateTrue	-1	True, will use options

TristateFalse	0	False, will not use options
TristateUseDefault	-2	Use default setting

Code:
<% =FormatNumber(.77, 4, -1) %>

Output:
0.7700

UseParensForNegativeNumbers

The optional **UseParensForNegativeNumber** argument replaces a negative sign with parentheses around the number.

You must only use the constant or value from the Tristate CONSTANTS for this argument.

CONSTANT	VALUE	DESCRIPTION
TristateTrue	-1	True, will use options
TristateFalse	0	False, will not use options
TristateUseDefault	-2	Use default setting

Code:
<% =FormatNumber(-12345.67899, 2, 0, -1) %>

Output:
(12,345.68)

GroupDigit

The optional **GroupDigit** argument allows the use of the options specified in the Regional Settings Properties in the Control Panel to display a number.

You must only use the constant or value from the Tristate CONSTANTS for this argument.

CONSTANT	VALUE	DESCRIPTION
TristateTrue	-1	True, will use options
TristateFalse	0	False, will not use options
TristateUseDefault	-2	Use default setting

Code:
<% =FormatNumber(12345.67899, 2, 0, -1, -1) %>

Output:
12,345.68

FUNCTION: FormatPercent( )

Implemented in version 2.0

FormatPercent(Expression, NumDigitsAfterDecimal, IncludeLeadingDigit, UseParensForNegativeNumbers, GroupDigit)

The **FormatPercent** function return a formatted percent value for the numeric expression with a following percent sign (%). A 1.0 converts to 100% and 0.0 converts to 0%.

There is one mandatory argument.

Expression

The **Expression** argument is the number to be converted to a formatted percent.

Code:
```
<% =FormatPercent(.77) %>
```

Output:
77.00%

Note that this function rounds off values.

Code:
```
<% =FormatPercent(.678999) %>
```

Output:
67.90%

There are 4 optional arguments.

NumDigitsAfterDecimal

The optional **NumDigitsAfterDecimal** argument allows you to choose the number of digits after the decimal.

Code:
```
<% =FormatPercent(.123456789, 4) %>
```

Output:
12.3457%

IncludeLeadingDigit

The optional **IncludeLeadingDigit** argument includes the leading zero.

You must only use the constant or value from the Tristate CONSTANTS for this argument.

CONSTANT	VALUE	DESCRIPTION
TristateTrue	-1	True, will use options

TristateFalse	0	False, will not use options
TristateUseDefault	-2	Use default setting

Code:
<% =FormatPercent(.0098, 4, -1) %>

Output:
0.9800%

UseParensForNegativeNumbers

The optional **UseParensForNegativeNumber** argument replaces a negative sign with parentheses around the number.

You must only use the constant or value from the Tristate CONSTANTS for this argument.

CONSTANT	VALUE	DESCRIPTION
TristateTrue	-1	True, will use options
TristateFalse	0	False, will not use options
TristateUseDefault	-2	Use default setting

Code:
<% =FormatPercent(-.77, 2, 0, -1) %>

Output:
(77.00%)

GroupDigit

The optional **GroupDigit** argument allows the use of the options specified in the Regional Settings Properties in the Control Panel to display a percent.

You must only use the constant or value from the TRISATE CONSTANTS for this argument.

CONSTANT	VALUE	DESCRIPTION
TristateTrue	-1	True, will use options
TristateFalse	0	False, will not use options
TristateUseDefault	-2	Use default setting

Code:
<% =FormatPercent(.77, 2, 0, -1, -1) %>

Output:
77.00%

STATEMENT: Function

Implemented in version 1.0

Function . . . End Function

The **Function** statement creates a function, assigns a name, and allows you to list the arguments (if any) which are to be passed into the function.

The major difference between a function and a subroutine, is that a function can return a value back to where it was called from. The returned value must be assigned to the function name (or **Set** if it is an object) somewhere inside the function.

However, you do not not have to assign or **Set** a return value.

As an option, functions can be declared **Private** or **Public** (the default). One option for leaving a **Function** is to use **Exit** statements. You can call other functions or subroutines from within a function (nesting). You must end every function with **End Function** or you will get an error message.

Code:
```
<%
Function aardvark
  Rem you can place all of the code you desire inside a function
End Function
%>
```

Code:
```
<% Private Function aardvark
  Rem you can place all of the code you desire inside a function
End Function
%>
```

You can list more than one argument in the function.

Code:
```
<%
Function aardvark(myvar1, myvar2, mynumber, myarray)
  Rem you can place all of the code you desire inside a function
End Function
%>
```

When you call a function that is assigned to a variable and it has one or more arguments, you must enclosed the arguments inside parentheses, otherwise you will get an error.

Code:
```
<%
returnval = aardvark(myvar1, myvar2, mynumber, myarray)
%>
```

FUNCTION: GetLocale()

GetLocale

The **GetLocale** function returns the LCID (locale ID).

The LCID is a short string, decimal value, or hex value that uniquely identifies a geographic locale. The various geographic locales are based upon the user's language, country, and culture. For example, the locale "English - United States" can be designated as either "en-us", or "1033", or "0x0409".

This locale information is used to establish user preferences and formats for such things as alphabets, currency, dates, keyboard layout, and numbers. A list of these locales, and their return values, can be found here.

Code:

```
<% =GetLocale() %>
```

Output:

```
1033
```

Locale ID (LCID) Reference

Locale Description	Short String	Hex Value	Decimal Value
Afrikaans	af	0x0436	1078
Albanian	sq	0x041C	1052
Arabic - U.A.E.	ar-ae	0x3801	14337
Arabic - Bahrain	ar-bh	0x3C01	15361
Arabic - Algeria	ar-dz	0x1401	5121
Arabic - Egypt	ar-eg	0x0C01	3073
Arabic - Iraq	ar-iq	0x0801	2049
Arabic - Jordan	ar-jo	0x2C01	11265
Arabic - Kuwait	ar-kw	0x3401	13313
Arabic - Lebanon	ar-lb	0x3001	12289
Arabic - Libya	ar-ly	0x1001	4097
Arabic - Morocco	ar-ma	0x1801	6145
Arabic - Oman	ar-om	0x2001	8193
Arabic - Qatar	ar-qa	0x4001	16385
Arabic - Saudia Arabia	ar-sa	0x0401	1025
Arabic - Syria	ar-sy	0x2801	10241
Arabic - Tunisia	ar-tn	0x1C01	7169
Arabic - Yemen	ar-ye	0x2401	9217
Basque	eu	0x042D	1069
Belarusian	be	0x0423	1059
Bulgarian	bg	0x0402	1026
Catalan	ca	0x0403	1027

Chinese	zh	0x0004	4
Chinese - PRC	zh-cn	0x0804	2052
Chinese - Hong Kong	zh-hk	0x0C04	3076
Chinese - Singapore	zh-sg	0x1004	4100
Chinese - Taiwan	zh-tw	0x0404	1028
Croatian	hr	0x041A	1050
Czech	cs	0x0405	1029
Danish	da	0x0406	1030
Dutch	nl	0x0413	1043
Dutch - Belgium	nl-be	0x0813	2067
English	en	0x0009	9
English - Australia	en-au	0x0C09	3081
English - Belize	en-bz	0x2809	10249
English - Canada	en-ca	0x1009	4105
English - Ireland	en-ie	0x1809	6153
English - Jamaica	en-jm	0x2009	8201
English - New Zealand	en-nz	0x1409	5129
English - South Africa	en-za	0x1C09	7177
English - Trinidad	en-tt	0x2C09	11273
English - United Kingdom	en-gb	0x0809	2057
English - United States	en-us	0x0409	1033
Estonian	et	0x0425	1061
Farsi	fa	0x0429	1065
Finnish	fi	0x040B	1035
Faeroese	fo	0x0438	1080

French - Standard	fr	0x040C	1036
French - Belgium	fr-be	0x080C	2060
French - Canada	fr-ca	0x0C0C	3084
French - Luxembourg	fr-lu	0x140C	5132
French - Switzerland	fr-ch	0x100C	4108
Gaelic - Scotland	gd	0x043C	1084
German - Standard	de	0x0407	1031
German - Austrian	de-at	0x0C07	3079
German - Lichtenstein	de-li	0x1407	5127
German - Luxembourg	de-lu	0x1007	4103
German - Switzerland	de-ch	0x0807	2055
Greek	el	0x0408	1032
Hebrew	he	0x040D	1037
Hindi	hi	0x0439	1081
Hungarian	hu	0x040E	1038
Icelandic	is	0x040F	1039
Indonesian	in	0x0421	1057
Italian - Standard	it	0x0410	1040
Italian - Switzerland	it-ch	0x0810	2064
Japanese	ja	0x0411	1041
Korean	ko	0x0412	1042
Latvian	lv	0x0426	1062
Lithuanian	lt	0x0427	1063
Macedonian	mk	0x042F	1071
Malay - Malaysia	ms	0x043E	1086

Maltese	mt	0x043A	1082
Norwegian - Bokmål	no	0x0414	1044
Polish	pl	0x0415	1045
Portuguese - Standard	pt	0x0816	2070
Portuguese - Brazil	pt-br	0x0416	1046
Raeto-Romance	rm	0x0417	1047
Romanian	ro	0x0418	1048
Romanian - Moldova	ro-mo	0x0818	2072
Russian	ru	0x0419	1049
Russian - Moldova	ru-mo	0x0819	2073
Serbian - Cyrillic	sr	0x0C1A	3098
Setsuana	tn	0x0432	1074
Slovenian	sl	0x0424	1060
Slovak	sk	0x041B	1051
Sorbian	sb	0x042E	1070
Spanish - Standard	es	0x040A	1034
Spanish - Argentina	es-ar	0x2C0A	11274
Spanish - Bolivia	es-bo	0x400A	16394
Spanish - Chile	es-cl	0x340A	13322
Spanish - Columbia	es-co	0x240A	9226
Spanish - Costa Rica	es-cr	0x140A	5130
Spanish - Dominican Republic	es-do	0x1C0A	7178
Spanish - Ecuador	es-ec	0x300A	12298
Spanish - Guatemala	es-gt	0x100A	4106
Spanish - Honduras	es-hn	0x480A	18442

Spanish - Mexico	es-mx	0x080A	2058
Spanish - Nicaragua	es-ni	0x4C0A	19466
Spanish - Panama	es-pa	0x180A	6154
Spanish - Peru	es-pe	0x280A	10250
Spanish - Puerto Rico	es-pr	0x500A	20490
Spanish - Paraguay	es-py	0x3C0A	15370
Spanish - El Salvador	es-sv	0x440A	17418
Spanish - Uruguay	es-uy	0x380A	14346
Spanish - Venezuela	es-ve	0x200A	8202
Sutu	sx	0x0430	1072
Swedish	sv	0x041D	1053
Swedish - Finland	sv-fi	0x081D	2077
Thai	th	0x041E	1054
Turkish	tr	0x041F	1055
Tsonga	ts	0x0431	1073
Ukranian	uk	0x0422	1058
Urdu - Pakistan	ur	0x0420	1056
Vietnamese	vi	0x042A	1066
Xhosa	xh	0x0434	1076
Yiddish	ji	0x043D	1085
Zulu	zu	0x0435	1077

FUNCTION: GetObject( )

Implemented in version 2.0

GetObject(PathName, Class)

The **GetObject** function is used to access an automation object in a file and to assign that object to a object variable.

There are two optional arguments.

(PathName, Class)

The optional **PathName** argument is the full path to and the name of object file to be accessed.

If the **PathName** argument is not used, you must use the **Class Argument**.

The optional **Class** argument is the name of the application providing the object or the class of the object to create.

Code:

```
<% thepath = "C:\subdirectory\myobject.txt" %>
<% Set myobject = GetObject(thepath) %>
<% Set myobject = GetObject(thepath, "MyApplication.SomeObject") %>
```

FUNCTION: GetRef( )

Implemented in version 5.0

GetRef(FunctionOrSubName)

The **GetRef** function binds a function or subroutine to an event on a DHTML (Dynamic HTML) page.

The required **ObjectName** is the name of a DHTML object to which the DHTML event is associated.

The required **EventName** is the name of a DHTML event to which the function or subroutine will be bound.

The mandatory **FunctionOrSubName** argument is the name of a VBScript function or subroutine to which the DHTML event is to be associated.

The above must be used with the **Set** keyword in the following manner:

Code:
```
Set ObjectName.EventName = GetRef(FunctionOrSubName)
```

FUNCTION: InputBox( )

Implemented in version 1.0

InputBox(Prompt, Title, Default, Xpos, Ypos, HelpFile, Context)

The **InputBox** function creates a prompt for an input dialog box.

The dialog box usually asks a question and prompts a reply from the user.

If the user clicks the OK button or presses ENTER, the **InputBox** function returns all text inside the text box. If the user clicks the CANCEL button, a zero-length string "" is returned.

There is one mandatory argument.

Prompt

The **Prompt** argument is the message string (question) that appears in the dialog box.

Code:
```
<% INPUT TYPE="BUTTON" NAME="button0" VALUE="Click Here!" %>

< SCRIPT LANGUAGE="VBScript" >
Sub button0_onclick
  Dim returntext
  returntext = InputBox("This is an input box!")
End Sub
< /SCRIPT >
```

Output:

There are six optional arguments.

Title

The optional **Title** argument is the title that appears at the top of the dialog box window.

Code:
```
<% INPUT TYPE="BUTTON" NAME="button1" VALUE="Click Here!" %>

< SCRIPT LANGUAGE="VBScript" >
Sub button1_onclick
  Dim returntext promptext
  prompttext = "This is an input box!"
  prompttext = prompttext &(chr(13) & chr(10))
  prompttext = prompttext & "We added a second line!"
  returntext = InputBox(prompttext, "INPUT Function")
End Sub
< /SCRIPT >
```

Output:

Click Here!

Default

The optional **Default** argument is the text that will appear in the reply window in the dialog box.

Code:

```
<% INPUT TYPE="BUTTON" NAME="button2" VALUE="Click Here!" %>

< SCRIPT LANGUAGE="VBScript" >
Sub button2_onclick
  Dim returntext promptext titletext
  prompttext = "This is an input box!"
  titletext = "INPUT function"
  returntext = InputBox(prompttext, titletext, "Any text you wish...")
End Sub
< /SCRIPT >
```

Output:

Click Here!

Xpos

The optional **Xpos** argument determines the horizontal position of the dialog box in the viewing screen. It is the number of twips from the left side of the screen to the left edge of the dialog box. Twips are a graphical value used to set the ScaleMode property of an object.

Ypos

The optional **Ypos** argument determines the vertical position of the dialog box in the viewing screen. It is the number of twips from the top of the screen to the top edge of the dialog box. Twips are a graphical value used to set the ScaleMode property of an object.

Code:

```
<% INPUT TYPE="BUTTON" NAME="button3" VALUE="Click Here!" %>

< SCRIPT LANGUAGE="VBScript" >
Sub button3_onclick
  Dim returntext promptext titletext
  prompttext = "This is an input box!"
  titletext = "INPUT function"
  defaulttext = "Upper left corner"
  returntext = InputBox(prompttext, titletext, defaulttext, 150, 150)
End Sub
< /SCRIPT >
```

Output:

Click Here!

HelpFile
Context

The optional **HelpFile** argument specifies the location of the help file.

The optional **Context** argument specifies the help context number in the help file.

Not demonstrated.

FUNCTION: InStr( )

Implemented in version 1.0

InStr(Start, String, Substring, Compare)

The **InStr** function returns the numeric position of the first occurrence of a specified substring within a specified string when starting from the beginning (left end) of the string. You can have the search for the substring be sensitive to the case (upper versus lower), or not. The default is to be case sensitive (binary comparison).

An output of zero indicates no match.

The first argument is optional.

Start

The optional **Start** argument is the numeric position, counted from the left, which defines where to start the search for the substring. The search proceeds from the left to the right.

There are two mandatory arguments.

String

The **String** argument is the string in which you will search.

Substring

The **Substring** argument is the substring you are searching for.

Code:
```
<% =InStr("ABCDE ABCDE", "C") %>
```

Output:
3

Code:
```
<% =InStr(4, "ABCDE ABCDE", "C") %>
```

Output:
9

The fourth argument is optional.

Compare

The optional **Compare** argument can be used to set whether the search for the substring will be case sensitive, or not. You must only use either the constant or value of the COMPARISON CONSTANTS.

Note that when you use the Compare argument you must use the Start argument.

CONSTANT	VALUE	DESCRIPTION
VBBINARYCOMPARE	0	Binary comparison (case sensitive)
VBTEXTCOMPARE	1	Text Comparison (case insensitive)
VBDATEBASECOMPARE	2	Compare information inside database

In the example, by using VBBinaryCompare, or 0, for the **Compare** argument, all upper/lower case differences are obeyed in the search for the first match.

Code:

```
<% =InStr(1, "ABCDE ABCDE", "c", 0) %>
```

Output:

```
0
```

In the example, by using VBTextCompare, or 1, for the **Compare** argument, all upper/lower case differences are ignored in the search for the first match.

Code:

```
<% =InStr(1, "ABCDE ABCDE", "c", VBTextCompare) %>
```

Output:

```
3
```

FUNCTION: InStrRev( )

Implemented in version 2.0

InStrRev(String, Substring, Start, Compare)

The **InStrRev** function returns the numeric position of the first occurrence of a specified substring within a specified string when starting from the end (right end) of the string. You can have the search for the substring be sensitive to the case (upper versus lower), or not. The default is to be case sensitive (binary comparison).

An output of zero indicates no match.

There are two mandatory arguments.

String

The **String** argument is the string in which you will search.

Substring

The **Substring** argument is the substring you are searching for.

Code:
```
<% =InStrRev("ABCDE ABCDE", "C") %>
```

Output:
9

There are two optional arguments.

Start

The optional **Start** argument is the numeric position, counted from the left, which defines where to start the search for the substring.

In the example the search begins at postion 4, counted from the left, and the search goes from the right to left.

Code:
```
<% =InStrRev("ABCDE ABCDE", "C", 4) %>
```

Output:
3

Compare

The optional **Compare** argument can be used to set whether the search for the substring is case sensitive (upper versus lower), or not. You must only use either the constant or value of the COMPARISON CONSTANTS.

CONSTANT	VALUE	DESCRIPTION

VBBINARYCOMPARE	0	Binary comparison (case sensitive)
VBTEXTCOMPARE	1	Text Comparison (case insensitive)
VBDATABASECOMPARE	2	Compare information inside database

In the example, by using VBBinaryCompare, or 0, for the **Compare** argument, all upper/lower case differences are obeyed in both the search.

Code:

```
<% =InStrRev("ABCDE ABCDE", "c", 4, 0) %>
```

Output:

```
0
```

In the example, by using VBTextCompare, or 1, for the **Compare** argument, all upper/lower case differences are ignored in the search.

Code:

```
<% =InStrRev("ABCDE ABCDE", "c", 4, VBTextCompare %>
```

Output:

```
3
```

FUNCTION: Int( )

Implemented in version 1.0

Int(Number)

The **Int** function converts a decimal number (floating-point) to an integer number (fix-point).

There is one major difference between **Int** and **Fix**. **Int** rounds negative numbers down. **Fix** rounds negative numbers up.

Code:
```
<% =Int(123.456) %>
```

Output:
123

Positive numbers are not rounded up. The decimal point and all digits to the right are effectively chopped off.

Code:
```
<% =Int(123.899) %>
```

Output:
123

Negative numbers are rounded down (towards more negative).

Code:
```
<% =Int(-123.456) %>
```

Output:
-124

OPERATORS: Is

Implemented in version 1.0

Is

The **Is** operator is used to determine if two variables refer to the same object. The output is either **True** or **False**.

Code:

```
<% Set object1 = Request.ServerVariables("SCRIPT_NAME") %>
<% Set object2 = object1 %>
<% =object1 Is object2 %>
```

Output:

True

Code:

```
<% Set object1 = Server.CreateObject("Scripting.FileSystemObject") %>
<% Set object2 = Request.ServerVariables("SCRIPT_NAME") %>
<% =object1 Is object2 %>
```

Output:

False

FUNCTION: IsArray( )

Implemented in version 1.0

IsArray(Expression)

The **IsArray** function determines if the expression is an array.

Code:
```
<% Dim myarray(10) %>
<% =IsArray(myarray) %>
```

Output:
True

Code:
```
<% =IsArray("This is a string.") %>
```

Output:
False

FUNCTION: IsDate( )

Implemented in version 1.0

IsDate(Expression)

The **IsDate** function determines if the expression is a date.

Code:
```
<% mydate = "6/26/1943" %>
<% =IsDate(mydate) %>
```

Output:
True

Code:
```
<% =IsDate("This is a string.") %>
```

Output:
False

FUNCTION: IsEmpty( )

Implemented in version 1.0

IsEmpty(variant)

The **IsEmpty** function returns True when passed a Variant that has been declared but not initialized. It returns False in all other cases.

Code:
```
<% =IsEmpty(notbeen) %>
```

Output:
True

Code:
```
<% =IsEmpty("This is a string.") %>
```

Output:
False

FUNCTION: IsNull( )

Implemented in version 1.0

IsNull(Expression)

The **IsNull** function determines if the expression is **Null**.

Code:
```
<% nothere = null %>
<% =IsNull(nothere) %>
```

Output:
True

Code:
```
<% =IsNull("This is a string.") %>
```

Output:
False

FUNCTION: IsNumeric( )

Implemented in version 1.0

IsNumeric(Expression)

The **IsNumeric** function determines if the expression is a number.

Code:
```
<% =IsNumeric(123.456) %>
```

Output:
True

Code:
```
<% =IsNumeric("This is a string.") %>
```

Output:
False

FUNCTION: IsObject( )

Implemented in version 1.0

IsObject(Expression)

The **IsObject** function determines if the expression is an automation object.

Code:
```
<% Set anyvariable = Server.CreateObject("Scripting.FileSystemObject") %>
<% =IsObject(anyvariable) %>
```

Output:
True

Code:
```
<% =IsObject("This is a string, not an object.") %>
```

Output:
False

FUNCTION: Join( )

Implemented in version 2.0

Join(Array, Delimiter)

The **Join** function combines substrings (elements) of an array together into one long string with each substring separated by the delimiter string.

There is one mandatory argument.

Array

The **Array** argument is the name of the array to be joined.

Code:
```
<% Dim cowarray(10) %>
<% cowarray(0) = "How" %>
<% cowarray(1) = "now" %>
<% cowarray(2) = "brown" %>
<% cowarray(3) = "cow?" %>
<% Join(cowarray) %>
```

Output:
```
How now brown cow?
```

There is one optional argument.

Delimiter

The optional **Delimiter** argument specifies the characters (including blanks) you wish to place between each element. If this argument is not specified, the default is to place a blank space between each element. If you wish to have no spaces, use a double quote, "", for the argument.

Code:
```
<% Dim arraycow(10) %>
<% arraycow(0) = "How" %>
<% arraycow(1) = "now" %>
<% arraycow(2) = "brown" %>
<% arraycow(3) = "cow?" %>

<% =Join(arraycow, 1234) %>
<% =Join(arraycow, "****") %>
<% =Join(arraycow, "") %>
```

Output:
```
How1234now1234brown1234cow?1234123412341234123412341234
How****now****brown****cow?**************************
Hownowbrowncow?
```

FUNCTION: LBound( )

Implemented in version 1.0

LBound(ArrayName, Dimension)

The **LBound** function returns the lower limit for the elements in an array.

By default, in the current versions of **VBScript**, the **LBound** function always returns a zero.

There is one mandatory argument.

ArrayName

The **ArrayName** argument is the name of the array. Do not include the parenthesis with the array name.

Code:
```
<% Dim catarray(3) %>
<% catarray(0) = "Mountain lion" %>
<% catarray(1) = "Bobcat" %>
<% catarray(2) = "Jaguar" %>
<% =LBound(catarray) %>
```

Output:
```
0
```

There is one optional argument.

Dimension

The optional **Dimension** argument is used to identify which index you are determining the lower bounds for in a multi-dimensional array.

Code:
```
<% Dim fisharray(2,3) %>
<% fisharray(0,0) = "Neon tetra" %>
<% fisharray(0,1) = "Angle fish" %>
<% fisharray(0,2) = "Discus" %>
<% fisharray(1,0) = "Golden dojo" %>
<% fisharray(1,1) = "Clown loach" %>
<% fisharray(1,2) = "Betta" %>
<% =LBound(fisharray, 2) %>
```

Output:
```
0
```

FUNCTION: LCase( )

Implemented in version 1.0

LCase(String)

The **LCase** function converts a string to all lower case letters.

There is also a companion function **UCase** to convert to upper case letters.

Code:
```
<% =LCase("Aardvarks do not make good pets!") %>
```

Output:
aardvarks do not make good pets!

You can also use the name of the string.

Code:
```
<% astring="Leopards also make bad pets!" %>
<% =LCase(astring) %>
```

Output:
leopards also make bad pets!

FUNCTION: Left( )

Implemented in version 1.0

Left(String, Length)

The **Left** function returns the left portion of the designated string for a designated length.

There are two mandatory arguments.

String

The **String** argument is the name of the string you wish to truncate.

Length

The **Length** argument is the number of characters (including blanks) you wish to save counting from the left side towards the right side of the string.

Code:

```
<% =Left("abcde fghij klmno pqrst uvwxyz", 13) %>
```

Output:

```
abcde fghij k
```

FUNCTION: Len( )

Implemented in version 1.0

Len(String)
Len(Varname)

The **Len** function returns the length of the string or the byte size of the variable.

You can give the function the string or the string name, **String** or the variable name, **Varname**.

Code:
```
<% =Len("Clouded leopards are an endangered species.") %>
```

Output:
43

Code:
```
<% anystring ="Clouded leopards are an endangered species." %>
<% =Len(anystring) %>
```

Output:
43

Code:
```
<% anyvariable = 123.456 %>
<% =Len(anyvariable) %>
```

Output:
7

Property: TextStream.Line

Implemented in version 2.0

object.Line

The **Line** property is used to return the current line number in a text file. Upon opening a textfile and before any writing takes place, **Line** always returns '1'.

Code:

```
<%
dim filesys, text, readfile, contents, col, linumber
set filesys = CreateObject("Scripting.FileSystemObject")
Set text = filesys.CreateTextFile("c:\somefile2.txt")
text.Write "Find the last character in the text file"
text.close
set readfile = filesys.OpenTextFile("c:\somefile2.txt", 1, false)
do while readfile.AtEndOfStream <> true
    contents = readfile.Read(1)
    col = readfile.Column -1
    linumber = readfile.Line
loop
readfile.close
Response.Write "The last character in line " & linumber & " is '" & contents & "' and_
                        its column number is '" & col & "'."
%>
```

Output:

"The last character in line 1 is 'e' and its column number is '40'."

FUNCTION: LoadPicture( )

Implemented in version 2.0

LoadPicture(PictureName)

The **LoadPicture** function returns a reference to a picture object (i.e., loads the picture).

The **PictureName** argument is the name of the picture file.

Code:

```
<% LoadPicture(mypicture.gif) %>
```

FUNCTION: Log( )

Implemented in version 1.0

Log(Number)

The **Log** function returns the natural logarithm of a number.

There is a companion function **Exp** for the reverse operation.

You can not use a negative number.

Code:

```
<% =Log(26.2850411552082) %>
```

Output:

```
3.269
```

FUNCTION: LTrim( )

Implemented in version 1.0

LTrim(String)

The **LTrim** function removes extra blank spaces only on the left side of a string.

Note that HTML automatically removes extra blank spaces in this view.

Code:
```
<% =LTrim("   A sentence with extra spaces on both sides.    ") %>
```

Output:
A sentence with extra spaces on both sides.

OBJECT: Match

Implemented in version 5.0

The **Match** object is used to access the three read-only properties associated with the results of a search and match operation that uses a regular expression.

Simply put, a regular expression is a string pattern that you can compare against all or a portion of another string. However, in all fairness, be warned that regular expressions can get very complicated.

The **RegExp** object can be used to search for and match string patterns in another string. A **Match** object is created each time the **RegExp** object finds a match. Since, zero or more matches could be made, the **RegExp** object actually return a collection of **Match** objects which is refered to as a **Matches** collection.

The following code is a simplier, working version of a program published by Microsoft.

Code:

```
<%
'this sub finds the matches
Sub RegExpTest(strMatchPattern, strPhrase)
   'create variables
   Dim objRegEx, Match, Matches, StrReturnStr
   'create instance of RegExp object
   Set objRegEx = New RegExp

   'find all matches
   objRegEx.Global = True
   'set case insensitive
   objRegEx.IgnoreCase = True
   'set the pattern
   objRegEx.Pattern = strMatchPattern

   'create the collection of matches
   Set Matches = objRegEx.Execute(strPhrase)

   'print out all matches
   For Each Match in Matches
      strReturnStr = "Match found at position "
      strReturnStr = strReturnStr & Match.FirstIndex & ". Match Value is '"
      strReturnStr = strReturnStr & Match.value & "'." & "<BR>" & VBCrLf
      'print
      Response.Write(strReturnStr)
  :  Next
End Sub

'call the subroutine
RegExpTest "is.", "Is1 is2 Is3 is4"
%>
```

Output:

```
Match found at position 0. Match Value is 'Is1'.
Match found at position 4. Match Value is 'is2'.
```

Match found at position 8. Match Value is 'Is3'.
Match found at position 12. Match Value is 'is4'.

PROPERTIES

FirstIndex Property
This property returns the position, counted from the left with the first position being numbered zero, in a string where a match was made.

Syntax: **Match.FirstIndex**

Length Property
This property returns the length of the matched text found in a search string.

Syntax: **Match.Length**

Value Property
This property returns the actual text that was matched during the search.

Syntax: **Match.Value**

FUNCTION: Mid( )

Implemented in version 1.0

Mid(String, Start, Length)

The **Mid** function returns the portion of the designated string for a designated length starting from any position in the string.

There are two mandatory arguments.

String

The **String** argument is the name of the string you wish to truncate.

Start

The **Start** argument is a numeric position anywhere in the string counting from the left side. The portion of the string to the left of this position will be discarded.

Code:
```
<% =Mid("abcde fghij klmno pqrst uvwxyz", 8) %>
```

Output:
```
ghij klmno pqrst uvwxyz
```

There is one optional argument.

Length

The optional **Length** argument is the number of characters (including blanks) in the string you wish to save counting from the position designated by the **Start** argument.

Code:
```
<% =Mid("abcde fghij klmno pqrst uvwxyz", 8, 7) %>
```

Output:
```
ghij kl
```

FUNCTION: Minute( )

Implemented in version 1.0

Minute(Time)

The **Minute** function returns the minute by using the **Time** function as an argument.

Values will run from 0 to 59.

Code:
```
<% =Minute(Time) %>
```

Output:
7

OPERATOR: Mod

Implemented in version 1.0

Mod

The **Mod** operator divides two numbers and returns the remainder.

In the example, 5 divides into 26, 5 times, with a remainder of 1.

Code:
```
<% remainder = 26 Mod 5 %>
```

Output:
1

Note that for floating-point numbers, the numbers to the right of the decimal are simply ignored (i.e., in the example, 5.7176 is treated as 5 by **Mod**).

Code:
```
<% remainder = 26.1234 Mod 5.7176 %>
```

Output:
1

FUNCTION: Month( )

Implemented in version 1.0

Month(Date)

The **Month** function returns the number of the current month using using any valid date expression as an argument.

You can also use the **Date** and **Now** functions as the argument.

Code:
```
<% =Month(Date) %>
```

Output:
3

Code:
```
<% =Month("June 26, 1943") %>
```

Output:
6

FUNCTION: MonthName( )

Implemented in version 2.0

MonthName(Month, Abbreviate)

The **MonthName** function returns the name of the month.

There is one mandatory argument.

Month

The **Month** argument is the number of the month (i.e., 1 throught 12).

Code:
```
<% =MonthName(6) %>
```

Output:
June

There is one optional argument.

Abbreviate

The **Abbreviate** argument is a **Boolean** value which gives the option of having the returned month name being abbreviated to the first three characters.

If set to **True**, the name will be abbreviated. If set to **False** the name will not be abbreviated.

Code:
```
<% =MonthName(6, True) %>
```

Output:
Jun

CONSTANTS: MsgBox

Implemented in version 2.0

MsgBox Constants

You may use either the CONSTANT (left column) or the VALUE (center column) in your code and get the same results.

Specifying buttons and icons:

CONSTANT	VALUE	DESCRIPTION
VBOKOnly	0	Show OK button
VBOKCancel	1	Show OK and cancel buttons
VBAbortRetryIgnore	2	Show abort, retry, ignore buttons
VBYesNoCancel	3	Show yes, no cancel buttons
VBYesNo	4	Show yes, no buttons
VBRetryCancel	5	Show retry, cancel buttons
VBCritical	16	Show critical message icon
VBQuestion	32	Show warning query button
VBExclaimation	48	Show warning message icon
VBInformation	64	Show information message icon
VBFaultButton1	0	First button is default
VBFaultButton2	256	Second button is default
VBFaultButton3	512	Third button is default
VBFaultButton4	768	Fourth button is default

Message box response modality:

CONSTANT	VALUE	DESCRIPTION
VBApplicationModal	0	Current application will not continue until user responds to message box
VBSystemModal	4098	No application will continue until user responds to message box

Return values identifying which buttons were selected:

CONSTANT	VALUE	DESCRIPTION
VBOK	1	OK Button selected
VBCancel	2	Cancel button selected
VBAbort	3	Abort button selected
VBRetry	4	Retry button selected
VBIgnore	5	Ignore button selected
VBYes	6	Yes button selected
VBNo	7	No button selected

FUNCTION: MsgBox( )

Implemented in version 1.0

MsgBox(Prompt, Buttons, Title)

MsgBox(Prompt, Buttons, HelpFile, Context)

The **MsgBox** function creates a dialog box with a specified message and prompts the user to click a button, upon which the dialog box closes and a value relating to which button was clicked is returned. These values, along with their Constants, are listed in the table below.

CONSTANT	VALUE	BUTTON
VBOK	1	OK
VBCancel	2	Cancel
VBAbort	3	Abort
VBRetry	4	Retry
VBIgnore	5	Ignore
VBYes	6	Yes
VBNo	7	No

Click on the buttons to see a display of the various available types of Message Box.

Click Here! OK
Click Here! OK, Cancel
Click Here! Abort, Retry, Ignore
Click Here! Yes, No, Cancel
Click Here! Yes, No
Click Here! Retry, Cancel

Click Here! Critical OK
Click Here! Warning Query OK
Click Here! Warning Message OK
Click Here! Info O
Click Here! Warning Query, Abort, Retry, Cancel

There is one mandatory argument.

Prompt

The **Prompt** argument is the message string that appears in the message box.

The default message box is the OK.

Code:

```
< INPUT TYPE="BUTTON" NAME="button0" VALUE="Click Here!" >

< SCRIPT LANGUAGE="VBScript" >
Sub button0_onclick
   MsgBox "Please Click OK"
End Sub
< /SCRIPT >
```

Output:

There are four optional arguments.

Buttons

The optional **Buttons** argument must only use the constant or value in the MsgBox CONSTANTS.

CONSTANT	VALUE	DESCRIPTION
VBOKOnly	0	Show OK button
VBOKCancel	1	Show OK and cancel buttons
VBAbortRetryIgnore	2	Show abort, retry, ignore buttons
VBYesNoCancel	3	Show yes, no cancel buttons
VBYesNo	4	Show yes, no buttons
VBRetryCancel	5	Show retry, cancel buttons
VBCritical	16	Show critical message icon
VBQuestion	32	Show warning query button
VBExclamation	48	Show warning message icon
VBInformation	64	Show information message icon
VBDefaultButton1	0	First button is default
VBDefaultButton2	256	Second button is default
VBDefaultButton3	512	Third button is default
VBDefaultButton4	768	Fourth button is default

VBApplicationModal	0	Demands that the user respond to the dialog before allowing continuation of work in current application
VBSystemModal	4096	Causes suspension of all applications until the user responds to the dialog

Code:

```
< INPUT TYPE="BUTTON" NAME="button_1" VALUE="Click Here!"> >

< SCRIPT LANGUAGE="VBScript" >
Sub button_1_onclick
  MsgBox "Please Click OK", VBOKCancel
End Sub
< /SCRIPT >
```

Output:

Note you may use either the Title argument or the HelpFile, Context arguments. You cannot use both at the same time.

Title

The optional **Title** argument is the title that appears at the top of the message box window.

Code:

```
< INPUT TYPE="BUTTON" NAME="button_2" VALUE="Click Here!"> >

< SCRIPT LANGUAGE="VBScript" >
Sub button_2_onclick
  MsgBox "Please Click!", VBRetryCanel, "MsgBox Demo"
End Sub
< /SCRIPT >
```

Output:

HelpFile

The optional **HelpFile** argument is a string that specifies the help file that you wish to display. This must be either a .chm or .hlp file.

Context

The optional **Context** argument specifies the help context number in the help file of the topic you wish to display. If you have created your own custom help file, then the **Context** argument is mandatory.

Code:

```
MsgBox "Date is not valid", vbMsgBoxHelpButton, "help_folder/date_help_file.hlp", 71
```

Output:

Not demonstrated.

OPERATOR: Not

Implemented in version 1.0

Not

The **Not** operator is used to perform a logical negation on an expression. The expression must be of **Boolean** subtype and have a value of **True** or **False**. This operator causes a **True** expression to become **False**, and a **False** expression to become **True**.

The **Not** operator can also be used a "bitwise operator" to make a bit-by-bit inversion of the binary values of an integer. If the bit is a 1, then a 0 is returned. If the bit is a 0, then a 1 is returned. Thus, the binary value 101 becomes 010.

Code:

```
<% AnyExpression = True %>
<% =Not AnyExpression %>
```

Output:

```
False
```

FUNCTION: Now

Implemented in version 1.0

Now

The **Now** function returns the date and time as determined by your computer.

Code:
```
<% =Now %>
```

Output:
Friday, February 08, 2002 12:31:30

CONSTANTS: VBObjectError

Implemented in version 2.0

VBObjectError Constant

CONSTANT	VALUE
VBObjectError	-2147221504

FUNCTION: Oct( )

Implemented in version 1.0

Oct(Number)

The **Oct** function returns the octal value of an integer number.

Code:
```
<% =Oct(123) %>
```

Output:
173

You can also use a negative integer number.

Code:
```
<% =Oct(-123) %>
```

Output:
177605

For a number with a decimal point (floating-point), the digits to the right of the decimal point are ignored.

Code:
```
<% =Oct(123.456) %>
```

Output:
173

STATEMENT: On Error

Implemented in version 1.0

On Error Resume Next
On Error GoTo 0

The **On Error Resume Next** statement gives you a limited amount of error handling control by preventing program interruptions from runtime errors. When an error occurs, by using this statement, the line of code containing the error is simply skipped over and the program continues running.

Note that the error is not corrected, just ignored, and an error message is not displayed. Of course, your program may still crash or give erroneous output if the error involves a value required to successfully execute later portions of your code.

Code:
```
<% On Error Resume Next %>
```

The **On Error GoTo 0** statement is used to disable error handling.

Code:
```
<% On Error GoTo 0 %>
```

If you wish to know whether an error has occurred and of what type, you can insert the following code.

Code:
```
<% If Err.Number <> 0 Then %>
<% =Err.Number%>
<% Err.Clear %>
```

OPERATOR: Or

Implemented in version 1.0

Or

The **Or** operator is used to perform a logical disjunction on two expressions, where the expressions are **Null**, or are of **Boolean** subtype and have a value of **True** or **False**.

The **Or** operator can also be used a "bitwise operator" to make a bit-by-bit comparison of two integers. If one or both bits in the comparison are 1, then a 1 is returned. Otherwise, a 0 is returned.

When using the **Or** to compare **Boolean** expressions, the order of the expressions is important.

Code:
```
<% =True Or True %>
<% =True Or False %>
<% =False Or True %>
<% =False Or False %>

<% =True Or Null %>
<% =Null Or True %>
<% =False Or Null %>
<% =Null Or False %>
<% =Null Or Null %>
```

Output:
```
True
True
True
False

True
True
(Null output)
(Null output)
(Null output)
```

In this example, the **Or** performs a bitwise comparison on the 1 (in binary 001) and the 2 (in binary 010), and returns a 3 (in binary 011).

Code:
```
<%
Expression1 = 1
Expression2 = 2
Result = Expression1 Or Expression2
Response.Write "Result = " & Result
%>
```

Output:
```
Result = 3
```

Property: Pattern

object.Pattern

The **Pattern** property is used with a **RegExp** object variable to declare a string search pattern (also known as a regular expression).

Code:

```
<%
MyRegExpObject.Pattern = "red jaguar"
%>
```

The search pattern can also be a regular expression. Here is a brief introductory list of allowable special characters.

Character	Usage
*	Matches the previous character zero or more times
+	Matches the previous character one or more times
?	Matches the previous character zero or one times
.	Matches any single character except the newline
^	Matches the start of the input
$	Matches the end of the input
x|y	Matches either first or second character listed
(pattern)	Matches pattern
{number}	Matches exactly number times
{number,}	Matches number, or more, times (note comma)
{num1, num2}	Matches at least num1 and at most num2 times
[abc]	Matches any character listed between the [ ]
[^abc]	Matches all characters except those listed between the [ ]
[a-e]	Matches any characters in the specified range (a,b,c,d,e)
[^K-Q]	Matches all characters except in the specified range
\	Signifies that the next character is special or a literal.
\b	Matches only on a word boundary
\B	Matches only inside a word
\d	Matches only on a digit
\D	Matches only on a non-digit
\f	Matches only on a form feed character
\n	Matches only on a new line
\r	Matches only on a carriage return
\s	Matches only on a blank space
\S	Matches only on nonblank spaces
\t	Matches only on a tab
\v	Matches only on a vertical tab
\w	Matches only on A to Z, a to z, 1 to 9, and _
\W	Matches characters other than A to Z, a to z, 1 to 9, and _

\number	Matches any positive number
\octal	Matches any octal number
\xhex	Matches any hexadecimal number (x is required)

STATEMENT: Randomize

Randomize

The **Randomize** statement gives the **Rnd** function a new seed value for generating random numbers.

Code:

```
<% Randomize %>
<% =Rnd() %>
```

Output:

```
0.7953455
```

METHOD: TextStream.Read

Implemented in version 2.0

object.Read

This method reads the number of characters you specify from a **Textstream** file and returns them as a string. If you specify more characters than actual exist in the file, then **Read** only returns the actual number of characters that are in the file.

Code:

```
<%
dim filesys, text, readfile, contents
set filesys = CreateObject("Scripting.FileSystemObject")
Set text = filesys.CreateTextFile("c:\somefile2.txt")
text.Write "A quick example of the Read method"
text.close
set readfile = filesys.OpenTextFile("c:\somefile2.txt", 1, false)
contents = readfile.Read(7)
readfile.close
Response.Write "The first seven characters in the text file are '" & contents & "'."
%>
```

Output:

"The first seven characters in the text file are 'A quick'.

METHOD: TextStream.ReadAll

Implemented in version 2.0

object.ReadAll

This method reads the entire contents of a text file and returns it as a string.

Code:

```
<%
dim filesys, text, readfile, contents
set filesys = CreateObject("Scripting.FileSystemObject")
Set text = filesys.CreateTextFile("c:\somefile2.txt")
text.Write "A quick example of the ReadAll method"
text.close
set readfile = filesys.OpenTextFile("c:\somefile2.txt", 1, false)
contents = readfile.ReadAll
readfile.close
Response.Write "The file contains the following text - '" & contents & "'."
%>
```

Output:

"The file contains the following text - 'A quick example of the ReadAll method'.

METHOD: TextStream.ReadLine

Implemented in version 2.0

object.ReadLine

Reads a single line (excluding the newline character) from a **TextStream** file and returns the contents as a string.

Code:

```
<%
dim filesys, text, readfile, contents
set filesys = CreateObject("Scripting.FileSystemObject")
Set text = filesys.CreateTextFile("c:\somefile2.txt")
text.Write "A quick example of the ReadLine method"
text.close
set readfile = filesys.OpenTextFile("c:\somefile2.txt", 1, false)
contents = readfile.ReadLine
readfile.close
Response.Write "The line contains the following text - '" & contents & "'."
%>
```

Output:

"The line contains the following text - 'A quick example of the ReadLine method'.

OBJECT: RegExp

Implemented in version 5.0

The **RegExp** object is used to look for and match all occurrences of a search string pattern inside a target string.

Each time the **RegExp** object finds a match during the search, a **Match** object is created and added to a **Matches** collection.

The search pattern is declared using the **Pattern** property. It could, for example, be a simple string, such as "cost analysis". However, the search pattern can also be a regular expression. Regular expressions can range from being very simple to being extremely complex. The **Pattern** property page contains an introductory listing of special characters that can be used with regular expressions.

The following code is a simpler, working version of a program published by Microsoft. Note the use of the keyword **New** when you create a **RegExp** object using **Set**.

Code:

```
<%
'this sub finds the matches
Sub RegExpTest(strMatchPattern, strPhrase)
   'create variables
   Dim objRegEx, Match, Matches, StrReturnStr
   'create instance of RegExp object
   Set objRegEx = New RegExp

   'find all matches
   objRegEx.Global = True
   'set case insensitive
   objRegEx.IgnoreCase = True
   'set the pattern
   objRegEx.Pattern = strMatchPattern

   'create the collection of matches
   Set Matches = objRegEx.Execute(strPhrase)

   'print out all matches
   For Each Match in Matches
      strReturnStr = "Match found at position "
      strReturnStr = strReturnStr & Match.FirstIndex & ". Match Value is '"
      strReturnStr = strReturnStr & Match.value & "'."
      'print
      Response.Write(strReturnStr & "<BR>")
   Next
End Sub

'call the subroutine
RegExpTest "is.", "Is1 is2 Is3 is4"
%>
```

Output:

```
Match found at position 0. Match Value is 'Is1'.
```

Match found at position 4. Match Value is 'is2'.
Match found at position 8. Match Value is 'Is3'.
Match found at position 12. Match Value is 'is4'.

PROPERTIES

Global Property

This property is only used with a **RegExp** object variable and returns a **Boolean** value. **False** signifies that a search should only find the first occurrence of a match. **True** signifies that a search should find all occurrences of a match.

Syntax: **object.Global = [True | False]**

IgnoreCase Property

This property is only used with a **RegExp** object variable and returns a **Boolean** value. **False** signifies that a search should be case-sensitive (i.e., upper versus lower case). **True** signifies that a search should ignore case in a match.

Syntax: **object.IgnoreCase = [True | False]**

Pattern Property

This property defines the regular expression or search pattern string that is to be matched during the search.

Syntax: **object.Property = [SearchPatternString]**

METHODS

Execute Method

This method is used to execute the search and to look for matches of the search pattern string (or regular expression) and the target string. Each time a match is made, a **Match** object is created and added to a collection that is called a **Matches** collection.

Syntax: **object.Execute**

Replace Method

This method is used to replace text found in a regular expression search. Do not confuse this method with the **Replace** function.

Syntax: **object.Replace (String1, String2)**

Test Method

This method is used to determine if the search pattern occurs within a specified string and returns a Boolean value to signify the results of the search. **True** is returned if the pattern is found. Otherwise, **False** is returned.

Syntax: **object.Test**

METHOD: object.Replace

Implemented in version 5.0

object.Replace (String1, String2)

The **Replace** method is used to replace text in a regular expression search. It can only be used with a **RegExp** object variable.

Do not confuse this method with the **Replace** function.

The search string pattern is declared using the **Pattern** property. You can use the **Global** property to limit the search to the first occurrence of a match, or all occurrences.

There are two mandatory arguments. If the search string pattern is found in one or more occurrences inside the string designated by the **String1** argument, then, as set by the **Global** property, the first or all occurrences of the search string pattern will be replaced with the **String2** argument.

Code:

```
<%
Dim RegX
Set RegX = NEW RegExp
Dim MyString, SearchPattern, ReplacedText
MyString = "Ocelots make good pets."
SearchPattern = "good"
ReplaceString = "bad"
RegX.Pattern = SearchPattern
RegX.Global = True
ReplacedText = RegX.Replace(MyString, ReplaceString)
Response.Write(ReplacedText)
%>
```

Output:

"Ocelots make bad pets."

STATEMENT: Rem

Rem

The **Rem** statement allow you to insert comments into your code.

Code:

```
<% Rem This is one of two ways to comment in VBScript %>
<% ' This is the second way %>
```

FUNCTION: Replace( )

Replace(String, FindSubstring, ReplaceSubstring, Start, Count, Compare)

The **Replace** function replaces a specified substring within a specified string with a new specified substring and returns the modified string.

There are three mandatory arguments.

String

The **String** argument is the string to be searched.

FindSubstring

The **FindSubstring** argument is the substring you are searching for inside the string.

ReplaceSubstring

The **ReplaceSubstring** argument is the the new substring string that you wish to insert inside the string.

Code:
```
<% =Replace("How now brown cow?", "brown", "purple") %>
```

Output:
```
How now purple cow?
```

There are three optional arguments.

Start

The optional **Start** argument specifies the position number, counting from the left, where you wish to start the search.

Note that the portion of the string to the left of the start position will be discarded.

Code:
```
<% =Replace("How now brown cow?", "brown", "purple", 7) %>
```

Output:
```
w purple cow?
```

Count

The optional **Count argument** specifies how many times to replace the substring.

Code:
```
<% =Replace("red blue red-blue redblue bluered", "blue", "purple", 1, 3) %>
```

Output:
```
red purple red-purple redpurple bluered
```

Compare

The optional **Compare** argument must only use either the constant or value of the COMPARISON CONSTANTS.

CONSTANT	VALUE	DESCRIPTION
VBBinaryCompare	0	Binary comparison
VBTextCompare	1	Text Comparison
VBDataBaseCompare	2	Compare information inside database

In the example, by using VBBinaryCompare, or 0, for the **Compare** argument, all upper/lower case differences are obeyed in both the search and replace.

Code:
```
<% =Replace("red blue red-blue redblue bluered", "BLUE", "PURPLE", 1, 3, 0) %>
```

Output:
red blue red-blue redblue bluered

In the example, by using VBTextCompare, or 1, for the **Compare** argument, all upper/lower case differences are ignored in the search and obeyed in the replace.

Code:
```
<% =Replace("red blue red-blue redblue bluered", "BLUE", "PURPLE", 1, 3, 1) %>
```

Output:
red PURPLE red-PURPLE redPURPLE bluered

FUNCTION: RGB( )

Implemented in version 2.0

RGB(red, green, blue)

The **RGB** function returns an integer that defines an RGB color value.

RGB is an old television term. It refers to the primary colors that are used in color television (red, green, and blue). These three colors are blended together inside a television tube to create the colors displayed on the screen. The same concept is applied to computer monitors.

There are three mandatory arguments.

The **red** mandatory agrument defines the red component of the color and must be an integer number ranging from 0 to 255.

The **green** mandatory agrument defines the green component of the color and must be an integer number ranging from 0 to 255.

The **blue** mandatory agrument defines the blue component of the color and must be an integer number ranging from 0 to 255.

If the integer argument exceeds 255, it will be treated as 255. Note, that there are 256*256*256 = 16,777,215 different color permutations.

Code:

```
<%; = RGB(0, 0, 0) %>
<%; = RGB(94, 71, 177) %>
<%; = RGB(255, 255, 255) %>
```

Output:

```
0
11618142
16777215
```

FUNCTION: Right( )

Right(String, Length)

The **Right** function returns the right portion of the designated string for a designated length starting from the right side.

There are two mandatory arguments.

String

The **String** argument is the name of the string you wish to truncate.

Length

The **Length** argument is the number of characters (including blanks) you wish to save counting from the right side towards the left side of the string.

Code:

```
<% =Right("abcde fghij klmno pqrst uvwxyz", 13) %>
```

Output:

```
pqrst uvwxyz
```

FUNCTION: Rnd( )

Rnd(Number)

The **Rnd** function generates a pseudo-random number greater than or equal to 0.0 and less than 1.0.

The **Number** argument is optional, but different numbers will give different pseudo-random numbers.

Note that the number generated is pseudo-random, because the same output tends to be repeated over and over. You can use the **Randomize** statement to over come this problem.

Code:
```
<% =Rnd() %>
```

Output:
0.7055475

Code:
```
<% =Rnd(127.89) %>
```

Output:
0.533424

Code:
```
<% =Rnd(-127.89) %>
```

Output:
0.6953956

Code:
```
<% Randomize %>
<% =Rnd() %>
```

Output:
0.1414095

The following code will generate a random number between any limits you choose. (Since the **Int** function always rounds down, we add one to the difference between the limits.)

Code:
```
<% upperlimit = 50000.0 %>
<% lowerlimit = -30000.0 %>
<% =Int((upperlimit - lowerlimit + 1)*Rnd() + lowerlimit) %>
```

Output:
30366

FUNCTION: Round( )

Round(Number, NumDecimalPlaces)

The **Round** function rounds off a floating-point number (decimal) to a specified number of decimal places.

There is one mandatory argument.

Number

The **Number** argument is the number you wish to round off.

If the number of decimal places to round off to is not specified, the number is rounded off to an integer.

Code:
```
<% =Round(1.123456789) %>
```

Output:
1

Code:
```
<% =Round(9.87654321) %>
```

Output:
10

Note that negative numbers are rounded down (more negative).

Code:
```
<% =Round(-2.899999999) %>
```

Output:
-3

Even numbers that are composed of the exact decimal .5, such as 2.5, 4.50, or 22.500000, are rounded down (towards the negative direction).

Negative, even numbers are rounded up (towards the postive direction).

Odd number that are composed of the exact decimal .5, such as 1.5, 3.50, or 21.500000, are rounded up (towards the positive direction).

Negative, odd numbers are rounded down (towards the negative direction).

Code:
```
<% =Round(4.50) %>
<% =Round(3.5) %>
<% =Round(2.5000) %>
<% =Round(1.5) %>
<% =Round(0.50000) %>
<% =Round(-0.50) %>
<% =Round(-1.5) %>
```

```
<% =Round(-2.50000000) %>
<% =Round(-3.5) %>
<% =Round(-4.5) %>
```

Output:
```
4
4
2
2
0
0
-2
-2
-4
-4
```

There is one optional argument.

NumDecimalPlaces

The optional **NumDecimalPlaces** argument specifies how many decimal places to round off to.

Code:
```
<% =Round(1.123456789, 6) %>
```

Output:
```
1.123457
```

Note that negative numbers are rounded down (more negative).

Code:
```
<% =Round(-2.899999999, 2) %>
```

Output:
```
-2.9
```

FUNCTION: RTrim( )

RTrim(String)

The **RTrim** function removes extra blank spaces only on the right side of a string.

Note that HTML automatically removes extra blank spaces in this view.

Code:
```
<% =RTrim("   A sentence with extra spaces on both sides.    ") %>
```

Output:
A sentence with extra spaces on both sides.

FUNCTION: ScriptEngine

ScriptEngine

The **ScriptEngine** function returns the name of the scripting language being used.

Code:

```
<% =ScriptEngine %>
```

Output:

```
VBScript
```

FUNCTION: ScriptEngineBuildVersion

ScriptEngineBuildVersion

The **ScriptEngineBuildVersion** function returns the name of the script engine being used.

Code:

```
<% =ScriptEngineBuildVersion %>
```

Output:

```
2926
```

FUNCTION: ScriptEngineMajorVersion

ScriptEngineMajorVersion

The **ScriptEngineMajorVersion** function returns the major version number of the script engine being used.

Code:
```
<% =ScriptEngineMajorVersion %>
```

Output:
4

FUNCTION: ScriptEngineMinorVersion

Implemented in version 5.0

ScriptEngineMinorVersion

The **ScriptEngineMinorVersion** function returns the minor version number of the script engine being used.

Code:
```
<% =ScriptEngineMinorVersion %>
```

Output:
```
0
```

FUNCTION: Second( )

Second(Time)

The **Second** function returns the second using the **Time** function as an argument.

Values will run from 0 to 59.

Code:

```
<% =Second(Time) %>
```

Output:

```
56
```

STATEMENT: Select Case

Select Case

The **Select Case** conditional statement selectively executes different groups of code by comparing a variable to a **Case** (a series of conditions). If one of the cases (conditions) is satisfied, then the code associated with that case is executed.

You may specify multiple, comma-delimited conditions for each **Case** as in the first example below. If there not is a match, then the **Case** is skipped over and the next **Case** is considered, and so on for each **Case**.

Note that it is quite possible that none of the cases will be satisfied. Therefore, you may wish to use the optional **Case Else** statement. It will only execute if none of the other conditions match.

You must end the **Select Case** statement with **End Select** or you will get an error message.

Code:

```
<%
Select Case finalnumber
Case 1, 5
  result = "The result is 1 or 5"
Case 2
  result = "The result is 2"
Case 3
  result = "The result is 3"
End Select
%>
```

Code:

```
<%
Select Case firstname
Case "Brenn"
  Welcome = "Hello Brenn"
Case "Fred"
  Welcome = "Hello Fred"
Case "Susan"
  Welcome = "Hello Susan"
Case Else
  Welcome = "Hello world"
End Select
%>
```

STATEMENT: Set

Set

The **Set** statement assigns the object reference to a variable or property. (i.e., create an instance of a specific object)

The keyword **New** is used in conjunction with **Set** to create an instance of a **Class** or **RegExp**.

The keyword **Nothing** is used to unassign the object reference from the variable or property.

Code:

```
<%
Dim Anyvariable
Set Anyvariable = Server.CreateObject("Scripting.FileSystemObject")
%>

<%
Dim SearchPattern
Set SearchPattern = New RegExp
%>

<%
Set Anyvariable = Nothing
Set SearchPattern = Nothing
%>
```

FUNCTION: SetLocale()

SetLocale(LCID)

The **SetLocale** function sets the LCID (locale ID).

There is one mandatory argument, the LCID, which is a short string, decimal value, or hex value that uniquely identifies a geographic locale. The various geographic locales are based upon the user's language, country, and culture. For example, the locale "English - United States" can be designated as either "en-us", or "1033", or "0x0409".

This locale information is used to establish user preferences and formats for such things as alphabets, currency, dates, keyboard layout, and numbers. A list of these locales, and their return values, can be found here.

If the LCID is set to zero, the locale will be set by the system.

Code:

```
<% =SetLocale(0) %>
```

Output:

```
1033
```

FUNCTION: Sgn()

Sgn(Number)

The **Sgn** function returns the sign of a number.

-1 indicates a negative number.
 0 indicates the number zero.
 1 indicates a positive number.

Code:
```
<% =Sgn(-127.89) %>
```

Output:
-1

Code:
```
<% =Sgn(0) %>
```

Output:
0

Code:
```
<% =Sgn(127.89) %>
```

Output:
1

FUNCTION: Sin( )

Sin(Number)

The **Sin** function returns the sine for a number (angle).

Code:
```
<% =Sin(45.0) %>
```

Output:
0.850903524534118

You can also use a negative number (angle).

Code:
```
<% =Sin(-45.0) %>
```

Output:
-0.850903524534118

FUNCTION: Space( )

Space(Number)

The **Space** function creates a string with the specified number of blank spaces.

Note that HTML will only show one blank space on the same line.

Code:

```
<% ="Line above space." %>
<% =Space(13) %>
<% ="Line below space." %>
```

Output:

Line above space.

Line below space.

FUNCTION: Split( )

Split(Expression, Delimiter, Count, Compare)

The **Split** function separates a string into substrings and creates a one-dimensional array where each substring is an element.

There is one mandatory argument.

Expression

The **Expression** argument is a string expression.

Code:
```
<% mystring = "How now brown cow?" %>
<% myarray = Split(mystring) %>
<% =myarray(0) %>
<% =myarray(1) %>
<% =myarray(2) %>
<% =myarray(3) %>
```

Output:
```
How
now
brown
cow?
```

There are three optional arguments.

Delimiter

The optional **Delimiter** argument specifies the characters (including blanks) you wish to use to separate the expression. The default is a single empty space, " ".

Code:
```
<% mystring = "How now brown cow?" %>
<% myarray = Split(mystring, "ow") %>
<% =myarray(0) %>
<% =myarray(1) %>
<% =myarray(2) %>
<% =myarray(3) %>
<% =myarray(4) %>
```

Output:
```
H
n
br
n c
?
```

Count

The optional **Count** argument specifies the number of elements to return.

Code:

```
<% mystring = "How now brown cow?" %>
<% myarray = Split(mystring, " ", 2) %>
<% =myarray(0) %>
<% =myarray(1) %>
```

Output:

```
How
now brown cow?
```

Compare

The optional **Compare** argument must only use the constant or value of the COMPARISON CONSTANTS.

CONSTANT	VALUE	DESCRIPTION
VBBinaryCompare	0	Binary comparison
VBTextCompare	1	Text Comparison
VBDataBaseCompare	2	Compare information inside database

In the example, by using VBTextCompare for the **Compare** argument, the split occurs at the b and ignores the upper case difference between the **Delimiter** argument "B".

Code:

```
<% mystring = "How now brown cow?" %>
<% myarray = Split(mystring, "B", 2, 1) %>
<% =myarray(0) %>
<% =myarray(1) %>
```

Output:

```
How now
rown cow?
```

Code:

```
<% mystring = "How now brown cow?" %>
<% myarray = Split(mystring, "B", 2, VBTextCompare) %>
<% =myarray(0) %>
<% =myarray(1) %>
```

Output:

```
How now
rown cow?
```

FUNCTION: Sqr( )

Sqr(Number)

The **Sqr** function returns the square root of a positive number.

Code:
```
<% =Sqr(4) %>
```

Output:
2

Code:
```
<% =Sqr(123.456789) %>
```

Output:
11.1111110605556

FUNCTION: StrComp( )

StrComp(String1, String2, Compare)

The **StrComp** function compares two strings to see if they are the same. You can have the comparison be sensitive, or not, to the case (upper versus lower) of the characters in the two strings. The default is to be case sensitive (binary comparison).

If the strings are the same, the output is zero. If the strings are different, the output will be a 1 or -1 depending on the order of the strings.

There are two mandatory arguments.

String1

The **String1** argument is the first of two strings to compare.

String2

The **String2** argument is the second of two strings to compare.

Code:
```
<% =StrComp("Mountains of the Moon", "Mountains of the Moon") %>
```

Output:
0

Code:
```
<% =StrComp("Mountains of the Moon", "Red sails at sunset") %>
```

Output:
-1

There is one optional argument.

Compare

The optional **Compare** argument can be used to set whether the comparison is case sensitive (upper versus lower), or not. You must only use the constant or the value of the COMPARISON CONSTANTS.

CONSTANT	VALUE	DESCRIPTION
VBBinaryCompare	0	Binary comparison (case sensitive)
VBTextCompare	1	Text Comparison (case insensitive)

Code:
```
<% =StrComp("Mountains of the Moon", "mountains of the moon", 0) %>
```

Output:
-1

Code:
```
<% =StrComp("Mountains of the Moon", "mountains of the moon", 1) %>
```

Output:
0

Code:
```
<% =StrComp("RED", "red", VBTextCompare) %>
```

Output:
0

CONSTANTS: String

String Constants

You may use either the CONSTANT (left column) or the VALUE (center column) in your code and get the same results.

CONSTANT	VALUE	DESCRIPTION
VBCR	Chr(13)	Carriage return
VBCrLf	Chr(13) & Chr(10)	Combined carriage return and line feed
VBFormFeed	Chr(12)	Form feed
VBLF	Chr(10)	Line feed
VBNewLine	Chr(13) & Chr(10) Or Chr(10)	Newline character appropriate for platform
VBNullChar	Chr(0)	Character value of zero
VBNullString	String Of Value Zero	Null string
VBTab	Chr(9)	Horizontal (row) tab
VBVerticalTab	Chr(11)	Vertical (column) tab

FUNCTION: String( )

String(Number, Character)

The **String** function creates a string with a single character repeated the specified number of times.

Both arguments are mandatory.

Number

The **Number** argument must an integer and defines the number of times to repeat the character.

Character

The **Character** argument is a single character.

It must be inside a pair of double quotes.

Code:

```
<% =String(33, "!") %>
```

Output:

```
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

FUNCTION: StrReverse( )

StrReverse(String)

The **StrReverse** function reverses the characters (including blanks) in a string.

Code:

```
<% =StrReverse("abcde fghij klmno pqrst uvwxyz") %>
```

Output:

```
zyxwvu tsrqp onmlk jihgf edcba
```

STATEMENT: Sub

Sub

The **Sub** statement creates a subroutine, assigns a name, and allows you to list arguments which are to be passed into the subroutine.

The major difference between a subroutine and a function, is that a subroutine cannot return any value back to where it was called from.

As an option, subroutines can be declared **Private** or **Public**. You can call other subroutines and functions from within a subroutine (nesting). You must end every subroutine with **End Sub** or you will get an error message. One option for leaving a subroutine is to use **Exit** statements.

Code:
```
<% Sub aardvark %>
  <% Rem you can place all of the code you desire inside a subroutine %>
<% End Sub %>
```

Code:
```
<% Private Sub aardvark %>
  <% Rem you can place all of the code you desire inside a subroutine %>
<% End Sub %>
```

You can list more than one argument in the subroutine.

Code:
```
<%
Sub aardvark(myvar1, myvar2, mynumber, myarray)
  Rem you can place all of the code you desire inside a subroutine
End Sub
%>
```

FUNCTION: Tan( )

Tan(Number)

The **Tan** function returns the tangent for a number (angle).

Code:
```
<% =Tan(45.0) %>
```

Output:
1.61977519054386

You can also use a negative number (angle).

Code:
```
<% =Tan(-45.0) %>
```

Output:
-1.61977519054386

EVENT: Terminate

Implemented in version 5.0

Terminate

The **Terminate** event is an optional subroutine that can be called when you terminate an instance of an object using the **Class ... End Class** statement. You can place any valid VBScript code inside the subroutine that you want to run when the class is terminated.

The similar **Initialize** event can be called when you create an instance of an object.

Note, as shown in the code example, that the subroutine name must be composed of the word Class, an underscore (_) and the word Terminate().

Code:

```
Class DevGuru
...
  Private Sub Class_Terminate()
    'place any valid VBScript code here
  End Sub
...
End Class
```

OBJECT: TextStream

Implemented in version 2.0

The **TextStream** object provides sequential access to the contents of any file where the contents are in text-readable form. You can create an instance of the **TextStream** object using the **CreateTextFile** or **OpenTextFile** methods of the **FileSystemObject** object, or alternatively by using the **OpenAsTextStream** method of the **File** object.

In the following code the **CreateTextFile** method is used on the **FileSystemObject** to return the **TextStream** object, "testfile". This **TextStream** object is then manipulated using the **WriteLine** and **Close** methods.

Code:

```
<%
Dim filesys, testfile
Set filesys = CreateObject("Scripting.FileSystemObject")
Set testfile= filesys.CreateTextFile("c:\somefile.txt", True)
testfile.WriteLine "Your text goes here."
testfile.Close
%>
```

Every instance of a **TextStream** object has an internal pointer that points to a specific place in the file. When first opening the stream, this pointer is positioned at the first character in the file. The pointer will then move around as you manipulate the **TextStream** object using its methods and properties.

PROPERTIES

AtEndOfLine Property

Returns a Boolean value. If the file pointer is positioned immediately before the file's end-of-line marker, the value is **True**, and **False** otherwise.

Syntax: **object.AtEndOfLine**

AtEndOfStream Property

Returns a Boolean value. If the file pointer is positioned at the end of a file, the value is **True**, and **False** otherwise.

Syntax: **object.AtEndOfStream**

Column Property

Returns the current position of the file pointer within the current line. Column 1 is the first character in each line.

Syntax: **object.Column**

Line Property

This property returns the current line number in a text file.

Syntax: **object.Line**

METHODS

Close Method
Closes a currently open **TextStream** file.

Syntax: **object.Close**

Read Method
This method reads the number of characters you specify from a **Textstream** file and returns them as a string.

Syntax: **object.Read(characters)**

ReadAll Method
This method reads the entire contents of a text file and returns it as a string.

Syntax: **object.ReadAll**

ReadLine Method
Reads a single line (excluding the newline character) from a **TextStream** file and returns the contents as a string.

Syntax: **object.ReadLine**

Skip Method
Causes the file pointer to skip a specified number of characters when reading a **TextStream** file. Can be a positive or negative number.

Syntax: **object.Skip(characters)**

SkipLine Method
Moves the file pointer from its current position to the beginning of the next line.

Syntax: **object.SkipLine**

Write Method
This method writes a supplied string to an open **TextStream** file.

Syntax: **object.Write(string)**

WriteLine Method
Writes a supplied string to a **TextStream** file, followed by a new line character.

Syntax: **object.WriteLine([string])**

WriteBlankLines Method
Used to write a number of consecutive newline characters (defined with the **lines** parameter) to a **TextStream** file.

Syntax: **object.WriteBlankLines(lines)**

METHOD: TextStream.Close

Implemented in version 2.0

object.Close

This method is used to close a currently open **TextStream** file.

Code:

```
<%
Dim filesys, testfile
Set filesys = CreateObject("Scripting.FileSystemObject")
Set testfile= filesys.CreateTextFile("c:\somefile.txt", True)
testfile.WriteLine("Your text goes here.")
testfile.Close
%>
```

METHOD: TextStream.Skip

Implemented in version 2.0

object.Skip(numberofcharacters)

This method causes the file pointer to skip forward the integer number of characters specified in the argument when reading a **TextStream** file. The skipped characters are simply ignored.

Code:

```
<%
dim filesys, text, readfile, contents
set filesys = CreateObject("Scripting.FileSystemObject")
Set text = filesys.CreateTextFile("c:\somefile2.txt")
text.WriteLine "A quick example of the Skip method"
text.close
set readfile = filesys.OpenTextFile("c:\somefile2.txt", 1, false)
readfile.Skip(10)
contents = readfile.ReadLine
readfile.close
Response.Write "From column 11 the line contains the following text - '" &_
                        contents & "'."
%>
```

Output:

"From column 11 the line contains the following text - 'ample of the Skip method'."

METHOD: TextStream.SkipLine

Implemented in version 2.0

object.SkipLine

Moves the file pointer from its current position to the beginning of the next line.

Code:

```
<%
dim filesys, text, readfile, contents
set filesys = CreateObject("Scripting.FileSystemObject")
Set text = filesys.CreateTextFile("c:\somefile2.txt")
text.WriteLine "A quick example of the SkipLine method"
text.WriteLine "Well, maybe not so quick!"
text.WriteLine "This should be enough!"
text.close
set readfile = filesys.OpenTextFile("c:\somefile2.txt", 1, false)
readfile.SkipLine
contents = readfile.ReadLine
readfile.close
Response.Write "Line 2 contains the following text - '" & contents & "'."
%>
```

Output:

"Line 2 contains the following text - 'Well, maybe not so quick!'."

METHOD: TextStream.Write

Implemented in version 2.0

object.Write

This method writes a supplied string to the begining of a **TextStream** file if opened for writing (provided it is the first write since opening the file); if the file is open for appending, the string will be added to the end of the file.

Code:

```
<%
Dim filesys, testfile
Set filesys = CreateObject("Scripting.FileSystemObject")
Set testfile= filesys.CreateTextFile("c:\somefile.txt", True)
testfile.Write "Your text goes here."
testfile.Close
%>
```

METHOD: TextStream.WriteLine

Implemented in version 2.0

object.WriteLine

This method writes a supplied string to a **TextStream** file, followed by a new line character.

Code:

```
<%
Dim filesys, testfile
Set filesys = CreateObject("Scripting.FileSystemObject")
Set testfile= filesys.CreateTextFile("c:\somefile.txt", True)
testfile.WriteLine "Your text goes here."
testfile.Close
%>
```

METHOD: TextStream.WriteBlankLines

Implemented in version 2.0

object.WriteBlankLines

Used to write a number of consecutive newline characters (defined with the **lines** parameter) to a **TextStream** file.

Code:

```
<%
Dim filesys, testfile
Set filesys = CreateObject("Scripting.FileSystemObject")
Set testfile= filesys.CreateTextFile("c:\somefile.txt", True)
testfile.WriteBlankLines(3)
testfile.Close
%>
```

FUNCTION: Time

Time

The **Time** function returns the current time as defined by your computer.

Code:

```
<% =Time %>
```

Output:

2:07:56 PM

FUNCTION: Timer

Implemented in version 5.0

Timer

The **Timer** function returns the number of seconds that have passed since midnight (12:00:00 AM).

Timer keeps track of the seconds to at least seven decimal places of accuracy. So, you can use **Timer** as a "stop watch" to find the start and finish times of an operation. Then simply display the difference to find the elapsed time. However, for very short elapsed time periods, the difference may displayed as zero.

Code:

```
<% =Now %>
<% =Timer %>
```

Output:

```
10/27/99 2:14:36 PM
51276.99
```

FUNCTION: TimeSerial( )

TimeSerial(Hour, Minute, Second)

The **TimeSerial** function converts the arguments into the variant of subtype Date.

There are three mandatory arguments.

Hour

The **Hour** argument is the hour as a string or integer.

Minute

The **Minute** argument is the minute as a string or integer.

Second

The **Second** argument is the second as a string or integer.

Code:
```
<% =TimeSerial(12, 11, 10) %>
```

Output:
12:11:10 PM

Code:
```
<% =TimeSerial("12", "11", "10") %>
```

Output:
12:11:10 PM

FUNCTION: TimeValue( )

TimeValue(String)

The **TimeValue** function converts an argument into the variant of subtype Date.

The **String** argument must be a string in valid date format or you will get an error message.

Code:
```
<% mystring = "2:23:17 PM" %>
<% =TimeValue(mystring) %>
```

Output:
2:23:17 PM

Code:
```
<% =TimeValue("14:23") %>
```

Output:
2:23:00 PM

FUNCTION: Trim( )

Trim(String)

The **Trim** function removes extra blank spaces on both the right and left side of a string.

Note that HTML automatically removes extra blank spaces in this view.

In this example, note that the extra white spaces are removed from both sides.
Code:

```
<% =Trim("   How now brown cow?     ") %>
```

Output:
How now brown cow?

CONSTANTS: Tristate

Tristate Constants

You may use either the CONSTANT (left column) or the VALUE (center column) in your code and get the same results.

CONSTANT	VALUE	DESCRIPTION
TristateTrue	-1	True
TristateFalse	0	False
TristateUseDefault	-2	Use default setting

FUNCTION: TypeName( )

TypeName(Varname)

The **TypeName** function is used to determine the subtype of a named variable.

There is a companion function **VarType** that returns the subtype number in place of the subtype name.

The output is the name of the subtype.

Code:
```
<% whatsubtype="This is a string subtype" %>
<% =TypeName(whatsubtype) %>
```

Output:
String

Code:
```
<% whatsubtype=12345 %>
<% =TypeName(whatsubtype) %>
```

Output:
Integer

FUNCTION: UBound( )

UBound(ArrayName, Dimension)

The **UBound** function returns the upper limit for the number of elements in an array (the elements can be empty).

There is one mandatory argument.

ArrayName

The **ArrayName** argument is the name of the array. Do not include the parenthesis with the array name.

Code:
```
<% Dim katarray(5) %>
<% katarray(0) = "Mountain lion" %>
<% katarray(1) = "Bobcat" %>
<% katarray(2) = "Jaguar" %>
<% =UBound(katarray) %>
```

Output:
5

There is one optional argument.

Dimension

The optional **Dimension** argument is used to identify which index you are determining the upper bounds for in a multi-dimensional array.

Code:
```
<% Dim arrayfish(4,6) %>
<% arrayfish(0,0) = "Neon tetra" %>
<% arrayfish(0,1) = "Angle fish" %>
<% arrayfish(0,2) = "Discus" %>
<% arrayfish(1,0) = "Golden dojo" %>
<% arrayfish(1,1) = "Clown loach" %>
<% arrayfish(1,2) = "Betta" %>
<% =UBound(arrayfish, 2) %>
```

Output:
6

FUNCTION: UCase( )

UCase(String)

The **UCase** function converts a string to all upper case letters.

There is also a companion function **LCase** to convert to lower case letters.

Code:
```
<% =UCase("Aardvarks do not make good pets!") %>
```

Output:
AARDVARKS DO NOT MAKE GOOD PETS!

CONSTANTS: VarType

VarType Constants

You may use either the CONSTANT (left column) or the VALUE (center column) in your code and get the same results.

CONSTANT	VALUE	DESCRIPTION
VBEmpty	0	Uninitialized
VBNull	1	Contains no valid data
VBInteger	2	Integer subtype
VBLong	3	Long subtype
VBSingle	4	Single subtype
VBDouble	5	Double subtype
VBCurrency	6	Currency subtype
VBDate	7	Date subtype
VBString	8	String subtype
VBObject	9	Object
VBError	10	Error subtype
VBBoolean	11	Boolean subtype
VBVariant	12	Variant (only use for arrays of variants)
VBDataObject	13	Data access object
VBDecimal	14	Decimal subtype
VBByte	17	Byte subtype
VBArray	8192	Array

FUNCTION: VarType( )

VarType(Varname)

The **VarType** function is used to determine the subtype of a named variable.

The output of this function is a value which is a VarType CONSTANT.

CONSTANT	VALUE	DESCRIPTION
VBEmpty	0	Uninitialized
VBNull	1	Contains no valid data
VBInteger	2	Integer subtype
VBLong	3	Long subtype
VBSingle	4	Single subtype
VBDouble	5	Double subtype
VBCurrency	6	Currency subtype
VBDate	7	Date subtype
VBString	8	String subtype
VBObject	9	Object
VBError	10	Error subtype
VBBoolean	11	Boolean subtype
VBVariant	12	Variant (only use for arrays of variants)
VBDataObject	13	Data access object
VBDecimal	14	Decimal subtype
VBByte	17	Byte subtype
VBArray	8192	Array

Code:

```
<% whatsubtype="This is a string subtype" %>
<% =VarType(whatsubtype) %>
```

Output:

```
8
```

Code:

```
<% whatsubtype=12345 %>
<% =VarType(whatsubtype) %>
```

Output:

```
2
```

FUNCTION: Weekday( )

Weekday(Date, FirstDayofWeek)

The **Weekday** function returns the number of the day of the week.

The output is defined by the value in the **Date And Time Constants**.

CONSTANT	VALUE	DESCRIPTION
VBSunday	1	Sunday
VBMonday	2	Monday
VBTuesday	3	Tuesday
VBWednesday	4	Wednesday
VBThursday	5	Thursday
VBFriday	6	Friday
VBSaturday	7	Saturday
VBFirstJan1	1	Week of January 1
VBFirstFourDays	2	First week of the year that has at least four days
VBFirstFullWeek	3	First full week of the year
VBUseSystem	0	Use the date format of the computer's regional settings
VBUseSystemDayOfWeek	0	Use the first full day of the week as defined by the system settings

There is one mandatory argument.

Date

The **Date** argument is any valid date expression and you may use the **Date** and **Now** functions.

Code:
```
<% =WeekDay(Date) %>
```

Output:
```
3
```

Code:
```
<% =WeekDay(Now) %>
```

Output:
3

Code:
<% =WeekDay("6/26/1943") %>

Output:
7

There is one optional argument.

FirstDayofWeek

The optional **FirstDayofWeek** argument must only use the constants or values defined above in the Date And Time Constants.

In this example, Monday is defined to be the first day of the week. Therefore, Saturday becomes 6.

Code:
<% =WeekDay("6/26/1943", VBMonday) %>

Output:
6

FUNCTION: WeekdayName( )

WeekdayName(WeekDay, Abbreviate, FirstDayofWeek)

The **WeekdayName** function returns the full spelling of the name of the day of the week.

There is one mandatory argument.

WeekDay

The **WeekDay** argument is the number representing the day of the week as defined below by the value in the **Date And Time Constants**.

CONSTANT	VALUE	DESCRIPTION
VBSunday	1	Sunday
VBMonday	2	Monday
VBTuesday	3	Tuesday
VBWednesday	4	Wednesday
VBThursday	5	Thursday
VBFriday	6	Friday
VBSaturday	7	Saturday
VBFirstJan1	1	Week of January 1
VBFirstFourDays	2	First week of the year that has at least four days
VBFirstFullWeek	3	First full week of the year
VBUseSystem	0	Use the date format of the computer's regional settings
VBUseSystemDayOfWeek	0	Use the first full day of the week as defined by the system settings

Code:

```
<% =WeekdayName(7) %>
```

Output:

Saturday

There are two optional arguments.

Abbreviate

The optional **Abbreviate** argument is a Boolean value which gives the option of having the returned day name being abbreviated to the first three characters.

If set to **True**, the name will be abbreviated. If set to **False** the name will not be abbreviated.

Code:
```
<% =WeekdayName(7, True) %>
```

Output:
Sat

FirstDayofWeek

The optional **FirstDayofWeek** argument must only use the constants or values defined above in the Date And Time Constants.

In this example, Monday is defined to be the first day of the week. Therefore, Saturday becomes 6.

Code:
```
<% =WeekdayName(6, False, 2) %>
```

Output:
Saturday

Code:
```
<% =WeekdayName(6, False, VBMonday) %>
```

Output:
Saturday

STATEMENT: While

While . . . WEnd

The **While** conditional statement repeats a block of code as long as a conditional test is **True**.

You must end **While** statements with **WEnd** or you will get an error message. You can place **While** statements inside of other conditional statements.

Code:

```
<%
While testnumber = 1000
  Rem You can place all of the code you desire inside a While loop
WEnd
%>
```

Code:

```
<%
While Hour(Time) < 6
  Rem You can place all of the code you desire inside a While loop
WEnd
%>
```

STATEMENT: With ... End With

With object
any legal code
End With

Implemented in version 5.0

The **With** statement allows you to execute code on the named object. It is very important to understand that you can only specify a single object to be acted upon. You cannot list several objects.

You can nest **With** statements inside of other **With** statements or inside other conditional statements, such as an **If Else**. However, do not jump into or out of **With ... End With** blocks of code. The problem is that either the **With** or **End With** portion of the statement may not be executed and this could cause errors or generate erroneous results.

Between the object name and **End With**, you may place any legal block of VBScript code. For example, you could place statements assigning values to the properties of the designated object, as shown below.

Code:

```
<%
With CatObject
  Rem assign properties
  .CatName = "Amy"
  .CatAge = 3
  .CatColor = "black, orange, white"
End With
%>
```

FUNCTION: DatePart( )

Implemented in version 2.0

DatePart(Interval, Date, FirstDayofWeek, FirstWeekofYear)

The **DatePart** function returns the designated part of the date.

There are two mandatory arguments.

Interval

The **Interval** argument designates the the type of time interval.

Only the following settings may be used. You must place the setting inside a pair of double quotes.

SETTING	DESCRIPTION
YYYY	Year
Q	Quarter
M	Month
Y	Day Of Year
D	Day
W	WeekDay
WW	Week Of Year
H	Hour
N	Minute
S	Second

Date

The **Date** argument is the date and time you designate.

Code:
```
<% =Date %>
<% =DatePart("D", Date) %>
```

Output:
```
3/16/99
16
```

The weekday is given as a number from the Date And Time CONSTANTS which are listed

below. In the example, 7 equals Saturday.

Code:
```
<% =DatePart("W", 1/1/2000) %>
```

Output:
7

There are two optional arguments.

FirstDayofWeek

The **FirstDayofWeek** argument must only use the constants or values defined below in the Date And Time CONSTANTS.

CONSTANT	VALUE	DESCRIPTION
VBSunday	1	Sunday
VBMonday	2	Monday
VBTuesday	3	Tuesday
VBWednesday	4	Wednesday
VBThursday	5	Thursday
VBFriday	6	Friday
VBSateurday	7	Saturday
VBFirstJan1	1	Week of January 1
VBFirstFourDays	2	First week of the year thathas at least four days
VBFirstFullWeek	3	First full week of the year
VBUseSystem	0	Use the date format of the computer's regionsl settings
VBUseSystemDayOfWeek	0	Use the first full day of the week as defined by the system settings

In this example, Tuesday is defined to be the first day of the week. Therefore, Saturday becomes the 5th day of the week.

Code:
```
<% =DatePart("W", "1/1/2000", 3) %>
```

Output:
5

Code:
```
<% =DatePart("W", "1/1/2000", VBTUESDAY) %>
```

Output:
5

FirstWeekofYear

The **FirstWeekofYear** argument must only use the constants or values defined in the Date And Time CONSTANTS which are listed above.

This example returns how many full 7 day weeks there were in 1999.

Code:
```
<% =DatePart("WW", "12/31/1999", 1, VBFIRSTFULLWeek) %>
```

Output:
52

FUNCTION: DateSerial( )

Implemented in version 1.0

DateSerial(Year, Month, Day)

The DateSerial function converts the arguments into the variant of subtype Date.

There are three mandatory arguments.

Year

The **Year** argument is the year as a string or integer.

Month

The **Month** argument is the month as a string or integer.

Day

The **Day** argument is the Day as a string or integer.

Code:
```
<% =DateSerial(1943, 6, 26) %>
```

Output:
6/26/43

Code:
```
<% =DateSerial("43", "6", "26") %>
```

Output:
6/26/43

CONSTANTS: Date And Time

Implemented in version 2.0

Date And Time Constants

You may use either the CONSTANT (left column) or the VALUE (center column) in your code and get the same results.

CONSTANT	VALUE	DESCRIPTION
VBSunday	1	Sunday
VBMonday	2	Monday
VBTuesday	3	Tuesday
VBWednesday	4	Wednesday
VBThursday	5	Thursday
VBFriday	6	Friday
VBSaturday	7	Saturday
VBFirstJan1	1	Week of January 1
VBFirstFourDays	2	First week of the year that has at least four days
VBFirstFullWeek	3	First full week of the year
VBUseSystem	0	Use the date format of the computer's regional setting
VBUseSystemDayOfWeek	0	Use the first day of the week defined by system settings

FUNCTION: DateValue( )

Implemented in version 1.0

DateValue(Date)

The **DateValue** function converts an argument into the variant of subtype Date.

The **Date** argument must be a string in valid date format or you will get an error message.

Code:
```
<% mystring = "06/26/1943" %>
<% =DateValue(mystring) %>
```

Output:
6/26/1943

Code:
```
<% =DateValue("1/1/2001") %>
```

Output:
1/1/2001

FUNCTION: Day( )

Implemented in version 1.0

Day(Date)

The **Day** function returns the number of the current day of the month using any valid date expression as an argument.

You can also use the **Date** and **Now** functions as the argument.

Code:
```
<% =Day(Date) %>
```

Output:
16

Code:
```
<% =Day("6/26/1943") %>
```

Output:
26

FUNCTION: Hex( )

Implemented in version 1.0

Hex(Number)

The **Hex** function returns the hexadecimal value of an integer number.

Code:
```
<% =Hex(123) %>
```

Output:
7B

You can also use a negative integer number.

Code:
```
<% =Hex(-123) %>
```

Output:
FF85

For a number with a decimal point (floating-point), the digits to the right of the decimal point are ignored.

Code:
```
<% =Hex(123.456) %>
```

Output:
7B

FUNCTION: Hour( )

Implemented in version 1.0

Hour(Time)

The **Hour** function returns the hour by using the **Time** function as an argument.

The output is base upon a 24 clock (military time, values will run from 0 to 23).

Code:
```
<% =Hour(Time) %>
```

Output:
14

STATEMENT: If ... Then ... Else

Implemented in version 1.0

If . . . Then

The **If** conditional statement executes groups of statements only when a single test condition is **True**.

You can place **If** statements inside of other conditional statements. You must end all **If** statements with **End If** or you will get an error message. Also, the **Then** must be on the same line as the **If**.

Code:
```
<%
If mydate = "6/26/1943" Then
  Rem You can place any code you desire here
End If
%>
```

Code:
```
<%
If somenumber < 71 Then
  Rem You can place any code you desire here
End If
%>
```

If . . . Then . . . Else

The **If** part of the **If ... Else** statement conditionally executes groups of statements only when a single test condition is **True**.

The **Else** part of the **If ... Else** statement conditionally executes groups of statements only when the test **If** condition is **False**.

You can place **If ... Else** statements inside of other conditional statements. You must end all **If ...Else** statements with **End If** or you will get an error message. Also, the **Then** must be on the same line as the **If**.

Code:
```
<%
If somenumber < 71 Then
   Rem You can place any code you desire here
Else
  Rem You can place any code you desire here
End If
%>
```

If . . . Then . . . ElseIf

The **If ... ElseIf** statements allows you to test multiple conditions.

The **If** part of the **If ... ElseIf** statement conditionally executes groups of statements only when

the test condition is **True**.

The **ElseIf** part of the **If ... ElseIf** statement conditionally executes groups of statements only when the test **If** condition is **False** and the test condition for that specific **ElseIf** part is **True**.

You can place **If ... ElseIf** statements inside of other conditional statements. You must end all **If ...ElseIf** statements with **End If** or you will get an error message. Also, the **Then** must be on the same lines as the **If** and the **ElseIf**.

Code:
```
<%
If somenumber < 43 Then
    Rem You can place any code you desire here
ElseIf somenumber < 77 Then
   Rem You can place any code you desire here
ElseIf somenumber < 94 Then
   Rem You can place any code you desire here
End If
%>
```

Code:
```
<%
If somenumber < 43 Then
    Rem You can place any code you desire here
ElseIf somenumber < 77 Then
   Rem You can place any code you desire here
ElseIf somenumber < 94 Then
   Rem You can place any code you desire here
Else
   Rem You can place any code you desire here
End If
%>
```

OPERATOR: Imp

Implemented in version 1.0

Imp

The **Imp** operator is used to perform a logical implication on two expressions, where the expressions are **Null**, or are of **Boolean** subtype and have a value of **True** or **False**.

The **Imp** operator can also be used a "bitwise operator" to make a bit-by-bit comparison of two integers. If both bits in the comparison are the same (both are 0's or 1's), then a 1 is returned. If the first bit is a 0 and the second bit is a 1, then a 1 is returned. If the first bit is a 1 and the second bit is a 0, then a 0 is returned.

The order of the expressions is important.

Code:
```
<% =True Imp True %>
<% =True Imp False %>
<% =False Imp True %>
<% =False Imp False %>

<% =True Imp Null %>
<% =Null Imp True %>
<% =False Imp Null %>
<% =Null Imp False %>
<% =Null Imp Null %>
```

Output:
```
True
False
True
True

(Null output)
True
True
(Null output)
(Null output)
```

Code:
```
<% AnyExpression = True %>
<% SomExpression = False %>
<% =AnyExpression Imp SomeExpression %>
```

Output:
```
False
```

EVENT: Initialize

Implemented in version 5.0

Initialize

The **Initialize** event is an optional subroutine that can be called when you create an instance of a object using the **Class ... End Class** statement. You can place any valid VBScript code inside the subroutine that you want to run when the class is initialized.

The similar **Terminate** event can be called when you close an object.

Note, as shown in the code example, that the subroutine name must be composed of the word Class, an underscore (_), and the word Initialize().

Code:

```
Class DevGuru
...
  Private Sub Class_Initialize()
    'place any valid VBScript code here
  End Sub
...
End Class
```

STATEMENT: Private

Implemented in version 1.0

Private

The **Private** statement is used to declare a new variable or array. You can declare more than one variable and/or array at a time. Memory is also allocated. You cannot declare a variable and assign a value to it at the same time.

If declared within a script, the results are the same as using **Dim** or **Public**, the private variable is available for use in the entire script. If declared in a **Class**, the private variable or array is not available as a public property of the **Class**.

Code:

```
<%
Private  MyNumber
%>

<%
' First declare
Private  MyNumber
' Then assign a value
MyNumber = 1895.405
%>

<%
Private  SomeNumber, MyVar, AnotherString
%>
```

STATEMENT: Property Get

Implemented in version 5.0

Property Get ... End Property

The **Property Get** statement block allows you to perform a procedure that will return the value of a property.

The **Property Get** statement block can only be used inside a **Class** statement block and cannot be used inside of any other procedure. There are two other related statements that also can only be used within a **Class** block, **Property Let** and **Property Set** that, respectively, permit you to assign the value of a property and to set a reference to an object.

The **Property Get** statement allows you to declare the name of the procedure, to accept and change the values of optional arguments, and to perform a series of statements that ultimately return the value of a property.

By default, the **Property Get** procedure is **Public**, but you can declare it to be **Private**. You can use the optional **Exit Property** statement to immediately exit the from inside the **Property Get** procedure, but do not forget to assign the return value before doing such an exit.

Note how the returned value is assigned to the procedure name, "ProcedureName" in the syntax example. It can be assigned from an expression or set to an object.

The **End Property** is required.

Syntax:

[**Public** | **Private**] **Property Get** ProcedureName [arguments, ..., ...]
    Rem You may place code here
    [ProcedureName = expression | **Set** ProcedureName = expression]
    [**Exit Property**]
    Rem You may place code here
    [ProcedureName = expression | **Set** ProcedureName = expression]
End Property

STATEMENT: Property Let

Implemented in version 5.0

Property Let ... End Property

The **Property Let** statement block allows you to perform a procedure that assigns the value of a property.

The **Property Let** statement block can only be used inside a **Class** statement block and cannot be used inside of any other procedure. There are two other related statements that also can only be used within a **Class** block, **Property Get** and **Property Set** that, respectively, permit you to return the value of a property and to set a reference to an object.

The **Property Let** statement allows you to declare the name of the procedure, to accept and change the values of optional arguments, and to perform a series of statements that ultimately assign the value of the property.

There is one mandatory argument. You must give the name of a variable to which the value is to be assigned. This same name must also appear on the right side of the calling expression. (i.e., the name "VariableName" is used in the syntax example.)

By default, the **Property Let** procedure is **Public**, but you can declare it to be **Private**. You can use the optional **Exit Property** statement to immediately exit the from inside the **Property Let** procedure.

The **End Property** is required.

Syntax:

[**Public** | **Private**] **Property Get** ProcedureName ([arguments, ..., ...] VariableName)
    Rem You may place code here
    [**Exit Property**]
    Rem You may place code here
End Property

STATEMENT: Property Set

Implemented in version 5.0

Property Set ... End Property

The **Property Set** statement block allows you to perform a procedure that sets a reference to an object.

The **Property Set** statement block can only be used inside a **Class** statement block and cannot be used inside of any other procedure. There are two other related statements that also can only be used within a **Class** block, **Property Get** and **Property Let** that, respectively, permit you to return the value of a property and to assign the value of a property.

The **Property Set** statement allows you to declare the name of the procedure, to accept and change the values of optional arguments, and to perform a series of statements that ultimately sets the reference to the object.

There is one mandatory argument. You must give the name of a reference variable to which the value is to be assigned. This same name must be used on the right side of the object reference assignment. (i.e., the name "ReferenceName" is used in the syntax example.)

By default, the **Property Set** procedure is **Public**, but you can declare it to be **Private**. You can use the optional **Exit Property** statement to immediately exit the from inside the **Property Set** procedure.

The **End Property** is required.

Syntax:

[**Public** | **Private**] **Property Set** ProcedureName ([arguments, ..., ...] ReferenceName)
    Rem You may place code here
    [**Exit Property**]
    Rem You may place code here
End Property

STATEMENT: Public

Implemented in version 1.0

Public

The **Public** statement is used to declare a new public variable or array. You can declare more than one variable and/or array at a time. Memory is also allocated. You cannot declare a variable and assign a value to it at the same time.

The purpose of **Public** has changed with the addition of **Class** to VBScript. The recommendation is to only use **Public** inside a **Class**. A variable declared using **Public** in a **Class** becomes a public property of that **Class**. That means that other code contained in the same **Class** can access the variable.

If declared within a script, the results are the same as using **Dim** or **Private**, the public variable or array is available for use in the entire script.

Code:

```
<%
Public  YourNumber
%>

<%
' First declare
Public  YourNumber
' Then assign a value
YourNumber = 1895.405
%>

<%
Public  SomeNumber, MyVar, AnotherString
%>
```

OPERATOR: Xor

Xor

The **Xor** operator is used to perform a logical exclusion on two expressions, where the expressions are **Null**, or are of **Boolean** subtype and have a value of **True** or **False**.

The **Xor** operator can also be used a "bitwise operator" to make a bit-by-bit comparison of two integers. If both bits are the same in the comparison (both are 0's or 1's), then a 0 is returned. Otherwise, a 1 is returned.

Code:
```
<% =True Xor True %>
<% =True Xor False %>
<% =False Xor True %>
<% =False Xor False %>
```

Output:
```
False
True
True
False
```

Code:
```
<% expression1 = True %>
<% expression2 = False %>
<% =expression1 Xor expression2 %>
```

Output:
```
True
```

In this example, the **Xor** performs a bitwise comparison on the 3 (in binary 011) and the 5 (in binary 101), and returns a 6 (in binary 110).

Code:
```
<%
Expression1 = 3
Expression2 = 5
Result = Expression1 Xor Expression2
Response.Write "Result = " & Result
%>
```

Output:
```
Result = 6
```

FUNCTION: Year( )

Year(Date)

The **Year** function returns the number of the current year using any valid date expression as an argument.

You can also use the **Date** and **Now** functions as the argument.

Code:
```
<% =Year(Now) %>
```

Output:
1999

Code:
```
<% =Year("6/26/43") %>
```

Output:
1943