VBSCRIPT

- ✓ Scripting Language
- ✓ Light version of Microsoft's Programming Language Visual Basic
- ✓ Server side and client side both
- ✓ Client side is only supported by Internet Explorer

Client-Side Example

```
<html>

<body>

<script type="text/vbscript">

Document.write("Hello World!")

</script>

</body>

</html>
```

Variables

Declaring variables in ASP is simple, especially since all variables are of Variant type. What does this mean to you? You don't have to declare if your variable is an integer, string, or object. You just declare it, and it has the potential to be anything. To declare a variable in ASP/VBScript we use the Dim statement.

```
<%@ LANGUAGE="VBSCRIPT" %>
<%
'Commented lines starting with an apostrophe
'are not executed in VBScript
'First we will declare a few variables.

Dim myText, myNum
myText = "Have a nice day!"
myNum = 5
Response.Write(myText)

'To concatenate strings in VBScript, use the ampersand
Response.Write(" My favourite number is " & myNum)
%>
```

In ASP/VBScript, it is possible not to declare variables at all. A variable can appear in the program, though it has never been declared. It is called default declaring. Variable in this case will be of Variant type.

However, such practice leads to errors and should be avoided. For VB to consider any form or module variable that was not declared explicitly as erroneous, Option Explicit statement should appear in the form or module main section before any other statements. Option Explicit demands explicit declaration of all variables in this form or module. If module contains Option Explicit statement, than upon the attempt to use undeclared or incorrectly typed variable name, an error occurs at compile time.

In naming variables in VBScript you must be aware of these rules:

- Variables must begin with a letter not a number or an underscore
- They cannot have more than 255 characters
- They cannot contain a period (.) , a space or a dash
- They cannot be a predefined identifier (such as dim, variable, if, etc.)
- Case sensitivity is not important in VBScript

The variable value will be kept in memory for the life span of the current page and will be released from memory when the page has finished executing. To declare variables accessible to more than one ASP file, declare them as session variables or application variables.

Contstants

Constants just as variables are used to store information. The main difference between constants and variables is that constant value can not be changed in the process of running program. If we attempt to re-assign the value of the constant we'll get a run time error.

It can be mathematic constants, passwords, paths to files, etc. By using a constant you "lock in" the value which prevents you from accidentally changing it. If you want to run a program several times using a different value each time, you do not need to search throughout the entire program and change the value at each instance. You only need to change it at the beginning of the program where you set the initial value for the constant.

To declare a constant in VBScript we use the Const keyword. Have a look at the following example:

```
Const myConst = "myText"

'it is allowed to declare a few constants on the one line
Const PI = 3.14159, Wg  = 2.78
```

Conditional Statements

If ... Then ... Else Statement

The If Statement is a way to make decisions based on a variable or some other type of data. For example, you might have a script that checks if Boolean value is true or false or if variable contains number or string value.

Use the if statement to execute a statement if a logical condition is true. Use the optional else clause to execute a statement if the condition is false. The syntax for If statement looks as follows:

```
if condition then
  statements_1
else
```

```
  statements_2
end if
```

Condition can be any expression that evaluates to true or false. If condition evaluates to true, statements_1 are executed; otherwise, statements_2 are executed. statement_1 and statement_2 can be any statement, including further nested if statements.

You may also compound the statements using elseif to have multiple conditions tested in sequence. You should use this construction if you want to select one of many sets of lines to execute.

```
if condition_1 then
  statement_1
[elseif condition_2 then
  statement_2]
...
[elseif condition_n_1 then
  statement_n_1]
[else
  statement_n]
end if
```

Let's have a look at the examples. The first example decides whether a student has passed an exam with a pass mark of 57

```
<%@ language="vbscript"%>
<%
Dim Result
Result = 70

if Result >= 57 then
   response.write("Pass <br />")
else
   response.write("Fail <br />")
end if
%>
```

Next example use the elseif variant on the if statement. This allows us to test for other conditions if the first one wasn't true. The program will test each condition in sequence until:

- It finds one that is true. In this case it executes the code for that condition.
- It reaches an else statement. In which case it executes the code in the else statement.
- It reaches the end of the if ... elseif ... else structure. In this case it moves to the next statement after the conditional structure.

```
<%@ language="vbscript"%>
<%
Dim Result
Result = 70

if Result >= 75 then
   response.write("Passed: Grade A <br />")
elseif Result >= 60 then
```

```
   response.write("Passed: Grade B <br />")
elseif Result >= 45 then
   response.write("Passed: Grade C <br />")
else
   response.write("Failed <br />")
end if
%>
```

Select Case Statement

The Select statements work the same as if statements. However the difference is that they can check for multiple values. Of course you do the same with multiple if..else statements, but this is not always the best approach.

The Select statement allows a program to evaluate an expression and attempt to match the expression's value to a case label. If a match is found, the program executes the associated statement. The syntax for the Select statement as follows:

```
select case expression
  case label_1
    statements_1
  case label_2
    statements_2
  ...
  case else
   statements_n
end select
```

The program first looks for a case clause with a label matching the value of expression and then transfers control to that clause, executing the associated statements. If no matching label is found, the program looks for the optional Case Else clause, and if found, transfers control to that clause, executing the associated statements. If no Case Else clause is found, the program continues execution at the statement following the end of Select. Use break to prevent the code from running into the next case automatically.

Let's consider an example:

```
<%@ language="vbscript">
<%
Dim Flower
Flower = "rose"

select case flower
 case "rose"
   response.write(flower & " costs $2.50")
 case "daisy"
   response.write(flower & " costs $1.25")
 case "orchild"
   response.write(flower & " costs $1.50")
 case else
   response.write("There is no such flower in our shop")
end select
%>
```

Looping Statements

ASP performs several types of repetitive operations, called "looping". Loops are set of instructions used to repeat the same block of code till a specified condition returns false or true depending on how you need it. To control the loops you can use counter variable that increments or decrements with each repetition of the loop.

The two major groups of loops are For..Next and Do..Loop. The For...Next statements are best used when you want to perform a loop a specific number of times. The Do...Loop statements are best used to perform a loop an undetermined number of times. In addition, you can use the Exit keyword within loop statements.

- The For ... Next Loop
- The For Each ... Next Loop
- The Do ... Loop
- The Exit Keyword

The For ... Next Loop

For...Next loops are used when you want to execute a piece of code a set number of times. The syntax is as follows:

```
For counter = initial_value to  finite_value [Step increment]
  statements
Next
```

The For statement specifies the counter variable and its initial and finite values. The Next statement increases the counter variable by one. Optional the Step keyword allows to increase or decrease the counter variable by the value you specify.

Have a look at the very simple example:

```
<%
For i = 0 to 10 Step 2 'use i as a counter
 response.write("The number is " & i & "<br />")
Next
%>
```

The preceding example prints out even numbers from 0 to 10, the
 tag puts a line break in between each value.

Next example generates a multiplication table 2 through 9. Outer loop is responsible for generating a list of dividends, and inner loop will be responsible for generating lists of dividers for each individual number:

```
<%
response.write("<h1>Multiplication table</h1>")
response.write("<table border=2 width=50%")

For i = 1 to 9            'this is the outer loop
  response.write("<tr>")
  response.write("<td>" & i & "</td>")
```

```
  For j = 2 to 9          'inner loop
     response.write("<td>" & i * j & "</td>")
  Next 'repeat the code and move on to the next value of j

  response.write("</tr>")
Next 'repeat the code and move on to the next value of i

response.write("</table>")
%>
```

The For Each ... Next Loop

The For Each...Next loop is similar to a For...Next loop. Instead of repeating the statements a specified number of times, the For Each...Next loop repeats the statements for each element of an array (or each item in a collection of objects).

The following code snippet creates drop-down list where options are elements of an array:

```
<%
Dim bookTypes(7) 'creates first array
bookTypes(0)="Classic"
bookTypes(1)="Information Books"
bookTypes(2)="Fantasy"
bookTypes(3)="Mystery"
bookTypes(4)="Poetry"
bookTypes(5)="Humor"
bookTypes(6)="Biography"
bookTypes(7)="Fiction"

Dim arrCars(4) 'creates second array
arrCars(0)="BMW"
arrCars(1)="Mercedes"
arrCars(2)="Audi"
arrCars(3)="Bentley"
arrCars(4)="Mini"

Sub createList(some_array) 'takes an array and creates drop-down list
  dim i
  response.write("<select name=""mylist"">" & vbCrLf) 'vbCrLf stands for Carriage Return and Line
Feed
  For Each item in some_array
     response.write("<option value=" & i & ">" & item & "</option>" & vbCrLf)
     i = i + 1
  Next 'repeat the code and move on to the next value of i
  response.write("</select>")
End Sub

'Now let's call the sub and print out our lists on the screen
Call createList(bookTypes) 'takes bookTypes array as an argument
Call createList(arrcars) 'takes arrCars array as an argument
%>
```

The Do..while Loop

The Do...Loop is another commonly used loop after the For...Next loop. The Do...Loop statement repeats a block of statements an indefinite number of times. The statements are repeated either while a condition is True or until a condition becomes True. The syntax looks as follows:

```
Do [While|Until] condition
  statements
Loop
```

Here is another syntax:

```
Do
  statements
Loop [While|Until] condition
```

In this case the code inside this loop will be executed at least one time. Have a look at the examples:

The example below defines a loop that starts with i=0. The loop will continue to run as long as i is less than, or equal to 10. i will increase by 1 each time the loop runs.

```
<%
Dim i 'use i as a counter
i = 0 'assign a value to i

Do While i<=10 'Output the values from 0 to 10
  response.write(i & "<br \>")
  i = i + 1 'increment the value of i for next time loop executes
Loop
%>
```

Now let's consider a more useful example which creates drop-down lists of days, months and years. You can use this code for registration form, for example.

```
<%
'creates an array
Dim month_array(11)
month_array(0) = "January"
month_array(1) = "February"
month_array(2) = "March"
month_array(3) = "April"
month_array(4) = "May"
month_array(5) = "June"
month_array(6) = "July"
month_array(7) = "August"
month_array(8) = "September"
month_array(9) = "October"
month_array(10) = "November"
month_array(11) = "December"

Dim i
response.write("<select name=""day"">" & vbCrLf)
i = 1
```

```
Do While i <= 31
  response.write("<option value=" & i & ">" & i & "</option>" & vbCrLf)
  i = i + 1
Loop
response.write("</select>")

response.write("<select name=""month"">" & vbCrLf)
i = 0
Do While i <= 11
  response.write("<option value=" & i & ">" & month_array(i) & "</option>" & vbCrLf)
  i = i + 1
Loop
response.write("</select>")

response.write("<select name=""year"">")
i = 1900
Do Until i = 2005
  response.write("<option value=" & i & ">" & i & "</option>" & vbCrLf)
  i = i + 1
Loop
response.write("</select>")
%>
```

Note: Make sure the condition in a loop eventually becomes false; otherwise, the loop will never terminate.

The Exit Keyword

The Exit keyword alters the flow of control by causing immediate exit from a repetition structure. You can use the Exit keyword in different situations, for example to avoid an endless loop. To exit the For...Next loop before the counter reaches its finite value you should use the Exit For statement. To exit the Do...Loop use the Exit Do statement.

Have a look at the example:

```
<%
response.write("<p><strong>Example of using the Exit For statement:</strong><p>")

For i = 0 to 10
  If i=3 Then Exit For
  response.write("The number is " & i & "<br />")
Next

response.write("<p><strong>Example of using the Exit Do statement:</strong><p>")

i = 5
Do Until i = 10
 i = i - 1
 response.write("The number is " & i & "<br />")
 If i < 10 Then Exit Do
Loop
%>
```

Arrays

The VBScript arrays are 0 based, meaning that the array element indexing starts always from 0. The 0 index represents the first position in the array, the 1 index represents the second position in the array, and so forth.

There are two types of VBScript arrays - static and dynamic. Static arrays remain with fixed size throughout their life span. To use static VBScript arrays you need to know upfront the maximum number of elements this array will contain. If you need more flexible VBScript arrays with variable index size, then you can use dynamic VBScript arrays. VBScript dynamic arrays index size can be increased/decreased during their life span.

Static Arrays

Let's create an array called 'arrCars' that will hold the names of 5 cars:

```
<%
'Use the Dim statement along with the array name
'to create a static VBScript array
'The number in parentheses defines the array's upper bound
Dim arrCars(4)
arrCars(0)="BMW"
arrCars(1)="Mercedes"
arrCars(2)="Audi"
arrCars(3)="Bentley"
arrCars(4)="Mini"

'create a loop moving through the array
'and print out the values
For i=0 to 4
response.write arrCars(i) & "<br>"
Next     'move on to the next value of i
%>
```

Here is another way to define the array in VBScript:

```
<%
'we use the VBScript Array function along with a Dim statement
'to create and populate our array
Dim arrCars
arrCars = Array("BMW","Mercedes","Audi","Bentley","Mini") 'each element must be separated by a comma

'again we could loop through the array and print out the values
For i=0 to 4
response.write arrCars(i) & "<br>"
Next
%>
```

Dynamic Arrays

Dynamic arrays come in handy when you aren't sure how many items your array will hold. To create a dynamic array you should use the Dim statement along with the array's name, without specifying upper bound:

```
<%
Dim arrCars
arrCars = Array()
%>
```

In order to use this array you need to use the ReDim statement to define the array's upper bound:

```
<%
Dim arrCars
arrCars = Array()
Redim arrCars(27)
%>
```

If in future you need to resize this array you should use the Redim statement again. Be very careful with the ReDim statement. When you use the ReDim statement you lose all elements of the array. Using the keyword PRESERVE in conjunction with the ReDim statement will keep the array we already have and increase the size:

```
<%
Dim arrCars
arrCars = Array()
Redim arrCars(27)
Redim PRESERVE arrCars(52)
%>
```

Multidimensional Arrays

Arrays do not have to be a simple list of keys and values; each location in the array can hold another array. This way, you can create a multi-dimensional array.

The most commonly used are two-dimensional arrays. You can think of a two-dimensional array as a matrix, or grid, with width and height or rows and columns. Here is how you could define two-dimensional array and display the array values on the web page:

```
<%@ LANGUAGE="VBSCRIPT" %>
<%
Dim arrCars(2,4)

'arrCars(col,row)
arrCars(0,0) = "BMW"
arrCars(1,0) = "2004"
arrCars(2,0) = "45.000"
arrCars(0,1) = "Mercedes"
arrCars(1,1) = "2003"
arrCars(2,1) = "57.000"
arrCars(0,2) = "Audi"
arrCars(1,2) = "2000"
arrCars(2,2) = "26.000"
```

```
arrCars(0,3) = "Bentley"
arrCars(1,3) = "2005"
arrCars(2,3) = "100.00"
arrCars(0,4) = "Mini"
arrCars(1,4) = "2004"
arrCars(2,4) = "19.00"

Response.Write(" <TABLE border=0>")
Response.Write("<TR><TD>Row</TD> <TD>Car</TD>")
Response.Write("<TD>Year</TD><TD>Price</TD></TR>")

'The UBound function will return the 'index' of the highest element in an array.
For i = 0 to UBound(arrCars, 2)
Response.Write("<TR><TD>#" & i & "</TD>")
Response.Write("<TD>" & arrCars(0,i) & "</TD>")
Response.Write("<TD>" & arrCars(1,i) & "</TD>")
Response.Write("<TD>" & arrCars(2,i) & "</TD></TR>")
Next

Response.Write("</TABLE>")

%>
```

OR You can loop the above array as:

```
For i=0 to ubound(arrCars,1)

 For j=0 to ubound(arrCars,2)

        Response.write(arrCars(i,j) & "<br/>")

  Next

Next
```

Functions and Procedures

Functions and procedures provide a way to create re-usable modules of programming code and avoid rewriting the same block of code every time you do the particular task. If you don't have any functions/procedures in your ASP page, the ASP pages are executed from top to bottom, the ASP parsing engine simply processes your entire file from the beginning to the end. VBScript functions and procedures, however, are executed only when called, not inline with the rest of the code. A function or procedure can be reused as many times as required, thus saving you time and making for a less clustered looking page.

You can write functions in ASP similar to the way you write them in Visual Basic. It is good programming practice to use functions to modularize your code and to better provide reuse. To declare a subroutine (a function that doesn't return a value, starts with the Sub keyword and ends with End Sub), you simply type:

```
<%@ LANGUAGE="VBSCRIPT" %>
<%
```

```
Sub subroutineName( parameter_1, ... , parameter_n )
  statement_1
  statement_2
   ...
  statement_n
end sub
%>
```

A function differs from a subroutine in the fact that it returns data, start with Function keyword and end with End Function. Functions are especially good for doing calculations and returning a value. To declare a function, the syntax is similar:

```
<%@ LANGUAGE="VBSCRIPT" %>
<%
Function  functionName( parameter_1, ... , parameter_n )
  statement_1
  statement_2
   ...
  statement_n
end function
%>
```

Have a look at the code for a procedure that is used to print out information on the page:

```
<%@ LANGUAGE="VBSCRIPT" %>
<%
Sub GetInfo(name, phone, fee)
  Response.write("Name: "& name &"<br>")
  Response.write("Telephone: "& telephone &"<br>")
  Response.write("Fee: "& fee &"<br>")
End Sub
%>
```

Now let's consider how to call the sub. There are two ways:

```
<%
'the first method
Call GetInfo("Mr. O'Donnel","555-5555",20)
'the second one
GetInfo "Mr. O'Donnel","555-5555",20
%>
```

In each example, the actual argument passed into the subprocedure is passed in the corresponding position. Note that if you use the Call statement, the arguments must be enclosed in parentheses. If you do not use call, the parentheses aren't used.

Now let's look at the code for a function that takes an integer value and returns the square of that value. Also included is code to call the function.

```
<%
Function Square(num)
  Square = num * num
end function
```

```
  'Returns 25
  Response.Write(Square(5))

  'Should print "45 is less than 8^2"
  if 40 < Square(7) then
    Response.Write("45 is less than 8^2")
  else
    Response.Write("8^2 is less than 40")
  end if
%>
```

How to process the data submitted from HTML form

The great advantage of ASP is possibility to respond to user queries or data submitted from HTML forms. You can process information gathered by an HTML form and use ASP code to make decisions based off this information to create dynamic web pages. In this tutorial we will show how to create an HTML form and process the data.

Before you can process the information, you need to create an HTML form that will send information to your ASP page. There are two methods for sending data to an ASP form: POST and GET. These two types of sending information are defined in your HTML form element's method attribute. Also, you must specify the location of the ASP page that will process the information.

Below is a simple form that will send the data using the POST method. Information sent from a form with the POST method is invisible to others and has no limits on the amount of information to send. Copy and paste this code and save it as "form.html".

```
<html>
<head>
<title>Process the HTML form data with the POST method</title>
</head>
<body>
<form method="POST" action="process.asp" name="form1">
<table width="70%" border="0" cellspacing="0" cellpadding="0">
<tr>
  <td>name:</td>
  <td colspan="2"><input type="text" name="name"></td>
</tr>
<tr>
  <td>email:</td>
  <td colspan="2"><input type="text" name="email"></td>
</tr>
<tr>
  <td>comments:</td>
  <td colspan="2"><textarea name="comment" cols="40" rows="5"></textarea></td>
</tr>
<tr>
  <td> </td>
  <td colspan="2"><input type="submit" name="Submit" value="Submit"></td>
</tr>
</table>
```

```
</form>
</body>
</html>
```

Next, we are going to create our ASP page "process.asp" that will process the data. In our example we decided to send data with the POST method so to retrieve the information we can use the ASP 'Request.From' command. Copy and paste this code and save it in the same directory as "form.html".

```
<%@ Language="VBscript" %>
<html>
<head>
<title>Submitted data</title>
</head>

<body>
<%
'declare the variables that will receive the values
Dim name, email, comment
'receive the values sent from the form and assign them to variables
'note that request.form("name") will receive the value entered
'into the textfield called name
name=Request.Form("name")
email=Request.Form("email")
comment=Request.Form("comment")

'let's now print out the received values in the browser
Response.Write("Name: " & name & "<br>")
Response.Write("E-mail: " & email & "<br>")
Response.Write("Comments: " & comment & "<br>")
%>
</body>
</html>
```

Note: If you want to process the information sent through an HTML form with the GET method you should use the 'Request.QueryString' command . In the preceding example you should replace all instances of Form with QueryString. But remember that the data sent from a form with the GET method is visible to everyone (it will be displayed in the browser's address bar) and has limits on the amount of information to send.

The Dictionary Object

The Dictionary object is used to store information in name/value pairs (referred to as key and item). The Dictionary object might seem similar to Arrays, however, the Dictionary object is a more desirable solution to manipulate related data.

Comparing Dictionaries and Arrays:

- Keys are used to identify the items in a Dictionary object
- You do not have to call ReDim to change the size of the Dictionary object

- When deleting an item from a Dictionary, the remaining items will automatically shift up
- Dictionaries cannot be multidimensional, Arrays can
- Dictionaries have more built-in functions than Arrays
- Dictionaries work better than arrays on accessing random elements frequently
- Dictionaries work better than arrays on locating items by their content

The following example creates a Dictionary object, adds some key/item pairs to it, and retrieves the item value for the key gr:

```
<%
Dim d
Set d=Server.CreateObject("Scripting.Dictionary")
d.Add "re","Red"
d.Add "gr","Green"
d.Add "bl","Blue"
d.Add "pi","Pink"
Response.Write("The value of key gr is: " & d.Item("gr"))
%>
```

Output:

The value of key gr is: Green

The Dictionary object's properties and methods are described below:

Properties

1.CompareMode

The CompareMode property sets or returns the comparison mode for comparing keys in a Dictionary object.

Syntax
DictionaryObject.CompareMode[=compare]

Parameter	Description
Compare	Optional. Specifies the comparison mode. Can take one of the following values: 0 = vbBinaryCompare - binary comparison 1 = vbTextCompare - textual comparison 2 = vbDatabaseCompare - database comparison

Example

```
<%
dim d
set d=Server.CreateObject("Scripting.Dictionary")
d.CompareMode=1
d.Add "n","Norway"
d.Add "i","Italy"

'The Add method will fail on the line below!

d.Add "I","Ireland" 'The letter i already exists
%>
```

2.Count

The Count property returns the number of key/item pairs in the Dictionary object.

Syntax
DictionaryObject.Count

Example

```
<%
dim d
set d=Server.CreateObject("Scripting.Dictionary")
d.Add "n","Norway"
d.Add "i","Italy"
d.Add "s","Sweden"
Response.Write("The number of key/item pairs: " & d.Count)
set d=nothing
%>
```

Output:

The number of key/item pairs: 3

3. Item

The Item property sets or returns the value of an item in a Dictionary object.

Syntax
DictionaryObject.Item(key)[=newitem]

Parameter	Description

Key	Required. The key associated with the item
Newitem	Optional. Specifies the value associated with the key

Example
<%
Dim d
Set d=Server.CreateObject("Scripting.Dictionary")
d.Add "re","Red"
d.Add "gr","Green"
d.Add "bl","Blue"
d.Add "pi","Pink"
Response.Write("The value of key pi is: " & d.Item("pi"))
%>

Output:

The value of key pi is: Pink

4. Key

The Key property sets a new key value for an existing key value in a Dictionary object.

Syntax
DictionaryObject.Key(key)=newkey

Parameter	**Description**
Key	Required. The name of the key that will be changed
Newkey	Required. The new name of the key

Example
<%
Dim d
Set d=Server.CreateObject("Scripting.Dictionary")
d.Add "re","Red"
d.Add "gr","Green"
d.Add "bl","Blue"
d.Add "pi","Pink"
d.Key("re")="r"
Response.Write("The value of key r is: " & d.Item("r"))

%>

Output:

The value of key r is: Red

Methods

1.Add

The Add method adds a new key/item pair to a Dictionary object.

Syntax
DictionaryObject.Add(key,item)

Parameter	Description
key	Required. The key value associated with the item
item	Required. The item value associated with the key

Example
<%
Dim d
Set d=Server.CreateObject("Scripting.Dictionary")
d.Add "re","Red"
d.Add "gr","Green"
d.Add "bl","Blue"
d.Add "pi","Pink"
Response.Write("The value of key gr is: " & d.Item("gr"))
%>
Output:
The value of key gr is: Green

2. Exists

The Exists method returns a Boolean value that indicates whether a specified key exists in the Dictionary object. It returns true if the key exists, and false if not.

Syntax
DictionaryObject.Exists(key)

Parameter	Description

key	Required. The key value to search for

Example

```
<%
dim d
set d=Server.CreateObject("Scripting.Dictionary")
d.Add "n","Norway"
d.Add "i","Italy"
d.Add "s","Sweden"

if d.Exists("n")=true then
  Response.Write("Key exists!")
else
  Response.Write("Key does not exist!")
end if

set d=nothing
%>
```

Output:

Key exists!

3.Items

The Items method returns an array of all the items in a Dictionary object.

Syntax

```
DictionaryObject.Items
```

Example

```
<%
dim d,a,i
set d=Server.CreateObject("Scripting.Dictionary")
d.Add "n","Norway"
d.Add "i","Italy"
d.Add "s","Sweden"

Response.Write("<p>Item values:</p>")
a=d.Items
for i=0 to d.Count-1
  Response.Write(a(i))
  Response.Write("<br />")
next
```

```
set d=nothing
%>
```
Output:
Item values:
Norway
Italy
Sweden

4.Keys

The Keys method returns an array of all the keys in a Dictionary object.

Syntax
DictionaryObject.Keys

Example
```
<%
dim d,a,i
set d=Server.CreateObject("Scripting.Dictionary")
d.Add "n","Norway"
d.Add "i","Italy"
d.Add "s","Sweden"

Response.Write("<p>Key values:</p>")
a=d.Keys
for i=0 to d.Count-1
  Response.Write(a(i))
  Response.Write("<br />")
next

set d=nothing
%>
```

Output:

Key values:
n
i
s

5.Remove

The Remove method removes one specified key/item pair from the Dictionary object.

Syntax
DictionaryObject.Remove(key)

Parameter	Description
key	Required. The key associated with the key/item pair to remove

Example

```
<%
dim d,a,i
set d=Server.CreateObject("Scripting.Dictionary")
d.Add "n","Norway"
d.Add "i","Italy"
d.Add "s","Sweden"

d.Remove("n")

Response.Write("<p>Key values:</p>")
a=d.Keys
for i=0 to d.Count-1
  Response.Write(a(i))
  Response.Write("<br />")
next

set d=nothing
%>
```

Output:

```
Key values:
i
s
```

6.RemoveAll

The Remove method removes all the key/item pairs from a Dictionary object.

Syntax

```
DictionaryObject.RemoveAll
```

Example

```
<%
dim d,a,i
set d=Server.CreateObject("Scripting.Dictionary")
d.Add "n","Norway"
d.Add "i","Italy"
d.Add "s","Sweden"

d.RemoveAll
```

```
Response.Write("<p>Key values:</p>")
a=d.Keys
for i=0 to d.Count-1
  Response.Write(a(i))
  Response.Write("<br />")
next
set d=nothing
%>
```

Output:
Key values:
(nothing)

FileSystemObject Object

The FileSystemObject object is used to access the file system on a server.This object can manipulate files, folders, and directory paths. It is also possible to retrieve file system information with this object.The following code creates a text file (c:\test.txt) and then writes some text to the file:

```
<%
dim fs,fname
set fs=Server.CreateObject("Scripting.FileSystemObject")
set fname=fs.CreateTextFile("c:\test.txt",true)
fname.WriteLine("Hello World!")
fname.Close
set fname=nothing
set fs=nothing
%>
```

The FileSystemObject object's properties and methods are described below:

Properties

Drives

The Drives property returns a collection of all Drive objects on the computer.

Methods

1.BuildPath

The BuildPath method appends a name to an existing path.

Syntax
[newpath=]FileSystemObject.BuildPath(path,name)

Parameter	Description
Path	Required. The path to append a name to
Name	Required. The name to append to the path

Example
```
<%
dim fs,path
set fs=Server.CreateObject("Scripting.FileSystemObject")
path=fs.BuildPath("c:\mydocuments","test")
response.write(path)
set fs=nothing
%>
```
Output:
c:\mydocuments\test

2.CopyFile

The CopyFile method copies one or more files from one location to another.

Syntax
FileSystemObject.CopyFile source,destination[,overwrite]

Parameter	Description
Source	Required. The file or files to copy (wildcards can be used}
destination	Required. Where to copy the file or files (wildcards cannot be used}
overwrite	Optional. A Boolean value that specifies whether an existing file can be overwritten. True allows existing files to be overwritten and False prevents existing files from being overwritten. Default is True

Example
```
<%
dim fs
set fs=Server.CreateObject("Scripting.FileSystemObject")
fs.CopyFile "c:\mydocuments\web\*.htm","c:\webpages\"
set fs=nothing
%>
```

3.CopyFolder

The CopyFolder method copies one or more folders from one location to another.

Syntax
FileSystemObject.CopyFolder source,destination[,overwrite]

Parameter	Description
source	Required. The folder or folders to copy (wildcards can be used)
destination	Required. Where to copy the folder or folders (wildcards cannot be used)
overwrite	Optional. A Boolean value that indicates whether an existing folder can be overwritten. True allows existing folders to be overwritten and False prevents existing folders from being overwritten. Default is True

Examples

```
<%
'copy all the folders in c:\mydocuments\web
'to the folder c:\webpages

dim fs
set fs=Server.CreateObject("Scripting.FileSystemObject")
fs.CopyFolder "c:\mydocuments\web\*","c:\webpages\"
set fs=nothing
%>
<%
'copy only the folder test from c:\mydocuments\web
'to the folder c:\webpages
dim fs
set fs=Server.CreateObject("Scripting.FileSystemObject")
fs.CopyFolder "c:\mydocuments\web\test","c:\webpages\"
set fs=nothing
%>
```

4.CreateFolder

Creates a new folder

5.CreateTextFile

The CreateTextFile method creates a new text file in the current folder and returns a TextStream object that can be used to read from, or write to the file.

Syntax
FileSystemObject.CreateTextFile(filename[,overwrite[,unicode]])

FolderObject.CreateTextFile(filename[,overwrite[,unicode]])

Parameter	**Description**
Filename	Required. The name of the file to create
Overwrite	Optional. A Boolean value that indicates whether an existing file can be overwritten. True indicates that the file can be overwritten and False indicates that the file can not be overwritten. Default is True
Unicode	Optional. A Boolean value that indicates whether the file is created as a Unicode or an ASCII file. True indicates that the file is created as a Unicode file, False indicates that the file is created as an ASCII file. Default is False

Example for the FileSystemObject object

```
<%
dim fs,tfile
set fs=Server.CreateObject("Scripting.FileSystemObject")
set tfile=fs.CreateTextFile("c:\somefile.txt")
tfile.WriteLine("Hello World!")
tfile.close
set tfile=nothing
set fs=nothing
%>
```

Example for the Folder object

```
<%
dim fs,fo,tfile
Set fs=Server.CreateObject("Scripting.FileSystemObject")
Set fo=fs.GetFolder("c:\test")
Set tfile=fo.CreateTextFile("test.txt",false)
tfile.WriteLine("Hello World!")
tfile.Close
set tfile=nothing
set fo=nothing
set fs=nothing
%>
```

6.DeleteFile

The DeleteFile method deletes one or more specified files.

Note: An error will occur if you try to delete a file that doesn't exist.

Syntax
FileSystemObject.DeleteFile(filename[,force])

Parameter	Description
filename	Required. The name of the file or files to delete (Wildcards are allowed)
force	Optional. A Boolean value that indicates whether read-only files will be deleted. True indicates that the read-only files will be deleted, False indicates that they will not be deleted. Default is False

Example

```
<%
dim fs
Set fs=Server.CreateObject("Scripting.FileSystemObject")
fs.CreateTextFile("c:\test.txt",True)
if fs.FileExists("c:\test.txt") then
  fs.DeleteFile("c:\test.txt")
end if
set fs=nothing
%>
```

7.DeleteFolder

The DeleteFolder method deletes one or more specified folders.

Note: An error will occur if you try to delete a folder that does not exist.

Syntax
FileSystemObject.DeleteFolder(foldername[,force])

Parameter	Description
foldername	Required. The name of the folder or folders to delete (Wildcards are allowed)
force	Optional. A Boolean value that indicates whether read-only folders will be deleted. True indicates that read-only folders will be deleted, False indicates that they will not be deleted. Default is False

Example

```
<%
dim fs
set fs=Server.CreateObject("Scripting.FileSystemObject")
if fs.FolderExists("c:\temp") then
  fs.DeleteFolder("c:\temp")
end if
set fs=nothing
%>
```

8.GetFile

The GetFile method returns a File object for the specified path.

Syntax
FileSystemObject.GetFile(path)

Parameter	Description
path	Required. The path to a specific file

Example

```
<%
dim fs,f
set fs=Server.CreateObject("Scripting.FileSystemObject")
set f=fs.GetFile("c:\test\test.htm")
Response.Write("The file was last modified on: ")
Response.Write(f.DateLastModified)
set f=nothing
set fs=nothing
%>
```

Output:
The file was last modified on 01/01/20 4:23:56 AM

9.**MoveFile**

The MoveFile method moves one or more files from one location to another.

Syntax
FileSystemObject.MoveFile source,destination

Parameter	Description
source	Required. The path to the file/files to be moved. Can contain wildcard characters in

	the last component.
destination	Required. Where to move the file/files. Cannot contain wildcard characters

Example
<%
dim fs
set fs=Server.CreateObject("Scripting.FileSystemObject")
fs.MoveFile "c:\web*.gif","c:\images\"
set fs=nothing
%>

10.**OpenTextFile**

The OpenTextFile method opens a specified file and returns a TextStream object that can be used to access the file.

Syntax
FileSystemObject.OpenTextFile(fname,mode,create,format)

Parameter	**Description**
fname	Required. The name of the file to open
mode	Optional. How to open the file 1=ForReading - Open a file for reading. You cannot write to this file. 2=ForWriting - Open a file for writing. 8=ForAppending - Open a file and write to the end of the file.
create	Optional. Sets whether a new file can be created if the filename does not exist. True indicates that a new file can be created, and False indicates that a new file will not be created. False is default
format	Optional. The format of the file 0=TristateFalse - Open the file as ASCII. This is default. -1=TristateTrue - Open the file as Unicode. -2=TristateUseDefault - Open the file using the system default.

Example
<%
dim fs,f
set fs=Server.CreateObject("Scripting.FileSystemObject")
set f=fs.OpenTextFile(Server.MapPath("testread.txt"),8,true)

```
f.WriteLine("This text will be added to the end of file")
f.Close
set f=Nothing
set fs=Nothing
%>
```

The TextStream Object

The TextStream object is used to access the contents of text files.

The following code creates a text file (c:\test.txt) and then writes some text to the file (the variable f is an instance of the TextStream object):

```
<%
dim fs,f
set fs=Server.CreateObject("Scripting.FileSystemObject")
set f=fs.CreateTextFile("c:\test.txt",true)
f.WriteLine("Hello World!")
f.Close
set f=nothing
set fs=nothing
%>
```

To create an instance of the TextStream object you can use the CreateTextFile or OpenTextFile methods of the FileSystemObject object, or you can use the OpenAsTextStream method of the File object.

The TextStream object's properties and methods are described below:

Properties

Property	Description
AtEndOfLine	Returns true if the file pointer is positioned immediately before the end-of-line marker in a TextStream file, and false if not
AtEndOfStream	Returns true if the file pointer is at the end of a TextStream file, and false if not
Column	Returns the column number of the current character position in an input stream
Line	Returns the current line number in a TextStream file

Example:

```
<%
dim fs,f,t,x
set fs=Server.CreateObject("Scripting.FileSystemObject")
```

```
set f=fs.CreateTextFile("c:\test.txt")
f.write("Hello World!")
f.close

set t=fs.OpenTextFile("c:\test.txt",1,false)
do while t.AtEndOfLine<>true
  x=t.Read(1)
loop
t.close
Response.Write("The last character is: " & x)
%>
```

Output:
The last character of the first line in the text file is: !

Methods

1.Close

The Close method closes an open TextStream file.

2. Read

The Read method reads a specified number of characters from a TextStream file and returns the result as a string.

Syntax
TextStreamObject.Read(numchar)

Parameter	Description
Numchar	Required. The number of characters to read from the file

Example

```
<%
dim fs,f,t,x
set fs=Server.CreateObject("Scripting.FileSystemObject")
set f=fs.CreateTextFile("c:\test.txt")
f.write("Hello World!")
f.close

set t=fs.OpenTextFile("c:\test.txt",1,false)
x=t.Read(5)
t.close
```

```
Response.Write("The first five characters are: " & x)
%>
```
Output:
The first five characters are: Hello

3.ReadAll

Reads an entire TextStream file and returns the result

4.ReadLine

The ReadLine method reads one line from a TextStream file and returns the result as a string.

Syntax
TextStreamObject.ReadLine

Example
```
<%
dim fs,f,t,x
set fs=Server.CreateObject("Scripting.FileSystemObject")
set f=fs.CreateTextFile("c:\test.txt")
f.writeline("Line 1")
f.writeline("Line 2")
f.writeline("Line 3")
f.close
set t=fs.OpenTextFile("c:\test.txt",1,false)
x=t.ReadLine
t.close
Response.Write("The first line in the file ")
Response.Write("contains this text: " & x)
%>
```
Output:
The first line in the file contains this text: Line 1

5.Write

The Write method writes a specified text to a TextStream file.

Note: This method write text to the TextStream file with no spaces or line breaks between each string.

Syntax
TextStreamObject.Write(text)

Parameter	Description

Text	Required. The text to write to the file

Example
```
<%
dim fs,f
set fs=Server.CreateObject("Scripting.FileSystemObject")
set f=fs.CreateTextFile("c:\test.txt",true)
f.write("Hello World!")
f.write("How are you today?")
f.close
set f=nothing
set fs=nothing
%>
```

The file test.txt will look like this after executing the code above:

Hello World!How are you today?

6.WriteLine

The WriteLine method writes a specified text and a new-line character to a TextStream file.

Syntax
TextStreamObject.WriteLine(text)

Parameter	Description
text	Optional. The text to write to the file. If you do not specify this parameter, a new-line character will be written to the file

Example
```
<%
dim fs,f
set fs=Server.CreateObject("Scripting.FileSystemObject")
set f=fs.CreateTextFile("c:\test.txt",true)
f.WriteLine("Hello World!")
f.WriteLine("How are you today?")
f.WriteLine("Goodbye!")
f.close
set f=nothing
set fs=nothing
%>
```

The file test.txt will look like this after executing the code above:

Hello World!
How are you today?
Goodbye!

The File Object

The File object is used to return information about a specified file.

To work with the properties and methods of the File object, you will have to create an instance of the File object through the FileSystemObject object. First; create a FileSystemObject object and then instantiate the File object through the GetFile method of the FileSystemObject object or through the Files property of the Folder object.

The following code uses the GetFile method of the FileSystemObject object to instantiate the File object and the DateCreated property to return the date when the specified file was created:

Example

```
<%
Dim fs,f
Set fs=Server.CreateObject("Scripting.FileSystemObject")
Set f=fs.GetFile("c:\test.txt")
Response.Write("File created: " & f.DateCreated)
set f=nothing
set fs=nothing
%>
```

The File object's properties and methods are described below:

Properties

Property	Description
Attributes	Sets or returns the attributes of a specified file
DateCreated	Returns the date and time when a specified file was created
DateLastAccessed	Returns the date and time when a specified file was last accessed
DateLastModified	Returns the date and time when a specified file was last modified

Drive	Returns the drive letter of the drive where a specified file or folder resides
Name	Sets or returns the name of a specified file
ParentFolder	Returns the folder object for the parent of the specified file
Path	Returns the path for a specified file
ShortName	Returns the short name of a specified file (the 8.3 naming convention)
ShortPath	Returns the short path of a specified file (the 8.3 naming convention)
Size	Returns the size, in bytes, of a specified file
Type	Returns the type of a specified file

Methods

Method	**Description**
Copy	Copies a specified file from one location to another
Delete	Deletes a specified file
Move	Moves a specified file from one location to another
OpenAsTextStream	Opens a specified file and returns a TextStream object to access the file

Debugging ASP and Error Handling

Regardless of your level of experience, you will encounter programmatic errors, or *bugs*, that will prevent your server-side scripts from working correctly. For this reason, debugging, the process of finding and correcting scripting errors, is crucial for developing successful and robust ASP applications, especially as the complexity of your application grows. Various tools like Visual Studio can be used to debug ASP Applications.

The Microsoft Script Debugger

The Microsoft Script Debugger is a powerful debugging tool that can help you quickly locate bugs and interactively test your server-side scripts. You can do the following things using the features of Microsoft Script Debugger:

- Run your server-side scripts one line at a time.

- Open a command window to monitor the value of variables, properties, or array elements, during the execution of your server-side scripts.
- Set pauses to suspend execution of your server-side scripts (using either the debugger or a script command) at a particular line of script.
- Trace procedures while running your server-side script.

Enabling Debugging

Before you can begin debugging your server-side scripts, you must first configure your Web server to support ASP debugging.

After enabling Web server debugging, you can use either of the following methods to debug your scripts:

- Open Script Debugger and use it to run and debug your ASP server-side scripts.
- Use Internet Explorer to request an .asp file. If the file contains a bug or an intentional statement to halt execution, Script Debugger will automatically start, display your script, and indicate the source of the error.

Scripting Errors

While debugging your server-side scripts you might encounter several types of errors. Some of these errors can cause your scripts to execute incorrectly, halt the execution of your program, or return incorrect results.

Syntax Errors

A *syntax* error is a commonly encountered error that results from incorrect scripting syntax. For example, a misspelled command or an incorrect number of arguments passed to a function generates an error. Syntax errors can prevent your script from running.

Runtime Errors

Run-time errors occur after your script commences execution and result from scripting instructions that attempt to perform impossible actions. For example, the following script contains a function that divides a variable by zero (an illegal mathematical operation) and generates a run-time error:

```
<SCRIPT Language= "VBScript" RUNAT=SERVER>
  Result = Findanswer(15)
  Document.Write ("The answer is " &Result)

  Function Findanswer(x)
  'This statement generates a run-time error.
   Findanswer = x/0
  End Function
</SCRIPT>
```

Bugs that result in run-time errors must be corrected for your script to execute without interruption.

Logical Errors

A *logical* error can be the most difficult bug to detect. With logical errors, which are caused by typing mistakes or flaws in programmatic logic, your script runs successfully, but yields incorrect results. For example, a server-side script intended to sort a list of values may return an inaccurate ordering if the script contains a > (greater than) sign for comparing values, when it should have used a < (less than) sign.

Just-In-Time(JIT) Debugging

When a run-time error interrupts execution of your server-side script, the Microsoft Script Debugger automatically starts, displays the .asp file with a statement pointer pointing to the line that caused the error, and generates an error message. With this type of debugging, called? *Just-In-Time* (JIT) debugging, your computer suspends further execution of the program. You must correct the errors with an editing program and save your changes before you can resume running the script.

Breakpoint Debugging

When an error occurs and you cannot easily locate the source of the error, it is sometimes useful to preset a *breakpoint*. A breakpoint suspends execution at a specific line in your script. You can set one or many different breakpoints in Microsoft Script Debugger before a suspect line of script and then use the debugger to inspect the values of variables or properties set in the script. After you correct the error, you can clear your breakpoints so that your script can run uninterrupted.

To set a breakpoint, open your script with Script Debugger, select a line of script where you want to interrupt execution, and from the **Debug** menu choose **Toggle Breakpoint**. Then use your Web browser to request the script again. After executing the lines of script up to the breakpoint, your computer starts the Script Debugger, which displays the script with a statement pointer pointing to the line where you set the breakpoint.

The Break at Next Statement

In certain cases, you may want to enable the Script Debugger **Break at Next Statement** if the next statement that runs is not in the .asp file that you are working with. For example, if you set **Break at Next Statement** in an .asp file residing in an application called Sales, the debugger will start when you run a script in any file in the Sales application, or in any application for which debugging has been enabled. For this reason, when you set **Break at Next Statement**, you need to be aware that whatever script statement runs next will start the debugger.

VBScript Stop Statement Debugging

You can also add breakpoints to your server-side scripts written in VBScript by inserting a **Stop** statement at a location before a questionable section of server-side script. For example, the following server-side script contains a**Stop** statement that suspends execution before the script calls a custom function:

VBScript

```
<%
  intDay = Day(Now())
  lngAccount = Request.Form("AccountNumber")
  dtmExpires = Request.Form("ExpirationDate")
```

```
  strCustomerID  =  "RETAIL" & intDay & lngAccount & dtmExpires

  'Set breakpoint here.
  Stop

  'Call registration component.
  RegisterUser(strCustomerID)
%>
```

When you request this script, the debugger starts and automatically displays the .asp file with the statement pointer indicating the location of the **Stop** statement. At this point you could choose to inspect the values assigned to variables before passing those variables to the component.

The ASPError Object

The ASPError object was implemented in ASP 3.0 and is available in IIS5 and later.

The ASPError object is used to display detailed information of any error that occurs in scripts in an ASP page.

Note: The ASPError object is created when Server.GetLastError is called, so the error information can only be accessed by using the Server.GetLastError method.

The ASPError object's properties are described below (all properties are read-only):

Properties

Property	Description
ASPCode	Returns an error code generated by IIS
ASPDescription	Returns a detailed description of the error (if the error is ASP-related)
Category	Returns the source of the error (was the error generated by ASP? By a scripting language? By an object?)
Column	Returns the column position within the file that generated the error
Description	Returns a short description of the error
File	Returns the name of the ASP file that generated the error
Line	Returns the line number where the error was detected
Number	Returns the standard COM error code for the error

Source	Returns the actual source code of the line where the error occurred

Error Handling

Handling errors is achieved by using the **On Error Resume Next** statement**.** It simply tells the ASP interpreter to continue the execution of the ASP Script if there is an error instead of throwing an exception and stopping the execution.

To trap and handle the errors, we need to use the Err Object and its properties

<% if Err.Number <> 0 Then

Handle_Error(Err.Description)

Err.Clear

End If

%>

<% Sub Handle_Error(errordesc)

'Write error to a log file

%>

On Error GoTo 0 Statement is used to disable any error handling

Error handling allows you to display friendly error messages for the end users and the same time it helps you debug the asp application.