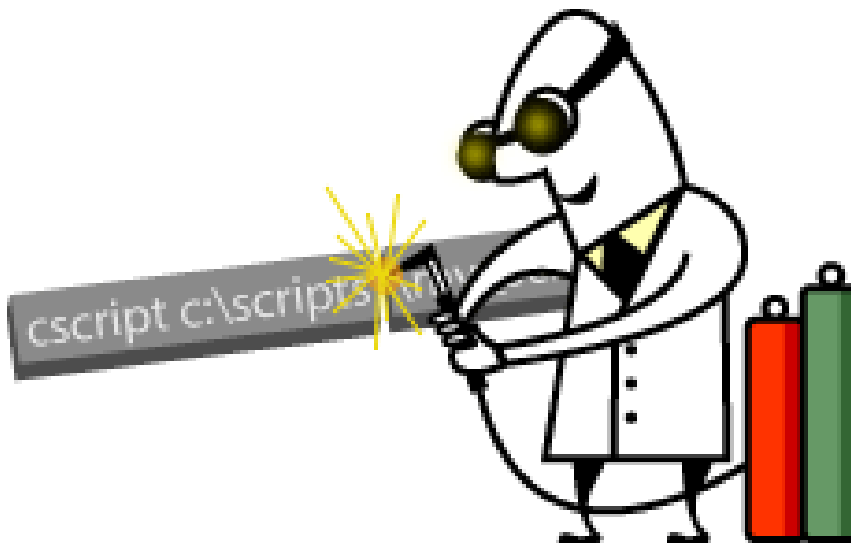


# 2013

## VBScript Concepts



**Automate your work...**

Mayur Kale

[mayur.kale@tcs.com](mailto:mayur.kale@tcs.com)

Corporate ADS Team

# Contents

---

1. VBScript – The Basics .....	3
2. All about Variables .....	5
2.1. Variables Naming Restrictions .....	5
2.2. How Do I Create a Variable? .....	5
2.3. Dim Statement .....	5
2.4. Declaring Variables Explicit and Implicit .....	6
3. Flow Structures .....	7
3.1. If...Then .....	7
3.2. If...Then...Else .....	7
3.3. Select Case .....	8
3.4. Control Structures to Make Code Repeat .....	8
4. Files and Folders .....	13
5. Working with String .....	18
6. WshShell Object .....	21
7. Windows Management Instrumentation(WMI) .....	23
7.1. WBEMTEST .....	23
7.2. WMI Tasks .....	25
8. WMIQuery Language .....	26
8.1. Moniker String .....	26
9. Windows Registry .....	31
9.1. Local Registry Access .....	31
9.2. Remote Registry Access .....	33
9.3. Simple Method .....	36
10. Windows Services .....	37
11. Procedures .....	38
12. Functions .....	40
13. Error Handling .....	42

# VBScript – The Basics

---

VBScript is a Microsoft scripting language. A scripting language available by default with Microsoft Windows. Microsoft Visual Basic Scripting Edition (VBScript) is an easy-to-use scripting language that enables system administrators to create powerful tools for managing their Microsoft Windows based computers. Scripts can not only make your work go faster, they can make your job easier. And once you learn the basic rules of the road, they're not all that difficult to operate.

VBScript is a programming language that is often viewed as a dialect of VBA (Visual Basic for Applications), although it is really its own language. The VBScript language attempts to balance flexibility, capability and ease of use. VBA is a subset of Visual Basic that was developed to automate Microsoft Office applications, whereas VBScript was originally developed to support Server-side and Client-side web applications. Although VBScript and VBA provide many of the same features, there are some differences between them, primarily due to the applications they were each developed to support.

## The Microsoft Visual Basic Family

VBScript is part of a family of Microsoft programming languages that support object-oriented programming. This family of products is derived from the Basic programming language, first developed in 1964

## File Extension

VBScript is having .vbs extension. (Please refer below sample snap)



## Where to run/execute Script?

We can execute Script either in Cscript or Wscript Tool. Available in all windows operating system. Location of tool – C:\Windows\System32

Command – Cscript new.vbs

```
D:\>cscript new.vbs
Microsoft (R) Windows Script Host Version 5.8
Copyright (C) Microsoft Corporation. All rights reserved.
```

By default VBScript execute in Wscript tool, we can change it Cscript.

Command – Cscript //H:cscript

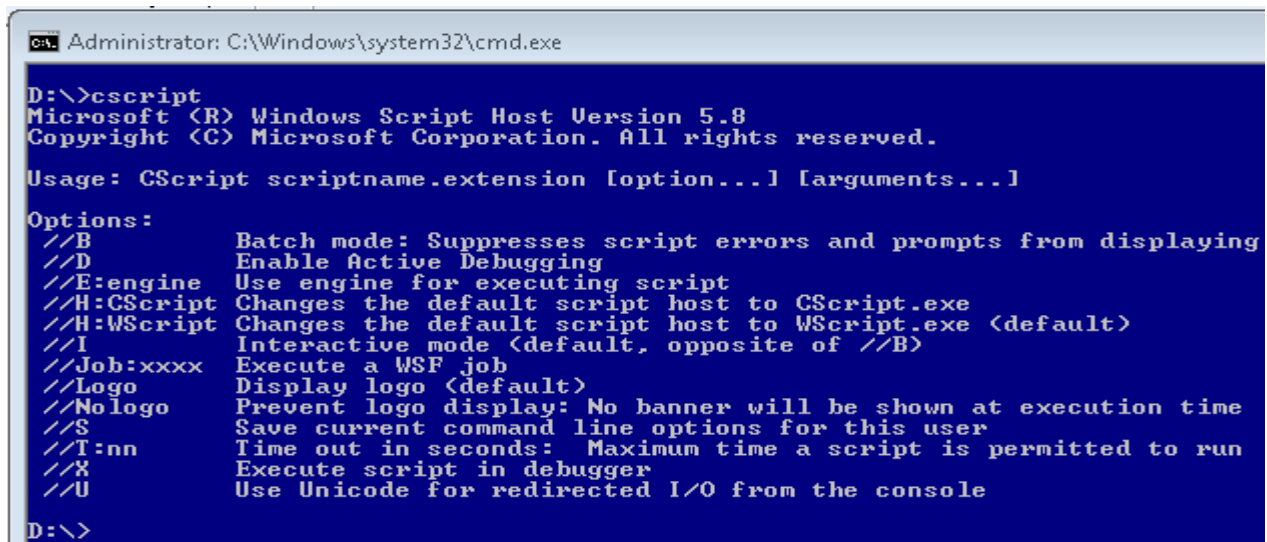
```
D:\>cscript //h:cscript
Microsoft (R) Windows Script Host Version 5.8
Copyright (C) Microsoft Corporation. All rights reserved.

The default script host is now set to "cscript.exe".

D:\>
```

To disable default Microsoft logo while running any vbscript.

Command – cscript new.vbs //nologo



```
C:\Windows\system32\cmd.exe
D:\>cscript
Microsoft (R) Windows Script Host Version 5.8
Copyright (C) Microsoft Corporation. All rights reserved.

Usage: CScript scriptname.extension [option...] [arguments...]

Options:
//B      Batch mode: Suppresses script errors and prompts from displaying
//D      Enable Active Debugging
//E:engine Use engine for executing script
//H:CScript Changes the default script host to CScript.exe
//H:WScript Changes the default script host to WScript.exe (default)
//I      Interactive mode (default, opposite of //B)
//Job:xxxx Execute a WSF job
//Logo   Display logo (default)
//NoLogo Prevent logo display: No banner will be shown at execution time
//S      Save current command line options for this user
//T:nn   Time out in seconds: Maximum time a script is permitted to run
//X      Execute script in debugger
//U      Use Unicode for redirected I/O from the console

D:\>
```

# All about Variables

---

A variable is a virtual container in the computer's memory or placeholder that refers to a computer memory location where you can store program information that may change during the time your script is running. Where the variable is stored in computer memory is unimportant. What is important is that you only have to refer to a variable by name to see or change its value.

## Variables Naming Restrictions

Variable names follow the standard rules for naming anything in VBScript.

A variable name:

- ⤴ Must begin with an alphabetic character.
- ⤴ Must not exceed 255 characters.
- ⤴ Must be unique in the scope in which it is declared.
- ⤴ Make sure you never create variables that have the same name as keywords already used by VBScript. These keywords are called reserved words and include terms such as Date, Minute, Second, Time, and so on.

## How Do I Create a Variable?

When you create a variable, you have to give it a name. That way, when you need to find out what's contained in the variable, you use its name to let the computer know which variable you are referring to. You have two ways to create a variable. The first way, called the explicit method, is where you use the Dim keyword to tell VBScript you are about to create a variable. You then follow this keyword with the name of the variable. If, for example, you want to create a variable called Data, you would enter

**Dim nData**(And the variable will then exist.)

## Dim Statement

The Dim statement declares and allocates storage space in memory for variables. The Dim statement is used either at the start of a procedure or the start of a global script block.

## Syntax

Dim varname[[([subscripts])][, varname[[([subscripts])]]] . . .

## Parameter Description

varname:- Name of the variable; follows standard variable naming conventions.

Subscripts:- An array and optionally specifies the number and extent of array dimensions up to 60 multiple dimensions may be declared

## Notes

When variables are first initialized with the Dim statement, they have a value of Empty.

i.e. **nData** = 0 and/or **nData** = "".

## Declaring Variables Explicit and Implicit

You can declare variables explicitly in your script using the Dim statement, the Public statement, and the Private statement.

**Dim nData**

You can declare multiple variables by separating each variable name with a comma.

**Dim** nTop, nBottom, nLeft, nRight

You can also declare a variable implicitly by simply using its name in your script. That is not generally a good practice because you could misspell the variable name in one or more places, causing unexpected results when your script is run.

For that reason, the Option Explicit statement is available to require explicit declaration of all variables. The Option Explicit statement should be the first

### Option Explicit Statement

Forces explicit declaration of all variables in a script. If used, the Option Explicit statement must appear in a script before any other statements. When you use the Option Explicit statement, you must explicitly declare all variables using the Dim, Private, Public, or ReDim statements.

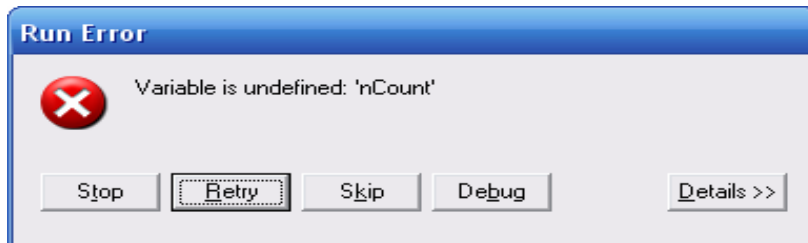
Use Option Explicit to avoid incorrectly typing the name of an existing variable or to avoid confusion in code where the scope of the variable is not clear. The following example illustrates use of the Option Explicit statement.

**Option Explicit** ' Force explicit variable declaration.

**Dim** MyVar ' Declare variable.

nCount = 10 ' Undeclared variable generates error.

MyVar = 10 ' Declared variable does not generate error



### VBScript Constants

Sometimes in writing code, you will want to refer to values that never change. The values for True and False, for example, are always -1 and 0, respectively. Values that never change usually have some special meaning, such as defining True and

False. These values that never change are called constants. The constants True and False are sometimes referred to as implicit constants because you do not need to do anything to reference their constant names. They are immediately available in any code you write.

You create user-defined constants in VBScript using the Const statement. Using the Const statement, you can create string or numeric constants with meaningful names and assign them literal values.

For example:

**Const** TIMEOUT = 54

**Const** MY\_STRING\_CONSTANT = "Hello World"

# FLOW STRUCTURES

---

VBScript gives you a variety of ways to direct the flow of your code. The mechanisms used to accomplish this in VBScript are called control structures. They are called structures because you construct your code around them, much like you build and finish a house around its structure. You can use each control structure to make your code travel in different ways, depending on how you want the decision to be made.

## If...Then

The first control structure you should know about is If...Then. The syntax for this control structure is given as

```
If condition = True Then
    ... the code that executes if the condition is satisfied
End If
```

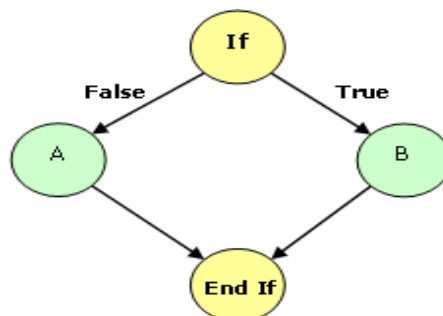
Where condition is some test you want to apply to the conditional structure. If the condition is true, the code within the If and End If statements is executed. If the condition is not true, the code within these statements is skipped over and does not get executed.

## If...Then...Else

The syntax for this control structure is given as

```
If condition = True Then
    ...this is the code that executes if the condition is satisfied
Else
    ...this is the code that executes if the condition is not satisfied
End If
```

Where condition is some test you want to apply to the conditional structure. If the condition is true, the code within the If and Else statements is executed. If the condition is not true, the code within Else and End If statements is executed



A variation on the If...Then...Else statement allows you to choose from several alternatives. Adding ElseIf clauses expands the functionality of the If...Then...Else statement so you can control program flow based on different possibilities:

```

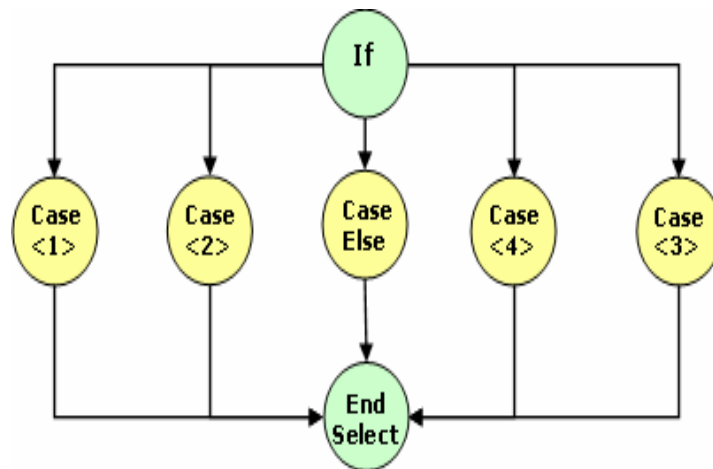
If condition1 = True Then
    ...the code that executes for condition1
ElseIf condition2 = True Then
    ...the code that executes for condition2
ElseIf condition3 = True Then
    ...the code that executes for condition3
End If

```

You can add as many ElseIf clauses as you need to provide alternative choices.

### Select Case

Allows for conditional execution of a block of code, typically out of three or more code blocks, based on some condition. Use the Select Case statement as an alternative to complex nested If...Then...Else statements.



### Example

The following example uses Select Case to read a variable populated by the user and determine the name of the user's operating system:

```

Dim FavCol
FavCol = "red"
Select Case FavCol
Case "Black"
    MsgBox("your FavColour is Black")
Case "red"
    MsgBox("your FavColour is Red")
Case "Yellow"
    MsgBox("your FavColour is Yellow")
Case Else
    MsgBox("Now your just confusing")
End Select

```



## Control Structures to Make Code Repeat

On occasion, you will need to write code that repeats some set of statements. Oftentimes, this will occur when you need to perform some calculation over and over or when you have to apply the same calculations or processing to more than one variable, such as changing the values in an array. This section shows you all the control structures you can use in VBScript to control code in your program that repeats.

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

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

### For...Next Statement

Defines a loop that executes a given number of times, as determined by a loop counter. To use the For...Next loop, you must assign a numeric value to a counter variable. This counter is either incremented or decremented automatically with each iteration of the loop. In the For statement, you specify the value that is to be assigned to the counter initially and the maximum value the counter will reach for the block of code to be executed. The Next statement marks the end of the block of code that is to execute repeatedly, and also serves as a kind of flag that indicates the counter variable is to be modified.

### Syntax

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

### Arguments

counter:- Numeric variable used as a loop counter. The variable can't be an array element or an element of a user-defined type.

Start:- Initial value of counter.

End:- Final value of counter.

Step:- Amount counter is changed each time through the loop. If not specified, step defaults to one.

Statements:- One or more statements between For and Next that are executed the specified number of times.

### Example

```
For var = 0 to 5
    msgbox("hello")

Next
msgbox("Finish")
```

The following example causes a procedure called “wscript.echo x” to execute 50 times. The For statement specifies the counter variable x and its start and end values. The Next statement increments the counter variable by 1 as default.

### Step keyword

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

```
Dim j, total
For j = 1 To 10 Step 2
total = total + j
Next
MsgBox"The total is "& total
```

### Exit For

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

### Syntax

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

### Note

When you use a positive step value, make sure the finish value is greater than the start value, or the loop will not execute at all.

```
For i = 10 to 1 Step 2 ' Incorrect
For i = 1 to 10 Step 2 ' Correct
```

When you use a negative step value, make sure the start value is greater than the finish value, or the loop won't execute at all.

```
For i = 1 to 10 Step -1 ' Incorrect
For i = 10 to 1 Step -1 ' Correct
```

Never use a step value of zero. In this case, VBScript will enter an infinite loop, and your program might run indefinitely.

```
For i = 1 to 10 Step 0 ' Incorrect
For i = 1 to 10 Step 3 ' Correct
```

### For Each...Next Statement

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

## Syntax

```
For Each element In group
    [statements]
[Exit For]
    [statements]
Next
```

## Arguments

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

Group:- Name of an object collection or array.

Statements:- One or more statements that are executed on each item in group.

## Notes

The For...Each code block is executed only if group contains at least one element.

All statements are executed for each element in group in turn until either there are no more elements in group, or the loop is exited prematurely using the Exit For statement. Program execution then continues with the line of code following Next. For Each...Next loops can be nested, but each element must be unique.

## Example:

```
Set colOperatingSystems = objWMIService.ExecQuery("Select
Caption,TotalVisibleMemorySize from Win32_OperatingSystem")

For Each objOperatingSystem in colOperatingSystems
    osname = objOperatingSystem.Caption
    RAM = Round(objOperatingSystem.TotalVisibleMemorySize / 1024) & " MB"
Next
```

## Do...Loops Statement

Repeatedly executes a block of code while or until a condition becomes True.

## Syntax

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

## Arguments

condition:- Numeric or string expression that is True or False. If condition is Null, condition is treated as False.

Statements:- One or more statements that are repeated while or until condition is True.

## Example:

```
Do
nCtr = nCtr + 1 ' Modify loop control variable
MsgBox"Iteration "&nCtr& " of the Do loop..." &vbCrLf
' Compare to upper limit
```

```
If nCtr = 10 Then Exit Do
Loop
```

Adding the Until keyword after Do instructs your program to Do something Until the condition is True. Its syntax is:

```
Do Until condition
    code to execute
Loop
```

**Example:**

```
Do until objInputFile.AtEndOfStream
    objInputFile.ReadLine
Loop
```

**While...Wend Statement**

The While...Wend statement executes a series of statements as long as a given condition is True.

**Syntax**

```
While condition
    Version [statements]
Wend
```

**Arguments**

condition:- Numeric variable used as a loop counter. The variable can't be an array element or an element of a user-defined type.

Statements:- One or more statements between For and Next that are executed the specified number of times.

If condition evaluates to True, the program code between the While and Wend statements executed. After the Wend statement is executed, control is passed back up to the While statement, where condition is evaluated again. When condition evaluates to False, program execution skips to the first statement following the Wend statement.

Link :-<http://technet.microsoft.com/en-us/library/ee176997.aspx>

# FILES AND FOLDERS

---

The File System Object (FSO) model provides an object-based tool for working with folders and files. It allows you to use the familiar object.method syntax with a rich set of properties, methods, and events to process folders and files. You can also employ the traditional Visual Basic statements and commands.

The FSO model gives your application the ability to create, alter, move, and delete folders, or to determine if and where particular folders exist. It also enables you to get information about folders, such as their names and the date they were created or last modified.

The FSO model makes processing files much easier as well. When processing files, your primary goal is to store data in an efficient, easy-to-access format. You need to be able to create files, insert and change the data, and output (read) the data. Although you can store data in a database, doing so adds a significant amount of overhead to your application. You may not want to have such overhead, or your data access requirements may not call for the extra functionality associated with a full-featured database. In this case, storing your data in a text file or binary file is the most efficient solution. The FSO model, contained in the Scripting type library (**Scrrun.dll**), supports the creation and manipulation of text files through the TextStream object; however, the FSO model does not support binary files

The first thing we do in this script is use the CreateObject method to create an instance of the Scripting.FileSystemObject:

```
Set objFSO = CreateObject("Scripting.FileSystemObject")
```

## FileSystemObject Methods

Method	Description
BuildPath	Appends a name to an existing path.
CopyFile	Copies one or more files from one location to another.
CopyFolder	Recursively copies a folder from one location to another.
CreateFolder	Creates a folder.
CreateTextFile	Creates a specified file name and returns a TextStream object.
DeleteFile	Deletes a folder and its contents.
DeleteFolder	Deletes a folder and its contents.
DriveExists	Indicates the existence of a drive.
FileExists	Indicates the existence of a file.
FolderExists	Indicates the existence of a folder.
GetAbsolutePathName	Returns a complete and unambiguous path from a provided path specification.
GetBaseName	Returns the base name of a path.
GetDrive	Returns a Drive object corresponding to the drive in a path
GetDriveName	Returns a string containing the name of the drive for a path.
GetExtensionName	Returns a string containing the extension for the last component in a path.
GetFile	Returns a File object corresponding to the file in a path.
GetFileName	Returns the last component of a path that is not part of the drive specification.
GetFolder	Returns a Folder object corresponding to the folder in a specified path.

GetParentFolderName	Returns a string containing the name of the parent folder.
GetSpecialFolder	Returns the special folder requested.
GetTempName	Returns a randomly generated temporary file or folder name.
MoveFile	Moves one or more files from one location to another.
MoveFolder	Moves one or more folders from one location to another.
OpenTextFile	Opens a file and returns a TextStream object

## 1) Reading and Writing Text Files

### Creating Text Files

The FileSystemObject allows you to either work with existing text files or create new text files from scratch. To create a brand-new text file, simply create an instance of the FileSystemObject and call the CreateTextFile method, passing the complete path name as the method parameter.

```
Set objFSO = CreateObject("Scripting.FileSystemObject")
Set objFile = objFSO.CreateTextFile("C:\FSO\ScriptLog.txt")
objFile.Close
```

In the code shown above, the CreateObject function returns the FileSystemObject. The CreateTextFile method then creates the file as a TextStream object (objFile), and the Close method flushes the buffer and closes the file.

### Opening Text Files

Working with text files is a three-step process. Before you can do anything else, you must open the text file. This can be done either by opening an existing file or by creating a new text file. (When you create a new file, that file is automatically opened and ready for use.) Either approach returns a reference to the TextStream object.

For reading (parameter value = 1, constant = ForReading).

Files opened in this mode can only be read from. To write to the file, you must open it a second time by using either the ForWriting or ForAppending mode.

For writing (parameter value 2, constant = ForWriting).

Files opened in this mode will have new data replace any existing data. (That is, existing data will be deleted and the new data added.) Use this method to replace an existing file with a new set of data.

For appending (parameter value 8, constant = ForAppending).

Files opened in this mode will have new data appended to the end of the file. Use this method to add data to an existing file.

```
Const ForReading = 1
Const ForWriting = 2
Const ForAppending = 8
Set objFSO = CreateObject("Scripting.FileSystemObject")
Set objTextFile = objFSO.OpenTextFile("c:\scripts\servers.txt", ForReading)
```

### Read Text Files

```
Set objFSO = CreateObject("Scripting.FileSystemObject")
Set objFile = objFSO.OpenTextFile("ScriptLog.txt", 1)
```

```
Wscript.Echo "Reading file the first time:"
strContents = objFile.ReadAll
Wscript.Echo strContents
```

```
Wscript.Echo "Reading file the second time:"
Do until objFile.AtEndOfStream
    strLine = objFile.ReadLine
    Wscript.Echo strLine
Loop
```

## Write Text Files

```
Set objFSO = CreateObject("Scripting.FileSystemObject")
Set objFile = objFSO.OpenTextFile("ScriptLog.txt", 2)

strLine = "Script started..."
strContents = strLine & vbCrLf & "Processing..." & vbCrLf & "Script End."
objFile.Write(strContents)
objFile.WriteLine "Thanks"
```

Above example just showed that you can write either one line at a time (using WriteLine) or an entire file (using Write). You can do the same with reading. We saw how to read one line at a time with the ReadLine method; we can also read the entire contents of a text file into memory all at once using the ReadAll method.

## Closing Text Files

Any text files opened by a script are automatically closed when the script ends. Because of this, you do not have to explicitly close text files any time you open them.

```
Set objFSO = CreateObject("Scripting.FileSystemObject")
Set objFile = objFSO.OpenTextFile("C:\FSO\ScriptLog.txt", 1)
objFile.Close
```

## 2) Files and Folders Exists

```
If objFSO.FolderExists("C:\scripts") Then
    If objFSO.FileExists("C:\scripts\test.txt") Then
        Wscript.Echo "Folder and file exist"
    Else
        Wscript.Echo "Folder exists, file doesn't"
    End If
Else
    Wscript.Echo "Folder does not exist"
End If
```

## 3) Files Copy

```
Set objFSO = CreateObject("Scripting.FileSystemObject")
Set objFile = objFSO.GetFile("C:\scripts\test.txt")

objFile.Copy "C:\scripts\temp\"
```

Incidentally, the Copy method does accept one more parameter. This parameter specifies whether or not you want to overwrite the file if a file by the same name already exists in the destination folder. The default is True, meaning the file will be overwritten. If you don't want existing files to be overwritten,

pass False as the second parameter to the Copy method:

```
objFile.Copy "C:\scripts\temp\", False
```

#### 4) Folder Copy

```
Set objFSO = CreateObject("Scripting.FileSystemObject")
Set objFolder = objFSO.GetFolder("C:\scripts")

objFolder.Copy "C:\temp"
```

We can also copy files and folders without having to first connect to the file or folder object. In addition to the Copy method, the FileSystemObject provides two other methods for copying files and folders: CopyFile and CopyFolder. Let's take a look at CopyFile first:

```
Set objFSO = CreateObject("Scripting.FileSystemObject")
objFSO.CopyFile "C:\scripts\test.txt", "C:\scripts\temp"
```

#### 5) MoveFile

To move that file rather than copy it, simply replace the Copy method with the Move method:

```
Set objFSO = CreateObject("Scripting.FileSystemObject")
Set objFile = objFSO.GetFile("C:\test.txt")

objFile.Move "d:\vbs\"
```

Other than the obvious fact that Move moves and Copy copies, there's one big difference between these methods that you need to know about. While the Copy method accepts a second parameter that determines whether or not an existing file is overwritten - with a default of True - the Move method doesn't accept any other parameters. Not only that, but the default, and, in this case, the only behavior is just the opposite. If you run the script we just showed you and the file test.txt already exists in C:\scripts\temp, the file will not be moved and you'll receive an error message:

C:\scripts\test.vbs(4, 1) Microsoft VBScript runtime error: File already exists

#### 6) Delete File

The Delete method deletes a specific file or folder.  
Here's the script:

```
Set objFSO = CreateObject("Scripting.FileSystemObject")
Set objFile = objFSO.GetFile("C:\scripts\temp\test.txt")

objFile.Delete
```

You can force a delete of everything, including the read-only files and folders, by passing True to the Delete method:

```
objFolder.Delete True
```



Just like with copy and move, you can delete sets of files and folders using wildcards and the file- and folder-specific implementations of the methods, in this case DeleteFile and DeleteFolder. Here's an example using DeleteFile:

```
Set objFSO = CreateObject("Scripting.FileSystemObject")
objFSO.DeleteFile "C:\scripts\temp\*.txt"
```

This script deletes all files in the C:\scripts\temp folder that end with .txt. As with the Delete method, DeleteFile also takes an additional parameter specifying whether to force the deletion of read-only files. The default is False, don't delete read-only files.

And here's how you delete a bunch of folders:

```
Set objFSO = CreateObject("Scripting.FileSystemObject")
objFSO.DeleteFolder "C:\scripts\x"
```

This script deletes all folders (and their contents) that begin with the letter x from the C:\scripts folder

# Working with String

---

## Right

The Right function takes a string and returns a certain number of characters from the right side of that string. Here's what the definition of the Right function looks like:

```
string Right(string, integer)
```

### Sample Script

```
str = "football"  
intCharacters = 4  
  
strNew = Right(str, intCharacters)  
Wscript.Echo strNew
```

## Left

The Left function takes a string and returns a certain number of characters from the left side of that string. Here's what the definition of the Left function looks like:

```
string Left(string, integer)
```

### Sample Script

```
str = "football"  
intCharacters = 4  
  
strNew = Left(str, intCharacters)  
Wscript.Echo strNew
```

## Mid

The Mid function takes a string and number as a input and returns a certain number of characters from the Middle part of that string. Here's what the definition of the Mid function looks like:

```
string Mid(string, integer [, integer])
```

### Sample Script

```
str = "football"  
intCharacters = 4  
intStart = 3  
  
strNew = Mid(str, intStart, intCharacters)  
Wscript.Echo strNew
```

The last parameter, the one where we put in the number of characters to retrieve, has square brackets around it. This means that parameter is optional, so you don't actually have to tell Mid how many characters to retrieve. What happens if you don't? Well, in that case Mid simply returns all the characters in the string from the starting position on. In other words, suppose we remove intCharacters from our script, like this:

## Sample Script

```
str = "football"  
intStart = 3  
  
strNew = Mid(str, intStart)  
Wscript.Echo strNew
```

## Len

The Len function tells you the length of a string. Here's the definition:

```
integer Len(string)
```

## Sample Script

```
str = "football"  
  
intLen = Len(str)  
Wscript.Echo intLen
```

## LCase

Converts all the alphabetic characters in a string to their lowercase equivalents.

```
UserName = "mAYUR"  
Wscript.Echo LCase(UserName)
```

## UCase

Converts all the alphabetic characters in a string to their uppercase equivalents.

```
UserName = "mAYUR"  
Wscript.Echo UCase(UserName)
```

## InStr

The InStr function provides a way to search for the presence of a string within another string. InStr works by reporting back the character position where the substring is first found. For example, suppose you are looking for the letters ment in the word developmental. InStr returns the value 8 because the letters ment are found beginning in the eighth character position.

```
Wscript.Echo InStr("developmental", "ment")
```

By contrast, if you searched for the letters mant, InStr would return 0, indicating that the substring could not be found.

## Sample Script

```
TestString = "This is a test string being searched for two different words."  
PresentString = "test"  
AbsentString = "strong"  
Wscript.Echo InStr(TestString, PresentString)  
Wscript.Echo InStr(TestString, AbsentString)
```

When the preceding script runs, the values 11 and 0 will be returned. The value 11 indicates that the word test can be found beginning with the eleventh character. The value 0 indicates that the word strong could not be found anywhere within the string being searched.

## Split

The Split function separates a string into substrings and creates a one-dimensional array where each substring is an element.

```
Split(expression[, delimiter[, count[, compare]])
```

### Arguments

**expression** : Required. String expression containing substrings and delimiters. If expression is a zero-length string, Split returns an empty array, that is, an array with no elements and no data.

**delimiter** : Optional. String used to identify substring limits. If omitted, the space character (" ") is assumed to be the delimiter. If delimiter is a zero-length string, a single-element array containing the entire expression string is returned.

**count** : Optional. Number of substrings to be returned; -1 indicates that all substrings are returned. If omitted, all substrings are returned.

**compare** : Optional. Numeric value indicating the kind of comparison to use when evaluating substrings. See Settings section for values. If omitted, a binary comparison is performed.

vbBinaryCompare	0	Perform a binary comparison.
vbTextCompare	1	Perform a textual comparison.

## Sample Script

```
Dim MyString, MyArray, i
MyString = "VBScript|is|fun!"
MyArray = Split(MyString, "|", -1, 1)
' MyArray(0) contains "VBScript".
' MyArray(1) contains "is".
' MyArray(2) contains "fun!".

For i = 0 to UBound(MyArray)
    MsgBox (MyArray(i))
Next
```

# WshShell Object

---

You create a WshShell object whenever you want to run a program locally, manipulate the contents of the registry, create a shortcut, or access a system folder. The WshShell object provides the Environment collection. This collection allows you to handle environmental variables (such as WINDIR, PATH, or PROMPT)

A script can use either the Run method or the Exec method to run a program in a manner similar to using the Run dialog box from the Start menu. Regardless of the method used, the program starts, and runs in a new process.

## Run Method

In Run method : your script will not have access to the standard input, output, and error streams generated by the program being run. A script cannot use the Run method to run a command-line tool and retrieve its output.

For example, suppose you want to run Ping.exe and then examine the output to see whether the computer could be successfully contacted. This cannot be done using the Run command. Instead, you would need to ping the computer, save the results of the ping command to a text file, open the text file, read the results, and then parse those results to determine the success or failure of the command.

## Syntax

object.Run(strCommand, [intWindowStyle], [bWaitOnReturn])

## Arguments

**object** : WshShell object.

**strCommand**: String value indicating the command line you want to run. You must include any parameters you want to pass to the executable file.

**intWindowStyle** : Optional. Integer value indicating the appearance of the program's window. Note that not all programs make use of this information.

**bWaitOnReturn** : Optional. Boolean value indicating whether the script should wait for the program to finish executing before continuing to the next statement in your script. If set to true, script execution halts until the program finishes, and Run returns any error code returned by the program. If set to false (the default), the Run method returns immediately after starting the program, automatically returning 0 (not to be interpreted as an error code).

The following table lists the available settings for intWindowStyle.

intWindowStyle	Description
0	Hides the window and activates another window.
1	Activates and displays a window. If the window is minimized or maximized, the system restores it to its original size and position. An application should specify this flag when displaying the window for the first time.
2	Activates the window and displays it as a minimized window.
3	Activates the window and displays it as a maximized window.

## Sample Script 1

```
strCommand = "ping.exe 127.17.251.98"
Set WshShell = CreateObject("WScript.Shell")

i = WshShell.Run(strCommand, 1, True)
wscript.echo i
```

## Sample Script 2

```
Const MAXIMIZE_WINDOW = 3
Set objShell = WScript.CreateObject("WScript.Shell")
objShell.Run "notepad.exe", MAXIMIZE_WINDOW
```

Although this approach works, it is somewhat complicated. If you need access to command-line output, you should use the Exec method instead. The following script also parses the output generated by Ping.exe. However, it does so by using the Exec method and by directly reading the output. There is no need to create, open, read, and delete a temporary file.

## Exec Method

The Exec method returns a WshScriptExec object, which provides status and error information about a script run with Exec along with access to the StdIn, StdOut, and StdErr channels. The Exec method allows the execution of command line applications only. The Exec method cannot be used to run remote scripts. Do not confuse the Exec method with the Execute method.

## Syntax

```
object.Exec(strCommand)
```

## Arguments

**Object** :WshShell object.

**strCommand** :String value indicating the command line used to run the script. The command line should appear exactly as it would if you typed it at the command prompt.

## Sample Script

```
strCommand = "ping.exe 127.17.251.98"

Set WshShell = CreateObject("WScript.Shell")

Set WshShellExec = WshShell.Exec(strCommand)

strOutput = WshShellExec.StdOut.ReadAll

WScript.Echo strOutput           'write results to default output
MsgBox strOutput
```

# Windows Management Instrumentation(WMI)

---

Windows Management Instrumentation (WMI) is the infrastructure for management data and operations on Windows-based operating systems. You can write WMI scripts or applications to automate administrative tasks on remote computers but WMI also supplies management data to other parts of the operating system and products, for example System Center Operations Manager, formerly Microsoft Operations Manager (MOM), or Windows Remote Management (WinRM).

We can use WMI from client applications and scripts. It provides an infrastructure that makes it easy to both discover and perform management tasks. In addition, you can add to the set of possible management tasks by creating your own WMI providers.

## WMI from these five angles.

- Think of WMI as a database holding information about a computer's disk, services, processor and objects.
- Regard WMI as a method to automate the collection of hardware and software data.
- View WMI as a pipe connecting magically to the inner secrets of the Microsoft operating system.
- Approach WMI as a distinctive dialect of VBScript with its own WQL language.
- Treat WMI as a tool rather like a microscope to probe, and to measure the operating system's properties.

If you think about it, the operating system knows everything! Windows Server 2008 / 2003 must know how much memory each process is using, how much free space there is on each partition, which devices are on which Bus. With WMI scripting, you can tap into the operating system's CIM library and thus query information about any aspect of the Windows Server 2008 / 2003 or XP.

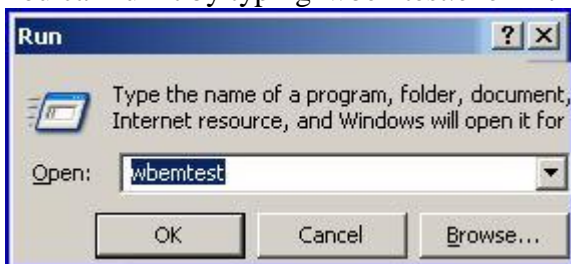
## WBEMTEST - A Tool for Learning About WMI Objects

WBEMTEST is a built-in Windows program which will show you the WMI objects, classes and methods. WMI Tester (Wbemtest.exe) is a tool that provides the basic functionality for executing WQL queries.

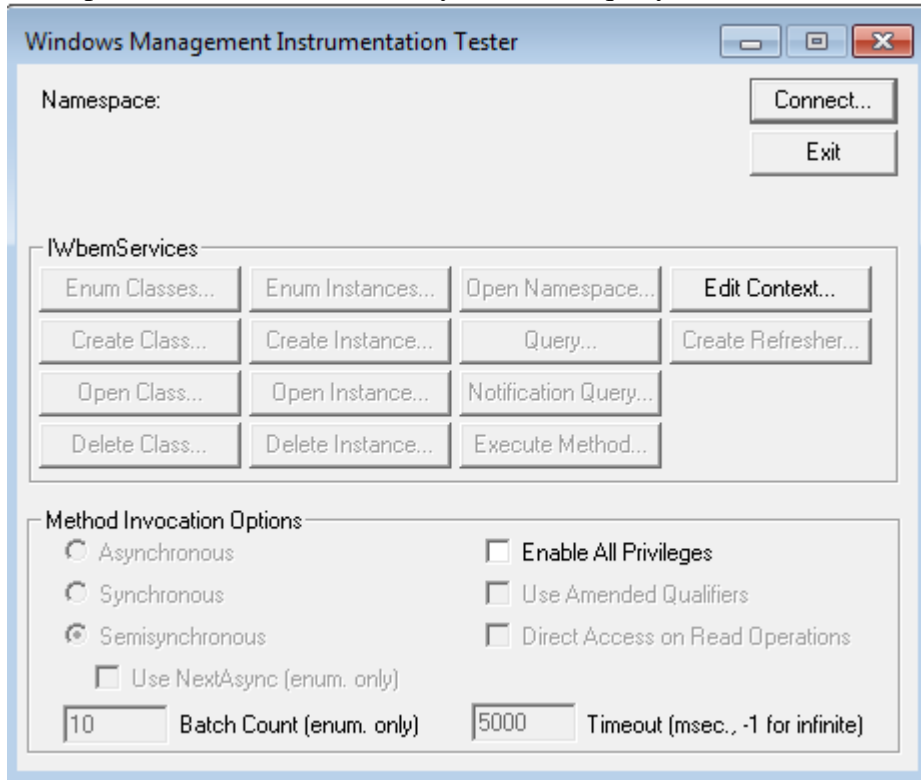
### Launching WBEMTEST

To launch WBEMTEST, all that you need to do is type the name WBEMTEST in the run dialog box. This executable is available on Windows 2000 and later machines.

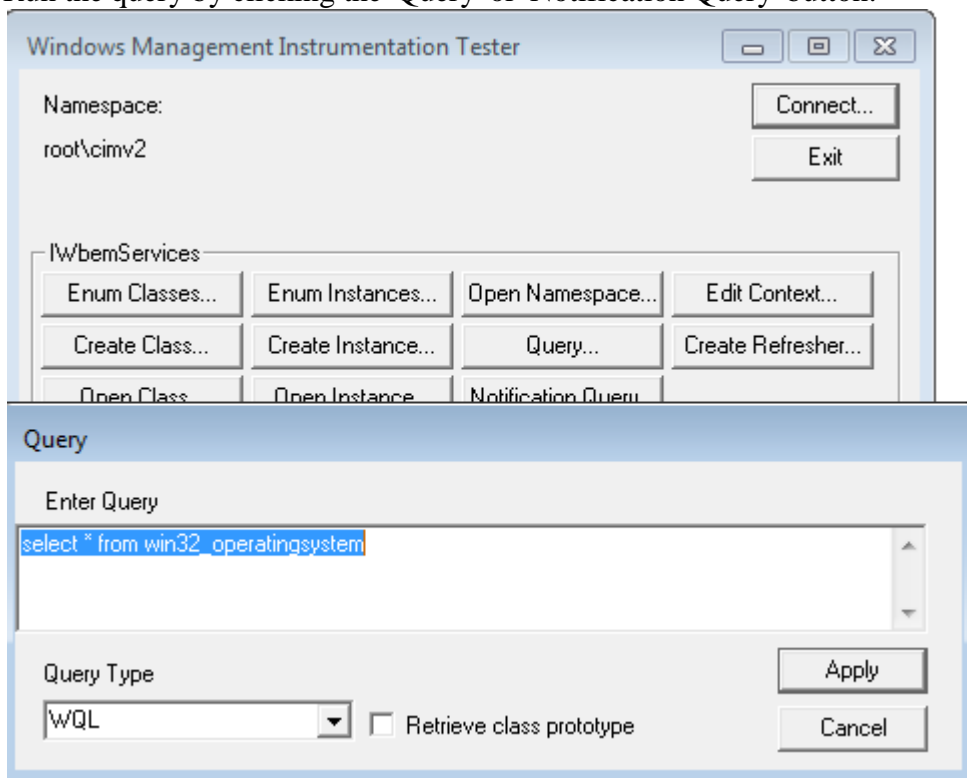
1. You can run it by typing 'wbemtest.exe' in the Run box:



2. Once you launch WBEMTEST, note that the Namespace says: root\default. We want to change this to CIMV2, so click on the Connect... Button. You first need to connect to the WMI namespace that contains the class you want to query (Root\Cimv2 in most cases)

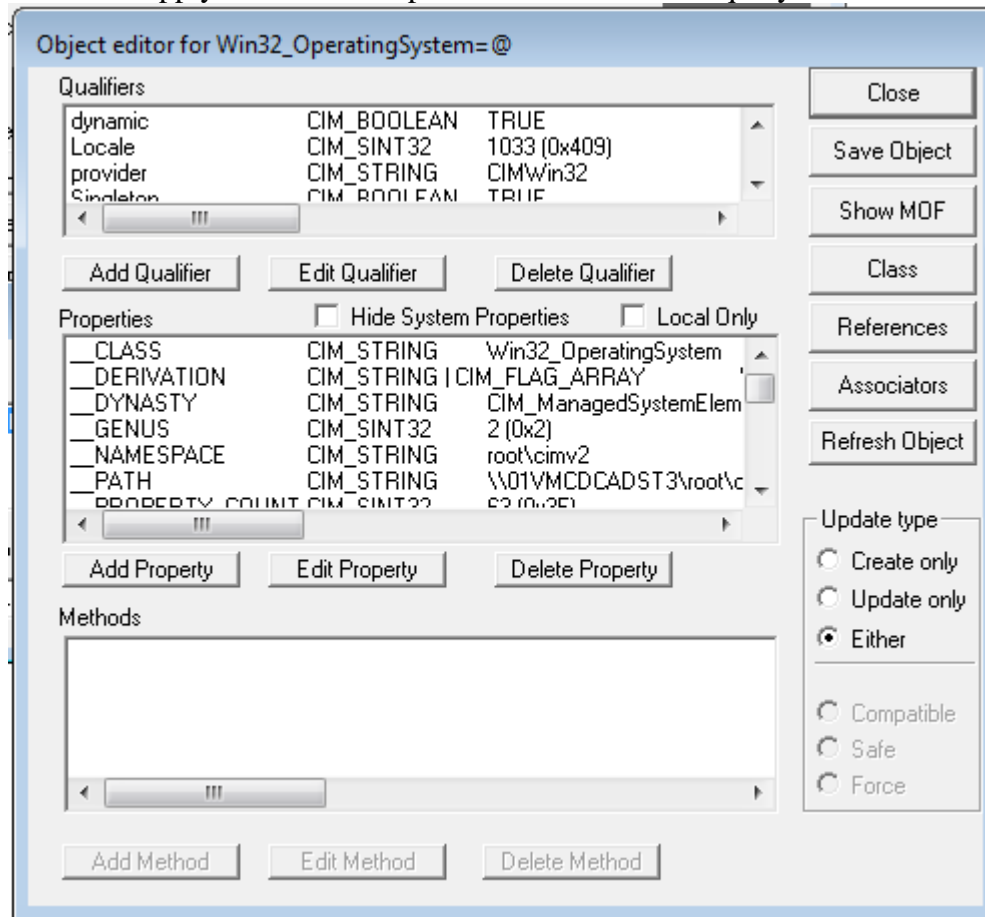


- 1) Run the query by clicking the 'Query' or 'Notification Query' button:





- 2) Click the 'Apply' button. This opens a window with the query results:



## WMI Tasks

- 1) Information of Computer Hardware and Computer Software
- 2) Manage Events log
- 3) Manage Processes
- 4) Manage Registry
- 5) Manage Services
- 6) Manage Scheduled Tasks
- 7) Handle Disk and File Systems
- 8) Printers
- 9) Networking
- 10) Desktop Management
- 11) Dates and Times
- 12) Performance Monitoring

# WMI QUERY LANGUAGE

---

WMI Query Language (WQL) isn't so much a dialect as it is a language within a language. You use a scripting language such as VBScript to access and manipulate WMI objects, but you use WQL to retrieve the exact object or objects you want to work with.

As you've probably gathered from the name, WQL is a query language, much like SQL, a standard used to query databases. WQL is actually a subset of SQL, so if you're familiar with SQL you're way ahead of the game on this one. In the same way that we use If Then and For Next statements to talk to the computer in VBScript, we use specific statements to talk to WMI to retrieve objects. In order to understand the language, we need to look at some of these statements.

WMI Connect to any computer

```
strComputer = "."  
Set objWMIService = GetObject("winmgmts:"  
    & "{impersonationLevel=impersonate}!\\\" & strComputer & "\\root\\cimv2")
```

strComputer to "." (meaning the local machine)

## Introduction to WMI - The Moniker

A moniker has the following parts:

- The prefix WinMgmts: (mandatory)
- A security settings component (optional)
- A WMI object path component (optional)

The following default assignments are allowed when specifying the object path:

- The computer machine name can be omitted from the object path, in which case the local machine name is assumed.
- The namespace can be omitted from the object path, in which case the default namespace is assumed. This is determined by the value of the registry key HKEY\_LOCAL\_MACHINE\Software\Microsoft\WBEM\Scripting\Default Namespace, the default value is root\\cimv2 for Windows XP and Windows 2000.
- All string literals are case-insensitive.

## Example Moniker Strings

1) The following moniker identifies the default namespace on the local computer

```
"WinMgmts:"
```

2) The following moniker identifies the default namespace on the computer myServer.

```
"WinMgmts://myServer"
```

3) The following moniker identifies the root\cimv2 namespace on the myServer computer.

```
"WinMgmts://myServer/root/cimv2"
```

4) The following moniker identifies the root\cimv2 namespace on the local server.

```
"WinMgmts:root/cimv2"
```

## Impersonation level

In the context of WMI, impersonation governs the degree to which your script will allow a remote WMI service to carry out tasks on your behalf. DCOM supports four levels of impersonation: Anonymous, Identify, Impersonate, and Delegate.

Name	Value	Description
<b>wbemImpersonationLevelAnonymous</b>	1	Hides the credentials of the caller. Calls to WMI may fail with this impersonation level.
<b>wbemImpersonationLevelIdentify</b>	2	Allows objects to query the credentials of the caller. Calls to WMI may fail with this impersonation level.
<b>wbemImpersonationLevelImpersonate</b>	3	Allows objects to use the credentials of the caller. This is the recommended impersonation level for WMI Scripting API calls.
<b>wbemImpersonationLevelDelegate</b>	4	Allows objects to permit other objects to use the credentials of the caller. This impersonation, which will work with WMI Scripting API calls but may constitute an unnecessary security risk, is supported only under Windows 2000.

You cannot use the Delegate impersonation level unless all the user accounts and computer accounts involved in the transaction have all been marked as Trusted for delegation in Active Directory. This helps minimize the security risks. Although a remote computer can use your credentials, it can do so only if both it and any other computers involved in the transaction are trusted for delegation.

As noted, Anonymous impersonation hides your credentials and Identify permits a remote object to query your credentials, but the remote object cannot impersonate your security context. (In other words, although the remote object knows who you are, it cannot "pretend" to be you.) WMI scripts accessing remote computers using one of these two settings will generally fail. In fact, most scripts run on the local computer using one of these two settings will also fail.

Impersonate permits the remote WMI service to use your security context to perform the requested operation. A remote WMI request that uses the Impersonate setting typically succeeds, provided your credentials have sufficient privileges to perform the intended operation. In other words, you cannot use WMI to perform an action (remotely or otherwise) that you do not have permission to perform outside

## Registry Path

HKEY\_LOCAL\_MACHINE\SOFTWARE\Microsoft\WBEM\Scripting\Default Impersonation Level

## Select

The statement you'll see most, and absolutely can't do without, is the Select statement. This is how you start a query, by saying that you want to Select something. The Select keyword is followed by what it is you want to select.

```
Set colItems = objWMIService.ExecQuery("Select * from Win32_PnPEntity")
```

We start with our Select keyword. What is it we want to select? In this case we want to select everything, which is represented with the wildcard character (\*). But we're not done there; after all, we can't really say we want to select everything. What exactly would "everything" be? Instead we need to add another keyword, the From keyword, and narrow the query down to everything from...where? Well, everything from the Win32\_PnPEntity class. Or, more accurately, all the properties associated with all the instances of the Win32\_PnPEntity class.

Suppose you wanted only the description of each device and not the manufacturer, name, and service? That seems simple enough; just don't echo the other properties:

```
strComputer = "."
Set objWMIService = GetObject("winmgmts:"
    & "{impersonationLevel=impersonate}!\\" & strComputer & "\root\cimv2")

Set colItems = objWMIService.ExecQuery("Select * from Win32_PnPEntity")

For Each objItem in colItems
    Wscript.Echo "Description: " & objItem.Description
    Wscript.Echo
Next
```

## Example

Win32\_OperatingSystem class (Windows)

The Win32\_OperatingSystem WMI class represents a Windows-based operating system installed on a computer. Any operating system that can be installed on a computer that can run a Windows-based operating system is a descendent or member of this class. Win32\_OperatingSystem is a singleton class.

```
class Win32_OperatingSystem : CIM_OperatingSystem
{
    stringBootDevice;
    stringBuildNumber;
    stringBuildType;
    stringCaption;
    stringCodeSet;
    stringCountryCode;
    stringCreationClassName;
    stringCSCreationClassName;
    stringCSDVersion;
    stringCSName;
```

sint16CurrentTimeZone;  
booleanDataExecutionPrevention\_Available;  
boolean DataExecutionPrevention\_32BitApplications;  
booleanDataExecutionPrevention\_Drivers;  
    uint8 DataExecutionPrevention\_SupportPolicy;  
boolean Debug;  
string Description;  
boolean Distributed;  
uint32EncryptionLevel;  
uint8ForegroundApplicationBoost;  
uint64FreePhysicalMemory;  
uint64FreeSpaceInPagingFiles;  
uint64FreeVirtualMemory;  
datetimeInstallDate;  
uint32LargeSystemCache;  
datetimeLastBootUpTime;  
datetimeLocalDateTime;  
string Locale;  
string Manufacturer;  
uint32MaxNumberOfProcesses;  
uint64MaxProcessMemorySize;  
stringMUILanguages[];  
string Name;  
uint32NumberOfLicensedUsers;  
uint32NumberOfProcesses;  
uint32NumberOfUsers;  
uint32OperatingSystemSKU;  
string Organization;  
stringOSArchitecture;  
uint32OSLanguage;  
uint32OSProductSuite;  
uint16OSType;  
stringOtherTypeDescription;  
Boolean PAEEnabled;  
stringPlusProductID;  
stringPlusVersionNumber;  
booleanPortableOperatingSystem;  
boolean Primary;  
uint32ProductType;  
stringRegisteredUser;  
stringSerialNumber;  
uint16ServicePackMajorVersion;  
uint16ServicePackMinorVersion;  
uint64SizeStoredInPagingFiles;  
string Status;  
uint32SuiteMask;  
stringSystemDevice;  
stringSystemDirectory;  
stringSystemDrive;

```

uint64TotalSwapSpaceSize;
uint64TotalVirtualMemorySize;
uint64TotalVisibleMemorySize;
string Version;
stringWindowsDirectory;
};

strComputer = "."
SetobjWMIService = GetObject("winmgmts:" _
&"{impersonationLevel=impersonate}!\" _
&strComputer&"\root\cimv2")
SetcolOperatingSystems = objWMIService.ExecQuery _
("Select * from Win32_OperatingSystem")

ForEachobjOperatingSystemincolOperatingSystems
Wscript.EchoobjOperatingSystem.ServicePackMajorVersion _
&"."&objOperatingSystem.ServicePackMinorVersion
Next

```

We can do following things with Win32\_operatingSystem WMI

- ...determine if a service pack has been installed on a computer?
- ...determine when the operating system was installed on a computer?
- ...determine which version of the Windows operating system is installed on a computer?
- ...determine which folder is the Windows folder (%Windir%) on a computer?
- ...determine what hotfixes have been installed on a computer?
- ...determine if I need to activate the operating system on a computer?
- ...activate a copy of Windows XP or Windows Server 2003?

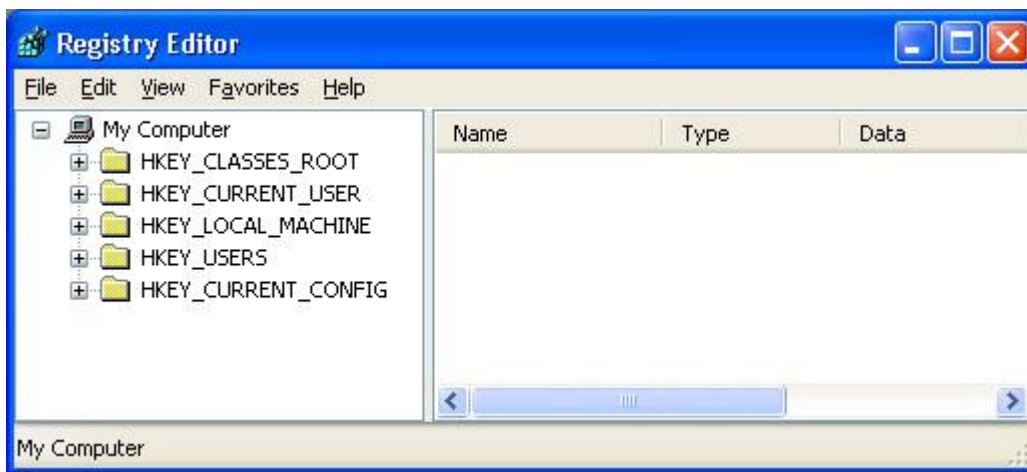
# WINDOWS REGISTRY

---

## A Little Background First

As you probably know, the registry is pretty much at the heart of Windows, storing information related to the operating system and the applications running on it. In a way, the registry does a lot of the staging for Windows.

Registry holds a lot more than just looks. It also stores information about the types of hardware you're running, your locale settings, application information (such as version numbers and install dates), and so on. All this is stored in a nice, convenient hierarchy that you can see by opening the Registry Editor, `regedit.exe`:



## Local Registry Access

Windows Script Host (WSH) provides a class, the `WshShell` class (created by calling `CreateObject` on `Wscript.Shell`), that gives you local access to the registry. `WshShell` has three methods that allow you to work with the registry: **RegRead**, **RegWrite**, and **RegDelete**

This script starts by creating the `WScript.Shell` object we already mentioned:

```
Set objShell = WScript.CreateObject("WScript.Shell")
```

The next thing we do is call the `RegRead` method on that object, passing it the full registry path to the value we want to look at:

```
iWordWrap = objShell.RegRead("HKCU\Software\Microsoft\Notepad\fWrap")
```

Our path started with "HKCU." As it turns out, HKCU stands for `HKEY_CURRENT_USER`. You can use the fully-spelled-out version, but WSH also allows you to use abbreviated versions of all the root keys:

Root Key	Abbreviation
HKEY_CLASSES_ROOT	HKCR
HKEY_CURRENT_USER	HKCU
HKEY_LOCAL_MACHINE	HKLM
HKEY_USERS	HKEY_USERS
HKEY_CURRENT_CONFIG	HKEY_CURRENT_CONFIG

```
Wscript.Echo iWordWrap
```

A value of 0 means word wrap is turned off; 1 means it's turned on. We can add a little bit of logic to our script to make that more obvious:

```
Set objShell = WScript.CreateObject("WScript.Shell")
iWordWrap = objShell.RegRead _
    ("HKCU\Software\Microsoft\Notepad\fWrap")
If iWordWrap = 0 Then
    Wscript.Echo "Word wrap is turned off"
Else
    Wscript.Echo "Word wrap is turned on"
End If
```

Suppose we now want to change that value - we want word wrap to always be turned on. Here's a script that writes a value to the registry:

```
Set objShell = WScript.CreateObject("WScript.Shell")
objShell.RegWrite "HKCU\Software\Microsoft\Notepad\fWrap", 1, "REG_DWORD"
```

We've passed three parameters to the RegWrite method:

- "HKCU\Software\Microsoft\Notepad\fWrap" - The path to the registry value we want to change. (Notice we again use the abbreviated version of the root key, HKCU.)
- 1 - The new value we're writing to the key value.
- "REG\_DWORD" - The data type of the value we're changing.

There are several data types used by the registry. Here's a list of the types you'll be working with most often:

Registry Data Type	VBScript Data Type
REG_SZ	String
REG_DWORD	Integer
REG_BINARY	Array of Integers (Binary Number)
REG_EXPAND_SZ	String (Expandable String)
REG_MULTI_SZ	Array of Strings



The last thing we're going to do with WSH on the local machine is to delete a key. (This is the part where you need to be careful. Delete the wrong key or value and you could have some problems you don't really want.)

Here's a script that deletes the new key value we created:

```
Set objShell = WScript.CreateObject("WScript.Shell")
objShell.RegDelete "HKCU\Software\Microsoft\Notepad\NewKey"
```

Once again we create a WshShell object, then we simply call the RegDelete method. We need to pass RegDelete only one parameter, the path to the key we want to delete.

## Remote Registry Access

We were able to use WSH to work with the local registry, which, as you saw, was pretty simple. To work remotely we need to use Windows Management Instrumentation (WMI). Keep in mind that you don't need separate scripts for local and remote access, you can use WMI for both. WMI is a little more complicated.

The StdRegProv class contains methods that manipulate system registry keys and values. StdRegProv is preinstalled in the WMI namespaces root\default and root\cimv2

You can use the methods to perform the following tasks:

- Verify the access permissions for a user.
- Create, enumerate, and delete registry keys.
- Create, enumerate, and delete named values.
- Read, write, and delete data values.

Unlike WSH, when you call methods in WMI to work with the registry you can't simply pass in an abbreviation for the root name, or even the full name of the root. Instead you have to specify a hexadecimal value that represents that root key. Here's a list of the root keys and their corresponding hex values:

Root Key	Hex Value
HKEY_CLASSES_ROOT	&H80000000
HKEY_CURRENT_USER	&H80000001
HKEY_LOCAL_MACHINE	&H80000002
HKEY_USERS	&H80000003
HKEY_CURRENT_CONFIG	&H80000005

## Create Registry Key

```
Const HKEY_CURRENT_USER = &H80000001
strComputer = "."
Set objRegistry = GetObject("winmgmts:\\." & strComputer &
"\root\default:StdRegProv")
strKeyPath = "SOFTWARE\Script Center"
objRegistry.CreateKey HKEY_CURRENT_USER, strKeyPath
```

## Create Registry Value

```
Const HKEY_CURRENT_USER = &H80000001
strComputer = "."
Set objRegistry = GetObject("winmgmts:\\." & strComputer &
"\root\default:StdRegProv")
strKeyPath = "SOFTWARE"
objRegistry.SetDWORDValue HKEY_CURRENT_USER, strKeyPath, "ms", "1"
```

Note :- Data Types available to Set / Create value

<b>SetBinaryValue</b>	Sets the binary data value of a named value.
<b>SetDWORDValue</b>	Sets the <b>DWORD</b> data value of a named value.
<b>SetExpandedStringValue</b>	Sets the expanded string data value of a named value.
<b>SetMultiStringValue</b>	Sets the multiple string values of a named value.
<b>SetQWORDValue</b>	Sets the QWORD data values of a named value.  <b>Windows Server 2003, Windows XP, Windows 2000, Windows NT 4.0, and Windows Me/98/95:</b> This method is not available.
<b>SetSecurityDescriptor</b>	Sets the security descriptor for a key.  <b>Windows Server 2003, Windows XP, Windows 2000, Windows NT 4.0, and Windows Me/98/95:</b> This method is not available.
<b>SetStringValue</b>	Sets the string value of a named value.

## Get / Display Registry Value

```
Const HKEY_CURRENT_USER = &H80000001

strComputer = "."

Set objRegistry = GetObject("winmgmts:\\." & strComputer &
"\root\default:StdRegProv")
strKeyPath = "SOFTWARE"
objRegistry.GetDWORDValue HKEY_CURRENT_USER, strKeyPath, "ms", value
wscript.echo value
```

<a href="#">GetBinaryValue</a>	Gets the binary data value of a named value.
<a href="#">GetDWORDValue</a>	Gets the <b>DWORD</b> data value of a named value.
<a href="#">GetExpandedStringValue</a>	Gets the expanded string data value of a named value.
<a href="#">GetMultiStringValue</a>	Gets the multiple string data values of a named value.
<a href="#">GetQWORDValue</a>	Gets the QWORD data values of a named value.  <b>Windows Server 2003, Windows XP, Windows 2000, Windows NT 4.0, and Windows Me/98/95:</b> This method is not available.
<a href="#">GetSecurityDescriptor</a>	Gets the security descriptor for a key.  <b>Windows Server 2003, Windows XP, Windows 2000, Windows NT 4.0, and Windows Me/98/95:</b> This method is not available.
<a href="#">GetStringValue</a>	Gets the string data value of a named value.

## Delete Registry Key

```

Const HKEY_CURRENT_USER = &H80000001

strComputer = "."

Set objRegistry = GetObject("winmgmts:\\." & strComputer &
"\root\default:StdRegProv")

strKeyPath = "SOFTWARE\Script Center"
objRegistry.DeleteKey HKEY_CURRENT_USER, strKeyPath

```

## Delete Registry Value

```

Const HKEY_CURRENT_USER = &H80000001

strComputer = "."

Set objRegistry = GetObject("winmgmts:\\." & strComputer &
"\root\default:StdRegProv")

strKeyPath = "SOFTWARE\Script Center"
objRegistry.Deletevalue HKEY_CURRENT_USER, strKeyPath, "ms"

```

## Simple Method

```
Windows Registry Editor Version 5.00
[HKEY_CLASSES_ROOT\lnkfile]
"test1"=-
```

- Note

The minus sign in: "test1"=-. There are no quotes around the (-). This deletes the value of test and test itself.

```
Windows Registry Editor Version 5.00
[HKEY_CLASSES_ROOT\lnkfile]
"test1"="" or "test1"="12"
```

- Note that "test1" now reads "test1"=". This re-creates the variable and sets it to blank.

# WINDOWS SERVICES

---

WMI tasks for services obtain information about services, including dependent or antecedent services. Use the Win32\_Service class to check the state of all of the services.

1) Determine which services are running and which ones are not?

```
strComputer = "."
SetObjWMIService = GetObject( _
"winmgmts:\\\"&strComputer&"\root\CIMV2")
SetcolItems = objWMIService.ExecQuery( _
"SELECT * FROM Win32_Service",,48)
ForEachobjItemincolItems
Wscript.Echo"Service Name: "&objItem.Name&VBNewLine _
&"State: "&objItem.State
Next
```

2) Start and stop services?

```
strComputer = "."
SetObjWMIService = GetObject("winmgmts:" _
&"{impersonationLevel=impersonate}!\" _
&strComputer&"\root\cimv2")
SetcolListOfServices = objWMIService.ExecQuery _
("Select * from Win32_Service Where Name ='Alerter'")
ForEachobjServiceincolListOfServices
objService.StartService()
Wscript.Echo"Started Alerter service"
Next
```

# PROCEDURES

---

Procedures are useful for scripts that need to carry out the same task over and over. For example, you might want to echo a message each time an error occurs within a script. To do this, you can include the message display code at every point in the script where an error might occur. Of course, this requires extra work, not only to write the original code but to maintain it as well; if you ever decide to change the displayed message, you will have to locate and change each instance within the code.

A better approach is to create a procedure that displays the message and then call that procedure each time an error occurs. As a result, you have to write and maintain only a single instance of the display code. VBScript allows you to create two types of procedures:

**Subroutines:** -Code that is run but typically does not return a value. For example, you might create a subroutine to display error messages.

**Functions:** -Code that is run and returns a value. Functions are often used to carry out mathematical equations. For example, you might pass free disk space to a function, and the function will, in turn, convert the free space from bytes to gigabytes and then return that value.

Calling a Procedure

## Syntax of define any procedure

```
Sub Procedure_name
    'Procedure definition.
End Sub
```

## Syntax of calling any procedure

```
Procedure_name
```

In general, VBScript processes each line of code in succession. Procedures are an exception to this rule. Neither

Subroutines nor functions are run unless they have been specifically invoked somewhere in the script. If a procedure has not been invoked, the procedure is simply skipped, regardless of where it appears within the script.

For example, the following script includes a subroutine embedded in the middle of the code. However, this subroutine is never actually called.

```
Wscript.Echo "A"
Sub EchoLine2
Wscript.Echo "B"
End Sub
Wscript.Echo "C"
```

When the preceding script runs under CScript, the following output appears in the command window. Because the subroutine was never called, the subroutine and all the code inside it was skipped and not run:

A  
C

To ensure that a subroutine runs, it must be called. This is done using a statement that consists solely of the subroutine name. For example, the following script echoes the message "A", calls the subroutine named EchoLineB, and then echoes the message "C".

```
Wscript.Echo "A"
```

```
EchoLineB  
Wscript.Echo "C"  
Wscript.Quit
```

```
Sub EchoLineB  
Wscript.Echo "B"  
End Sub
```

When the preceding script runs under CScript, the following output appears in the command window:

A  
B  
C

Procedures can be placed anywhere within a script, with no degradation of performance. However, the placement of procedures can affect the ease with which a script can be read and maintained

# FUNCTIONS

---

Like subroutines, functions provide a way for you to use one section of code multiple times within a script. Unlike subroutines, however, functions are designed to return a value of some kind. This is not necessarily a hard-and-fast rule.

In fact, when you create a function, VBScript automatically declares and initializes a variable that has the same name as the function. This variable is designed to hold the value derived by the function. Although there is no requirement that you use this variable, doing so makes it very clear that the value in question was derived by the function of the same name.

For example, the following script includes the statement `Wscript.EchoThisDate`. `ThisDate` also happens to be the name of a function that retrieves the current date. In this script, notice that: The `Wscript.Echo` statement actually performs two tasks.

- 1) First it calls the function `ThisDate`. The function, in turn, sets the value of the special variable `ThisDate` to the current date.
- 2) After the function has completed, `Wscript.Echo` then echoes the value of this special variable.

Option `Explicit` is used, and the variable `ThisDate` is never declared. However, no error occurs because VBScript internally declares and initializes this function variable for you.

```
Option Explicit
Wscript.EchoThisDate
Function ThisDate
ThisDate = Date
End Function
```

Note that this approach works only for a function, and not for a subroutine. The following code generates a run-time error because VBScript is unable to assign the date to the name of a subroutine:

```
Wscript.EchoThisDate
Sub ThisDate
ThisDate = Date
End Sub
```

## Passing Parameters to Functions

Functions are often used to carry out a mathematical equation and then return the result of this equation. For example, you might use a function to convert bytes to megabytes or convert pounds to kilograms.

For a function to carry out a mathematical equation, you must supply the function with the appropriate numbers. For example, if you want a function to add the numbers 1 and 2, you must supply the functions with those two values. The numbers 1 and 2 are known as parameters (or arguments), and the process of supplying a function with parameters is typically referred to as passing those values. To pass parameters to a function, simply include those values in the function call. For example, this line



of code calls the function AddTwoNumbers, passing the values 1 and 2:

```
AddTwoNumbers(1 , 2)
```

In addition to including the parameters within the function call, the function itself must make allowances for those parameters. This is done by including the appropriate number of variables in the Function statement. This line of code, for example, creates a function that accepts three parameters:

```
Function AddThreeNumbers(x, y, z)
```

If the number of parameters in the Function call does not match the number of parameters in the Function statement, an error will occur. For example, this script generates a "Wrong number of arguments" error. Why? Because two values are passed to the function, but the Function statement does not allow for any parameters:

```
x = 5
y = 10
Wscript.EchoAddTwoNumbers(x, y)
Function AddTwoNumbers
AddTwoNumbers = a + b
End Function
```

To correct this problem, include space for two parameters within the Function statement:

```
x = 5
y = 10
Wscript.EchoAddTwoNumbers(x, y)
Function AddTwoNumbers(a, b)
AddTwoNumbers = a + b
End Function
```

You might have noticed that the parameters used in the Function call (x and y) have different names from the parameters used in the Function statement (a and b). VBScript does not require the parameter names to be identical; if it did, this would limit your ability to call the function from multiple points within a script. (Although this would be possible, you would always have to assign new values to variables x and y before calling the function. This could be a problem if you did not want to assign new values to x and y.)

Instead, VBScript simply relies on the order of the parameters. Because x is the first parameter in the function call, the value of x is assigned to a, the first parameter in the Function statement. Likewise, the value of y, the second parameter in the Function call, is assigned to b, the second parameter in the Function statement.

# ERROR HANDLING

---

VBScript error-handling requires two elements that work together. You can turn it on with the "On Error Resume Next" statement and turn it off with "On Error GoTo 0". When it's turned on you can use the built-in Err object to get some information on what kind of error occurred.

## Syntax

`On Error resume next` - Enable error handling

`On Error goto 0` - Disable error handling

The meaning of the first seems clear -- if you get an error, ignore it and resume execution on the next statement.

Using the On Error GoTo 0 statement to turn error handling off. After this statement has been run, error handling will not take place until another On Error Resume Next statement has been encountered.

Before you can check for an error, you have to include the statement On Error Resume Next. If you check the Err object without first turning on error handling with On Error Resume Next, VBScript assumes that Err.Number is 0; in other words, that no error has occurred. The script will then continue to do whatever comes next, assuming that all is well. If an error has in fact occurred, it may cause the script to fail with an unhandled run-time error that brings everything grinding to a halt.

Putting On Error Resume Next at the beginning of the script, as we often do, makes it apply to the entire body of the script. But, as we'll see in later examples, its scope does not include functions or subroutines. If you want to handle errors within a function or subroutine, you must also include On Error Resume Next in each of them before checking the Err object.

You can turn error-handling off with On Error GoTo 0. So it's possible to turn error-handling on with On Error Resume Next just before you want to check the Err object, and turn it off after with On Error GoTo 0. This makes more explicit exactly where errors are being handled, but to the jaded eyes of the Scripting Guys it seems like a lot of work for minimal returns in most cases.

## Ignoring All Errors

By far the simplest form of error handling is the type that instructs a script to ignore all errors and continue running until every line of code has been processed. To create a script that ignores all errors and continues to run, place the On Error Resume Next statement at the beginning of the script.

### `On Error Resume Next`

```
Wscript.Echo "Line 1."  
Wscript.Echo "Line 2."  
Wscript.Echo "Line 3."  
Wscript.Echo "Line 4."
```

## Responding to Errors

Instead of having your script ignore errors, you can create code that periodically checks the error condition and then takes some sort of action.

Err has three properties that are generally useful:

Number (the default property) - integer  
Source - string  
Description - string

Err also provides two methods:

Clear  
Raise(IngNumber, strSource, strDescription)

Clear takes no parameters. It simply clears the values of all the properties of the previous error. It's very important to use Clear after each time you check Err. Otherwise, the information from the previous error will persist in the Err object and if you check again but no intervening error has occurred, the same error information will still be there and you may get a false positive on your error check.

With the Raise method, VBScript offers a little-known capability: you can use this method to create a VBScript error in one part of the script if something untoward occurs that would not cause the script engine to raise an error. Only the error number, lngNumber, is required; the other parameters are optional. The error number variable is called lngNumber here because user-defined VBScript errors (as well as VBScript-defined ones) are in the range 0 to 65535 (decimal). We've never used this capability ourselves, but it could come in handy if you have a working scripting library or application that doesn't offer thorough error-handling mechanisms.

```
On error Resume Next
strComputer = "01hw123"

Set objWMIService = GetObject("winmgmts:" _
    & "{impersonationLevel=impersonate}!\\" & strComputer & "\root\cimv2")

If Err.Number <> 0 Then
    'error handling:
    WScript.Echo "Error No: " & Err.Number & vbcrLf & "Source: " & Err.Source &
vbcrLf & "Desc: " & Err.Description
    Err.Clear
End If

Set colItems = objWMIService.ExecQuery("Select ScreenWidth,ScreenHeight From
Win32_DesktopMonitor where DeviceID = 'DesktopMonitor1'",,0)
    For Each objItem in colItems
        intHorizontal = objItem.ScreenWidth
        intVertical = objItem.ScreenHeight
    Next

If IsNull(intHorizontal) Then
WScript.Echo "No Values set in"
Else
WScript.Echo intHorizontal
WScript.Echo intVertical

End if
```

## Other Ways of Testing for Successful Connection to an Object

### Is Nothing

You can use the Is operator to compare an object with the Nothing keyword. If the object has not been instantiated, it Is Nothing. Is compares an object reference with another object reference or a keyword that can refer to an object to see if they are the same. Nothing is the equivalent of Null for an object reference.

```
On Error Resume Next
strComputer = "fictional"
Set objWMIService = GetObject("winmgmts:\\\" & strComputer & "\root\cimv2")
If objWMIService Is Nothing Then
    WScript.Echo "Unable to bind to WMI on " & strComputer
Else
    WScript.Echo "Successfully bound to WMI on " & strComputer
End If
```

### IsObject

Another technique similar to Is Nothing is the IsObject function, which is built into VBScript. IsObject also works with an object reference, verifying whether or not it is an object. In this case there's no comparison: IsObject is true if objPrinter refers to a valid object, and false if not.

```
On Error Resume Next
strPrinter = "TestPrinter"
Set objPrinter = GetObject _
    ("winmgmts:root\cimv2:Win32_Printer.Name=\"" & strPrinter & "\"")
If IsObject(objPrinter) Then
    WScript.Echo "Connected to printer " & strPrinter
Else
    WScript.Echo "Unable to connect to printer " & strPrinter
End If
```

### Advantages

- Scripts will continue to run without interruption.
- Scripts can complete a major share of their tasks, even if a problem arises. For example, a script might be able to successfully copy 99 of 100 files, even if one file could not be copied.
- End users will not be presented with error messages that they must respond to.

### Disadvantages

- Difficult to debug problems because no message is displayed indicating that an error occurred, nor is any clue given as to where the error might have taken place.
- Can leave a computer in an uncertain state, wherein some actions have been carried out while others have not.