# Scripting Excel

Drives, files and folders are the lifeblood of any organization; this makes file system administration one of the most important responsibilities assigned to system administrators. Of course, file system administration is also one of the more difficult responsibilities to carry out, simply because files and folders are scattered on multiple hard disks and multiple computers throughout the organization. Scripts can help make file system management much easier, particularly when the files and folders managed, are located on remote computers.

## Creating and Terminating an Instance of MS-Excel

Let's start with the simplest possible script, one that creates an instance of Microsoft Excel and then adds a new workbook to that instance:

```
Set oXlsApp = CreateObject("Excel.Application")
oXlsAp.Workbooks.Add
```

By running the preceding script, you really did create a brand-new instance of Microsoft Excel. Press CTRL-ALT-DEL and take a look at the Processes tab in the Task Manager. You should see an instance of **Excel.exe**

By default, any time you use a script to create an instance of a Microsoft Office application, that application runs in a window that is not visible on screen. Excel is there; you just can't see it

This is a real, live instance of Microsoft Excel. As you'll soon see, you can programmatically read data from it or, for that matter, do pretty much anything else you can do with Excel.

The only functionality you lose when Excel runs in an invisible window is the ability to type something on the keyboard and have the application reacts to those keystrokes. And that's what makes the default behavior useful.

Suppose you were running a script that created a report using Excel, and suppose Excel was visible the whole time the script was running. A user (even yourself) could accidentally hit a key on the keyboard and ruin the entire report. A user (even yourself) could simply close Excel, ruining not only the report, but also causing your script to blow up. (After all, the script will be trying to send commands to an instance of Excel that no longer exists.) By running Excel invisibly, you can sidestep problems like that.

What if you would like Excel to be visible on screen? No problem just set the Visible property to True.

```
Set oXlsApp = CreateObject("Excel.Application")
oXlsAp.Workbooks.Add
oXlsApp.Visible = True
```

```
Wait 10
MsgBox "The script is now complete."
```

What happens when you run this script? Well, an instance of Excel will be created, and it will appear on your screen. There will be a 10-second pause, and then a message will appear telling you that the script is now complete. When you click OK, the script will immediately terminate (as soon as Microsoft® VBScript reaches the end of a script, the script process terminates).

## Returning a Collection of Disk Drives

Before you can manage disk drives on a computer, you need to know which disk drives are actually available on that computer. The **FileSystemObject** allows you to return a collection of all the drives installed on a computer, including removable drives and mapped network drives (in other words, any drive with a drive letter).

To return this collection, create an instance of the **FileSystemObject**, and then create a reference to the **Drives** property. After the collection has been returned, you can use a **For Each** loop to iterate through the collection.

For example, the following script returns a collection of all the drives installed on a computer and then echoes the drive letter for each drive in the collection.

```
Set oFSO = CreateObject("Scripting.FileSystemObject")
Set colDrives = oFSO.Drives
For Each oDrive in oDrivesCol
   MsgBox "Drive letter: " & oDrive.DriveLetter
Next
```

## Binding to a Specific Disk Drive

If you know in advance which drive you want to bind to (for example, drive C, or the shared folder \\accounting\receivables), you can use the **GetDrive** method to bind directly to the drive. This allows you to retrieve information for a specific drive, without having to return and iterate through an entire collection.

The **GetDrive** method requires a single parameter: the driver letter of the drive or the **UNC** path to the shared folder. To specify a drive letter, you can use any of the following formats:

- C
- C:
- C:\

The following script creates an instance of the **FileSystemObject**, uses the **GetDrive** method to bind directly to drive C, and then echoes the amount of available space on the drive.

```
Set oFSO = CreateObject("Scripting.FileSystemObject")
Set oDrive = oFSO.GetDrive("C:")
MsgBox "Available space: " & oDrive.AvailableSpace
```

## Enumerating Disk Drive Properties

The **Drives** collection is typically used for inventory or monitoring purposes; as a system administrator, you need to know what drives are available on a computer, as well as details such as the drive serial number and the amount of free space on the drive. After you have returned a drives collection or an individual drive object, you can retrieve any of his properties.

To enumerate the drives installed on a computer, create an instance of the **FileSystemObject**, create a reference to the **Drives** property, and then use a **For Each** loop to iterate through the set of drives. For each drive in the collection, you can echo any or all of the individual drive object properties

```
Set oFSO = CreateObject("Scripting.FileSystemObject")
Set oDrivesCol = oFSO.Drives
For Each oDrive in oDrivesCol
   sOut = sOut & "Available space: " & oDrive.AvailableSpace & vbCrLf
   sOut = sOut & "Drive letter: " & oDrive.DriveLetter & vbCrLf
   sOut = sOut & "Drive type: " & oDrive.DriveType & vbCrLf
   sOut = sOut & "File system: " & oDrive.FileSystem & vbCrLf
   sOut = sOut & "Free Space: " & oDrive.FreeSpace & vbCrLf
   sOut = sOut & "Is ready: " & oDrive.IsReady & vbCrLf
   sOut = sOut & "Path: " & oDrive.Path & vbCrLf
   sOut = sOut & "Root folder: " & oDrive.RootFolder & vbCrLf
   sOut = sOut & "Serial number: " & oDrive.SerialNumber & vbCrLf
   sOut = sOut & "Share name: " & oDrive.ShareName & vbCrLf
   sOut = sOut & "Total size: " & oDrive.TotalSize & vbCrLf
   sOut = sOut & "Volume name: " & oDrive.VolumeName & vbCrLf
Next
```

```
Available space: 10234975744
Drive letter: C
Drive type: 2
File system: NTFS
Free space: 10234975744
Is ready: True
Path: C:
Root folder: C:\
Serial number: 1343555846
Share name:
Total size: 20398661632
Volume name: Hard Drive
```

## Ensuring That a Drive is Ready

The previous script has a potential flaw in it: If there is no floppy disk in the floppy disk drive that is being checked or no CD in the CD-ROM drive, the script will fail with a "Drive not ready" error. **Drives** that are not ready create problems for scripts that use the **FileSystemObject**; although the **FileSystemObject** can identify the existence of those drives, your script will fail if it attempts to access disk drive properties such as **AvailableSpace** or **FreeSpace**.

If a drive is not ready (which typically means that a disk has not been inserted into a drive that uses removable disks), you can retrieve only the following four drive properties:

- DriveLetter, DriveType, IsReady, ShareName

Any attempt to retrieve the properties of another drive will trigger an error. Fortunately, the **IsReady** property allows the script to check whether a drive is ready before attempting to retrieve any of the properties that can trigger an error. The **IsReady** property returns a **Boolean** value; if the value is **True**, the drive is ready, and you can retrieve all the properties of the drive. If the value is **False**, the drive is not ready, and you can return only **DriveLetter**, **DriveType**, **IsReady**, and **ShareName**.

The following script returns a collection of disk drives installed on a computer. For each drive, the script uses the **IsReady** property to ensure that the drive is ready. If it is, the script echoes the drive letter and the amount of free space. If the drive is not ready, the script echoes only the drive letter, one of the four properties that can be accessed even if a drive is not ready.

```
Set oFSO = CreateObject("Scripting.FileSystemObject")
Set oDrivesCol = oFSO.Drives
For Each oDrive in oDrivesCol
   If oDrive.IsReady = True Then
      sOut = sOut & "Drive letter: " & oDrive.DriveLetter & vbCrLf
      sOut = sOut & "Free Space: " & oDrive.FreeSpace & vbCrLf
   Else
      sOut = sOut & "Drive letter: " & oDrive.DriveLetter & vbCrLf
   End If
Next
```

This problem does not occur with **WMI**. If there is no disk in drive **A** or no **CD** in the **CD-ROM** drive, the script does not fail. Instead, **WMI** simply reports the amount of free space as **Null**.

## Managing Folders

**Disk** drive properties such as **FreeSpace** and **TotalSize** provide global information that is important. However, disk drive information is necessary, but not sufficient, for managing a file system. Although it is important to know which drive a file is stored on, you also need to know the folder in which that file is stored. In addition, many other system management tasks take place at the folder level: Folders are copied, folders are moved, folders are deleted, folder contents are enumerated.

The **FileSystemObject** can return detailed information about the folders on a disk drive. In addition, the **FileSystemObject** provides a number of methods for carrying out such tasks as copying, moving, and deleting folders, and for enumerating the files and subfolders within a folder.

## Binding Folders

In the **Windows Shell**, folders are **C**omponent **O**bject **M**odel (**COM**) objects. This means that, before you can access the properties of an individual folder, you must create an object reference to that folder, a process commonly referred to as binding. You can bind to a folder by creating an instance of the **FileSystemObject** and then using the **GetFolder** method to connect to the folder.

When using the **GetFolder** method, you must:

- **Specify the path name to the folder.** The path can be referenced by either a local path or a **UNC** path (i.e \\accounting\receivables). However, you cannot use wildcards within the path name. In addition, you cannot create a single object reference that binds to multiple folders at the same time. If you need to work with multiple folders, you either need to use **WMI** (which can return a collection of folders) or create a separate object reference for each folder.
- **Use the Set keyword when assigning the path to a variable.** The **Set** keyword is required because it indicates that the variable in question is an object reference.

Although wildcard characters are not allowed, you can use the dot (.) to bind to the current folder, dot-dot (..) to bind to the parent folder of the current folder, and the backslash (\) to bind to the root folder. For example, the following code statement binds to the current folder:

```
Set oFolder = oFSO.GetFolder(".")
```

## Verifying That a Folder Exists

Most folder operations, including copying, moving, and deleting, require the specified folder to exist before the operation can be carried out; after all, a script cannot copy, move, or delete a folder that does not exist. If the script attempts to bind to a nonexistent folder, the script will fail with a "Path not found" error.

To avoid this problem, you can use the **FolderExists** method to verify that a folder exists before attempting to bind to it. **FolderExists** takes a single parameter (the path name to the folder) and returns a **Boolean** value: **True** if the folder exists, **False** if the folder does not.

## Creating a Folder

It is unlikely that you will ever sit down, implement your file system infrastructure (that is, your folders and subfolders), and then never have to touch that infrastructure again. Instead, a file system tends to be dynamic: because of ever-changing needs, existing folders might be deleted and new folders might be created. For example, if your organization provides users with storage space on file servers, you need to create a new folder each time a new user account is created.

The **FileSystemObject** gives script writers the ability to programmatically create folders, a capability that can make your scripts even more powerful and more useful. For example, the following script checks to see whether a specified folder exists. If the folder exists, the script uses the **GetFolder** method to bind to the folder. If the folder does not exist, the script echoes a message to that effect.

```
Set oFSO = CreateObject("Scripting.FileSystemObject")
If oFSO.FolderExists("C:\FSO") Then
   Set oFolder = oFSO.GetFolder("C:\FSO")
   MsgBox "Folder binding complete."
Else
   MsgBox "Folder does not exist?"
End If
```

Although this approach prevents the script from crashing, you might prefer that your script create the folder rather than simply report that the folder does not exist. To do this, create an instance of the **FileSystemObject**, and then call the **CreateFolder** method, passing the complete path to the new folder as the sole parameter. For example, the following script creates a new folder named C:\FSO.

```
Set oFSO = CreateObject("Scripting.FileSystemObject")
Set oFolder = oFSO.CreateFolder("C:\FSO")
```

If the folder already exists, a "File exists" error will occur. Because of that, you might want to check for the existence of the folder before trying to create (or, in that case, re-create) it.

**Note**

The **FileSystemObject** can only create folders on the local computer. If you need to create folders on a remote computer, you will need to use the **WshController** object. Alternatively, you can create a folder locally and then use **WMI** to move that folder to the remote computer. (The folder must be created and then moved because **WMI** does not have a method for creating folders.)

## Deleting a Folder

From time to time, folders need to be deleted. For example, you might have a file server that includes a folder for each individual user. When a user leaves the organization, the folder belonging to that user should be deleted; this helps ensure that the orphaned folder does not use up valuable disk space. Likewise, you might have a script that stores temporary files within a folder. Before the script finishes, you might want to delete that folder and thus remove all the temporary files.

The **DeleteFolder** method provides a way to delete a folder and all its contents. The **DeleteFolder** method requires a single parameter: the path of the folder to be deleted. For example, the following code deletes the folder C:\FSO and everything in it.

```
Set oFSO = CreateObject("Scripting.FileSystemObject")
oFSO.DeleteFolder("C:\FSO")
```

The **DeleteFolder** method deletes all items immediately; it does not ask for confirmation of any kind or send the items to the Recycle Bin.

### Using Wildcards to Delete Folders

One of the main advantages of using scripts as a management tool is that scripts can operate on multiple items at the same time. For example, rather than delete a series of folders one by one, you can use scripts to delete a set of folders in a single operation.

The **FileSystemObject** allows you to use wildcard characters to delete a specific set of folders. For example, suppose you have the folder structure shown in

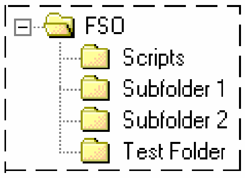


**Figure 1** and you want to delete all the subfolders beginning with the letter S.

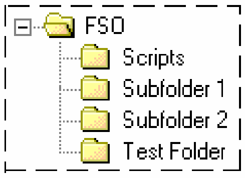


**Figure 1 – Sample Folder Structure**

This can be done by using the following command; when run against the sample folder structure, the command deletes the folders Scripts, Subfolder1, and Subfolder2:

```
oFSO.DeleteFolder("C:\FSO\S*")
```

This command deletes all the subfolders beginning with the letters Su, meaning only Subfolder1 and Subfolder2 will be deleted:

```
oFSO.DeleteFolder("C:\FSO\Su*")
```

Wildcard characters can appear only in the final part of the path parameter. For example, this command, which features a wildcard character in the middle of the path parameter, generates a "Path not found" error:

```
oFSO.DeleteFolder("C:\*\Subfolder1")
```

## Copying a Folder and Its Contents

The ability to copy a folder, and every item contained within that folder, is important in system administration. Sometimes you need to copy folders in order to create backups; by having the same folder on Computer A that you have on Computer B, you are less likely to experience data loss should Computer B unexpectedly fail. At other times, you might want to deploy all the files contained in a particular folder to a large number of computers. Using a script to copy this folder to each computer is far more efficient than performing the task manually.

The **CopyFolder** method allows you to copy a folder and its contents to another location. When used without any wildcard characters, the **CopyFolder** method functions like the **Xcopy /E** command: It copies all the files and all the subfolders, including any empty subfolders. The **CopyFolder** method requires two parameters:

- **Source folder** (the folder being copied). This folder can be specified either as a local path (C:\Scripts) or as a UNC path (\\helpdesk\scripts).
- **Destination folder** (the folder that will hold the copied information). This folder can also be specified either as a local path or as a UNC path. If the destination folder does not exist, the script automatically creates the folder.

In addition, the **CopyFolder** method accepts an optional third parameter, *Overwrite*. When this parameter is set to **True**, the default setting, the script overwrites any existing folders in the destination folder. For example, if you are copying a folder named Scripts, and the destination already contains a folder by that name, the destination folder will be replaced by the newly copied information. By setting this parameter to **False**, the script will not overwrite existing information and instead generates a run-time error.

**Note**

The **CopyFolder** method stops the moment it encounters an error, even if the script contains an **On Error Resume Next** statement. For example, suppose the script has 100 subfolders to copy, and **CopyFolder** successfully copies 3 of those subfolders before encountering an error. At that point, the **CopyFolder** method ends and the script fails; the script will not even attempt to copy the remaining 97 subfolders.

**Note**

Because **CopyFolder** is a single operation, there is no way to track its progress; you simply have to wait until the operation has finished. If you want to monitor the progress of the copy command, you should use the **Shell Application** object instead. This object is discussed in "Chapter 11 – Shell32" in this book.

### Using Wildcards to Copy Folders

The **CopyFolder** command copies the files stored in a folder as well as the files stored in any subfolders of that folder. This can be a problem; after all, what if you want to copy only the files in C:\FSO and not all the files stored in C:\FSO\Subfolder1, C:\FSO\Subfolder2, and C:\FSO\Subfolder3?

Unfortunately, there is no straightforward method for copying the files in a parent folder without also copying the files stored in child folders. You can use wildcard characters to limit the set of subfolders that are copied; for example, the following command copies only those folders that start with the letters log. However, when you use wildcard characters, no files other than those in the specified folders will be copied, not even files that begin with the letters log:

```
oFSO.CopyFolder "C:\Scripts\Log*",  "C:\Archive",  True
```

When the preceding line of code is run, the folders C:\Scripts\Logs and C:\Scripts\Logfiles are copied, along with all the files stored within those folders. However, the files within the C:\Scripts folder are not copied.

When you use the **CopyFolder** method, you cannot copy only the files in a folder without also copying the files in any subfolders. To copy only the files and not the subfolders, use the **CopyFile** method instead.

## Moving a Folder and Its Contents

When you copy a folder from one location to another, you end up with duplicate copies of the information. Sometimes that is exactly what you want. On other occasions, however, you do not want two copies of the information; instead, you want to move the sole copy from Computer A to Computer B, or from hard disk C to hard disk D.

Moves such as this are often done to free disk space on a particular drive; for example, you might periodically move seldom-accessed folders to an archive drive. Alternatively, you might have a monitoring script that logs information to the local computer. When monitoring is complete, you might want that information uploaded to a central monitoring station and then deleted from the local computer. That way, the local computer will be prepared for the next round of monitoring.

The **MoveFolder** method accepts two parameters:

- **Source folder** (the folder to be moved). This folder can be specified either as a local path or as a **UNC** path.
- **Destination folder** (the location where the folder is to be moved). This folder can be specified either as a local path or as a **UNC** path.

If the destination folder does not exist, the source folder will be moved. If the destination folder already exists, however, the move operation will fail. You cannot use **MoveFolder** to overwrite an existing folder.

**Note**

**MoveFolder** method cannot perform any sort of rollback should the script fail. For example, suppose a network connection fails before a script has been able to move all the files from one computer to another. In a case such as that, you will end up with some files on Computer A, some files on Computer B, and possibly even a file or two lost in transit. However, there is no way for **MoveFolder** to roll back the failed transactions and restore the two computers to their previous states.

Because of that, you might want to use two methods, **CopyFolder** and **DeleteFolder**, when transferring folders and their contents across the network. You can use **CopyFolder** to copy the folder from Computer A to Computer B. If the copy operation succeeds, you can then use **DeleteFolder** to delete the folder on Computer A. If the operation fails, you can cancel the delete command and rest assured that the folder and all its contents are still safely stored on Computer A.

## Renaming a Folder

The **FileSystemObject** does not include a method, such as **RenameFolder**, that provides an obvious way to rename a folder. However, you can rename a folder by using the **MoveFolder** method and maintaining the same relative location. For example, suppose you have a folder with the following path:

```
C:\Scripts\PerformanceMonitoring\Servers\Domain Controllers\Current Logs
```

If you rename the folder by using the Rename command in Windows Explorer, the path remains identical except for the endpoint, the folder itself:

C:\Scripts\PerformanceMonitoring\Servers\Domain Controllers\Archived Logs

The **MoveFolder** method enables you to achieve the same end result by moving the folder from C:\Scripts\PerformanceMonitoring\Servers\Domain Controllers\Current Logs to C:\Scripts\PerformanceMonitoring\Servers\Domain Controllers\Archived Logs. The net result is exactly the same as that of using Windows Explorer to rename the folder.

For example, the following script uses **MoveFolder** to rename the folder C:\FSO\Samples to C:\FSO\Scripts. Before the script runs, Samples is the only subfolder in C:\FSO. After the script runs, Scripts is the only subfolder in C:\FSO.

Furthermore, Scripts contains all the files and subfolders previously contained in Samples.

```
Set oFSO = CreateObject("Scripting.FileSystemObject")
oFSO.MoveFolder "C:\FSO\Samples" , "C:\FSO\Scripts"
```

## Enumerating Folder Properties

Because folders are **COM** objects, they have properties that can be retrieved and enumerated. To retrieve detailed information about a specified folder, you can use the **Folder** object, one of the components of the **FileSystemObject**.

To retrieve the properties of a folder, a script must:

1. Create an instance of the **FileSystemObject**.
2. Use the **GetFolder** method to bind to an individual folder.
3. Echo (or manipulate) the properties shown in Folder Object on page 83

When working with folder properties, note that the **Files** property and the **Subfolders** property both return collections rather than a single item. In addition, the **Attributes** property is returned as a bitmap value.

The following code uses the **GetFolder** method to bind to the folder C:\FSO and then echoes a number of properties for that folder

```
Dim oFSO, oFolder, sOut
Set oFSO = CreateObject("Scripting.FileSystemObject")
Set oFolder = oFSO.GetFolder("C:\FSO")
sOut = sOut &  "Date created: " & oFolder.DateCreated & vbCrLf
sOut = sOut &  "Date last accessed: " & oFolder.DateLastAccessed & vbCrLf
sOut = sOut &  "Date last modified: " & oFolder.DateLastModified & vbCrLf
sOut = sOut &  "Drive: " & oFolder.Drive & vbCrLf
sOut = sOut &  "Is root folder: " & oFolder.IsRootFolder & vbCrLf
sOut = sOut &  "Name: " & oFolder.Name & vbCrLf
sOut = sOut &  "Parent folder: " & oFolder.ParentFolder & vbCrLf
sOut = sOut &  "Path: " & oFolder.Path & vbCrLf
sOut = sOut &  "Short name: " & oFolder.ShortName & vbCrLf
sOut = sOut &  "Short path: " & oFolder.ShortPath & vbCrLf
sOut = sOut &  "Size: " & oFolder.Size & vbCrLf
sOut = sOut &  "Type: " & oFolder.Type & vbCrLf
MsgBox sOut
Set oFSO = Nothing
```

```
Date created: 2/7/2002 10:27:50 AM
Date last accessed: 2/13/2002 8:57:18 AM
Date last modified: 2/13/2002 8:57:18 AM
Drive: C:
Is root folder: False
Name: FSO
Parent folder: C:\
Path: C:\FSO
Short name: FSO
Short path: C:\FSO
Size: 0
Type: File Folder
```

## Managing Folder Attributes

File systems typically support the concept of attributes, information about a file or folder that goes beyond the folder name and size. For example, if you right-click a folder in Windows Explorer and then click Properties, you can access the attributes for that folder.

The **FileSystemObject** can be used to return several important attributes of a folder. These attributes are the - DriveType Constants in Table 4 on page 111
The values listed are the only values that can be retrieved or configured by using the **FileSystemObject**. Although this seems simple enough, the data returned to you by the **FileSystemObject** can be confusing at first. For example, if you echo the value of the **Attributes** property for a folder, you might see a value like 20, a value that does not appear in the list of valid attribute values.

In addition, you will receive only a single value, even if a folder has all possible attributes (that is, it is a hidden, compressed system folder ready for archiving). In a case such as this, your script will not display the values 2, 4, 16, 32, and 2048 but instead will display the value 2102. This is because attribute values are always returned in the form of a bitmap.

**Note**

With attributes, the term bitmap refers to the way data is stored and returned. It should not be confused with bitmap images, such as .BMP files.

### Working with Bitmaps

A bitmap is like a set of switches that can be either on or off. If a particular switch is off, that switch has the value 0. If the switch is on, at least in the case of a folder object, it has one of the values shown in the attributes The value of the bitmap is equal to the sum of all the switches.

For example, a highly simplified illustration of a folder object bitmap is shown in Figure 2. In this example, only one individual switch, Directory, is on. Directory has the value 16. Because the other switches are off, each has the value 0. The total value for the bitmap is thus 16. If you queried the Attributes value for this folder, the script would return 16.

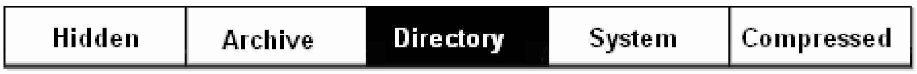


**Figure 2 First Sample Bitmap Representation**

By comparison, the folder object shown in Figure 3 has three switches activated: Hidden (with the value 2), Directory (with the value 16), and Compressed (with the value 2048). The value for this bitmap would thus be 2 + 16 + 2048, or 2066. This is also the value that would be returned by a script querying this folder for its **Attributes** value.

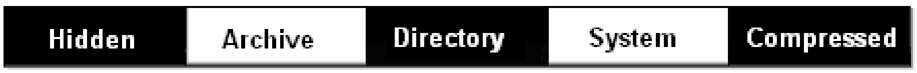


**Figure 3 Second Sample Bitmap Representation**

Bitmaps are designed so that there is only one possible way to achieve a given value. The only way for a folder attribute to return the value 2066 is for it to be a hidden and compressed folder. It is mathematically impossible to return a 2066 with any other combination of attributes.

This design enables you to take the return value and determine which switches have been set and which ones have not; in turn, this allows you to determine the attributes of the folder. If you receive the return value 2066, you know that the only way to receive that value is to have a hidden and compressed folder.

Fortunately, you do not have to perform any sort of mathematical calculations to derive the individual attributes. Instead, you can use the logical AND operator to determine whether an individual switch is on or off. For example, the following code sample checks to see whether the folder is hidden; if it is, the script echoes the message "Hidden folder."

```
If oFolder.Attributes And 2 Then
    MsgBox "Hidden folder."
End If
```

Although the If Then statement might appear a bit strange, it makes a little more sense when read like this: "If the attributes switch with the value 2 is on, then ..." Likewise, this statement would read, "If the attributes switch with the value 16 is on, then ..."

```
If oFolder.Attributes AND 16 Then
```

The following function binds to the folder C:\FSO and then returns the folder attributes as a string.

```
Function GetFolderAttrString( ByVal sFolderName )
   Dim oFSO, oFolder
   Dim sTmp

   Set oFSO = CreateObject("Scripting.FileSystemObject")
   Set oFolder = oFSO.GetFolder(sFolderName)
   If oFolder.Attributes And 16 Then sTmp = "D"
   If oFolder.Attributes And 2 Then sTmp = sTmp & "H"
   If oFolder.Attributes And 4 Then sTmp = sTmp & "S"
   If oFolder.Attributes And 32 Then sTmp = sTmp & "A"
   If oFolder.Attributes And 2048 Then sTmp = sTmp & "C"
   GetFolderAttrString = sTmp
   Set oFolder = Nothing : Set oFSO = Nothing
End Function
MsgBox GetFolderAttrString("C:\RECYCLER")
```

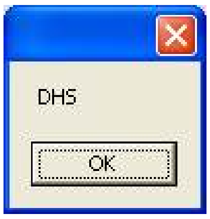

DHS

**Figure 4 Folder Attributes**

## Changing Folder Attributes

As explained in "Working with Bitmaps," individual folder attributes can be likened to switches. If the switch for Hidden is on, the folder is a hidden folder. If the switch for Hidden is off, the folder is not a hidden folder.

This analogy can be carried further by noting that light switches are typically under your control: you can choose to turn them on, or you can choose to turn them off. The same thing is true of folder attributes: as with other switches, you can turn these attribute switches on, or you can turn them off.

You can use scripts to toggle these switches on or off (for example, to hide or unhide a folder). The easiest way to change folder attributes is to use the following procedure:

1. Use the **GetFolder** method to bind to the folder.
2. Check for the value of the attribute you want to change.
   For example, if you want to unhide a folder, check to see whether the folder is hidden.
3. If the folder is hidden, use the logical operator **XOR** to toggle the switch and change it to not hidden. If the folder is not hidden, be careful not to use **XOR**. If you do, the switch will be toggled, and the folder will end up hidden.

For example, the following code uses the AND operator to check whether the switch with the value 2 (hidden folder) has been set on the folder C:\FSO. If it has, the script then uses the XOR operator to turn the switch off and unhide the folder.

```
Set oFSO = CreateObject("Scripting.FileSystemObject")
Set oFolder = oFSO.GetFolder("C:\FSO")
If oFolder.Attributes AND 2 Then
 oFolder.Attributes = oFolder.Attributes XOR 2
End If
```

## Enumerating Files in a Folder

Except for a few rare cases, folders exist solely to act as storage areas for files. Sometimes these folders are required by the operating system; for example, the operating system expects to find certain files in certain folders. In other cases, folders are created as a way to help system administrators manage their computers, or as a way to help users manage their documents. System administrators might place their scripts in a folder named Scripts and their trouble-shooting tools in a folder named Diagnostic Tools; users might place their budget spreadsheets in a folder named Budgets and their payroll information in a folder named Timecards.

Of course, the fact that a folder exists is often of limited use; you must also know what files are stored within that folder. Administrators need to know whether a particular script is stored in C:\Scripts; users need to know whether a particular spreadsheet is stored in C:\Budgets.

The **Folder** object includes a **Files** property that returns a collection of all the files stored in a folder. To retrieve this collection, a script must:

1. Create an instance of the **FileSystemObject**.

2. Use the **GetFolder** method to bind to the appropriate folder.
3. Set an object reference to the **Files** property of the folder.
4. Use a **For Each** loop to enumerate all the files and their properties. The script does not have to bind to each file individually in order to access the file properties.

For example, the following code retrieves a collection of files found in the folder C:\FSO and then echoes the name and size (in bytes) of each file.

```
Set oFSO = CreateObject("Scripting.FileSystemObject")
Set oFolder = oFSO.GetFolder("C:\FSO")
Set oFilesCol = oFolder.Files
For Each oFile in oFilesCol
   MsgBox oFile.Name, oFile.Size
Next
```

As with most collections, you have no control over the order in which information is returned; that is, you cannot specify that files be sorted by name, by size, or by any other criteria. If you want to sort the file collection in a particular way, you need to copy the collection to an **array**, a **Dictionary**, or a disconnected recordset and then sort the items.

## Enumerating Subfolders

In addition to knowing which files are stored in a folder, you need to know which subfolders are stored in a folder; this allows you to develop a complete picture of the folder infrastructure. The **Folder** object includes a **Subfolders** property that returns a collection consisting of the top-level subfolders for a folder.

Top-level subfolders are those folders contained directly within a folder; subfolders contained within those subfolders are not part of the collection. For example, in the sample folder structure shown in Figure 5, only Subfolder 1 and Subfolder 2 are top-level subfolders of the folder Scripts. As a result, only Subfolder 1 and Subfolder 2 are returned as part of the Subfolders property.

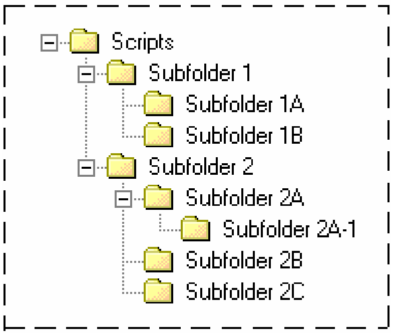


**Figure 5 Sample Folder Structure**

To obtain a subfolder collection, a script must:

1. Create an instance of the **FileSystemObject**.
2. Use the **GetFolder** method to bind to a folder.
3. Create an object reference to the **Subfolders** property. This is required because collections are considered objects.

After you have obtained the object reference to the collection, you can then use a **For Each** loop to enumerate each of the subfolders in that collection. The following code binds to the folder C:\FSO and then echoes the name and size of each subfolder. In addition to the folder name, you can echo any of the folder properties

```
Set oFSO = CreateObject("Scripting.FileSystemObject")
Set oFolder = oFSO.GetFolder("C:\FSO")
Set oSubfoldersCol = oFolder.Subfolders
For Each oSubfolder in oSubfoldersCol
   MsgBox oSubfolder.Name, oSubfolder.Size
Next
```

## Enumerating Subfolders Within Subfolders

Depending on how your file system has been designed, simply knowing the top-level subfolders of a folder might provide sufficient information to map the folder infrastructure. In most file systems, however, folders are nested within folders that are, in turn, nested within other folders. The **Subfolders** collection can tell you that the folder C:\Accounting contains two subfolders: 2001 and 2002. However, it cannot tell which subfolders, if any, are contained within C:\Accounting\2001 and C:\Accounting\2002.

Fortunately, you can use **recursion** to enumerate all the subfolders within a set of subfolders. For example, the **Subfolders** collection. To return the complete set of subfolders (for example, Subfolder 1A and Subfolder 1B), you need to use a recursive function, a function that can call itself. The following example of a script that can enumerate all the subfolders of a folder (as well as any subfolders within those subfolders) is:

1. Creates an instance of the **FileSystemObject**.
2. Uses the **GetFolder** method to bind to the folder C:\Scripts.
3. **GetFolder** is used to return a folder object for C:\Scripts. In turn, the path C:\Scripts is passed as a parameter to the recursive subroutine *ShowSubfolders*. This subroutine will enumerate all the subfolders of C:\Scripts, as well as any subfolders within those subfolders.
4. Retrieves a collection consisting of all the subfolders of the folder C:\Scripts. This collection has two items: Subfolder1 and Subfolder 2.
5. Echoes the folder path of the first item in the collection, Subfolder 1. The subroutine then uses the name of that folder as a parameter passed to itself. In other words, the script now runs the subroutine *ShowSubFolders* using Subfolder 1 as the parameter.
6. Retrieves a collection consisting of all the subfolders of Subfolder 1. This collection has two items: Subfolder1A and Subfolder 1B.
7. Writes to datatable the folder path of the first item in the collection, Subfolder 1A. The subroutine then uses the name of that folder as a parameter passed to itself. In other words, it now runs the function *ShowSubFolders* using Subfolder 1A as the parameter.
8. Passes control to the next item in the collection, Subfolder 1B. This occurs because Subfolder 1A has no subfolders. The subroutine calls itself using Subfolder 1B as the parameter.
9. Finishes recursing through Subfolder 1. This occurs because Subfolder 1B has no subfolders. The script then returns to the second item (Subfolder 2) in the original collection, and repeats the entire process.

```
Option Explicit
Dim oFSO
Dim nRow : nRow = 1
Set oFSO = CreateObject("Scripting.FileSystemObject")
ShowSubfolders oFSO.GetFolder("C:\Scripts")

Sub ShowSubFolders(Folder)
   Dim Subfolder
   For Each Subfolder in Folder.SubFolders
       DataTable.LocalSheet.SetCurrentRow nRow
        nRow = nRow + 1
        DataTable("Folder", dtLocalSheet) = Subfolder.Path
        Msgbox Subfolder.Path
   Next
End Sub
```

```
C:\scripts\Subfolder 1
C:\scripts\Subfolder 1\Subfolder 1A
C:\scripts\Subfolder 1\Subfolder 1B
C:\scripts\Subfolder 2
C:\scripts\Subfolder 2\Subfolder 2A
C:\scripts\Subfolder 2\Subfolder 2A\Subfolder 2A-1
C:\scripts\Subfolder 2\Subfolder 2B
C:\scripts\Subfolder 2\Subfolder 2C
```

## Managing Files

Managing a file system ultimately requires managing the individual files stored within that file system. As a **QuickTest** programmer or a System admin, maybe you want to keep track of the files stored on a computer. For example, you need to know whether the correct diagnostic tools have been copied to a server. You need to know whether certain files (such as games or media files) are being stored on a file server, despite an organizational policy that forbids users to store such files. You need to know whether files have been stored on a computer for months without being accessed and thus are serving no purpose other than using up valuable hard disk space.

In addition to keeping track of these files, you must dynamically manage them as well: Files need to be copied, files need to be moved, files need to be renamed, files need to be deleted. The **FileSystemObject** provides methods that can help you carry out all these administrative tasks.

## Binding to a File

The **FileSystemObject** provides a number of methods, such as the **CopyFile** and **DeleteFile** methods, that allow a script to act on a file without creating an instance of the **File** object. Other tasks, however, require the **File** object. For example, to retrieve a list of file properties, a script must first bind to that file and then retrieve the properties.

The **GetFile** method allows you to bind to an individual file. To do this, you create an instance of the **FileSystemObject** and then create an instance of the File object. When using the **GetFile** method in a script, you must:

- **Specify the path to the file.** The path can be referenced by using either a local path or a **UNC** path (for example, \\accounting\receivables\scriptlog.txt). However, you cannot use wildcards within the path, nor can you specify multiple files. **GetFile** can bind to only a single file at a time.
- **Use the Set keyword when assigning the path to a variable.** The **Set** keyword is required because it indicates that the specified variable is an object reference.

For example, the following code binds to the file C:\FSO\ScriptLog.txt.

```
Set oFSO = CreateObject("Scripting.FileSystemObject")
oFSO.GetFile("C:\FSO\ScriptLog.txt")
```

In general, it is a good idea to pass the absolute path as the **GetFile** parameter; this ensures that the script will always be able to locate the file in question. However, it is possible to use relative paths. For example, the following code sample will work provided that ScriptLog.txt is in the same folder as the script attempting to bind to it:

```
oFSO.GetFile("ScriptLog.txt")
```

Likewise, the next code sample will work if ScriptLog.txt is in the parent folder of the script attempting to bind to it:

```
oFSO.GetFile(".\ScriptLog.txt")
```

Please note, however, that the **FileSystemObject** will not use the path environment variable to search for files. For example, you can start Calculator from the command prompt by typing calc.exe, regardless of the current drive or directory, because the operating system searches all folders in the path to locate the file. This does not happen with the **GetFile** method. The following code sample will fail unless the script is running in the C:\Windows\System32 folder, the same folder where calc.exe is located:

```
oFSO.GetFile("calc.exe")
```

## Verifying That a File Exists

Sometimes it is important simply to know whether a file exists. This might be done as part of a software inventory; for example, you might want to check all your mail servers and see whether a particular script file is present.

Knowing whether a file exists is also important when using scripts to carry out file system management tasks; as you might expect, attempting to copy, move, delete, or otherwise manipulate a file that does not exist will generate a run-time error. To avoid this kind of error, you can use the **FileExists** method to verify the existence of the file. The **FileExists** method requires a single parameter (the path to the file) and returns a **Boolean** value: True if the file exists; **False** if it does not.

The following code uses the **FileExists** method to verify the existence of the file C:\FSO\ScriptLog.txt. If the file exists, the script uses the **GetFile** method to bind to the file. If the file does not exist, the script echoes the message, "File does not exist."

```
Set oFSO = CreateObject("Scripting.FileSystemObject")
If oFSO.FileExists("C:\FSO\ScriptLog.txt") Then
    Set objFile = oFSO.GetFile("C:\FSO\ScriptLog.txt")
Else
    MsgBox "File does not exist."
End If
```

You cannot use wildcard characters to verify whether a particular set of files (such as .txt files) exists in a folder, nor can you use wildcards to verify whether any files at all exist in a folder. For example, the following code sample does not result in an error but always returns the value **False**, regardless of how many files are in the folder:

```
MsgBox oFSO.FileExists("C:\FSO\*.*")
```

If you need to verify the existence of a file based on some criteria other than the path, you have two options:

- Use the **GetFolder** object to bind to the folder, retrieve the **Files** property, and then iterate through the collection of files looking for the files of interest. For example, you could enumerate all the files and file name extensions, and keep track of how many have the .doc extension.
- Use The **Items** method on **FolderItem** object, using **Windows Shell**, the , as same as **FSO** you have to iterate and looking for the file
- Use **WMI**. **WMI** allows you to create more targeted queries, such as selecting all the files with the .doc file name extension. You can then use the **Count** method to determine the number of items in the collection returned to you. If

**Count** is greater than 0, at least one file was found with the .doc extension.

## Deleting a File

The ability to delete files by using the **FileSystemObject** enables you to create scripts that can automatically perform tasks such as disk cleanup operations. For example, you might have a script that periodically searches for and deletes all temporary files (files with the .tmp file name extension). Alternatively, the script might delete files based on some other criteria, such as those that have not been accessed in the past six months, or those with a particular file name extension (such as .bmp or .mp3).

You can delete a file by creating an instance of the **FileSystemObject** and then calling the **DeleteFile** method, passing the path to the file as the parameter. For example, the following code deletes the file C:\FSO\ScriptLog.txt.

```
Set oFSO = CreateObject("Scripting.FileSystemObject")
oFSO.DeleteFile("C:\FSO\ScriptLog.txt")
```

By default, the **DeleteFile** method will not delete a read-only file; if fact, a run-time error will occur if you attempt to delete such a file. To avoid errors, and to delete read-only files, add the optional *Force* parameter. When the *Force* parameter is set to **True**, the **DeleteFile** method can delete any file. For example, this line of code deletes the file ScriptLog.txt, even if that file is marked as read-only:

```
oFSO.DeleteFile("C:\FSO\ScriptLog.txt", True)
```

### Deleting a Set of Files

There might be occasions when you require a script to delete a single, specified file. More likely, though, you will want to use scripts to delete multiple files. For example, at the end of the week, you might want to delete a set of log files that has been archived or delete all the temporary files that have been created but not removed.

Wildcard characters allow you to delete a set of files within a single folder. However, you cannot use the **DeleteFile** method to directly delete files from multiple folders. Instead, your script needs to iterate through the folders and use the **DeleteFile** method to individually delete the files in each folder. To delete files from multiple folders in a single operation (for example, to delete all the .TMP files stored anywhere on a computer), you should use **WMI** instead of the **FileSystemObject** or **Windows Shell**.

To delete a set of files, call the **DeleteFile** method, supplying the path of the folder and the wildcard string required to delete files based on name or file name extension. For example, this line of code deletes all the .doc files in the C:\FSO folder:

```
oFSO.DeleteFile("C:\FSO\*.doc")
```

This line of code deletes all the files with the letters log somewhere in the file name:

```
oFSO.DeleteFile("C:\FSO\*log.*")
```

As noted previously, the **DeleteFile** method does not delete any documents marked as read-only. If a script attempts to delete a read-only document, a run-time error will occur, and the **DeleteFile** method will stop, even if the script uses the **On Error Resume Next** statement. For example, suppose you are trying to delete 1,000 .txt files, and one of those files is marked as read-only. As soon as the script attempts to delete that file, an error will occur, and the **DeleteFile** method will stop. The script will make no attempt to delete any other files, even though none of them are read-only.

Because of that, you can use an optional second parameter, *Force*, that can be set to **True**. When the *Force* parameter is set to **True**, the **DeleteFile** method can delete read-only documents. When the *Force* parameter is set to **False** (the default value), the **DeleteFile** method cannot delete read-only documents.

The following code deletes all the .txt files in the folder C:\FSO. To ensure that all files, including read-only files, are deleted, the *Force* parameter is set to **True**.

```
Set oFSO = CreateObject("Scripting.FileSystemObject")
oFSO.DeleteFile("C:\FSO\*log.*", True)
```

**Tip**

What if you want to delete all files except those marked as read-only? In that case, you can retrieve the complete set of files by using the **Folder** object **Files** property. You can then cycle through the collection, check to see whether each individual file is read-only, and, if it is not, delete the file. Or using the Windows **Shell Object**, by applying the **Filter** method over a **FolderItem** object, then retrieve the **FileItems** collection, and iterate the collection by deleting each member.

## Copying a File

Copying files, either from one folder to another on a single computer or from one computer to another, is a common administrative task. For example, you might want to copy a new monitoring script to all your servers or replace an outdated DLL with a newer version. The **CopyFile** method provides a way to perform these tasks programmatically.

The CopyFile method has two required parameters and one optional parameter:

- *Source.* Path to the file being copied. This can be either a path on the local computer or a UNC path to a remote computer.
- *Destination*. Path to the location where the file is to be copied. This can also be a local path or a UNC path.

To specify that the file keep the same name in its destination location, put a trailing backslash after the destination folder:

```
oFSO.CopyFile "C:\FSO\Scriptlog.txt", "D\Archive"
```

To give the file a new name in its destination location, specify a full file name as the destination:

```
oFSO.CopyFile "C:\FSO\Scriptlog.txt", "D\Archive\NewFile.txt"
```

If the destination folder does not exist, it will automatically be created.

- *Overwrite*. Optional, By default, the **CopyFile** method will not copy a file if a file by that same name exists in the destination location. This can be a problem; among other things, this prevents you from replacing an older version of a file with a newer version. To allow the **CopyFile** method to copy over existing files, set the optional **Overwrite** parameter to **True**.

When specifying the destination folder, it is important to include the trailing backslash (for example, D:\Archive\). If the backslash is there, **CopyFile** will copy the file into the Archive folder. If the backslash is not there, **CopyFile** will try to create a new file named D:\Archive. If the folder D:\Archive already exists, a "Permission denied error" will be generated, and the copy procedure will fail.

The **CopyFile** method will also fail if you attempt to overwrite an existing read-only file, even if you have set the *OverWrite* parameter to **True**. To copy over a read-only file, you must first delete the file and then call the *CopyFile* method.

### Copying a Set of Files

Wildcard characters provide a way to copy an entire set of files as long as these files are all in the same folder. You can copy a set of files using the same parameters used to copy a single file, but you must include a wildcard as part of the source parameter. For example, the following code copies all the .txt files found in C:\FSO to D:\Archive.

```
Set oFSO = CreateObject("Scripting.FileSystemObject")
oFSO.CopyFile "C:\FSO\*.txt" , "D:\Archive\" , True
```

Using wildcards with the **CopyFile** method allows you to copy all the files in a folder without copying any subfolders in that folder; the **CopyFolder** method, by contrast, copies both files and subfolders. The following code statement copies all the files in the C:\FSO folder without copying any subfolders:

```
oFSO.CopyFile "C:\FSO\*.*" , "D:\Archive\"
```

## Moving a File

Instead of copying a file, you might want to move it. For example, if a disk is running low on space, you might want to move a file to a new location. If a computer is changing roles, you might want to move certain diagnostic tools to its replacement. In either case, you do not want two or more copies of the file; you want one copy of the file, stored in a new place.

The **MoveFile** method enables you to move a file from one location to another. The **MoveFile** method works exactly like the **CopyFile** method: You create an instance of the **FileSystemObject**, call the **MoveFile** method, and pass two parameters:

- The complete path to the file to be moved.
- The complete path to the new location, making sure to include the trailing backslash.

For example, the following code moves C:\FSO\ScriptLog.log to the Archive folder on drive D.

```
Set oFSO = CreateObject("Scripting.FileSystemObject")
oFSO.MoveFile "C:\FSO\ScriptLog.log" , "D:\Archive\"
```

## Moving a Set of Files

You can also use wildcard characters to move multiple files in a single operation. For example, to move all the files in the FSO folder that begin with the letters data, use the parameter C:\FSO\Data*.*. Wildcard characters are especially useful for moving all the files of a particular type because file types are usually denoted by file name extensions. For example, the script in Listing 4.25 moves all the log files (with the .log file name extension) from the FSO folder on drive C to the Archive folder on drive D.

```
Set oFSO = CreateObject("Scripting.FileSystemObject")
oFSO.MoveFile "C:\FSO\*.log" , "D:\Archive\"
```

## Rename a File

The **FileSystemObject** does not include a direct method for renaming a file. However, in much the same way that a folder can be renamed using the **MoveFolder** method, files can be renamed using the **MoveFile** method. To rename a file, call the **MoveFile** method but leave the file in its current folder.

For example, the following code renames ScriptLog.txt to BackupLog.txt. Technically, the script actually moves C:\FSO\ScriptLog.txt to a new path: C:\FSO\BackupLog.txt. The net result, however, is that the file named ScriptLog.txt is now named BackupLog.txt.

```
Set oFSO = CreateObject("Scripting.FileSystemObject")
oFSO.MoveFile "C:\FSO\ScriptLog.log" , "D:\FSO\BackupLog.txt"
```

## Managing File Properties

Files have a number of properties that are extremely useful for managing a file system. For example, the **DateLastAccessed** property tells you the date when someone last opened the file. This property can be used to identify files that are taking up disk space yet are never used. Similarly, the **Size** property tells you the size of a file in bytes. This helps you to better analyze disk usage; you can tell whether a single file might be using up more than its fair share of storage space.

Traditionally, system administrators have accessed file properties by using either Windows Explorer or command-line tools. Although these tools can return information about the files on a computer, they are not always designed to save this data or to act on it. In addition, many of these tools have only a limited ability to be automated, making it more difficult for system administrators to periodically sweep their hard drives and search for files that meet specific criteria.

Fortunately, detailed information about any file on a computer can also be retrieved by using the **FileSystemObject**; among other things, this allows you to automate the process of querying the file system for information about a file or group of files.

To access file properties, a script must:

1. Create an instance of the **FileSystemObject**.
2. Use the **GetFile** method to create an object reference to a particular file. The

script must pass the path of the file as the **GetFile** parameter.

3. Echo (or otherwise manipulate) the appropriate file properties

```
Set oFSO = CreateObject("Scripting.FileSystemObject")
Set oFile = oFSO.GetFile("c:\windows\system32\scrrun.dll")
sOut = sOut &  "Date created: " & oFile.DateCreated & vbCrLf
sOut = sOut &  "Date last accessed: " & oFile.DateLastAccessed & vbCrLf
sOut = sOut &  "Date last modified: " & oFile.DateLastModified & vbCrLf
sOut = sOut &  "Drive: " & oFile.Drive & vbCrLf
sOut = sOut &  "Name: " & oFile.Name & vbCrLf
sOut = sOut &  "Parent folder: " & oFile.ParentFolder & vbCrLf
sOut = sOut &  "Path: " & objFile.Path & vbCrLf
sOut = sOut &  "Short name: " & oFile.ShortName & vbCrLf
sOut = sOut &  "Short path: " & oFile.ShortPath & vbCrLf
sOut = sOut &  "Size: " & oFile.Size & vbCrLf
sOut = sOut &  "Type: " & oFile.Type & vbCrLf
MsgBox sOut
```

```
Date created: 10/29/2001 10:35:36 AM
Date last accessed: 2/14/2002 1:55:44 PM
Date last modified: 8/23/2001 4:00:00 AM
Drive: c:
Name: scrrun.dll
Parent folder: C:\Windows\system32
Path: C:\Windows\system32\scrrun.dll
Short name: scrrun.dll
Short path: C:\Windows\system32\scrrun.dll
Size: 147483
Type: Application Extension
```

## Enumerating File Attributes

Like folders, files also have attributes that can be retrieved and configured using the FileSystemObject. Also like folders, file attributes are returned as a bitmap value. (For more information on bitmap values and how to use them, see Managing Folder Attributes on page 12.) File attributes can include any or all of the values described in Table 5 on page 112

To retrieve the attributes of a file, use the **GetFile** method to bind to the file. After you have created an object reference to the file, you can use the logical **AND** operator to determine the file attributes. If the file does not have any attributes configured, the **Attributes** value will be 0.

The following function binds to the folder C:\FSO\ScriptLog.txt and then returns the folder attributes as a string.

```
Function GetFileAttrString( ByVal sFileName )
   Dim oFSO, oFolder
   Dim sTmp

   Set oFSO = CreateObject("Scripting.FileSystemObject")
   Set oFile = oFSO.GetFolder(sFileName)
   If oFile.Attributes And 1 Then sTmp = sTmp & "R"
   If oFile.Attributes And 2 Then sTmp = sTmp & "H"
   If oFile.Attributes And 4 Then sTmp = sTmp & "S"
   If oFile.Attributes And 32 Then sTmp = sTmp & "A"
   If oFile.Attributes And 64 Then sTmp = sTmp & "L"
   If oFile.Attributes And 2048 Then sTmp = sTmp & "C"
   GetFileAttrString = sTmp
   Set oFile = Nothing : Set oFSO = Nothing
End Function
MsgBox GetFileAttrString("C:\FSO\ScriptLog.txt")
```

## Configuring File Attributes

In addition to enumerating file attributes, the **FileSystemObject** provides a way to configure the following attributes:

- ReadOnly, Hidden, System, Archive

To configure a file attribute, the script should use the following procedure:

1. Use the **GetFile** method to bind to the file.
2. Check for the attribute you want to change.
3. For example, if you want to make a file read-only, check to see whether the file has already been marked read-only.
4. If the file is not read-only, use the logical operator **XOR** to toggle the switch. This will mark the file as read-only. If the file is already read-only, be careful not to use **XOR**. If you do, the switch will be toggled, and the read-only attribute will be removed.

The following code uses the **AND** operator to check whether the switch with the value 1 (read-only) has been set on the file C:\FSO\TestScript.vbs. If the file is not read-only, the script uses the **XOR** operator to turn the switch on and mark the file as read-only.

```
Set oFSO = CreateObject("Scripting.FileSystemObject")
Set oFile = oFSO.GetFolder("C:\FSO\TestScript.vbs")
If oFile.Attributes AND 1 Then
   oFile.Attributes = oFile.Attributes XOR 1
End If
```

You can also simultaneously remove the ReadOnly, Hidden, System, and Archive attributes by using the following code statement:

```
objFile.Attributes = objFile.Attributes AND 0
```

## Parsing File Paths

A path is a hierarchical series of names that allow you to pinpoint the exact location of a file or folder. In that respect, paths are similar to street addresses: they provide information that tells you precisely where to locate an object. A street address such as One Main Street, Redmond, WA, tells you precisely where to find a particular residence. Likewise, the path C:\FSO\Scripts\ScriptLog.txt tells you precisely where to locate a particular file. Just as only one building can be located at One Main Street, Redmond, WA, only one file can be located at C:\FSO\Scripts\ScriptLog.txt.

Complete paths such as C:\FSO\Scripts\ScriptLog.txt are very important because they provide the only way to uniquely identify a file or folder location. Because of that, there will be times when your script will need the complete path.

At other times, however, you might want only a portion of the path. For example, you might want to extract only the file name or only the file name extension. To allow you to parse paths and extract individual path components, the **FileSystemObject** provides the methods

- GetAbsolutePathName, GetParentFolderName, GetFileName, GetBaseName, GetExtensionName

The following code parses the path for the file ScriptLog.txt. This script works only if ScriptLog.txt is in the same folder as the script doing the parsing. If the two files are stored in different folders, you must pass the complete path to the **GetFile** method (for example, C:\FSO\Scripts\ScriptLog.txt).

```
Set oFSO = CreateObject("Scripting.FileSystemObject")
Set oFile = objFSO.GetFile("ScriptLog.txt")
sOut = sOut & "Absolute path: " & oFSO.GetAbsolutePathName(oFile) & vbCrLf
sOut = sOut & "Parent folder: " & oFSO.GetParentFolderName(oFile) & vbCrLf
sOut = sOut & "File name: " & oFSO.GetFileName(oFile) & vbCrLf
sOut = sOut & "Base name: " & oFSO.GetBaseName(oFile) & vbCrLf
sOut = sOut & "Extension name: " & oFSO.GetExtensionName(oFile)
MsgBox sOut
```

```
Absolute path: C:\FSO\Scripts\ScriptLog.txt
Parent folder: C:\FSO\Scripts
File name: ScriptLog.txt
Base name: ScriptLog
Extension name: txt
```

## Retrieving the File Version

File versions that are incompatible or out-of-date can create considerable problems. For example, a script that runs fine on Computer A, where version 2.0 of a particular DLL has been installed, might fail on Computer B, which has version 1.0 of that DLL installed.

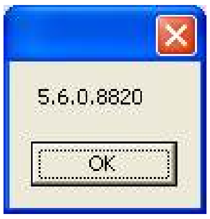
These problems can be difficult to troubleshoot, because you are likely to get back an error saying that the object does not support a particular property or method. This is because the version of the object installed on Computer B does not support the new property or method. If you try to debug the script on Computer A, you will have difficulty finding the problem because the version of the object installed on Computer A does support the property or method in question.

The **GetFileVersion** method allows you to retrieve version information from a file. To use this method, a script must:

1. Create an instance of the **FileSystemObject**.
2. Call the **GetFileVersion** method, passing the path to the file as the sole parameter.

For example, the script in Listing 4.31 retrieves the file version for Scrrun.dll.

```
Set oFSO = CreateObject("Scripting.FileSystemObject")
MsgBox oFSO.GetFileVersion("c:\windows\system32\scrrun.dll")
```



5.6.0.8820

**Figure 6 Version Number for Scrrun.dll**

Version numbers are typically displayed in four parts, such as 5.6.0.8820, rather than a single number (such as version 1 or version 5). Version number 5.6.0.8820 contains the following parts:

- 5 - The major file part.
- 6 - The minor file part. The major and minor parts together represent the way a version is typically referred to. In conversation, you would likely refer to version 5.6 rather than version 5.6.0.8820.
- 0 - The build part. This is typically 0.
- 8820 - The private file part.

**Note**

Not all files types support versioning. Executable files and DLLs typically support versioning; plain-text files, including scripts, typically do not.

## Reading and Writing Text Files

One of the more powerful tools available for programmers, is the text file. This might seem hard to believe in an age of high-resolution graphics and multi-user databases. Nevertheless, simple text files, such as those created in Notepad, remain a key element in system administration. Text files are lightweight and low maintenance: They use up very little disk space and require no additional software of any kind to be installed on the computer. Text files are easy to work with and are extremely portable: A text file created by using a script can be copied and viewed on almost any computer in the world, including computers that do not run a Windows operating system.

In addition to their convenience, text files provide a quick, easy, and standardized way to get data both into a script and out of a script. Text files can be used to hold arguments that would otherwise need to be typed at the command line or hard-coded into a script; rather than typing 100 server names at the command line, a script can simply read those names from a text file. Likewise, text files provide a quick and easy way to store data retrieved from a script. This data could be written directly to a database; however, that requires additional configuration on the server, additional coding in the script, and additional overhead when the script runs. Instead, data can be saved to a text file and then later imported into a database.

The **FileSystemObject** provides a number of methods for both reading from and writing to 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 **CreateTexFile** method, passing the complete path name as the method parameter. For example, the following code creates a new text file named ScriptLog.txt in the C:\FSO folder.

```
Set oFSO = CreateObject("Scripting.FileSystemObject")
Set oFile = oFSO.CreateTextFile("c:\FSO\ScriptLog.txt")
```

If the file does not exist, the **CreateTextFile** method creates it. If the file does exist, the **CreateTextFile** method will overwrite the existing file and replace it with the new, blank file. If you prefer that the existing file not be overwritten, you can include the optional *Overwrite* parameter. When this parameter is **False**, existing files are not overwritten; when this parameter is **True** (the default value), existing files are overwritten. For example, the following code sample does not overwrite the file C:\FSO\ScriptLog.txt if that file already exists:

```
Set oFile = oFSO.CreateTextFile("C:\FSO\ScriptLog.txt", False)
```

If you set the *Overwrite* parameter to **False** and the file already exists, a run-time error will occur. Because of that, you might want to check for the existence of the file and then, if the file exists, take some other action, such as allowing the user to specify an alternative file name for the new file.

## Creating File Names Within the Script

One way to avoid the problems that can occur if a file already exists is to allow the script to generate a unique file name. Because the file name generator does not create meaningful file names, this is probably not a good approach for naming log files and other files that you might need to refer to in the future. However, it does provide a way to ensure unique file names for scripts that require a temporary file. For example, you might have your script save data in **HTML** or **XML** format, have that data displayed in a **Web** browser, and then have this temporary file deleted as soon as the **Web** browser is closed. In a situation such as that, you can use the **GetTempFile** name method to generate a unique file name.

To generate a unique file name, a script must create an instance of the **FileSystemObject** and then call the **GetTempName** method (with no parameters). For example, the following code uses a **For Next** loop to create 10 random file names.

```
Set oFSO = CreateObject("Scripting.FileSystemObject")
For i = 1 to 10
   sTempFile = oFSO.GetTempName
   MsgBox sTempFile
Next
```

**Note**

The file names generated by **GetTempName** are not guaranteed to be unique, partly because of the algorithm used to generate the names and partly because there are only a finite number of possible names; file names are limited to eight characters, and the first three characters are always **rad**. For example, in a test script that created 10,000 file names, one right after another, 9,894 names were unique. The remaining 106 were duplicates (53 pairs of duplicated names).

## 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.

After you have a reference to the **TextStream** object, you can either read from or write to the file. However, you cannot simultaneously read from and write to the same file. In other words, you cannot open a file, read the contents, and then write additional data to the file, all in the same operation. Instead, you must read the contents, close the file, and then reopen and write the additional data.

When you open an existing text file, the file can be opened either for reading or for writing. When you create a new text file, the file is open only for writing, if for no other reason than that there is no content to read.

Finally, you should always close a text file. Although this is not required (the file will generally be closed as soon as the script terminates), it is good programming practice.

To open a text file:

1. Create an instance of the **FileSystemObject**.
2. Use the **OpenTextFile** method to open the text file. The **OpenTextFile** method requires two parameters: the path to the file and one of the following values:
   - **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.

You must use the appropriate parameter when opening the file. For example, if you open a file for reading and then attempt to write to the file, you will receive a "Bad file mode" error. You will also receive this error if you attempt to open anything other than a plain-text file. (It is worth noting that both HTML and XML files are plain-text files.)

However, you cannot use the constants without first defining them. This is due to the fact that **VBScript** does not have intrinsic access to **COM** object constants. The following script sample will fail and return an "Invalid procedure call or argument" error because the **ForReading** constant has not been explicitly defined. Because it has not been defined, **ForReading** is automatically assigned the value 0, and 0 is not a valid parameter for **OpenTextFile**.

## 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. Nevertheless, it is a good idea to always close text files when you are finished with them. Not only is this good programming practice, but problems will occur if you try to do one of the following without first closing the file:

- **Delete the file**. As noted previously, you might occasionally write scripts that create a temporary file, use that file for some purpose, and then delete the file before the script terminates. If you attempt to delete an open file, however, you will encounter an "Access denied" error because the operating system will not allow you to delete an open file.
- **Reread the file.** There might be times when you need to read the same file multiple times within a script. For example, you might open a text file, save the entire contents of the file to a string variable, and then search that string for the existence of a particular error code. If the code is found, you might then read the file on a line-by-line basis, extracting each line where the error was recorded.

## Reading Text Files

Reading data from a text file is a standard procedure used in many enterprise scripts. You might use this capability to:

If you try to read an open file multiple times, however, you either will not receive the expected results or will encounter a run-time error.

- Read in command-line arguments. For example, a text file might contain a list of computers, with the script designed to read in the list and then run against each of those computers.
- Programmatically search a log file for specified conditions. For example, you might search a log file for any operations marked Error.
- Add the contents of a log file to a database. For example, you might have a service or an application that saves information in plain-text format. You could write a script that reads in the text file and then copies the relevant information to a database.

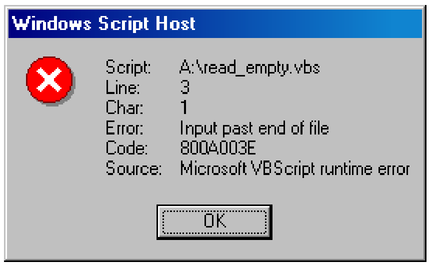
The **FileSystemObject** can be used to read the contents of a text file. When using the **FileSystemObject**, keep the following limitations in mind:

- The **FSO** can read only ASCII text files. You cannot use the **FSO** to read Unicode files or to read binary file formats such as Microsoft Word or Microsoft Excel.
- The **FileSystemObject** reads a text file in one direction: from the beginning to the end of the text file. In addition, the **FSO** reads only line by line. If you need to go back to a previous line, you must return to the beginning of the file and read forward to the required line.
- You cannot open a file for simultaneous reading and writing. If you open a file for reading, you must open the file a second time if you want to modify the contents. If you attempt to read a file after opening it in write mode, you will receive a "bad file mode" error.

## Verifying the Size of a File

Windows will sometimes create text files that are empty - that is, files that contain no characters and have a file size of 0 bytes. This often occurs with log files, which remain empty until a problem is recorded there. For example, if problems occur with a user logon (such as a user attempting to log on with an incorrect password or user account), those problems will be recorded in the Netlogon.log file. Until such a problem occurs, however, the Netlogon.log file remains empty.

Empty files represent a problem for script writers, because a VBScript run-time error will occur if you attempt to read such a file. If you try to read an empty file, an error message similar to the one shown in Figure 4.8 appears.


Windows Script Host

```
Script:   A:\read_empty.vbs
Line:     3
Char:     1
Error:    Input past end of file
Code:     800A003E
Source:   Microsoft VBScript runtime error
```

**Figure 7 - Empty File Error Message**

If there is a chance that a file might be empty, you can avoid errors by checking the file size before attempting to read the file. To do this, the script must:

1. Create an instance of the **FileSystemObject**.
2. Use the **GetFile** method to bind to the file.
3. Use the **Size** property to ensure that the file is not empty before attempting open it.

The following code binds to the file C:\Windows\Netlogon.log. The script checks the size of the file; if the size is greater than 0, the script opens and reads the file. If the file size is 0, the script echoes the message "The file is empty."

```
Set oFSO = CreateObject("Scripting.FileSystemObject")
Set oFile = oFSO.GetFile("C:\Windows\Netlogon.log")
If oFile.Size > 0 Then
   Set oTxtFile = oFSO.OpenTextFile("C:\Windows\Netlogon.log", 1)
   sContents = oTxtFile.ReadAll
   MsgBox sContents
   oTxtFile.Close
Else
   MsgBox "The file is empty."
End If
```

## Reading an Entire Text File

The **ReadAll** method provides the easiest way to read a text file: You simply call the method, and the entire text file is read and stored in a variable. Having the contents of the text file stored in a single variable can be useful in a number of situations. For example, if you want to search the file for a particular item (such as an error code), it is easier to search a single string than to search the file line by line.

Likewise, if you want to concatenate (combine) text files, the **ReadAll** method provides the quickest and easiest method. For example, suppose you have a set of daily log files that you want to combine into a single weekly log file. To do that, a script can:

- Open the text file for Monday and use **ReadAll** to store the entire contents in a variable.
- Open the weekly log file for appending, and write the contents of the variable to the file. This is possible because any formatting (such as line breaks or tabs) that is read in from the Monday file is preserved in the variable.
- Repeat steps 1 and 2 until the entire set of daily files has been copied into the weekly log.

**Note**

Although it is easier to search a single string, it is not necessarily faster. The **ReadAll** method took less than a second to search a 388-KB test file of approximately 6,000 lines. Reading and searching the file on a line-by-line basis also took less than a second

## Reading a Text File Line by Line

For some purposes, text files typically serve as flat-file databases, with each line of the file representing a single record in the database. For example, scripts often read in a list of server names and then carry out an action against each of those servers. In those instances, the text will look something like the following:

```
atl-dc-01
atl-dc-02
atl-dc-03
atl-dc-04
```

When a text file is being used as a flat-file database, a script will typically read each record (line) individually and then perform some action with that record. For example, a script (using the preceding sample text file) might read in the name of the first computer, connect to it, and carry out some action. The script would then read in the name of the second computer, connect to it, and carry out that same action. This process would continue until all the records (lines) in the text file have been read.

The **ReadLine** method allows a script to read individual lines in a text file. To use this method, open the text file, and then set up a **Do Loop** that continues until the **AtEndOfStream** property is True. (This simply means that you have reached the end of the file.) Within the **Do Loop**, call the **ReadLine** method, store the contents of the first line in a variable, and then perform some action. When the script loops around, it will automatically drop down a line and read the second line of the file into the variable. This will continue until each line has been read (or until the script specifically exits the loop).

For example, the following code opens the file C:\FSO\ServerList.txt and then reads the entire file line by line, echoing the contents of each line to the screen.

```
Option Explicit
Dim oFSO, oTxtFile, sLine
Set oFSO = CreateObject("Scripting.FileSystemObject")
Set oTxtFile = oFSO.OpenTextFile("C:\FSO\ServerList.txt", 1)
```

```
Do Until oTxtFile.AtEndOfStream
   sLine = oTxtFile.ReadLine
   MsgBox sLine
Loop
objFile.Close
```

## "Reading" a Text File from the Bottom to the Top

As noted previously, the **FileSystemObject** can read a text file only from the beginning to the end; you cannot start at the end and work your way backwards. This can sometimes be a problem when working with log files. Most log files store data in chronological order: The first line in the log is the first event that was recorded, the second line is the second event that was recorded, and so on. This means that the most recent entries, the ones you are perhaps most interested in, are always located at the very end of the file.

There might be times when you want to display information in reverse chronological order - that is, with the most recent records displayed first and the oldest records displayed last. Although you cannot read a text file from the bottom to the top, you can still display the information in reverse chronological order. To do this, a script must:

1. Create an array to hold each line of the text file.
2. Use the **ReadLine** method to read each line of the text file and store each line as a separate element in the array.
3. Display the contents of the array on screen, starting with the last element in the array (the most recent record in the log file) and ending with the first element in the array (the oldest log file).

For example, the following code reads in the file C:\FSO\ScriptLog.txt, storing each line as an element in the array *arrFileLines*. After the entire file has been read, the contents are echoed to the screen, beginning with the last element in the array. To do this, the **For Loop** begins with the last element (the upper bound of the array) and incrementally works down to the first element (the lower bound).

```
Option Explicit
Dim oFSO, oTxtFile, sLine, i, arrFileLines
Set oFSO = CreateObject("Scripting.FileSystemObject")
Set oTxtFile = oFSO.OpenTextFile("C:\FSO\ServerList.txt", 1)
Do Until oTxtFile.AtEndOfStream
   Redim Preserve arrFileLines(i)
   arrFileLines(i) = oTxtFile.ReadLine
   i = i + 1
Loop
oTxtFile.Close
For i = Ubound(arrFileLines) to LBound(arrFileLines) Step -1
    MsgBox arrFileLines(i)
Next
```

## Reading a Text File Character by Character

In a fixed-width text file, fields are delimited by length: Field 1 might consist of the first 15 characters on a line, Field 2 might consist of the next 10 characters, and so on. Thus a fixed-width text file might look like the following:

```
Server                    Value                     Status
atl-dc-01                 19345                     OK
atl-printserver-02        00042                     OK
atl-win2kpro-05           00000                     Failed
```

In some cases, you might want to retrieve only the values, or only the status information. The value information, to pick one, is easy to identify: Values always begin with the 26th character on a line and extend no more than 5 characters. To retrieve these values, you need to read only the 26th, 27th, 28th, 29th, and 30th characters on each line.

The **Read** method allows you to read only a specified number of characters. Its sole parameter is the number of characters to be read. For example, the following code sample reads the next 7 characters in the text file and stores those 7 characters in the variable *sCharacters*:

```
sCharacters = oTxtFile.Read(7)
```

By using the **Skip** and **SkipLine** methods, you can retrieve selected characters from a text file. For example, the following code reads only the sixth character in each line of a text file. To do this, the script must:

1. Skip the first five characters in a line, using **Skip**(5).
2. Read the sixth character, using **Read**(1).
3. Skip to the next line of the file.

```
Option Explicit
Dim oFSO, oTxtFile, sCharacters
Set oFSO = CreateObject("Scripting.FileSystemObject")
Set oTxtFile = oFSO.OpenTextFile("C:\FSO\ScriptLog.txt", 1)
Do Until oTxtFile.AtEndOfStream
   objFile.Skip(5)
   sCharacters = oTxtFile.Read(1)
   MsgBox sCharacters
   oTxtFile.SkipLine
Loop
```

## Write to Text Files

Writing data to a text file is another powerful aid in writing scripts. Text files provide a way for you to permanently save data retrieved by a script; this data can be saved either instead of or in addition to being displayed on the screen. Text files also provide a way for you to keep a log of the actions carried out by a script. This can be especially useful when creating and debugging scripts. By having the script record its actions in a text file, you can later review the log to determine which procedures the script actually carried out and which ones it did not.

The **FileSystemObject** gives you the ability to write data to a text file. To write data using the **FSO**, a script must do the following:

1. Create an instance of the FileSystemObject.
2. Use the OpenTextFile method to open the text file. You can open the text file in one of two ways:
   - **For writing** (parameter value 2, constant = ForWriting). Files opened in

this mode will have new data replace any existing data in its entirety. (That is, existing data will be deleted and the new data added.) Use this mode 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 mode to add data to an existing file.

3. Use either the **Write**, **WriteLine**, or **WriteBlankLines** method to write to the file.
4. Close the text file, using the **Close** method.

One weakness with the **FileSystemObject** is that it cannot be used to directly modify specific lines in a text file; for example, you cannot write code that says, in effect, "Skip down to the fifth line in this file, make a change, and then save the new file." To modify line 5 in a 10-line text file, a script must instead:

1. Read in the entire 10-line file.
2. Write lines 1-4 back to the file.
3. Write the modified line 5 to the file.
4. Write lines 6-10 back to the file

## Overwriting Existing Data

For example, suppose you have a script that runs every night, retrieving events from the event logs on your domain controllers, writing those events to a database, and recording which computers were successfully contacted and which ones were not. For historical purposes, you might want to keep track of every success and every failure over the next year. This might be especially useful for a new script just being put into use, or for a network with suspect connectivity or other problems that crop up on a recurring basis.

On the other hand, you might simply want to know what happened the last time the script ran. In other words, you do not want a log file that contains data for the past 365 days. Instead, you want a log file that contains only the most recent information. That allows you to open the file and quickly verify whether or not the script ran as expected.

When you open a text file in **ForWriting** mode, any new data you write to the file replaces all the existing data in that file. For example, suppose you have the complete works of Shakespeare stored in a single text file. Suppose you then run a script that opens the file in **ForWriting** mode and writes the single letter a to the file. After the file has been written and closed, it will consist only of the letter a. All the previously saved data will be gone.

The following code opens the text file C:\FSO\ScriptLog.txt in **ForWriting** mode and then writes the current date and time to the file. Each time this script is run, the old date and time are replaced by the new date and time. The text file will never contain more than a single date-time value.

```
Option Explicit
Const ForWriting = 2
Dim oFSO, oTxtFile
Set oFSO = CreateObject("Scripting.FileSystemObject")
Set oTxtFile = oFSO.OpenTextFile("C:\FSO\ScriptLog.txt", ForWriting)
oTxtFile.Write Now
oTxtFile.Close
```

## Appending New Data to Existing Data

Some scripts are designed to run at regularly scheduled intervals and then collect and save a specific kind of data. These scripts are often used to analyze trends and to look for usage over time. In these instances, you typically do not want to overwrite existing data with new data.

For example, suppose you have a script that monitors processor usage. At any given point in time, processor usage could be anywhere from 0 percent to 100 percent by itself, that single data point is meaningless. To get a complete picture of how much a processor is being utilized, you need to repeatedly measure and record processor usage. If you measure processor use every few seconds and get back data like 99 percent, 17 percent, 92 percent, 90 percent, 79 percent, 88 percent, 91 percent, you can assume processor use is very high. However, this can only be determined by comparing processor use over time.

By opening a text file in **ForAppending** mode, you can ensure that existing data is not overwritten by any new data; instead, that new data is appended to the bottom of the text file.

## Managing Files and Folders Using WMI

Scripts designed to help with file system management typically rely on the **FileSystemObject**. There are several reasons why the primary scripting enabler for file system management should be **WMI**, and in particular, the **Win32_Directory** and **CIM_Datafile** classes.

The **FileSystemObject** and **WMI** have overlapping functionality: You can use either one to copy, delete, move, rename, or otherwise manipulate files and folders. However, **WMI** has two major advantages over the **FileSystemObject**. First, **WMI** works as well on remote computers as it does on the local computer. By contrast, the **FileSystemObject** is designed to work solely on the local computer; to use the **FileSystemObject** against a remote computer, you typically have to configure both computers to allow remote script execution using the **WSHController** Object.

Second, **WMI** can work with collections of files and folders across an entire computer. For example, using **WMI** you can delete all the .mp3 files on a computer by using a simple script that essentially says, "Locate all the .mp3 files on this computer, and then delete them." By contrast, the **FileSystemObject** is designed to work with a single folder at a time. To delete all the .mp3 files on a computer, you need to bind to each folder on the computer, check for the existence of .mp3 files, and then delete each one.

**WMI** does have some limitations, however. Enumerating files and folders using WMI can be slow, and processor-intensive. For example, on a Microsoft® Windows® 2000-based computer with approximately 80,000 files, the **FileSystemObject** returned a list of all files in less than 5 minutes. By contrast, WMI required over 30 minutes returning the same list. During that time, processor use averaged about 30 percent, often spiking above 50 percent. Although you would normally not need to retrieve a list of every single file on a computer, it is clearly not advisable to use **WMI** if you ever do need to perform this task.

Moreover, processor-intensive **WMI** queries cannot always be stopped simply by terminating the script. Suppose you start a query that returns a list of all the files on the file system. After a few minutes, you change your mind and terminate the script. There is a very good chance that the query will continue to run, using up memory and draining system resources, even though the script has been stopped. This is because the script and its query run on separate threads. To stop a query such as this, you typically have to stop and then restart the **WMI** service.

In addition, you cannot speed up a file or folder query by requesting only a subset of file or folder properties.

On a computer running Windows 2000, this query, which returns all the properties for all the files, took about 30 minutes to complete:

```
"Select * From CIM_Datafile"
```

Although this query returns only the name property of the files, it required the same amount of time to complete:

```
"Select Name From CIM_Datafile"
```

## Comparing WMI and the FileSystemObject

Automation developers, who write scripts to also manage files and folders typically, use the **FileSystemObject** rather than **WMI**. This is due more to familiarity than to anything else; most of the same core capabilities are available using either the **FileSystemObject** or **WMI**. The leads to an obvious question: when should you use the **FileSystemObject**, and when should you use **WMI**? Or does it even matter?

There is no simple answer to that question; instead, the best approach usually depends on what your script needs to accomplish. When choosing a method for managing files and folders, you should consider the impact of:

- Speed of execution.
- The ability to recover from errors.
- The ability to run against remote computers.
- Ease of use.

### Speed

If your goal is to enumerate all the files on a hard disk, the **FileSystemObject** will be much faster. For example, on a Windows 2K system, with a relatively modest 21,963 files, the **FileSystemObject** required 68 seconds to return a list of all the files on drive C. By contrast, **WMI** took nearly 10 times as long (661 seconds) to complete the same operation.

With more targeted queries, however, **WMI** can be both faster and more efficient. For example, the **FileSystemObject** required 90 seconds to return a list of the 91 .bmp files on the Windows 2000-based test computer. It actually takes longer for the **FileSystemObject** to return a subset of files than it does to return the entire set of files; this is because the **FileSystemObject** does not allow you to use SQL-type queries. Instead, it uses recursion to return a list of all the files on the computer and then, in this case, individually checks each file to see whether the extension is .bmp.

Using a filtered query, **WMI** returned the list of .bmp files in 18 seconds. **WMI** is faster in this case because it can request a collection of all .bmp files without having to return the entire file set and then check the file name extension of each item in the collection.

## Error Handling

The **FileSystemObject** sometimes provides a faster solution; it rarely, if ever, provides a more robust solution. The **FileSystemObject** provides no ability to recover from an error, even if your script includes the **On Error Resume Next** statement.

For example, suppose you have a script designed to delete 1,000 files from a computer. If an error occurs with the very first file, the script fails, and no attempt is made to delete any of the remaining files. If an error condition occurs, the **FileSystemObject** - and the script - immediately terminates. If an error occurs partway through an operation, you will have to manually determine which portions of the procedure succeeded and which portions did not.

**WMI** is better able to recover from a failed operation. If **WMI** is unable to delete the first of the 1,000 files, it simply reports an error condition and then attempts to delete the next file in the collection. By logging any errors that occur, you can easily determine which portions of a procedure succeeded and which ones did not.

**Note** : You can log these individual errors because **WMI** treats each file operation separately; if you have 1,000 files, **WMI** treats this as 1,000 separate deletions. The **FileSystemObject**, by comparison, treats each file operation as one procedure, regardless of whether you have 1 file, 10 files, or 1,000 files.

**WMI** is also able to more intelligently deal with file and folder access permissions. For example, suppose you write a script to enumerate all the files on a hard drive. If **WMI** encounters a folder that you do not have access to, the script simply skips that folder and continues. The **FileSystemObject**, however, attempts to list the files in that folder. That operation will fail because you do not have access to the folder. In turn, the **FileSystemObject**, and your script, will also fail. This is a problem with Windows 2000-based computers because they typically include a System Volume Information folder that, by default, grants access only to the operating system. Without taking precautions to work around this folder, any attempt to enumerate all the files on a computer using the **FileSystemObject** is guaranteed to fail.

## Remote Computers

One of the major benefits of **WMI** is that a script originally designed to run on the local computer can be easily modified to run on a remote computer. For example, this script sets the name of the computer (the variable *sComputer*) to a dot ("."). In **WMI**, specifying "." as the computer name causes the script to run against the local computer.

```
sComputer = "."
Set oWMI = GetObject("winmgmts:" & "!\\" & sComputer & "\root\cimv2")
```

To run the script against a remote computer (for example, atl-dc-01), simply change the value of the variable *sComputer*:

```
sComputer = "atl-dc-01"
Set oWMI = GetObject("winmgmts:" & "!\\" & sComputer & "\root\cimv2")
```

For most file and folder operations, this is the only change required to make a script work on a remote computer.

Working with remote computers using the **FileSystemObject** is more complicated. It is easy to access a shared folder using the **FileSystemObject**; simply connect to the folder using the Universal Naming Convention (UNC) path (for example, \\atl-dc-01\scripts). However, it is much more difficult to carry out such tasks as searching a remote computer for all files of a specified type. For the most part, there are only two ways to carry out this procedure:

- Configure an instance of the **WSHController** object, which allows you to run a script against a remote computer as if that script was running locally.
- Connect to the administrative shared folders of the remote computer (for example, using the path \\atl-dc-01\C$ to connect to drive C on the remote computer). This approach works, provided the administrative shared folders on the remote machines to - not disabled.

Depending on the operation you are attempting, you might also have to determine what disk drives are installed on the remote computer. **WMI** can return all the files within the file system, regardless of the drive they are stored on. By contrast, the **FileSystemObject** can work only on a disk-by-disk basis. Before you can search a computer for files, you must first obtain a list of all the disk drives and then individually search each drive.

## Easy to Use

**WMI** allows you to work with collections and to create queries that return a targeted set of items. This makes **WMI** easier to use for tasks that require you to do such things as identify all the read-only folders on a computer or delete all the .mp3 files in a file system; you issue a request, and **WMI** returns a collection of all those items, regardless of their physical location on the computer. This means that you can accomplish the task in far fewer lines of code than it would take to accomplish the same task using the **FileSystemObject**.

For example, this **WMI** query returns a collection of all the .mp3 files installed on all the disks on a computer:

```
"SELECT * FROM CIM_DataFile WHERE Extension = 'MP3'"
```

To achieve the same result using the **FileSystemObject**, you must write a script that:

1. Returns a list of all the disk drives on the computer.
2. Verifies that each disk drive is ready for use.
3. Recursively searches each drive in order to locate all the folders and subfolders.
4. Recursively searches each folder and subfolder to locate all the files.
5. Checks each extension to see whether the file is an .mp3 file.
6. Keeps track of each .mp3 file.

## Managing Files and Folders Using the Windows Shell Object

The Windows operating system features another **COM** object, the **Shell** object that includes a number of properties and methods useful in managing file systems. Because the **Shell** object offers capabilities not available using either the **FileSystemObject** or **WMI**, you should also consider it when writing scripts for file system management. The **Shell** is the portion of the Windows operating system tasked with managing and providing access to such things as:

- Files and folders
- Network printers
- Other networked computers
- Control Panel applications
- The Recycle Bin

The **Shell** namespace provides a way to manage these objects in a tree-structured hierarchy. At the top of this hierarchy is the Desktop; directly below the Desktop are virtual folders such as My Computer, My Network Places, and Recycle Bin. Within each of these virtual folders are other items (such as files, folders, and printers) that can also be managed using the **Shell**. If you start Windows Explorer, you see a visual representation of the Shell

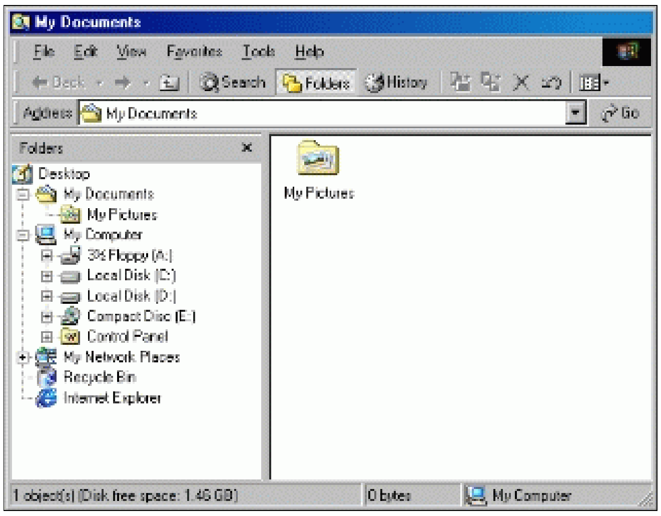


**Figure 8 The shell Namespace**

The Shell itself is composed largely of a series of **COM** objects, many of which can be accessed using **VBScript**. Included among these **COM** objects are folders. Within the Windows operating system, folders are individual **COM** objects that possess:

- Properties, such as a size and a creation date.
- Items (typically files stored within the folder).
- Methods (known as verbs), which represent actions such as Copy and Delete - that can be carried out on the folder.

Folder objects - and all the properties, items, and methods belonging to those objects - can be accessed through the **Shell** object. The **Shell** object provides a way to programmatically reproduce all the features found in the **Windows Shell**. This means that file system management tasks - which typically involve working with files and folders - carried out using the **Shell** object.

Scripting the **Shell** object is not as intuitive as scripting with **WMI** or the **FileSystemObject**. For example, to bind to a file using a **Shell** object script, you must:

1. Create an instance of the Shell object.
2. Create an instance of a Folder object.
3. Create a collection of items in the folder.
4. Iterate through the collection until you find the desired file.

This is considerably more complicated than using the **FileSystemObject** or **WMI**. On the other hand, the **Shell** object does offer a number of capabilities not found in either **WMI** or the **FileSystemObject**, including the ability to:

- Retrieve extended properties for a file or folder (for example, the artist, album title, and track number for an audio file).
- Display a progress dialog box while copying or moving folders.
- Retrieve the locations of all the special folders on a computer.
- Carry out any of the commands found on the shortcut menu when you right-click a file or folder.

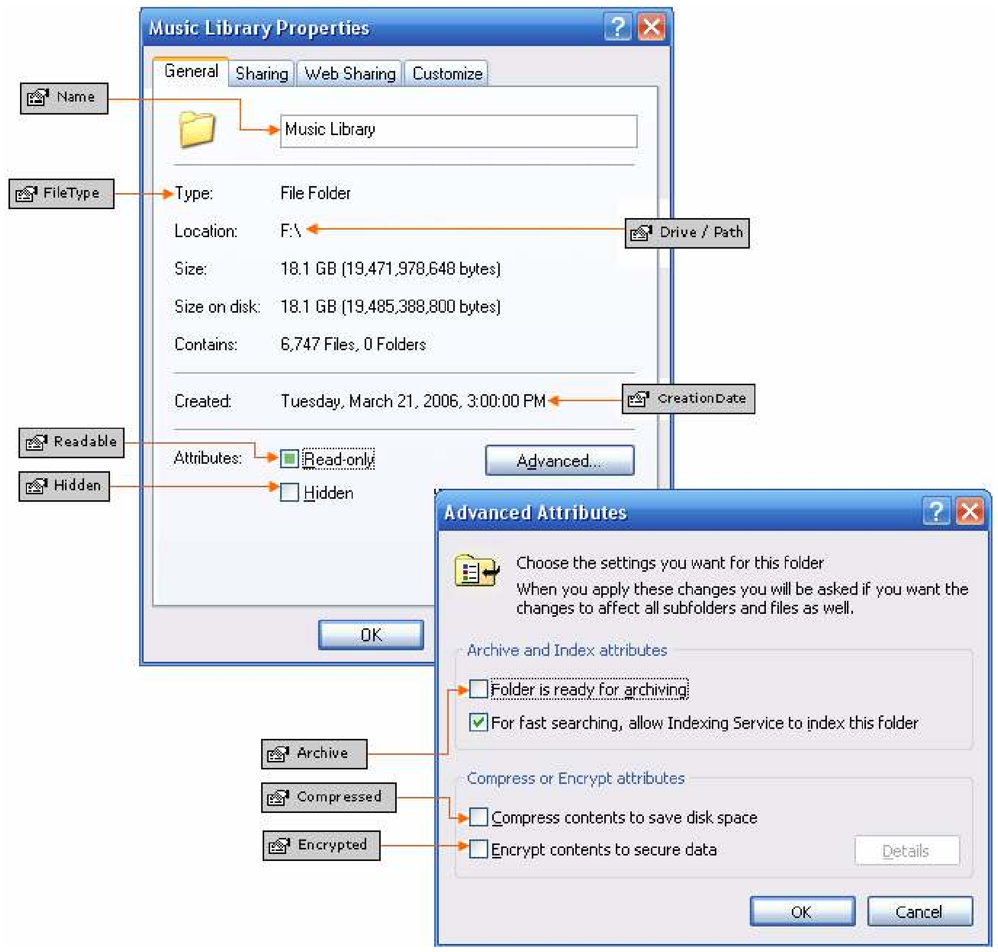
## Folders and Folders Object

Folders are file system objects designed to contain other file system objects. This does not mean that all folders are alike, however. Instead, folders can vary considerably. Some folders are operating system folders, which generally should not be modified by a script. Some folders are read-only, which means that users can access the contents of that folder but cannot add to, delete from, or modify those contents. Some folders are compressed for optimal storage, while others are hidden and not visible to users.

## Win32_Directory Class

**WMI** uses the **Win32_Directory** class to manage folders. Significantly, the properties and methods available in this class are identical to the properties and methods available in the **CIM_DataFile** class, the class used to manage files. This means that after you have learned how to manage folders using **WMI**, you will, without any extra work, also know how to manage files.

In addition, the **Win32_Directory** and **CIM_DataFile** classes share the same set of methods.



Music Library

Type: File Folder

Location: F:\

Size: 18.1 GB (19,471,978,648 bytes)

Size on disk: 18.1 GB (19,485,388,800 bytes)

Contains: 6,747 Files, 0 Folders

Created: Tuesday, March 21, 2006, 3:00:00 PM

Attributes: 🟩 Read-only ☐ Hidden

Choose the settings you want for this folder

When you apply these changes you will be asked if you want the changes to affect all subfolders and files as well.

Archive and Index attributes

☐ Folder is ready for archiving

☑ For fast searching, allow Indexing Service to index this folder

Compress or Encrypt attributes

☐ Compress contents to save disk space

☐ Encrypt contents to secure data

**Figure 9 Win32_Directory and Windows Explorer**

## Win32_Directory.AccessMask Property

### Description

List of access rights to the given file or directory held by the user or group on whose behalf the instance is returned. This property is only supported under Windows NT and Windows 2000. On Windows 98 and on Windows NT and Windows 2000 FAT volumes, the FULL_ACCESS value is returned instead, which indicates no security has been set on the object.

**Data Type**

Numeric (uint32)

**Possible Values**

| Constant | Value | Description |
|---|---|---|
| FILE_READ_DATA | &H0 | Grants the right to read data from the file. |
| FILE_LIST_DIRECTORY | &H0 | Grants the right to list the contents of the directory. |

| | | |
|---|---|---|
| FILE_WRITE_DATA<br>FILE_ADD_FILE | &H1<br>&H1 | Grants the right to write data to the file.<br>Grants the right to create a file in the directory. |
| FILE_APPEND_DATA | &H4 | Grants the right to append data to the file. |
| FILE_ADD_SUBDIRECTORY | &H4 | Grants the right to create a subdirectory. |
| FILE_READ_EA | &H8 | Grants the right to read extended attributes. |
| FILE_WRITE_EA | &H10 | Grants the right to write extended attributes. |
| FILE_EXECUTE | &H20 | Grants the right to execute a file. |
| FILE_TRAVERSE | &H20 | The directory can be traversed. |
| FILE_DELETE_CHILD | &H40 | Grants the right to delete a directory and all the files it contains (its children), even if the files are read-only. |
| FILE_READ_ATTRIBUTES | &H80 | Grants the right to read file attributes. |
| FILE_WRITE_ATTRIBUTES | &H100 | Grants the right to change file attributes. |
| DELETE | &H10000 | Grants delete access. |
| READ_CONTROL | &H20000 | Grants read access to the security descriptor and owner. |
| WRITE_DAC | &H40000 | Grants write access to the discretionary ACL. |
| WRITE_OWNER | &H80000 | Assigns the write owner. |
| SYNCHRONIZE | &H100000 | Synchronizes access and allows a process to wait for an object to enter the signaled state. |

**Table 1 Access Mask and Permissions Values**

## Win32_Directory.Archive Property

**Description**

The archive bit is used by backup programs to identify files that should be backed up.

**Note**

- Data Type is Boolean
- If **True**, the file should be archived.

## Win32_Directory.Compressed Property

**Description**

**WMI** recognizes folders compressed using **WMI** itself or using the graphical user interface; it does not, however, recognize .ZIP files as being compressed.

**Note**

- Data Type is Boolean
- If **True**, the file is compressed.

## Win32_Directory.CompressionMethod Property

**Description**

Algorithm or tool used to compress the logical file. If it is not possible (or not desired) to describe the compression scheme (perhaps because it is not known), use the following words: "Unknown" to represent that it is not known whether the logical file is compressed; "Compressed" to represent that the file is compressed but either its compression scheme is not known or not disclosed; and "Not Compressed" to represent that the logical file is not compressed.

**Note**

- Data Type is String
- Method used to compress the file system object. Often reported simply as "Compressed."

## Win32_Directory.CreationDate Property

**Description**

Date that the file system object was created.

**Note**

- Data type is DateTime
- To Convert to a **VBScript** Date sub-type use **WMI** WbemScripting.SWbemDateTime object

## Win32_Directory.Drive Property

**Description**

Drive letter (including colon) of the file. ("c:")

**Note**

- Data Type is String

## Win32_Directory.EightDotThreeFileName Property

**Description**

MS-DOS®-compatible name for the folder.
For example, **EightDotThreeFileName** for the folder C:\Program Files might be C:\Progra~1.

**Note**

- Data Type is String

## Win32_Directory.Encrypted Property

**Description**

Boolean value indicating whether or not the folder has been encrypted.

**Note**

- Data Type is Boolean

## Win32_Directory.EncryptionMethod Property

**Description**

Algorithm or tool used to encrypt the logical file. If it is not possible (or not desired) to describe the encryption scheme (perhaps for security reasons), use the following words: "Unknown" to represent that it is not known whether the logical file is encrypted; "Encrypted" to represent that the file is encrypted but either its encryption scheme is not known or not disclosed; and "Not Encrypted" to represent that the logical file is not encrypted.

**Note**

- Data Type is String

## Win32_Directory.Extension Property

**Description**

File extension (without the dot). Examples: "txt", "mof", "mdb"

**Note**

- Data Type is String

## Win32_Directory.FileName Property

**Description**

File name (without the dot or extension) of the file. Example: "autoexec"

**Note**

- Data Type is String

## Win32_Directory.FileSize Property

**Description**

Size of the file system object, in bytes. Although folders possess a *FileSize* property, the value 0 is always returned.

**Note**

- Data Type is Numeric (uint64)
- To determine the size of a folder, use the **FileSystemObject** or add up the size of all the files stored in the folder.

## Win32_Directory.FSName Property

**Description**

Type of file system (NTFS, FAT, FAT32) installed on the drive where the file or folder is located.

**Note**

- Data Type is String

## Win32_Directory.Hidden Property

**Description**

Boolean value indicating whether the file system object is hidden.

**Note**

- Data Type is Boolean
- If **True**, the file is hidden.

## Win32_Directory.InUseCount Property

**Description**

Number of 'file opens' that are currently active against the file.

**Note**

- Data Type is Numeric (uint64)

## Win32_Directory.LastAccessed Property

**Description**

Date that the object was last accessed.

**Note**

- Data Type is DateTime
- To Convert to a **VBScript** Date sub-type use WMI WbemScripting.SWbemDateTime object

## Win32_Directory.LastModified Property

**Description**

Date that the object was last modified.

**Note**

- Data Type is DateTime
- To Convert to a **VBScript** Date sub-type use WMI WbemScripting.SWbemDateTime object

## Win32_Directory.Name Property

**Description**

Full path name of the file system object. For example: c:\windows\system32\wbem.

**Note**

- Data Type is String

## Win32_Directory.Path Property

**Description**

Path of the file. This includes leading and trailing backslashes.
Example: "\windows\system\"

**Note**

- Data Type is String

## Win32_Directory.Readable Property

**Description**

Boolean value indicating whether you can read items in the folder.

**Note**

- Date Type is Boolean

## Win32_Directory.Status Property

**Description**

Current status of the object. Various operational and non-operational statuses can be defined.

**Note**

Data Type is String

## Win32_Directory.System Property

**Description**

Boolean value indicating whether the object is a system folder.

**Note**

- Date Type is Boolean

## Win32_Directory.Writeable Property

**Description**

Boolean value indicating whether you can write to the folder.

**Note**

- Date Type is Boolean

## Win32_Directory.TakeOwnerShip Method

**Description**

The **TakeOwnerShip** method obtains ownership of the logical file specified in the object path. If the logical file is actually a directory, then **TakeOwnerShip** acts recursively, taking ownership of all the files and subdirectories the directory contains.

**Syntax**

```
object.TakeOwnerShip( )
```

**Note**

- Returns a value of 0 (zero) if the request was successful, and any other number to indicate an error.

**Example**

```
Dim oOutParams
Set oOutParams = GetObject("winmgmts:").ExecMethod _
      ("Win32_Directory.Name='C:\\wmi_demo'", "TakeOwnerShip")
Msgbox oOutParams.ReturnValue
```

## Win32_Directory.Copy Method

**Description**

The **Copy** method copies the logical directory entry file or directory specified in the object path to the location specified by the input parameter. A copy is not supported if overwriting an existing logical file is required.

**Syntax**

```
object.Copy( FileName )
```

**Note**

Returns a value of 0 (zero) if the file was successfully renamed, and any other number to indicate an error.

**Arguments**

| Parameter | Description |
|---|---|
| *FileName* | Fully-qualified new name of or directory. |

## Win32_Directory.Rename Method

**Description**

The **Rename** method renames the directory entry file specified in the object path. A rename is not supported if the destination is on another drive or if overwriting an existing logical file is required.

**Syntax**

```
object.Rename( FileName )
```

**Note**

Returns a value of 0 (zero) if the file was successfully renamed, and any other number to indicate an error.

**Arguments**

| Parameter | Description |
|---|---|
| *FileName* | Fully-qualified new name of or directory. |

**Example**

```
Dim oCollection, oItem
Dim nRes
Set oCollection = GetObject("winmgmts:").ExecQuery _
      ("Select * from Win32_Directory Where Name='c:\\wmi_demo'")
If oCollection.Count = 1 Then
   For Each oItem in oCollection
      MsgBox "Folder data before rename", _oItem.getObjectText_
      nRes = objItem.Rename("c:\wmi_demo1")
      If nRes > 0 Then
         Msgbox "Failed: error #" & nRes
      End If
    Next
End If
```

## Win32_Directory.Delete Method

**Description**

The **Delete** method will delete the logical file (or directory) specified in the object path.

**Syntax**

```
object.Delete( )
```

**Note**

- Returns a value of 0 (zero) if the file was successfully deleted, and any other number to indicate an error.

## Win32_Directory.Compress Method

**Description**

The **Compress** method compresses the logical directory entry file (or directory) specified in the object path.

**Syntax**

```
object.Compress( )
```

**Note**

- Returns a value of 0 (zero) if the file was successfully compressed, and any other number to indicate an error.

## Win32_Directory.Uncompress Method

**Description**

The **Uncompress** method uncompresses the logical directory entry file (or directory) specified in the object path.

**Syntax**

```
object.Uncompress( )
```

**Note**

- Returns a value of 0 (zero) if the file was successfully compressed, and any other number to indicate an error.

## Win32_Directory.GetEffectivePermission Method

**Description**

The **GetEffectivePermission** method determines whether the user has all required permissions specified in the Permissions parameter for the **Win32_Directory** object, directory, and share where the directory entry file is located, if the file or directory are on a share.

**Syntax**

```
object.GetEffectivePermission( Permissions )
```

**Arguments**

| Parameter | Description |
|---|---|
| *Permissions* | A Number(uint32) specifying the Bitmap of permissions that the caller can inquire about. To see the list of the permission parameters see Table 1 on page 43 |

**Note**

- Returns **true** when the caller has the specified permissions, and **false** when the caller does not have the specified permissions.

## Win32_Directory Methods

For more information on **Win32_Directory** class methods you can look in the following link:
http://msdn.microsoft.com/library/en-us/wmisdk/wmi/win32_directory.asp

## The FileSystemObject Object

As the name implies, the **FileSystemObject** (**FSO**) is designed to help you manage the file system. The **FileSystemObject** allows you to retrieve information about essential file system elements, including disk drives, folders, and files; it also includes methods that allow you to perform common administrative tasks, such as copying, deleting, and moving files and folders. In addition, the **FileSystemObject** enables you to read from and write to text files.

The **FileSystemObject** simplifies the task of dealing with any type of file input and output and for dealing with the system file structure itself. Rather than resorting to complex calls to the **Win32 API**. this object allows the developer to easily handle files and navigate the underlying directory structures. This is especially useful for those developers or administrators who are creating scripts that are used for system administration or maintenance.

## The FSO Object Model

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

To access the File System object model, you must first create an instance of the **FileSystemObject** object, the only externally creatable object in the model. From there, you can navigate through the object model.

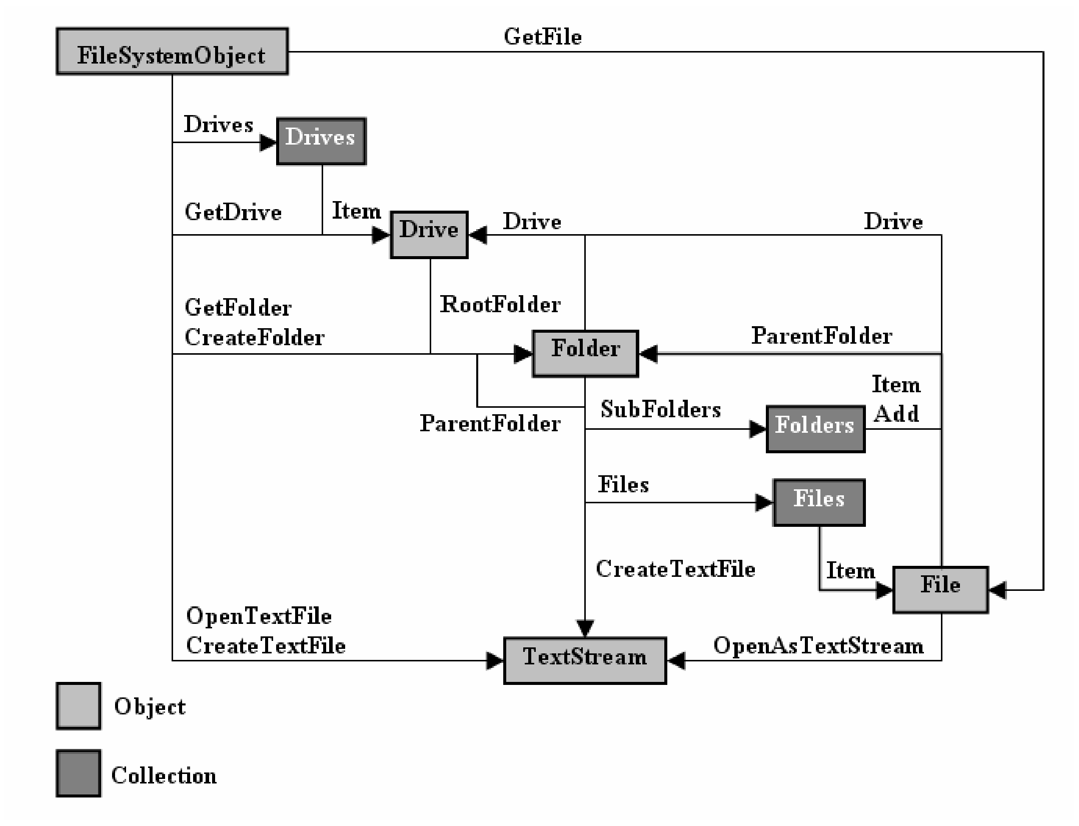


**Figure 10 The FileSystemObject object model**

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

Contains methods and properties that allow you to create, delete, or move folders. Also allows you to query the system for folder names, paths, and various other properties.

- Folders – Collection
  Provides a list of all the folders within a Folder.
- TextStream – Object
  Allows you to read and write text files.

## Programming the FileSystemObject

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

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

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

## The FileSystemObject Properties and methods

The **FileSystemObject** object is at the top level of the File System object model and is the only externally creatable object in the hierarchy; that is, it's the only object you can create using the **CreateObject** function or the host object model's object creation facilities. For example, the following code instantiates a **FileSystemObject** object named fso:

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

The **FileSystemObject** object represents the host computer's file system as a whole. Its members allow you to begin navigation into the file system, as well as to access a variety of common file system services. For information about the **FileSystemObject** object's properties and methods, see the entry for each property and method.

### FileSystemObject.Drives Property

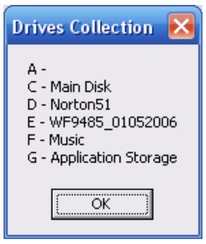
**Description**

**Drives** is a read-only property that returns the **Drives** collection; each member of the collection is a **Drive** object, representing a single drive available on the system. Using the collection object returned by the **Drives** property, you can iterate all the drives on the system using a **For...Next** loop, or you can retrieve an individual **Drive** object, which represents one drive on the system, by using the **Drives** collection's Item method.

**Example**

```
Function ShowDriveList()
   Dim oFso, oDrive, oDrvColl
   Dim sTmp, sShareVol

   Set oFso = CreateObject("Scripting.FileSystemObject")
   Set oDrvColl = oFso.Drives
   For Each oDrive in oDrvColl
      sShareVol = Empty
      sTmp = sTmp & oDrive.DriveLetter & " - "
      If oDrive.DriveType = 3 Then
         sShareVol = oDrive.ShareName
      ElseIf oDrive.IsReady Then
         sShareVol = oDrive.VolumeName
      End If
      sTmp = sTmp & sShareVol & vbNewLine
   Next
   ShowDriveList = sTmp
End Function
```



```
A -
C - Main Disk
D - Norton51
E - WF9485_01052006
F - Music
G - Application Storage
```

**Figure 11 - Drives Collection**

## FileSystemObject.BuildPath Method

### Description

The **BuildPath** method creates a single string representing a path and filename or simply a path by concatenating the path parameter with the folder or filename, adding, where required, the correct path separator for the host system.

### Syntax

```
object.BuildPath(path, name)
```

### Arguments

| Parameter | Description |
|---|---|
| *path* | Required. Existing path to which name is appended. *Path* can be absolute or relative and need not specify an existing folder. |
| *name* | Required. Name being appended to the existing path. |

### Note

- *Path* can be an absolute or relative path and doesn't have to include the drive name.
- Neither *Path* nor *Name* has to currently exist.

**Tip**

- **BuildPath** is really a string concatenation method rather than a file system method; it does not check the validity of the new folder or filename. If you intend that the method's return value be a path, you should check it by passing it to the **FolderExists** method; if you intend that the method's return value be a path and filename, you should verify it by passing it to the **FileExists** method.
- The only advantage to using the **BuildPath** function as opposed to concatenating two strings manually is that the function selects the correct path separator.

## FileSystemObject.CopyFile Method

**Description**

The **CopyFile** method copies a file or files from one folder to another.

**Syntax**

```
object.CopyFile( source, destination[, overwrite] )
```

**Arguments**

| Parameter | Description |
|---|---|
| *source* | Required. Character string file specification, which can include wildcard characters, for one or more files to be copied. |
| *destination* | Required. Character string destination where the file or files from source are to be copied. Wildcard characters are not allowed. |
| *overwrite* | Optional. **Boolean** value that indicates if existing files are to be overwritten. If **true**, files are overwritten; if **false**, they are not. The default is **true**. Note that **CopyFile** will fail if destination has the read-only attribute set, regardless of the value of *overwrite*. |

**Note**

- The default value for *overwrite* is **True**.
- The *source* path can be relative or absolute.
- The *source* filename can contain wildcard characters; the *source* path can't.
- Wildcard characters can't be included in *destination*.

**Tip**

- If the destination path or file is read-only, the **CopyFile** method fails, regardless of the value of *overwrite* and generates runtime error 70, "Permission Denied."
- If *overwrite* is set to **False** and the file exists in Destination, a trappable error, runtime error 58, "File Already Exists", is generated.
- If an error occurs while copying more than one file, the **CopyFile** method exits immediately, thereby leaving the rest of the files uncopied. There is no rollback facility to undo copies made prior to the error.
- Both *source* and *destination* can include relative paths, that is, paths that are relative to the current folder. The current folder is the folder in which the script is stored. The symbol to indicate the parent of the current folder is (..); the symbol to indicate the current folder is (.).
- *source* must include an explicit filename. For instance, under **DOS**, you could copy all of the files in a directory with a command in the format of:

```
Copy c:\data c:\backup
or:
Copy c:\data\ c:\backup
```

- Which would copy all the files from the C:\data directory to C:\bckup. The *source* argument cannot take any of these forms; instead, you must include some filename component. For example, to copy all of the files from C:\data, the **CopyFile** statement would take the form:

```
oFso.CopyFile "C:\data\*.*", "C:\backup"
```

- To specify multiple files, the Source argument can include the * and ? wildcard characters. Both are legacies from DOS. * matches any characters in a filename that follow those characters that are explicitly specified. For instance, a Source argument of File* matches File01.txt, File001.txt, and File.txt, since all three filenames begin with the string "File"; the remainder of the filename is ignored. ? is a wildcard that ignores a single character in a filename comparison. For instance, a Source argument of Fil?01.txt copies File01.txt and Fil_01.txt, since the fourth character of the filename is ignored in the comparison.
- If you want the source and the destination directories to be the same, you can copy only a single file at a time, since Destination does not accept wildcard characters.
- If the path specified in Destination does not exist, the method does not create it. Instead, it generates runtime error 76, "Path not found."
- If the user has adequate rights, the source or destination can be a network path or share name. For example:

```
oFso.CopyFile "c:\Rootone\*.*", "\\NTSERV1\d$\RootTwo\"
oFso.CopyFile "\\NTSERV1\RootTest\test.txt", "c:\RootOne"
```

- The **CopyFile** method copies a file or files stored in a particular folder. If the folder itself has subfolders containing files, the method doesn't copy these; use the **CopyFolder** method.
- The **CopyFile** method differs from the **Copy** method of the **File** object in two ways:
  - You can copy any file anywhere in a file system without having to first instantiate it as a **File** object.

- You can copy multiple files in a single operation, rather than copying only the file represented by the **File** object.

## FileSystemObject.CopyFolder Method

**Description**

The **CopyFolder** method copies the contents of one or more folders, including their subfolders, to another location.

**Syntax**

```
object.CopyFolder( source, destination[, overwrite] )
```

**Arguments**

| Parameter | Description |
|---|---|
| *source* | Required. Character string folder specification, which can include wildcard characters, for one or more folders to be copied. |
| *destination* | Required. Character string destination where the folder and subfolders from source are to be copied. Wildcard characters are not allowed. |
| *overwrite* | Optional. Boolean value that indicates if existing folders are to be overwritten. If **true**, files are overwritten; if false, they are not. The default is **true**. |

**Note**

- *source* must end with either a wildcard character or no path separator. If it ends with a wildcard character, all matching subfolders and their contents will be copied. Wildcard characters can be used in *source* only for the last component.
- Wildcard characters can't be used in *destination*.
- All subfolders and files contained within the *source* folder are copied to *destination* unless disallowed by the wildcard characters. That is, the **CopyFolder** method is recursive.
- If *destination* ends with a path separator or *source* ends with a wildcard, **CopyFolder** assumes that the folder stated in *source* exists in *destination* or should otherwise be created. For example, given the following folder structure:

```
C:\
   Rootone
      SubFolder1
      SubFolder2
   RootTwo
```

The code oFso.**CopyFolder** "c:\Rootone\*", "C:\RootTwo" produces this folder structure:

```
:\
   Rootone
      SubFolder1
      SubFolder2
   RootTwo
```

```
        SubFolder1
        SubFolder2
```

The code oFso.**CopyFolder** "c:\Rootone", "C:\RootTwo\" produces this folder structure:

```
C:\
   Rootone
      SubFolder1
      SubFolder2
   RootTwo
      Rootone
         SubFolder1
         SubFolder2
```

**Tip**

- If the destination path or any of the files contained in *destination* are set to read-only, the **CopyFolder** method fails, regardless of the value of *overwrite*.
- If *overwrite* is set to **False**, and the source folder or any of the files contained in *source* exists in *destination*, runtime error 58, "File Already Exists," is generated.
- If an error occurs while copying more than one file or folder, the **CopyFolder** function exits immediately, leaving the rest of the folders or files uncopied. There is no rollback facility to undo the copies prior to the error.
- If the user has adequate rights, both the source or destination can be a network path or share name. For example:

```
oFso.CopyFolder "c:\Rootone", "\\NTSERV1\d$\RootTwo\"
oFso.CopyFolder "\\NTSERV1\RootTest", "c:\RootOne"
```

## FileSystemObject.CreateFolder Method

**Description**

The **CreateFolder** method creates a single new folder in the path specified and returns its **Folder** object.

**Syntax**

```
object.CreateFolder( foldername )
```

**Arguments**

| Parameter | Description |
|---|---|
| *foldername* | Required. String expression that identifies the folder to create. |

**Note**

- Wildcard characters aren't allowed in *foldername*.
- *foldername* can be a relative or absolute path.
- If no path is specified in *foldername*, the current drive and directory are used.

- If the last folder in *foldername* already exists, the method generates runtime error, "File already exists."

**Tip**

- If *foldername* is read-only, the **CreateFolder** method fails.
- If *foldername* already exists, the method generates runtime error 58, "File already exists."
- If the user has adequate rights, *foldername* can be a network path or share name. For example:

```
oFso.CreateFolder "\\NTSERV1\d$\RootTwo\newFolder"
oFso.CreateFolder "\\NTSERV1\RootTest\newFolder"
```

You must use the **Set** statement to assign the **Folder** object to an object variable. For example:

```
Dim oFso, oFolder
Set oFso = CreateObject("Scripting.FileSystemObject")
Set oFolder = oFso.CreateFolder("MyFolder")
```

## FileSystemObject.CreateTextFile Method

**Description**

The **CreateTextFile** method creates a new file and returns its **TextStream** object.

**Syntax**

```
object.CreateTextFile(filename[, overwrite[, unicode]])
```

**Arguments**

| Parameter | Description |
|---|---|
| *filename* | Required. String expression that identifies the file to create. |
| *overwrite* | Optional. Boolean value that indicates whether you can overwrite an existing file. The value is **true** if the file can be overwritten, **false** if it can't be overwritten. If omitted, existing files are not overwritten. |
| *unicode* | Optional. Boolean value that indicates whether the file is created as a Unicode or ASCII file. The value is true if the file is created as a Unicode file, false if it's created as an ASCII file. If omitted, an ASCII file is assumed. |

**Note**

- Wildcard characters aren't allowed in *filename*.
- *filename* can be a relative or absolute path.
- If no path is specified in filename, the script's current drive and directory are used. If no drive is specified in *filename*, the script's current drive is used.
- If the path specified in *filename* doesn't exist, the method fails. To prevent this error, you can use the **FileSystemObject** object's **FolderExists** method to insure that the path is valid.
- The default value for *overwrite* is **False**.

- If Unicode is set to **True**, the file is created in Unicode; otherwise, it's created as an ASCII text file. The default value for Unicode is **False**.

**Tip**

- The newly created text file is automatically opened only for writing. If you subsequently wish to read from the file, you must first close it and reopen it in read mode.
- If the path referred to in *filename* is set to read-only, the **CreateTextFile** method fails regardless of the value of *overwrite*.
- If the user has adequate rights, *filename* can contain a network path or share name. For example:

```
oFso.CreateTextFile "\\NTSERV1\RootTest\myFile.doc"
```

- You must use the **Set** statement to assign the **TextStream** object to your local object variable.
- The **CreateTextFile** method of the **Folder** object is identical in operation to that of the **FileSystemObject** object.

## FileSystemObject.DriveExists Method

**Description**

The **DriveExist** method determines whether a given drive (of any type) exists on the local machine or on the network. The method returns **True** if the drive exists or is connected to the machine, and returns **False** if not.

**Syntax**

```
object.DriveExist( drivespec )
```

**Arguments**

| Parameter | Description |
|---|---|
| *drivespec* | Required. A drive letter or a complete path specification. |

**Note**

- If *drivespec* is a Windows drive letter, it doesn't have to include the colon. For example, "C" works just as well as "C:".
- Returns **True** if the drive exists or is connected to the machine, and returns **False** if not.

**Tip**

- **DriveExists** doesn't note the current state of removable media drives; for this, you must use the **IsReady** property of the **Drive** object representing the given drive.
- If the user has adequate rights, **DriveSpec** can be a network path or share name. For example:

```
If oFso.DriveExists("\\NTSERV1\d$") Then
```

This method is ideal for detecting any current drive around the network before calling a function in a remote **ActiveX** server located on that drive.

## FileSystemObject.FileExists Method

**Description**

The **FileExist** method determines if a given file exists.

**Syntax**

```
object.FileExist(filespec )
```

**Arguments**

| Parameter | Description |
|---|---|
| *filespec* | Required. The name of the file whose existence is to be determined. A complete path specification (either absolute or relative) must be provided if the file isn't expected to exist in the current folder. |

**Note**

- Returns **True** if the file exists or is connected to the machine, and returns **False** if not.
- *filespec* can't contain wildcard characters.
- *filespec* can include either an absolute or a relative path, that is, a path that is relative to the current folder. The current folder is the folder in which the script is running, or the folder specified in the "Start in" text box of the shortcut used to launch the script. The symbol to indicate the parent of the current folder is (..); the symbol to indicate the current folder is (.). If *filespec* does not include a path, the current folder is used.

**Tip**

- If the user has adequate rights, *filespec* can be a network path or share name. For example:

```
If oFso.FileExists("\\TestPath\Test.txt") Then
```

## FileSystemObject.FolderExists Method

**Description**

The **FolderExist** method determines whether a given folder exists; the method returns **True** if the Folder exists, and returns **False** if not.

**Syntax**

```
object.FolderExist(folderspec )
```

**Arguments**

| Parameter | Description |
|---|---|
| *folderspec* | Required. The name of the folder whose existence is to be determined. A complete path specification (either absolute or relative) must be provided if the folder isn't expected to exist in the current folder. |

**Note**

- *folderspec* can't contain wildcard characters.
- *folderspec* cannot include a filename as well as a path. In other words, the entire *folderspec* string can only include drive and path information.
- If *folderspec* does not include a drive specification, the current drive is assumed.
- *folderspec* is interpreted as an absolute path if it begins with a drive name and a path separator, and it is interpreted as an absolute path on the current drive if it begins with a path separator. Otherwise, it is interpreted as a relative path.

**Tip**

- If the user has adequate rights, *folderspec* can be a network path or share name. For example:

```
If oFso.FolderExists("\\NTSERV1\d$\TestPath\") Then
```

Among its string manipulation methods, the Scripting Runtime library lacks one that will extract a complete path from a path and filename.

## FileSystemObject.GetAbsolutePathName Method

**Description**

The **GetAbsolutePathName** method returns a complete and unambiguous path from a provided path specification.

**Syntax**

```
object.GetAbsolutePathName(pathspec )
```

**Arguments**

| Parameter | Description |
|---|---|
| *pathspec* | Required. Path specification to change to a complete and unambiguous path. |

**Note**

- (.) returns the drive letter and complete path of the current folder.
- (..) returns the drive letter and path of the parent of the current folder.
- If *pathspec* is simply a filename without a path, the method concatenates the complete path to the current directory with the filename. For example, if the current folder is C:\Documents\MyScripts, then the method call:

```
sFileName = oFso.GetAbsolutePathName("MyFile.txt")
```

- produces the string "C:\Documents\MyScripts\MyFile.txt".
- All relative pathnames are assumed to originate at the current folder. This means, for example, that (.) returns the drive letter and complete path of the current folder, and that (..) returns the drive letter and path of the parent of the current folder.
- If a drive isn't explicitly provided as part of *pathspec*, it's assumed to be the current drive.
- Wildcard characters can be included in *pathspec* at any point.

**Tip**

- An absolute path provides a complete route from the root directory of a particular drive to a particular folder or file. In contrast, a relative path describes a route from the current folder to a particular folder or file.
- For mapped network drives and shares, the method doesn't return the full network address. Rather, it returns the fully qualified local path and locally issued drive letter.
- The **GetAbsolutePathName** method is really a string conversion and concatenation method, rather than a file system method. It merely returns a string, but doesn't verify that a given file or folder exists in the path specified.

## FileSystemObject.GetBaseName Method

**Description**

The **GetBaseName** method returns a complete and unambiguous path from a provided path specification.

**Syntax**

```
object.GetBaseName(path )
```

**Arguments**

| Parameter | Description |
|---|---|
| *path* | Required. The path specification for the component whose base name is to be returned. |

**Note**

- The file extension of the last element in *path* isn't included in the returned string.

**Tip**

- **GetBaseName** doesn't verify that a given file or folder exists in *path*.
- In stripping the "file extension" and returning the base name of *path*, **GetBaseName** has no intelligence. That is, it doesn't know whether the last component of *path* is a path or a filename. If the last component includes one or more dots, it simply removes the last one, along with any following text. Hence, **GetBaseName** returns a null string for a *path* of (.) and it returns (.) for a *path* of (..). It is, in other words, really a string manipulation function, rather than a file function.

## FileSystemObject.GetDrive Method

**Description**

The **GetDrive** method obtains a reference to a **Drive** object for the specified drive.

**Syntax**

```
object.GetDrive(drivespec )
```

**Arguments**

| **Parameter** | **Description** |
|---|---|
| *drivespec* | Required. The *drivespec* argument can be a drive letter (c), a drive letter with a colon appended (c:), a drive letter with a colon and path separator appended (c:\), or any network share specification (\\computer2\share1). |

**Note**

- If *drivespec* is a local drive or the letter of a mapped drive, it can consist of only the drive letter (e.g., "C"), the drive letter with a colon ("C:"), or the drive letter and path to the root directory (e.g., "C:\") without generating a runtime error.
- If *drivespec* is a share name or network path, **GetDrive** ensures that it exists as part of the process of creating the Drive object; if it doesn't, the method generates runtime error 76, "Path not found."
- If the specified drive isn't connected or doesn't exist, runtime error 67, "Device unavailable," occurs.

**Tip**

- Individual **drive** objects can be retrieved from the **Drives** collection by using the **Drives** property. This is most useful, though, if you want to enumerate the drives available on a system. In contrast, the **GetDrive** method provides direct access to a particular **Drive** object.
- If you are deriving the *drivespec* string from a path, you should first use **GetAbsolutePathName** to insure that a drive is present as part of the path. Then you should use **FolderExists** to verify that the path is valid before calling **GetDriveName** to extract the drive from the fully qualified path. For example:

```
Dim oFso, oDrive
Set oFso = CreateObject("Scripting.FileSystemObject")
sPath = oFso.GetAbsolutePathName(sPath)
If oFso.FolderExists(sPath) Then
   Set oDrive = oFso.GetDrive(oFso.GetDriveName(sPath))
End If
```

- If *drivespec* is a network drive or share, you should use the **DriveExists** method to confirm the required drive is available prior to calling the **GetDrive** method.
- You must use the **Set** statement to assign the **Drive** object to a local object variable.

## FileSystemObject.GetDriveName Method

**Description**

The **GetDriveName** method returns the drive name of a given path.

**Syntax**

```
object.GetDriveName(path )
```

**Arguments**

| Parameter | Description |
|---|---|
| *path* | Required. The *drivespec* argument can be a drive letter (c), a drive letter with a colon appended (c:), a drive letter with a colon and path separator appended (c:\), or any network share specification (\\computer2\share1). |

**Note**

- If the drive name can't be determined from the given path, a zero-length string (" ") is returned.

**Tip**

- For local and mapped drives, **GetDriveName** appears to look for the colon as a part of the drive's name to determine whether a drive name is present. For network drives, it appears to look for the computer name and drive name.
- **GetDriveName** is really a string-parsing method rather than a file system method. In particular, it does not verify that the drive name that it extracts from *path* actually exists on the system.
- *path* can be a network drive or share.

## FileSystemObject.GetExtensionName Method

**Description**

The **GetExtensionName** method returns the extension of the file element of a given path.

**Syntax**

```
object.GetExtensionName(path )
```

**Arguments**

| Parameter | Description |
|---|---|
| *path* | Required. The *path* specification for the component whose extension name is to be returned. |

**Note**

- If the extension in *path* can't be determined, a zero-length string (" ") is returned.

**Tip**

- **GetExtensionName** is a string parsing method rather than a file system method. It does not verify that *path* is valid, does not verify that the filename designated in *path* exists, and does not even guarantee that the value it returns is a valid file extension. In other words, **GetExtensionName** has no intelligence. It simply parses a string and returns the text that follows the last dot of the last element.

- *path* can be a network drive or share.

## FileSystemObject.GetFile Method

**Description**

The **GetFile** method returns a reference to a **File** object.

**Syntax**

```
object.GetFile( filespec )
```

**Arguments**

| Parameter | Description |
|---|---|
| *filespec* | Required. The *filespec* is the path (absolute or relative) to a specific file. |

**Note**

- *filespec* can be an absolute or a relative path.
- If *filespec* is a share name or network path, **GetFile** ensures that the drive or share exists as part of the process of creating the **File** object.
- If any part of the path in *filespec* can't be contacted or doesn't exist, an error occurs.

**Tip**

- The object returned by **GetFile** is a File object, not a **TextStream** object. A **File** object isn't an open file; the point of the **File** object is to perform methods such as copying or moving files and interrogating a file's properties. Although you can't write to or read from a **File** object, you can use the **File** object's **OpenAsTextStream** method to obtain a **TextStream** object. You can also save yourself a step by calling the **FileSystemObject** object's **OpenTextFile** method.
- You should first use **GetAbsolutePathName** to create the required *filespec* string.
- If *filespec* includes a network drive or share, you could use the **DriveExists** method to confirm that the required drive is available prior to calling the **GetFile** method.
- Since **GetFile** generates an error if the file designated in *filespec* doesn't exist, you should call the **FileExists** method before calling **GetFile**.
- You must use the **Set** statement to assign the **File** object reference to a local object variable.

## FileSystemObject.GetFileName Method

**Description**

The **GetFileName** method returns the filename element of a given path.

**Syntax**

```
object.GetFileName( pathspec )
```

**Arguments**

| Parameter | Description |
|---|---|
| *pathspec* | Required. The path (absolute or relative) to a specific file. |

**Note**

- If the filename can't be determined from the given *pathspec*, a zero-length string (" ") is returned.
- *pathspec* can be a relative or absolute reference.

**Tip**

- **GetFileName** doesn't verify that a given file exists in *pathspec*.
- *pathspec* can be a network drive or share.
- Like all the **Getx** Name methods of the **FileSystemObject** object, the **GetFileName** method is more a string manipulation routine that an object-related routine. **GetFileName** has no built-in intelligence (and, in fact, seems to have even less intelligence than usual for this set of methods); it simply assumes that the last element of the string that is not part of a drive and path specified is in fact a filename. For example, if Path is C:\Windows, the method returns the string "Windows"; if Path is C:\Windows\ (which unambiguously denotes a folder rather than a filename), the method still returns the string "Windows

## FileSystemObject.GetFileVersion Method

**Description**

The **GetFileName** method retrieves version information about the file specified in *pathspec*.

**Syntax**

```
object.GetFileVersion( pathspec )
```

**Arguments**

| Parameter | Description |
|---|---|
| *pathspec* | Required. The path (absolute or relative) to a specific file. |

**Note**

- *pathspec* should include the path as well as the name of the file. The path component can be either an absolute or a relative path to the file.
- If path information is omitted, **VBScript** attempts to find *pathspec* in the current folder.
- This function reports version information in the format: Major_Version.Minor_Version.0.Build
- If a file does not contain version information, the function returns an empty string (" ").

**Tip**

- The files that can contain version information are executable files (.exe) and dynamic link libraries (.dll).

- If you're want to replace a private executable or DLL with another, be particularly careful with version checking, since it has been a particularly serious source of error. Ensuring that the new version of the file should be installed requires that any one of the following conditions be true:
  - It has the same major and minor version but a later build number than the existing file.
  - It has the same major version but a greater minor version number than the existing file.
  - It has a higher version number than the existing file.
- It's also a good idea to copy the replaced file to a backup directory.
- If you're thinking to replace a system executable or DLL with another, it's best to use a professional installation program for this purpose.
- Although this function is listed in the type library and is actually implemented in the Scripting Runtime, no documentation for it is available in the HTML Help file.

## FileSystemObject.GetFolder Method

**Description**

The **GetFolder** method returns a reference to a **Folder** object.

**Syntax**

```
object.GetFolder( folderspec )
```

**Arguments**

| Parameter | Description |
|---|---|
| *folderspec* | Required. The *folderspec* is the path to a specific folder. |

**Note**

- *folderspec* can be an absolute or relative path.
- If *folderspec* is a share name or network path, **GetFolder** ensures that the drive or share exists as part of the process of returning the **Folder** object.
- If any part of *folderspec* doesn't exist, an error occurs.

**Tip**

- You should first use **GetAbsolutePathName** to create the required **FolderPath** string.
- If **FolderPath** includes a network drive or share, you could use the **DriveExists** method to confirm the required drive is available prior to calling the **GetFolder** method.
- Since **GetFolder** requires that **FolderPath** is the path to a valid folder, you should call the **FolderExists** method to verify that **FolderPath** exists.
- The **GetFolder** method allows you to directly obtain an object reference to a particular folder. You can also use the Item property of the **Folders** collection object for cases in which you must navigate the file system to reach a particular folder, or for those cases in which you're interested in

enumerating the subfolders belonging to a particular folder.

- You must use the **Set** statement to assign the **Folder** object reference to a local object variable.

## FileSystemObject.GetParentFolderName Method

**Description**

The **GetParentFolderName** method returns the folder name immediately preceding the last element of a given path. In other words, if *path* ends in a filename, the method returns the path to the folder containing that file. If *path* ends in a folder name, the method returns the path to that folder's parent.

**Syntax**

```
object.GetFolderParentName(path)
```

**Arguments**

| Parameter | Description |
|---|---|
| *path* | Required. The path specification for the component whose parent folder name is to be returned. |

**Note**

- If the parent folder name can't be determined from *path*, a zero-length string (" ") is returned.
- *path* can be a relative or absolute reference.

**Tip**

- **GetParentFolderName** doesn't verify that any element of Path exists.
- Path can be a network drive or share.
- **GetParentFolderName** assumes that the last element of the string that isn't part of a drive specifier is the parent folder. It makes no other check than this. As with all the **Getx** Name methods of the **FileSystemObject** object, the **GetParentFolderName** method is more a string parsing and manipulation routine than an object-related routine.

## FileSystemObject.GetSpecialFolder Method

**Description**

The **GetParentFolderName** method returns a reference to a **Folder** object of one of the three special system folders: System, Temporary, and Windows.

**Syntax**

```
object.GetSpecialFolder( folderspec )
```

**Arguments**

| Parameter | Description |
|---|---|
| *folderspec* | Required. The name of the special folder to be returned. Can be any of the constants shown in the Settings section. |

### Settings

| Constant | Val | Description |
|---|---|---|
| WindowsFolder | 0 | Contains files installed by the Windows operating system. |
| SystemFolder | 1 | Contains libraries, fonts, and device drivers. |
| TemporaryFolder | 2 | Used to store temporary files. Its path is found in the TMP environment variable. |

**Note**

- As the previous table shows, the Scripting Runtime type library defines constants of the **SpecialFolderConst** enumeration that can be used in place of their numeric equivalents. You can use them in your scripts in either of two ways. You can define the constants yourself by adding the following code to your script:

```
Const WindowsFolder = 0
Const SystemFolder = 1
Const TemporaryFolder = 2
```

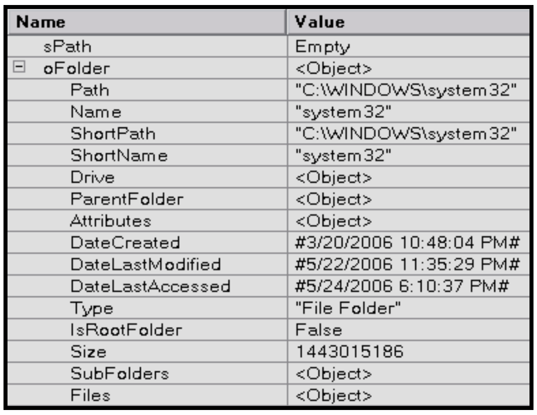
**Tip**

- You can use the **Set** statement to assign the **Folder** object reference to a local object variable. However, if you're interested only in retrieving the path to the special folder, you can do it with a statement like the following:

```
Dim oFso, oFolder, sPath
Set oFolder = oFso.GetSpecialFolder(1)        'Folder object
sPath = oFso.GetSpecialFolder(FolderConst)    'Folder name
or:

sPath = oFso.GetSpecialFolder(FolderConst).Path
```

- The first statement works because the *folderspec* property is the **Folder** object's default property. Since the assignment isn't to an object variable, it's the default property's value, rather than the object reference, that is assigned to sPath.



| Name | Value |
|---|---|
| sPath | Empty |
| ⊟ oFolder | <Object> |
| Path | "C:\WINDOWS\system32" |
| Name | "system32" |
| ShortPath | "C:\WINDOWS\system32" |
| ShortName | "system32" |
| Drive | <Object> |
| ParentFolder | <Object> |
| Attributes | <Object> |
| DateCreated | #3/20/2006 10:48:04 PM# |
| DateLastModified | #5/22/2006 11:35:29 PM# |
| DateLastAccessed | #5/24/2006 6:10:37 PM# |
| Type | "File Folder" |
| IsRootFolder | False |
| Size | 1443015186 |
| SubFolders | <Object> |
| Files | <Object> |

**Figure 12 Set a SpecialFolder**

## FileSystemObject.GetTempName Method

### Description

The **GetTempName** method returns a system-generated temporary file or folder name.

### Syntax

```
object.GetTempName( )
```

**Note**

- **GetTempName** doesn't create a temporary file or folder; it simply provides a name you can use with the **CreateTextFile** method.

**Tip**

- As a general rule, you shouldn't create your own temporary filenames. Windows provides an algorithm within the Windows API to generate the special temporary file and folder names so that it can recognize them later.
- If you are calling **GetTempName** as the first step in creating a temporary file, you can also call the **GetSpecialFolder** method to retrieve the path of the temporary directory, as follows:

```
Option Explicit
Const TemporaryFolder = 2
Dim sTempPath, oFolder, oFso
Set oFso = CreateObject("Scripting.FileSystemObject")
sTempPath = oFso.GetSpecialFolder(TemporaryFolder)
sTempPath = oFso.BuildPath(sTempPath, oFso.GetTempName)
Msgbox sTempPath
```

## FileSystemObject.MoveFile Method

### Description

The **MoveFile** method moves a file from one folder to another.

### Syntax

```
object.MoveFile( source, destination )
```

### Arguments

| Parameter | Description |
|---|---|
| *source* | Required. The path to the file or files to be moved. The source argument string can contain wildcard characters in the last path component only. |
| *destination* | Required. The path where the file or files are to be moved. The destination argument can't contain wildcard characters. |

**Note**

- If *source* contains wildcard characters or if *destination* ends in a path separator, *destination* is interpreted as a path; otherwise, its last component is interpreted as a filename.

- If the *destination* file exists, an error occurs.
- *source* can contain wildcard characters, but only in its last component. This allows multiple files to be moved.
- *destination* can't contain wildcard characters.
- Both *source* and *destination* can be either absolute or relative paths.
- Both *source* and *destination* can be network paths or share names.

**Tip**

- **MoveFile** resolves both arguments before beginning the operation.
- Any single file move operation is atomic; that is, any file removed from source is copied to destination. However, if an error occurs while multiple files are being moved, the execution of the function terminates, but files already moved aren't moved back to their previous folder. If a fatal system error occurs during the execution of this method (like a power failure), the worst that can happen is that the affected file is copied to the destination but not removed from the source. There is no rollback capabilities built into the **MoveFile** method, since, because the copy part of this two-stage process is executed first, the file can't be lost. But while there is no chance of losing data, particularly in multi-file operations, it's more difficult to determine whether the move operations have succeeded. This is because an error at any time while files are being moved causes the **MoveFile** method to be aborted.
- You can use the **GetAbsolutePath**, **FolderExists**, and **FileExists** methods prior to calling the **MoveFile** method to ensure its success.
- The **MoveFile** method differs from the File object's Move method by allowing you to directly designate a file to be moved rather than requiring that you first obtain an object reference to it. It also allows you to move multiple files rather than the single file represented by the **File** object.

## FileSystemObject.MoveFolder Method

**Description**

The **MoveFolder** method moves a folder along with its files and subfolders from one location to another.

**Syntax**

```
object.MoveFolder( source, destination )
```

**Arguments**

| Parameter | Description |
|---|---|
| *source* | Required. The path to the folder or folders to be moved. The source argument string can contain wildcard characters in the last path component only. |
| *destination* | Required. The path where the folder or folders are to be moved. The destination argument can't contain wildcard characters. |

## FileSystemObject.OpenTextFile Method

### Description

The **OpenTextFile** method opens (and optionally first creates) a text file for reading or writing.

### Syntax

```
object.OpenTextFile(filename, iomode, create, format)
```

### Arguments

| Parameter | Description |
|---|---|
| *filename* | Required. String expression that identifies the file to open. |
| *iomode* | Optional. Can be one of three constants: **ForReading**, **ForWriting**, or **ForAppending**.<br>For a list of IOMode Arguments see Table 2 on page 111. |
| *create* | Optional. Boolean value that indicates whether a new file can be created if the specified filename doesn't exist. The value is **True** if a new file is created, **False** if it isn't created. If omitted, a new file isn't created. |
| *format* | Optional. One of three Tristate values used to indicate the format of the opened file. If omitted, the file is opened as ASCII.<br>For a list of Format Arguments see Table 3 on page 111. |

**Note**

- The path element of *filename* can be relative or absolute.
- The default *iomode* setting is **ForReading** (1).
- The default *format* setting is ASCII (False).
- If another process has opened the file, the method fails with a "Permission Denied" error. Both *source* and *destination* can be either absolute or relative paths.
- Both *source* and *destination* can be network paths or share names.
- You can use the **GetAbsolutePath** and **FileExists** methods prior to calling the **OpenTextFile** method to ensure its success.
- The path element of *filename* can be a network path or share name.
- As the table listing values for the *iomode* parameter shows, the Scripting Runtime type library defines constants of the *iomode* enumeration that can be used in place of their numeric equivalents. You can use them in your scripts in either of two ways. You can define the constants yourself by adding the IOMode Arguments to your script:
- The value of *iomode* can be only that of a single constant. Assigning more than one value generates runtime error 5, "Invalid procedure call or argument."
- As the table listing values for the *format* parameter shows, the Scripting Runtime type library defines constants of the Tristate enumeration that can be used in place of their numeric equivalents. You can use them in your scripts in either of two ways. You can define the constants yourself by adding the Format Arguments to your script:

## Drives Collection Object

**Figure 13 Drives collection**

All drives connected to the current machine are included in the **Drives** collection, even those that aren't currently ready (like removable media drives with no media inserted in them). The **Drives** collection object is read-only.

The Drives collection cannot be created; instead, it is returned by the Drives property of the FileSystemObject object, as the following code fragment illustrates:

```
Dim oFso, oDrives
Set oFso = CreateObject("Scripting.FileSystemObject")
Set oDrives = oFso.Drives
```

For an overview of the file system object model, including the library reference needed to access it, see Figure 10 - "File System Object Model" on page 52.

Creatable: No

Returned By

- **FileSystemObject**.**Drives** property

### Drives.Item Property

**Description**

The **Item** property returns a **Drive** object whose key is drive letter.

**Syntax**

```
object.Item( key )
```

**Arguments**

| Parameter | Description |
|---|---|
| *key* | Required. The drive letter. |

**Note**

- This is an unusual collection, since the drive's index value (its ordinal position in the collection) can't be used; attempting to do so generates runtime error 5, "Invalid procedure call or argument." Since attempting to retrieve a **Drive** object for a drive that doesn't exist generates runtime error 68, it's a good idea to call the **FileSystemObject** object's **DriveExists** method beforehand.

## Drives.Count Property

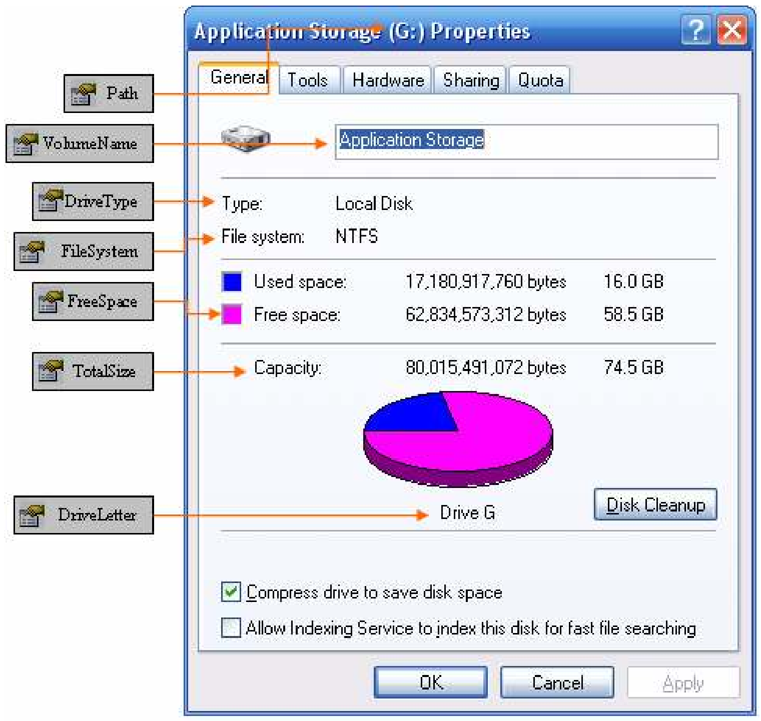
**Description**

The **Count** property returns the number of **Drive** objects in the collection.

**Note**

- Return data type is Long

## Drive Object



Application Storage

Type: Local Disk
File system: NTFS

| | | | |
|---|---|---|---|
| 🟦 | Used space: | 17,180,917,760 bytes | 16.0 GB |
| 🟪 | Free space: | 62,834,573,312 bytes | 58.5 GB |
| | Capacity: | 80,015,491,072 bytes | 74.5 GB |

Drive G

[✓] Compress drive to save disk space
[ ] Allow Indexing Service to index this disk for fast file searching

**Figure 14 Drive Object**

- Represents a single drive connected to the current machine, including a network drive. By using the **Drive** object, you can interrogate the system properties of any drive. In addition, you can use the **Folder** object returned by the **Drive** object's **RootFolder** property as your foothold into the physical drive's file system.
- A new instance of the **Drive** object cannot be created. Instead, a **Drive** object that represents an existing physical drive typically is retrieved from the **FileSystemObject** object's **Drives** collection, as in the following code fragment, which retrieves an object reference that represents the C: drive:

```
Dim oFso, oDrive
Set oFso = CreateObject("Scripting.FileSystemObject")
set oDrive = oFso.Drives("C")
```

- For an overview of the File System object model, including the library reference needed to access it, see Figure 10 - "File System Object Model" on page 52.
- Creatable : No
- Returned By
  - File.Drive property
  - Folder.Drive property
  - FileSystemObject.Drives.Item property
  - FileSystemObject.GetDrive method

## Drive.AvailableSpace Property

**Description**

The **AvailableSpace** property returns the amount of space available to a user on the specified drive or network share.

**Note**

- Reports the amount of free space on the drive, in bytes. To report the amount of available space in kilobytes, divide this value by 1,024. To report the amount of available space in megabytes, divide this value by 1,048,576 (1,024 * 1,024).
- Returns the number of bytes unused on the disk. Typically, the **AvailableSpace** property returns the same number as the **Drive** object's **FreeSpace** property, although differences may occur on systems that support quotas.
- In early versions of the **Scripting Runtime**, **AvailableSpace** was capable of storing only values that ranged from 0 to 2^31, or 2,147,483,648; in other words, in the case of drives with over 2 GB free, it failed to accurately report the amount of available free space.
- The AvailableSpace property reports the amount of space available to the user running the script. If disk quotas are in use on the drive, this value might be less than the total amount of free space available on the drive.
- In order to check the amount of available space on the drive, the drive must be ready. Otherwise, an error is likely to result. This makes it worthwhile to check the value of the **IsReady** property before attempting to retrieve a drive's free space, particularly if your script is iterating the **Drives**

collection.

## Drive.DriveLetter Property

**Description**

The **DriveLetter** property returns the drive letter of a physical local drive or a network share. Read-only.

**Note**

- The drive letter used for this drive on the current machine (e.g., C ). In addition, its value is an empty string ("") if the drive is a network share that has not been mapped to a local drive letter.
- Drive letter assigned to the drive. The drive letter does not include the trailing colon; thus, a floppy disk drive will be reported as **A** rather than **A:**

## Drive.DriveType Property

**Description**

The **DriveType** property returns a value indicating the type of a specified drive.

**Note**

- Integer value indicating the type of drive. Values include:
  - 1 - Removable drive
  - 2 - Fixed drive (hard disk)
  - 3 - Mapped network drive
  - 4 - CD-ROM drive
  - 5 - RAM disk
- A value (see the following table) indicating the type of drive. Any remote drive is shown only as remote. For example, a shared CD-ROM or Zip drive that is both remote and removable is shown simply as remote (i.e., it returns a value of 3) on any machine other than the machine on which it's installed.
- For a list of DriveType Constans see Table 3 on page 111

## Drive.FileSystem Property

**Description**

The **FileSystem** property returns the type of file system in use for the specified drive.

**Note**

- The installed file-system; returns FAT, FAT32, NTFS, or CDFS. In order to determine that the file-system in place, a device must be present on removable drives or runtime error 71, "Disk not ready," results.

## Drive.FreeSpace Property

**Description**

The **FreeSpace** property returns the amount of free space available to a user on the specified drive or network share.

**Note**

- Reports the amount of free space on the drive, in bytes. To report the amount of free space in kilobytes, divide this value by 1,024. To report the amount of free space in megabytes, divide this value by 1,048,576 (1,024 * 1,024).
- The number of bytes unused on the disk. Typically, its value is the same as the **Drive** object's **AvailableSpace** property, although differences may occur on computer systems that support quotas.
- In early versions of the scripting Runtime, the property was capable of storing only values that ranged from 0 to $2^{31}$, or 2,147,483,648. In other words, in the case of drives with over 2 GB free, it failed to accurately report the amount of available free space.
- Unlike the **AvailableSpace** property, **FreeSpace** reports the total amount of free space available on the drive, regardless of whether disk quotas have been enabled.

## Drive.IsReady Property

**Description**

The **IsReady** property returns **True** if the specified drive is ready; **False** if it is not.

**Note**

- Indicates whether a drive is accessible.
- For hard drives, this should always return **True**. For removable media drives, **True** is returned if media is in the drive; otherwise, **False** is returned.
- A number of **Drive** object properties raise an error if the drive they represent is not ready. You can use the **IsReady** property to check the status of the drive and prevent your script from raising an error.

## Drive.Path Property

**Description**

The **Path** property returns the path for a specified file, folder, or drive.

**Note**

- The drive name followed by a colon (e.g., C:). (Note that it does not include the root folder.) This is the default property of the **Drive** object.
- For local drives, this will be the drive letter and the trailing colon (for example, A:). For mapped network drives, this will be the Universal Naming

Convention (UNC) path for the drive (for example, \\Server1\SharedFolder).

## Drive.RootFolder Property

**Description**

The **RootFolder** property returns a **Folder** object representing the root folder of a specified drive.

**Note**

- Gives you access to the rest of the drive's filesystem by exposing a **Folder** object representing the root folder.

## Drive.SerialNumber Property

**Description**

The **SerialNumber** property returns the decimal serial number used to uniquely identify a disk volume.

**Note**

- Serial number assigned to the drive by the manufacturer. For floppy disk drives or mapped network drives, this value will typically be 0.

## Drive.ShareName Property

**Description**

The **ShareName** property returns the network share name for a specified drive.

**Note**

- For a network share, returns the machine name and share name in UNC format (e.g., \NTSERV1\TestWork). If the **Drive** object does not represent a network drive, the **ShareName** property returns a zero-length string ("").

## Drive.TotalSize Property

**Description**

The **TotalSize** property returns the total space, in bytes, of a drive or network share.

**Note**

- The total size of the drive in bytes. In early versions of the Scripting Runtime, the **TotalSize** property was capable of storing only values that ranged from 0 to $2^{31}$, or 2,147,483,648. In other words, in the case of drives larger than 2 GB, it failed to accurately report the total drive size.
- In order to check the amount of total space on the drive, the drive must be ready. Otherwise, a "Disk not ready" error is likely to result. This makes it worthwhile to check the value of the **IsReady** property before attempting to retrieve a drive's free space, particularly if your script is iterating the

**Drives** collection.

## Drive.VolumeName Property

**Description**

The **VolumeName** property sets or returns the volume name of the specified drive (if any).

**Note**

- The drive's volume name, if one is assigned (e.g., DRIVE_C). If a drive or disk has not been assigned a volume name, the **VolumeName** property returns an empty string (""). This is the only read/write property supported by the **Drive** object.
- In order to retrieve the volume name, the drive must be ready. Otherwise, a "Disk not ready" error is likely to result. This makes it worthwhile to check the value of the **IsReady** property before attempting to retrieve a drive's volume name, particularly if your script is iterating the **Drives** collection.

## Folders Collection Object

**Figure 15 Drives collection**

The **Folders** collection object is a container for **Folder** objects. Normally, you'd expect to access a single object from the collection of that object; for example, you'd expect to access a Folder object from the **Folders** collection object. However, things are the other way around here: you access the **Folders** collection object from an instance of a **Folder** object. This is because the first **Folder** object you instantiate from the **Drive** object is a **Root Folder** object, and from it you instantiate a subfolders collection. You can then instantiate other Folder and subfolder objects to navigate through the drive's filesystem.

The **Folders** collection is a subfolder of any **Folder** object. For instance, the top-level Folders collection (representing all of the folders in the root directory of a particular drive) can be can be instantiated as follows:

```
Dim oFso, oFolders
Set oOFso = CreateObject("Scripting.FileSystemObject")
Set oFolders = oOFso.Drives("C").RootFolder.SubFolders
```

The **Folders** collection object is one of the objects in the File System object model; see Figure 10 - "File System Object Model" on page 52.

- Creatable : No
- Returned By:
    - Folders.SubFolders

## Folders.Count Property

**Description**

The **Count** property returns the number of **Folders** objects in the collection.

**Note**

- Return Data Type is Long

## Folders.Item Property

**Description**

The **Item** property returns a **Folder** object whose key is folder name.

**Syntax**

```
object.Item( key )
```

**Arguments**

| Parameter | Description |
|---|---|
| *key* | Required. The folder name. |

**Note**

- Retrieves a particular **Folder** object from the **Folders** collection object. You can access an individual **folder** object by providing the exact name of the folder without its path. However, you can't access the item using its ordinal number. For example, the following statement returns the **Folder** object that represents the Root_Two folder:

```
Set oSubFolder = oSubFolders.Item("Root_Two")
```

## Folders.Add Method

**Description**

The **Add** method adds a new folder to a **Folders** collection.

**Syntax**

```
object.Add( foldername )
```

**Arguments**

| Parameter | Description |
|---|---|
| *foldername* | Required. The name of the new **Folder** being added. |

**Note**

- You can't use a path specifier in *foldername*; you can use only the name of the new folder.
- The location of the new folder is determined by the parent to which the **Folders** collection object belongs. For example, if you are calling the **Add** method from a Folders collection object that is a child of the root **Folder**

object, the new folder is created in the root (i.e., it's added to the root's subfolders collection). For example:

```
Dim oFso, oRoot, oChild, oRootFolders
Set oFso = CreateObject("Scripting.FileSystemObject")
Set oRoot = oFso.Drives("C").RootFolder
Set oRootFolders = oRoot.SubFolders
Set oChild = oRootFolders.Add("Downloads")
Set oChlid = Nothing : Set oRootFolders = Nothing : Set oRoot = Nothing
Set oFso = Nothing
```

## Folder Object

- The **Folder** object allows you to interrogate the system properties of the folder and provides methods that allow you to copy, move, and delete the folder. You can also create a new text file within the folder.
- The **Folder** object is unusual because with it, you can gain access to a **Folders** collection object. The more usual method is to extract a member of a collection to gain access to the individual object. However, because the **Drive** object exposes only a **Folder** object for the root folder, you have to extract a Folders collection object from a **Folder** object (the collection represents the subfolders of the root). From this collection, you can navigate downward through the filesystem to extract other **Folder** objects and other **Folders** collections. A Boolean property, **IsRootFolder**, informs you of whether the **Folder** object you are dealing with currently is the root of the drive.
- The **Folder** object is one of the objects in the Filesystem object model; for an overview of the model, see Figure 10 - "File System Object Model" on page 52.
- Creatable : No
- Returned By
  - Drive.RootFolder property
  - FileSystemObject.CreateFolder method
  - FileSystemObject.GetFolder method
  - File.ParentFolder property
  - Folder.SubFolders.Item property
  - Folder.ParentFolder property
  - Folders.Add method

### Folder.Attributes Property

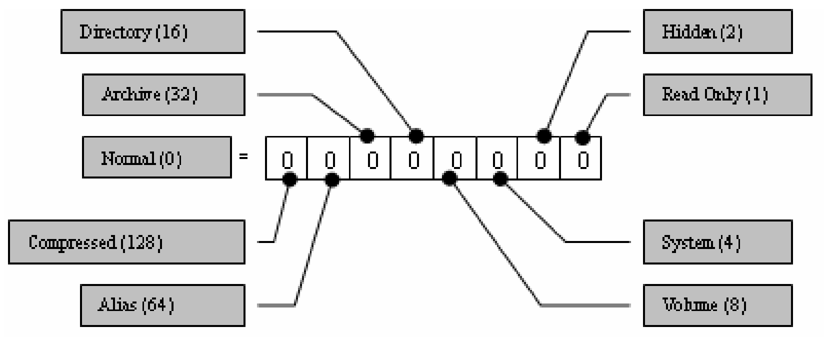


**Figure 16 Attributes bitmap representation**

**Description**

The **Attributes** property sets or returns the attributes of files or folders. Read/write or read-only, depending on the attribute.

**Syntax**

```
object.Attributes
```

**Note**

- More information about bitmaps see Working with Bitmaps on page 12
- As Figure 16 shows, the Scripting Runtime type library defines constants of the **FileAttribute** enumeration that can be used in place of their numeric equivalents. You can use them in your scripts in either of two ways. You can define the constants yourself by adding the following code to your script or add the constants on an external vbs file.

```
Const Normal = 0
Const ReadOnly = 1
Const Hidden = 2
Const System = 4
Const Directory = 16
Const Archive = 32
Const Alias = 64
Const Compressed = 128
```

You can determine which flag is set by using a logical **AND** along with the value returned by the property and the value of the flag you'd like to test. For example:

```
If fso.Attributes And ReadOnly Then ' Folder is read-only
```

To clear a flag, And the value of the **Attributes** property with a Long in which the flag you want to clear is turned off. For example, the following code clears a **Folder** object's read-only flag:

```
fso.Attributes = fso.Attributes And (Not ReadOnly)
```

## Folder.DateCreated Property

**Description**

The **DateCreated** returns the date and time that the specified file or folder was created.

**Note**

- The date and time the folder was created.

## Folder.DateLastAccessed Property

**Description**

The **DateLasAccessed** returns the date and time that the specified file or folder was last accessed.

**Note**

- The date and, if it's available from the operating system, the time that the folder was last accessed.

## Folder.DateLastModified Property

**Description**

The **DateLasModified** returns the date and time that the specified file or folder was last modified.

**Note**

- The date and time the folder was last modified.

## Folder.Drive Property

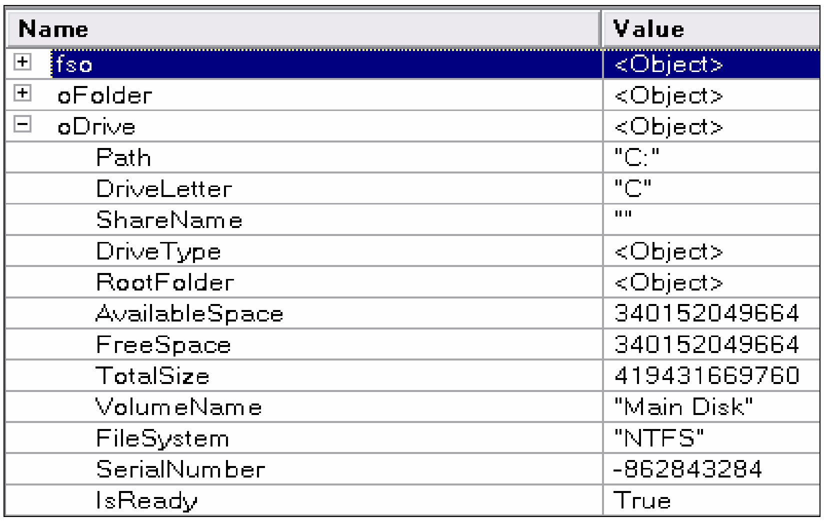
**Description**

The **Drive** returns the drive letter of the drive or the drive object on which the specified file or folder resides.

**Note**

- Returns a **Drive** object representing the drive on which the folder resides; the property is read-only.

```
Option Explicit
Dim fso, oFolder, oDrive
Dim sDrive
Set fso = CreateObject("Scripting.FileSystemObject")
Set oFolder = fso.GetFolder("C:\My Documents\Book\V1")
sDrive = oFolder.Drive                ' Returns C:
'--- Or
sDrive = oFolder.Drive.DriveLetter    ' Returns C
'--- Retrieve the drive object
Set oDrive = oFolder.Drive            ' Returns the drive object
```



| Name | Value |
|---|---|
| ⊞ fso | <Object> |
| ⊞ oFolder | <Object> |
| ⊟ oDrive | <Object> |
| Path | "C:" |
| DriveLetter | "C" |
| ShareName | "" |
| DriveType | <Object> |
| RootFolder | <Object> |
| AvailableSpace | 340152049664 |
| FreeSpace | 340152049664 |
| TotalSize | 419431669760 |
| VolumeName | "Main Disk" |
| FileSystem | "NTFS" |
| SerialNumber | -862843284 |
| IsReady | True |

**Figure 17 Drive object**

## Folder.Files Property

**Description**

The **Files** property returns a **Files** collection consisting of all **File** objects contained in the specified folder, including those with hidden and system file attributes set.

## Folder.IsRootFolder Property

**Description**

The **Files** property returns **True** if the specified folder is the root folder; **False** if it is not.

## Folder.Name Property

**Description**

The **Name** property Sets or returns the name of a specified file or folder.

**Note**

Folder name, not including path information. For example, the Name of the folder C:\Windows\System32 is System32.

## Folder.ParentFolder Property

**Description**

The **ParentFolder** property returns the folder object for the parent of the specified file or folder.

**Note**

- Returns a **folder** object representing the folder that's the parent of the current folder. It returns **Nothing** if the current object is the root folder of its drive (i.e., if its **IsRootFolder** property is **True**).

## Folder.Path Property

**Description**

The **Path** property returns the path for a specified file, folder, or drive.

**Note**

- Returns the complete path of the current folder, including its drive. It is the default property of the **Folder** object.

## Folder.ShortName Property

**Description**

The **ShortName** property returns the short name used by programs that require the earlier 8.3 naming convention.

**Note**

- Returns a DOS 8.3 folder name without the folder's path. The property is read-only.

## Folder.ShortPath Property

### Description

The **ShortPath** property returns the short path used by programs that require the earlier 8.3 file naming convention.

## Folder.Size Property

### Description

The **Size** property returns the size, in bytes, of all files and subfolders contained in the folder.

### Note

- Returns the total size of all files, subfolders, and their contents in the folder structure, starting with the current folder. The property is read-only.
- In previous versions of the Scripting Runtime, this property failed to accurately report the size of a folder whose files and subfolders occupied more than 2 GB of disk space.
- Attempting to retrieve the value of a Folder object's **Size** property when that folder is a drive's root folder (that is, its **IsRootFolder** property returns **True**) generates a runtime error.

## Folder.SubFolders Property

### Description

The **Size** property returns a **Folders** collection consisting of all folders contained in a specified folder, including those with hidden and system file attributes set.

## Folder.Type Property

### Description

The **Type** property returns information about the type of a file or folder.

### Note

- Returns the description of a filesystem object, as recorded in the system registry. For **Folder** objects, the property always returns "File Folder."

## Folder.Copy Method

### Description

The **Copy** method copies the current folder and its contents, including other folders, to another location.

**Syntax**

```
object.Copy ( destination, overwrite )
```

**Arguments**

| Parameter | Description |
|---|---|
| *destination* | Required. Destination where the file or folder is to be copied. |
| *overwrite* | Optional. Boolean value that is **True** (default) if existing files or folders are to be overwritten; **False** if they are not. |

**Note**

- Wildcard characters can't be used in *destination*.
- The folder and all subfolders and files contained in the source folder are copied to *destination*. That is, the **Copy** method is recursive.
- Unlike the **FileSystemObject**.**CopyFolder** method, there is no operational difference between ending *destination* with a path separator or not.
- If the *destination* path or any of the files contained in the *destination* structure are set to read-only, the **Copy** method will fail regardless of the value of *overwrite* and will generate a "Permission denied" error.
- If *overwrite* is set to **False**, and the source folder or any of the files contained in the *destination* structure exists in the *destination* structure, then trappable error 58, "File Already Exists," is generated.
- If an error occurs while copying more than one file, the **Copy** method exits immediately, leaving the rest of the files uncopied. There is also no rollback facility to undo the copies prior to the error.
- If the user has adequate rights, *destination* can be a network path or share name. For example:

```
oFolder.Copy "\\NTSERV1\d$\RootTwo\"
```

## Folder.CreateTextFile Method

**Description**

The **CreateTextFile** method creates a new file at the specified location and returns a **TextStream** object for that file.

**Syntax**

```
object.CreateTextFile(filename, overwrite, unicode)
```

**Arguments**

| Parameter | Description |
|---|---|
| *filename* | Required. String expression that identifies the file to create. |
| *overwrite* | Optional. Boolean value that indicates whether you can overwrite an existing file. The value is **true** if the file can be overwritten, **false** if it can't be overwritten. |
| *unicode* | Optional. Boolean value that indicates whether the file is created as a Unicode or ASCII file. The value is true if the file is created as a Unicode file, false if it's created as an ASCII file. If omitted, an ASCII file is assumed. |

**Note**

- *filename* can be a relative or absolute path. Wildcard characters are not allowed in *filename*.
- If no path is specified in *filename*, the script's current drive and directory are used. If no drive is specified in *filename*, the script's current drive is used.
- The default value for *overwrite* is **False**.
- If Unicode is set to **True**, a Unicode file is created; otherwise it's created as an ASCII text file.
- The default value for Unicode is **False**.

**Tip**

- If the path specified in *filename* does not exist, the method fails. To prevent this error, you can use the **FileSystemObject** object's **FolderExists** method to be sure that the path is valid.
- The newly created text file is automatically opened only for writing. If you subsequently wish to read from the file, you must first close it and reopen it in read mode.
- If the file referred to in *filename* already exists as a read-only file, the **CreateTextFile** method fails regardless of the value of *overwrite*.
- You must use the **Set** statement to assign the **TextStream** object to a local object variable.
- If the user has adequate rights, *filename* can contain a network path, or share name. For example:

```
oFolder.CreateTextFile "\\NTSERV1\RootTest\myFile.doc"
```

- The **CreateTextFile** method in the **Folder** object is identical in operation to that in the **FileSystemObject** object.

## Folder.Delete Method

**Description**

The **Delete** method removes the folder specified by the **Folder** object and all its files and subfolders.

**Syntax**

```
object.Delete ( force )
```

**Arguments**

| Parameter | Description |
|---|---|
| *force* | Optional. Boolean value that is **True** if files or folders with the read-only attribute set are to be deleted; **False** (default) if they are not. |

**Note**

- If any of the files within the folder are open, the method fails with a "Permission Denied" error.
- The **Delete** method deletes all the contents of the given folder, including subfolders and their contents.
- The default setting for *force* is **False**. If any of the files in the folder or its subfolders are set to read-only, the method will fail.
- If *force* is set to **False** and any of the files in the folders are set to read-only, the method fails.
- The **Delete** method deletes a folder and its files and subfolders permanently; it does not move the folder or its files and subfolders to the Recycle Bin.
- If an error occurs while deleting more than one file in the folder, the **Delete** method exits immediately, thereby leaving the rest of the folders or files undeleted. There is also no rollback facility to undo the deletions prior to the error.
- Unlike the **FileSystemObject.DeleteFolder** method, which accepts wildcard characters in the path parameter and can therefore delete multiple folders, the **Delete** method deletes only the single folder represented by the **Folder** object.
- Immediately after the **Delete** method executes, the Folders collection object containing the **Folder** object is automatically updated. The deleted folder is removed from the collection, and the collection count is reduced by one. You shouldn't try to access the deleted **Folder** object again, and you should set the local object variable to **Nothing**.

## Folder.Move Method

**Description**

The **Move** method moves a folder structure from one location to another.

**Syntax**

```
object.Move( destination )
```

**Arguments**

| Parameter | Description |
|---|---|
| *destination* | Required. The path to the location where the file is to be moved. |

**Note**

- Wildcard characters can't be used in *destination*.
- If any of the files within the folder being moved are open, an error is generated.
- All subfolders and files contained within the source folder are copied to *destination*, unless disallowed by the wildcard characters. That is, the **Move** method is recursive.
- *destination* can be either an absolute or a relative path..
- If a fatal system error (like a power failure) occurs during the execution of this method, the worst that can happen is that the folder is copied to the destination but not removed from the source. There are no rollback capabilities built into the **Folder**.**Move** method; since, the copy part of this two-stage process is executed first, the folder can't be lost.
- If an error occurs in the middle of a move operation, the operation is terminated, and the remaining files and folders in the folder aren't moved.
- If a folder or a file by the same name already exists in destination, the method generates runtime error 58, "File already exists." To prevent this, you can use the **FileSystemObject.FolderExists** and **GetAbsolutePath** methods prior to calling the **Move** method.
- Unlike the **FileSystemObject.MoveFolder** method, which accepts wildcard characters in the source parameter and can therefore move multiple folders, the **Move** method moves only the single folder represented by the **Folder** object and its contents.
- Immediately after the **Move** method executes, the **Folders** collection object containing the **Folder** object is automatically updated, the moved folder is removed from the collection, and the collection count is reduced by one. You shouldn't try to access the moved folder object again from the same Folders collection object.
- *object*, the **Folder** object reference, remains valid after the folder has been moved. Its relevant properties (the **Drive**, **ParentFolder**, **Path**, and **ShortPath** properties, for example) are all updated to reflect the folder's new path after the move.
- If the user has adequate rights, the destination can be a network path or share name. For example:

```
fso.Move "\\NTSERV1\d$\RootTwo\"
```

## Files Collection Object

**Figure 18 Files collection**

The **Files** collection object is one of the objects in the File System object model; for an overview of the model, including the library reference needed to access it, see Figure 10 - "File System Object Model" on page 52

The **Files** collection object is a container for File objects that returned by the **Files** property of any **Folder** object. All files contained in the folder are included in the **Files** collection object. You can obtain a reference to a **Files** collection object using a code fragment like the following:

```
Dim oFso, oFiles
Set oFso = CreateObject("Scripting.FileSystemObject")
Set oFiles = oFso.Drives("C:").RootFolder.SubFolders("Windows").Files
```

This code returns the **Files** collection for the Windows folder.

You can obtain a reference to an individual **File** object using the **Files** collection object's Item property; this takes the exact filename, including the file extension, as an argument. To iterate through the collection, you can use the **For Each...Next** statement.

- Creatable : No
- Returned By
  - Folder.Files property.

## Files.Count Property

**Description**

The **Count** property returns the number of **Files** objects in the collection.

## Files.Item Property

**Description**

The **Item** property returns a **File** object whose key is file name.

**Syntax**

```
object.Item( key )
```

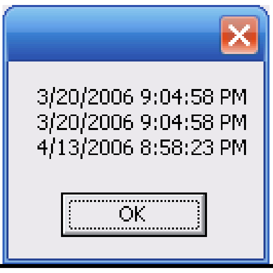
**Arguments**

| Parameter | Description |
|---|---|
| *key* | Required. The file name. |

**Note**

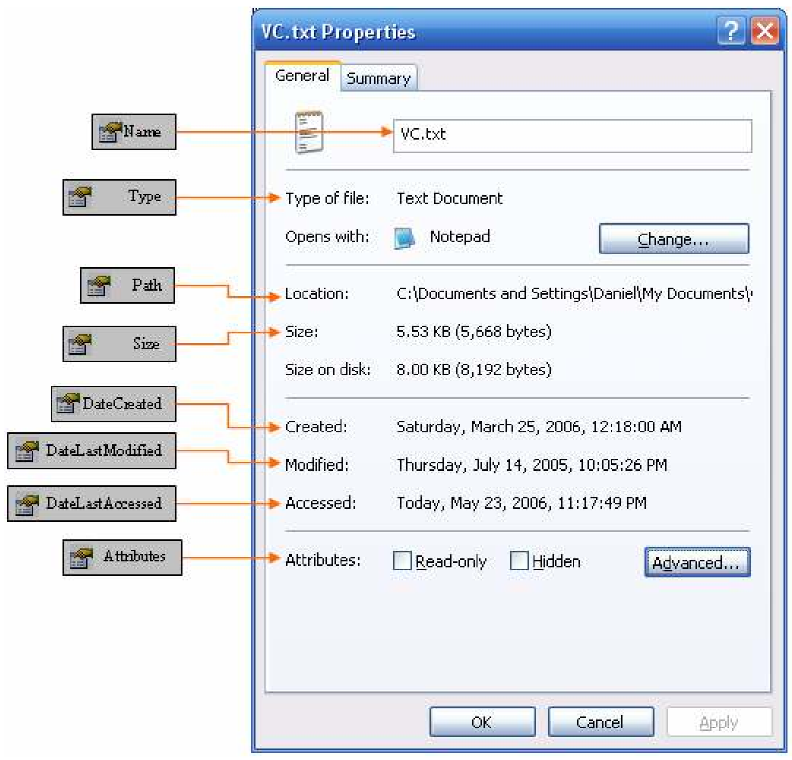
- Takes the filename (including the file extension) as a parameter and returns the **File** object representing the file with that name. Individual **File** objects can't be accessed by their ordinal position in the collection. Item is the **Files** collection object's default property. The code fragment shown next uses the Item property to retrieve the autoexec.bat **File** object.

```
Option Explicit
Dim oFso, oFiles, oFile
Set oFso = CreateObject("Scripting.FileSystemObject")
Set oFiles = oFso.Drives("C:").RootFolder.Files
Set oFile = oFiles.Item("autoexec.bat")
MsgBox oFile.DateCreated & vbCrLf & _
            oFile.DateLastModified & vbCrLf & _
            oFile.DateLastAccessed
```



```
3/20/2006 9:04:58 PM
3/20/2006 9:04:58 PM
4/13/2006 8:58:23 PM
```

## File Object



VC.txt Properties

VC.txt

Type of file: Text Document

Opens with: Notepad

Location: C:\Documents and Settings\Daniel\My Documents\

Size: 5.53 KB (5,668 bytes)

Size on disk: 8.00 KB (8,192 bytes)

Created: Saturday, March 25, 2006, 12:18:00 AM

Modified: Thursday, July 14, 2005, 10:05:26 PM

Accessed: Today, May 23, 2006, 11:17:49 PM

Attributes: ☐ Read-only ☐ Hidden

**Figure 19 File Object**

The **File** object represents a disk file that can be a file of any type and allows you to interrogate the properties of the file and to move upward in the filesystem hierarchy to interrogate the system on which the file resides. The process of instantiating a **File** object, for example, assigning a reference from the **File** object's Item property to a local object variable, doesn't open the file. An open file represented in the File System object model by a **TextStream** object, which can be generated by the **File** object's **OpenAsTextStream** method.

There are several methods of retrieving a reference to an existing **File** object:

If you want to work with a particular file, you can retrieve a reference to it directly by calling the **GetFile** method of the **FileSystemObject** object. For example:

```
Dim oFso, oFile
Set oFso = CreateObject("Scripting.FileSystemObject")
Set oFile = oFso.GetFile("C:\Documents\MyReport.doc")
```

Allows you to retrieve a reference to a **File** object representing the MyReport.doc file without having to use the **File** System object model to navigate the filesystem.

If you want to work with a file as a member of a folder or of a set of files, you can retrieve a reference to a **File** object that represents it from the Item property of the **Files** collection. (The **Files** collection is returned by the **Files** property of a **Folder** object.) The following code fragment, for instance, retrieves a reference to a file named MyReport.doc that is a member of the Documents folder:

```
Dim oFso, oFile
Set oFso = CreateObject("Scripting.FileSystemObject")
Set oFile = oFso.Drives("C").RootFolder.SubFolders("Rep").Files("Report.doc")
```

Note that a **File** object represents an existing file; you cannot create a **File** object representing a new file. (You can, however, create a new **TextStream** object that represents a new text file by calling the **Folder** object's **CreateTextFile** method.)

- Creatable : No
- Returned By
  - FileSystemObject.GetFile method
  - Files.Item

## File.Attributes Property

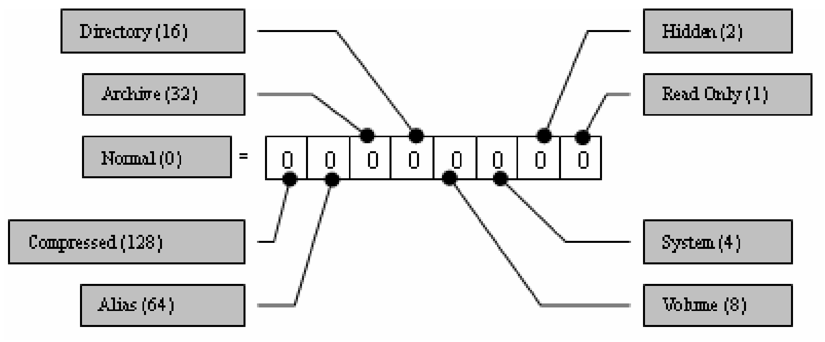


**Figure 20 Attributes bit representation**

### Description

The **Attributes** property sets or returns the attributes of files or folders. Read/write or read-only, depending on the attribute.

**Syntax**

```
object.Attributes[= newattributes]
```

**Arguments**

| **Parameter** | **Description** |
|---|---|
| *newattributes* | Optional. If provided, *newattributes* is the new value for the attributes of the specified object. |

**Note**

- More information about bitmaps see Working with Bitmaps on page 12
- As Figure 16 shows, the Scripting Runtime type library defines constants of the **FileAttribute** enumeration that can be used in place of their numeric equivalents. You can use them in your scripts in either of two ways. You can define the constants yourself by adding the following code to your script or add the constants on an external vbs file.

```
Const Normal = 0
Const ReadOnly = 1
Const Hidden = 2
Const System = 4
Const Directory = 16
Const Archive = 32
Const Alias = 64
Const Compressed = 128
```

You can determine which flag is set by using a logical **AND** along with the value returned by the property and the value of the flag you'd like to test. For example:

```
If oFso.Attributes And ReadOnly Then ' File is read-only
```

To clear a flag, And the value of the **Attributes** property with a Long in which the flag you want to clear is turned off. For example, the following code clears a **Folder** object's read-only flag:

```
oFso.Attributes = oFso.Attributes And (Not ReadOnly)
```

## File.DateCreated Property

### Description

The **DateCreated** returns the date and time that the specified file or folder was created.

### Note

- The date and time the file was created.

## File.DateLastAccessed Property

### Description

The **DateLasAccessed** returns the date and time that the specified file or folder was last accessed.

### Note

- The date and time the file was last accessed. Whether the property includes the date and time or only the date depends on the operating system; Windows 95, Windows 98, and Windows ME, for instance, only return the date, while Windows NT, Windows 2000, and Windows XP return the date and time. The property is read-only.

## File.DateLastModified Property

### Description

The **DateLasModified** returns the date and time that the specified file or folder was last modified.

### Note

- The date and time the file was last modified; the property is read-only.

## File.Drive Property

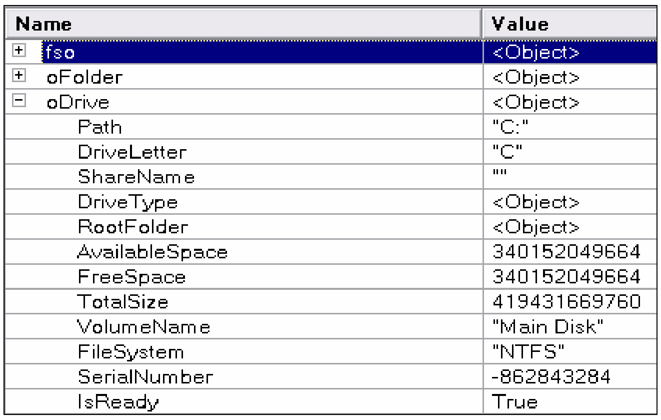
### Description

The **Drive** returns the drive letter of the drive or the drive object on which the specified file or folder resides.

### Note

- The property is read-only.
- Drive letter and trailing colon (for example, C:) representing the drive on which the file is stored.

```
Option Explicit
Dim fso, oFile, oDrive
Dim sDrive
Set fso = CreateObject("Scripting.FileSystemObject")
Set oFile = fso.GetFolder("C:\My Documents\Report.txt")
sDrive = oFile.Drive                ' Returns C:
'--- Or
sDrive = oFile.Drive.DriveLetter    ' Returns C
'--- Retrieve the drive object
Set oDrive = oFile.Drive            ' Returns the drive object
```



| Name | Value |
|---|---|
| ⊞ fso | <Object> |
| ⊞ oFolder | <Object> |
| ⊟ oDrive | <Object> |
| Path | "C:" |
| DriveLetter | "C" |
| ShareName | "" |
| DriveType | <Object> |
| RootFolder | <Object> |
| AvailableSpace | 340152049664 |
| FreeSpace | 340152049664 |
| TotalSize | 419431669760 |
| VolumeName | "Main Disk" |
| FileSystem | "NTFS" |
| SerialNumber | -862843284 |
| IsReady | True |

**Figure 21 Drive object**

## File.Name Property

### Description

The **Name** property Sets or returns the name of a specified file or folder.

### Note

- File name, not including path information. For example, the Name of the file C:\Windows\System32\Scrrun.dll is Scrrun.dll.

## File.ParentFolder Property

### Description

The **ParentFolder** property returns the folder object for the parent of the specified file or folder.

**Note**

- The property is read-only.
- Name of the folder in which the file is stored. For example, the **ParentFolder** of C:\Windows\System32\Scrrun.dll is Windows.

## File.Path Property

**Description**

The **Path** property returns the path for a specified file, folder, or drive.

**Note**

- Returns the full path to the file from the current machine, including drive letter or network path/share name (for example, C:\Windows\System32\Scrrun.dll).
- the property is read-only.
- **Path** is the default property of the **File** object.

## File.ShortName Property

**Description**

The **ShortName** property returns the short name used by programs that require the earlier 8.3 naming convention.

**Note**

- Returns a DOS 8.3 filename.

## File.ShortPath Property

**Description**

The **ShortPath** property returns the short path used by programs that require the earlier 8.3 file naming convention.

**Note**

- For example, the file C:\Windows\Program Files\MyScript.vbs might have The ShortName C:\Windows\Progra~1\MyScript.vbs.

## File.Size Property

**Description**

The **Size** property returns the size, in bytes, of the specified file.

**Note**

- The **Size** property holds a long integer, meaning that it accurately reports file sizes from 0 to 2,147,483,648 bytes. In previous versions of **VBScript**, the property failed to accurately report the size of large files of over 2 GB.

## File.Type Property

**Description**

The **Type** property returns information about the type of a file or folder.

**Note**

- Returns a string containing the registered type description as recorded in the registry. This is the type string displayed for the file in Windows Explorer (for example, "Microsoft Word Document").
- If a file doesn't have an extension, the type is simply "File." When a file's type isn't registered, the type appears as the extension and "File."
- The property is read-only.

## File.Copy Method

**Description**

The **Copy** method copies a specified file or folder from one location to another.

**Syntax**

```
object.Copy ( destination, overwrite )
```

**Arguments**

| Parameter | Description |
|---|---|
| *destination* | Required. Destination where the file or folder is to be copied. |
| *overwrite* | Optional. Boolean value that is **True** (default) if existing files or folders are to be overwritten; **False** if they are not. |

**Note**

- Wildcard characters can't be used in *destination*.

**Tip**

- If the *destination* path is set to read-only, the **Copy** method fails regardless of the *overwrite* setting and generates a "Permission denied" error.
- If *overwrite* is False and the file already exists in *destination*, runtime error 58, "File Already Exists," is generated.
- If the user has adequate rights, *destination* can be a network path or share name. For example:

```
MyFile.Copy "\\NTSERV1\d$\RootTwo\"
MyFile.Copy "\\NTSERV1\RootTest"
```

## File.Move Method

**Description**

The **Move** method moves a file from one folder to another.

**Syntax**

```
object.Move ( destination )
```

**Arguments**

| Parameter | Description |
|---|---|
| *destination* | Required. The path to the location where the file is to be moved. |

**Note**

- The file represented by *object* must not be open or an error occurs.
- Wildcard characters can't be used in *destination*.
- *destination* can be either an absolute or a relative path.

**Tip**

- If a fatal system error occurs during the execution of this method (like a power failure), the worst that can happen is that the file is copied to the destination but not removed from the source. There are no rollback capabilities built into the File.**Move** method; however, because the copy part of this two-stage process is executed first, the file can't be lost.
- If a folder or a file by the same name already exists in destination, the method generates runtime error 58, "File exists." To prevent this, you can use the **FileSystemObjec.FileExists** and **GetAbsolutePath** methods prior to calling the **Move** method.
- Unlike the **FileSystemObject.MoveFile** method, which accepts wildcard characters in the path parameter and can therefore move multiple files, the **Move** method moves only the single file represented by *object*.
- As a result of the **Move** method, the **Files** collection object originally containing *object* of is automatically updated, the file is removed from it, and the collection count is reduced by one. You shouldn't try to access the moved file object again in the same **Folders** collection object.
- The **File** object reference, remains valid after the file has been moved. Its relevant properties (the **Drive**, **ParentFolder**, **Path**, and **ShortPath** properties, for example) are all updated to reflect the file's new path after the move.
- If the user has rights, destination can be a network path or share name:

```
fso.Move "\\NTSERV1\d$\RootTwo\myfile.doc"
```

## File.Delete Method

**Description**

The **Delete** method removes the current file.

**Syntax**

```
object.Delete ( force )
```

**Arguments**

| Parameter | Description |
|---|---|
| *force* | Optional. Boolean value that is **True** if files or folders with the read-only attribute set are to be deleted; **False** (default) if they are not. |

**Note**

- The **Delete** method deletes a file permanently; it does not move it to the Recycle Bin.
- If the file is open, the method fails with a "Permission Denied" error.
- The default setting for *force* is **False**.
- If *force* is set to **False**, and the file is read-only, the method will fail.

**Note**

- Unlike the **FileSystemObject** object's **DeleteFile** method, which accepts wildcard characters in the path parameter and can therefore delete multiple files, the **Delete** method deletes only the single file represented by *object*.
- As a result of the **Delete** method, the **Files** collection object containing *object* is automatically updated, the deleted file is removed from the collection, and the collection count is reduced by one. You shouldn't try to access the deleted file object again; you should set *object* to **Nothing**.

## File.OpenAsTextStream Method

**Description**

The **OpenAsTextStream** method opens a specified file and returns a **TextStream** object that can be used to read from, write to, or append to.

**Syntax**

```
object.OpenAsTextStream (iomode, format)
```

**Arguments**

| Parameter | Description |
|---|---|
| *iomode* | Optional. Indicates input/output mode. Can be one of three constants: **ForReading**, **ForWriting**, or **ForAppending**.<br>For a list of IOMode Arguments see Table 2 on page 111. |
| *format* | Optional. One of three Tristate values used to indicate the format of the opened file. If omitted, the file is opened as ASCII.<br>For a list of Format Arguments see Table 3 on page 111. |

**Note**

- The default value for *iomode* is 1, **ForReading**.
- The Scripting Runtime type library defines constants of the *iomode* enumeration that can be used in place of their numeric equivalents for the *iomode* argument. You can use them in your scripts in either of two ways. You can define the constants yourself by adding the following code to your script or by adding the constants of IOMode Arguments to an external **vbs** file.
- The default value for format is 0 or ASCII (TristateFalse).
- The Scripting Runtime type library defines constants of the **Tristate** enumeration that can be used in place of their numeric equivalents for the *format* argument. You can use them in your scripts in either of two ways. You can define the constants yourself by adding the following code to your script or by adding the constants of Format Arguments to an external **vbs** file.

**Tip**

- If another process has opened the file, the method fails with a "Permission Denied" error.
- The **TextStream** object is so named for a very good reason: it is designed to work with text files rather than binary files. Although it is possible to use the **OpenAsTextStream** method to open a binary file, an enormous number of subtle bugs may crop up when you manipulate binary data as text. Because of this, if you want to work with binary files, you should use some technology (like the **ADO** binary file object) or programming language that's more amenable to processing binary files.

## TextStream Object

Most commonly, the **TextStream** object represents a text file. As of Windows Script Host 2.0 and **VBScript** 5.5, however, it also represents any input/output stream, such as standard input, standard output, and the standard error stream. Depending on the precise character of the I/O stream, you can open a **TextStream** object to read from, append to, or write to the stream. The **TextStream** object provides methods to read, write, and close the text file or I/O stream.

When dealing with files, note that the **TextStream** object represents the file's contents or internals; the **File** object represents the file's externals or the file as an object in the filesystem.

The **TextStream** object is one of the objects in the File System object model; for an overview of the model, including the library reference needed to access it, see Figure 10 - "File System Object Model" on page 52

The availability of **TextStream** object properties depends on the precise character of the **TextStream** object; some properties are available only when the stream is opened in read mode (indicated by an R in the Availability field); others are available in both read and write modes (indicated by a RW in the Availability field). All of the following **TextStream** object properties are read-only:

- Creatable : No
- Returned By
  - File.**OpenTextStream** Method
  - FileSystemObject.**CreateTextFile** Method
  - FileSystemObject.**OpenTextFile** Method

## TextStream.AtEndOfLine Property

**Description**

The **AtEndOfLine** property is a flag denoting whether the end-of-a-line marker has been reached (**True**) or not (**False**). Relevant only when reading a file.

**Note**

- When reading a standard input stream from the keyboard, the end of a line is indicated by pressing the Enter key.
- The **AtEndOfLine** property applies only to **TextStream** files that are open for reading; otherwise, an error occurs.

## TextStream.AtEndOfStream Property

**Description**

The **AtEndOfStream** property returns a Boolean value indicating whether the end of an input stream has been reached.

**Note**

- When reading a standard input stream from the keyboard, the end of a line is indicated by pressing the Enter key.

## TextStream.Column Property

**Description**

The **Column** property returns a read-only property that returns the column number of the current character position in a **TextStream** file.

**Note**

- Returns the column number position of the file marker. The first column position in the input stream and in each row is 1.
- Examining the value of the **Column** property is most useful in input streams after calls to the **TextStream** object's **Read** and **Skip** methods. Although it is less useful for output streams, it can be used after a call to the **TextStream** object's **Write** method.

## TextStream.Line Property

**Description**

The **Line** property returns the current line number in a **TextStream** file.

**Note**

- Returns the line number position of the file marker. Lines in the text stream are numbered starting at 1.
- Unless the end of the text stream has been reached, the value of the Line property is incremented after calls to the **ReadAll**, **ReadLine**, and **SkipLine** methods. Similarly, in output streams, it is incremented after calls to the **WriteLine** and **WriteBlankLines** methods.

## TextStream.Close Method

**Description**

The **Close** method closes the current **TextStream** object

**Note**

- Although calling the **Close** method does not invalidate the object reference, you shouldn't try to reference a **TextStream** object that has been closed.

**Tip**

- After closing the **TextStream** object, set *object* to **Nothing**.
- If you are writing to a file-based text stream, text is automatically written to the file. You do not have to call the **Save** method to commit changes to a disk file before calling the **Close** method.

## TextStream.Read Method

**Description**

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

**Syntax**

```
object.Read ( characters )
```

**Arguments**

| Parameter | Description |
|---|---|
| *characters* | Required. Number of characters you want to read from the file. |

**Note**

- Files opened for writing or appending can't be read; you must first close the file and reopen it using the **ForReading** constant.
- After the read operation, the file pointer advances Characters *characters*, unless the end of the file is encountered.
- If the number of characters available to be read are less than *characters*, all *characters* will be read.
- When reading the standard input stream from the keyboard, program execution pauses until an end-of-line or end-of-stream character is encountered. However, only the first Characters *characters* of the stream are read. If at least Characters *characters* are available in the input stream

for subsequent read operations, program execution does not pause to wait for further keyboard input. The usual technique is to process keystrokes in a loop until the end-of-stream marker is encountered.

## TextStream.ReadAll Method

**Description**

The **ReadAll** method reads the entire file or input stream into memory.

**Syntax**

```
object.ReadAll ()
```

**Note**

- For large files, use the **ReadLine** or **Read** methods to reduce the load on memory resources.
- Files opened for writing or appending can't be read; you must first close the file and reopen it using the **ForReading** constant.
- When used to read the standard input stream from the keyboard, the **ReadAll** method pauses program execution and polls the keyboard until the **AtEndOfStream** symbol is encountered. For this reason, the **ReadAll** method should not be executed repeatedly in a loop.

## TextStream.ReadLine Method

**Description**

The **ReadLine** method reads a line of the text file or input stream into memory, from the start of the current line up to the character immediately preceding the next end-of-line marker.

**Note**

- Files opened for writing or appending can't be read; you must first close the file and reopen it using the **ForRead** constant.
- The **ReadLine** method causes the file pointer to advance to the beginning of the next line, if there is one.
- When used to retrieve standard input from the keyboard, the **ReadLine** method pauses program execution and waits until the end-of-line character (i.e., the Enter key) has been pressed. Unless your script expects to retrieve just one line of input, it's best to call the **ReadLine** method repeatedly in a loop.

## TextStream.Skip Method

**Description**

The **Skip** method skips a specified number of characters when reading a **TextStream** file.

**Syntax**

```
object.Skip ( characters )
```

**Arguments**

| **Parameter** | **Description** |
|---|---|
| *characters* | Required. Number of characters to skip when reading a file. |

**Note**

- As a result of the skip operation, the file pointer is placed at the character immediately following the last skipped character.
- The **Skip** method is available only for input streams, that is, for files or streams opened in **ForReading** mode.

## TextStream.SkipLine Method

**Description**

The **SkipLine** method ignores the current line when reading from a text file.

**Note**

- The **SkipLine** method is available only for files opened in **ForReading** mode.
- After the **SkipLine** method executes, the internal file pointer is placed at the beginning of the line immediately following the skipped line, assuming that one exists.

## TextStream.Write Method

**Description**

The **Write** method writes a specified string to a **TextStream** file.

**Syntax**

```
object.Write ( string )
```

**Arguments**

| **Parameter** | **Description** |
|---|---|
| *string* | Required. The text you want to write to the file. |

**Note**

- The file marker is set at the end of string. As a result, subsequent writes to the file adjoin each other, with no spaces inserted. To write data to the file in a more structured manner, use the **WriteLine** method.

## TextStream.WriteBlankLines Method

**Description**

The **WriteBlankLines** method writes a specified number of newline characters to a **TextStream** file.

**Syntax**

```
object.WriteBlankLines ( lines )
```

**Arguments**

| Parameter | Description |
|---|---|
| *lines* | Required. Number of newline characters you want to write to the file. |

**Example**

```
oTxtFile.Write ("This is line 1.")
oTxtFile.Write ("This is line 2.")
```

The resulting text file looks like this:

```
This is line 1.This is line 2.
```

## TextStream.WriteLine Method

**Description**

The **Write** method writes a specified string and newline character to a **TextStream** file.

**Syntax**

```
object.WriteLine ( [string] )
```

**Arguments**

| Parameter | Description |
|---|---|
| *string* | Optional. The text you want to write to the file. If omitted, a newline character is written to the file. |

**Note**

- Writes a string immediately followed by a newline character to a text file.

**Example**

```
oTxtFile.WriteLine ("This is line 1.")
oTxtFile.WriteLine ("This is line 2.")
```

The resulting text file looks like this:

```
This is line 1.
This is line 2.
```

## Q&A

## How to Enumerate Folders and Folder Properties?

The primary advantage of using scripts for file system management is the fact that scripts can carry out tasks that would be too tedious and time-consuming to perform using either the graphical user interface or a command-line tool. For Example, you have to test that requires you to verify the following in various clients:

- A folder named Scripts exists on each of client.
- The Scripts folder is hidden.
- The Scripts folder is marked as read-only.
- The Scripts folder is compressed.

Scripts can be used to carry out tasks such as these because, within the Windows shell, folders are actually **COM** objects. As **COM** objects, folders have properties that can be retrieved, properties that answer questions such as:

- Is this folder hidden?
- Is this folder read-only?
- Is this folder compressed?

You can retrieve the properties of any folder in the file system using the **Win32_Directory** class. More information about the **WMI** object, properties an methods can be found in: **Win32_Directory Class** on page 41

To retrieve the properties for a single folder, construct a Windows Query Language (WQL) query for the **Win32_Directory** class, making sure that you include the name of the folder. For example, this query binds to the folder D:\Archive:

```
" SELECT * FROM Win32_Directory WHERE Name = 'D:\\Archive'"
```

**Remark**: When specifying a file or folder name in a WQL query, be sure you use two backslashes (\\) to separate path components.

The following sample code contains a script that retrieves properties for the folder C:\Scripts. To carry out this task, the script must perform the following steps:

1. Create a variable to specify the computer name.
2. **WMI** Namespace
   a. Use a **GetObject** call to connect to the **WMI** namespace root\cimv2, and set the impersonation level to "impersonate."
   b. Use a **GetObject** method call to the **WMI** namespace winmgmts:
3. Query
   a. Use the **ExecQuery** method to query the **Win32_Directory** class.
   b. Use the **Get** method to query the **Win32_Directory** class
4. To limit data retrieval to a specified folder, a Where clause is included restricting the returned folders to those where Name equals C:\\Scripts. You must include both backslashes (\\) in the specified name.

```
Set oFolder = GetObject("winmgmts:").Get _
            ("Win32_Directory.Name='F:\\Music Library'")
MsgBox "Archive: " & oFolder.Archive
MsgBox "Caption: " & oFolder.Caption
MsgBox "Compressed: " & oFolder.Compressed
MsgBox "Compression method: " & oFolder.CompressionMethod
MsgBox "Creation date: " & oFolder.CreationDate
MsgBox "Encrypted: " & oFolder.Encrypted
MsgBox "Encryption method: " & oFolder.EncryptionMethod
MsgBox "Hidden: " & oFolder.Hidden
MsgBox "In use count: " & oFolder.InUseCount
MsgBox "Last accessed: " & oFolder.LastAccessed
MsgBox "Last modified: " & oFolder.LastModified
MsgBox "Name: " & oFolder.Name
MsgBox "Path: " & oFolder.Path
MsgBox "Readable: " & oFolder.Readable
MsgBox "System: " & oFolder.System
```

## How to Enumerate All the Folders on a Computer?

If you need to enumerate all the folders on a computer, be aware that this query can take an extended time to complete. For example, on a Windows 2000-based computer with 5,788 folders, a script that returns the name of each folder required 429 seconds to complete.

The next example contains a script that returns a list of all of the folders on a computer. To  carry out this task, the script must perform the following steps:

- Use a **GetObject** call to connect to the WMI namespace winmgmts:
- Use the **InstancesOf** method to query the **Win32_Directory** class.
- This returns a collection of all the folders on the computer.
- For each folder in the collection, echo the folder name.

```
Set colFolders = GetObject("winmgmts:").InstancesOf("Win32_Directory")
For Each oFolder in colFolders
    Reporter.ReportEvent micGeneral, "Name", oFolder.Name
Next
```

## How to Enumerate the Subfolders of a Folder?

Instead of enumerating all the folders and subfolders on a computer, a more common task is examining the subfolders for a single folder. For example, you might have a folder named Users, and you might encourage your users to store their documents in this folder. Enumerating the subfolders within the Users folder can tell you which users have set up personal folders within that parent folder.

The **Win32_Subdirectory** class is an association class that allows you to associate a folder with its subfolders (or with its parent folder). Association classes typically have very few properties; their purpose is simply to derive the associations between objects. The **Win32_Subdirectory** class, for example, has only two properties

- GroupComponent. Returns the parent folder of a folder.
- PartComponent. Returns the first-level subfolders of a folder

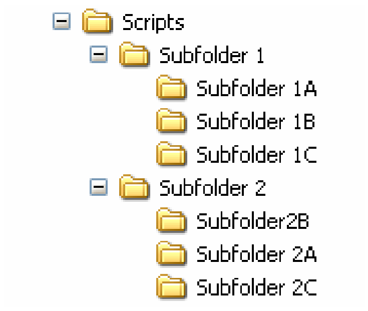


**Figure 22 Sample Folder Structure**

## How to Rename All the Files in a Folder?

## Can i read a text file from the bottom up?

The **FileSystemObject** is extremely useful, but it also has its limitations; one of the major limitations is the fact that it can only read a file from top to bottom. In this case, what we do is go ahead and read the file from top to bottom, beginning with line 1 and ending with line whatever. However, rather than immediately echoing these lines to the screen, we'll store them in an array, with each line in the file representing one element in the array.

```
Option Explicit
Dim arrFileLines()
Dim oFso, oFile
Dim i : i = 0

Set oFso = CreateObject("Scripting.FileSystemObject")
Set oFile = oFso.OpenTextFile("C:\Sample.txt", 1)
Do Until oFile.AtEndOfStream
     Redim Preserve arrFileLines(i)
     arrFileLines(i) = oFile.ReadLine
     i = i + 1
Loop
oFile.Close
For i = Ubound(arrFileLines) to LBound(arrFileLines) Step -1
    MsgBox arrFileLines(i)
Next
Set oFso = Nothing
```

## How can i count the number of lines in a text file?

Well, we begin by using the **FileSystemObject** to open the file for reading. Next we simply read the entire text file, using the ReadAll method. When we use ReadAll, we read in every line of the text file. Because the FileSystem object can only read from the beginning of a file to the end of the file, that means that when ReadAll is finished we must be on the very last line of the file; it's impossible for us to be anywhere else. Consequently, all we have to do is echo the value of the Line property, which reports the line number of the current line. Because we are on the last line, the Line property in this case also tells us the number of lines in the file. Simple.

**Scripting Guys Remark:** Of course, you might be thinking, "Oh, sure, open up and read an entire file just to get the line count? How long will that take?" Surprisingly enough, not very long at all. We tested this script on a text file with just over 20,000 lines. On a regular old laptop computer (2.39 GhZ, 512 MB of RAM) the script took 1 second to complete. Your results might vary, but they probably won't vary by much.

```
Function NumberOfLines (ByVal sFileName)
  Const ForReading = 1
  Dim oFso, oTxtFile
  Set oFso = CreateObject("Scripting.FileSystemObject")
  Set oTxtFile = oFso.OpenTextFile(sFileName, ForReading)
  oTxt.ReadAll
  NumberOfLines = oTxtFile.Line
  Set oTxtFile = Nothing : Set oFso = Nothing
End Function
```

## How can i count the number of times a word appears in a log file?

Your first thought was to use the **InStr** function to see if for example, the word *Failure* appears anywhere in each line of the log file; you could then keep a running tally of the number of times you found the word. The simplest and most effective way, is to use regular expression. For more details see Chapter 05

```
Function MatchesFound (ByVal sFileName, ByVal sString, ByVal bIgnoreCase)
  Const FOR_READING = 1
  Dim oFso, oTxtFile, sReadTxt, oRegEx, oMatches
  Set oFso = CreateObject("Scripting.FileSystemObject")
  Set oTxtFile = oFso.OpenTextFile(sFileName, FOR_READING)
  sReadTxt = oTxtFile.ReadAll
  Set oRegEx = New RegExp
  oRegEx.Pattern = sString
  oRegEx.IgnoreCase = bIgnoreCase
  oRegEx.Global = True
  Set oMatches = oRegEx.Execute(sReadTxt)
  MatchesFound = oMatches.Count
  Set oTxtFile = Nothing : Set oFso = Nothing : Set oRegEx = Nothing
End Function
```

# Appendix 5.A

## IOMode Arguments

| Constant | Value | Modality |
|---|---|---|
| **ForReading** | 1 | Open a file for reading only. You can't write to this file. |
| **ForWriting** | 2 | Open a file for writing. |
| **ForAppending** | 8 | Open a file and write to the end of the file. |

**Table 2 IOMode Arguments**

## Format Arguments

| Constant | Value | Modality |
|---|---|---|
| **TristateTrue** | -1 | Open the file as Unicode. |
| **TristateFalse** | 0 | Open the file as ASCII. |
| **TristateUseDefault** | -2 | Open the file using the system default. |

**Table 3 - Format Arguments**

## DriveType Constants

| Constant | Value | Description |
|---|---|---|
| Hidden | 2 | Indicates that the folder is hidden, and not visible by default in My Computer or Windows Explorer. |
| System | 4 | Indicates that the folder is a System folder. In general, it is a good idea not to modify the properties of a system folder. |
| Directory | 16 | Standard value applied to all folders. All folders accessed by the FileSystemObject will have, at a minimum, the bit value 16. |
| Archive | 32 | Archive bit used by backup programs to determine the files and folders that need to be backed up. Enabling the archive bit will ensure that the folder is backed up during the next incremental backup. Disabling the archive bit will prevent the folder from being backed up during the next incremental backup. |
| Compressed | 2048 | Indicates whether Windows compression has been used on the folder. |

**Table 4 - DriveType Constants**

## File Attributes Used by the FileSystemObject

| Constant | Value | Description |
|---|---|---|
| Normal | 0 | File with no attributes set. |
| Read-only | 1 | File can be read but cannot be modified. |
| Hidden | 2 | File is hidden from view in Windows Explorer or My Computer. |
| System | 4 | File is needed by the operating system. |
| Archive | 32 | File is flagged as requiring backup. |
| Alias | 64 | File is a shortcut to another file. |
| Compressed | 2048 | File has been compressed. |

**Table 5 - File Attributes Used by the FileSystemObject**