The

Guide to:

Managing Files & Folders with Windows PowerShell

The

Guide to:
Managing Files & Folders with PowerShell

by **Zeaun Zarrieff**

The **One Hour Expert** Guide to Managing Files & Folders with *Windows PowerShell*

By Zeaun Zarrieff, MCITP EA

Information Technology Expert

Gigatas, Inc.

30 S. Michigan

Chicago, IL 60603

Published in the United States of America

Table of Contents

Preface: Why PowerShell?

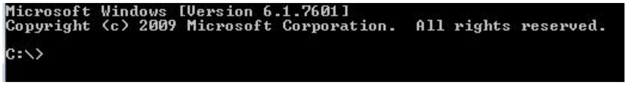
If you are reading this, then you must already have some knowledge of what Windows PowerShell is. What you may not know is the sheer power that PowerShell places at your very fingertips for monitoring, manipulating, and crafting your Microsoft Windows experience. Of course Windows has had scripting and command line solutions for years (Batch files, VBScript, WMI, etc.) but PowerShell has taken a step that no other scripting language has ever taken. PowerShell does away with abstract language and cryptic code and instead exposes raw scripting power with human-readable and intuitive commands (called affectionately CMDlets).

Rather than waste your time with case studies, testimonials, and other information in an effort to sway your opinion in favor of PowerShell, I'd rather dive right in to the nitty gritty of the matter. Let's go.

Consider the following comparison examples:

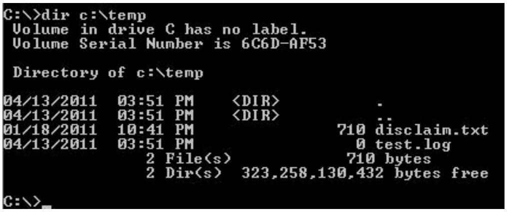
Let's say I want to see the contents of a directory using the standard Windows interface. I'd most likely do so by double-clicking My Computer, then browsing to the directory in question. It's an okay method if I only want to look at what is there, but that's about it.

Consider instead if I want to look at the contents of a directory using the old windows command line (CMD.exe). I would probably start at a prompt like this:



```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\>
```

From there I would type the "dir" command, followed by the directory I want to see the contents of. In this example I'll use the directory "c:\temp", as indicated below:

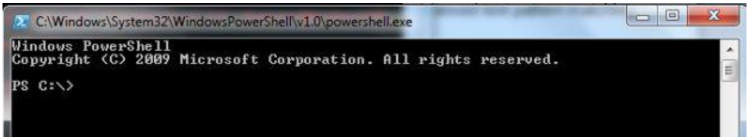


```
C:\>dir c:\temp
 Volume in drive C has no label.
 Volume Serial Number is 6C6D-AF53

 Directory of c:\temp

04/13/2011  03:51 PM    <DIR>          .
04/13/2011  03:51 PM    <DIR>          ..
01/18/2011  10:41 PM               710 disclaim.txt
04/13/2011  03:51 PM                 0 test.log
               2 File(s)            710 bytes
               2 Dir(s)  323,258,130,432 bytes free

C:\>_
```

Great. Now I know what's there, but what if I want to actually do something with the information you have gathered, such as create a log file listing information about the files found there. If all you want is a list of file names, then you would do something like "dir c:\temp /b > files.txt". But what if you want to include other info, like file size, creation date, last-modified date, or other things? You would need to employ some rather advanced text pattern matching (also called text parsing) to pull out the info you need.

PowerShell simplifies this process. Go ahead and open PowerShell. If you are using Windows XP or Server 2003, click Start > Run then type PowerShell and press Enter. If you are using Windows Vista or Windows 7, click Start and then type PowerShell and press Enter.

You should see a window like this:

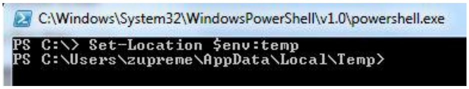

C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe

```
Windows PowerShell
Copyright (C) 2009 Microsoft Corporation. All rights reserved.

PS C:\>
```

Welcome to PowerShell. Now let's go to your profile's temp directory. In this directory we can safely create and delete stuff without worrying about damaging your system. Type the following to go there:

Set-Location $env:temp

Great. Now you should see something like:


C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe

```
PS C:\> Set-Location $env:temp
PS C:\Users\zupreme\AppData\Local\Temp>
```

This indicates that you are now in your personal temp directory. Let's see what's in there by typing:

Get-ChildItem

Did you get all that? The folder probably has lots of stuff in it and it all went by very quickly, but you probably noticed that it looks very similar to what you saw when we typed "dir" using the old Windows Command Line. But that's where the similarity ends. In PowerShell the Get-ChildItem command doesn't return just text, it returns "objects". Objects in PowerShell are just like objects in real life in one key way: They have attributes. Files, for example have a length, they have a creation date, they each have a name, a language, and much more. PowerShell makes it very easy to isolate this information and do stuff with it in ways that are almost impossible using the old windows command line or the regular windows point-and-click interface. What if you want to separate all of the files in a directory into separate directories based on file type, or on creation date, or on the letter of the alphabet the file starts with? PowerShell makes these common types of tasks very easy.

If you will just set aside one hour, and give this small book your undivided attention, you will be an Expert at managing files and folders with Windows PowerShell.

File Properties

If you are reading this book, I assume that you already have some knowledge of the Windows file system. If you don't, I suggest that you pick up the One Hour Expert Guide to Windows Files & Folders for a primer.

PowerShell, being an object-oriented command line, let's you get very familiar with what's happening "under the hood" of the filesystem. Go ahead and open PowerShell (if you don't already have it open) and go to your temp directory. Remember that you can do this by typing the following:

```
Set-Location $env:temp
```

No let's quickly create a file that we can use for analysis. Please type the following to create this file:

```
Echo text > text.txt
```

This should have created a file called "text.txt" containing the term "text". Let's verify this by typing the following:

```
Test-Path text.txt
```

If the file was created successfully PowerShell should have responded to the command above with the word "True". If it came back as "False" please try creating the file again.

The "Test-Path" command is how, in PowerShell, you can verify the existence of any file or folder. Let's say that you want to quickly check to see if there are any text files in a given directory, or any files containing a given term in the file name. Test-Path lets you do this easily.

If I want to check for the existence of any text files in the current directory, I can type:

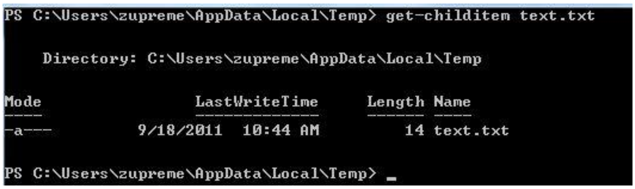
```
Test-Path *.txt
```

The star, properly called an "Asterisk" is what, in the command line world, we call a "Wildcard". A wildcard essentially matches any text. So the wildcard term, "*.txt" would match any file with a "txt" file extension. If there are any files matching your search term in the directory you searched PowerShell returns a "True". If there are none present, PowerShell returns a "False".

Now, since let's look more closely at the file we created earlier. Please type the following into PowerShell:

```
Get-ChildItem text.txt
```

You should see something similar to:



```
PS C:\Users\zupreme\AppData\Local\Temp> get-childitem text.txt

    Directory: C:\Users\zupreme\AppData\Local\Temp

Mode                LastWriteTime     Length Name
----                -------------     ------ ----
-a---         9/18/2011  10:44 AM         14 text.txt

PS C:\Users\zupreme\AppData\Local\Temp> _
```

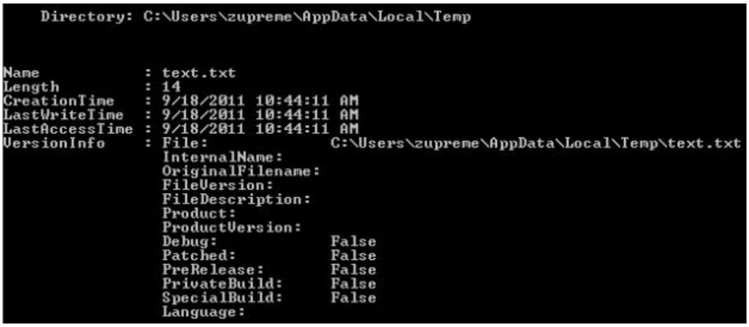
That's great info. You might be wondering what "Length" means. No it doesn't mean how many words or letters are in the file. It's PowerShell's way of referring to the file size in bytes (bytes are how computers measure size).

Now let's see what other properties are there, that we don't currently see. Please type:

Get-ChildItem text.txt | fl

Note: The symbol separating "text.txt" and "fl" is called a "pipe". It is found on mist keyboards just above the enter key, on the same button as the backslash. You can type it by holding "SHIFT" and pressing the backslash key. The Pipe essentially takes whatever is to the left of the Pipe and does with it whatever is on the right of it. In this case we are using the "fl" command to display the full output of the "Get-ChildItem text.txt" command.

You should now see something like this:



```
    Directory: C:\Users\zupreme\AppData\Local\Temp


Name           : text.txt
Length         : 14
CreationTime   : 9/18/2011 10:44:11 AM
LastWriteTime  : 9/18/2011 10:44:11 AM
LastAccessTime : 9/18/2011 10:44:11 AM
VersionInfo    : File:             C:\Users\zupreme\AppData\Local\Temp\text.txt
                 InternalName:
                 OriginalFilename:
                 FileVersion:
                 FileDescription:
                 Product:
                 ProductVersion:
                 Debug:            False
                 Patched:          False
                 PreRelease:       False
                 PrivateBuild:     False
                 SpecialBuild:     False
                 Language:
```

Not bad eh? PowerShell gives you all of that info in an easy-to-read format. More important than how easily it reads is how easily you can extract that info in a usable way. But first let's learn how to get the contents of a file. You can easily do this like so:

Get-Content text.txt

Now let's display to the screen the file name, the file size, and the contents. Do this by typing the following:

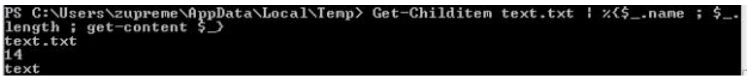
Get-ChildItem text.txt | %{$_.name ; $_.length ; get-content $_}

Don't get intimidated by this command string. It's really very easy to understand.

Let’s take it one element at a time.

- **Get-ChildItem**
- Is the actual command. It gets a file in the current directory.
- **Text.txt**
- Is the file we are getting the properties of.
- “|”
- Is the Pipe. It takes what is to the left of it and does what is to the right of it.
- **“%”**
- Is the percent sign. In PowerShell it means to repeat what comes after it for each item passed to it by the pipe. It will come in handy when we need to repeat tasks over and over again for multiple files or folders.
- “**{“ and “}”**
- Tells PowerShell where the commands to be repeated, or in this example applied, begin and end.
- “**$_.name”**
- Means the name of the file being worked on.
- “**$_.length”**
- Means the file size of the current file.
- “**Get-Content”**
- Gets the contents of the current file.
- **“$_”**
- means the current file and in this case tells “Get-Content”
- **The semicolons (“;”)**
- separate commands and essentially tell PowerShell to do what is to the left of it, then do what is to the right of it, in that order.

You should have gotten the following as output:



```
PS C:\Users\zupreme\AppData\Local\Temp> Get-Childitem text.txt | %{$_.name ; $_.
length ; get-content $_}
text.txt
14
text
```

Not bad, but only you will know what any of the output means if you don’t put labels on each line. Let’s revise our earlier command as follows:

```
Get-ChildItem text.txt | %{“File Name: “ + $_.name ; “File Size: ” + $_.length ; “Data: ” ; get-content $_}
```

Now you have a well laid out template for a file report. Test this by going back to PowerShell, hitting the up arrow on your keyboard (this returns you to last command run) and changing the “test.txt” to “*.log”. Assuming that your temp directory contains some log files, as most do, then you should get a report on all of the log files in the current directory. You can make this more readable by adding a short horizontal line between entries. You can do so by adding “-----” (including the quotes) as an additional command in your string. When typed fully this will look like:

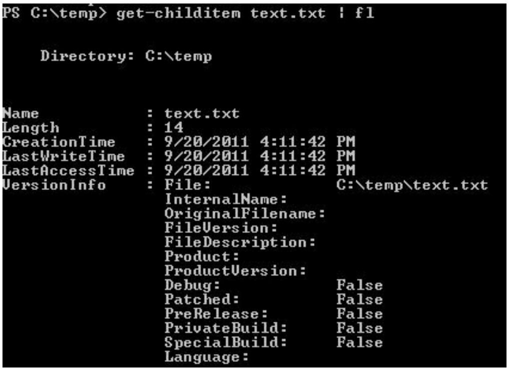
Get-ChildItem *.log | %{“File Name: “ + $_.name ; “File Size: ” + $_.length ; “Data: ” ; get-content $_ ; “-----“}

See how much more readable that was? If you would rather see the output one page at a time just add another Pipe and the “More” command. The full command string now looks like:

Get-ChildItem *.log | %{“File Name: “ + $_.name ; “File Size: ” + $_.length ; “Data: ” ; get-content $_ ; “-----“} | more

Even better. Now that you understand all of that let’s go back to an earlier command:

Get-ChildItem text.txt | fl



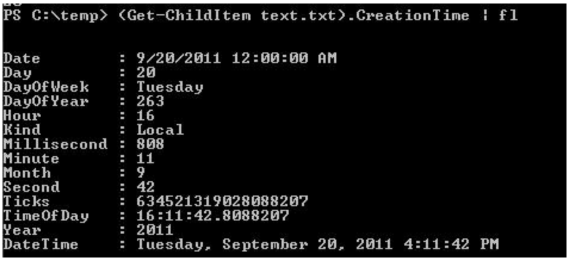
```
PS C:\temp> get-childitem text.txt | fl


    Directory: C:\temp


Name           : text.txt
Length         : 14
CreationTime   : 9/20/2011 4:11:42 PM
LastWriteTime  : 9/20/2011 4:11:42 PM
LastAccessTime : 9/20/2011 4:11:42 PM
VersionInfo    : File:             C:\temp\text.txt
                 InternalName:
                 OriginalFilename:
                 FileVersion:
                 FileDescription:
                 Product:
                 ProductVersion:
                 Debug:            False
                 Patched:          False
                 PreRelease:       False
                 PrivateBuild:     False
                 SpecialBuild:     False
                 Language:
```

This command shows you the metadata for the file in question. Metadata means simply “Information about the Data”. The CreationTime entry, for example, lets you know when the file was created. “That’s great but the DIR command already gave me that”, you might say.

True, but the DIR command doesn’t give you the day of the month, the day of the week, the day of the year, the hour, the minute, the second, and more already broken out and ready to use. Since PowerShell is an Object-Oriented Command Line, even the attributes of an object are often objects themselves, which have their own attributes.

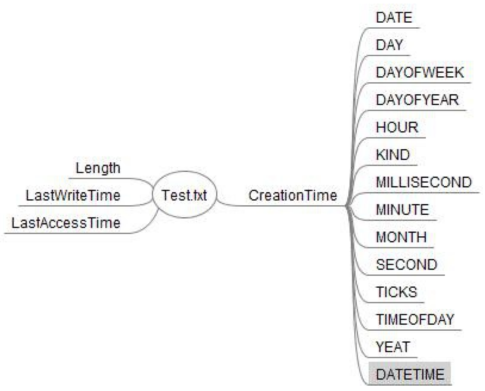
You can run the following command to verify this:

(Get-ChildItem text.txt).CreationTime | fl



```
PS C:\temp> (Get-ChildItem text.txt).CreationTime | fl


Date        : 9/20/2011 12:00:00 AM
Day         : 20
DayOfWeek   : Tuesday
DayOfYear   : 263
Hour        : 16
Kind        : Local
Millisecond : 808
Minute      : 11
Month       : 9
Second      : 42
Ticks       : 634521319028088207
TimeOfDay   : 16:11:42.8088207
Year        : 2011
DateTime    : Tuesday, September 20, 2011 4:11:42 PM
```

If you are a very visual-minded person, as I am, then you might better understand this structure by reviewing the diagram below:



You can see that while “CreationTime” is an attribute of the object “Test.txt”, “Day” is an attribute of the object “CreationTime”. This is how PowerShell handles file and folder attributes and it gives you a great deal of power as a computer user. Imagine if you had a whole folder full of files and you wanted to categorize them according to the date they were created. With PowerShell it becomes easy using the information you now understand. File attributes in PowerShell are a very powerful resource for managing your filesystem.

Moving & Copying Files

What a great world this would be if every file on every computer was created and lived in the same location forever. You'd always know where to find things, you'd never need to move things around, and all would be right with the universe.

Unfortunately, that's not reality. Files and folders get created, deleted, moved and copied all the time. With Windows PowerShell you can really take full control over the placement and movement of your files. Make sure you are in your temp directory (type "cd $env:temp" just to be certain) and then run the following command:

```
Get-ChildItem | %{$abc = $_.name[0] ; new-item -itemtype directory $abc ; move-item $_ $abc}
```

This simple command will take every file or folder in your current directory, will attempt to create a folder with the first letter of the file or folder as the folder name, and will move each file into that folder. Once run, this simple command string will organize, usually in under a second, even the most junk-filled folders.

Now run the command below to see your now-pristinely organized directory:

```
Get-ChildItem
```

You can also run the following command to see the full structure that was created:

```
Get-ChildItem -recurse
```

Now let's put things back, more or less how they were, so that we are ready for the next command:

```
Get-ChildItem -recurse | %{move-item $_.fullname .}
```

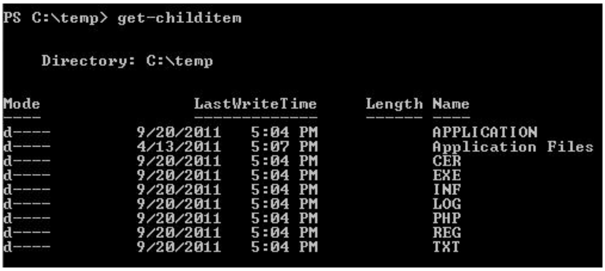
If you run the, "Get-ChildItem" command you should see your files restored to the main temp directory. The alphabetical folders are still there as well, but they should now be empty of files. Let's clean up those alphabet folders now:

```
Get-ChildItem | where-object{$_.name.length -eq 1} | remove-item
```

"But I want to organize the files by file type, not by first letter", you say. No problem. The command would be re-written thus:

Get-ChildItem | %{$ext = $_.name.split(“.”)[1].toupper() ; new-item -itemtype directory $ext ; move-item $_ $ext}

Now you should see directories created for each file extension, with files of that type located appropriately within them.



```
PS C:\temp> get-childitem


    Directory: C:\temp


Mode                LastWriteTime     Length Name
----                -------------     ------ ----
d----         9/20/2011   5:04 PM            APPLICATION
d----         4/13/2011   5:07 PM            Application Files
d----         9/20/2011   5:04 PM            CER
d----         9/20/2011   5:04 PM            EXE
d----         9/20/2011   5:04 PM            INF
d----         9/20/2011   5:04 PM            LOG
d----         9/20/2011   5:04 PM            PHP
d----         9/20/2011   5:04 PM            REG
d----         9/20/2011   5:04 PM            TXT
```

Not bad eh?

Now let’s examine the pieces to the commands given above:

First:

Get-ChildItem | %{$abc = $_.name[0] ; new-item -itemtype directory $abc ; move-item $_ $abc}

- **Get-ChildItem**
- Gets a list of all files and folders in the current directory
- **|**
- The Pipe tells PowerShell to pass the information to the left of it to the command on the right of it. in this case it passes the list of files and folders
- **%**
- The percent sign tells PowerShell to do the same thing to each item in the list of files
- **{**
- The Open Bracket ells PowerShell where the list of commands to run on the objects passed to it by the Pipe begin
- **$abc =**
- Creates a temporary container (called a “Variable”) for the information to the right side of the equal sign
- **$_.name[0]**
- Refers to the first letter of the file name currently being worked on by PowerShell. Remember that to a computer Zero is always the first number in a sequence (not the number One like we human beings prefer)
- **;**
- The semicolon tells PowerShell to run the command to the left of it, then the

command to the right of it

• **New-item**

• Tells PowerShell to create a new item

• –**itemtype directory**

• Tells PowerShell what type of item to create (in this case a directory)

• **$abc**

• Tells PowerShell to name the directory according to the contents of our temporary container (in this case the 1st letter of the file name about to be moved)

• **;**

• The semicolon tells PowerShell to run the command to the left of it, then the command to the right of it

• **Move-Item**

• Tells PowerShell to move an item

• **$_**

• Tells PowerShell which item to move ($_ always refers to the current object we are working on)

• **$abc**

• Tells PowerShell where to move the item (to our newly created folder)

• **}**

• The Close Bracket tells PowerShell where the command string ends

Please review the items listed above very closely. Don't move on until you understand them very well.

Ok next:

Get-ChildItem

1. **Get-ChildItem** tells PowerShell to return a list of all items in the current directory

And next

Get-ChildItem -recurse | %{move-item $_.fullname .}

• **Get-ChildItem –recurse**

• Tells PowerShell to return a list of all items in the current directory and of every directory within it.

• If you really want to see this command at work, just run “get-ChildItem c:\windows –recurse” (just hold CTRL and press the letter “C” when you want it to stop)

• |

• The Pipe tells PowerShell to pass the information to the left of it to the command on the right of it. in this case it passes the list of files and folders

• %

• The percent sign tells PowerShell to do the same thing to each item in the list of files

• {

• The Open Bracket ells PowerShell where the list of commands to run on the objects passed to it by the Pipe begin

• Move-item

• Tells PowerShell to move an item

• $_.fullname

• Tells PowerShell which item to move

• We are using the FullName attribute instead of the Name attribute because the files we want are not in the current directory. The Name attribute doesn’t include the full path, like FullName does.

• .

• The Dot, or Period, is used in PowerShell to indicate the directory you are currently working in. In this instance it tells PowerShell where to move the file you are currently working on.

• }

• The Close Bracket tells PowerShell where the command string ends

Next up:

Get-ChildItem | where-object{$_.name.length -eq 1} | remove-item

• Get-ChildItem

• tells PowerShell to return a list of all items in the current directory

• |

• The Pipe tells PowerShell to pass the information to the left of it to the command on the right of it. In this case it passes the list of files and folders in the current directory

• Where-object

• Tells PowerShell to only run the command after the next Pipe on items matching the parameters inside the brackets

• {

• The Open Bracket ells PowerShell where the list of parameters we are looking to filter by begins

• **$_.name.length**

• Tells PowerShell to filter items according to the length of the item name

• **–eq 1**

• Tells PowerShell to only work on files where the property to the left of this (in this case the length of the item name) is equal to One.

• **}**

• The Close Bracket tells PowerShell where the filtering parameters end

• **|**

• The Pipe tells PowerShell to pass the information to the left of it to the command on the right of it. In this case it passes the list of files and folders which match our filtering rule

• **Remove-item**

• Tells PowerShell to remove any files or folders which are passed to it by the Pipe

And another One:

Get-ChildItem | %{$ext = $_.name.split(“.”)[1].toupper() ; new-item -itemtype directory $ext ; move-item $_ $ext}

• **Get-ChildItem**

• Gets a list of all files and folders in the current directory

• **|**

• The Pipe tells PowerShell to pass the information to the left of it to the command on the right of it. In this case it passes the list of files and folders

• **%**

• The percent sign tells PowerShell to do the same thing to each item in the list of files

• **{**

• The Open Bracket ells PowerShell where the list of commands to run on the objects passed to it by the Pipe begin

• **$ext =**

• Creates a temporary container (called a “Variable”) for the information to the right side of the equal sign

• **$_.name.split(“.”)[1].toupper()**

• Tells PowerShell to split the file names into what comes before the dot (the actual name) and what comes after the dot (the extension), to grab just the extension, and to convert it to uppercase

• **;**

• The semicolon tells PowerShell to run the command to the left of it, then the command to the right of it

• **New-item**

- Tells PowerShell to create a new item
- –**itemtype directory**
- Tells PowerShell what type of item to create (in this case a directory)
- **$ext**
- Tells PowerShell to name the directory according to the contents of our temporary container (in this case the extension of the file about to be moved)
- **;**
- The semicolon tells PowerShell to run the command to the left of it, then the command to the right of it
- **Move-Item**
- Tells PowerShell to move an item
- **$_**
- Tells PowerShell which item to move ($_ always refers to the current object we are working on)
- **$ext**
- Tells PowerShell where to move the item (to our newly created folder)
- **}**
- The Close Bracket tells PowerShell where the command string ends

If you take your time and review the commands listed above very carefully you will be a wizard at moving files in PowerShell in no time. Of course if you simply want to move a single file to a single destination, you can always type:

Move-item *filename.ext destination*

If you want to move all files of a particular type, say Text Files, to the same destination you could type:

Move-item **.txt destination*

Practice yourself by creation some files and folders, moving them around, then deleting them.

The Copy-Item command works essentially the same as the Move-Item command, with one key exception: It copies the item or items specified instead of moving them, thus it leaves the original copy alone and simply creates a new copy in the location you specify.

The simplest use of this command is:

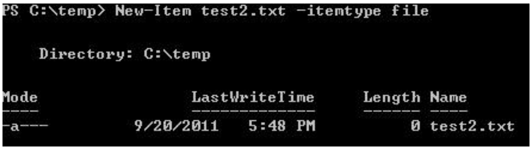
```
Copy-Item c:\myfile.log d:\myfile.log
```

Deleting Files

So you want to delete some files, eh. PowerShell makes this easy. First the simple method:

First create an empty file:

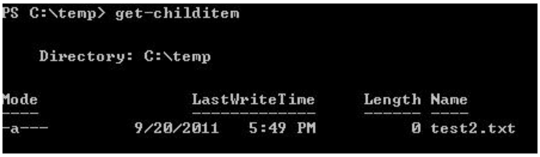
New-Item test2.txt –itemtype file



```
PS C:\temp> New-Item test2.txt -itemtype file

    Directory: C:\temp

Mode                LastWriteTime     Length Name
----                -------------     ------ ----
-a---         9/20/2011   5:48 PM          0 test2.txt
```

Go ahead and verify that the file is there

Get-ChildItem



```
PS C:\temp> get-childitem

    Directory: C:\temp

Mode                LastWriteTime     Length Name
----                -------------     ------ ----
-a---         9/20/2011   5:49 PM          0 test2.txt
```

Now delete the file:

Remove-Item test2.txt

Verify that the file was deleted:

Get-ChildItem

Great job! Now you might be thinking about how to delete everything within a given folder at the same time. Not hard at all:

Get-ChildItem | remove-item -force

The “-force” parameter tells PowerShell that you are sure that you want to delete these items and don’t want to be bugged by confirmation messages.

Of course if you didn’t have any items in that directory the command above won’t do very much. Feel free to create some files and then delete them all at the same time with the command above.

And of course the same Piping and filtering functionality that we have been practicing all along works great for the Remove-item command. Take some time to experiment within your own temp directory. Just be careful not to try deleting things outside of there until you are confident in your skills. PowerShell’s Remove-Item command, used the wrong way, could easily wipe most of the files and data from your computer.

Reading Files

Ok. So now you know how to move files, delete files, copy files, and create files. But what about reading files? Easy Easy Easy.

First create file with stuff in it:

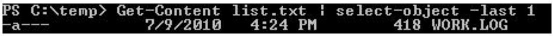
```
Get-ChildItem c:\ > list.txt
```

Now let's read the file:

```
Get-Content list.txt
```

That was simple. But you may not always want to read the whole file. You might just want the last line (which might be the case when looking at log files). In that case you would run:

```
Get-Content list.txt | select-object –last 1
```



```
PS C:\temp> Get-Content list.txt | select-object -last 1
-a---          7/9/2010   4:24 PM        418 WORK.LOG
```

Great. Now we have the last line of the file. If we want the last 10 lines, could you guess what we would type?

```
Get-Content list.txt | select-object –last 10
```

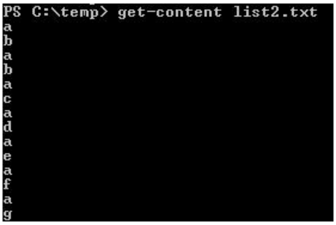
That gives us the last 10 lines, as you would expect. What if, instead, you had a file with lots of repeating lines, and you only wanted unique lines returned?

Let's first create a file with some repeating lines:

```
("a,b,a,b,a,c,a,d,a,e,a,f,a,g").split(",") > list2.txt
```

This command makes a list of comma-separated letters, splits them into separate lines by comma, and then dumps the list into a file called "List2.txt". Check the file contents with:

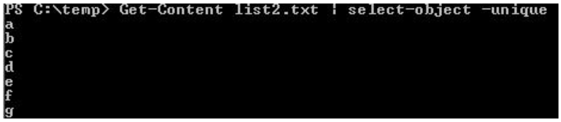
```
Get-Content list2.txt
```



```
PS C:\temp> get-content list2.txt
a
b
a
b
a
c
a
d
a
e
a
f
a
g
```

Now let's return the contents of the file, with the duplicate lines filtered out:

Get-Content list2.txt | select-object –unique



```
PS C:\temp> Get-Content list2.txt | select-object -unique
a
b
c
d
e
f
g
```

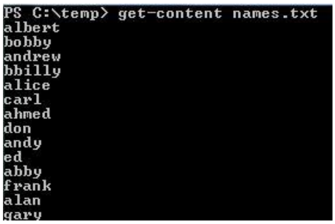
If you wanted to only read the lines which start with a given letter, for example, you could run:

Get-Content list2.txt | where-object{ $_ -like "a*"}

Your options for reading files in PowerShell are almost limitless. This flexibility makes it very easy to also extract information from one file and put that info into another file. For example, create a file with several names in it:

(“albert,bobby,andrew,bbilly,alice,carl,ahmed,don,andy,ed,abby,frank,alan,gary”).split(“,”)
> names.txt

This produces a file which looks like:



```
PS C:\temp> get-content names.txt
albert
bobby
andrew
bbilly
alice
carl
ahmed
don
andy
ed
abby
frank
alan
gary
```

Now, do you remember how to display only lines starting with a given letter?

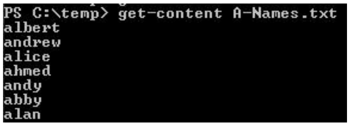
Get-Content names.txt | where-object{ $_ -like "a*"}

Great. Now let’s tell PowerShell to drop all of the names starting with the letter “A” into a file called “A-names.txt”.

get-content names.txt | where-object{ $_ -like "a*"} > A-Names.txt

Now verify the contents of the new file:

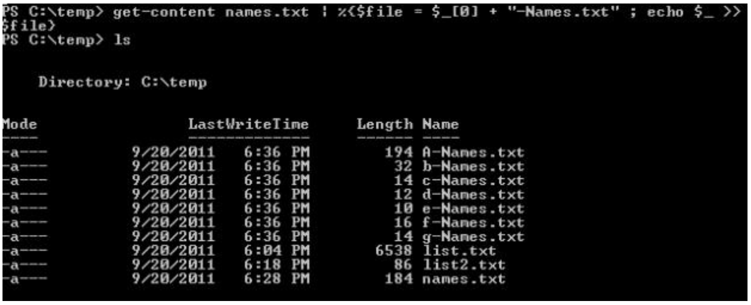
get-content A-Names.txt



```
PS C:\temp> get-content A-Names.txt
albert
andrew
alice
ahmed
andy
abby
alan
```

Good work. But what if you wanted to do that same thing for all of the names in the file? By now you probably could figure this out on your own, but here you go:

get-content names.txt | %{$file = $_[0] + "-Names.txt" ; echo $_ >> $file}



```
PS C:\temp> get-content names.txt | %{$file = $_[0] + "-Names.txt" ; echo $_ >>
$file}
PS C:\temp> ls


    Directory: C:\temp


Mode                LastWriteTime     Length Name
----                -------------     ------ ----
-a---         9/20/2011   6:36 PM        194 A-Names.txt
-a---         9/20/2011   6:36 PM         32 b-Names.txt
-a---         9/20/2011   6:36 PM         14 c-Names.txt
-a---         9/20/2011   6:36 PM         12 d-Names.txt
-a---         9/20/2011   6:36 PM         10 e-Names.txt
-a---         9/20/2011   6:36 PM         16 f-Names.txt
-a---         9/20/2011   6:36 PM         14 g-Names.txt
-a---         9/20/2011   6:04 PM       6538 list.txt
-a---         9/20/2011   6:18 PM         86 list2.txt
-a---         9/20/2011   6:28 PM        184 names.txt
```

Let’s break that one down, just in case you don’t quite understand yet.

- **Get-Content**
- Gets the contents of a file
- **Names.txt**
- Tells PowerShell what file to get the contents of
- **|**
- The Pipe Passes the output of Get-Content to the what’s comes after the Pipe
- **%**
- Tells PowerShell to repeat what comes inside of the brackets for every line of the file being read

- **{**
- Tells PowerShell where the command string starts
- **$file**
- A temporary place-holder, or variable, for the name of the file to be written to
- **=**
- An equal sign tells PowerShell the value of the variable will be whatever is to the right of it
- **$_[0] + “-Names.txt”**
- Tells PowerShell to combine the first character of the line being read with the suffix “-names.txt”
- **;**
- A semicolon tells PowerShell to run the commands to the left of it, followed by the commands to the right
- **Echo $_**
- Returns the data for the current line of the file being read
- **>>**
- Tells PowerShell to append to the output file (a single Greater-Than sign would have overwritten the file instead)
- **$file**
- Tells PowerShell the name of the file to be created and written to
- **}**
- Tells PowerShell where the command string ends.

Note: in the screenshot above I used “LS” instead of “Get-ChildItem”. “LS” is just an alias for Get-ChildItem and can generally be used interchangeably.

Renaming Files

Alas, in spite of our best efforts, we sometimes make mistakes. If you are like me one of those all-too-frequent mistakes is to accidentally type the wrong letter when creating files, folders, and so forth. Fortunately PowerShell makes renaming files and folders as easy as:

Rename-item *this.txt that.txt*

You could rename all files ending with a TXT extension to LOG files like this:

```
Get-ChildItem *.txt | %{$n = $_.basename + ".log"; rename-item $_ $n}
```

Let’s walk through this command string step-by-step:

- **Get-ChildItem *.txt**
- Gets a list of all files and folders in the current directory with TXT extensions
- **|**
- The Pipe tells PowerShell to pass the information to the left of it to the command on the right of it. In this case it passes the list of files and folders
- **%**
- The percent sign tells PowerShell to do the same thing to each item in the list of files
- **{**
- **$n = $_.basename + “.log”**
- Creates a temporary variable (or placeholder) for our filename and adds the LOG extension to it
- **;**
- A semicolon tells PowerShell to run the commands to the left of it, followed by the commands to the right
- Rename-Item $_ $n
- Tells PowerShell to rename the item being currently worked on to the filename we specified in our temporary variable (or placeholder).
- **}**
- Tells PowerShell where the command string ends.

And that’s pretty much all there is to File and Folder Renaming in Windows PowerShell. As always: Practice makes Perfect!

In Closing

You now have the information you need in order to effectively manage files and folders in Windows PowerShell. However, in order to hone your newfound skill and turn it into true expertise you must practice. Your Windows Temp Directory is your playground. Use it as such and feel free to create, destroy, move and change files therein to your heart's content.

Once you are done with managing Files and Folders, be sure to review the other books in the **One Hour Expert** series on Windows PowerShell to gain expert status in other areas as well.