Working with Files Using PowerShell Cmdlets

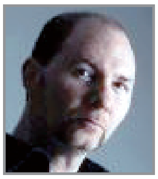
Here are 7 cmdlets that let you easily manage your files

Windows PowerShell offers four different ways to work with files. Your options include using cmdlets, using DOS commands, using Windows Management Instrumentation (WMI), and using Microsoft .NET Framework methods.

- Using cmdlets. There are a number of cmdlets geared specifically toward files. With these cmdlets, you can manage files and file paths as well as work with the contents of files.
- Using DOS commands. PowerShell is fully compatible with DOS commands. Hence, anything that you can do using DOS, you can do with PowerShell. Even the useful xcopy command is recognized by PowerShell.
- Using WMI. WMI offers yet another mechanism for managing files (e.g., changing file properties, searching or renaming a file). Best of all, you can run WMI commands remotely.
- Using Microsoft .NET Framework methods. The .NET System.IO namespace is available through the PowerShell command line. These include the System.IO.File and System.IO.FileInfo classes.

In this discussion, I'll concentrate on the cmdlets geared specifically toward files.The cmdlets you can use for working with files include:

- Get-ChildItem
- Get-Item
- Copy-Item
- Move-Item



Rob Gravelle

resides in Ottawa, Canada, and is the founder of Gravelle Web Design. Rob has built systems for intelligence-related organizations such as Canada Border Services and for commercial businesses.

Email

Gravelle Web Design

Rob Gravelle

- New-Item
- Remove-Item
- Rename-Item

Using the Get-ChildItem Cmdlet

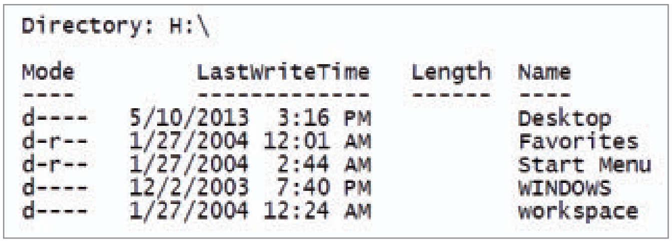
The Get-ChildItem cmdlet retrieves the items found within one or more specified locations. A location can be a file system container, such as a directory, or a location exposed by another provider, such as a registry subtree or certificate store. You can use this cmdlet's -Recurse parameter to get items in all subfolders as well.

Used without parameters, the Get-ChildItem cmdlet retrieves all child items (i.e., subfolders and files) in the current location. For example, if your current location is the H root directory and you run the command

```
Get-ChildItem
```

you'll get results similar to that shown in Figure 1.

Figure 1
Running Get-ChildItem Without Any Parameters



```
Directory: H:\

Mode          LastWriteTime    Length  Name
----          -------------    ------  ----
d----    5/10/2013   3:16 PM           Desktop
d-r--    1/27/2004  12:01 AM           Favorites
d-r--    1/27/2004   2:44 AM           Start Menu
d----    12/2/2003   7:40 PM           WINDOWS
d----    1/27/2004  12:24 AM           workspace
```

By using parameters, you can hone in on the information you need. For example, the following command retrieves all the .log files in the C root directory, including subdirectories:

```
Get-ChildItem C:\* -Include *.log -Recurse -Force
```

As you can see, this command uses the -Include, -Recurse, and -Force parameters. You use the -Include parameter to retrieve specific items. It supports the use of wildcards and is ideal for specifying a filename extension. The -Recurse parameter directs PowerShell to retrieve subfolders in addition to files. The -Force parameter adds hidden files and system files to the output.

Note that, when you run this command, you'll probably get a bunch of access denied errors. Depending on your machine's security settings and policies, some directories (e.g., Recycle Bin, Start menu, user folders) are restricted and can't be read. You can suppress these errors by including the -ErrorAction SilentlyContinue parameter.

The following command will produce the same results as the previous one because the -Path parameter accepts wildcards:

```
Get-ChildItem -Path C:\*.log -Recurse -Force
```

With some PowerShell cmdlet parameters, you can omit the parameter name if you supply that parameter in the position expected by PowerShell. That's the case with the Get-ChildItem cmdlet's -Path parameter. So, the following command would produce the same results as the previous command:

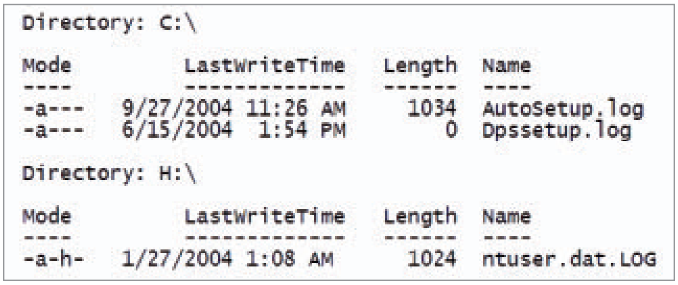
```
Get-ChildItem C:\*.log -Recurse -Force
```

The -Path parameter can accept multiple arguments, separated by comma. For example, suppose that you want to retrieve the .log files from two locations: the C root directory and the H root directory, which is the current directory (i.e., the default location). To accomplish this, you need to include the argument C:\* to get all the log files from the C root directory and the argument * to get all the log files from the H root directory. (Because the H root directory is the default location, you don't need to include H:\.) You need to separate the two arguments with a comma, like this:

```
Get-ChildItem C:\*, * -Include *.log -Force
```

In the sample results in Figure 2, notice the "h" attribute in the Mode column for the H root directory. This attribute denotes that the ntuser.dat.LOG file is hidden. It shows up because the -Force parameter was used.

Figure 2
Running Get-ChildItem with Parameters



```
Directory: C:\

Mode          LastWriteTime    Length  Name
----          -------------    ------  ----
-a---    9/27/2004 11:26 AM      1034  AutoSetup.log
-a---    6/15/2004  1:54 PM         0  Dpssetup.log

Directory: H:\

Mode          LastWriteTime    Length  Name
----          -------------    ------  ----
-a-h-    1/27/2004 1:08  AM      1024  ntuser.dat.LOG
```

Although not shown in these examples, you can refer to Get-ChildItem by aliases. There are three built-in aliases: *dir* (like the DOS *dir* command), *gci*, and *ls* (like the *ls* UNIX command).

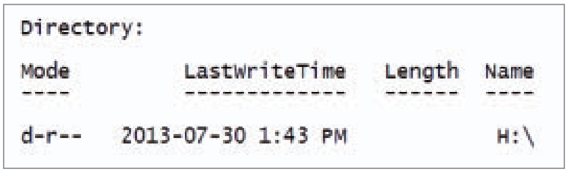
Using the Get-Item Cmdlet

The Get-Item cmdlet retrieves the specified items from the specified locations. Like Get-ChildItem, Get-Item can be used to navigate through different types of data stores. Unlike Get-ChildItem, Get-Item doesn't have a default location, so you must always supply at least one location using the -Path parameter. Although the parameter is required, including the parameter name isn't. For example, here's a simple command that uses a period to retrieve information about the current directory (the H root directory in this case):

```
Get-Item .
```

Figure 3 shows the results.

The Get-Item cmdlet lets you use the wildcard character * to return all the contents of the item (i.e., all the child items). For example, the following command returns all the contents of the current directory (the H root directory in this case). Both the period and asterisk



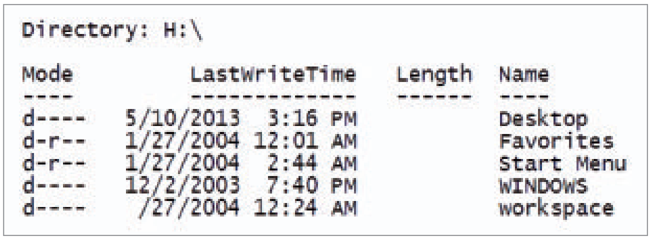
```
Directory:

Mode            LastWriteTime    Length  Name
----            -------------    ------  ----

d-r--    2013-07-30 1:43 PM              H:\
```

Figure 3
Using Get-Item to Retrieve Information About the Current Directory

characters can be used as components in a path, but you must still include backslash folder separators:

```
Get-Item .\*
```

You can see the results in Figure 4.



```
Directory: H:\

Mode            LastWriteTime    Length  Name
----            -------------    ------  ----
d----    5/10/2013   3:16 PM             Desktop
d-r--    1/27/2004 12:01 AM              Favorites
d-r--    1/27/2004   2:44 AM             Start Menu
d----    12/2/2003   7:40 PM             WINDOWS
d----     /27/2004 12:24 AM              workspace
```

Figure 4
Using Get-Item to Return All the Contents of the Current Directory

It's important to understand that all PowerShell cmdlets, including the Get-Item cmdlet, return objects. The Get-Item cmdlet returns System.IO.DirectoryInfo objects, which contain numerous methods and properties you can use. To see those methods and properties, you can send, or pipe, the results of a Get-Item command to the Get-Member cmdlet. If you want to see only the properties, you can run the command:

```
Get-Item . | Get-Member -MemberType Property
```

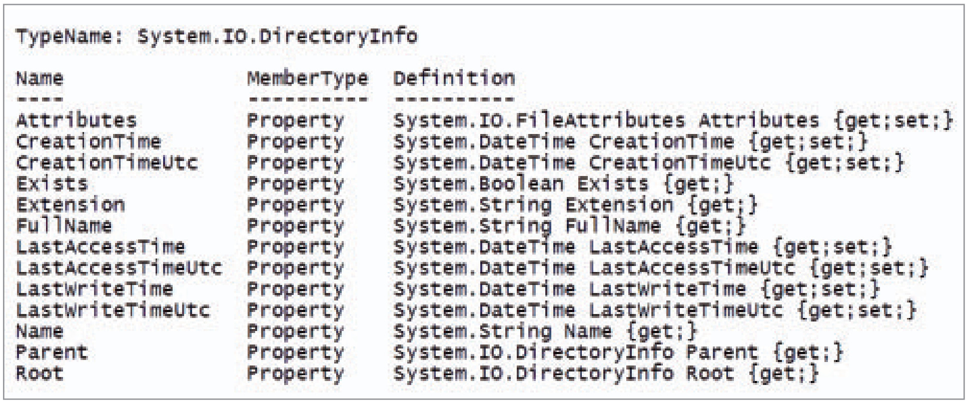
As you can see in Figure 5, there are many properties, including the LastAccessTime property, which returns the date and time when the specified directory was last accessed.

For instance, if you want to find out when the current directory was last accessed, you'd run the command:

```
(Get-Item .).LastAccessTime
```

In this command, notice that the *Get-Item* . call is enclosed in parentheses and that there's a period between the closing parenthesis and LastAccessTime. The parentheses around the *Get-Item* . call cause the returned objects to be stored in memory so that you can perform additional operations on them. In this case, the operation is the retrieval of the returned object's LastAccessTime property value. In PowerShell, you use the dot notation to access object member properties and methods, which is why you need to include the period between the closing parenthesis and LastAccessTime.

Figure 5
Learning About the Properties of the System.IO.DirectoryInfo Object



```
TypeName: System.IO.DirectoryInfo

Name               MemberType  Definition
----               ----------  ----------
Attributes         Property    System.IO.FileAttributes Attributes {get;set;}
CreationTime       Property    System.DateTime CreationTime {get;set;}
CreationTimeUtc    Property    System.DateTime CreationTimeUtc {get;set;}
Exists             Property    System.Boolean Exists {get;}
Extension          Property    System.String Extension {get;}
FullName           Property    System.String FullName {get;}
LastAccessTime     Property    System.DateTime LastAccessTime {get;set;}
LastAccessTimeUtc  Property    System.DateTime LastAccessTimeUtc {get;set;}
LastWriteTime      Property    System.DateTime LastWriteTime {get;set;}
LastWriteTimeUtc   Property    System.DateTime LastWriteTimeUtc {get;set;}
Name               Property    System.String Name {get;}
Parent             Property    System.IO.DirectoryInfo Parent {get;}
Root               Property    System.IO.DirectoryInfo Root {get;}
```

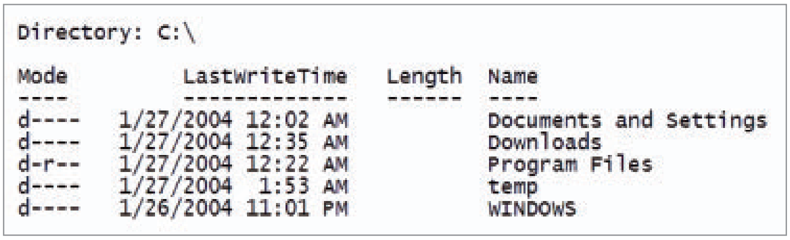
There's a collection of special properties named NoteProperty that you can use to narrow your output to a particular type of object. You can use the Get-Member cmdlet with the -MemberType NoteProperty parameter to learn about the special properties in this collection:

```
Get-Item . | Get-Member -MemberType NoteProperty
```

If you run this command, you'll find that the collection returns six properties: PSChildName, PSDrive, PSIsContainer, PSParentPath, PSPath, and PSProvider. The PSIsContainer NoteProperty tells you whether the object is a container (i.e., a directory). It returns True when the object is a directory and False when it's a file. You can use this property to limit the Get-Item cmdlet's output to directories:

```
Get-Item C:\* | Where-Object { $_.PSIsContainer }
```

Let's take a closer look at this command, whose results are in Figure 6. First, you're piping all the contents of the C root directory to the Where-Object cmdlet, which lets you filter objects. In this case, you're using the PSIsContainer NoteProperty to filter the output so it returns only directories. The $_ automatic variable represents each file object as it is passed to the command through the pipeline. If you're unfamiliar with how to use the Where-Object cmdlet, see "PowerShell Basics: Filtering Objects."



```
Directory: C:\

Mode            LastWriteTime    Length  Name
----            -------------    ------  ----
d----     1/27/2004 12:02 AM             Documents and Settings
d----     1/27/2004 12:35 AM             Downloads
d-r--     1/27/2004 12:22 AM             Program Files
d----     1/27/2004  1:53 AM             temp
d----     1/26/2004 11:01 PM             WINDOWS
```

Figure 6
Limiting the Get-Item Cmdlet's Output to Directories Only

Like Get-ChildItem, you can refer to Get-Item by an alias. Get-Item has one built-in alias: *gi*.

Using the Copy-Item Cmdlet

The Copy-Item cmdlet is PowerShell's implementation of the DOS copy command and the UNIX cp command, except that Copy-Item is designed to work with the data exposed by any provider. The cmdlet's first two parameters are -Path (which you use to specify the item you want to copy) and -Destination (which you use to specify where you want to copy that item). They're positional so the parameter names can be omitted. For example, the following command copies the test.txt file in the C:\Scripts folder to the C:\Backups\Scripts folder:

```
Copy-Item C:\Scripts\test.txt C:\Backups\Scripts
```

The -Path parameter accepts wildcards, so you can copy multiple files at once. For example, this command copies all the files in the C:\Scripts folder to the C:\Backups\Scripts folder:

```
Copy-Item C:\Scripts\* C:\Backups\Scripts
```

To get more fine-grained control over a copying operation, you can use the -Recurse, -Filter, and -Force parameters. For instance, the following command copies all .txt files contained in C:\Scripts to C:\Temp\Text:

```
Copy-Item -Path C:\Scripts -Filter *.txt -Recurse `
  -Destination C:\Temp\Text
```

Note that the backtick at the end of the first line is PowerShell's line continuation character.

With a little wrangling, you can plug the FullName property into the -Path parameter to copy a carefully compiled list of file objects using either the Get-Item or Get-ChildItem cmdlet:

```
Get-ChildItem C:\* -include *.txt |
  Where-Object { $_.PSIsContainer -eq $false -and `
  $_.LastAccessTime -gt ($(Get-Date).AddMonths(-1))} |
  ForEach-Object { Copy-Item $_.FullName C:\Temp}
```

This statement is really three separate commands combined. The first command (i.e., the command on the first line) retrieves all the .txt files in the C root directory. The second command (i.e., the command on the second and third lines) then whittles down the list of text files so that it contains only the file objects whose LastAccessTime property is greater than one month ago. The third command (i.e., the command on the last line) inserts each filename into the Copy-Item's

-Path property using the ForEach-Object cmdlet. If you're unfamiliar with how to use the ForEach-Object cmdlet, see "PowerShell Basics: Filtering Objects."

Too complicated for your tastes? You'll be happy to know that you can accept input from the pipeline. Just be sure to include the -Destination parameter name so that Copy-Item knows what to do with the input, because that parameter isn't in the expected position:

```
Get-ChildItem C:\* -Include *.log |
  Copy-Item -Destination C:\Temp
```

Although not shown in these examples, you can refer to Copy-Item by aliases. There are three built-in aliases: *copy*, *cp*, and *cpi*.

Using the Move-Item Cmdlet

The Move-Item cmdlet is similar to the Copy-Item cmdlet. In fact, if you replace Copy-Item with Move-Item in any of the commands in the previous section, the commands will behave in much the same way, except that the original files will be deleted in the source folder.

However, there's one notable difference. If you run the same Copy-Item command twice, you'll find that PowerShell overwrites the existing file in the destination folder without any warning. The Move-Item cmdlet is more cautious is this regard and will throw an error instead. For example, if you run the command

```
Get-ChildItem C:\* -Include *.txt |
  Where-Object `
  { $_.LastAccessTime -gt ($(Get-Date).AddMonths(-1))} |
  ForEach-Object { Move-Item $_.FullName C:\Temp}
```

you'll receive the error *Cannot create a file when that file already exists*. Using the -Force parameter will modify this behavior so that Move-Item overwrites the existing file.

In addition to the -Force parameter, you can use the -Recurse and -Filter parameters in your Move-Item commands to fine-tune them. For example, the following command moves the text files in the C:\Scripts folder and its subfolders to the C:\Temp\Text folder. In this case, you need to include the -Destination parameter name because you're not using that parameter in the position that PowerShell expects:

```
Move-Item C:\Scripts -Filter *.txt -Recurse `
  -Destination C:\Temp\Text
```

Like Copy-Item, Move-Item has three built-in aliases. Those aliases are *move*, *mv*, and *mi*.

Using the New-Item Cmdlet

The New-Item cmdlet performs the dual role of directory and file creator. (It can also create registry keys and entries in the registry.) When you want to create a file, you need to include the -Path parameter and the -ItemType parameter. As you've seen before, the -Path parameter is positional, so the -Path parameter name isn't required as long as you specify the path and name (i.e., pathname) immediately after the cmdlet name. You must also include the -ItemType parameter with the "file" flag. Here's an example:

```
New-Item 'C:\Documents and Settings\Nate\file.txt' `
  -ItemType "file"
```

The -Path parameter can accept an array of strings so that you can create multiple files at once. You just need to separate the paths with commas. In addition, you need to put the *-ItemType "file"* parameter first, which means you also need to include the -Path parameter name because it's no longer the first parameter after the cmdlet name:

```
New-Item -ItemType "file" -Path "C:\Temp\test.txt", `
  "C:\Documents and Settings\Nate\file.txt", `
  "C:\Test\Logs\test.log"
```

If a file with the exact same pathname already exists, you'll get an error. However, you can include the -Force parameter so that New-Item will overwrite the existing file.

What's really interesting about the New-Item cmdlet is that it lets you insert text into a file by means of the -Value parameter:

```
New-Item 'C:\Documents and Settings\Nate\file.txt' `
  -ItemType "file" -Force `
  -Value "Here is some text for my new file."
```

Remember to include the -Force parameter if the file already exists. Otherwise, you'll receive an error.

The -Value parameter can accept piped input, which is a great way to redirect the output of other cmdlets to a file. You just need to convert the output objects to a string using the Out-String cmdlet. (If you don't do this, New-Item will create a new file for each object.) For example, this command retrieves information about all the files in the C root directory, converts the file information to a string, then writes that information to the H:\C Listing.txt file:

```
Get-ChildItem C:\* | Out-String |
  New-Item -Path "H:\C Listing.txt" -ItemType "file" -Force
```

The New-Item cmdlet has only one built-in alias: *ni*.

Using the Remove-Item Cmdlet

The Remove-Item cmdlet does exactly what you'd expect: It permanently deletes a resource from the specified drive. By permanently, I mean that it doesn't transfer the resource to the Recycle Bin. Hence, if you use Remove-Item to delete a file, there's no way to retrieve it, other than through a file restore program.

You specify which file to delete with the Remove-Item cmdlet's -Path parameter. It's positional, so you don't need to include the -Path parameter name if the pathname immediately follows the cmdlet name. For example, here's a command to delete the test.txt file previously copied to the C:\Backups\Scripts folder:

```
Remove-Item "C:\Backups\Scripts\test.txt"
```

Let's take a look at another example. The following command removes all the .txt files (as indicated by the -Include parameter) in the C:\Scripts folder, except for any files that have the string value test anywhere in the filename (as indicated by the -Exclude parameter):

```
Remove-Item C:\Scripts\* -Include *.txt -Exclude *test*
```

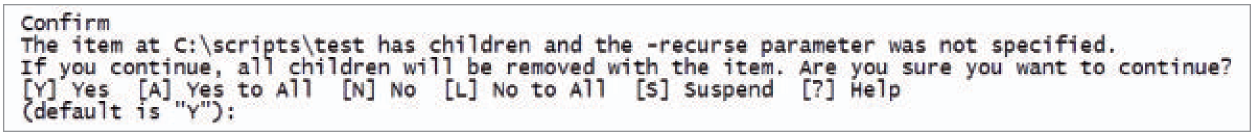
Being such an inherently dangerous tool, Remove-Item comes with a couple of fail-safes. First, if you attempt to delete everything from a folder that contains non-empty subfolders, you'll get a Confirm prompt. For instance, suppose that C:\Scripts contains non-empty subfolders and you run the command:

```
Remove-Item C:\Scripts\*
```

You'll be asked to confirm that you want to delete the non-empty subfolders, as Figure 7 shows.

If you want to run a script that uses Remove-Item to delete the entire contents of a folder, including the contents in subfolders, you

Figure 7
Receiving a Confirm Prompt When Using Remove-Item



```
Confirm
The item at C:\scripts\test has children and the -recurse parameter was not specified.
If you continue, all children will be removed with the item. Are you sure you want to continue?
[Y] Yes  [A] Yes to All  [N] No  [L] No to All  [S] Suspend  [?] Help
(default is "Y"):
```

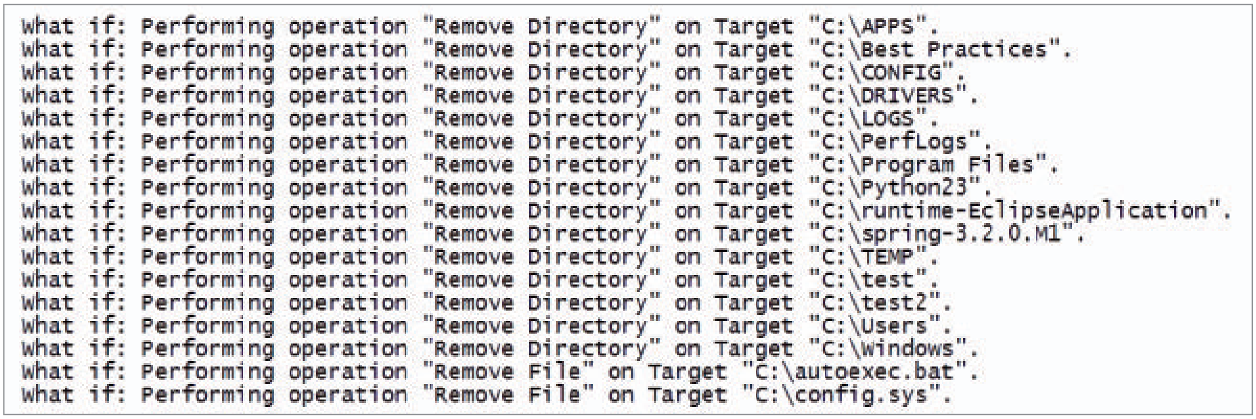
need a way to have Remove-Item run without any user interaction. The way to do that is to include the -Recurse flag.

The Remove-Item cmdlet's second fail-safe is the -WhatIf parameter. If you include this parameter in a Remove-Item command, PowerShell will display what items would be deleted instead of actually deleting them. Due to the destructive nature of delete operations, it's an especially good idea to try your Remove-Item commands with the -WhatIf parameter first, like this:

```
Remove-Item c:\* -Recurse -WhatIf
```

Figure 8 shows sample results. Note that your results might include an error statement along the lines of *Cannot remove the item at 'C:\Users' because it is in use*. This occurs if the current working directory is a subfolder of the directory you're trying to remove (in this example, a subfolder under the C root directory).

Figure 8
Using Remove-Item's -WhatIf Parameter



```
What if: Performing operation "Remove Directory" on Target "C:\APPS".
What if: Performing operation "Remove Directory" on Target "C:\Best Practices".
What if: Performing operation "Remove Directory" on Target "C:\CONFIG".
What if: Performing operation "Remove Directory" on Target "C:\DRIVERS".
What if: Performing operation "Remove Directory" on Target "C:\LOGS".
What if: Performing operation "Remove Directory" on Target "C:\PerfLogs".
What if: Performing operation "Remove Directory" on Target "C:\Program Files".
What if: Performing operation "Remove Directory" on Target "C:\Python23".
What if: Performing operation "Remove Directory" on Target "C:\runtime-EclipseApplication".
What if: Performing operation "Remove Directory" on Target "C:\spring-3.2.0.M1".
What if: Performing operation "Remove Directory" on Target "C:\TEMP".
What if: Performing operation "Remove Directory" on Target "C:\test".
What if: Performing operation "Remove Directory" on Target "C:\test2".
What if: Performing operation "Remove Directory" on Target "C:\Users".
What if: Performing operation "Remove Directory" on Target "C:\Windows".
What if: Performing operation "Remove File" on Target "C:\autoexec.bat".
What if: Performing operation "Remove File" on Target "C:\config.sys".
```

When it comes to aliases, Remove-Item is in a league of its own. It has six built-in aliases: *del*, *erase*, *rd*, *ri*, *rm*, and *rmdir*.

Using the Rename-Item Cmdlet

The Rename-Item cmdlet is handy when you want to rename a resource within a PowerShell provider namespace. The Rename-Item cmdlet's first parameter is -Path and its second parameter is -NewName. As its name suggests, the -NewName parameter specifies the new name for the resource. It's important to note that this parameter expects the name only, without the path. If Rename-Item detects a path, it'll throw an error. For example, if you want to rename the *C Listing.txt* file in the H root directory to *c_listing.txt*, you'd run the command:

```
Rename-Item -Path "H:\C Listing.txt" -NewName c_listing.txt
```

Because -Path and -NewName are positional parameters, you can omit the parameter names as long as they're in the expected positions:

```
Rename-Item "H:\C Listing.txt" c_listing.txt
```

One limitation of the Rename-Item cmdlet is that the -NewName parameter expects a single string without wildcards. However, you can work around this by iterating through items in a directory. You just need to pipe the Get-ChildItem cmdlet's output to the -Path parameter and include the -NewName parameter.

For example, here's a command that iterates through all the files in the current directory and renames each file by replacing all the spaces in the filenames with underscores:

```
Get-ChildItem * |
  Where-Object { !$_.PSIsContainer } |
  Rename-Item -NewName { $_.name -replace ' ','_' }
```

Let's go through how this command works. The Get-ChildItem cmdlet's output is piped to the Where-Object cmdlet, which filters the output so it returns only files. This is achieved by using the PSIsContainer NoteProperty with the -not (!) logical operator. (Alternatively, you could use *$_.PSIsContainer -eq $false*, like was done in a previous example.) The filtered output (i.e., the file objects) is piped to the Rename-Item cmdlet. The value of Rename-Item's -NewName parameter is a script block. This script block will be executed before the Rename-Item cmdlet is executed. In the script block, the $_ automatic variable represents each file object as it is passed to the command through the pipeline. The -replace comparison operator replaces the spaces in each filename (' ') with the underscore character ('_'). Note that you could also use the expression '\s' to target spaces because the first parameter accepts regular expressions. Even hidden files can be renamed, thanks to the -Force parameter.

The Rename-Item cmdlet has two built-in aliases. Those aliases are *ren* and *rni*.

The Magnificent 7

In this tutorial, you learned about all the ways that PowerShell can interact with files. In particular, you examined PowerShell's built-in cmdlets for working with files, which includes the Get-ChildItem, Get-Item, Copy-Item, Move-Item, New-Item, Remove-Item, and Rename-Item cmdlets. ■