Windows PowerShell in Action

Errors, Updates and Clarifications

Bruce Payette
July 2007

This document lists all known errors in the book "Windows PowerShell in Action" by Bruce Payette, published by Manning Publications. It also includes answers to reader's questions that were of general interest, reader suggestions on how to improve some of the examples and finally, some additional material that was omitted from the book. Thanks to everyone who contributed to this effort, especially the members of the Manning Author Online Forum at:

http://www.manning-sandbox.com/forum.jspa?forumID=248

There have been two printings of the first edition of Windows PowerShell in Action. The first printing was released in February 2007 and a second printing was done in March 2007. With the second printing, the publisher and I took the opportunity to correct all known errors in the book at that time. Additional errors have been found since the second printing and are incorporated in this document. Where something was corrected in the second printing, this will be indicated.

Regards,
Bruce Payette
Windows PowerShell Technical Lead
Microsoft Corporation
July 2007

Table of Contents

Errata

This document is divided into two parts. This first section covers the known errors in the book. The second section covers reader questions and additional material.

Acknowledgements

Yes – errata for the acknowledgements! One important person I neglected to mention in the Acknowledgements in the book was Kenneth Hanson. Kenneth, along with Jeff Jones was responsible for the namespace provider design in PowerShell. Kenneth was also responsible for a lot of the PowerShell type system design. The material in section 3.3 of the book was adapted from the PowerShell type-system functional specification written by Kenneth.

Inside-Cover Examples

There were a number of errors in front cover examples in the first printing:

The `for` loop example:

```
$a=1; while ($a -lt 10) { $a }
```

should read:

```
$a=1; while ($a -lt 10) { $a; ++$a } #correction
```

and the `foreach` loop example:

```
foreach ($file in dir -recurse -filter *.cs |sort length) {$_.filename
}
```

should read:

```
foreach ($file in dir -recurse -filter *.cs |sort length) { $file.name
}
```

These errors were corrected in the second printing (March. 2007)

Chapter 1

In the last example in section 1.4.5 on page 23, the text describes the example as counting from 0 to 9 but the output makes it appear to be counting from 1 to 10:

```
PS (1) > while ($i++ -lt 10) {if ($i % 2) {"$i is odd"} else {"$i is
even" } }
1 is odd
2 is even
3 is odd
4 is even
5 is odd
6 is even
7 is odd
8 is even
9 is odd
10 is even
```

This is because to the way the postfix ++ operator works. First the loop condition check is done, then the variable is incremented so that when it's used in the loop, it's one more than what it was in the loop condition check.

Chapter 2

On page 32, in listings 2.2 and Listing 2.3, in the line that looks like:

```
"$Parameter1:$_"
```

there should either be spaces around the colon or it should be written as:

```
"${Parameter1}:$_"
```

Without the colon or the braces, the sequence `$Parameter:$_` is being treated as though you were referencing the variable '$_' in the namespace 'Parameter'. As a result, nothing is output. There's also a missing close brace for the process clause in listing 2.2. The corrected listings look like:

Listing 2.2

```
function Write-InputObject
{
    param($Parameter1)
    process {
        if ($Parameter1)
        {
            "${Parameter1}:$_"
        } else {
            "$_"
        }
    }
}
```

Listing 2.3

```
param($Parameter1)
process {
    if ($Parameter1)
    {
        "${Parameter1}:$_"
    } else {
        "$_"
    }
}
```

Chapter 3

No errors have been found in chapter 3.

Chapter 4

On page 110, the first paragraph reads

"The result of this expression was true, which means that the match succeeded. It also means that `$matched` should be set, so let's look at what it contains:"

This sentence should have said `$matches` not `$matched`.

Chapter 5

On page 117, in Table 5.1, in the second line of the –isnot operator section:

`$true -isnot [object] $true The null value is the only thing that isn't an object.`

Should read

`$null -isnot [object] $true The null value is the only thing that isn't an object.`

The example should be showing the `$null` is not an object. The result and description columns are correct but the example itself is wrong. The value being tested should be `$null` not `$true`.

In the unary + example in Table 5.2 on page 118, the example should read

```
+ "123"
```

instead of

```
+ " "
```

On page 138, in Table 5.4, the fifth example showing the use of the percentage formatter, the output column repeats the expression instead of showing the output. Line should read:

```
{0:p}  Display a number as a percentage.  "{0:p}" -f .123      12.30 %
```

where the output column shows `12.30 %`.

Chapter 6

On page 171, in section 6.8.1, in the example showing the use of the `Foreach-Object` cmdlet:

```
1..10 | foreach ($_}
```

the opening parenthesis '(' after `foreach` should actually be an opening brace '{' as in

```
1..10 | foreach {$_}
```

Otherwise you'll get a syntax error.

Chapter 7

On page 190, in the middle of the page at the end of section 7.2.4, there is a line which reads, "And if I set a variable in a function, can I still use that *function* after the script has exited?". This line should read "...can I still use that *variable*...".

On page 191, at the top of the page, there is a line which reads, "Function two sets the variable $x to a particular *variable*...". This should read "Function two sets the variable $x to a particular *value*...".

On page 210, in the middle of the page there's a line which reads:
"The arguments to PowerShell.exe are actually accumulated and then treated as a script to *exit*." This statement should read "....treated as a script to *execute*."

In the summary section for chapter 7, the first bullet on page 213 talks about using the `param` keyword in functions but the use of `param` is never shown with functions, only with scripts. Here's how it works:

A typical function is written as

```
function plus ($x, $y) { $x + $y }
```

Using the `param` statement, this function can be rewritten as

```
function plus { param ($x, $y) $x + $y }
```

In fact, PowerShell internally transforms the first syntax into the second on as you can see when you look at how a function is stored:

```
PS (1) > function plus ($x, $y) { $x + $y }
PS (2) > $function:plus # show the stored function
param($x, $y) $x + $y
```

As we can see in the output, the original form has been rewritten to use the `param` statement. This is discussed in more detail in chapter 8, but should have been introduced in chapter 7.

> Author's Note: It's been interesting to watch how people write functions in PowerShell. I had expected the more conventional syntax
>
> ```
> function foo ($x, $y) { ... }
> ```
>
> to be used by most people but it seems that in practice, people are mostly defining functions using the `param` statement. This is somewhat surprising and, I presume, represents how internal consistency (functions, scripts and scriptblocks are all the same) is more important than familiarity. But I could be wrong about that…

Chapter 8

On page 221, the function definition at the top of page is shown as

```
function foo {"2+2"}
```

which is incorrect. There should be no quotes around 2+2. The correct function definition is:

```
function foo {2+2}
```

On page 240, the code snippet in the middle of the page:

```
foreach ($member in $properties) {
```

should be changed to

```
    foreach ($member in $members) {
```

Chapter 9

On page 259, towards the end of section 9.1.2, the text says:

"Let's just check the TargetObject member of the last error object to verify that it's the file name "nobother".

However the output of the example shows:

```
PS (6) > $errs[0].TargetObject
C:\Documents and Settings\brucepay\nofuss
PS (7) >
```

The problem in the example is that the expression `$errs[0]` should read `$errs[2]` since 2 is the index of the last error, not 0.

On page 269, the first line of the second paragraph on that page reads:

“There are other important applications of the `trap` statement in function definitions.”

This line incorrectly says “trap” when it should say “throw”. The corrected line should read:

“There are other important applications of the `throw` statement in function definitions.”

On page 282, at the bottom of the page, the output of the `showstack` script appears as:

```
At C:\Temp\showstack.ps1:5 char:17+ function e { gss <<<< }
```

when it should read:

```
At C:\Temp\showstack.ps1:5 char:17+ function e { gcs <<<< }
```

The name of the function to display the stack is “gcs” not "gss".

Chapter 10

On page 302, in section 10.1.1, the second paragraph from the bottom reads:

“The next most frequent word is "and", which is used 126 times.”

This should be changed to:

“The next most frequent word is "to", which is used 126 times. “

On page 306, in Table 10.1, there are a number of errors:

1. In the first line (Get-Location) "pwd" is mentioned as the way to get the current directory in cmd.exe. This is incorrect. To get current directory in cmd.exe you have to use "cd" with no arguments.
2. For (Set-Location) the command “chdir" is mentioned as being one of the UNIX sh equivalent command. This was true for the original Bourne shell and C shells. It appears to have been dropped by more modern shells leading some readers to report this as an error.
3. For (Rename-Item) the "rn" is mentioned as "cmd" equivalent command when it should be "ren" or "rename".
4. For (Rename-Item) the "ren" is mentioned as a UNIX command. This should be "mv" as this command is used both for moving and renaming files

On page 309, in the two code examples immediately before section 10.2.2, there are some problems. The code should read:

```
function notepad
{
    $args | %{ notepad.exe (resolve-path $_).ProviderPath }
}
```

The closing brace after ProviderPath was missing. And the code for run-exe should read

```
function run-exe
{
    $cmd, $files = $args
    $cmd = @(get-command -type application $cmd)[0].Definition
    $files | %{ & $cmd (resolve-path $_).ProviderPath}
}
```

In the last function in the script, the line should start with “$files” instead of “$file”. Both of these issues have been corrected in the second printing (March 2007) and were correct in the downloadable code archive.

On pages 315 and 317, a reader pointed out that the functions `Get-HexDump` and `Get-MagicNumber` can be simplified. Both these samples use formatting operator -

`f` to produce hex number and string output is used as input for `[string].PadLeft()` to produce the hex number of fixed width. `PadLeft()` call can be avoided by specifying precision for -f operator itself.

In the `Get-HexDump` function, the line:

```
$hex = $record | %{ " " + ("{0:x}" -f $_).PadLeft(2,"0")}
```

can be simplified to:

```
$hex = $record | %{ " {0:x2}" -f $_ }
```

And in the `Get-MagicNumber`, the lines:

```
$hex1 = ("{0:x}" -f ($mn[0]*256+$mn[1])).PadLeft(4, "0")
$hex2 = ("{0:x}" -f ($mn[2]*256+$mn[3])).PadLeft(4, "0")
```

can be simplified to:

```
$hex1 = "{0:x4}" -f ($mn[0]*256+$mn[1])
$hex2 = "{0:x4}" -f ($mn[2]*256+$mn[3])
```

The original samples are completely correct but these changes illustrate a way to make the code simpler and less verbose.

Chapter 11

On page 348, there is a spelling error in the second line of the second paragraph on the page. The line:

“Even when we specify the full name of the assembly, we didn’t have to tell the system *were* to find the file. “
Should read:

“Even when we specify the full name of the assembly, we didn’t have to tell the system *where* to find the file.”

The word “were” should be changed to “where”.

On page 363, the Get-RSS function can be simplified. The variable `$wc` is set but not used. Change:

```
function Get-RSS ($url)
```

```
{
    $wc = New-Object net.webclient
    $xml = [xml](New-Object net.webclient).DownloadString($url)
    $xml.rss.channel.item| select-object title,link
}
```

To use the variable as shown in:

```
function Get-RSS ($url)
{
    $wc = New-Object net.webclient
    $xml = [xml] $wc.DownloadString($url)
    $xml.rss.channel.item| select-object title,link
}
```

On page 367, a reader pointed out that there is security bug in the `Invoke-WebServer` script which leaves this script is open to code injection attacks.

The problem is that the regular expression used to validate the data received from the user is not *anchored*.(An anchored pattern must match the entire string instead of just part of the string.) Since it's unanchored, as long as part of the string matches the pattern, the rest of the string can contain any PowerShell code. To fix this, change the line with regular expression in it from:

```
if ($expression -match '[0-9.]+ *[-+*/%] *[0-9.]+')
```

to:

```
if ($expression.trim() -match '^[0-9.]+ *[-+*/%] *[0-9.]+$')
```

There are three changes: adding a trim() method call to strip leading an trailing blanks, a ^ was added to the to the front of the regular expression to anchor it to the beginning of the string, and a '$' was added to the end of the pattern to anchor it to the to the end of the input string.

Even though this is just a toy sample, part of the intent of the sample was to illustrate how to stop code injection attacks. The moral of this story is get "many eyes" to review this type of code and make sure that no possible attacks have been overlooked.

Chapter 12

On page 395 , the last example on the page uses "get-progids" with an 's' instead of "get-progid" (no 's'). Change the line

```
(get-progids word.application) | select-object -First 1
```

to:

```
(get-progid word.application) | select-object -First 1
```

> Author's Note: this function was originally called Get-ProgIDs but was changed to conform to the PowerShell convention of always using the singular form of nouns even when the command may return multiple results. This convention is used because the English language, used for all cmdlet names, is irregular in how the plural form of a noun is created. Sometimes pluralizing a noun just means adding an s to the word (e.g. prodid -> progids), sometimes the plural and singular forms are the same (e.g. sheep and sheep), sometimes 'en' is used instead of 's' (ox -> oxen) and so on. Here are some more examples:
>
> > Index -> indices
> > formula -> formulae/formulas
> > automaton -> automata
>
> Given how irregular this is, it is more discoverable/predictable to always use the singular form.

On page 422, the first example demonstrating `Get-WmiObject` uses the WMI class:

```
Win32Reg_AddRemovePrograms
```

as the class to retrieve. Unfortunately this class is only available on systems that have the SMS client component installed. (SMS - Systems Management Server - is a system management product available from Microsoft.)

This class, however, only provides an interface to information in the registry so you can obtain approximately the same information by doing

```
cd HKLM:\software\Microsoft\Windows\CurrentVersion\Uninstall
dir | Get-ItemProperty -name displayname, publisher 2> $null
```

On page 434, there is a typo in paragraph 4:

Change:

"As soon *was* we plug in (or turn on) a USB device"

to:
"As soon *as* we plug in (or turn on) a USB device "

On page 436, there is a typo in paragraph 1 where the word “property” was used when it should have said “properly”.

The line:

“First let’s make sure that the system was *property* updated.

should read:

“First let’s make sure that the system was *properly* updated. “

Chapter 13

On page 444, in section 13.2.2, the STRIDE explanation is missing the explanation for the ‘D’ component: "Denial of Service". The line:

“STRIDE is an acronym for Spoofing, Tampering, Repudiation, Information Disclosure, and Elevation of Privilege.”

Should read:

“STRIDE is an acronym for Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service and Elevation of Privilege.”

Note that this explanation of “D” as “Denial of Service” is properly included in table 13.1.

On page 450, there is a typo on the line before section 13.3.2. The line

“we talk *abut* how to enable scripting”

should read

“we talk *about* how to enable scripting

On page 452, there are errors in the examples that use Get-ItemProperty. There should be a space between “Get-ItemProperty” and “.” in the examples. The lines

```
PS (3) > Get-ItemProperty. ExecutionPolicy
```

and

```
PS (5) > Get-ItemProperty . ExecutionPolicy
```

should read:

```
PS (3) > Get-ItemProperty . ExecutionPolicy
```

and

```
PS (5) > Get-ItemProperty . ExecutionPolicy
```

respectively.

Appendix A

On page 487, the last line of section A.1.9 reads:

"It *let's* you define parameterized macros that can expand simple strings into more complex PowerShell expressions."

There shouldn't be an apostrophe in "lets".

On page 489 in section A.2.2, the 'where' example on this page will fail with a syntax error because the numeric literal is specified as "10M" instead of "10MB". The line:

```
get-process | where { $_.VS -gt 10M } | stop-process
```

should read

```
get-process | where { $_.VS -gt 10MB } | stop-process
```

> Author's Note: Late in the development process we (the PowerShell team) decided to change the multipliers for numeric literals from k/m/g to kb/mb/gb (case-insensitive). This was done after I'd written this section and when I went back to fix the examples I missed some. You may also encounter this problem when running scripts that were written for the pre-release version of PowerShell.

Appendix B

Section B.6 includes a number of examples that use the PowerShell event log for demonstrations. These examples incorrectly state that the name of the PowerShell event log is "PowerShell" when they should say "Windows PowerShell". Again, this is something that was changed late in the product release cycle.

Appendix C

Corrections for the published PowerShell grammar will be handled elsewhere.

Additional Topics and Discussions

Comment Syntax in PowerShell

This appears to have been missed in the book. Comments begin with a 'number sign' ('#") symbol and continue to the end of the line. The '#' character must be at the beginning of a *token* for it to start a comment. Here's an example that illustrates what this means:

```
PS (1) > echo hi#there
hi#there
```

In this example, the number sign is in the middle of the token "hi#there" and so is not treated as starting of a comment. In the next example, there is a space before the number sign.

```
PS (2) > echo hi #there
hi
```

Now the '#' it is treated as starting a comment and the following text is not displayed. Of course it can be preceded by characters other than a space and still start a comment. It can be preceded by any statement-terminating or expression-terminating character like a bracket, brace or semicolon as shown in the next couple of examples:

```
PS (3) > (echo hi)#there
hi
PS (4) > echo hi;#there
Hi
```

In both of these examples, the # symbol indicates the start of a comment. Finally, you need to take into account whether you are in expression mode or command mode. In command mode, as shown in the next example, the + symbol is included in the token "hi+#there".

```
PS (5) > echo hi+#there
hi+#there
```

But in expression mode, it's parsed as its own token. Now the '#" indicates the start of a comment and the overall expression results in an error.

```
PS (6) > "hi"+#there
You must provide a value expression on the right-hand side of the '+'
operator.
```

```
At line:1 char:6
+ "hi"+ <<<< #there
```

See section 2.2.3 for more information about expression mode and command mode parsing rules.

The '#' symbol is also allowed in function names:

```
PS (3) > function hi#there { "Hi there" }
PS (4) > hi#there
Hi there
```

The reason for allowing the # in the middle of tokens was to make it easy to accommodate providers that used '#" as part of their path names. Since people conventionally include a space before the beginning of a comment, this doesn't appear to cause any to difficulties.

Passing [switch] Parameters between Functions

A reader asked how one can optionally pass switch parameters when calling one function from another function. For example, consider a function `foo` which has a switch parameter 's'. From function `bar`, we want to call

```
foo
```

in some cases and

```
foo -s
```

in others.

Switch parameters were designed with exactly this scenario in mind. Here's how it works. While switch parameters don't require arguments, they can take one *if you specify the parameter with a trailing colon* as in:

```
dir -recurse: $true
```

Here's an example showing how the two functions mentioned previously would work. We'll define a `bar` function passes it's `$x` switch parameter to the `-s` switch parameter on `foo`. First let's define the `foo` function:

```
PS (77) > function foo ([switch] $s) { "s is $s" }
PS (78) > foo -s
s is True
```

```
PS (79) > foo
s is False
```

Now define `bar` which will call `foo`.

```
PS (80) > function bar ([switch] $x) { "x is $x"; foo -s: $x }
```

Let's call `bar` without passing `-x`.

```
PS (81) > bar
x is False
s is False
```

And we see that `$s` emitted from `foo` is false. Now call `bar` again, but specify `-x` this time.

```
PS (82) > bar -x
x is True
s is True
```

And we see that specifying `-x` has caused `-s` to be set to true as well.

Files, Variable Notation and the Current Drive Location

After reading the discussions on pages 88 and 144, a reader had a question about files and variable notation. He ran the following example:

```
PS (1) > $pwd.path >test.txt
PS (2) > ${c:test.txt}
C:\Temp
PS (3) > d:
PS (4) > $pwd.path >test.txt
PS (5) > ${d:test.txt}
D:\
PS (6) > ${c:test.txt}
C:\Temp
PS (7) > pwd

Path
----
D:\
```

and asked the question:

> Where can I look at what either c: or d: contains? If it is the 'current working directory' on the c or e drive, why could I access the ${c:test.txt} even though the current working directory was e:\ ?

What's happening here is that each `PSDrive` has its own notion of current working directory. In the variable notation, if the path after the drive specifier is a relative path, then it will be resolved relative to the current working directory for the specified drive. This is why `${c:test.txt}` still worked after doing a `cd` to `e:`. If the drive is not specified, then the current directory for the current drive is used. This works for commands as well.

```
PS (8) > gc c:test.txt
C:\Temp
```

will look up `test.txt` relative to the current directory for the C: drive.

```
gc test.txt
```

will look up test.txt relative to the current directory on the current drive. If the current drive is D: then it will be equivalent to

```
gc d:test.txt
```

To see what the current location for each drive is, you can use the `Get-PSDrive` cmdlet:

```
PS (9) > get-psdrive -name C,D | ft -auto

Name Provider   Root CurrentLocation
---- --------   ---- ---------------
C    FileSystem C:\             Temp
D    FileSystem D:\
```

Processing Numbers Stored in a Text File

A user had a question about performing mathematical operations on a set of numbers stored in a text file. The core of the task was to read strings from a text file and have those strings interpreted as numeric objects instead of strings. While this is easy to do in PowerShell you need to specify what *kind* of numbers you want them to be and this is what was causing the problem. If you specify `[int]`, then the numbers will be rounded to integers. If you want them to be double-precision floating point numbers, then use `[double]`. Let's look at a simple example. We'll assign a string containing 4 numbers to a variable.

```
PS (12) > $s = "1 2 3.3 4"
```

This string contains a mix of integer and floating-point numbers. Let's split the string and then convert each element of the resulting array into an integer:

```
PS (13) > [int[]] $s.Split()
1
2
3
4
```

We can see that because we specified `[int]`, 3.3 has been rounded down to 3. Now let's do this again, but using a `[double[]]` cast this time:

```
PS (14) > [double[]] $s.Split()
1
2
3.3
4
PS (15) >
```

Now 3.3 remains as 3.3 however all of the other numbers are also double-precision floats.

> Author's Note: in designing PowerShell, at one point we had considered only supporting a single numeric type: double. A number of scripting languages do this because having a single numeric type simplifies the language and, on modern processor architectures, integer arithmetic is not much faster than floating point (if at all). However, as we expanded our interoperation with .NET it became clear that this wouldn't be sufficient for all of the scenarios we wanted to cover. We had to eventually support all of the numeric types supported by .NET because they were required to work with the various classes making up the framework. Well – not quite all numeric types: PowerShell V1 doesn't support the unsigned integral types. This decision was reasonable at the time because the Common Language Specification (CLS) didn't permit conforming classes to use unsigned types. Support for unsigned numeric types will be added in version 2 of PowerShell based in user requests.

And now, one last variation on this exercise: instead of an array where everything is the same type, we want to split the string directly into a structure with each element converted to a distinct type. We can do this with a hashtable and multiple assignment as shown:

```
PS (15) > $f = @{}
PS (16) > [int] $f.a, [int] $f.b, [double] $f.c, [int] $f.d =
$s.split()
PS (17) > $f

Name Value
```

```
---- -----
a 1
b 2
d 4
c 3.3
```

In this example, each string is converted to the desired type and then an element is added to the hashtable to hold the converted value.

How ForEach-Object Processes its Arguments

A reader observed that in the example on page 301:

```
$words | % {$h=@{}} {$h[$_] += 1}
```

that, although the code worked properly, it seemed inconsistent when reading the help text for this cmdlet. The help text says that the -Process parameter is the only positional parameter, and that it's in position 1. Therefore, according to the help file, since the begin clause isn't positional, the example shouldn't work. This led the reader to assume that either there was an error in the help file, or that they were misunderstanding the idea of positional parameters.

In fact the help file is correct (the signatures are extracted from the code) however the way it works is the tricky bit.

If you look at the signature of the -Process parameter, you'll see that, yes, it is positional but it also takes a collection of scriptblocks and receives all remaining unbound arguments. So, in the case of

```
dir | foreach {$sum=0} {$sum++} {$sum}
```

The `-Process` parameter is getting an array of three scriptblocks whereas `-Begin` and `-End` are empty. Now here's the trick. If `-Begin` is empty and `-Process` has more than two scriptblocks in the collection, then the first one is treated as the Begin scriptblock and the second one is treated as the Process scriptblock. If Begin is specified, but End is not and there are two scriptblocks, then the first one is treated as the Process clause and the second one is the End clause. Finally, if both Begin and End are specified then the remaining arguments will be treated as multiple Process clauses. This allows

```
dir | foreach {$sum=0} {$sum++} {$sum}
dir | foreach -begin {$sum=0} {$sum++} {$sum}
dir | foreach {$sum=0} {$sum++} -end {$sum}
dir | foreach -begin {$sum=0} {$sum++} -end {$sum}
```

and

```
dir | foreach -begin {$sum=0} -process {$sum++} -end {$sum}
```

to all work as expected.

> Author's Note: While documenting the `ForEach-Object` cmdlet we considered trying to explain all of this but on reflection, we decided that we were probably better off *not* trying to document the details since it required a complicated solution to achieve a simple experience. We're reconsidering this and will probably update the documentation to include the detailed explanation.

Where-Object and Get-Content's –ReadCount Parameter

A reader had a question about how Get-Content's `-ReadCount` parameter actually works. He was trying to use Where-Object to filter the output of Get-Content where the read count was greater than one and it appeared that some objects were being skipped.

Here's what's going on. Unfortunately the –ReadCount parameter has a very confusing name. From the PowerShell user's perspective, it has nothing to do with reading. What it effectively does is control the number for records *written* to the next pipeline element. The following examples illustrate this. The file `test.txt` is a simple text file that contains 10 lines. We'll use the `Foreach-Object` cmdlet (through its alias '%') to count the length of each object being passed down the pipeline. We'll use the @(…) construct to guarantee that we're always treating $_ as an array. Here are the examples with –readcount varying from 1 to 4.

```
PS (119) > gc test.txt -ReadCount 1 | % { @($_).count } | select -fir 1
1
PS (120) > gc test.txt -ReadCount 2 | % { @($_).count } | select -fir 1
2
PS (121) > gc test.txt -ReadCount 3 | % { @($_).count } | select -fir 1
3
PS (122) > gc test.txt -ReadCount 4 | % { @($_).count } | select -fir 1
4
```

In each case where `-ReadCount` is greater than 1, the variable `$_` is set to a collection of objects where the object count of that collection is equivalent to the value specified by `-ReadCount`. Taking another example, we'll use Foreach-Object (through its alias '?') to filter the pipeline:

```
PS (127) > gc test.txt -read 5 | ? {$_ -like '*'} | % { $_.count }
5
5
```

We can see that the filter result contained two collections of 5 objects each written to the pipeline for a total of 10 objects. Now let's use `ForEach-Object` and the `if` statement to filter the list:

```
PS (128) > (gc test.txt -read 10 | % {if ($_ -match '.') {$_}} |
>>> Measure-Object).count
>>>
10
```

This time we see a count of ten because the value of `$_` in the `ForEach-Object` cmdlet is unraveled when written to the output pipe. And now let's look at one final example:

```
PS (130) > (gc test.txt -read 4 | %{$_} | ? {$_ -like '*a*'} |
>>> Measure-Object).count
>>>
10
```

Here we've inserted one more `ForEach-Object` command between the gc and the `Where-Object` which simply unravels the collections in $_ and so we again see a count of 10.

> Author's Note: Here's the annoying thing: from the `Get-Content` developer's perspective, it actually *is* doing a read of `-ReadCount` objects from the provider. `Get-Content` reads `-ReadCount` objects then writes them as a single object to the pipeline instead of unraveling them. (I suspect that this is actually a bug that's turned into a feature…) Anyway, the name makes perfect sense to the developer and absolutely no sense to the user. This is why developers always have to be aware of the user's perspective even if it doesn't precisely match the implementation details.

In summary, whenever `-ReadCount` is set to a value greater than 1 for performance reasons and these collections are sent through the pipeline to `Where-Object`, the user has to take extra steps to deal with unraveling the batched collections of objects.

Value Types, Reference Types, Variables and [ref]

In this section we'll cover some of the deep details of how arguments are passed to scripts and function in PowerShell.

PowerShell inherits the semantics of .NET types. This means that reference types like arrays and collections are passed by reference and value types like numbers are passed by value. Passing an object by reference means that a pointer to the object is passed

instead of a copy of the value of the object. Passing by value means that a copy of the object is passed. So what are the implications of this for PowerShell?

The default collection type in PowerShell is `[object[]]` which has immutable (or fixed) length. This means that in order to add an element to the array, you have to copy the old values into new array that is large enough to hold the result. Section 3.2.4 in the book goes into these issues in some detail. As long as you stick to arrays, you should rarely be surprised by how things work. However, then you get into working with collections that have mutable length like `[System.Collections.ArrayList]`, you may find some surprises. If you use the `Add()` method on this type to add an element to the collection, the collection itself is changed instead of a new one being created. This means that all of the things that reference the collection will see the change. There aren't many places in PowerShell were this is an important distinction but the `$error` variable is one such case. If you try to "save" a copy of `$error` in another variable, all you're doing is creating a new reference to the collection it contains so you're "saved" collection will actually continue to change. Simple assignment isn't enough to create a copy of the contents of objects of this type. The easiest way to guarantee a copy of an enumerable collection is to use `Write-Object` as shown in the following examples:

```
PS (142) > $a = 1,2,3 # set up the original object
PS (143) > $b = $a # $b gets a reference to the collection in $a
PS (144) > $c = $a | Write-Output # $c gets a copy
PS (145) > $a[1] = "Hello" # change the second element in $a
PS (146) > "$a" # now look at the contents of $a
1 Hello 3
PS (147) > "$b" # $b has also been change since it has a reference
1 Hello 3
PS (148) > "$c" # but $c has not because it contains a copy of $a
1 2 3
```

Now that we're talked about reference objects, let's talk about *reference variables*. A reference (or `[ref]`) variable is not the same as a reference type. A reference variable is one that has a pointer to another variable instead of a value.
This is another complexity that was introduced into PowerShell to allow for interoperation with .NET and (especially) COM.

When calling a script, function or method, passing a variable as

```
[ref] $a
```

passes a reference to the variable itself, not the value of the variable (which may in turn, be a value or reference type.) This is shown in the following example. First we define a function that takes a variable reference as an argument:

```
function foo ([ref] $v) { [string] $v.value ; $v.value = "Scalar" }
```

This function emits the original value of the variable, and then changes it to the string "Scalar".

Now set the variable `$orig` and then save a reference to the value assigned to `$orig` in `$copy` .

```
PS (157) > $orig = 1,2,3
PS (158) > $copy = $orig # $copy contains a reference to $orig's value
```

And call the function `foo` passing in a reference to `$copy`.

```
PS (159) > foo ([ref] $copy)
1 2 3
```

This prints out the value of argument variable as we defined it previously, and then changes the variable, but not the collection referenced by the variable:

PS (160) > "$copy"
Scalar

So we see that the variable has, indeed, changed:

```
PS (161) > "$orig"
1 2 3
```

however the original variable has not.

We hadn't originally planned on having [ref] variables in PowerShell but they turned out to be necessary in order to use some COM and .NET APIs. They are however, difficult to understand, even for full-time developers. In fact, the .NET guidelines discourage creating APIs that use by-reference arguments for this reason.

Variable Objects and Constrained Variables

Now let's look at some additional topics on variables. Like everything else in PowerShell, variables themselves are objects. You can retrieve the variable object (as opposed to the variable value) using the Get-Variable cmdlet. First we'll define a variable `$var`

```
PS (1) > $var = 12
```

Now let's retrieve it and see what the variable object itself contains:

```
PS (2) > get-variable var | fl *


Name        : var
Description :
Value       : 12
Options     : None
Attributes  : {}
```

We can see that it contains the variable value, but it also contains the variables name as well as some other characteristics. Of course the curious user will wonder if one can rename a variable by assigning to the name property:

```
PS (4) > (get-variable var).value = 10
PS (5) > $var
10
```

But you can't – it's a read-only property maintained by the PowerShell runtime. You can however, change the value of the variable:

```
PS (8) > (get-variable var).value = 10
PS (9) > $var
10
```

Another thing you can do with variables is add *constraints* to them. Constraints limit the set of possible valid values that can be assigned to a variable. We already know how add type constraints to a variable, but you can also add addition constraints as well. For example, you can add a constraint that limits the range of values that can be assigned to a variable. Unfortunately, in version 1 of PowerShell, this is a bit tricky to do – there isn't any special syntax to do this like there is for type constraints. Here's an example of how to do it. First we need to create a constraint object:

```
PS (6) > $ra = new-object `
>> System.Management.Automation.ValidateRangeAttribute 5,10
>>
```

This gives us a constraint object that can be used to limit the values of a variable to the range 5..10. (These validation attributes are documented in the PowerShell SDK available at http://msdn.microsoft.com) Next, we have to add this attribute to the list of attributes attached to the variable:

```
PS (7) > (get-variable var).attributes.add($ra)
```

Now let's verify that it's been added:

```
PS (8) > get-variable var | fl *
```

```
Name        : var
Description :
Value       : 10
Options     : None
Attributes  : {System.Management.Automation.ValidateRangeAttribute}
```

We can see that the list of attributes now contains the one we added. Let's see what happens when we try an assign a variable that isn't in the valid range.

```
PS (9) > $var = 3
Cannot validate because of invalid value (3) for variable var.
At line:1 char:5
+ $var <<<<  = 3
```

As expected, we get an error message indicating that the value is out of range. But a valid value

```
PS (10) > $var = 7
PS (11) > $var
7
```

still works properly. Finally, let's look at the options field:

```
PS (12) > (get-variable var).options
```
None

At this point, there are no variable options associated with the variable. Let's set the "readonly" option:

```
PS (13) >  (get-variable var).options = "readonly"
```

And verify that it's been set:

```
PS (14) >  (get-variable var).options
ReadOnly
```

It has so let's try and change the value of the variable:

```
PS (15) > $var=13
Cannot overwrite variable var because it is read-only or constant.
At line:1 char:6
+  $var <<<< =13
PS (16) >
```

and this fails as it should. For more information on the options that can be set on a variable, refer to the help content for Set-Variable

```
Get-Help -full Set-Variable
```

Summary

This concludes the errata and updates for Windows PowerShell in Action as of July 2007. As additional issues come up – errors, questions, or suggestions this document will be updated to contain the additional material.