Windows PowerShell 4.0 For the IT Professional - Part 1

Module 8: Operators 1

Student Lab Manual

Version 2.0

Conditions and Terms of Use

Microsoft Confidential - For Internal Use Only

Contents

Lab 8: Operators 1

Introduction

In this lab, we will introduce Windows PowerShell's most common operators. An operator is a language element that you can use in a command or expression. Windows PowerShell supports several types of operators to help you manipulate values.

Objectives

After completing this lab, you will be able to:

- Identify the most commonly used operators in Windows PowerShell
- Use comparison operators to evaluate object property values
- Use logical operators to perform evaluations on groups of comparison operators

Prerequisites

Start all VMs provided for the workshop labs.
Logon to WIN8-WS as:

Username: **Contoso\Administrator**

Password: **PowerShell4**

Estimated time to complete this lab

60 minutes

NOTE: These exercises use many Windows PowerShell commands. You can type these commands into the Windows PowerShell Integrated Scripting Environment (ISE) or the Windows PowerShell console. For some exercises, you can load pre-typed lab files into the Windows PowerShell ISE, allowing you to select and execute individual commands. Each lab has its own folder under **C:\PShell\Labs** on **WIN8-WS**.

Some exercises in this workshop may require running the Windows PowerShell console or ISE as an elevated user (Run as Administrator).

We recommend that you connect to the virtual machines (VMs) for these labs through Remote Desktop rather than connecting through the Hyper-V console. This allows you to use copy and paste between VMs and the host machine. If you are using the online hosted labs, then you are already using Remote Desktop.

Exercise 8.1: Comparison Operators

Introduction

Operators allow us to evaluate a condition by comparing two values and then take an appropriate action. Windows PowerShell does not use the traditional operator symbols used in many other languages. The differences are in the table below.

PowerShell – case insensitive	PowerShell – case sensitive	Traditional	Description
-eq / -ieq	-ceq	=	Equal to
-ne / -ine	-cne	<>, !=	Not equal to
-lt / -ilt	-clt	<	Less than
-le / -ile	-cle	<=	Less than or equal to
-gt / -igt	-cgt	>	Greater than
-ge / -ige	-cge	>=	Greater than or equal to

All operators are prefixed with a hyphen (-). To use a comparison operator, specify the values you want to compare, together with an operator that separates these values. The values on either side of the operator are referred to as "operands".

```
<left-hand operand> <operator> <right-hand operand>
```

Objectives

After completing this exercise, you will be able to:

- Understand Windows PowerShell comparison operator syntax
- Perform simple conditional evaluations

Task 8.1.1: Comparison Operators

1. Comparison operators return a true or false output – a Boolean data type - when used to compare two ***scalar*** values. A scalar value holds one unit of data, or in the case of Windows PowerShell, a single object.
   Type the statements in the "Condition" column into the Windows PowerShell ISE command pane.
   Complete the "Result" column with the result of the condition.

#	Condition	Result
1	9 –eq 9	
2	4 –lt 7	
3	12 –ge 12	
4	“Hello” –eq “hello”	
5	5 –le 6	
6	64 –le 64.9	
7	2,3,4,5,6 –lt 4	
8	“e” –ge “b”	
9	“Server” –lt “Workstation”	
10	1,2,9,3,4,3,6,7,1 –eq 3	

True
True
True
True
True
True
2
3
True
rue
3
3

Why do you think conditions 7 & 10 do not return a Boolean (true or false) result?

We are not comparing two scalar values. The left-hand operand is an array.

2. There are two variants of the comparison operators. Each operator can be prefixed with a “c” to force case-sensitivity, or an “i” to indicate the comparison is case-insensitive (the default). Type the commands below to view their result.

```
“Microsoft” -ceq “microsoft”
12 -ceq 12
“bing” -cne “Bing”
“xbox” -ieq “XBOX”
```

3. The commands below collect GPO data about the current user and computer and saves it in an Xml file. We then read the Xml file into a variable called $report.

   Type the commands into the console or ISE.

```
Get-GPResultantSetOfPolicy -ReportType Xml -Path
C:\PShell\Labs\Lab_8\ComputerReport.xml
[xml]$report = Get-Content C:\PShell\Labs\Lab_8\ComputerReport.xml
```

4. Now, query the report. The example below returns all enabled GPO’s that apply to the local computer.

```
$report.Rsop.ComputerResults.GPO | Where-Object enabled -eq $true
```

5. Finally, return all GPOs where the initial configuration has been changed. The VersionDirectory property stores the number of changes for a GPO.

```
$report.rsop.ComputerResults.GPO | Where-Object VersionDirectory -gt 0
```

Task 8.1.2: Operators and Types

Comparison operators can only compare two objects of the same type. PowerShell will try to convert the data type of the right-hand operand to the same data type as the left-hand operand.

1. Perform the comparison operations in the table below. Tick the appropriate "success" or "fail" column for each result. (Use the <Up Arrow> in the Windows PowerShell console to edit the previous line.)

Expression	Success	Fail
(Get-Item C:\Windows\WindowsUpdate.log).length –gt 300		
(Get-Item C:\Windows\WindowsUpdate.log).CreationTime –gt "Today"		
(Get-Item C:\Windows\WindowsUpdate.log).length –le "50000"		
(Get-Item C:\Windows\WindowsUpdate.log).Exists –eq 0		
(Get-date) -gt "Hello"		
7 –gt "Hello"		

Success!

FAIL! Error: Cannot convert value "Today" to type "System.DateTime"

Success!

Succeeds but returns FALSE

Success! (false returned, but successful execution)

FAIL! Error: Cannot convert value "Hello" to type "System.DateTime"

FAIL! Error: Cannot convert value "Hello" to "System.Int32"

Task 8.1.3: Containment Operators

In task 8.1.1, two expressions failed to return a Boolean value (#7 & #10). The Windows PowerShell comparison operators do not work as expected with non-scalar values. (More than one object).

1. The containment operators, **-contains** and **–in**, test whether a value exists within a collection of objects and return a Boolean (true or false) result.

   The following commands each produce a collection of objects. We can use the -contains operator to determine whether a single value exists within that collection.

   Type the commands into the Windows PowerShell ISE command pane.

```
1,2,3,4,5,6,7,8,9,10 -contains 3
"Windows","XBox","Surface","Bing","Office" -ccontains "xbox"
(Get-SmbShare | Select-Object -ExpandProperty name) -contains "ADMIN$"
(Get-Process).Name -icontains "winlogon"
(Get-ADComputer -Filter *).DNSHostName -contains "2012R2-MS.contoso.com"
```

True

False

True

True

True

NOTE: Windows PowerShell will automatically iterate (loop through) a single property for a collection of objects. e.g.

(Get-ChildItem C:\Windows).FullName

2. The **-in** operator works in exactly the same way as the **–contains** operator, except that it is used in the opposite order, which you may find easier to read.

```
"Sales" -In "Marketing","IT","Sales","Finance","HR"
```

3. Modify the commands below to use the **–contains** or **–in** operator to answer the following questions as per the example below.

```
(Get-Process | Select-Object -ExpandProperty processName) -contains "lsass"
```

> **NOTE:** "Select-Object -ExpandProperty" specifies a property to select and indicates that an attempt should be made to expand that property. Wildcards are permitted in the property name.
>
> For example, if the specified property is an array, each value of the array is included in the output. If the property contains an object, the properties of that object are displayed in the output.

a. Does the output from the command below contain "BUILTIN\Users"
True / **False**

```
( (Get-Acl C:\Windows).Access | Select-Object -ExpandProperty IdentityReference)
```

True

```
( (Get-Acl C:\Windows).Access |
Select-Object -ExpandProperty
IdentityReference) -Contains "BUILTIN\Users"

"BUILTIN\Users" -IN ( (Get-Acl
C:\Windows).Access |
Select-Object -ExpandProperty
IdentityReference)
```

b. Does the output from the command below contain "KB2898108"
True / **False**

```
(Get-HotFix | Select-Object -ExpandProperty HotfixID)
```

False

```
(Get-HotFix | Select-Object -ExpandProperty
HotfixID) -contains "KB2898108"
```

c. Does the output from the command below contain "SMTP"
True / **False**

```
(Get-WindowsOptionalFeature -Online | Select-Object -ExpandProperty FeatureName)
```

False

Exercise 8.2: Logical Operators

Introduction

Logical operators allow multiple comparison operator evaluations in a single expression. Logical operators use the following syntax.

<statement> {-AND | -OR | -XOR} <statement>

The four logical operators and their results are in the table below.

Operator	Description	Left operand	Right Operand	Result
-and	Both operands must be True	True/False False True	False/True False True	False False True
-or	One operand must be True	True/False False True	False/True False True	True False True
-xor	Only one of the operands must be True	True/False False True	False/True False True	True False False
-not	Reverses the result	-	True False	False True

Objectives

After completing this exercise, you will be able to:

- Understand how to use Windows PowerShell's four logical operators to combine comparison operators.

Task 8.2.1: Logical AND

1. Use the Get-ADGroup Cmdlet to display all Active Directory group objects. The –Filter parameter accepts a wildcard (*) character to return all Groups.

```
Get-ADGroup –Filter *
```

2. Using the –Filter parameter, return Universal Security groups.

```
Get-ADGroup -Filter {groupcategory -eq "Security" -and groupscope -eq "Universal"}
```

3. Modify the above command to return Universal Distribution groups.

```
Get-ADGroup -Filter {groupcategory -ne "Security" -and groupscope -eq
"Universal"}
```

4. Next, return all Universal groups whose SamAccountName attribute starts with the letter 'E'.

```
Get-ADGroup -Filter {samaccountname -like "E*" -and groupscope -eq
"Universal"}
```

Task 8.2.2: Logical OR

1. Use the Get-ADGroup Cmdlet to display all Active Directory group objects whose samaccountname property starts with the letter "D*" OR the groupscope is "Universal". Write the command below.

```
Get-ADGroup -Filter {samaccountname -like
"D*" -or groupscope -eq "Universal"}
```

Task 8.2.3: Logical XOR

The logical Exclusive OR operator expects one operand to be true and the other false, but not both, to return a true output.

1. Execute the following –xor operations on two arrays.

```
1,2,3,4,5 -contains 2 -xor 5,6,7,8,9 -contains 8
1,2,3,4,5 -contains 2 -xor 5,6,7,8,9 -contains 12
```

```
False
True
```

Task 8.2.4: Logical NOT

The NOT operator negates the following statement. The –not operator is a unary logical operator as it only operates on a single statement. The operator uses the syntax below.

{! | -NOT} <statement>

1. For example, we could use the –not operator to invert a boolean result

```
(Get-Date).IsDaylightSavingTime()
True

-not (Get-Date).IsDaylightSavingTime()
False
```

Exercise 8.3: String Comparison Operators

Introduction

Windows PowerShell provides text-processing capabilities via simple wildcard character matching and by leveraging the .NET regular expression language library (commonly referred to as "regex").

Objectives

After completing this exercise, you will be able to:

- Use the following operators to perform wildcard and regular expression string matching operations: -like, -notlike, -match, -notmatch.

Task 8.3.1: Wildcard operators

1. Windows PowerShell's **–like** operator can be combined with the "*" and "?" characters to provide a powerful string matching feature. Many Windows PowerShell Cmdlet parameter values accept wildcards.

Character	Description
*	Matches zero or more characters
?	Matches only one character in the specified position
[ ], [-]	Matches a range of characters or only specified characters

Execute the commands below.

```
Get-Childitem –Path C:\PShell\Lab*
Get-Childitem -Recurse –Path C:\PShell\Labs\*.ps1
Get-Childitem –Path C:\Windows\system32\A?c*.dll
Get-WindowsOptionalFeature -FeatureName *Hyper* -Online
Get-NetIPAddress -IPAddress ?0.0.1.220
```

The Alphabetical range character ([ ]) provides flexible multiple character matching

```
Get-ChildItem C:\Windows\[ab]*
Get-ChildItem C:\Windows\[a-h]b*
Get-ChildItem C:\Windows\system32\[u-w]a*.dll
```

2. Create wildcard matches to satisfy the following search criteria and write them below.
   a. Return all .exe files in C:\Windows\System32 whose names start with the letter "d", "g" or "n".

Get-ChildItem -Path C:\Windows\System32\[dgn]*.exe

b. Get all folders whose name ends with the character "n" in C:\Windows\System32

Get-ChildItem -Directory -Path C:\Windows\system32*n

c. Find all .txt files in C:\Windows\System32\WindowsPowerShell\v1.0, recursively. Use the Get-ChildItem Cmdlet's –filter parameter.

Get-ChildItem -Path C:\Windows\System32\WindowsPowerShell\v1.0 -filter *.txt -Recurse

Task 8.3.2: Regular Expression Operators

The regular expression language consists of several character groups, listed below. A full discussion of their use is beyond the scope of this workshop. These are covered in much more detail online here: http://go.microsoft.com/fwlink/?LinkId=133231.

Regular Expression group name	Description
Character Escapes	A backslash (\ ) preceding any character indicates that character either has a special meaning or should be interpreted literally
Character Classes	Matches one or more characters in a set
Anchors	Matches depending on the current position in the string to be matched
Grouping Constructs	Allows expression matches to capture sub-strings of an input string
Quantifiers	Controls the frequency that a previous element must occur for a match to succeed
Backreference Constructs	Causes a previously used expression to be identified later in the same regular expression.
Alternation Constructs	Allows a regular expression to implement either/or logic
Substitutions	Used to control replacement patterns

1. Windows PowerShell's **–match** operator can be used, in a binary form, to test whether a particular text string matches a regular expression pattern. To match a simple string, ensure it is the left-hand operand in a **–match** operator statement, and the regex is on the right-hand side.

Type and execute the two regular expression examples below, in the Windows PowerShell ISE console pane.

As above, maybe list the examples together?

```
'This is a simple string' -match 'This'
'This is a simple string' -match '^T\w+'
```

The first example shows the most simple regex match, an exact sub-string.

```
'This is a simple string' -match 'This'
```

2. The next example uses special control characters to perform the same match.

```
'This is a simple string' -match '^T\w+'
```

   a. The caret (^) specifies the string starts with an uppercase 'T'.
   b. The '\w' is a character class that represents a word character.
   c. The '+' is a Quantifier that matches the previous element one or more times.

3. What output did you receive from both examples?

True

4. Fortunately, Windows PowerShell automatically stores the –match operator's result in the $Matches automatic variable. Type the $Matches variable name and press Enter to view the result of the last match.
5. The $Matches variable holds a hash table object (covered in module 15).

```
$Matches
```

6. Storing the results in a hash table becomes very useful if multiple sub-expressions are used. The example below specifies two separate regex patterns, in parentheses.

```
'This is a simple string' -match '(^T\w+)(.+)'

$Matches

Name                           Value
----                           -----
2                              is a simple string
1                              This
0                              This is a simple string
```

Individual keys in the hash table can be accessed using array syntax.

```
$Matches[0]
This is a simple string
```

An alternative dotted syntax can also be used.

```
$Matches.2
is a simple string
```

7. The Select-String Cmdlet accepts a regular expression through its –Pattern parameter. To use this Cmdlet with a regex, first define the string containing the pattern we want to match. In this case we use a legacy command output string (ipconfig.exe).

```
$data = ipconfig /displaydns
```

8. Next, define the regex pattern and store it in the variable $pattern.

```
$pattern = "\s:\s\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}"
```

   This is a much more complex pattern at first glance, but if you break it down into smaller parts, it is actually surprisingly simple.

Regex pattern	Description	Matched result
\s	Matches a single whitespace character	‘ ’
:\s	Matches the literal colon ‘:’ character followed by a whitespace character	‘: ’
\d{1,3}\.	Decimal number between 1 and 3 characters long followed by a literal ‘.’ Character (for a total of 4 matches). Final 4th match has no following ‘.’	123.123.123.123

9. Pipe the $data variable, containing the output of the ipconfig.exe command, to the Select-String Cmdlet. Use the $pattern variable as the value for the Cmdlet’s –Pattern parameter.

```
$data | Select-String –Pattern $pattern
```

You should receive output similar to that shown below.

```
      A (Host) Record . . . : 10.0.1.200
```

Exercise 8.4: Other Numeric Operators

Introduction

Windows PowerShell implements a number of other numeric operators.

Objectives

After completing this exercise, you will be able to:

- Easily create ranges of integers
- Redirect various output streams to files
- Calculate and convert between file sizes

Task 8.4.1: Numeric Range Operator

1. The numeric range operator generates a sequence of integers. Type the commands below.

```
1..10
```

2. Negative integers are also supported.

```
-10..0
```

Task 8.4.2: Numerical Multiplier

1. Windows PowerShell offers a number of built-in integer suffixes to easily compute accurate file sizes of 1024 bytes and an exponent between 1 and 5.

Suffix	Description	Example
kb	kilobyte 1024^1	1kb 1024
mb	megabyte 1024^2	1mb 1048576
gb	gigabyte 1024^3	1gb 1073741824
tb	terabyte 1024^4	1tb 1099511627776
pb	petabyte 1024^5	1pb 1125899906842624

2. The multiplier suffixes are *string literals* parsed by Windows PowerShell and can be used with the arithmetic operators. Write the answers to the following questions below.
   a. 10mb + 60mb _____
   b. 120gb – 40.5gb _____
   c. 1kb * 6.8mb _____
   d. 4503454 / 1mb _____
3. The division operator can convert between all multiplier suffixes. Convert the output contained in the $total variable into the specified unit and record the result below.

Show that they are converted by the parser e.g.
`(1MB).GetType()`

4. Convert to megabytes _____

```
$total = (Get-ChildItem C:\windows\System32 -file | Measure-Object -Property
length -sum).Sum
```

5. Convert to gigabytes _____

```
$total = (Get-ChildItem -Recurse C:\Windows\System32 -ErrorAction SilentlyContinue
| Measure-Object -Property length -sum).Sum
```

6. Convert to kilobytes _____

```
$total = (Get-CimInstance -ClassName Win32_PhysicalMemory |
Measure-Object -Property capacity -sum).Sum
```

$total / 1MB

```
$total = (Get-ChildItem C:\windows\System32 -
file | Measure-Object -Property length -
sum).Sum
```

$total / 1GB

$total / 1KB