Windows PowerShell 4.0 For the IT Professional - Part 1

Module 12: Operators 2

Student Lab Manual

Version 2.0

Conditions and Terms of Use

Microsoft Confidential - For Internal Use Only

Copyright and Trademarks

Contents

Lab 12: Operators 2

Introduction

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:

- Use Arithmetic operators
- Use Non-numeric operators
- Use Compound assignment
- Use Split and Join operators
- Use Format operators
- Use the Replace operator

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 12.1: Arithmetic Operators

Introduction

Arithmetic operators calculate numeric values. You can use one or more arithmetic operators to add, subtract, multiply, and divide values, and to calculate the remainder (modulus) of a division operation.

Objectives

After completing this exercise, you will be able to:

- Use numeric and non-numeric operators
- Use compound operators

Task 12.1.1: Numeric operations

1. The help topic about_Operator_Precedence explains that Windows PowerShell processes operators in the following order:

Operator	Description
()	Operator precedence
-	Negation
/,*,%	Division, Multiplication, Modulus
+,-	Addition, Subtraction

Operator precedence can be controlled using parentheses. Type the examples below to observe this behavior.

```
10 + 5 / 2
(10 + 5) / 2
5 * 2 + 1
5 * (2 + 1)
10 / 5 * 2
(10 / 5) * 2
```

2. Type and execute the following commands. In the "Type Name" column, record the "Name" property returned by the GetType() method.

#	Command	Type Name
1	`(10 + 45).GetType()`	Int32
2	`(10 - 6.2).GetType()`	Double
3	`(10 / 2).GetType()`	Int32

4	`(15 % 3).GetType()`	
5	`(20 * 11.5).GetType()`	
6	`(1 - 0.56).GetType()`	
7	`(8GB + 11TB).GetType()`	
8	`(0x45 + 0x12).GetType()`	
9	`(1,2,3,4,5 + 6,7,8,9,10).GetType()`	
10	`(-7+3 ).GetType()`	

Int32

Double

Double

Int64

Int32

Object[]

Why do you think commands 2, 5 and 6 returned a Double data type rather than in Int32?

PowerShell automatically converts the result of an expression to a data type of sufficient size to not lose information.

NOTE: Windows PowerShell will automatically change the data type of an arithmetic operation result to a different, larger, data type if required. This is referred to as ***widening***.

Task 12.1.2: Non-numeric operations

1. The multiplication and addition operators also work with string data types. The left-hand operand sets the data type for the arithmetic operation. If it is a string, the addition symbol (+) no longer represents numeric addition, but string concatenation.

```
“Hello ” + “World”
```

2. Type the expression below.

```
“Hello ” * 10
```

3. Type the command in reverse order.

```
10 * “Hello”
```

Why do you think reversing the order of the operands causes the operation to fail?

Cannot convert “Hello” to an Integer!

4. Arrays can also be concatenated and multiplied. Try the commands below.

```
1,2,3,4,5 + 6,7,8,9,10
2,4,6,8,10 * 3
$array = “Microsoft”,”XBox”,”Bing”,”Lumia”
$array + “Windows Server”
$array
```

Task 12.1.3: Compound Assignment Operators

1. A variable (described in detail in a previous module) can be assigned a value using the assignment operator (=). Windows PowerShell supports a number of other assignment operators, known as compound assignment operators, which re-assign the result of an operation to a variable.
   Type the commands below. Notice how the value stored in the $var variable changes.

```
$var = 100
$var += 10 ; $var
$var -= 20 ; $var
$var *= 5 ; $var
$var /= 10 ; $var
$var %= 4.5 ; $var
```

2. Compound operators also act on string and array data types.
   Enter the examples below.

```
$array = 1,2,3,4,5
$array += 6
$array
$string = “Hello ”
$string += “ World”
$string
```

Exercise 12.2: Split & Join Operators

Introduction

The split operator splits a string of characters into an array when a specified delimiter character is found, whereas the join operator concatenates strings with an optional delimiter character. Both operators are usable in one of the forms listed below.

1. Unary, where a single operand is specified
2. Binary, where two operands are used.

Objectives

After completing this exercise, you will be able to:

- Use Split and Join operators.

Task 12.2.1: Split Operator

1. The Unary form of the operator will only split the string on occurrences of the <Space> character.

```
-split “I am a space delimited string”
```

2. The binary form allows the use of a delimiter character.

```
“Split,me,on,all,commas” –split “,”
```

3. The delimiter can be preserved, and output, by enclosing it in smooth parentheses.

```
“Split,me,on,all,commas” –split “(,)”
```

4. Split also has the ability to control the maximum number of separate substrings returned. In the last array position, the remaining strings are collected.

```
“Finance,Marketing,IT,Sales,Support,Operations” –split “,”, 4
```

5. Specify delimiter ranges within square braces [ ].

```
"Split;me,on.lots'of:characters" –split “[;,:.']"
```

Task 12.2.2: Join Operator

1. The join operator also has unary and binary forms. You must enclose the array of strings in parentheses, or store them in a variable first. Type the examples below.

```
-join (“Microsoft”,”XBox”,”Windows Phone”,”Bing”)

$strings = “Microsoft”,”XBox”,”Windows Phone”,”Bing”
-join $strings
```

2. The binary form allows a specific delimiter character, added between each string.

```
“Microsoft”,”XBox”,”Windows Phone”,”Bing” –join “:”
```

3. Split the “PSModulePath” environment variable on each occurrence of the semi-colon character (;). Save the result in a new variable, named $modulePaths.

```
$modulePaths = $env:PSModulePath -split ";"
```

4. Append a new string “C:\PShell” to the $modulePaths variable.

```
$modulePaths += “C:\PShell”
```

Join the array of strings back to a single string, separated with the semi-colon character (;). Store the joined string in a new variable called $modifiedPaths. Verify the new path was appended successfully.

```
$modifiedPaths = $modulePaths -join ";"
```

Task 12.2.3: Replace Operator

1. The replace operator will replace all instances of one substring with another, in an input string.

```
“Hello name” –replace “name”, “Matt”
```

Exercise 12.3: Format Operator

Introduction

The format operator formats strings. A format string contains one or more format specifier characters that indicate how a value is to be converted. The format operation string sits on the left-hand side of the operator, and the source objects sit on the right-hand side of the operator.

Objectives

After completing this exercise, you will be able to:

- Use the Format operator

Task 12.3.1: Format Operator

1. The format operator (-f) provides a simple way to perform complex string formatting. The syntax to format a string is below.

{ index [,alignment] [:formatString] } –f "string(s)" , " to be formatted"

The table displays the most common string and number formatting specifiers.

Format specifier	Name
"C" or "c"	Currency
"D" or "d"	Decimal
"E" or "e"	Exponential (scientific)
"F" or "f"	Fixed-point
"G" or "g"	General
"N" or "n"	Number
"P" or "p"	Percent
"R" or "r"	Round-trip
"X" or "x"	Hexadecimal
y,M,d - H,m,s	Date - Time

Type the examples below to understand the format operator syntax. You do not need to type the comments.

The numbers {0}{1}{2}..{*n*} refer to the order of the string on the right-hand side of the –f operator.

```
"{0}{0}{0}" -f "Hello" # repeat

"{0}{1}{1}{0}" -f "Left", "Right" # repeat & reverse sequence
```

```
"{0:C}" -f 215.67 # current locale currency

[math]::PI # longer expression of PI

"{0:N}" -f [math]::PI # round PI to 2 decimal places

"{0:N4}" -f [math]::PI # round PI to 4 decimal places

"{0:P} - {1:P}" -f 0.4, 0.6 # convert to a percentage & adds a dash character
separator

"{0:D8}" -f 1234 # pad with zeros to 8 characters

"{0:X}" -f 255 # convert to hexadecimal

"{0:yyyyMMdd.HHmmss}" -f (Get-Date) # extract date and time parts from Get-Date

"{0:F5}" -f 12345 # add fixed floating point

1..10 | ForEach-Object {"{0:0#}" -f $_} # pad single digit numbers with a leading
zero (0)
```

Note that [math]::PI is a "static" , which we referred to in module 11.

2. Import the XML file C:\PShell\Labs\Lab_12\products.xml and save it in a new variable called $products.

```
$products = Import-Clixml C:\PShell\Labs\Lab_12\products.xml
```

3. Display each product's "Name" property, followed by a colon character and a space, then format the "Price" property as a Currency, using the –f operator. Pipe the $products array of objects to a Foreach-Object cmdlet.

```
$products | ForEach-Object { "{0}: {1:c}" -f $_.Name, $_.Price }
```

Exercise 12.4: Bitwise Operators

Introduction

The bitwise operators BAND, BOR and BXOR and BNOT act on the binary representation of an integer.

Objectives

After completing this exercise, you will be able to:

- Use the bitwise operators.

Task 12.4.1: Bitwise OR operator

1. Open the script C:\PShell\Labs\Lab_12\Optimize-RecycleBin.ps1
   The script contains a single function, called Optimize-RecycleBin, which uses a Windows API function to empty the recycle bin. The function is below.

```
Function Optimize-RecycleBin {

Param($deleteBehaviour)

$signature = @'
    [DllImport("Shell32.dll",CharSet=CharSet.Unicode)]
    public static extern uint SHEmptyRecycleBin(
        IntPtr hwnd,
        string pszRootPath,
        uint dwFlags);
'@

$type = Add-Type -MemberDefinition $signature -Name FileUtil -Namespace W32Tools -
PassThru

    if ($type::SHEmptyRecycleBin([System.IntPtr]::Zero,$null,$deleteBehaviour) -eq
0 )
    {
        return "Recycle Bin Emptied"
    }
}
```

   This function requires the combination of up to three binary values to control visual and audio confirmation of the delete operation. We can use the bitwise OR operator (-bor) to combine these values and change the behavior.
   Create the variables below.

```
$NOCONFIRMATION = 0x00000001
$NOPROGRESSUI = 0x00000002
$NOSOUND = 0x00000004
```

2. Combine the variables with the –bor operator and record the results in the table below.

Binary Operation	Result
`$NOCONFIRMATION –bor $NOPROGRESSUI`	
`$NOSOUND –bor $NOPROGRESSUI`	
`$NOSOUND –bor $NOPROGRESSUI –bor $NOCONFIRMATION`	

3

6

7

3. Execute the function with the –bor operator, as shown below. Make sure that the recycle bin contains one or more deleted items before executing the next command.

```
Optimize-RecycleBin -deleteBehaviour ($NOSOUND -bor $NOCONFIRMATION)
Optimize-RecycleBin -deleteBehaviour ($NOSOUND -bor $NOPROGRESSUI)
Optimize-RecycleBin -deleteBehaviour ($NOCONFIRMATION -bor $NOSOUND -bor
$NOPROGRESSUI)
```

Task 12.4.2: Bitwise AND & OR Operators

1. The bitwise AND operator (-band) is useful when you need to determine whether a bit is set to 1, or "turned on". The System.IO.FileAttributes .NET datatype is an enumeration that stores each possible attribute value as a bit representation.

   Make sure people understand that you can test/set bits !

   Type the line below to view all the possible enumeration values.

```
[Enum]::GetNames([System.IO.FileAttributes])
```

2. Create a new file object and save a reference to it in a variable called $myFile.

```
$myFile = New-Item –Path C:\PShell\Labs\Lab_12\FileAttributes.txt –ItemType File
```

3. Display the attributes property

```
$myFile.Attributes
```

4. Check whether the "Offline" and "Archive" bits are set.

> Note: The bitwise operators return a bit position value, not a Boolean value.

```
($myFile.Attributes -band 0x1000) -eq 0x1000 # Offline
($myFile.Attributes -band 0x20) -eq 0x20 # Archive
```

5. We can iterate through each attribute in the enumeration and use the –band operator to display whether each bit is turned on, or off (true or false).

```
[Enum]::GetValues([System.IO.FileAttributes]) |
Foreach-Object {
$result = ($myFile.Attributes –band $_) –eq $_
```

```
"$_ : $result"
}
```

The bitwise OR operator will change a bit from 0 to 1 ("off" to "on" / "false" to "true").

NOTE: Its best practice to store numerical values that represent a bit position and state in a variable with a descriptive name. This will allow another person to easily read and understand the code.

Type the command below to set the Temporary bit.

```
$temporaryBit = 0x100
$myFile.Attributes = $myFile.Attributes –bor $temporaryBit
```

How would you check that the bit has been set correctly?

$myFile.Attributes
OR
($myFile.Attributes –band $temporaryBit) -eq $temporaryBit

Task 12.4.3: Bitwise Exclusive OR Operator (XOR)

1. Combining the bitwise AND (-band) with the bitwise XOR (-bxor) operator allows us to test the state of a given bit, and modify it if desired.

   Test whether the Archive bit is enabled (set to 1).
   If it is 1, set the bit to 0 using the –bxor operator.

```
if ($myFile.Attributes -band 0x20) # if bit is enabled (1)
{
    # disable bit (0)
    $myFile.Attributes = $myFile.Attributes -bxor 0x20
}
```

2. View the file attributes to confirm the operation succeeded.

```
$myFile.Attributes
```

3. A simpler, more readable, way is to use the enumeration datatype, rather than the hexadecimal value. First, re-enable the Archive Bit.

```
$myFile.Attributes = $myFile.Attributes -bxor 0x20
```

4. Enter and execute the modified code below.

```
if ($myFile.Attributes -band [System.IO.FileAttributes]::Archive) # if bit is
enabled (1)
{
    # disable bit (0)
    $myFile.Attributes = $myFile.Attributes -bxor
[System.IO.FileAttributes]::Archive
}
```

5. View the file attributes to confirm the operation succeeded.

```
$myFile.Attributes
```