Windows PowerShell 4.0 For the IT Professional - Part 1

Module 11: Variables and Data Types

Student Lab Manual

Version 2.0

Conditions and Terms of Use

Microsoft Confidential - For Internal Use Only

Copyright and Trademarks

Contents

Lab 11: Variables and Data Types

Introduction

A variable is a unit of memory in which values are stored. In Windows PowerShell variables are represented by text strings that begin with a dollar sign ($) such as $a, $process, or $my_var. Because variables point to objects, you can think of the variable as the object itself. Windows PowerShell lets you create variables – essentially named objects – to preserve output to use later. If you are used to working with variables in other shells, remember that Windows PowerShell variables are objects, not text.

Objectives

After completing this lab, you will be able to:

- Create and access variables
- Understand data types
- Understand command parsing

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

105 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 11.1: User-Defined Variables

Introduction

User-created variables are created and maintained by the user. By default, the variables that you create at the Windows PowerShell command line exist only while the Windows PowerShell window is open, and they are lost when you close the window.

Objectives

After completing this exercise, you will be able to:

- Create, modify and delete variables.
- Understand variables and data types
- Understand variable naming recommendations

Task 11.1.1: Creating, Modifying & Deleting Variables

Variables are defined and assigned values in a single statement. The assignment operator (=) stores the object(s) on the right-hand side of the operator in the dollar-prefixed variable name on the left-hand side.

1. Type and execute the following command in the Windows PowerShell ISE command pane. Each command creates a new variable and then accesses the variable to display its value.

NOTE: Variables names will TAB complete in both the console and ISE after typing the leading dollar character.

```
$a = 10
$a

$b = “Hello World”
$b

$c = Get-Process
$c

$d = $e = $f = “Multiple variables can be initialized to the same value”
$d ; $e ; $f
```

2. There are a number of ways to create variables. Previously, we used the –Outvariable parameter to assign the object emitted from a Cmdlet to a new variable.

```
Get-Process –Name explorer –Outvariable expProcess
$expProcess
```

3. The *-Variable Cmdlets also allow variable management. Find the Cmdlet to create a new variable. Use it to create a variable named “myVar2”. Write the command you used below.

```
New-Variable –Name myVar2
```

4. The New-Variable Cmdlet has a number of extra options in the form of parameters. Review the help for this Cmdlet, find these parameters and write their names below.

1. -Description
2. Option
3. Visibility
4. Scope

5. Use Get-Help to investigate the *-Variable Cmdlet functionality and syntax. Use the discovered Cmdlets to complete the tasks below.
   a. Create a new variable named “myName” with the Description “This is my name” and the value of <your name>.
   b. Modify the variable’s value to contain the string “Windows Azure”
   c. Remove the variable created in step 6a. How can you ensure it was deleted?
   d. Create a new variable, with the option “ReadOnly”, named “myReadonly” with the value “I’m read only!”
   e. Try to reassign the $myReadOnly variable another value. Did it work?

No.

   f. Use the Remove-Variable’s –Force switch parameter to delete the variable.
   g. Create a new variable, with the option “Constant”, called “myConstant” with the value 42.
   h. Assign the $myConstant variable a different value using either the assignment operator or one of the *-Variable Cmdlets. What happens?

You receive an error stating that the variable is read-only or constant and cannot be re-assigned.

   i. Remove the $myConstant variable using the –Force parameter. Was this successful. How can you delete a constant variable?

Constant variables can only be removed by closing the session in which they were defined.

NOTE: Variable names are not case-sensitive. Variable names can include spaces and special characters, but these are difficult to use and should be avoided.

How could you retrieve a list of all of the currently defined variables? Write the possible commands below.

1.

2.

Commented [A9]: 1.Get-ChildItem –path Variable:
2.Get-Variable

Task 11.1.2: Variables and Data Types

Windows PowerShell automatically converts a variable's data type to the data type of the assigned object, referred to as "casting".

1. Type and execute the following command in the Windows PowerShell ISE command pane. Notice that the semi-colon (;) character is used to separate commands on the same line.

```
$myVar = 10 ; $myVar ; $myVar.Gettype()
$myVar = 92.9 ; $myVar ; $myVar.Gettype()
$myVar = "Microsoft" ; $myVar ; $myVar.Gettype()
$myVar = $true ; $myVar ; $myVar.Gettype()
$myVar = [xml]"<root><child>Some Text Data</child></root>" ; $myVar ;
$myVar.Gettype()
$myVar = Get-Process –Name lsass ; $myVar ; $myVar.Gettype()
$myVar = Get-Volume ; $myVar ; $myVar.Gettype()
```

2. Automatic casting may be over-ridden by the user with the cast operator ([ ]). This operator converts, or limits, the specified type. Enter the command below.

```
[int]$integer = 23
```

3. Type the example below to view this behavior.

```
$integer = 4.3 ; $integer
$integer = 4.6 ; $integer
```

4. What happens to the value stored in the variable?

Its rounded to the nearest integer (whole number).

5. Strings containing numeric characters (0-9) have an implicit cast to the correct numeric data type. Type the code below.

```
$result = 5 + "56"
$result.Gettype()
```

Task 11.1.3: Variable Naming

1. Variables must begin with a dollar character (“$”) and can have practically any name of any length. Variable names can include spaces and special characters, but those are awkward to use. Use the recommended alphanumeric characters and underscores only.

   Curly braces can be used to create and access variable names with spaces and most special characters. Type and execute the following command in the Windows PowerShell ISE command pane. Are you able to set and then retrieve their values once defined?

```
$myVa%r2 = 45
${my var 2} = 56
${*#&my^%$var2} = 78
```

FAIL

SUCCESS

SUCCESS

2. If the variable name must contain curly braces, the backtick (`) character can be used to escape their special meaning when defining and accessing the variable.

```
${my`{variable_contains_braces`}} = “The ‘backtick’ is an escape character”
${my`{variable_contains_braces`}}
```

3. You can access other variables, such as environment variables, using this approach.

```
Get-Item $env:ProgramFiles
Test-Connection $env:USERDNSDOMAIN
```

Exercise 11.2: Automatic Variables

Introduction

Windows PowerShell automatically creates variables for its own state data storage. These variables are accessible from the Windows PowerShell ISE command pane, scripts or functions and contain useful information about the Windows PowerShell environment and dependencies. Automatic variables are created for each session and their values cannot be re-assigned.

Objectives

After completing this exercise, you will be able to:

- Identify variables that store state information for Windows PowerShell.

Task 11.2.1: Automatic variables

1. List the variables in the variable provider drive using either of the methods below.

```
Get-ChildItem Variable:
Get-Variable
```

2. Review the conceptual help topic about_Automatic_Variables and answer the questions below.

   Which variable:

   a. Contains the exit code of the last Windows-based program that was run?
   _____ $LastExitCode

   b. Represents an empty value?
   _____ $NULL

   c. Contains the full path of the Windows PowerShell installation directory?
   _____ $PShome

   d. Holds information about the current Windows PowerShell version?
   _____ $PSversionTable

   e. Contains the full path of the user's home directory?
   _____ $pwd

   f. Holds a collection of objects representing the most recent errors?
   _____ $error

   g. Contains an object that represents the current host application for Windows PowerShell?
   _____ $host

Exercise 11.3: Variable Attributes

Objectives

In this exercise, you will:

- Validate data using variable attributes.

Scenario

You released a script for use on your team, but some team members are supplying invalid data to the script. You need a way to validate user input before the script continues.

Task 11.3.1: Validate numeric input

1. Type and execute the following command in the Windows PowerShell ISE command pane. Prompt a user for input and ensure that the data type of the response is an integer.

```
[int]$num = Read-Host –Prompt "Enter a number"
```

2. Answer **abc**
3. Run step 1 again.
4. Respond **1.5**
5. Print what was stored in the variable.

```
$num
```

6. Run step 1 again.
7. Respond **42**
8. Print what was stored in the variable.

```
$num
```

9. Run step 1 again.
10. Respond **Hello**
11. Print what was stored in the variable.

```
$num
```

Decimal values were accepted. Why do you think this is the case?

[int] conversion. 1.5 was rounded up to 2

Task 11.3.2: Validating input is within a specified range

1. Type and execute the following command in the Windows PowerShell ISE command pane.

```
[ValidateRange(200,300)]$numRange = Read-Host "Enter a number between 200 and 300"
```

2. Respond **abc**
3. Run step 1 again.
4. Respond **199**
5. Run step 1 again.
6. Respond **250**
7. Display the $numRange variable.
8. Type and execute the following command in the Windows PowerShell ISE command pane.

```
[ValidateSet("High","Medium","Low")]$strRange = Read-Host "Enter a valid priority"
```

9. Respond **abc**
10. Run step 8 again.
11. Respond **Top**
12. Run step 8 again.
13. Respond **Medium**
14. Display the $strRange variable.
15. Experiment with the following variable validation attributes to see how they are used.

```
[ValidateCount(0,5)]$arry = 1,2,3,4,5
[ValidateLength(1,3)]$str = "123"
[ValidatePattern('[0-9]+')]$str= "123"
```

Exercise 11.4: Strings

Objectives

After completing this exercise, you will be able to:

- Use and understand literal strings and here strings
- Understand variable sub-expressions

For more information see “Get-Help about_Quoting_Rules”.

Task 11.4.1: Literal Strings

1. Type and execute the following command in the Windows PowerShell ISE command pane, using double quotes to contain the string.

```
$specialChars = "`,$,",{,&,("
```

Does this command work correctly?

NO

2. Try the same command with single quotes. Does this command work?

YES

```
$specialChars = ‘`,$,",{,&,(’
```

3. Define a new string.

```
$preamble = ‘PowerShell interprets the following characters with special meaning ’
```

4. Concatenate the two strings.

```
$preamble + $specialChars
```

5. Now enclose the command above in single quotes. Write the result below.

```
'$preamble + $specialChars'
```

Task 11.4.2: Expandable Strings

1. Expandable strings can easily concatenate a string with the value held in a variable.
2. The command below will return the current Relative Identifier (RID) Master role holder name and store it in the variable $RIDMaster.

```
$RIDMaster = (Get-ADDomain).RIDMaster
```

3. Use double quotes to add a label describing the data held in the variable and display the variable's value.

```
"RID Master: $RIDMaster"
```

4. Remove the double quotes and replace with single quotes. What difference do you notice?

Expansion dos NOT occur

5. The script below will calculate the number of Security Identifiers (SIDS) issued by an Active Directory Domain. Open the script C:\PShell\Labs\Lab_11\Get-RIDPoolCount.ps1 in the Windows PowerShell ISE.

```
$RidManagerPath = 'CN=RID Manager$,CN=System,DC=Contoso,DC=com'
$RID = Get-ADObject -Identity $RidManagerPath -Properties rIDAvailablePool

[int32]$totalSIDS = $RID.rIDAvailablePool / ([math]::Pow(2,32))
[int64]$temp64 = $totalSIDS * ([math]::Pow(2,32))
[int32]$currentRidPoolCount = $RID.rIDAvailablePool - $temp64

$ridsremaining = $totalSIDS - $currentRidPoolCount
```

6. Add labels, using an expandable string, for the following variables, to the end of the script. Save then execute it to display the RID information.
    a. Total SIDS - $totalSIDS
    b. Issued SIDS - $currentRIDPoolCount
    c. Remaining SIDS - $ridsRemaining

Task 11.4.3: Here Strings & variable sub-expressions

1. A here-string is a single-quoted or double-quoted string in which quotation marks are interpreted literally. A here-string can span multiple lines. All the lines in a here-string are interpreted as strings even though they are not enclosed in quotation marks.

   The start delimiter for a Here-String uses an '@' character followed by either a single or double-quote (literal or expandable string). This opening symbol must not be followed by any other text. The end delimiter uses the same characters in reverse, and must be the first characters on the line.

Open C:\PShell\Labs\Lab_11\MailMerge.ps1 in the Windows PowerShell ISE. The script below gets a user object from Active Directory, merges some of its properties into a here-string, then sends it to a local printer.

When accessing an object's property within an expandable string, the statement must be resolved via a sub-expression $() as demonstrated below. This works the same in normal double-quoted strings as it does with double-quoted here-strings.

```
$user = Get-ADUser J.Smith -Properties *

$mailMerge = @"
$($user.givenName) $($user.Surname)

$($user.StreetAddress)
$($user.l)
$($user.State)
$($user.PostalCode)
$($user.Country)

Dear $($user.GivenName),

Your Microsoft Active Directory account has been created with the following
information.
Please review and notify us in the event of incorrect personal details.

Logon Name: $($user.samaccountname)
Division: $($user.Division)
Department: $($user.Department)
Company: $($user.Company)
"@

$mailMerge | Out-Printer -Name "Microsoft XPS Document Writer"
```

2. Execute the script. When prompted save the printed file. Open the file to view the merged data.

Exercise 11.5: Using Types

Objectives

After completing this exercise, you will be able to:

- Use type member discovery
- Know how to strongly-type a variable
- Leverage type casting

Task 11.5.1: Type Member Discovery

1. As previously demonstrated, everything in Windows PowerShell is an object and objects are instances of data types. We can use the Get-Member Cmdlet to uncover this information.
   Type the examples below and write the "TypeName", displayed at the top of each block of output.

```
"Hello World" | Get-Member
```

TypeName: _____ System.String

```
12.6 | Get-Member
```

TypeName: _____ System.Double

```
Get-Date | Get-Member
```

TypeName: _____ System.DateTime

```
Get-Service | Get-Member
```

TypeName: _____ System.ServiceProcess.ServiceController

2. Some data types also implement members of the class, not the object instance. These are known as Static members and can be displayed with the Get-Member Cmdlet's –Static switch parameter.
   Type the commands below to display the Static members of the String data type. Both commands display the same member information.

```
"Hello World" | Get-Member –Static
[System.String] | Get-Member -Static
```

3. Static members (properties and method) are accessed using the static member operator (::) directly on the data type name, which is enclosed in square braces.
4. First, get the local computer's network card MAC Address and store it in the variable $mac.

```
$mac = (Get-NetAdapter -InterfaceIndex 3).MacAddress
```

5. The MAC Address octets are delimited with a hypen (-) character. There is a requirement to replace this with a colon (:) delimiter character. Split the string on each hyphen character and save the octets in a variable $octets.

```
$octets = $mac.Split("-")
```

6. Finally, use the string data type's Join() static method to concatenate the octets into a colon delimited string.

```
[string]::Join(":" , $octets)
```

7. This same task can also be accomplished using the string method Replace.

```
(Get-NetAdapter -InterfaceIndex 3).MacAddress.Replace('-',':')
```

8. The System.Environment data type has a number of useful static members. Use Get-Member to list only the static methods.

```
[System.Environment] | Get-Member -Static -MemberType Method
```

9. How many methods did you find? _____

10. Type the commands below in a new script window in the Windows PowerShell ISE, or copy them from the script file C:\PShell\Labs\Lab_11\SystemInfo.ps1.
Add a comment on each line detailing the data type returned by each member.

```
[System.Environment]::Is64BitOperatingSystem    # boolean
[System.Environment]::SystemPageSize
[System.Environment]::MachineName
[System.Environment]::ProcessorCount
[System.Environment]::SystemDirectory

$OSVersion = [System.Environment]::OSVersion
$OSVersion.Platform
$OSVersion.Version

[string]::IsNullOrEmpty($OSVersion.ServicePack)

$userName = [System.Environment]::UserName
$userDomain = [System.Environment]::UserDomainName
$NTAccount = [string]::Join("\",$userName,$userDomain)
$NTAccount.ToUpper()

$sysDirSize = (Get-Childitem -Path ([System.Environment]::SystemDirectory) `
-Recurse -ErrorAction SilentlyContinue |
Measure-Object -Property length -Sum).Sum /1GB

[Math]::Round($sysDirSize, 2)
[system.environment]::GetFolderPath("MyDocuments")
```

11. To make this easier to run, encapsulate the code above as a function called Get-SystemInfo. Save the changes in a script file with the same name C:\PShell\Labs\Lab_11\Get-SystemInfo.ps1

```
function Get-SystemInfo {
<code from step 10 goes here>
}
```

Task 11.5.2: Strongly Typing Variables

1. Windows PowerShell automatically sets a variable's data type during assignment.

```
$myVar = 10
```

2. Display the Variable's data type.

```
$myVar.GetType()
```

3. Assigning a string to the same variable succeeds.

```
$myVar = "Hello"
```

4. We can explicitly set a variable's type by specifying the data type within square braces [ ] and prepending it to the variable name. This known as 'type casting'.

```
[int]$myVar = 10
```

5. Now, assign a string to $myVar.

```
$myVar = "Hello"
```

6. Did the previous command succeed? Explain why or why not below.

NO. type was cast to [INT] so it refused to accept a System.String

7. We can remove the explicit data type by casting the variable to a System.Object data type.

```
[object]$myVar = "Hello"
```

8. Try assigning the result of running the Get-Date Cmdlet to the $myVar variable. Did it work? **Yes / No**

YES

9. Enter the command below.

```
$xmlDoc = "<root><child>some data</child></root>"
```

10. Write the data type of $xmlDoc below.

System.String

11. The syntax of the string used above is XML, however Windows PowerShell interprets it as a simple string. We can strongly-type the variable to an XML object (System.XML.XMLDocument) to allow us access this as structured XML data.

    Notice that we also cast an integer to a Char data type to create the Euro (€) currency symbol.

```
$euro = [char]8364

[xml]$xmlDoc = `
```

```
"<book><title>1984</title><author>Orwell</author><price>" + $euro +
"10</price></book>"
```

12. Pipe the variable to Get-Member and write the type name below.

System.Xml.Document

13. Access the XML Document's nodes using dot-notation.

```
$xmlDoc.book
$xmlDoc.book.author
$xmlDoc.book.title
$xmlDoc.book.price
```

14. Access the XML Document nodes using the familiar dot-notation and record the data types of the 'book' node, below.

```
$xmlDoc.book | Get-Member
```

book: _____

Task 11.5.3: Typecast operators

1. Create a new string that represents a date in the following format "dd/MM/yyyy"

```
"17/06/2014"
```

2. Confirm that a string object was created.

```
"17/06/2014" | Get-Member # just a string
```

3. Cast the string to a DateTime object.

```
[DateTime]"17/06/2014"
```

4. Did this work correctly? What does the error indicate?

Cannot convert value "17/06/2014" t type "System.DateTime". Error: "String was not recognised as a valid DateTime"

5. This time, use the –as type operator to cast the string to DateTime object. Do you receive any output?

NO. Same error. USA-specific date format is the default.

6. Use the Get-Culture Cmdlet to view the current culture settings. What displayName is returned?

English (United States)

7. To correctly parse date strings using a specific culture's date format, the DateTime data type's static Parse() method can be used. This method requires two arguments, a string in the target culture's date format, and an object that represents the target culture.
8. Use the GetCultures() static method from the System.Globalization.CultureInfo class to return all of the currently installed culture types.

```
[System.Globalization.CultureInfo]::GetCultures("AllCultures")
```

9. Find the "Australian" culture type and save the "Name" property in a new variable named $EnAU.

```
$EnAU = [System.Globalization.CultureInfo]::GetCultures("AllCultures") |
Where-Object displayname -match "Australia"
```

10. Finally, we can cast the localized date string "dd/MM/yyyy" to a DateTime object using Date formatting rules of the target culture.

```
$parsedCultureDate = [DateTime]::Parse("17/06/2014",$EnAU)
```

11. The object's data type can now be confirmed using the –is or –isNot type operators.

```
$parsedCultureDate –is [DateTime]
$parsedCultureDate –isNot [DateTime]
```

Exercise 11.6: Command Parsing

Introduction

When you enter a command at the command prompt, Windows PowerShell breaks the command text into a series of segments called "tokens" and then determines how to interpret each "token."

When processing a command, the Windows PowerShell parser operates in one of two modes:

Expression mode - character string values must be contained in quotation marks. Numbers not enclosed in quotation marks are treated as numerical values (rather than as a series of characters).

Argument mode - each value is treated as an expandable string unless it begins with one of the following special characters:

- Dollar sign $
- at sign @
- single quotation mark '
- double quotation mark "
- or an opening parenthesis (

Objectives

After completing this exercise, you will be able to:

- Understand command-parsing modes.
- Identify and use special and escape characters.

Task 11.6.1: Command Parsing Modes

1. Type the following command. Notice that the parameter value ‘6*5’ is interpreted as a simple string and displayed. This is an example of argument mode parsing.

```
Write-Output –InputObject 6*5
```

2. If we now enclose the parameter value in parentheses, the Windows PowerShell command parser enters expression parsing mode and the characters ‘6*5’ are interpreted as two integers and a multiplication operator.

```
Write-Output –InputObject (6*5)
```

3. Accessing a variable’s leading dollar ($) character also causes expression mode to be entered.

```
$int = 5+5
$int
```

4. Command mode can be entered using the call operator (&).
   Enter the example below.

```
$cmd = "get-command"
```

5. Type the variable name $cmd. The contents are displayed as a string of characters.
6. Now, prepend the variable with the call operator (&) and press enter. What output do you receive?

System.String as expected.

The output of the Get-Command cmdlet. i.e. the list of commands available.

7. Make a copy of the script from Module 5 in C:\PShell\Labs\Lab_5\Create-BpaReport.ps1 and rename the copy to include a space in its name, such as "Create BPA Report.ps1".

```
$folderPath = "C:\PShell\Labs\Lab_5"
Copy-Item –Path $folderPath\Create-BpaReport.ps1 `
–Destination "$folderPath\Create Bpa Report.ps1"
```

8. Change you current working directory to C:\PShel\Labs\Lab_5.

```
Set-Location –Path $folderPath
```

9. Type the relative path prefix ".\" and press <Tab> until you see the name of the script. Note that since the path contains spaces, Windows PowerShell will enclose the script path in single quotes and prefix the call operator (&).

Prefix the command with the call operator and enclose the script path in quotes PS C:\> & 'C:\PShell\Labs\Lab_5\Create BPA Report.ps1'

Task 11.6.2: Escape and special characters

Windows PowerShell's escape character is the backtick (`) or grave accent character (ASCII 96). This character applies a special interpretation to the character immediately following it.

For more information, refer to the conceptual help file about_Escape_Characters.

The backtick has a number of uses:

- Indicate a literal
- Indicate line continuation
- Indicate special characters

1. When an escape character precedes a variable, it prevents the variable's value from being displayed. The example below uses the backtick to produce a string literal $userDomain within an expandable string.

```
$userDomain = $Env:UserDomain
"The value of `$userDomain is: $userDomain"
```

2. The backtick can also escape space characters in a string. Type the following commands to view their effect.

```
Get-ChildItem C:\Program Files
Get-ChildItem C:\Program` Files
Get-ChildItem "C:\Program Files"
```

3. When using Cmdlets with many parameters or writing complex scripts, it is useful to split long commands over multiple lines. The backtick character also provides this functionality. The following command uses the Get-CIMInstance Cmdlet to return all services that are set to Automatic start mode but are not running. Type the command exactly as specified below.

```
Get-CimInstance -Namespace root/cimv2 -OperationTimeoutSec 10 `
 -Query "SELECT * FROM Win32_Service WHERE StartMode='Auto' AND State='Stopped'"
```

4. The backtick can be paired with a number of other characters to provide special instructions to the parser. These special characters should be stated within quotation marks in order to work correctly. Note that these characters are case sensitive!

Special Character	Description
`0	Null
`a	Alert
`b	Backspace
`f	Form feed
`n	New line
`r	Carriage return
`t	Horizontal tab
`v	Vertical tab

Type the commands below.

```
“Each `n word `n is `n on `n a `n new `n line”
“Insert `t tab `t characters `t between `t words”
“Make the computer speaker beep `a”
```

Task 11.6.3: Stop Parsing Character

When executing external command-line programs from Windows PowerShell, the ability to prevent the parser from executing at a certain point is very useful. The stop parsing special symbol (--%) provides this functionality.

1. Create a new folder called TestFolder in C:\PShell
2. Use the icacls.exe external command to grant permissions to the new folder for the local Guest account on WIN8-WS.
   First, try the command without the stop parsing special character.

```
icacls C:\PShell\TestFolder /grant WIN8-WS\Guest:(CI)(OI)F
```

Then with the stop parsing character.

```
icacls C:\PShell\TestFolder --% /grant WIN8-WS\Guest:(CI)(OI)F
```

Exercise 11.7: Default Parameter Values

Objectives

In this exercise, you will:

- Create, disable, and remove a PSDefaultParameterValue

Scenario

You have junior administrators who are learning PowerShell. You are concerned that they may accidentally make sweeping changes to the environment. You need to implement a safety measure as an automatic checkpoint before they make changes to the environment.

Task 11.7.1: Working with PS Default Parameter Values

HINT: See about_Parameters_Default_Values in PowerShell help.

1. Create a default parameter value that sets the **WhatIf** parameter to true for any cmdlet beginning with "New".

```
$PSDefaultParameterValues = @{"New*:WhatIf"=$true}
```

2. Attempt to create a new alias "gh" for Get-Help.

```
New-Alias gh Get-Help
```

3. Disable the default parameter without deleting it.

```
$PSDefaultParameterValues.Add("Disabled",$true)
```

4. Attempt to create a new alias "gh" for Get-Help.

```
New-Alias gh Get-Help
```

5. View the default parameters.

```
$PSDefaultParameterValues
```

6. Clear all default parameter values.

```
$PSDefaultParameterValues = $null
```

NOTE: To override default parameters use this syntax:

New-Alias gh Get-Help -WhatIf:$false

Task 11.7.2: Implementing PS Default Parameter Values

HINT: See about_Parameters_Default_Values in PowerShell help.

Question A: PS default parameter values disappear once the session is closed. How could you make them persist for new sessions?

Add them to the profile.

Question B: What is the benefit of forcing the **-WhatIf** switch for Cmdlets that make changes?

Keep you from shooting yourself in the foot by accident. ☺

Question C: How would you modify the example in this lab so that it would affect any Cmdlet that changes or creates? Write your modified script line below:

$PSDefaultParameterValues = @{"Set-*:WhatIf"=$true;"New-*:WhatIf"=$true}