Windows PowerShell 4.0 For the IT Professional - Part 1

Module 16: Scope

Student Lab Manual

Version 2.0

Conditions and Terms of Use

Microsoft Confidential - For Internal Use Only

Contents

Lab 16: Scope

Introduction

Windows PowerShell protects access to variables, aliases, functions, and Windows PowerShell drives (PSDrives) by limiting where they can be read and changed. By enforcing a few simple rules for scope, Windows PowerShell helps to ensure that you do not inadvertently change an item that should not be changed.

Objectives

After completing this lab, you will be able to:

- Identify the multiple scopes for variables.
- Include other scripts using dot sourcing.
- Use profiles to customize your scripting environment.

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

30 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 16.1: Scopes

Introduction

An item included in a scope is visible in the scope in which it was created and in any child scopes, unless it is explicitly made private.

An item can only be changed within the scope in which it was created, unless you explicitly specify a different scope.

If you create an item in a scope and the item shares its name with an item in a different scope the original item might be hidden by the new item, but it is not overridden or changed.

Visualize bubbles within bubbles. When the inner bubble pops, all its contents are gone.

We have found this a very effective way to explain scopes to the target audience for this workshop.

Objectives

After completing this exercise, you will be able to:

- Understand local, script, global and private scopes

Task 16.1.1: Local Scope

$local - the current scope (whatever that may be at the time)
Below is the sample code. Examine the script, then execute it and consider the outputs.

1. Open and execute the following script within the Windows PowerShell ISE script pane. C:\Pshell\Labs\Lab_16\LocalScope.ps1

```
$time = get-date
$day  = (get-date).day

function get-theTime() {
   #create a local $time variable
   $time=(get-date).Minute
   $milli = (get-date).Millisecond
   Write-Output "Function: ``$Time'  variable is $time"
   Write-Output "Function: ``$day'   variable is $day"
   Write-Output "Function: ``$milli' variable is $milli"
 }

get-theTime
Write-Output "Script:   ``$Time'  variable is $time"
Write-Output "Script:   ``$Day'   variable is $day"
Write-Output "Script:   ``$milli' variable is $milli"
Write-Output "NOTE: ``$milli' only existed within the life of the function call!"
```

Task 16.1.2: Private Scope

$private – these cannot be seen outside the current scope.

Below is the sample code. Examine the script, then execute it and consider the outputs.

1. Open and execute the following script within the ISE script pane.
   C:\Pshell\Labs\Lab_16\PrivateScope.ps1

```
$time = get-date
$Private:Pday = (get-date).Day

# new-variable -name Pday -Option private -Value (get-date).day

function Look-AtVars() {
   Write-Output "Function: ``$time' is viewable and has a value of $time"
   Write-Output "Function: ``$Pday' is private and has a value of $Pday"
}

Write-Output "Script:   ``$time' is viewable and has a value of $time"
Write-Output "Script:   ``$Pday' is private and has a value of $Pday"
Look-AtVars
Write-Output "NOTE: the Pday was not viewable from within the function!"
```

Task 16.1.3: Script Scope

$script - the things within the script as the script runs.

Below is the sample code.

1. Type in the following script path, within the ISE script pane. Highlight the line, and then press the <F8> key to run it.

```
C:\Pshell\Labs\Lab_16\ScriptScope.ps1
```

2. Consider the outputs. Are the variables with the displayed values still present within the ISE?

```
 $time = get-date

function Set-ScriptVar() {
   $time = 1200
   Write-Output "Function: ``$time' has a value of $time"
   Write-Output "Function: script level ``$time' has a value of $script:time"
   $time = 1300
   $script:time = (get-date).AddYears(-6)
   Write-Output "Function: altered value of ``$time' to $time"
   Write-Output "Function: altered value of script level ``$time' to $script:time"
 }

Write-Output "Script:   ``$time' has a value of $time"

Set-ScriptVar

Write-Output "Script:   ``$time' has a value of $time"
Write-Output "NOTE: the year was changed from within the function!"
```

Task 16.1.4: Global Scope

$global - automatic variables, preference variables, etc. $home for example.

Below is the sample code.

1. Run the following script within the ISE script pane.

```
C:\Pshell\Labs\Lab_16\GlobalScope1.ps1
```

2. Examine the output based on the following logic. Are the variables still present?

```
 # get-Variable -Scope Global

 Write-Output "``$HOME'                    is: $HOME"
 Write-Output "``$PSCULTURE'               is: $PSCULTURE"
 Write-Output "``$ERRRORACTIONPREFERENCE' is: $ERRORACTIONPREFERENCE"
 Write-Output " "
 $global:MYGLOBAL="MyGlobalVariable"
 Write-Output "``$MYGLOBAL'                is: $MYGLOBAL"
 Write-Output "NOTE: Global ``$MYGLOBAL' will be viewable in other scripts and
functions"
 Write-Output " "
 Write-Output "Try this: Get-Variable -Scope Global"
```

3. Without leaving the ISE, run the following script. Type in the below, highlight it and press <F8>.

```
C:\Pshell\Labs\Lab_16\GlobalScope2.ps1
```

4. Examine the outputs. Compare the values from the previous steps.

```
# without exiting the ISE
#GlobalScope2.ps1

 function Get-GlobalVar() {
   Write-Output "Function: ``$MYGLOBAL' has a value of $MYGLOBAL"
 }

 function Set-GlobalVar() {
   $global:MYGLOBAL="aNewValueForGlobal"
   Write-Output "Function: ``$MYGLOBAL' has a value of $MYGLOBAL"
 }
 Write-Output "``$HOME'                    is: $HOME"
 Write-Output "``$PSCULTURE'               is: $PSCULTURE"
 Write-Output "``$ERRRORACTIONPREFERENCE' is: $ERRORACTIONPREFERENCE"
 Write-Output "``$MYGLOBAL'                is: $MYGLOBAL"
 Write-Output " "
 Write-Output "Script:   ``$MYGLOBAL' has a value of $MYGLOBAL"

 Get-GlobalVar
 Write-Output "NOTE: the global variable has been accessed from a separate
script!"
 Write-Output " "

 Set-GlobalVar
 Write-Output "Script:   ``$MYGLOBAL' now has a value of $MYGLOBAL"
 Write-Output "NOTE: the global variable has been set from a separate script!"
```

Formatted: Heading 2, Space After: 0 pt, Line spacing: single, Tab stops: Not at 9.78 cm + 10.56 cm

Exercise 16.2: Dot Sourcing

Introduction

Each script runs in its own scope. The functions, variables, aliases, and drives created in the script exist only within the script scope. You cannot access these items or their values in the scope in which the script runs. To run a script in a different scope, you can specify a scope, such as Global, Script, Local or Private, or you can dot source the script.

Objectives

After completing this exercise, you will be able to:

- Understand the dot sourced scope.

Task 16.2.1: Dot Sourced Scope

The dot-sourcing feature lets you run a script in the *current* scope instead of in the script scope. When you run a script that is dot sourced, the commands in the script run as though you had typed them at the command prompt. The functions, variables, aliases, and drives that the script creates, are now within the scope in which you are working. After the script runs, you can use the created items and access their values in your session. To dot source a script, type a dot (.) and a space before the script path.

1. Review the following sample script

```
#DotSourcedScope.ps1

$DotSourcedTime = get-date
$DotSourcedDay  = (get-date).day
$DotSourcedNum  = "17061961"

Write-Output "'`$DotSourcedTime' variable is $DotSourcedTime"
Write-Output "'`$DotSourcedDay'  variable is $DotSourcedDay"
Write-Output "'`$DotSourcedNum'  variable is $DotSourcedNum"

Write-Output "The above variables are still present and accessible."
```

2. Within the ISE, type in the command below. Highlight it and press <F8>. Note this is not using dot sourcing.

```
C:\pshell\labs\lab_16\DotSourcedScope.ps1
```

3. Use Intellisense in the ISE to see if the variables are in your session. Type "$Dot". Do you see any variable names show up for auto-completion?

   Yes / No

4. Within the ISE, type in the command below. Highlight it and press <F8>. This time we *are* using dot sourcing.

```
. C:\pshell\labs\lab_16\DotSourcedScope.ps1
```

5. Use Intellisense in the ISE to see if the variables are in your session. Type "$Dot". Do you see any variable names show up for auto-completion?

   Yes / No

YES

After the dot sourced script runs, the functions and variables that the script creates will exist within the current scope, rather than being discarded.

Exercise 16.3: Profiles

Introduction

Customizations for your Windows PowerShell environment can be set within a *Profile* script. You might define convenient Functions and Aliases for your session. As you saw in "scopes" these settings will only exist within a given "scope" or lifetime. Each time Windows PowerShell is launched, the profile script is executed. This means that whatever runs in the profile will automatically be in the scope of your new Windows PowerShell session.

Objectives

After completing this exercise, you will be able to:

- Understand the profiles used by PowerShell console and ISE.

Task 16.3.1: Profiles

1. The "profile" script, if present, will run when PowerShell starts. There are several profile files. Type the commands below into the Windows PowerShell ISE and into a PowerShell console and review.

```
$PROFILE
$PROFILE.CurrentUserCurrentHost
$PROFILE.CurrentUserAllHosts
$PROFILE.AllUsersCurrentHost
$PROFILE.AllUsersAllHosts
```

2. To which file(s) does the "*AllHosts" locations refer?

Profile.ps1

3. To which file(s) does the "*CurrentHost" profiles refer?

Microsoft.PowerShellISE_profile.ps1

4. Are the $HOME and $PSHOME global variables leveraged to generate the appropriate paths?

Yes, indirectly. However you do not see them embedded in the paths.

Task 16.3.2: Creating a Profile script

1. Creating a profile is as simple as creating any other file. It is simply a Windows PowerShell script file (.ps1).

```
#
If (-NOT (test-path $PROFILE)) {
  New-Item -Type file -Path $PROFILE –Force
}
#
```

> Note: To create an "AllUsers" profile, by referencing "$PROFILE.AllUsersCurrentHost", you must start Windows PowerShell with Run as Administrator.

2. Type and execute the following script to ensure the profile folder exists.

```
$BasePath = Split-Path $PROFILE.CurrentUserAllHosts -Parent
if ((Test-Path $BasePath) -eq $false)
{
    New-Item -Path (Split-Path $BasePath -Parent) -Name (Split-Path $BasePath -
Leaf) -ItemType Directory
}
```

> Note: On a default Windows install, you may have to create the **WindowsPowerShell** folder within your **Documents** directory, before you can create the file (you will be prompted to create it if it does not exist).

3. Create a new profile script using the command below. The CurrentUserAllHost property contains the path to $Home\Documents\WindowsPowerShell\profile.ps1

```
New-Item $PROFILE.CurrentUserAllHosts -ItemType File -Force
```

4. Open the new profile script in the Windows PowerShell ISE, from a Windows PowerShell console.

Earlier builds created the file for you if not present. This changed at some point!

```
ISE $PROFILE.CurrentUserAllHosts
```

The WindowsPowerShell console and ISE can have separate profile scripts. Because we want to create a profile that applies to both, we must use the **CurrentUserAllHosts** Property. You can have multiple different profiles loaded at the same time.

5. Add the function and alias below to the profile script to display the current time. Since an Alias is unable to take parameters, create a function to return only the time portion from the Get-Date Cmdlet, and then configure the alias to refer to the new function.

```
function Get-Time {
  Get-Date -DisplayHint Time
```

```
}

New-Alias -Name Time -Value Get-Time
```

6. Save your new profile script. Whenever you start Windows PowerShell, the script will run and create the function and alias.
7. Exit your Windows PowerShell ISE or Console session.
8. Relaunch Windows PowerShell to load the new profile.
9. Enter the "Time" alias to confirm the profile loaded the alias and function.

Task 16.3.3: Integrate Scripting Environment (ISE) Profile

Finally, create a Windows PowerShell ISE specific profile to add a menu item to launch the Bing search page.

1. Start the Windows PowerShell ISE to create and edit the Windows PowerShell ISE specific profile.

```
ISE $HOME\Documents\WindowsPowerShell\Microsoft.PowerShellISE_profile.ps1
```

2. The $psISE variable exists only within the Windows PowerShell ISE and allows access to its user interface. From within our profile, we call the appropriate Add() method to place a new item on the "Add-ons" menu. Add the following code into your profile script in Notepad.

```
$psISE.CurrentPowerShellTab.AddOnsMenu.SubMenus.Add(
    "Bing It!", {
        Start "http://www.bing.com.au"
    },
    "Control+Alt+1"
)
```

3. Save the new profile.
4. Close the Windows PowerShell ISE.
5. Relaunch the Windows PowerShell ISE. You should see your new menu item under "Add-ons".
6. It is possible to alter many settings within the ISE environment. To see just some of the possibilities, explore at the Options property on the $psISE object.

```
$psISE.Options
```

7. Try altering the console background color:

```
$psISE.Options.ConsolePaneBackgroundColor="orange"
```

8. Once you have experimented with the various Options properties, you can reset to default by using the RestoreDefaults method.

```
# It is useful to know how to reset your modifications!
```

```
$psISE.Options.RestoreDefaults()
```