**PowerShell Study Notes**

**October 2011**

**Amr Eldib**

**@AmrEldib**

Table of Contents

[Introduction 3](#_Toc307838332)

[UI 3](#_Toc307838333)

[Aliases 3](#_Toc307838334)

[Basic Commands 4](#_Toc307838335)

[Get-Command 4](#_Toc307838336)

[Get-Help 4](#_Toc307838337)

[Get-ChildItems 4](#_Toc307838338)

[Set-Location 4](#_Toc307838339)

[Clear-Host 4](#_Toc307838340)

[Piping Commands 5](#_Toc307838341)

[Providers 5](#_Toc307838342)

[Variables 6](#_Toc307838343)

[Variable Types 6](#_Toc307838344)

[Comparisons 7](#_Toc307838345)

[Implicit Type Conversion 7](#_Toc307838346)

[Strings 8](#_Toc307838347)

[String Interpolation 8](#_Toc307838348)

[String Formatting 8](#_Toc307838349)

[Wildcards 8](#_Toc307838350)

[Arrays 9](#_Toc307838351)

[Hash Tables 9](#_Toc307838352)

[Common Built-in Variables 10](#_Toc307838353)

[Branching 10](#_Toc307838354)

[If Statement 10](#_Toc307838355)

[Switch Statement 11](#_Toc307838356)

[Looping 11](#_Toc307838357)

[Script Blocks 12](#_Toc307838358)

[Functions 13](#_Toc307838359)

[Comments 13](#_Toc307838360)

[Adding Help to Functions 13](#_Toc307838361)

[Error Handling 14](#_Toc307838362)

[Working with Files 14](#_Toc307838363)

# Introduction

DOS was a command-line based operating system. CMD lived on when Windows replaced DOS. CMD was long overdue for replacement and PowerShell does exactly that.

PowerShell is based .NET and everything is a .NET object.

Commands in PowerShell are named “Cmdlets” (Command-lets).

PowerShell commands have a Verb-Noun syntax.

Verbs like: Get, Set, Out, Start, Stop, Restart, Add.

Nouns like: Help, Command, Service, Computer, Location, ChildItems.

# UI

Use can access PowerShell through:

* PowerShell Window: just like CMD but for PowerShell.
* PowerShell ISE (Integrated Scripting Environment): helps with writing and executing scripts along with typing regular commands. Better alternative to the PowerShell Window.
* Console2: if you’re a fan of Console2 you can use it for PowerShell. Here’s [how you can do that](http://gurustop.net/blog/2010/12/23/small-tip-using-console2-with-powershell/).

# Aliases

DOS and Linux commands work in PowerShell through Aliases.

PowerShell can accept aliases to its commands. For example these three commands are the same:

* Dir (DOS)
* LS (Linux)
* Get-ChildItems (PowerShell)

To find list of aliases use: **Get-Alias**

To create an alias use: **Set-Alias <Alias> <PowerShell-Command>**

Example: *Set-Alias List Get-ChildItems*

Aliases live as long as the PowerShell window is open.

To export the aliases list to a file (CSV is the export format) use the command:

**Export-Alias <Path-To-CSV-File> <Command-Filter>**

The command filter is optional.

To import the file back to PowerShell use: **Import-Alias <Path-To-CSV-File>**

# Basic Commands

## Get-Command

Gets all the PowerShell commands.

Example:

* *Get-Command*
* *Get-Command –verb “get”*
* *Get-Command –noun “service”*

## Get-Help

Gets basic information about Cmdlets and other elements of PowerShell commands.

Format: *Get-Help <Command> -<Optional-Parameter>*

Example:

* *Get-Help Get-Command*
* *Get-Help Get-Command –examples*
* *Get-Help Get-Command –detailed*
* *Get-Help Get-Command –full*

## Get-ChildItems

Lists sub items under the current location.

Example: *Get-ChildItems*

## Set-Location

Changes the current path whether it’s a directory or tree of objects.

Format: *Set-Location “<New-Location>”*

Example:

* *Set-Location C:\Windows*
* *Set-Location “C:\Program Files”*

## Clear-Host

Clears the screen.

# Piping Commands

In PowerShell, “Piping” is the process of chaining commands so that the output of the first command can be channeled as an input to the second command whose output will be the input of the third command and so forth.

The name comes from the pipe symbol “|” (usually Shift + the key above the left Enter) used to separate commands.

Example:

* *Get-ChildItems | where-object { $\_.Length –gt 100kb }*

The pipe takes the output of the “Get-ChildItems” and passes it to the “Where-Object” command which prints out the result of files that have length greater than 100 KB.

When writing piped commands on multiple lines you have to end each line with the pipe symbol (except the last line which ends with the last command).

# Providers

PowerShell uses providers which provide access to data and components that
would not otherwise be easily accessible at the command line. The data
is presented in a consistent format that resembles a file system drive ([Source](http://www.powershellpro.com/powershell-tutorial-introduction/powershell-providers/)).

To list PowerShell providers use: **Get-PSProvider**

We connect to PowerShell Providers by mounting the Providers PowerShell Drive (PSDrive). Most Providers have only one PSDrive, the exceptions are the FileSystem Provider (depends on the number of drives on the system) and the Registry Provider (HKLM and HKCU) ([Source](http://www.powershellpro.com/powershell-tutorial-introduction/powershell-providers/)).

To list PowerShell drives use: **Get-PSDrive**

To move/switch to a certain drive use: **Set-Location <PSDrive>:**

Example: *Set-Location Env:*

To get a list of currently loaded Snap-ins use: **Get-PSSnapIn**

To get a list of Snap-ins that are registered but not currently loaded use: **Get-PSSnapIn -Registered**

To add a snap-in use: **Add-PSSnapin <Snap-In-Name>**

Adding a snap-in will add a new drive that you can navigate to. For example, adding the “SqlServerCmdletSnapin100” snap-in will add the “SQL” drive.

To remove a snap-in use: **Remove-PSSnapIn <Snap-In-Name>**

# Variables

To create a variable just put a dollar sign ($) before the name of the variable and assign a value to it.

Example: *$hi = “Hello World”*

This is a shortcut for using the “New-Variable” cmdlet. You can use the long form.

Example: *New-Variable –Name hi –Value “Hello World”*

To assign a value to an existing variable, use the **Set-Variable** cmdlet.

Example: *Set-Variable –Name hi –Value 5*

To print out a variables value, just write the name of the variable after a dollar sign.

Example: *$hi*

To clear the content of a variable (like setting it to Null), use the **Clear-Variable** cmdlet.

Example: *Clear-Variable –Name hi*

This is a shortcut for using the “Write-Host” cmdlet. You can use the long form.

Example: *Write-Host $hi*

It’s also a shortcut for using the “Get-Variable” cmdlet. You can use the long form.

Example: *Get-Variable hi –valueonly*

The cmdlet **Get-Variable** (without any parameters) will list all the variables in PowerShell.

To remove a variable from memory, use the **Remove-Variable** cmdlet.

Example: *Remove-Variable –Name hi*

## Variable Types

To Get type of a variable use: **<Variable-Name>.GetType()**

Example: *$hi.GetType()*

PowerShell types are mutable. Assigning an integer value to a variable holding a string will change the type of the variable from string to integer.

You can declare a variable and assign a specific type to it. This will cause PowerShell to throw an error if you assign a wrong type of value to it.

To declare the type of a variable, write the .NET full name of the type before the variable name in square brackets.

Example: *[System.Int32]$myint = 42*

## Comparisons

PowerShell doesn’t use symbols for comparisons. Instead it uses short acronyms following a dash.

|  |  |
| --- | --- |
| Greater Than | -gt |
| Less Than | -lt |
| Equal To | -eq |
| Not Equal To | -ne |
| Greater Than or Equal | -ge |
| Less Than or Equal | -le |
| Like | -like |
| Not Like | -NotLike |
| Match based on regular expressions | -Match |
| Non-match based on regular expresions | -NonMatch |

Calculations are like any other language. You can use +, -, ++, -- and /

## Implicit Type Conversion

PowerShell converts types implicitly which can be very helpful. However, it can cause confusion when applied to comparisons.

When performing a comparison between two different types, PowerShell will convert the variable on the right side to the type of the variable on the left side to be able to perform the comparison. In this comparison:

*“42” –eq 42*

The 42 on the right is an integer but the one on the left is a string. To compare the two, PowerShell will convert the integer to string, resulting in “42” which matches the value on the left. This comparison will result into True.

However, this comparison:

*“042” –eq 42*

The 42 on the right will be converted to the string “42” which is NOT equal to the string “042”. This will result into False. But if you switch the sides like this:

*42 –eq “042”*

The string “042” will be converted to the integer 42 which will match the left side. This will result into True.

Implicit type conversion makes comparison in PowerShell a bit tricky, so watch out.

# Strings

You can single quotes or double quotes around strings.

The escape character in PowerShell is the backtick (left to the 1 key and below the Esc key).

Some escape sequences:

|  |  |
| --- | --- |
| Backspace | `b |
| New Line | `n |
| Carriage Return | `r |
| Carriage Return Line Feed | `r`n |
| Tab | `t |

A “Here String” is a way of writing text on multiple lines. Use (@”) before the lines of text and (@”) after the text. Make sure that each symbol is on its own line and is not mixed with the text.

|  |  |
| --- | --- |
| Correct | Incorrect |
| $heretext = @”Some text hereMore text hereBlank line above“@ | $heretext = @”Some text hereMore text hereBlank line above“@ |

## String Interpolation

PowerShell can replace variables with their values when printing out strings.

To display the name of the variable instead of its value, add a backtick before the name of the variable.

Example: *“The value of `$hi is $hi”* will result in *The value of $hi is Hello World*

## String Formatting

You can format strings just like you do in .NET using the String.Format method.

Example: *[string]::Format(“There are {0} items.”, $items)*

Or using the PowerShell shortcut

Example: *“There are {0} items.” –f $items*

## Wildcards

Here are some wildcards to be used with the –like and –match string comparisons:

|  |  |
| --- | --- |
| \* | Any number of any character |
| ? | Only one of any character |
| \*[a-z] | Any number of characters from a to z |
| \*[c-g] | Any number of characters from c to g |
| \*[1-9] | Any number of characters from 1 to 9 |
| \*[4-8] | Any number of characters from 4 to 8 |
| [4-8]{2} | Only 2 characters from 4 to 8 |
| [c-g]{3} | Only 3 characters from c to g |

# Arrays

To assign an array, simple list all the values separated by a comma.

Example: *$array = “value1”, “value2”*

Arrays are zero-based. To access the first value, use the index zero.

Example: *$array[0]*

To create an empty array, use the following syntax:

*$array = @()*

This syntax can also be used to create an array:

*$array = @(“value1”, “value2”)*

You can create an array of numeric range with this shortcut:

*$array = 2..8*

To check if an item exists in an array, use the **–Contains <Value-To-Check-For>**

Example: *$array –contains “value3”*

# Hash Tables

Hash tables are the PowerShell equivalent of .NET dictionaries.

To create a hash table, use this syntax:

*$hashtable = @{“Key1” = “Value1”; “Key2” = “Value2”; “key3” = “value3”}*

To get a single value, use a syntax similar to getting a value from an array:

*$hashtable[“Key1”]*

Or you can use this: *$hashtable.”Key1”*

The value “Key1” can be replaced by a variable or expression which will be evaluated and replaced with the correct value before getting the key value from the hash table.

To remove a key from table, use **$<TableName>.Remove(“<KeyName>”)**

Example: *$hashtable.Remove(“Key1”)*

You can search in keys or values:

Example:

* *$hashtable.keys – contains “key1”* OR *$ hashtable.Contains(“key1”)*
* *$hashtable.values –contains “value1”* OR *$ hashtable.ContainsValue(“value1”)*

You can list all the keys using **$<TableName>.Keys**

You can list all the values using **$<TableName>.Values**

# Common Built-in Variables

|  |  |
| --- | --- |
| $true | True value |
| $false | False value |
| $pwd | Current directory |
| $home | User’s home directory |
| $host | Info about the user’s machine |
| $pid | Process ID |
| $PSVersionTable | Info about the current version of PowerShell |
| $\_ | Current object |

# Branching

## If Statement

If statements are very similar to those in .NET (C# to be specific) except it doesn’t support “Else if”. Here’s an example:

*If ($hi –eq “Hello”)*

*{*

*“It equals Hello”*

*}*

*Else*

*{*

 *If ($hi –eq “Hi”)*

 *{*

 *“It equals Hi”*

*}*

*Else*

*{*

*“It’s something else”*

*}*

*}*

## Switch Statement

Also simple:

*Switch ($hi)*

*{*

*“Hello” { “It’s Hello”; break }*

*“Hi” { “It’s hi”; break }*

*Default { “Something else” }*

*}*

Make sure to **break** to skip matching the next values in the list, otherwise it will continue down the list which is a waste of time.

# Looping

You can loop using multiple commands including the **While** command:

*$i = 1*

*While ($i – le 5)*

*{*

*“`$i = $i”*

*$i = $i + 1*

*}*

The **Do While** Command:

*Do*

*{*

*“`$i = $i”*

*$i++*

*} While ($i – le 5)*

The **Do Until** Command:

*Do*

*{*

*“`$i = $i”*

*$i++*

*} Until ($i – le 5)*

The **Do While** command works (goes through another loop) if the command is true. The **Do** **Until** command works (goes through another loop) if the command is false.

The **For** command works for a specific number of times:

*For ($f =0; $f –le 5; $f++)*

*{*

*“`$i = $i”*

*}*

The **For Each** command loops over the items in an array:

*Foreach ($item in $array)*

*{*

*“`item = $item”*

*}*

# Script Blocks

A script block is the code inside curly brackets.

To put multiple commands on a single line user the semi-colon to separate them.

Writing a script block on its own will not execute it, but will just print it out.

*{Clear-Host; “Hello World”}*

To execute a script block, you need to add “&” before the script block.

*& {Clear-Host; “Hello World”}*

You can store a script block in a variable

*$script = {Clear-Host; “Hello World”}*

Within a script block, you can use the **Return** command to stop the execution and exit the script block. Nothing after the **Return** command will be executed.

Script blocks can accept parameters using the **$args** array.

Script blocks can accept parameters using the **param** command to make a list of input parameters.

*$script = {*

*Param ($my1param, $my2param)*

*“Here are the two input parameters: $my1param and $my2param”*

*}*

When calling (executing) the script block, you can pass the parameters by order or by name:

*& $script –my1param “Parameter1” –my2param “Parameter2”*

Or you can just pass enough characters of the name to make it unique

*& $script –my1 “Parameter1” –my2 “Parameter2”*

A script block can use the **Process**, **Being**,and **End** commands.

The **Process** command forces the execution of a script block within the script block.

The **Begin** command executes a script block before any other **Process** commands.

The **End** command executes a script block after any other **Process** commands.

**Begin**, **Process**, and **End** commands are usually used with PowerShell Pipeline.

# Functions

A function is a script block with a name.

*Function Get-Total ($n1, $n2)*

*{*

*Write-Host ($n1 + $n2)*

*}*

Variables are passed to functions by value (ByVal in .NET). You can pass variables by reference using this syntax:

*Function Get-Total([ref] $n1, [ref] $n2)*

When calling this function, you must also use **[ref]** in the call.

*Get-Total([ref] $myN1, [ref] $myN2)*

Functions can have switches to control additional functionality. To enable switches, use the keyword **[switch]** before the function parameter just like you would use the **[ref]**.

To enable a switch when calling a function, add the name of the switch after a dash like **<FunctionName> -<SwitchName>**

# Comments

To add a comments block in PowerShell, start with **<#** and end with **#>**.

# Adding Help to Functions

To add help to PowerShell functions, start a comments block and add special words for the different help sections you want to support. Here are some of them:

|  |  |
| --- | --- |
| .SYNOPSIS | A brief description of the command |
| .DESCRIPTION | Detailed description |
| .PARAMETER name | Description of each parameter |
| .EXAMPLE | Detailed example of how to use the command |
| .INPUTS | What pipeline inputs are supported |
| .OUTPUTS | What this function outputs |
| .NOTES | Any extra notes |
| .LINK | A URL for more info |

You can use the command **Get-Help About\_Comment\_Based\_Help** for help on how to add help to functions.

# Error Handling

To catch errors in a function, use the keyword **Trap** at the end of the function and execute a script block that handles the error.

Example:

*Function FuncWithError()*

*{*

 *<# Do something here that might throw an error #>*

 *Trap*

 *{*

 *Write-Host “An error occurred”*

 *Write-Host $\_.ErrorID*

 *Write-Host $\_.Exception.Message*

 *Continue*

 *}*

*}*

**Continue** will to move the line after the line that caused the error.

**Break** will exit the function in case an error occurred after executing the error handling script block. **Break** will also throw the exception to the parent script block.

You can build **Trap** script blocks in the same way you build **Catch** statements in .NET where you can specify a certain exception you want to catch.

*Trap [System.StackOverflowException]*

# Working with Files

To get the content of a file, use the **Get-Content** command.

The content of a file can be stored in a variable. This variable would be of type Array where each line of the file is an element in the array.

To change the content of a file, use the **Set-Content** command.

*Set-Content –Value $ContentArray –Path FileName.txt*

To add content to an already existing file, use the **Add-Content** command.

*Add-Content –Value $ContentArray –Path FileName.txt*