

# PowerShell

---

## Purpose of this module

In this module, we're going to take a look at PowerShell. PowerShell is Microsoft's next-generation programmable shell, based on the .NET framework. It is mainly used by system administrators to perform all kinds of Windows management tasks, but you can also use it to quickly automate actions. After this module, the student should be able to:

- understand what cmdlets are and how they work
- work with the Integrated Scripting Environment
- write PowerShell scripts
- access the Windows management instrumentation from within PowerShell
- access the .NET framework from within PowerShell

## Module Requirements

You must have completed the introductory module before you can start working on this module. The latest version of PowerShell must be installed.

## Homework BEFORE the exercise session

Before you can start with this module, you will need to install PowerShell. There is a very good chance that some version of PowerShell is already installed on your computer (PowerShell is in fact integrated into Windows, starting from Windows 7), and all versions up till now are compatible. However, newer versions of PowerShell have much improved developer tools, so make sure you have installed the latest version. This assignment is based on PowerShell 4<sup>1</sup>.

This assignment describes what is expected of you and gives some pointers to extra information and an occasional helpful tip. For the most part, however, you are expected to look up information online if you're stuck. Make sure you go through the preparatory material that has been made available for this module.

## Microsoft PowerShell

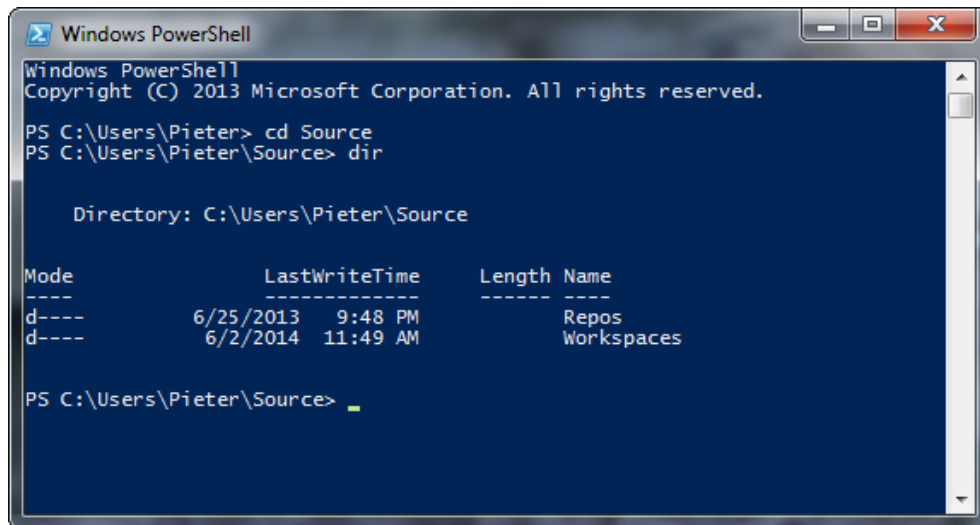
The traditional way to operate an operating system was through the command shell. It is a textual interface where users can type in commands. Newer shells improved upon older versions by implementing support for pipes and output redirection, and scripting (e.g. Bash on Linux, .bat files on Windows). However, one major hurdle in chaining shell commands together or writing shell scripts is the fact that everything is text-based, so many assumptions must be made about the output of specific

---

<sup>1</sup> <http://www.microsoft.com/en-us/download/details.aspx?id=40855>

commands in order to do anything other than the most basic tasks. If popular commands would suddenly change the way they output certain information, most scripts would break.

PowerShell is an improvement compared to other shells because it is the first shell whose commands output rich (.NET) objects instead of text. This gives the developer an unprecedented amount of control over the output, allows for rich scripting possibilities, and offers much better exception handling.

A screenshot of a Windows PowerShell window. The title bar says "Windows PowerShell". The window content shows the following text:

```
Windows PowerShell
Copyright (C) 2013 Microsoft Corporation. All rights reserved.

PS C:\Users\Pieter> cd Source
PS C:\Users\Pieter\Source> dir

    Directory: C:\Users\Pieter\Source

Mode                LastWriteTime         Length Name
----                -
d-----         6/25/2013   9:48 PM             Repos
d-----         6/2/2014  11:49 AM             Workspaces

PS C:\Users\Pieter\Source> _
```

## The Shell

Open a PowerShell window. Apart from the fact that the back color of the window is dark blue (instead of the black we are accustomed to), it pretty much looks like a normal shell window. Common commands like `cd`, `dir`, `ls`, `type`, `cat`, `cls`, `ren`, `md`, ... also work as expected. Note that these commands are case insensitive; `type`, `Type` and `TYPE` all execute the same command.

Create a new text file 'poem.txt' and add the following text to the file:

```
Poor old Dan was a bit of a schmuck
Ran out of gas in his brand new truck
Now late for his date
Boy, was she irate
No hanky panky was his bad luck
```

Use the `cd` command to navigate to the directory of the text file in the PowerShell window and use the `type` command to show the contents of the file:

```
type poem.txt
```

Unlike the `type` command in a DOS console window or the `cat` command of a Linux shell, the PowerShell version of `type` actually returns a .NET object that we can manipulate. In this case, it is an array of strings; each element in the array contains one line of the input file. Hence, we can call any method or property that is defined on an array of strings. Try the following (or other) examples, and investigate what is happening:

```
(type poem.txt).Length  
(type poem.txt)[0]  
(type poem.txt)[1].Replace("gas", "petrol")
```

## Cmdlets

PowerShell is built around the concept of *cmdlets* (pronounced *command-let*). A cmdlet is a small program that offers some kind of functionality, e.g. listing the contents of a directory, changing the current working directory, listing the running processes, etc. You can think of it as a .NET function with input parameters and an output result. An example is the **Get-Process** cmdlet that returns a list of running processes.

Cmdlets follow strict naming conventions<sup>2</sup>. It should be clear from the name what the cmdlet's purpose is. The names can typically be split into two parts separated by a hyphen. The first part is the verb. Its purpose is to clarify what type of action that will be executed by that cmdlet. Microsoft actually has a list of approved verbs<sup>3</sup> that can be used for cmdlet names, together with their implied meaning. The second part of the cmdlet's name is one or more nouns that describe the component that is being targeted by the cmdlet. Because of these guidelines, it is immediately apparent to the user that the **Get-Process** cmdlet returns one or more processes. Compare that to some frequently used Linux commands, such as **awk**, **cron**, **grep**, **less**, **ps**, ...

Even though these explicit naming conventions greatly enhance the user's understanding of what the commands do, it may sometimes be cumbersome to always type in these full command names, especially for very frequently used commands. That's why PowerShell also supports the concept of *aliases*. An alias is an alternative (typically shorter) name for a cmdlet. In fact, the commands that were mentioned in the previous section are all aliases for longer cmdlet names. In the rest of this assignment, we will always use the full cmdlet name.

| Cmdlet name   | Alias(es)     |
|---------------|---------------|
| Set-Location  | cd, chdir, sl |
| Get-ChildItem | dir, ls, gci  |
| Get-Content   | type, cat, gc |
| Clear-Host    | cls           |
| Rename-Item   | ren, rni      |
| New-Item      | md            |
| Get-Help      | help          |

## Getting Help

Getting to know PowerShell can be an intimidating task. Fortunately, it sports an excellent help cmdlet that you can use to get elaborate information on any cmdlet in the system. The **Get-Help** cmdlet takes as a parameter the name of the cmdlet (or its alias) you want to get help for. Try to get the help files of some of the cmdlets mentioned in the previous section.

---

<sup>2</sup> [http://msdn.microsoft.com/en-us/library/dd878270\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/dd878270(v=vs.85).aspx)

<sup>3</sup> [http://msdn.microsoft.com/en-us/library/ms714428\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/ms714428(v=vs.85).aspx)

If you want more detailed information about the command, you can add the **-Detailed** switch as a parameter to the call to **Get-Help**. Adding the **-Examples** switch lists a number of example with the corresponding descriptions of the cmdlet in question.

By default, PowerShell doesn't ship with a lot of help files. Since we are going to write PowerShell scripts, it might be a good idea to update the help database and store as much help locally as possible. The **Update-Help** cmdlet downloads this extra information from MSDN and stores it in the help database. However, running this cmdlet will result in an exception because it requires Administrator privileges. Run PowerShell as an Administrator, and call the **Update-Help** cmdlet. The update process should complete successfully.

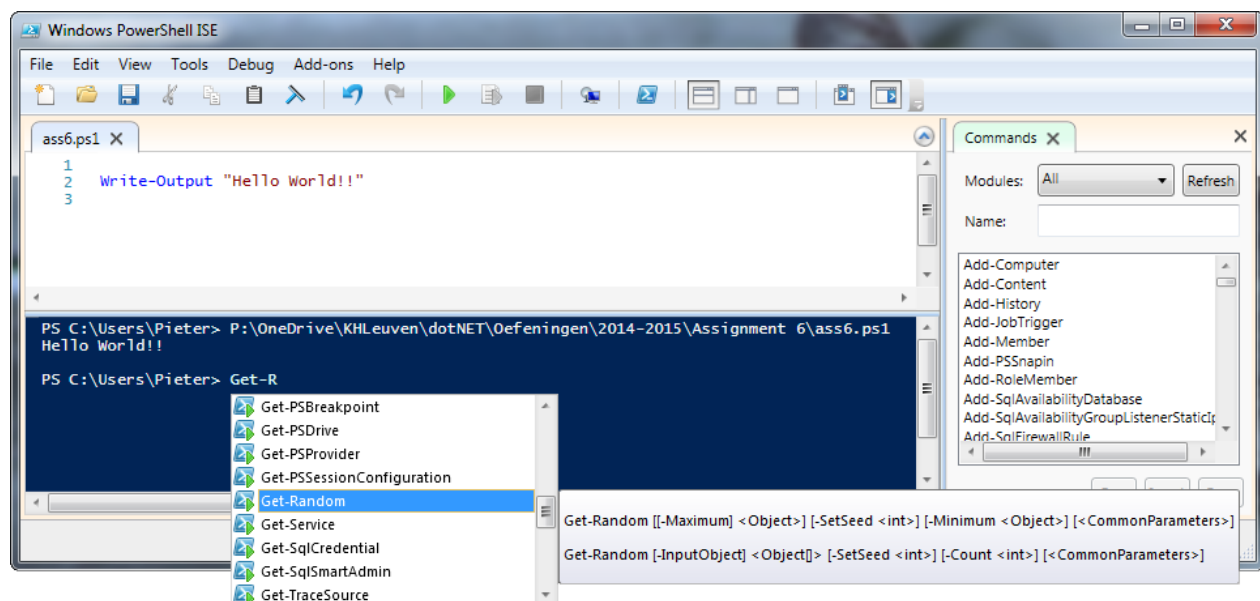
## PowerShell Integrated Scripting Environment

In addition to the help system, PowerShell also has an IDE to write scripts, called PowerShell ISE. The ISE main window is split in three sections. On the top-left, you see the script window. This is essentially a color coded text editor where you can open and edit your PowerShell scripts. On the bottom-left, you see an integrated shell. When you run your scripts, they will be executed in this shell. Finally, on the right you see the commands window that lists all the available PowerShell commands. Clicking a command will open a wizard-like panel that shows all the possible parameters of the cmdlet.

Create a short PowerShell script (simply executing a single cmdlet is enough) and try to run it by clicking on the green play icon in the toolbar. You might get an error stating that "running scripts is disabled on this system". By default, PowerShell disables running scripts that are not digitally signed. We can disable this behavior by executing the following command:

**Set-ExecutionPolicy RemoteSigned**

This command requires Administrator privileges in order to complete successfully. Hence, make sure you open a separate PowerShell window and run it as an Administrator.



## Piping and Output Redirection

PowerShell cmdlets are simple by design: they are designed to only do one specific task. This also implies that a cmdlet by itself is not very useful. The real strength of PowerShell (and indeed *any* shell) is that cmdlets can easily be combined. PowerShell uses the same syntax that we are used to from the Linux or Windows console.

The output of one command can be sent as input to another command by using the pipe ('|') character. On the left-hand side of the pipe is the cmdlet that is executed first. Its output is then *pip*ed as input to the cmdlet on the right-hand side of the pipe.

```
Get-ChildItem | Select-Object -First 3
```

The **Get-ChildItem** cmdlet is executed first, which returns a list of files and subdirectories in the current working directory. These results are then passed as input to the **Select-Object** cmdlet. **Select-Object** will select the first three items that it receives as input, and ignore all the rest. Execute the command above, and verify that the shell only shows (at most) three items.

Examine the command below, and try to figure out what it does and how it works. Use the **Get-Help** cmdlet to find the appropriate documentation.

```
Get-Process | Where-Object Name -NE "Idle" | Group-Object -Property Name | Sort-Object -Property Count -Descending | Select-Object -Property Count, Name -First 5
```

PowerShell also supports output redirection. Instead of showing the output of a command in the console, it can be redirected to a file by using an angle bracket ('>'). The following example shows how to write the list of processes to a file, and then show the contents of that file.

```
Get-Process > output.txt  
Get-Content output.txt
```

## Variables

It is sometimes useful to store intermediate outputs in variables. PowerShell supports the notion of variables in a similar way that they are supported in JavaScript. Variables can be defined implicitly, simply by using them. If the variable doesn't exist yet, it is created. They have a name that always starts with a dollar sign ('\$') and are (by default) untyped, meaning that you can store any data type in it. In the example below, a variable `$test` is created by assigning a string to it. Afterwards, an integer is assigned to it. Notice that the type of the variable actually changes from String to Int32.

```

PS:>$test = "Hello"
PS:>$test.GetType()

IsPublic IsSerial Name                                     BaseType
-----
True     True     String                                     System.Object

PS:>$test = 5
PS:>$test.GetType()

IsPublic IsSerial Name                                     BaseType
-----
True     True     Int32                                     System.ValueType

```

Variables can also be typed by specifying a data type when you declare them. The data type can be any class defined in the .NET framework (or in one of your own assemblies). If the variable is assigned to, PowerShell checks whether the data can be converted to the specified data type. If the data *is* of the correct type, no conversion is required of course. If the data is of a different type but can be converted, PowerShell automatically converts it. If the data type is of a different type and it cannot be converted to the correct type, an exception is thrown.

```

PS C:\Users\Pieter> [int] $test = 5
PS C:\Users\Pieter> $test
5

PS C:\Users\Pieter> $test = "17"
PS C:\Users\Pieter> $test
17

PS C:\Users\Pieter> $test = "bla"
Cannot convert value "bla" to type "System.Int32". Error: "Input string was not in a correct format."
At line:1 char:1
+ $test = "bla"
+ ~~~~~
+ CategoryInfo          : MetadataError: (:) [],
ArgumentTransformationMetadataException
+ FullyQualifiedErrorId : RuntimeException

```

As an alternative to setting variables by using the *equals* symbol, you can also set their value by calling the **Set-Variable** cmdlet.

```
Set-Variable -Name test -Value "bla"
```

## The Environment

PowerShell also has the concept of *environment variables*. These (string) variables are globally accessible and hold shell-instance-specific data. A number of these variables are initialized by default, but you can also add, modify and remove items. Editing a variable in one PowerShell instance will have no effect on another PowerShell instance. Every variable has a *case-sensitive* name that typically describes their content.

PowerShell has made it easy to access or modify environment variables by introducing the special global variable `$env`. Specific environment variables can be retrieved or edited by calling this `$env` variable and appending the name of the environment variable (separated by a colon).<sup>4</sup>

```
PS C:\Users\Pieter> $env:PROCESSOR_ARCHITECTURE
AMD64

PS C:\Users\Pieter> $env:USERNAME
Pieter
```

Windows defines a number of these variables by default. The next table lists the environment variables that should always be available (unless they have been specifically deleted in the shell instance).

| Case-sensitive variable name | Typical value  |
|------------------------------|--|
| ALLUSERSPROFILE              | C:\ProgramData   |
| APPDATA                      | C:\Users\username\AppData\Roaming  |
| CommonProgramFiles           | C:\Program Files\Common Files  |
| COMPUTERNAME                 | computername   |
| ComSpec                      | C:\WINDOWS\system32\cmd.exe  |
| HOMEDRIVE                    | C:   |
| HOMEPATH                     | \Users\username  |
| LOCALAPPDATA                 | C:\Users\username\AppData\Local  |
| LOGONSERVER                  | (Varies)   |
| NUMBER_OF_PROCESSORS         | (Varies)   |
| OS                           | Windows_NT   |
| Path                         | C:\WINDOWS\system32;C:\WINDOWS;<br>C:\WINDOWS\System32\Wbem;<br>C:\Windows\System32\WindowsPowerShell\v1.0 |
| PATHEXT                      | .COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;.MSC  |
| PROCESSOR_ARCHITECTURE       | (Varies)   |
| PROCESSOR_IDENTIFIER         | (Varies)   |
| PROCESSOR_LEVEL              | (Varies)   |
| PROCESSOR_REVISION           | (Varies)   |
| ProgramData                  | C:\ProgramData   |
| ProgramFiles                 | C:\Program Files   |
| PUBLIC                       | C:\Users\Public  |
| SESSIONNAME                  | (Varies)   |
| SystemDrive                  | C:   |
| SystemRoot                   | C:\WINDOWS   |
| TEMP                         | C:\users\username\AppData\Local\Temp   |
| TMP                          | C:\users\username\AppData\Local\Temp   |
| USERPROFILE                  | C:\Users\username  |

<sup>4</sup> In fact, PowerShell provides an `env:` provider that works much like the `c:` provider you use to access the hard disk. Because `env:` and `c:` are based on the same principles, the same commands apply to both. For example, you can delete environment variables in the same way you can delete files (`Remove-Item env:USERNAME`), you can list all variables like you list the contents of a directory (`Get-ChildItem env:`), and you can get the contents of a file like you get the contents of a variable (`{c:\myfile.txt}`).

## String expansion

PowerShell has a powerful string expansion mechanism built into it. When you define a string, PowerShell scans the string for variable names. If it encounters a variable name, the name is replaced with the actual contents of the variable (at run-time). For example, if we wish to create a personalized welcome message that uses the username, we could write something like this:

```
PS C:\>write-Host "Hello $env:USERNAME, welcome!"  
Hello Pieter, welcome!
```

However, sometimes we do not want to use this string expansion mechanism. All we have to do is to indicate that we wish the literal string value instead of the expanded version by using single quotes instead of double quotes.

```
PS C:\>write-Host 'The $env:USERNAME variable stores the username'  
The $env:USERNAME variable stores the username
```

We can also embed expressions in a string. An expression can be a calculation or a call to a cmdlet.

```
PS C:\>write-Host "There are $((Get-ChildItem).Length) files/subfolders in this  
folder."  
There are 9 files/subfolders in this folder.
```

Another type of string expansion is the support for special characters. In C#, we can encode special characters in strings by using a backslash (e.g. `\t` is a tab, `\r\n` is an enter, ...). PowerShell supports a very similar mechanism, but the backslash is replaced by the grave accent symbol (```).

```
PS C:\Users\Pieter>write-Output "a`r`nb"  
a  
b
```

## Short Exercises

Write a one-liner<sup>5</sup> for the following tasks:

- List all the files in the user's home directory and show only the name and the length.
- List all the files from the Windows directory that are larger than 10Kb. The result should be sorted by Length and saved in a file called "*username-largefilestxt*", where *username* should be replaced by the user's name. The windows directory should *not* be hardcoded (i.e. installations where Windows is not installed in `c:\windows` should also be supported).
- Recursively search through all text files in the user's home directory and check whether they contain the user's name. Display a list of (unique) filenames of the files that contain the given string.

---

<sup>5</sup> A *one-liner* is a command that fits on one line (without an enter)



Some useful cmdlets are: **Format-Table**, **Get-ChildItem**, **Get-Member**, **Select-String**, **Select-Object**, **Sort-Object**, **Where-Object**

## Writing Scripts

When using PowerShell, we do not have to write everything in one-liners, of course. We can abstract complex functionality away into functions, and then call these user-defined functions. These functions can be saved into script files.

The top-left window of PowerShell ISE is used to edit scripts. Use the **Write-Output** cmdlet to write a short 'hello world'-type program. Save the script file and run it by clicking on the green start button. The script will be executed in the shell window at the bottom of the screen.

We can create a function in a similar way as we are used to in JavaScript. The code listing below shows the definitions of three functions, each of them with two parameters. The second function shows how to specify the parameter's data type and the third function shows how to initialize the parameters to a default value (if the user didn't specify a value).

```
function Foo($a, $b) { ... }
function Bar([string] $a, [int] $b) { ... }
function Baz([string] $a = "", [int] $b = 1) { ... }
```

A function can contain a number of commands that are executed sequentially. The commands are placed on separate lines; no semi-columns are necessary to mark the end of a line. All output that is generated is returned from the function. The return keyword is present, but optional. It marks a logical exit point of the function, but is not necessary in order to actually return data from the function. The listing below shows two variations of the same function. The first uses the return keyword; the second simply outputs the value 1 (which is then captured as the return value).

```
function ReturnOne() {
    return 1
}

function ReturnOne() {
    1
}
```

A function can return more than one item. The following listing shows a function that returns multiple items.

```
function ReturnMany() {
    1
    2
    3
    4
}
```

```
PS C:\Users\Pieter>(ReturnMany).Length
4
PS C:\Users\Pieter>(ReturnMany)[1]
2
```

## Conditional Logic

Being able to easily write conditional logic is an important property of a programming language. To this end, PowerShell defines a number of operators to do just that. The following tables give an overview of all the operators that are available.

| Comparison operator | Description                   |
|---------------------|-------------------------------|
| -eq                 | Equals to                     |
| -ceq                | Equals to (case sensitive)    |
| -ne                 | Not equal to                  |
| -gt                 | Greater than                  |
| -ge                 | Greater than or Equal to      |
| -lt                 | Less than                     |
| -le                 | Less than or Equal to         |
| -like               | Wildcard comparison           |
| -notlike            | Wildcard comparison           |
| -match              | Regular expression comparison |
| -notmatch           | Regular expression comparison |
| -replace            | Replace operator              |
| -contains           | Containment operator          |
| -notcontains        | Containment operator          |

| Logical operator | Description |
|------------------|-------------|
| -not             | Not         |
| !                | Not         |
| -and             | And         |
| -or              | Or          |

| Bitwise operator | Description |
|------------------|-------------|
| -band            | Bitwise and |
| -bor             | Bitwise or  |

| Type operator | Description                             |
|---------------|---|
| -is           | Is of a type                            |
| -isnot        | Is not of a type                        |
| -as           | As a type, no error if conversion fails |

You would use these operators like you would use C#'s `&&`, `//` and `!` operators. Here are some examples:

```
$myVar -is "String"
$myVar -eq 123
$myVar -ceq $myVar2
"abcdef" -like "abc*"
```

In addition to these operators, PowerShell has two special variables **\$True** and **\$False** that store—as you probably already expect—the values True and False.

PowerShell's if statement works much like what you are used to in C#. Notice the lack of a space in the *elseif* statement.<sup>6</sup>

```
[int]$age = Read-Host "Enter your age"
if ($age -lt 16) {
    Write-Host "You are not allowed to drink alcohol."
} elseif ($age -lt 18) {
    Write-Host "You are not allowed to drink liquor."
} else {
    Write-Host "You can drink anything."
}
```

## Loops

PowerShell has a number of different loops, many of which you are already familiar with. The most commonly used loops are the For and ForEach loop. In addition, PowerShell also has support for the While, the Do..While and the Do..Until loops.

**ForEach:** the simplest and most useful loop. It performs an operation for each input object.

```
$array = 1,2,3,4,5
ForEach ($value in $array) { Write-Host $value }
```

**For:** a standard loop to execute a specific number of times.

```
For ($i=1; $i -lt 6; $i++) { Write-Host $i }
```

**While:** performs an operation as long as a condition is true.

```
$i = 1
while($i -lt 6) { Write-Host $i; $i++ }
```

**Do..While:** executes as long as a condition is true, but it always executes one iteration of the loop first before checking the condition

```
$i = 1
Do { Write-Host $i; $i++ }
while ($i -lt 6)
```

**Do..Until:** similar to the Do..While, but it executes as long as a condition is false.

---

<sup>6</sup> This example also shows why it is important to always specify the type of your variables if you know the type beforehand. If the \$age variable is not defined as an int, the example would be wrong. Input values like 2, 3, ..., 9 would give wrong results. The reason is that the comparison operators will convert the values 16 and 18 to a string and then compare the produced string with the input string. Try it and see if you can get a wrong result...

```
$i = 1
Do { Write-Host $i; $i++ }
Until ($i -gt 5)
```

Execute the code above and verify that these loops work the way you think they work.

## The User Profile

Whenever a PowerShell instance is started, it automatically executes a pre-defined script (if it exists). The location of this script is stored in the built-in global variable `$Profile`. Create the file (if it does not exist yet) and open it in notepad by executing the following commands:

```
if (!(Test-Path $profile)) { New-Item -Path $profile -Type file }
notepad $profile
```

Use the `Set-Location` cmdlet to change the startup directory of PowerShell. Open a new PowerShell window to verify that the initial location has indeed been changed.

One fun feature that we can adjust is *the prompt*. This is the text that appears in the console window before every command. In the previous examples, the prompt included the current working directory (e.g. the prompt was 'PS C:\Users\Pieter>') but it can be anything you like. You can overwrite the default prompt by defining a function named *prompt* in the profile script.

Create a function 'prompt' in the profile script that returns a string of the following format:

```
[time] location>
```

Obviously, *time* should be replaced with the current time and *location* should be the location of the current working directory. You can use the `Get-Date` and `Get-Location` cmdlets to get the required information.

## Windows Management Instrumentation

A major feature of PowerShell is that it can easily access the Windows Management Instrumentation (WMI) engine. WMI is a core Windows management technology; you can use WMI to manage both local and remote computers. WMI provides a consistent approach to carrying out day-to-day management tasks with programming or scripting languages.

We will not go into the details of WMI because that would lead us too far. However, you should know that you can access and manage a wealth of information from the WMI framework. In this assignment, we will focus on the Win32 WMI provider (which is one of the many providers in WMI that you can get information from).

WMI exposes information as a set of objects. Each object has a number of properties that describe the object and methods that can be used to perform actions on the object. For example, you could have an object that represents the logged on user's account. It exposes properties like *Name*, *Fullname*, *Domain*, ... and methods like *Rename*. Each of these objects has a specific name; the list of names and the

returns the requested object. The example below shows how to retrieve information about the computer's BIOS.

```
[11:02:12] C:\Users\Pieter>Get-WmiObject win32_BIOS  
SMBIOSBIOSVersion : V2.0  
Manufacturer      : American Megatrends Inc.  
Name              : Default System BIOS  
SerialNumber      : To Be Filled By O.E.M.  
Version           : 7597MS - 20091231
```

Note that by default this object shows the values of only 5 properties, but there are many more in the object. To get a list of all properties, pipe the returned object to the **Format-List** cmdlet and tell it to show all properties.

```
Get-WmiObject Win32_BIOS | Format-List -Property *
```

Use the **Get-WmiObject** cmdlet to retrieve information about other system devices (e.g. the sound card, video controller, the battery, ...).

## Accessing the .NET Framework

All cmdlets return (lists of) .NET objects. However, it is also possible to access .NET classes and objects without calling a cmdlet. We've already seen that we can specify the data type of variables by prepending them with the data type between square brackets (e.g. `[int]$myVariable`). The same brackets can also be used to access the static methods of that class by using the double column operator.

```
[11:21:48] C:\Users\Pieter>[DateTime]::Now  
Tuesday, July 01, 2014 11:21:52 AM
```

Obviously, we do not only want to access the static methods of .NET classes. The **New-Object** cmdlet allows us to instantiate classes, on which we can then call instance methods.

```
$wc = New-Object System.Net.WebClient  
$wc.DownloadFile("http://www.khleuven.be", "$env:HOMEPATH\kh1.html")
```

If the constructor has arguments, we can specify these arguments with the **ArgumentList** switch. The next example shows how we can initialize a **DriveInfo** instance that expects a string with the drive in question as a constructor parameter.

```
$hd = New-Object System.IO.DriveInfo -ArgumentList "$env:HOMEDRIVE"  
$hd
```

If you wish to access your own custom classes (that are not part of the .NET standard library), you can reference your assembly by calling the **Add-Type** cmdlet with the appropriate parameters. From then on, your classes are available in PowerShell.

## Longer Exercises

Write a function for the following tasks.

- Search all the \*.log files in the Windows directory (and its subdirectories) that contain the word 'Error'. Show for every log file how many times the word 'Error' occurs. If one of the cmdlets encounters an error (e.g. no access permissions for a file or directory), the error should *not* be shown.
- The file *trojans.csv* contains a (short) list of known Trojan processes. Every line contains the name of a process, a comma, and a description of the Trojan. Write a script that reads the file, and looks for matching processes in the list of running processes. If a matching process is found, a message should be shown to the user.

```
[15:35:12] Trojan_script.ps1
Potential Trojan found: explorer
Added by the BIFROSE-DE TROJAN! Note - the legitimate windows Explorer (same
filename) is located in %windir% and would not normally appear in Msconfig/Startup
unless you added it manually! This one is located in %System%

Potential Trojan found: lsass
Added by the BANCOS.V TROJAN!

Potential Trojan found: svchost
Added by the BANCOS.X TROJAN!

Potential Trojan found: WINLOGON
"Added by the DELF-LP TROJAN! Note - this malware actually changes the value data
of the ""(default)"" key in HKCU\Policies\Explorer\Run in order to force windows to
launch it at boot. The name field in MSConfig may be blank"
```

Some useful cmdlets are: **Get-ChildItem**, **Get-Content**, **Get-Process**, **Group-Object**, **Select-String**, **Sort-Object**, **Where-Object**, **Write-Host**