# Windows PowerShell 3.0: Step by step

Ed wilson

## Overzicht hoofdstukken

Hoofdstuk 1: Overview of Windows PowerShell 3.0

Hoofdstuk 2: Using Windows PowerShell Cmdlets

Hoofdstuk 3: Understanding and Using PowerShell Providers

Hoofdstuk 4: Using PowerShell Remoting and Jobs

Hoofdstuk 5: Using PowerShell Scripts

Hoofdstuk 6: Working with Functions

Hoofdstuk 7: Creating Advanced functions and Modules

Hoofdstuk 8: Using the Windows PowerShell ISE

Hoofdstuk 9: Working with Windows PowerShell Profiles

Hoofdstuk 18: Debugging Scripts

Hoofdstuk 19: Handling Errors

Hoofdstuk 20: Managing Exchange Server

### Inleiding:

Dit boek is geschreven door Ed Wilson. Het is de bedoeling van dit boek dat je een goede kennis opdoet van Windows PowerShell 3.0 technologie. Als je nieuw bent voor Windows Powershell, dan wordt er van je verwacht dat je je voornamelijk concentreert op de hoofdstukken 1 tot en met 3 en 5 tot en met 9. Aangezien wij beginnen vanaf nul, gaan wij deze volgorde nemen. Wij zullen de eerste 9 hoofdstukken, met uitzondering van hoofdstuk 4, doornemen en goed bijhouden wat belangrijk is voor dat hoofdstuk. Op deze manier krijgen we een bondige samenvatting voor beginnend powershellgebruik.

### Hoofdstuk 1: “Overview of Windows PowerShell 3.0”

Windows Powershell kan als vervanging gebruikt worden voor de CMD (command) shell. Zo kan je bijvoorbeeld gebruik maken van *cd* en *dir* in Windows PowerShell om te navigeren door je directorytree van je systeem. Belangrijk is dat je de structuur van cmdlets kent. Cmdlets, eigenlijk een andere naam voor commando’s, bestaan telkens uit \*\* een werkwoord\*\* gevolgd door een \*\*streepje\*\* en \*\*een zelfstandig\*\* naamwoord (bv: `Get-Help,Set-Service`). Get: Geeft informatie , Set: Veranderd informatie.

Het is mogelijk om meerdere commando’s tegelijk te gebruiken. Bv:

`> ipconfig /all >tshoot.txt; route print >>tshoot.txt`

Dit commando zal de uitvoer van ipconfig en route print in de tshoot.txt-file zetten.

Een van de belangrijkste updates van powershell, is het gebruik van `–whatif` en `–confirm`. Dit zijn argumenten die op het einde van een commando kunnen ingevoegd worden, zodanig dat vooraleer je een commando uitvoert, je eerst de mogelijk uitvoer krijgt. Op deze manier kom je niet voor verrassingen te staan, en zal je op voorhand weten wat het uitvoerende commando zal doen. Bij confirm zal je een confirmatie-vraag krijgen . vb:

1. Open Notepad door in de powershell `notepad` te typen.

2. Kijk wat het procesID is door gebruik te maken van het commando `Get-Process note\*`

Gebruik maken van een \* is een groot voordeel in powershell. Dit kan je beschouwen als een wildcard. Stel dat je een lijst wilt van alle get-commando’s die beginnen met een H, dan kan je get-H\* typen, en krijg je als resultaat alle mogelijk get commando’s die beginnen met een H.

3. Gebruik het –confirm/-whatif argument bij het stoppen van het proces van notepad:

`> Stop-Process –id 1768 –confirm`

Het processID van notepad is in ons geval 1768

4. Als je dit commando uitvoer zal je de vraag krijgen of je zeker bent dat je deze actie wilt uitvoeren. Dan moet de gebruiker vervolgens bevestigen of annuleren.

#### De helpfunctie

Vooraleer je start met het gebruik van windows powershell is het belan0grijk dat je het volgende commando uitvoert:

`> Update-Help \_module \* -Force`

Dit zorgt er voor dat je de laatste nieuwe versie hebt van de help-functie. De helpfunctie word enorm veel gebruikt. Daarom is het van belang dat je de laatste nieuwe info hebt.

Ik zou je nu kunnen uitleggen hoe de help-functie werkt, maar al doende leert men beter. Daarom type je het commando: `> Get-Help Get-Help.` Dit zal je een uitvoerige beschrijving geven over hoe de Get-help functie werkt. Dat is hoe de help-functie werkt. Je typt het commando `Get-Help` gevolgt door het commando waarover je hulp wilt krijgen. Stel dat je niet meer weet hoe het commando precies gaat, maar je weet dat het commando begint met een ‘p’, dan geef je het commando `Get-Help get-p\*`. Stel dat je wel weet hoe het commando geschreven wordt, maar je weet niet meer hoe de syntax daarvan uitziet, dan kan je gebruik maken van `Get-Help` met het `–examples` argument. (bv` Get-Help Get-PSDrive –examples`)

De helpfunctie heeft drie verschillende weergaven, een normale, een gedetailleerde en een volledige weergave. Dit zijn argumenten die op het einde van het commando moeten bijgevoegd worden. Om het verschil te weten tussen deze 3 raad ik u aan deze commando’s uit te proberen en te vergelijk.

```PowerShell

> Get-Help Get-Help –detailed

> Get-Help Get-Help –full

> Get-Help Get-Help

```

### Hoofdstuk 2: “Using Windows PowerShell Cmdlets”

Commando’s in powershell zijn niet hoofdletter gevoelig. Dit wil zeggen dat bijvoorbeeld Get-Childitem dezelfde uitvoer heeft als get-Childitem of get-childitem.

De tabtoets is een heel handige toets in het gebruik van PowerShell. Deze toets bied je de mogelijkheid om je commando automatisch te laten aanvullen. Zo kan je get-ch typen en vervolgs de tab-toets gebruiken om zo Get-ChildItem te krijgen. Stel dat er meerdere mogelijkheden zijn, dan kan je met de tab-toets al deze mogelijkheden overlopen.

Samen met de tabtoets, zijn de aliases een handige tool voor het gebruik van PowerShell commando’s. Een alias maakt het mogelijk om een commando in een verkorte versie te schrijven. Zo is het commando dir een verkorte versie van het commando Get-ChildItem. Een lijst van alle mogelijke aliases kan je krijgen met het commando Get-Alias sort. Het is ook mogelijk om zelf een alias te maken. Dit kan via het commando New-Alias VerkorteVersie VolledigCommando. (bv New-Alias gh Get-Help)

Dit hoofdstuk gaat vooral over het gebruik van commando’s. Daarom verwijs ik je graag door naar de cheatsheet van deze samenvatting. *(Windows PowerShell 3.0 Step by Step cheatsheet.xlsx)*

### Hoofdstuk 3:”Understanding and Using PowerShell Providers”

#### Wat zijn providers?

Belangrijk is dat je begrijp wat providers zijn. Microsoft Windows Powershell heeft constant een externe manier om aan informatie te geraken. Dit door gebruik te maken van providers. Dit zijn eigenlijk .NET programma’s die “lelijke” details verbergen voor de gebruiker, om zo snel en gemakkelijk aan informatie te geraken. Om een lijst te krijgen van alle providers, kunnen we het commando Get-PSProvider gebruiker. Zo zie je bijvoorbeeld dat er een provider is genaamd Alias.

Mogelijke providers zijn:

- Alias: Deze alias provider zorgt ervoor dat je makkelijk te gebruiken toegang hebt tot alle aliases in PowerShell. Om op je machine te kunnen werken met aliases gebruik je Set-Location en specifieer je die met Alias:\ drive.

- Certificate: geeft u de mogelijkheid te zoeken, kopiëren, verplaatsen , verwijderen van certificaten.

- Omgeving: De omgeving -provider in Windows PowerShell wordt gebruikt om toegang tot de systeem omgevingsvariabelen te voorzien . Als je een CMD ( commando) shell opent, en vervolgs typ je set, dan ziet u een lijst van alle te verkrijgen omgevingsvariabelen gedefinieerd op het systeem.

- Bestandsysteem: Deze provider is de gemakkelijkste om te begrijpen. De provider bied toegang tot het bestandsysteem van de computer. Wanneer Windows PowerShell gestart wordt, opent het automatisch in de gebruikers documentenfolder. Met deze provider kan je bestanden en documenten creëren, verwijderen, wijzigen,…

- Functie: deze provider bied u toegang tot de functies die gedefinieerd zijn binnen PowerShell. Hiermee kan je een lijst krijgen van alle functies op je systeem. Je kan ook functies toevoegen, verwijderen en wijzigen. Deze provider is gebaseerd op een bestandsysteem model.

- variabele: Net zoals bovenstaande providers, biedt deze provider toegang tot de variabele binnen Windows PowerShell. Zowel variabele gedefinieerd door gebruikers, als door het systeem. Je kan een lijst krijgen van commando’s die speciaal gemaakt zijn om te werken met variabelen. Dit gaat via de Get-Help \*variabele. Stel dat je enkel commando’s terug wilt dan moet je | Where-Object category –eq “cmdlet” hieraan toevoegen.

### Hoofdstuk 4: “Using PowerShell Remoting and Jobs”

Remoting is een van de grootste verbeteringen in PowerShell 3.0. De configuratie is gemakkelijker dan de vorige versie. Remoting in Windows PowerShell maakt meestal gebruik van de protocollen \*\*Distributed component object model (DCOM)\*\* en \*\*Remote procedure call (RPC)\*\* om verbinding te maken met de remote-machines. Het volgende commando geeft een lijst van commando’s die native remoting ondersteunen:

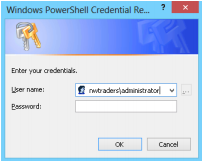
```PowerShell

get-help \* -Parameter computername | sort name | ft name, synopsis -auto –wrap

```PowerShell

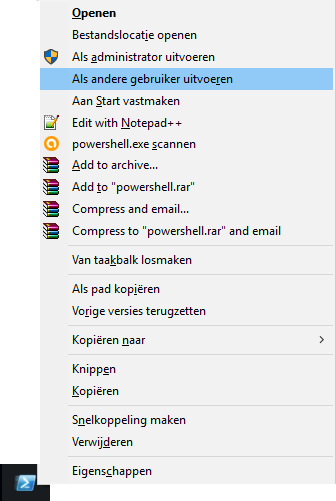
Als je dit commando uitvoert, dan zie je dat veel commando’s met de `–computername` parameter, gebruik maken van \*\*Web Service Management (WSMAN)\*\*, \*\*Common Information Model (CIM)\*\*, of \*\*Sessies\*\*.

Sommige commando’s hebben de eigenschap om \*\*credentials\*\* te specifiëren. Deze credentials maken het mogelijk om verschillende user accounts te gebruiken om zo een connectie te maken en data te ontvangen. Als je gebruik maakt van deze commando’s dan krijg je het volgende venster om je in te loggen op je accounts.



Jammer genoeg is het gebruik van deze commando’s niet altijd aan te raden. Deze commando’s vereisen specifieke services die gestart moeten worden, of openingen in de firewall. Daarom failen deze commando’s vaak als je deze gebruikt bij standaard remoting machines. Dit omdat deze machines geen ‘relaxed’ access rules hebbben.

Niet ieder commando heeft een ` –credential` parameter. Dit wil niet zeggen dat je het commando niet kan runnen via een andere gebruiker. Wanneer je op het Windows Powershell icoon rechts klikt, met Shift-key ingehouden, verschijnt er een venster waar je kan kiezen om te runnen als een andere gebruiker.



#### WinRM

Windows Remote Management (WinRM) is standaard geconfigureerd in Windows Server 2012 voor het ondersteunen van remote Windows PowerShell commando’s. Het is firewall-vriendelijk. Het maakt gebruik van CIM cmdlets. Vanaf je Windows Server 2012 up and running is, dan kan je gebruik maken van de functies van WinRM. Maar bij Windows 8 clients, staat deze functie nog uitgesteld. Om deze te activeren kan je gebruik maken van het volgende commando :

`Enable-PSRemoting –force`

Dit commando gaat voor jou de nodige services starten of herstarten. Alle vereisten worden voor jou correct geconfigureerd. Merk wel op dat deze functie het ping-commando op een windows 8 client systeem niet zal toestaan. Dit komt omdat het de remote management firewall exception niet toestaat. Pingen naar windows server 2012 zal correct verlopen, pingen naar een windows 8 client, niet.

Je kan ook manueel deze service opstarten. Dit doe je als volgt:

- Starten van de Service en automatisch starten bij opstarten:

```PowerShell

PS C:\> Set-Service -Name WinRM -ComputerName WC81-1 -StartupType Automatic -Status Running

```

=> creëert nog niet de nodige WinRM listeners for Remote system communication

- Deze listeners zijn eindpunt die remote systems toegang verlenen om te verbinden

- Deze creëren we via Modules

##### WS-Management Cmdlets

```PowerShell

PS C:\> Get-Command -Module Microsoft.WSMan.Management

CommandType Name Version Source

----------- ---- ------- ------

Cmdlet Connect-WSMan 3.0.0.0 Microsoft.WSMan.Management

Cmdlet Disable-WSManCredSSP 3.0.0.0 Microsoft.WSMan.Management

Cmdlet Disconnect-WSMan 3.0.0.0 Microsoft.WSMan.Management

Cmdlet Enable-WSManCredSSP 3.0.0.0 Microsoft.WSMan.Management

Cmdlet Get-WSManCredSSP 3.0.0.0 Microsoft.WSMan.Management

Cmdlet Get-WSManInstance 3.0.0.0 Microsoft.WSMan.Management

Cmdlet Invoke-WSManAction 3.0.0.0 Microsoft.WSMan.Management

Cmdlet New-WSManInstance 3.0.0.0 Microsoft.WSMan.Management

Cmdlet New-WSManSessionOption 3.0.0.0 Microsoft.WSMan.Management

Cmdlet Remove-WSManInstance 3.0.0.0 Microsoft.WSMan.Management

Cmdlet Set-WSManInstance 3.0.0.0 Microsoft.WSMan.Management

Cmdlet Set-WSManQuickConfig 3.0.0.0 Microsoft.WSMan.Management

Cmdlet Test-WSMan 3.0.0.0 Microsoft.WSMan.Management

```

- Nu kunnen we kijken of ons remote systeem WinRm geconfigureerd heeft door het te testen met het volgende commando: Test-WSMan –Computername <*computernaam>*

Als WinRM niet correct geconfigureerd is, zal het systeem een error teruggeven.

- We zien dat we WinRM Listeners moeten toestaan en zeker zijn dat WinRM klaar is om binnenkomende request te aanvaarden

#### CIM

### Remote Windows PowerShell session

Om een gemakkelijke configuratie op één enkele remote machine te gebruiken, maken we gebruik van een \*\*session\*\*. Om deze te gebruiken maken we gebruik van het commando:

` Enter-PSSession –ComputerName <computername>`

Je kan ook meerdere sessies openen door gebruik te maken van het commando:

`New-PSSession –ComputerName <computername>`

Om een lijst te krijgen van de sessions de open staan gebruik je het commando:

`Get-PSSession`

Om een sessie te verwijderen:

`Remove-PSSession`

#### Één enkel commando

Als je slechts 1 commando moet uitvoeren op een remote machine, dan moet je niet de hele procedure volgen om een sessie op te starten. Nee, je maakt gebruik van het commando:`Invoke-command`. Stel dat je het commando `-ScriptBlock {gps | select -Last 1}` wil uitvoeren op een remote machine genaamd \*\*ex1\*\*. Dan kan je dit doen door het volgende in te geven:

`Invoke-Command -ComputerName ex1 -ScriptBlock {gps | select -Last 1}`

Dit commando maakt het mogelijk om voor vele remote machines, hetzelfde commando in te geven. Je kan gebruik maken van een parameter, waar je alle namen ingeeft van de remote machines waar het commando moet ingegeven worden. Stel dat we bovenstaand commando terug willen invoegen, maar dit maal voor 5 computers genaamd: "dc1","dc3","ex1","sql1","wsus1".

$Computername: "dc1","dc3","ex1","sql1","wsus1"

Invoke-Command -ComputerName $Computername -ScriptBlock {gps | select -Last 1}

#### Windows PowerShell jobs

Jobs maken het mogelijk om een of meerdere commando’s te runnen in de achtergrond. Dit is heel handig als bepaalde commando’s veel tijd in beslag nemen. Op deze manier kan je toch verder werken zonder dat het proces van het commando onderbroken word. Stel dat je het commando -`ScriptBlock {get-process}` wil uitvoeren als job. Dan doe je dit als volgt:

`Start-Job -ScriptBlock {get-process}`

Als je de \*\*status\*\* van de job wilt nakijken, dan maak je gebruik van het commando `Get-Job –Name <name> of Get-Job –Id <id>`

Wanneer de status van de job om completed staat, dan kan je de `data ophalen` door gebruik te maken van het commando:

`Receive-Job –Name <name>`

\*\*Verwijderen\*\* van de job doe je met het commando

`Remove-Job –Name <name>`

### Using PowerShell Scripts

PowerShell scripts zijn \*\*ps1-bestanden\*\* waar vooraf gecodeerde instructies staan geschreven ,die we wensen uit te voeren. Taken die op \*\*routine\*\* basis moeten uitgevoerd worden, worden vaak in scripts gestopt. Op deze manier wordt er niet veel tijd verloren en kan men snel controleren of er zich geen fouten hebben voorgedaan. Wanneer men iets snel vanuit het niets wil opbouwen, kan men gebruik maken van scripts om de nodige instelling zo snel mogelijk te initialiseren. Het is ook een goeie manier om ‘geheimen’ te delen onder elkaar. Zo kan bijvoorbeeld een netwerkbeheerder een script maken die remote connect met een werkstation, waar gebruikers geen wachtwoord vereisen. Geen wachtwoorden voor een gebruiker is natuurlijk een beveiligingsrisico, daarom dat je dit niet wil delen met de buitenwereld. Dit script kan je wel doorsturen naar medebeheerders.

#### Scripting fundamentals

De meest basisvorm van een script bestaat uit enkele Windows PowerShell commands na elkaar. Bv

```PowerShell

Get-process –name Notepad | Stop-process`

```

Dit is een ‘script’ voor notepad af te sluiten na het uitvoeren hiervan. Je kan een script runnen door het bestand te verslepen naar de Windows PowerShell console. Je kan ook simpelweg dubbelklikken op het script-icoon. Als je in de console van Windows PowerShell zit te werken, kan je het script runnen door het volledige path naar het script te schrijven. Vergeet natuurlijk het ps1-extensie niet.

## ExecutionPolicy

Standaard, Windows PowerShell laat het uitvoeren van scripts niet toe. Script support kan worden gecontroleerd door gebruik te maken van group policy’s. Je kan ook gebruik maken van `Set-ExecutionPolicy` als je administratorrechten hebt. Er zijn \*\*6 opties\*\* die kunnen worden toegestaan met die commando:

- \*\*\*Restricted\*\*\*: Laad geen configuratie files, zoals profiles (Default).

- \*\*\*AllSigned\*\*\*: Vereist dat alle scripts en configuratie files gesigneerd zijn door lokale gebruiker of vertrouwde publishers.

- \*\*\*RemoteSigned: Vereist dat alle scripts en configuraties files die gedownload zijn van het internet, dat deze gesigneerd zijn door een vertrouwde publisher.

- \*\*\*UnRestricted\*\*\*: Laad alle configuratie files en runt ieder script. (vraagt wel toestemming als het een gedownload script is)

-\*\*\*Bypass\*\*\*: Blokkeert niet en geeft geen waarschuwingen, vraagt ook geen toestemming.

-\*\*\*Undefined\*\*\*: Verwijderd de huidige toegekende execution policy van de huidige scope.

Bij elke optie, is er de mogelijk voor de \*3 scopes\*:

- \*\*\*Process\*\*\*: Het execution policy zal hier enkel effect hebben op het huidige process.

- \*\*\*CurrentUser\*\*\*: Het execution policy zal hier enkel effect hebben op de huidige gebruiker.

-\*\*\* LocalMachine\*\*\*: Het execution policy zal hier effect hebben op iedere gebruiker of computer.

Er word aangeraden om gebruik te maken van \*\*RemoteSigned\*\*. Voor een lijst te krijgen van iedere policy voor elke scope maak je gebruik van het commando: `Get-ExecutionPolicy –List`

Een van de grote voordelen in het schrijven van een script, is dat je gebruik kan maken van variabelen. Op deze manier kan je makkelijk wijzigingen aanbrengen aan het script, zonder zelf in het script te moeten zoeken. Je veranderd gewoonweg de variabele. In het bovenstaande voorbeeld kan je Notepad bijvoorbeeld in een variabele `$Process` stoppen:

```PowerShell

$Process= “Notepad”

```

Ook kan je informatie halen uit een stop-proces. Standaard, wanneer je een stop-proces uitvoert, word er geen informatie teruggegeven. Door gebruik te maken van ` –passthrue` parameter bij het stop-proces, zal het proces object passeren langs de pipeline. Met het \*\*ForEach\*\*-object commando, kan je info halen uit een bepaald proces. Zo zal het volgende script:

```PowerShell

StopnotepadSilentlyContinuePassthru.ps1

$process = "notepad"

Get-Process-name $Process -erroraction silentlycontinue |

Stop-Process-passthru |

ForEach-Object{ $\_.name + ' with process ID: ' + $\_.ID + ' was stopped.'}

```

De volgende \*\*uitvoer\*\* hebben:

```PowerShell

notepad with process ID: 2088 was stopped.

notepad with process ID: 2568 was stopped.

```

#### Variabelen

Variabelen worden gebruikt in script voor informatie bij te houden voor later gebruik in een script. Variabelen kunnen \*\*elke type\*\* van data bijhouden. (tekst, nummers, objects,…)

Alle variabelenamen moeten beginnen het \*dollarteken $\*. Er zijn enkele voor gedefinieerde variabelen. Enkele belangrijke zijn:

- $^ : Deze variabele bevat het eerste token van de laatste input line in de shell.

- $$: Deze variabele bevat het laatste token van de laatste input line in de shell.

-$\_: Dit is het huidige pipeline object. Wordt vaak gebruikt in script blocks, filters, where-objects,…

-$?: Dit bevat de success/fail status van de laatste statement.

-$Args: Deze word gebruikt bij functies of scripts die parameters vereisen, maar waar geen *param* block is.

-$error: Slaat de laatste error op in deze variabele.

-…

#### Constanten

Constanten zijn gelijk variabelen met twee belangrijke uitzonderingen. De waarde veranderd nooit en kan ook niet verwijderd worden. Wanneer je een constanten wilt declareren, maak je gebruik van het commando Set-Variabele, waar je –option constant aan toevoegt om er een constante van te maken. Hoe dan ook in het script zelf wordt er niet verwezen naar de constante via Get-Constante, maar er wordt ook een dollarteken ($) voorgezet.

#### Scripting statements:

Enkele statements die men kan gebruiken in het schrijven van PowerShell Scripts zijn hieronder vernoemt. Basiskennis van programmeren (in java) zou voldoende moeten zijn om deze statements te begrijpen. Ik heb een klein voorbeeldje onder ieder statement gezet, de werking hiervan zal zo duidelijker worden

**- If**

$a = 5

If($a -eq 5)

{

'$a equals 5'

}

- **While**

$i = 0

$fileContents = Get-Content-path C:\fso\testfile.txt

While ( $i -le $fileContents.length )

{

$fileContents[$i]

$i++

}

Dit kleine scriptje met een while-lus gaat de content van een testfile.txt overlopen en deze in een variabele stoppen.

**- Do While**

$i = 0

$ary = 1..5 # Hier toon je aan dat je array uit 5 nummers zal bestaan.

do

{

$ary[$i]

$i++

} while ($i -lt 5)

Merk op dat we vergelijkingsoperatoren (zoals hierboven –lt) hier anders formuleren :

-lt betekend minder dan, -le betekend minder dan of gelijk aan.

-gt en –ge betekenen respectievelijke groter dan en groter of gelijk aan.

-eq en –nq betekenen respectievelijk gelijk aan en niet gelijk aan.

**- Do until**

$i = 0

$ary = 1..5

Do

{

$ary[$i]

$i ++

} Until ($i -eq 5)

**- For**

For($i = 0; $i -le 5; $i++)

{

'$i equals ' + $i

}

**- ForEach**

$ary = 1..5

Foreach ($i in $ary)

{

$i

}

In een ForEach binnen de {} kan ook een break/exit statement gebruikt worden gecombineerd in een if-lus. Voorbeeld:

$ary = 1..5

ForEach($i in $ary)

{

if($i -eq 3) { break/Exit } #Ofwel Break, ofwel Exit.

$i

}

"Statement following foreach loop"

Bij een break ga je enkel uit de ForEach-lus, bij een Exit ga je de verdere commando’s in het volledige script niet uitvoeren.

**-Switch**

$a = 2,3,5,1,77

Switch ($a)

{

1 { '$a = 1' ; break }

2 { '$a = 2' ; break }

3 { '$a = 3' ; break }

Default { 'unable to determine value of $a' }

}

"Statement after switch"

### Hoofdstuk 6: Working with Functions

De eerste stap om een functie te maken in Windows PowerShell is het vinden van \*\*een gepaste naam\*\*. De naam begint met het keywoord `Function` gevolgt door de \*\*naam\*\* van de functie.(combinatie werkwoord en zelfstandig naamwoord) . Een inputparameter is niet noodzakelijk bij het gebruik van functies. Maar kan natuurlijk altijd.

Een functie heeft het volgende patroon:

```PowerShell

Function Function-Naam

{

# Code hier invoegen

}

```

In het onderstaande voorbeeld gaan we een functie aanmaken die de systeemversie van een besturingssysteem teruggeeft. Deze functie kunnen we dan oproepen wanneer we willen met het $(naamfunctie). Voorbeeld:

```PowerShell

Function Get-OperatingSystemVersion

{

(Get-WmiObject –Class Win32\_OperatingSystem).Version

}

“This OS is version $(Get-OperatingSystemVersion)”

```

Zoals hierboven vermeld kan je gegevens meegeven als \*\*input\*\*. In het volgende voorbeeld geven we een path mee aan de functie, waar het bestand zich bevind dat moet gelezen worden, en waar we de content van vragen. Voorbeeld:

```PowerShell

$path = “C:\Test.txt”

Function Get-TextStatistics($path)

{

Get-Content –path $path

}

```

Functies worden vooral gebruikt om het gebruik van code te reduceren. Zo wordt de code performanter en compacter. Het maakt het ook makkelijker in het gebruik van scripts. Stel dat we een script hebben, waar twee of meerdere functies in gedefinieerd staan, dan kunnen we gebruik maken van deze functies door een `.` te zetten voor het script. In onderstaand voorbeeld hebben we een script `ConversionFunctions.ps1`, waar twee functies staan gedefinieerd die getallen convereteren naar meters of feet.

```PowerShell

ConversionFunctions.ps

Function Script:ConvertToMeters($feet)

{

"$feet feet equals $($feet\*.31) meters"

} #end ConvertToMeters

Function Script:ConvertToFeet($meters)

{

"$meters meters equals $($meters \* 3.28) feet"

} #end ConvertToFeet

```

Als we nu een getal willen converteren naar feet, dan moeten we eerst een punt schrijven voor het script waar de functie zich bevind. Daarna kunnen we de functie aanroepen.

```PowerShell

PS C:\> . C:\scripts\ConversionFunctions.ps1

PS C:\> convertToFeet 6

6 kilometers equals 19,68 feet

```

Het is ook mogelijk om een functie te schrijven waarvoor twee of meerdere inputparameters vereist zijn. Dit wordt op de volgende manier geschreven:

### Hoofdstuk 7: Creating Advanced functions and Modules

### Hoofdstuk 9: Working with Windows PowerShell Profiles