Netz-Weise
Lernen von den Besten.

Windows Powershell 3.0 Compressed

Von Holger Voges

www.Netz-Weise.de

Inhalt

Powershell-Pipeline

Wichtige Pipeline-Befehle

- Select-object
- Where-object
- Foreach-Object
- Group-Object
- Sort-Object
- Format-Table
- Format-List
- Export-csv
- Convertto-HTML

Variablen

Variablentypen

- String (wird mit „“ bzw. ‚‘ umschlossen)
- Here-String (alles zwischen **@“** und **„@** bzw. **@‘** und **@‘**wird komplett als Text behandelt
- Int
- Float
- Double
- Decimal
- Array
- Hash-Tabelle

Variablen in Powershell müssen nicht deklariert werden. Sie werden „nach Bedarf“ einfach angelegt. Variablen beginnen in Powershell immer mit einem $:

```
$Text = „Hallo Welt“
```

Variablen werden von Powershell automatisch einem Typ zugewiesen. Powershell nutzt hierzu die .net-Typen. Im obigen Beispiel wird die Variable automatisch dem Typ String zugewiesen.
Man kann einer Variablen einen festen Typ zuweisen, indem man den Variablentyp in eckigen Klammern vor den Variablennamen schreibt:

```
[String]$Text = 3 # konvertiert 3 automatisch in einen Text
```

Das Umwandeln von Variablen findet durch den Operator –as statt:

```
[string]$input –as [datetime] # -as wirft keine Fehler aus und konvertiert bei
datetime z.B. nach Culture
```

Arrays (Felder oder Sammlungen von Variablen) werden erzeugt, indem man mehrere Werte einfach mit Komma voneinander getrennt einer Variablen zuweist:

```
$Array = „Hans“,“Franz“,“Otto“,“Karl“
```

Statt Powershell-Arrays können auch Arraylisten aus dem .Net-Framework verwendet werden, die mehr Methoden haben und z.T. auch schneller sind

Erzeugen einer neuen Arrayliste und umwandeln eines Arrays in eine Arrayliste:

```
# neu anlegen
[system.collection.arrayList]$feldneu

# konvertieren in [system.collections.arrayList]
$feldneu = [system.collections.arrayList]$feld
```

Hash-Felder sind Wertepaar-Zuweisungen. Ein Hash-Feld-Eintrag besteht immer aus einem Namen und einem zugeordneten Wert.

Anlegen eines leeren Hash-Felder und Zuweisen eines Wertes:

```
$Haschfeld = @{}
$Haschfeld.add("Joint","Rauchen") #Einen Wert mit Namen Joint hinzufügen
$Haschfeld.add(„Keks“,“Essen“)
$Haschfeld                        #Das gesamte Feld ausgeben
$Haschfeld.Keks                   #Nur den Wert für das Feld Keks ausgeben
$Haschfeld[„keks“]                #Andere Variante, das Feld Keks auszugeben
```

Ein Hash-Feld anlegen und gleich Werte zuweisen – hier zur Nutzung für „Select-Object“. Der Wert Name gibt den Namen der auszugebenden Spalte an, Expression den auszuwertenden Ausdruck:

```
$spalte = @{ Name="Größe in KB"; Expression={ $_.Length/1KB }}
Get-childitem | Select-object Name,$Spalte
```

Variablentyp ermitteln

```
$text = "13"
$text.gettype().Fullname
```

Ermitteln, ob eine Variable vom Typ Array ist:

```
$Variable –is [Array]
```

Das klappt auch mit anderen Variablentypen:

```
$Variable –is [String]
$Variable –is [INT]
$Variable –is [Double]
```

Verkettete Typen-Umwandlung:

```
[char[]]"Guten Tag!" # Wandelt den Text in ein Array von [Char] um
[Byte[]][Char[]]"Hallo Welt" # Wandelt den Text in Char und anschliessend in den ASCII-Code um
$Alphabet = 65..90 | %{$_} # Ausgabe das Alphabets in Array-Form
```

Vergleichsoperatoren

-eq	= (gleich)
-neq	<> (ungleich)
-lt	< (kleiner als)
-gt	> (größer als)
-le	<= (kleiner oder gleich)
-ge	>= (größer oder gleich)
-like	Textvergleich, * steht für beliebige Zeichen
-notlike	Negierung von –like
-match	Textvergleich mit Regular Expressions
-contains	Überprüft, ob ein Wert in einer Sammlung (z.B. einem Array) vorkommt
-notcontains	Negierung von contains

Formatierte Ausgabe

Powershell bietet den Format-Operator –f an, um Ausgaben zu formatieren. Der Aufbau des Format-Parameters ist:

```
„{0}“ –f Eingabewert # Eingabewert ist z.B. ein Wert oder eine Variable
„{0:Formatoptionen}“ –f Eingabewert
```

{x} ist ein durchnummerierter Platzhalter, der Index gibt dabei den Übergabewert rechts vom –f an. Der Platzhalter {Index} ist ein String und muss in Anführungszeichen stehen!

Die einfachsten Formatoptionen sind 0 und #, wobei 0 für eine Pflichtstelle steht, # für eine optionale Stelle, die nur angegeben wird, wenn der Eingabewert auch ein vorhandene Stelle hat. Weiterhin gibt es aber auch noch eine ganze Reihe von weiteren vordefinierten Formatierungs-optionen.

```
„{0:#,##0.00} MB“ –f 100,7 # Ausgabe mit 1000-Trenner (=,) und 2 Nachkommatellen
100,70 MB
„{0:#,##0.00} MB“ –f 20100
20.100,00 MB
```

Um Spalten in Commandlets zu formatieren, wird ein Hashfeld als Übergabeparameter benutzt, das mind. 2 Werte enthält: Name = Spaltenname und Expression = {Formatierungsausdruck}, wobei der Formatierungsausdruck ein Kommandoblock ist und auch z.B. if-Abfragen enthalten kann. In der Expression steht $_ für die Pipeline-Variable, es kann in der Expression auf alle Spalten der Ausgabe zugegriffen werden, da sie erst in der Pipeline aufgerufen wird.

```
$Spalte = @{ Name=“Betrag“; Expression={„{0:C}“ –f $_.ID } }
get-process | Select-Object $Spalte # formatiert die ProzessID sinnloserweise als
# Währung – aber ist ja nur ein Beispiel
```

http://blogs.technet.com/b/heyscriptingguy/archive/2013/03/12/use-powershell-to-format-strings-with-composite-formatting.aspx

Powershell-Provider – Zugriff auf die Registry

Alle Registryschlüssel ausgeben, die Powershell im Namen tragen:

```
Get-childitem HKCU,HKLM –Include “*Powershell*” –Erroraction SilentlyContinue
```

Den Schlüssel Software anzeigen bzw. in $Software speichern

```
$software = Get-Item HKCU:\Software
```

Alle Unterschlüssel von Sofware anzeigen

```
Get-Childitem HKCU:\Software
```

Alle Werte von …\CurrentVersion anzeigen:

```
Get-Itemproperty 'HKLM:\Software\Microsoft\Windows NT\CurrentVersion'
```

Das gleich wie eben, aber über direkten Provider-Zugriff ohne Zuhilfename des Laufwerks HKLM:

```
Get-Itemproperty 'Registry::HKEY_LOCAL_MACHINE\Software\Microsoft\Windows
NT\CurrentVersion'
```

Speichern des Wertes ProductID aus dem Schlüssel …\CurrentVersion

```
$ID=Get-Itemproperty –Path 'HKLM:\Software\Microsoft\Windows NT\CurrentVersion' `
-Name ProductID
```

Mit dem Parameter –Expandproperty nur den String anzeigen, der sich in ProductID befindet, ohne das komplette Objekt anzuzeigen:

```
Get-Itemproperty 'HKLM:\Software\Microsoft\Windows NT\CurrentVersion' `
–ExpandProperty ProductID
```

Funktion zum Auslesen von Registry-Schlüsseln´:

```
Function Get-RegistryValue($Schlüssel,$Wert){
Get-Itemproperty "Registry::$Schlüssel* $Wert | Select-Object –Expandproperty $Wert
}
```

Die Installierte Software anzeigen:

```
Get-Itemproperty HKLM:\Software\Microsoft\Windows\CurrentVErsion\Uninstall\* |
Format-Table DisplayName, Publisher
```

Einen Wert überschreiben:

```
Set-Itemproperty –Path HKCU:\Software\Testschlüssel –Name NeuerWert –Value 100
```

Einen neuen Binärwert aus Hex-Zahlen erzeugen:

```
New-ItemProperty -Path 'HKCU:\Software\IvoSoft\ClassicExplorer' -name
"CSettingsDlg" -Value (&{'68,02,00,00,21,01,00,00'.split(",") | %{"0x"+$_}}) -
PropertyType BINARY
# String wird aufgesplittet und per Pipeline werden die Einzelwerte als Hex (0x)
markiert
```

Einen neuen Wert erzeugen mit Umgebungsvariable

```
Set-Itemproperty -Path HKCU:\Software\Testschlüssel –Name NeuerWert –Value
,%Windir%\Notepad.exe' –Type Expandstring
```

Umbenennen eines Schlüssels

```
Rename-Item –Path HKCU:\Software\Testschlüssel –NewName NeuerName
```

Löschen eines Wertes

```
Remove-ItemProperty HKCU:\Software\Testschlüssel NeuerWert
```

Löschen eines Schlüssels

```
Remove-Item
```

Bedingungen

If überprüft auf die Wahrheit eine Bedingung im ()-Block und führt den Script-Block aus, wenn der ()-Block $true ergibt

```
If () {}      # Erste Bedingung abprüfen
Elseif () {} # weitere Bedingung(en) abprüfen (Optional)
Else {}       # Default, falls keine Bedingung übereinstimmt (Optional)
```

Switch funktioniert wie if - $Wert wird mit den einzelnen Zahlen im Scriptblock verglichen. Allerdings bricht Switch nach einer Bedingung, die zu $true auswertet, nicht automatisch ab, sondern vergleicht trotzdem alle weiteren Fälle auch.

```
Switch ($wert)
{
	1 {„Zahl 1“}
	2 {„Zahl 2“}
	3 {„Zahl 3“}
}
```

```
Switch ($wert)
{
	{$_ -eq 1} {„Zahl 1“}
	{$_ -eq 2} {„Zahl 2“}
	{$_ -eq 3} {„Zahl 3“}
}
```

```
Switch ($wert)
{
	{$_ -le 5} {„Zahl kleiner 5“}
	{$_ -eq 6} {„Zahl 6“}
	{(($_ gt 6) –and ($_ le 10))} {„Zahl zwischen 7 und 10“}
	Default {„$_ ist außerhalb des untersuchten Bereichs“}
}
```

Switch kann mehr als 1 Ergebnis liefern, da alle Bedingungen geprüft werden! Um das zu verhindern, kann hinter jeden Scriptblock ein ;Break gesetzt werden.

Switch kann auch mehrere Werte bearbeiten, z.B. aus einem Array.

Schleifen

```
Do {…} While ()

While() {}

For ($i=1;$i -lt 100;$i++){}
```

Schleifen abbrechen:
Break > Abbruch und Beendigung der Schleife
Continue > Abbruch, nächster Schleifendurchlauf

Objekte

Get-Member zeigt die Eigentschaften und Methoden eines Objekts an:

```
Get-member –Membertype Property
‘Ich bin ein kleiner Text‘ | get-member –membertype *property
‘Ich bin ein kleiner Text‘.length
‘Ich bin ein kleiner Text‘.chars(4)
Dir $env:windir | get-member –membertype *property
$erg = Dir $env:windir | get-member –membertype *property | Group-Object Typename
Compare-object $erg[0].group $erg[1].Group –Passthru –Property Name
```

Beispiel: Dateiversion bestimmen:

```
Dir $env:Windir\system32\*.dll | select-object name, versioninfo
$file = Dir $env:Windir\system32\*.dll | select-object name, versioninfo –First 1
$Test.Versioninfo
$test.Versioninfo | get-member –membertype *property
$test.versioninfo | select-object *
Dir $env:Windir\system32\*.dll | select-object name,
{$_.versioninfo.Productversion}
```

Beispiel: Alter einer Datei anzeigen mit Hasharray:

```
New-item –path c:\ArchiveLogs -ErrorAction SilentlyContinue
$spalte1 = @{ Name='Alter'; Expression={(New-Timespan $_.Lastwritetime).Days} }
Get-childitem $env:Windir\*.log | Select-Object *, $Spalte1 | Where-Object
{$_.Alter -gt 1} | copy-item -destination C:\archivelogs
Get-childitem c:\archivelogs | select-object Name, $Spalte1
```

Eigene Objekte erzeugen

Mit Select-Object (kann nur NoteProperties hinzufügen)

```
$objektDerBegierde = “Gisele Bündchen” | select-object Brust, Bauch, Huefte
```

```
$objektDerBegierde.Brust = 90
```

Mit Hashfeldern

```
$hash=@{} # leeres Hashfeld erzeugen
$hash.Biosversion = { Get-CIMInstance Win32_Bios }.Version
$hash.Speicher = { Get-CIMInstance Win32_Computersystem }.TotalPhysicalMemory
$hash.Betriebssytem = { get-CIMInstance Win32_OperatingSystem }.Caption
$hash.Name = $env:Computername
$meinObject = New-object PSObject –Property $hash
```

Das Windows-Verzeichnis ($env:Windir) auslesen, und allen Objekten eine neue **Eigenschaft** Alter hinzufügen mit „Add-Member Noteproperty“ :

```
Get-Childitem $env:windir | foreach-object { $_ | Add-member NoteProperty Alter
((New-Timespan $_.CreationTime (Get-Date)).Days) -PassThru} | Format-Table Name,
Alter, CreationTime
```

Der Win.Ini eine neue **Methode** OpenEditor hinzufügen mit „Add-Member ScriptMethod“ :

```
$file = add-member -InputObject (Get-ChildItem $env:windir\win.ini) -MemberType
ScriptMethod -Name OpenEditor -Value { notepad.exe $this.Fullname } –PassThru
$file.OpenEditor
```

Ein Überblick, wann man Add-Member und wann Select-Object nutzen sollte:

Wenn man…	You should use...
Ein paar Eigenschaften fix einem bestehenden Objekt hinzufügen möchte	Select-Object
Eigenschaften hinzufügen möchte, die immer aktuell sein sollen oder eine Überprüfung von Werten durchführen sollen	Add-Member
Eine Sammlung von Werten als Objekt speichern will	Select-Object
Ein bestehendes Objekt nutzerfreundlicher gestalten will	Add-Member
Eine einzelne Eigenschaft einem Objekt hinzufügen möchte	Add-Member

Select-Object	Add-Member
`1 | Select-Object @{ Name='Foo' Expression={"Bar"} }, @{ Name='Bar' Expression={"Baz"} }`	`New-Object Object | Add-Member NoteProperty Foo Bar -passThru | Add-Member NoteProperty Bar Baz -passThru`

http://blogs.msdn.com/b/mediaandmicrocode/archive/2008/11/26/microcode-powershell-scripting-tricks-select-object-note-properties-vs-add-member-script-properties.aspx?Redirected=true

Statische Objekteigenschaften von Typen anzeigen lassen – statische Eigenschaften bzw. Methoden kann man direkt aus der Klasse aufrufen, sie benötigen kein Objekt.

```
[datetime] | Get-member –static
```

Der Aufruf von statischen Funktionen erfolgt über den Klassennamen in Klammern, zwei Doppelpunkte und die Methode/Eigenschaft.

Anwendungsbeispiel Klasse Convert:

```
[System.Convert] # konvertiert Datentypen, auch in andere Zahlensysteme:
[System.Convert]::ToSTring(42562,16) > Konvertiert in Hex
[System.Convert]::ToSTring(42562,2) > Konvertiert in Binär
```

[System.Net.DNS] # Netzwerkfunktionen (DNS-Auflösung)

```
[Systems.Environment]::UserDomainName
[Systems.Environment]::UserName
[Systems.Environment]::MachineName
[Systems.Globalization.Cultureinfo]::CurrentCulture
```

Assembly herausfinden, zu der der Typ gehört:

```
[System.Diagnostics.Process].Assembly
[Systems.Environment].Assembly.GetExportedTypes()  #zeigt die Typen an, die in der
# Assembly sind
```

Nur nach öffentlichen (Nutzbaren) Klassen suchen:

```
[Systems.Environment].Assembly.GetExportedTypes()  | where-object {$_.isPublic} |
where-object {$_.isClass} | select-object Fullname
```

Neue Objekte anlagen:

```
$webclient = New-Object System.Net.Webclient
$webclient.Downloadstring('http://www.newhorizons.de')
```

Assemblys nachladen:

```
[reflection.assembly]::LoadWithPartialName("Microsoft.VisualBasic")
```

Konstruktoren verwenden:

```
New-object System.String(".",1000)
```

Konstruktoren werden benötigt, wenn ein neues Objekt mit Vorgabewerten erstellt werden soll

Com-Objekte anlegen:

```
$update = New-object –comobject "Microsoft.Update.Autoupdate"
$Update.results
$update.DetectNow()
```

Funktionen

Parameter

Parameter mit param-Block deklarieren:

```
Function test
{
  Param(
  [Parameter(Mandatory=$true,
             HelpMessage='Ausgabe bei fehlendem Parameter')]
  [Alias('Eur')]
  [Double]
  $Betrag,

  [switch]
  $invers
  )
…
}
```

Argument-Autocompletion

Powershell 3.0 kann auch Argumente mit Autocompletion anzeigen. Dafür muß im Script aber vorgegeben werden, welche Werte / Typen für die Argumente erlaubt sind:

```
function Select-Color
{
    Param(
        [System.ConsoleColor]
        $Color
    )

    "You selected: $Color"
}

function Select-City
{
    Param (
        [ValidateSet('New York','Redmond','Hanover','Tokio')]
        $City
    )

    "You selected: $City"
}
```

http://www.powertheshell.com/argument-completion-in-powershell-3-0/

Parameter-Sets definieren:

```
Function test-Binding
{
  [CmdletBinding(DefaultParameterSetName='Name')]
  Param(
  [Parameter(ParameterSetName='ID',Position=0,Mandatory=$true)][INT]$ID,
  [Parameter(ParameterSetName='Name',Position=0,Mandatory=$true)][String]$Name
  )

  $set = $PSCmdlet.ParameterSetName
  "Sie haben Parameterset $set gewählt"
  If ($set –eq ‚ID‘)
       { „Die ID ist $ID“ }
  Else { „der Name lautet $Name“ }
…
}
```

Objekte by Reference übergeben:

```
Function Test-Parameter([Ref]$Wert)
{
   $Wert = "Neuer Inhalt"
}

PS> $daten = [Ref]'Alter Wert' #Übergabeobjekt muß auf Referenzobjekt sein!
PS> Test-Parameter $daten
PS> $daten.Value #[Ref] ist ein Container, Inhalt befindet sich in Value!
```

Werte aus Funktionen zurückliefern

Variablen innerhalb einer Funktion sind Funktionslokal, gehen also nach Beendigung der Funktion wieder verloren. Alle Werte, die außerhalb der Funktion verwendet werden sollen, müssen also aus der Funktion zurück geliefert werden.
Alle Daten, die in einer Funktion ausgegeben werden, können als Rückgabewerte in eine Variable gespeichert werden:

```
Function Write-ReturnValue
{
  Param(
    [int]$Wert1,
    [int]$Wert2
  )

  $Wert1 + $Wert2
}

$Ergebis = Write-ReturnValue -Wert1 10 -Wert2 20 # $Ergebnis enthält 30
```

Werden mehrere Rückgabewerte erzeugt, werden diese in einem Array gespeichert:

```
Function Write-ReturnValue
{
  Param(
    [int]$Wert1,
    [int]$Wert2
  )

  $Wert1 + $Wert2
  Write-Output "Das Ergebnis lautet $Wert1 + $Wert2"
}

$Ergebis = Write-ReturnValue -Wert1 10 -Wert2 20
$Ergebnis
$Ergebnis.gettype()
```

```
30
Das Ergebnis lautet 10 + 20

IsPublic IsSerial Name                                     BaseType
-------- -------- ----                                     --------
True     True     Object[]                                 System.Array
```

Rückgabewerte einer Funktion auf Array überprüfen:

```
$ergebnis = Test-Funktion 1
$ergebnis.GetType().isArray
$ergebnis –is [Array]
```

Rückgabe immer in Array einschliessen:

```
$ergebnis = @(Test-Funktion 1)
```

Um zu verhindern, dass Statusmeldungen einer Funktion in der Rückgabe landen, write-host verwenden

```
Function test
{
   Write-host „dies ist eine Debugmeldung“
   „Und diese Meldung ist ein Rückgabewert der Funktion“
}

$ausgabe = Test
Dies ist eine Debugmeldung

$ausgabe
Und diese Meldung ist ein Rückgabewert der Funktion
```

Alternativ mit Write-Debug

```
Function test
```

```
{
   Write-Debug „dies ist eine Debugmeldung“
}

$DebugPreference = Continue

Test
Debug: Dies ist eine Debugmeldung
```

Alternativ mit Write-Verbose

```
Function test
{
   Write-verbose „dies ist eine Debugmeldung“
}

Test -verbose
Ausführlich: Dies ist eine Debugmeldung
```

Cmlet-Binding (Standard-Parameter –verbose und –ErrorAction werden eingebunden):

```
Function test
{
  [CmdletBinding()] # Alle Standard-Parameter werden unterstützt
  Param(
  [Parameter(Mandatory=$true)]
  [Alias(‘Eur’)]
  [Double]
  $Betrag
   )
Write-Verbose(“Der Text enthält {0} Zeichen’ –f $text.Length)
…
}

Test 123 -verbose
```

-Whatif konfigurieren

Soll eine Funktion des Parameter Whatif unterstützen, muß dies explizit im [CmdletBinding] angegeben werden. Wird die Funktion dann mit –whatif aufgerufen, wird der Parameter automatisch auch an alle Commandlets weitergegeben, die innerhalb der Funktion aufgerufen werden.

```
function remove-windows
{
  [CmdletBinding(SupportsShouldProcess=$true)]
  param()

  remove-item -Path F:\Windows\*.*
}
```

Soll Whatif auf Code angewendet warden, muß die Variable $PSCmdlet abgefragt werden. $PSCmdlet.Shouldprocess ist fals, wenn –Whatif gesetzt wurde.

```
Function Enable-AutoPageFile
{
  [CmdletBinding(SupportsShouldProcess=$true)]
  Param()
  $computer=Get-WmiObject –Class Win32_Computersystem -EnableAllprivileges
  $computer.AutomaticManagedPagefile=$true
  If ( $PSCmdlet.ShouldProcess( “Lokaler Computer”,”Automatische
  Auslagerungsdatei einschalten“ ))
       { $Computer.Put()| out-null }
}

Enable-AutoPageFile –whatif

What If: Ausführen des Vorgangs “Automatische Auslagerungsdatei einschalten” für das Ziel “lokaler Computer”

# Der Scriptblock im If wird nicht ausgegeben, stattdessen wird der Text hinter
# ShouldProcess ausgegeben, wenn –whatif gesetzt ist
```

Eigenen Funktionen für Get-Help vorbereiten

(In der ISE über Strg-J > erweiterte Cmdlets erreichbar)

```
<#
.Synopsis
Hier steht die Zusammenfassung
.Description
Hier folgt eine ausführliche Beschreibung der Funktion
.Parameter Parameter1
Der erste Parameter
.Parameter Parameter2
Dieses ist der zweite Parameter
.Example
```

```
Test-funktion Hallo Du
Gibt die beiden Parameter farbig aus
.Notes
Dies ist das einfache Beispiel einer Funktion
.Link
About_comment_based_help
#>
```

Weitere mögliche Stichworte:
.Inputs
.Outputs
.Component
.Role
.Functionality
.Forwardhelp-Targetname
.ForwardhelpCategory
.RemotehelpRunspace <Variable>
.Externalhelp <XML-Hilfepfad>

Funktionen und Pipeline

Mit $element eine Funktion Pipelinefähig machen – dieser Modus sammelt erst alle Elemente der Pipeline im Speicher in der Variablen $Element. Die Funktion startet erst, wenn das vorherige Commandlet alle Objekte abgearbeitet hat. Dieser Prozess kann sehr viel Speicher verbrauchen und lange dauern!

```
function markExe
{
    $oldcolor = $host.ui.RawUI.ForeGroundColor

    foreach ($element in $input) {
        if ($element.Extension -eq '.exe') {
            $Host.Ui.RawUi.ForegroundColor ="red"
         } else {
            $host.ui.RawUI.ForegroundColor = $oldcolor
        }
        $element
    }
    $host.ui.RawUI.ForegroundColor = $oldcolor
}
```

Schneller Streamingmodus – alle Objekte werden direkt nacheinander bearbeitet

```
function markExe
{
   process {
        $oldcolor = $host.ui.RawUI.ForeGroundColor

            if ($_.Extension -eq '.exe') {
                $Host.Ui.RawUi.ForegroundColor ="red"
```

```
            } else {
                $host.ui.RawUI.ForegroundColor = $oldcolor
            }
            $_
        }
        $host.ui.RawUI.ForegroundColor = $oldcolor
    }
}
```

Mark-Exe mit Begin – Process -End

```
function mark-Exe
{
 begin {
    $oldcolor = $host.ui.RawUI.ForeGroundColor
 }

 process{
    if ($_.Extension -eq '.exe') {
        $Host.Ui.RawUi.ForegroundColor ="red"
    } else {
        $host.ui.RawUI.ForegroundColor = $oldcolor
    }
    $_
 }

 end {
    $host.ui.RawUI.ForegroundColor = $oldcolor
 }
}
```

Pipeline und direkte Argumente kombinieren

```
Function test
{
  Param(
  [Parameter(Mandatory=$true,
             ValueFromPipeline=$true)]
  [Alias('Eur')]
  [Double]
  $Betrag,

  [Double]
  [Alias('Kurs')]
  $Wechselkurs=1.35
  )
```

```
  Process {
       $Betrag * $Wechselkurs
  }

}
```

Funktionen schreibschützen:

```
Set-item Function:test –Option Readonly
```

Scripte

Wird ein Ordner für Scripte anlegt und in den Pfad ($env:Path) aufgenommen, können Scripte direkt über den Scriptnamen aufgerufen werden
$myinvocation : Diese Autovariable beinhaltet immer den Pfad, ausdem das Script aufgerufen wurde.
Festlegen, dass das Script PSH V2.0 braucht: #requires –Version 2
Festlegen, dass das Script ein bestimmtes Snap-In braucht: #requires –PsSnapin
`Quest.ActiveRoles.ADManagement
Festlegen, dass eine bestimmte Powershell-Version gebraucht wird: `#requires -version 2`

Mit

```
Set-StrictMode –Version 1.0
 # oder
Set StrictMode –Version 2.0
```

wirft Powershell eine Warnung aus, wenn auf nicht vorhandene Eigenschaften zugegriffen wird oder eine leere Variable referenziert wird..

Profile

Es gibt 6 Profile, 2 davon speziell für die ISE – die normalen Profile werden in der ISE nicht geladen.

```
PS:> $Profile | Format-List * -Force

AllUsersAllHosts       : C:\Windows\System32\WindowsPowerShell\v1.0\profile.ps1

AllUsersCurrentHost    :
C:\Windows\System32\WindowsPowerShell\v1.0\Microsoft.PowerShell_profile.ps1

CurrentUserAllHosts    : C:\Users\Holger\Documents\WindowsPowerShell\profile.ps1

CurrentUserCurrentHost :
C:\Users\Holger\Documents\WindowsPowerShell\Microsoft.PowerShell_profile.ps1

Length                 : 76
```

http://blogs.technet.com/b/heyscriptingguy/archive/2013/01/04/understanding-and-using-powershell-profiles.aspx

Module

Module sind Scriptdateien mit der Endung psm1. Sie müssen im Ordner Modules (User oder Maschine) in einem Ordner liegen, der heißt wie das Script.
In Powershell 2.0 müssen Module explizit über Import-Module geladen werden, um die Commandlets des Moduls nutzen zu können. Powershell 3.0 lädt Module automatisch, sobald ein Commandlet aufgerufen wird, dass in der Umgebung noch nicht zur Verfügung steht.

```
Get-Module  #listet die geladenen Module auf
Get-Module –listavailable
Import-Module xyz
Import-Module –force #um ein geladenes Modul erneut zu laden, z.B. bei Änderungen
Get-command –module xyz
Remove-Module xyz
```

Wenn nicht alle, sondern nur einzelne Funktionen oder Aliase des Moduls sichtbar sein sollen, können diese exportiert werden – sinnvoll z.B., um interne Hilfsfunktionen zu verbergen:

```
Export-ModuleMember -function Get-OSInfo
Export-ModuleMember -alias goi
```

Module können auch über ein Manifest gesteuert werden – dann wird die Manifest-Datei (.psd1) statt des Moduls importiert. Neben Metainformationen kann über ein Manifest z.B. festgelegt werden, dass mehrere Script-Files importiert werden.

```
New-modulemanifest
```

Die Pfade zu den Modulpfaden sind in der Umgebungsvariablen PSModulePath gespeichert:

```
PS:> $env:psmodulepath.split(";")
C:\Users\Holger\Documents\WindowsPowerShell\Modules
C:\Program Files\WindowsPowerShell\Modules
C:\Windows\system32\WindowsPowerShell\v1.0\Modules\
```

Powershell-Remoting

Powershell-REmoting läuft auf Port 5985 / 5986(SSL) als Webdienst
Die Prozesse der Remote-Sitzung heissen wsmprovhost
Sitzungen, die länger als 3 Minuten inaktiv sind, werden beendet
(Dir wsman:\localhost\shell\idletimeout)

Enable-PSRemoting

```
(Optional: Set-item wsman:\Localhost\clients\trustedhosts * -Force
Set-item wsman:\Localhost\clients\trustedhosts server_* -Force
Set-item wsman:\Localhost\clients\trustedhosts 10.10.10.* -Force
```

```
Invoke-command {Get-Service} –Computername DC1
Invoke-command {get-service} –Computername DC1 –Credential (Get-Credential)
```

```
Enter-PSSession –Computername DC1
Get-service
Exit-PSSession
```

```
$session = New-PSSession –computername DC1
Invoke-Command {$env:Computername; $wert=1}  –session $session
$session
Remove-PSSession $session
```

Fan-out (Session zu mehreren Rechnern)

```
$session = New-PSSession Computer1, Computer2, Server1
Invoke-command {md HKLM:\Software\test} –session $session
Get-Pssession -ID 2
Remove-PSSession –ID 2
Remove-PSSession *
```

Active Directory-Commandlets

AD-Objekte abrufen (22 Cmdlets)

- Get-ADAccountAuthorizationGroup
- Get-ADAccountResultantPasswordReplicationPolicy
- Get-ADComputer
- Get-ADComputerServiceAccount
- Get-ADDefaultDomainPasswordPolicy
- Get-ADDomain
- Get-ADDomainController
- Get-ADDomainControllerPasswordReplicationPolicy
- Get-ADDomainControllerPasswordReplicationPolicyUsage
- Get-ADFineGrainedPasswordPolicy
- Get-ADFineGrainedPasswordPolicySubject
- Get-ADForest
- Get-ADGroup
- Get-ADGroupMember
- Get-ADObject
- Get-ADOptionalFeature
- Get-ADOrganizationalUnit
- Get-ADPrincipalGroupMembership
- Get-ADRootDSE
- Get-ADServiceAccount
- Get-ADUser
- Get-ADUserResultantPasswordPolicy

AD-Objekte erstellen (7 Cmdlets)

- New-ADComputer
- New-ADFineGrainedPasswordPolicy
- New-ADGroup
- New-ADObject
- New-ADOrganizationalUnit
- New-ADServiceAccount
- New-ADUser

AD-Objekte entfernen (12 Cmdlets)

- Remove-ADComputer
- Remove-ADComputerServiceAccount
- Remove-ADDomainControllerPasswordReplicationPolicy
- Remove-ADFineGrainedPasswordPolicy
- Remove-ADFineGrainedPasswordPolicySubject
- Remove-ADGroup
- Remove-ADGroupMember
- Remove-ADObject
- Remove-ADOrganizationalUnit
- Remove-ADPrincipalGroupMembership
- Remove-ADServiceAccount
- Remove-ADUser

AD-Schreibvorgänge durchführen (15 Cmdlets)

- Set-ADAccountControl
- Set-ADAccountExpiration
- Set-ADAccountPassword
- Set-ADComputer
- Set-ADDefaultDomainPasswordPolicy
- Set-ADDomain
- Set-ADDomainMode
- Set-ADFineGrainedPasswordPolicy
- Set-ADForest
- Set-ADForestMode
- Set-ADGroup
- Set-ADObject
- Set-ADOrganizationalUnit
- Set-ADServiceAccount
- Set-ADUser

AD-Objekte hinzufügen (5 Cmdlets)

- Add-ADComputerServiceAccount
- Add-ADDomainControllerPasswordReplicationPolicy
- Add-ADFineGrainedPasswordPolicySubject
- Add-ADGroupMember
- Add-ADPrincipalGroupMembership

AD-Objekte und optionale AD-Funktionen deaktivieren (2 Cmdlets)

- Disable-ADAccount
- Disable-ADOptionalFeature

AD-Objekte und optionale AD-Funktionen aktivieren (2 Cmdlets)

- Enable-ADAccount
- Enable-ADOptionalFeature

AD-Objekte verschieben (3 Cmdlets)

- Move-ADDirectoryServer
- Move-ADDirectoryServerOperationMasterRole
- Move-ADObject

AD-Objekte umbenennen (1 Cmdlet)

- Rename-ADObject

AD-Dienstkontenkennwörter zurücksetzen (1 Cmdlet)

- Reset-ADServiceAccountPassword

AD-Objekte wiederherstellen (1 Cmdlet)

- Restore-ADObject

AD-Objekte suchen (1 Cmdlet)

- Search-ADAccount

AD-Dienstkonto deinstallieren (1 Cmdlet)

- Uninstall-ADServiceAccount

AD-Objekt entsperren (1 Cmdlet)

- Unlock-ADAccount

AD-Kontoablaufdatum zurücksetzen (1 Cmdlet)

- Clear-ADAccountExpiration

AD-Dienstkonto installieren (1 Cmdlet)

- Install-ADServiceAccount

http://blog.dikmenoglu.de/ADPowerShell+Befehle.aspx