



Powershell für Systemadministratoren

Release 1.0

STE

26.11.2017

Inhaltsverzeichnis

1 PowerShell auf der Konsole	3
2 PowerShell im Interaktiv-Modus	13
3 Variablen und Datentypen	19
4 Operatoren	33
5 Diagramme der strukturierten Programmierung	37
6 Bedingungen	47
7 Schleifen	57
8 Arrays und Hashs	63
9 Funktionen	77
10 Pipeline	107
11 Objekte in der PS	131
12 Fehlerbehandlung	147
13 Aufgaben	153
14 PS und Objektorientierung	181
15 Powershell und Klassendiagramm	183
16 Indizes und Tabellen	189

Summary

Release 1.0

Date 26.11.2017

Authors STE

Target developers and administrators

status development

KAPITEL 1

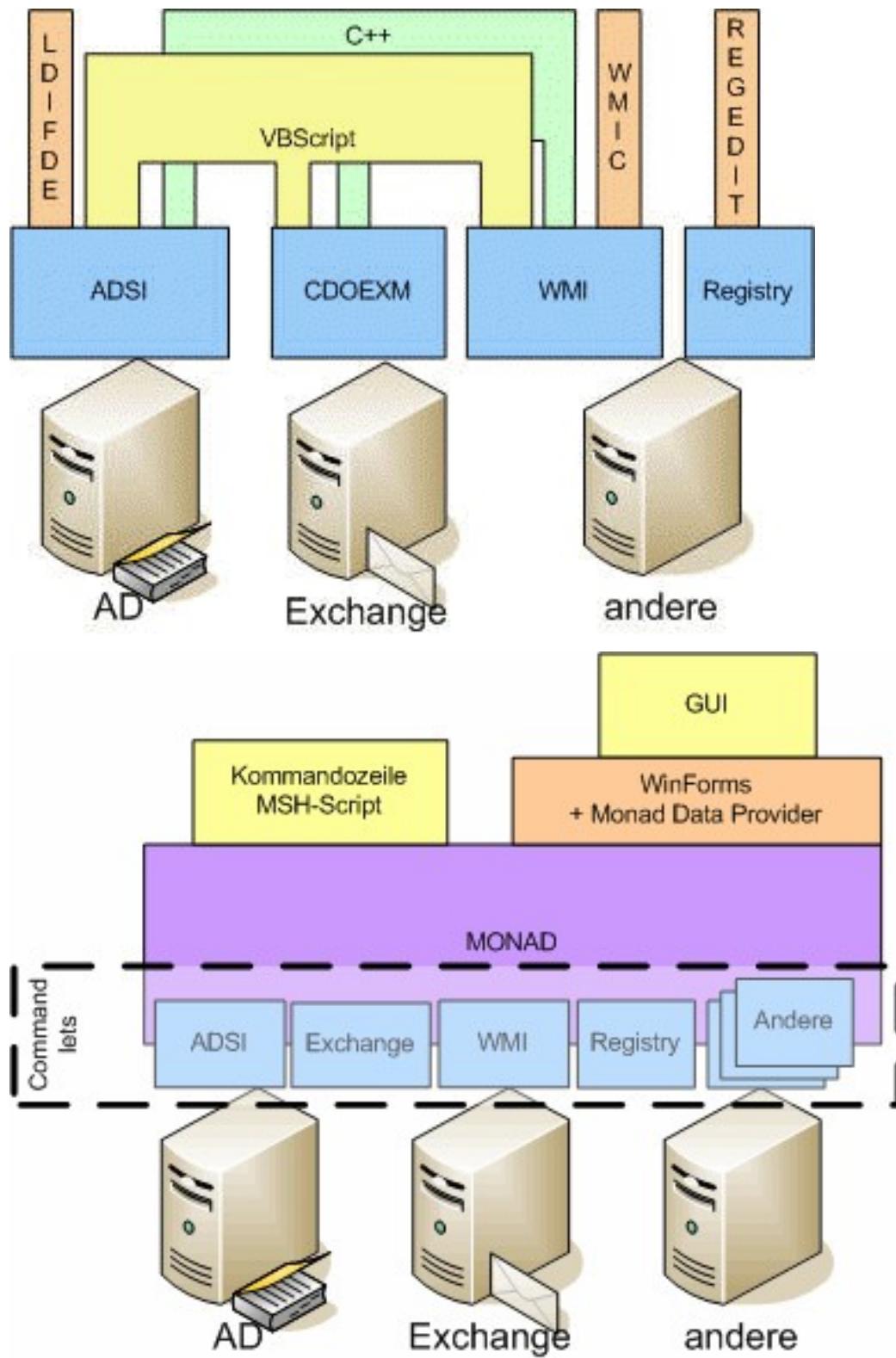
PowerShell auf der Konsole

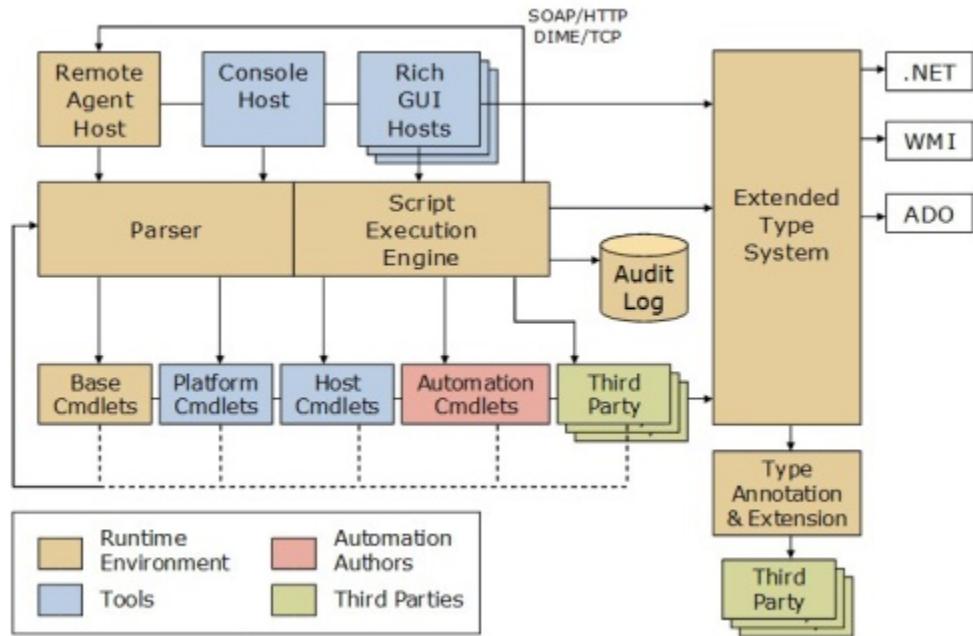
Während Microsoft bereits seit vielen Jahren für seine Anwender- und Betriebssystemsoftware bekannt ist, hat sie relativ lange Zeit wenig Ressourcen in die Bereitstellung von Scriptingumgebungen gesteckt.

Relativ lange war die DOS-Konsole mit seinen wenigen Befehlen die einzige Möglichkeit, Windows von der Konsole her anzupacken. Mit Beginn von Windows 95 wurde deshalb mit der Scriptsprache **vbscript** die Möglichkeit geschaffen, für den Systemadministrator eine Plattform zur Verfügung zu stellen, um Windows Betriebssysteme automatisieren zu können.

Mit dem Erscheinen der .NET-Programmierplattform und dem Wechsel von der alten COM-Welt zu dieser neuen Plattform gab es allerdings zunächst keine Unterstützung dieses Frameworks aus Sicht des Systemadministrators.

Powershell schließt diese Lücke. Es baut auf dem .NET-Framework 2.0 auf und ermöglicht den Zugriff sowohl auf die alte COM-Welt als auch auf die Klassen des .NET-Frameworks.



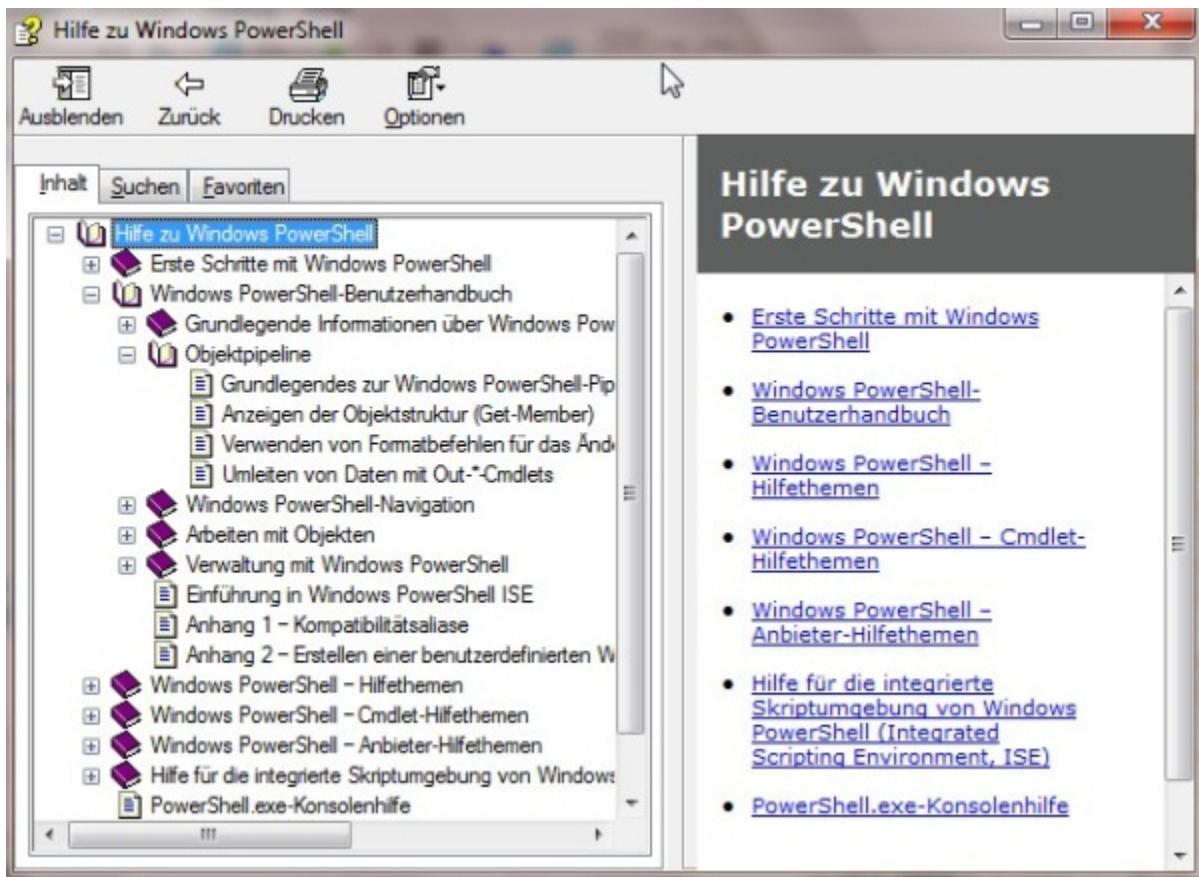


1.1 Installation von Powershell

PowerShell ist seit Windows 7 und Server 2008 ein integraler Bestandteil des Betriebssystems. Unter früheren Systemen kann man es nachinstallieren.

1.2 Hilfe und Tutorials

Nach der Installation von Powershell findet man eine lokalisierte Hilfe als CHM-Datei.



Weiterhin gibt es mittlerweile viele Webseiten bzw. Tutorials, die sich mit der PowerShell beschäftigen.

- <http://technet.microsoft.com/en-us/scriptcenter/dd742419.aspx>
- <http://powershellcommunity.org/>
- <http://powershell.com/>
- <http://channel9.msdn.com/tags/PowerShell/>

Mittlerweile gibt es eine ganze Reihe von Büchern, Amazon listet u.a. diese

- <http://www.amazon.de/registry/wishlist/D9P356SKZ575?reveal=unpurchased&filter=3&sort=date-added&layout=standard&x=2&y=1>

1.3 Umgang mit der Konsole und GUI

Der erste Eindruck von Powershell wird sicherlich von dem blauen Konsolenfenster bestimmt, welches sich nach einem Klick auf Powershell öffnet.

A screenshot of a Windows PowerShell window titled "Administrator: C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe". The window shows the following text:
Windows PowerShell
Copyright (C) 2009 Microsoft Corporation. Alle Rechte vorbehalten.
PowerTab version 0.99 Beta 2 PowerShell TabExpansion library
PowerTab Tabexpansion additions enabled : True
PS C:\Users\steinam> get-help

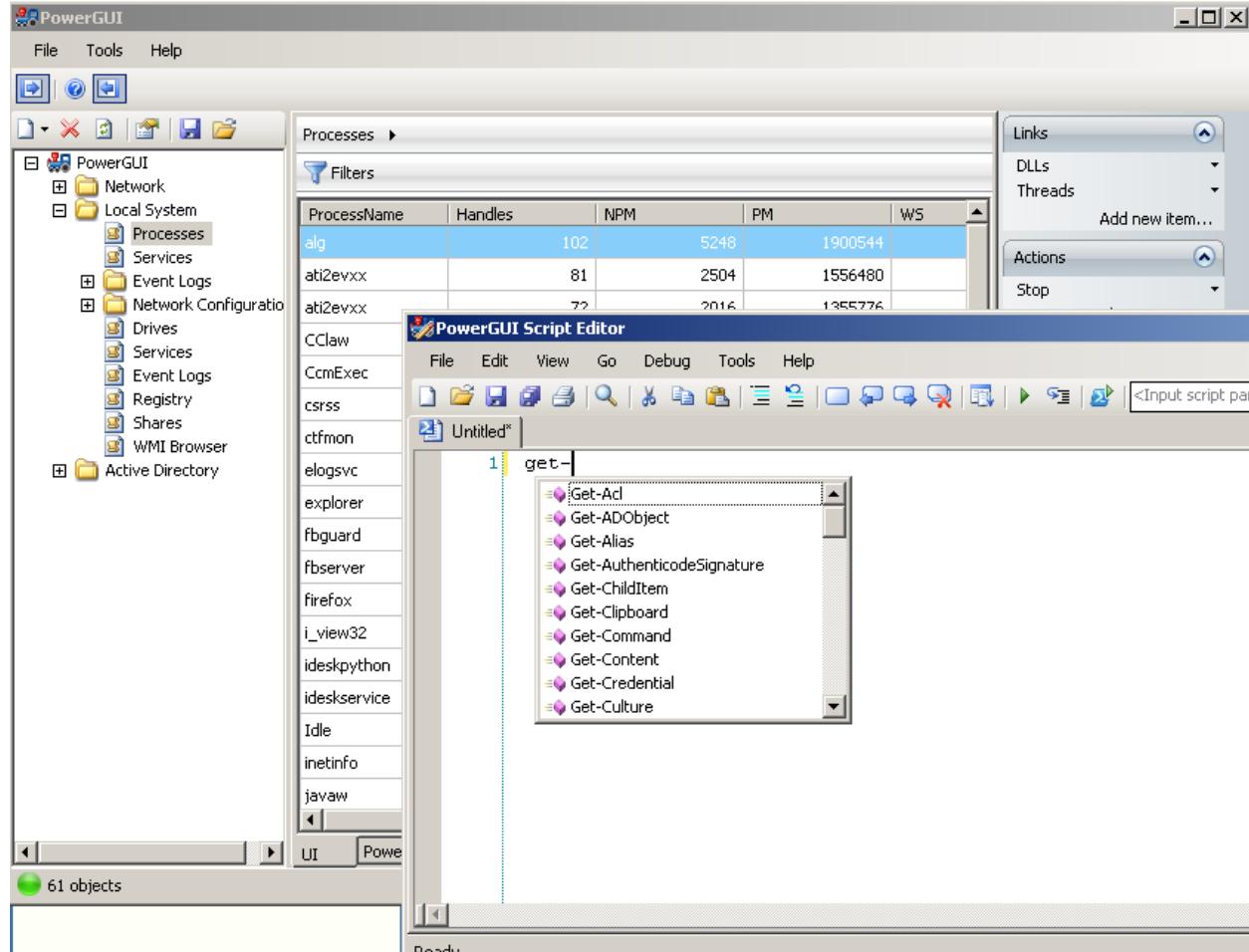
Während PS in der Version 1.0 über keine weiteren *grafischen* Elemente verfügte, hat Microsoft der Version 2.0 eine grafische Oberfläche spendiert.

A screenshot of the Windows PowerShell ISE window titled "Administrator: Windows PowerShell ISE". The interface includes a menu bar (Datei, Bearbeiten, Anzeigen, Debug, Hilfe), a toolbar with various icons, and a code editor window. The code editor shows a file named "Unbenannt1.ps1*" containing the command "Get-help". Below the code editor is a results pane displaying the output of the "get-help" command. At the bottom of the window, there is a status bar showing "Abgeschlossen" and "Ln 2 Spalte 1 18".

Daneben gibt es eine Reihe weiterer Programme, die ein grafische Benutzeroberfläche incl. Syntaxhighlightning und CodeCompleting bieten. Im Folgenden werden wir das Programm ([PowerGUI](#)) der Firma Quest Software benutzen.

Neben dem kostenlosen Skripteditor bietet die Firma auch einige interessante Commandlets zur Manipulation des ActiveDirectory und des ExchangeServers.

(PowerGUI) ist ein Freeware-Programm, welches sowohl über eine Konsole als auch ein Skriptfenster verfügt. Das Skriptfenster verfügt über Code-Completing und Syntax-Highlighting und ist deshalb für umfangreichere Programme die erste Wahl.



Die Wahl des Editors ist letztlich eine Geschmacksfrage.

1.3.1 Erste Schritte auf der Konsole:

- Mult-Line-Entries
- Wichtige Keyboard-Shortcuts
- Command-History
- Automatische Eingabe-Vervollständigung
- Text auswählen und einfügen

After PowerShell starts, its console window opens, and you see a blinking text prompt, asking for your input with no icons or menus. PowerShell is a command console and almost entirely operated via keyboard input. The prompt begins with „PS“ and after it is the path name of the directory where you are located. Start by trying out a few commands. For example, type:

1.3.2 Anpassen der Konsole

- Konsole-Eigenschaften
- Fonts und Fontgröße
- Window- und Buffersize
- Farben

1.3.3 Piping and Redirecting

Häufig werden die Informationen, die die PowerShell ausgibt, über mehrere Bildschirmseiten verteilt. Um dies besser kontrollieren zu können, kann man mit Hilfe des sog. *Piping* eine seitenweise Darstellung erreichen. Geben Sie zum besseren Verständnis einfach einmal folgende Befehle ein:

```
Get-Process
Get-Process | more (Enter)
```

Im 2. Befehl wird das *Piping*-Symbol | benutzt, um eine seitenweise Darstellung zu erreichen. Im späteren Verlauf wird auf dieses mächtige Konzept noch näher eingegangen.

Handles	NPM(K)	PM(K)	WS(K)	UM(M)	CPU(s)	Id	ProcessName
133	18	5436	2496	105	4.78	2388	avgn
164	27	95084	13576	169		1288	avguard
54	6	1308	2772	26		1420	avshadow
30	4	732	1940	25		1428	conhost
52	7	2124	6224	68	0.38	1788	conhost
493	11	1748	3280	47		368	csrss
350	12	1884	6172	49		428	csrss
113	11	3336	5504	78	0.48	2288	DTLite
79	7	1520	4856	63	0.08	2868	dwm
760	48	25688	49432	232	12.22	2964	explorer
374	38	102208	117564	305	696.11	2608	firefox
0	0	0	24	0		0	Idle
129	13	5860	7672	66		1392	inetinfo
49	8	1112	4096	68	0.03	2496	jusched
736	23	3872	8436	40		524	lsass
145	7	2152	3456	22		536	lsm
308	28	20976	9052	201	2.23	2356	LxUpdateManager
406	24	50340	48644	571	2.11	2916	powershell
193	16	3832	1756	57		1132	sched
636	35	20868	16272	121		2540	SearchIndexer
214	14	4256	6880	37		516	services
29	2	368	812	5		292	smss
285	20	6412	8596	98		1096	spoolsv
377	44	160124	70808	-1780		1412	sqlservr

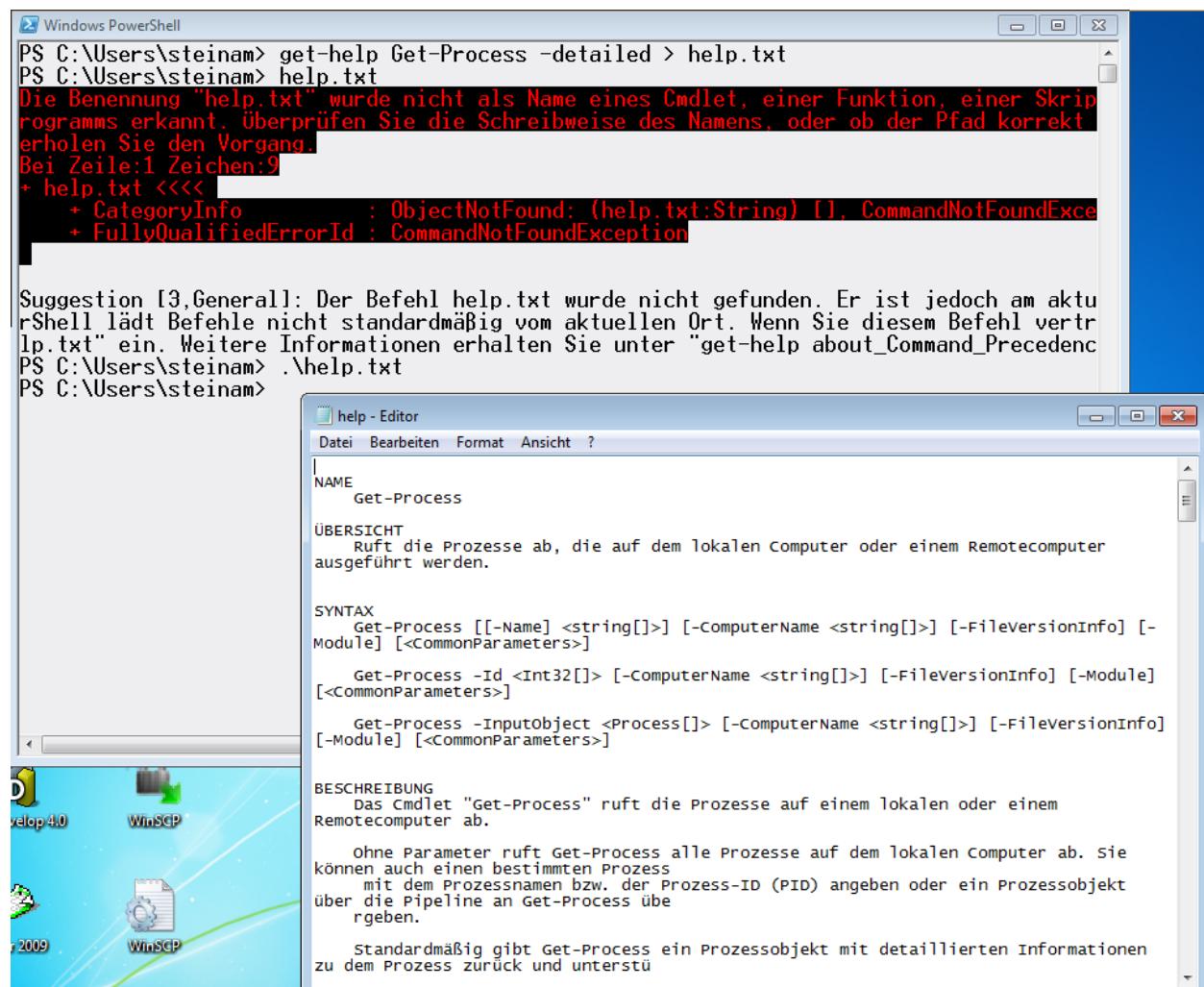
-- Fortsetzung --

Häufig will man die Ausgabe nicht nur auf dem Bildschirm sehen, sondern die Ergebnisse in einer Datei festhalten. Dazu dient das „>“ - Symbol, welches die Ausgabe in eine Datei umlenkt.

```
help > help.txt (Return)
```

Um nun die Datei in Notepad zu öffnen, muss man lediglich auf der Konsole folgenden Befehl eingeben.

```
. \help.txt
```



Die Ausgabe kann auch in eine bereits bestehende Datei umgeleitet werden. Der Text dieser Datei wird damit nicht überschrieben, sondern erweitert. Dies erfolgt mit dem Zeichen „>>“.

```
cmd /c help >> help.txt (Return)
```

Will man das Ergebnis eines Befehls weiterverarbeiten, muss man nicht nur mit redirecting arbeiten. Powershell kann das Ergebnis jedes Befehls in einer Variable speichern, auf die man innerhalb der bestehenden Powershell-Session Zugriff hat.

```
$result = ipconfig
$result
```

```
Windows PowerShell
PS C:\Users\steinam> $result = ipconfig
PS C:\Users\steinam> $result

Windows-IP-Konfiguration

Ethernet-Adapter LAN-Verbindung:

  Verbindungsspezifisches DNS-Suffix: Speedport_W_504V_Typ_A
  Verbindungslokale IPv6-Adresse . . . fe80::ed69:cc32:aac0:f26f%11
  IPv4-Adresse . . . . . 192.168.0.7
  Subnetzmaske . . . . . 255.255.255.0
  Standardgateway . . . . . 192.168.0.1

Tunneladapter isatap.Speedport_W_504V_Typ_A:

  Medienstatus. . . . . : Medium getrennt
  Verbindungsspezifisches DNS-Suffix: Speedport_W_504V_Typ_A

Tunneladapter LAN-Verbindung*:

  Verbindungsspezifisches DNS-Suffix:
  IPv6-Adresse . . . . . 2001:0:5ef5:79fd:18a6:2e96:3f57:fff8
  Verbindungslokale IPv6-Adresse . . fe80::18a6:2e96:3f57:fff8%13
  Standardgateway . . . . . ::

PS C:\Users\steinam> _
```

1.3.4 Auto-Completing

Powershell kennt viele eingebaute Befehle, die natürlich schwer zu merken sind. Um Tippfehler auszuschließen, können Sie nach Eingabe der ersten Buchstaben die *TAB*-Taste drücken. Sie erhalten dann einen Vorschlag zur Vervollständigung. Weiteres Drücken der *TAB*-Taste gibt einen weiteren Vorschlag.

1.3.5 Tastenkürzel

Taste	Bedeutung
(ALT)+(F7)	Deletes the current command history
(PgUp), (PgDn)	Display the first (PgUp) or last (PgDn) command you used in current session
(Enter)	Send the entered lines to PowerShell for execution
(End)	Moves the editing cursor to the end of the command line
(Del)	Deletes the character to the right of the insertion point
(Esc)	Deletes current command line
(F2)	Moves in current command line to the next character corresponding to specified character
(F4)	Deletes all characters to the right of the insertion point up to specified character
(F7)	Displays last entered commands in a dialog box
(F8)	Displays commands from command history beginning with the character that you already entered in the command line
(F9)	Opens a dialog box in which you can enter the number of a command from your command history to return the command.
(Arrow Left/Right)	Move one character to the left or right respectively
(Arrow up, down)	Repeat the last previously entered command
(Home)	Moves editing cursor to beginning of command line
(Backspace)	Deletes character to the left of the insertion point
(Ctrl)+(C)	Cancels command execution
(Ctrl)+(End)	Deletes all characters from current position to end of command line
(Ctrl)+(Arrow right) left,	(Ctrl)+(Arrow right) Move insertion point one word to the left or right respectively
(Ctrl)+(Home)	Deletes all characters of current position up to beginning of command line
(Tab)	Automatically completes current entry, if possible

1.3.6 Fragen

- Wie startet man die Powershell
- Welche beiden Varianten der Powershell gibt es
- Welchen Nutzen hat die TAB-Taste
- Wie kann man sich die zuletzt eingegebenen Befehle anzeigen lassen
- Wo kann man Informationen im Internet zur Powershell finden
- Was ist Piping
- Was ist Redirection
- Wie kann man die Farbe in der Powershell-Shell ändern

KAPITEL 2

PowerShell im Interaktiv-Modus

Powershell hat 2 Gesichter: zum Einen kann es zum Erstellen von Skripten dienen, zum Anderen kann man damit sofort Befehle ausführen.

2.1 PS als Rechner

Powershell kann auf der Konsole wie ein normaler Rechner verwendet werden.

```
2 + 4 (Enter)  
6
```

Alle grundlegenden arithmetischen Operationen sowie Klammerrechnung sind möglich:

```
(12+5) * 3 / 4.5 (Enter)  
11.333333333333  
  
(Dir *.txt).Count (Enter)  
12
```

Ein Komma statt eines Punktes scheint ein falsches Ergebnis zu bringen

```
4,2 + 2 (Enter)  
  
4  
2  
2  
  
4.2 + 2 (Enter)  
  
6,2
```

Kommas erzeugen immer einen Array. Der Punkt dient bei Gleitkommazahlen als Trennzeichen

2.2 Rechnen mit Zahlensystemen und Einheiten

Eine interessante Erweiterung in **Powershell** ist die Möglichkeit, mit den gängigen Einheiten der IT-Branche direkt zu rechnen. Powershell unterstützt die Größen GB, MB, KB. Sie müssen direkt hinter dem Wert stehen

```
4GB / 720MB (Enter)  
5.6888888888889  
  
1kb
```

Powershell erkennt hexadezimale Zaheln, wenn Sie mit 0x eingeleitet werden

```
12 + 0xAF (Enter)  
187  
  
0xAFFE (Enter)  
45054
```

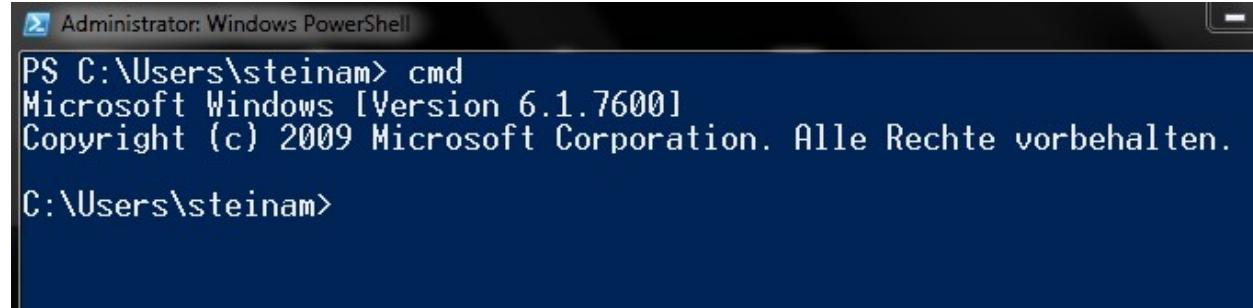
Siehe Übersicht arithmetische Operatoren. [Mathematische Operatoren](#):

2.3 Ausführen externer Befehle

Powershell kann wie die klassische Shell auch beliebige externe Befehle ausführen.

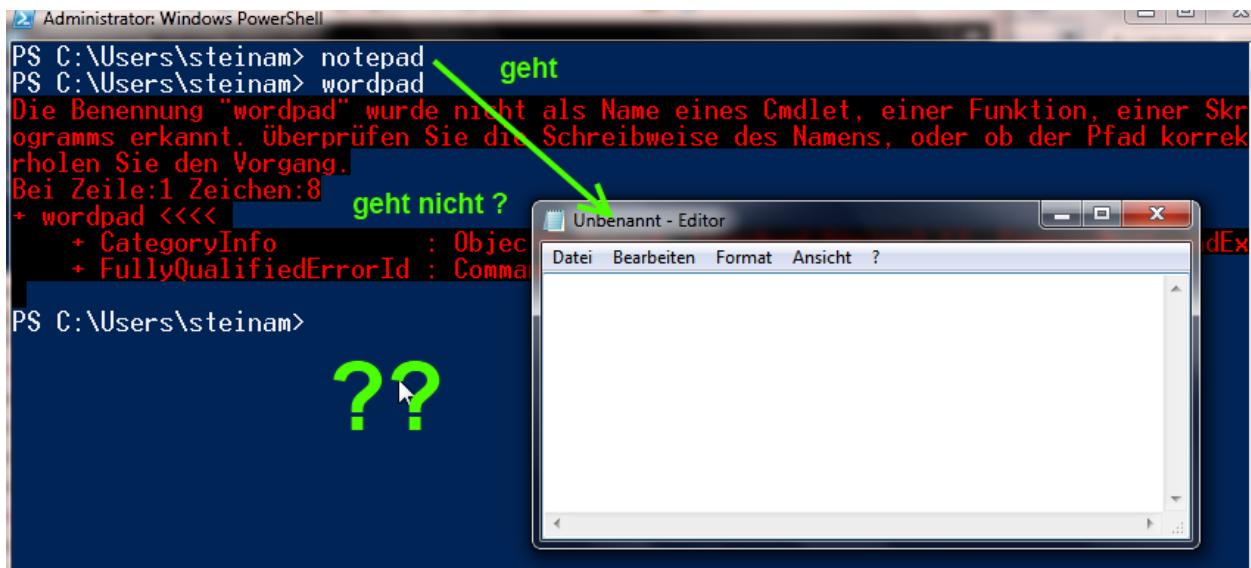
Selbst die alte Konsole kann gestartet werden.

```
cmd (Enter)
```



The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The title bar includes the standard window controls (minimize, maximize, close). The main area displays the command "PS C:\Users\steinam> cmd" followed by the output of the "cmd" command, which shows the Microsoft Windows [Version 6.1.7600] copyright information and the prompt "C:\Users\steinam>".

2.3.1 Sicherheitseinschränkungen



Wie die Abbildung zeigt, kann **Powershell** offenbar nicht jedes Programm starten. Powershell sucht die Programme in der PATH-Umgebungsvariable. Dies sind sog. „vertrauenswürdige“ Pfade.

2.4 Cmdlets: Powershell-eigene Befehle

Powershell's interne Befehle werden „cmdlet“ genannt. Sie bestehen häufig aus der Kombination eines Verbs mit einem Befehl, z.B. **Get-Command**.

```
Get-Command -commandType cmdlet
```

Folgende Tabelle listet die wichtigsten Möglichkeiten auf:

<i>Add</i>	<i>Export</i>
<i>Clear</i>	<i>Format</i>
<i>Compare</i>	<i>Get</i>
<i>Convert</i>	<i>Group</i>
<i>Copy</i>	<i>Import</i>
<i>Start</i>	<i>Measure</i>
<i>Stop</i>	<i>Move</i>
<i>Suspend</i>	<i>New</i>
<i>Tee</i>	<i>Out</i>
<i>Test</i>	<i>Read</i>
<i>Trace</i>	<i>Remove</i>
<i>Update</i>	<i>Rename</i>
<i>Write</i>	<i>Resolve</i>

Über das CmdLet **Get-Help** kann zu jedem Befehl eine ausführliche Hilfe herbeigeholt werden.

```
Get-Help Get-Command -detailed
```

2.5 Aliase

Powershell kennt für bestimmte Befehle Abkürzungen, was als Alias bezeichnet wird. Diese können anstelle der häufig langen Befehle eingegeben werden.

```
get-alias get-command
```

2.6 Dateien und Skripte aufrufen

Im Regelfall werden Powershell-Skripte in einem Editor geschrieben und entweder direkt darin ausgeführt oder die gespeicherten Skriptdateien über die Powershell-Konsole aufgerufen. Dem Aufruf der Skriptdatei muss in der Konsole der „.“ vorangehen bzw. der komplette Pfad zur Datei angegeben werden.

The screenshot displays two windows side-by-side. The top window is a standard Windows PowerShell window titled "Windows PowerShell". It shows the command PS C:\temp> .\hello.ps1 followed by the output "Geben Sie einen Namen ein: Steinam" and "Hello Steinam". The bottom window is a Windows PowerShell ISE window titled "Windows PowerShell ISE". It shows the menu bar "Datei", "Bearbeiten", "Ansicht", "Tools", "Debuggen", "Add-Ons", "Hilfe". Below the menu is a toolbar with various icons. A code editor window titled "hello.ps1" is open, containing the following PowerShell script:

```
1 $name = Read-Host "Geben sie einen Namen ein"
2
3 Write-Host "Hello $name"
```


KAPITEL 3

Variablen und Datentypen

Äeberblick

- Notwendigkeit von Variablen
- Notwendigkeit von Datentypen
- Beispielhafte Datentypen (int, string, double)
- Ergebnis von Datentypen und darauf wirkende Operatoren
- Casten (Umwandeln des Datentyps) einer Variable
- Konstante
- Automatische Powershell-Variablen
- Umgebungsvariablen
- GÄ¼ltigkeit von Variablen

Variablen speichern Informationen, um sie fÄ¼r andere Dinge zu benutzen. Sie sind somit ein Platzhalter/Stellvertreter fÄ¼r den gespeicherten Wert. Äeber den Namen der Variablen kann auf den Wert zugegriffen werden.

Variablen werden in der Powershell mit einem \$-Symbol eingeleitet. AnschlieÄÝend kann eine fast beliebige Zeichenfolge benutzt werden. Die Namen sind nicht case-sensitive, d.h. GroÄÝ- und Kleinschreibung wird nicht unterschieden.

```
$netto = 120
$MWStSatz = 0.19

$UST = $netto * $MWStSatz

$UST

$result = $netto + $UST

$text = "Nettowert $netto macht brutto $result"
```

```
$text  
  
$text = "Nettowert $netto macht brutto $netto + $USt"  
$text
```

```
Windows PowerShell  
PS C:\> $netto = 12  
PS C:\> $MWStSatz = 0.19  
PS C:\> $USt = $netto * $MWStSatz  
PS C:\> $USt  
2,28  
PS C:\> $result = $netto + $USt  
PS C:\> $text = "Nettowert $netto macht brutto $result"  
PS C:\> $text  
Nettowert 12 macht brutto 14.28  
PS C:\> $text = "Nettowert $netto macht brutto $netto + $USt"  
PS C:\> $text # was kommt raus ???
```

```
Windows PowerShell  
PS C:\> $netto = 12  
PS C:\> $MWStSatz = 0.19  
PS C:\> $USt = $netto * $MWStSatz  
PS C:\> $USt  
2,28  
PS C:\> $result = $netto + $USt  
PS C:\> $text = "Nettowert $netto macht brutto $result"  
PS C:\> $text  
Nettowert 12 macht brutto 14.28  
PS C:\> $text = "Nettowert $netto macht brutto $netto + $USt"  
PS C:\> $text  
Nettowert 12 macht brutto 12 + 2.28  
PS C:\> -
```

```
Windows PowerShell
PS C:\> $netto = 12
PS C:\> $MWStSatz = 0.19
PS C:\> $USt = $netto * $MWStSatz
PS C:\> $USt
2.28
PS C:\> $result = $netto + $USt
PS C:\> $text = "Nettowert $netto macht brutto $result"
PS C:\> $text
Nettowert 12 macht brutto 14.28
PS C:\> $text = "Nettowert $netto macht brutto $netto + $USt"
PS C:\> $text
Nettowert 12 macht brutto 12 + 2.28
PS C:\> $text = "Nettowert $netto macht brutto" + $netto + $USt
PS C:\> $text
Nettowert 12 macht brutto122.28
PS C:\> $text = "Nettowert $netto macht brutto" + ($netto + $USt)
PS C:\> $text
Nettowert 12 macht brutto14.28
PS C:\>
```

**Bemerkung:**

Nicht immer erhält man das, was man erwartet !!

Durch den + - Operator innerhalb des Strings werden die beiden Variablen nicht addiert, sondern hintereinander angefügt.

Die Klammer um die beiden Variablen sorgt dafür, dass erst die Berechnung innerhalb der Klammer ausgeführt wird, bevor die Übergabe an die \$text-Variablen erfolgt



3.1 Zuweisungsoperator

Mit Hilfe des Gleichheitszeichen = wird einer Variable ein Wert zugewiesen. Will man den Wert einer Variable erfahren, muss man nach Eingabe des Variablennamen lediglich <RETURN> drücken. Dabei wird man feststellen, dass eine Variable durchaus auch mehr Inhalt aufweisen kann.

```
$listing = Get-ChildItem c:\  
$listing  
Directory: Microsoft.PowerShell.Core\FileSystem::C:\  
Mode LastWriteTime Length Name  
---- ----- ---- --
```

d----	06.26.2007	15:36	2420
d----	05.04.2007	21:06	ATI
(..)			

3.2 Daten zwischen Variablen austauschen

Eine Variable kann auch den Wert einer anderen Variablen erhalten

```
$a = "Kurt"  
$b = "Sabine"  
  
$a = $b  
  
$a $b = $b $a
```

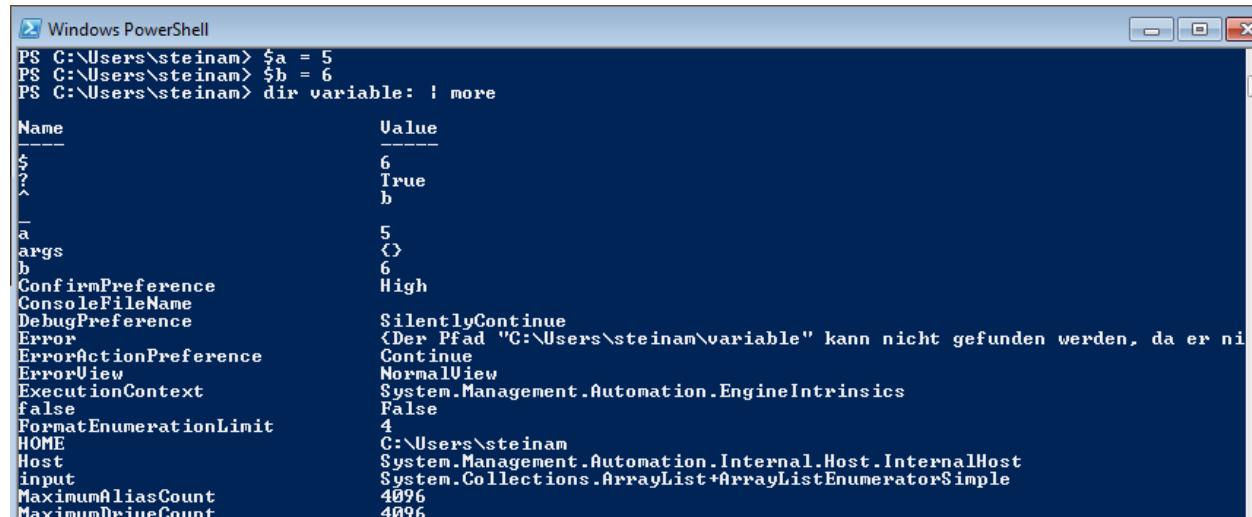
3.3 Variablen finden

Alle während einer **Powershell**-Session erzeugten Variablen bleiben erhalten, obwohl man sie durch das Scrollen des Bildschirms nicht mehr erkennen kann. Um sich einen Überblick über die derzeit benutzbaren Variablen zu verschaffen, kann man sich ein Listing des „*Variablen-Verzeichnisses*“ geben lassen.

```
Dir variable:
```

Ist man am Wert einer Variable interessiert kann man folgendes eingeben:

```
Dir variable:variablename*
```



Name	Value
---	---
6	6
?	True
x	b
a	5
args	<>>
b	6
ConfirmPreference	High
ConsoleFileName	SilentlyContinue
DebugPreference	<Der Pfad "C:\Users\steinam\variable" kann nicht gefunden werden, da er ni
Error	Continue
ErrorActionPreference	NormalView
ErrorView	System.Management.Automation.EngineIntrinsics
ExecutionContext	False
false	4
FormatEnumerationLimit	C:\Users\steinam
HOME	System.Management.Automation.Internal.Host.InternalHost
Host	System.Collections.ArrayList+ArrayListEnumeratorSimple
input	4096
MaximumAliasCount	4096
MaximumDriveCount	

Um zu überprüfen, ob eine Variable existiert, kann man mit Hilfe des Cmd-Lets *test-Path* prüfen, ob sie vorhanden ist.

```
# Verify whether the variable $value2 exists:  
Test-Path variable:\value2  
True  
# verify whether the variable $server exists:
```

```
Test-Path variable:\$server
False
```

Das Löschen einer Variablen läuft wie in normalen Verzeichnissen auch mit Hilfe des Befehls *del*

```
# create a test variable:
$test = 1
# verify that the variable exists:
Dir variable:\$te*
# delete variable:
del variable:\$test
# variable is removed from the listing:
Dir variable:\$te*
```

3.4 Konstanten

Normalerweise können Variablen immer wieder mit neuen Werten überschrieben werden. Sie sind aber nicht immer gewünscht, so z.B. bei dem Wert von Pi. Variable, deren Wert sich nicht ändern soll, werden **Konstante** genannt. Sie sind auch in Powershell zu definieren, allerdings etwas umständlich.

```
# Create new variable with description and write-protection:
New-Variable test -value 100 -description "test variable with write-protection" -
    -option ReadOnly
$test
100
# Variable contents cannot be modified:
$test = 200
The variable "test" cannot be overwritten since it is a constant or read-only.
At line:1 char:6
+ $test <<< = 200
```

Wurden Variablen auf diese Weise erzeugt, können Sie zumindest noch gelöscht werden. Um auch dies zu verhindern, kann man das New-Variable-Cmdlet mit der Option *constant* versehen.

```
#New-Variable cannot write over existing variables:
New-Variable test -value 100 -description "test variable with copy protection" -
    -option Constant
New-Variable : A variable named "test" already exists.
At line:1 Char:13
+ New-Variable <<< test -value 100 -description
"test variable with copy protection" -option Constant
# If existing variable is deleted, New-Variable can create
# a new one with the "Constant" option:
del variable:\$test -force
New-Variable test -value 100 -description "test variable with copy protection" -
    -option Constant
# variables with the "Constant" option may neither be
# modified nor deleted:
del variable:\$test -force
Remove-Item : variable "test" may not be removed since it is a constant or write-
    -protected. If the variable is write-protected, carry out the process with the Force_
    -parameter.
At line:1 Char:4
+ del <<< variable:\$test -force
```

3.5 „Automatische“ Powershell-Variablen

Beim Erzeugen einer **Powershell**-Session sind bereits einige Variablen vorbelegt. Eine Übersicht finden Sie hier [Powershell-Variablen](#)

Weitere Variablen finden Sie mit Hilfe des bereits oben erwähnten Dir-Befehls

```
dir variable:
```

3.6 Umgebungsvariablen

Häufig will man auch auf die Umgebungsvariablen des Betriebssystems zugreifen wollen. **Powershell** kennt dazu das Verzeichnis `env`, über welches man den Zugriff auf diese Werte bekommt.

```
dir env:
```

```
cd $env:homepath
```

```

Windows PowerShell
-a--- 10.06.2009 23:08      92 win.ini
-a--- 11.08.2010 15:10    1473901 WindowsUpdate.log
-a--- 14.07.2009 03:14     9728 winhlp32.exe
-a--- 14.07.2009 03:39    10240 write.exe

PS C:\Windows> dir env:

Name                           Value
---- 
ALLUSERSPROFILE                C:\ProgramData
APPDATA                         C:\Users\steinam\AppData\Roaming
CommonProgramFiles               C:\Program Files\Common Files
CommonProgramFiles(x86)          C:\Program Files (x86)\Common Files
CommonProgramW6432              C:\Program Files\Common Files
COMPUTERNAME                     STEINAM-WINUBOX
ComSpec                          C:\Windows\system32\cmd.exe
FP_NO_HOST_CHECK                 NO
HOMEDRIVE                        C:
HOMEPATH                         \Users\steinam
LOCALAPPDATA                      C:\Users\steinam\AppData\Local
LOGONSERVER                       \\STEINAM-WINUBOX
NUMBER_OF_PROCESSORS             1
OS                               Windows_NT
Path                             %SystemRoot%\system32\WindowsPowerShell\v1.0\*.COM;*.EXE;*.BAT;*.CMD;*.VBS;*.UBE;*.JS;*.JSE;*.WSF
PROCESSOR_ARCHITECTURE           AMD64
PROCESSOR_IDENTIFIER              Intel64 Family 6 Model 30 Stepping 5, Genuine
PROCESSOR_LEVEL                  6
PROCESSOR_REVISION                1e05
ProgramData                      C:\ProgramData
ProgramFiles                     C:\Program Files
ProgramFiles(x86)                C:\Program Files (x86)
ProgramW6432                      C:\Program Files
PSModulePath                     C:\Users\steinam\Documents\WindowsPowerShell\Modules
PUBLIC                            C:\Users\Public
SESSIONNAME                      Console
SystemDrive                       C:
SystemRoot                        C:\Windows
TEMP                             C:\Users\steinam\AppData\Local\Temp
TMP                              C:\Users\steinam\AppData\Local\Temp
USERDOMAIN                        steinam-winubox
USERNAME                          steinam
USERPROFILE                       C:\Users\steinam
VS100COMNTOOLS                     C:\Program Files (x86)\Microsoft Visual Studio
windir                           C:\Windows

PS C:\Windows> cd $env:homepath
PS C:\Users\steinam>

```

Umgebungsvariablen können wie *normale* Variablen angepasst und hinzugefügt werden. Diese bleiben allerdings nur in der jeweiligen Powershell-Sitzung erhalten.

Um sie permanent zu ändern, muss man das .NET-Framework zu Hilfe nehmen.

```
$oldValue = [environment]::GetEnvironmentvariable("Path", "User")
$newValue = ";c:\myTools"
[environment]::SetEnvironmentvariable("Path", $newValue, "User")
```

3.7 Gültigkeit von Variablen

Unter Gültigkeit wird verstanden, wann eine Variable für **Powershell** nutzbar ist.

Erläutern Sie die Ausgabe in folgender Abbildung:

The screenshot shows two windows. The top window is 'Administrator: Windows PowerShell' with the command PS C:\Windows\system32> notepad .\test.ps1 followed by the output of the script. The bottom window is 'test - Editor' showing the contents of the file 'test.ps1' which contains '\$windows = \$env:windir' and 'Windows Folder: \$windows'.

```
PS C:\Windows\system32> notepad .\test.ps1
PS C:\Windows\system32> $windows = "hello"
PS C:\Windows\system32> .\test.ps1
Windows Folder: C:\Windows
PS C:\Windows\system32> $windows
hello
PS C:\Windows\system32>

test - Editor
Datei Bearbeiten Format Ansicht ?
$windows = $env:windir
"Windows Folder: $windows"
```

Powershell erzeugt für jedes Skript eine eigene Variablenumgebung, die zunächst nichts miteinander zu tun hat. Der Aufruf des Skripts innerhalb der gleichen Umgebung mit Hilfe von *.test.ps1* führte zum Erzeugen einer neuen Umgebung.

Erst der richtige Aufruf innerhalb der Konsole führt dazu, dass das Skript gleichsam in der eigenen Variablenumgebung lautet. Finden Sie den kleinen Unterschied zum ersten Bild ??

The screenshot shows two windows. The top window is 'Administrator: Windows PowerShell' with the command PS C:\Windows\system32> notepad .\test.ps1 followed by the output of the script. The bottom window is 'test - Editor' showing the contents of the file 'test.ps1' which contains '\$windows = \$env:windir' and 'Windows Folder: \$windows'. The output in the PowerShell window shows that the variable \$windows is now defined in the current session.

```
PS C:\Windows\system32> notepad .\test.ps1
PS C:\Windows\system32> $windows = "hello"
PS C:\Windows\system32> .\test.ps1
Windows Folder: C:\Windows
PS C:\Windows\system32> $windows
C:\Windows
PS C:\Windows\system32>

test - Editor
Datei Bearbeiten Format Ansicht ?
$windows = $env:windir
"Windows Folder: $windows"
```

Erst wenn das Skript „dot-sourced“ aufgerufen wird, befindet es sich in der gleichen Variablenumgebung wie die Konsole und benutzt nun die gleichen Variablen.

The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell" and a Microsoft Word document titled "test - Editor". The PowerShell session shows the execution of a PowerShell script named "test.ps1". The script contains the following code:

```

PS C:\Windows\system32> notepad .\test.ps1
PS C:\Windows\system32> $windows = "hello"
PS C:\Windows\system32> .\test.ps1
Windows Folder: C:\Windows
PS C:\Windows\system32> $windows
C:\Windows
PS C:\Windows\system32>

```

The line ".\test.ps1" is highlighted with a red box. The Microsoft Word document shows the contents of the script:

```

$windows = $env:windir
"Windows Folder: $windows"

```

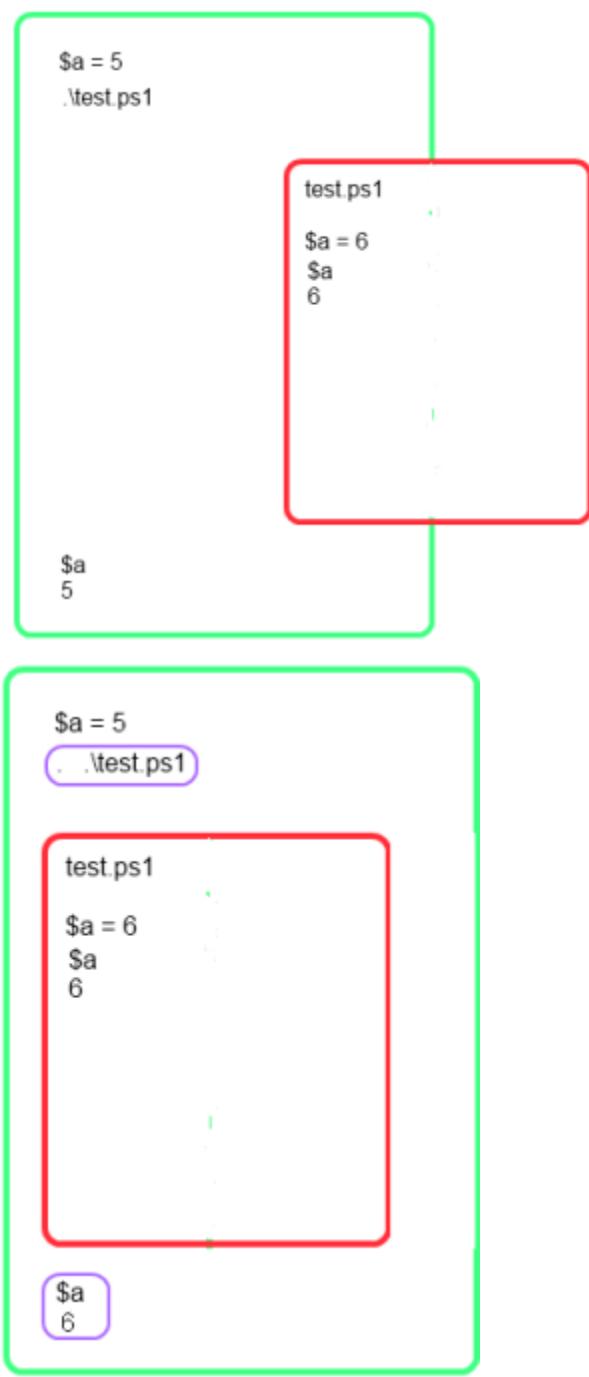
Die Gültigkeit von Variablen war mit den bisherigen Verfahren entweder als „ganz oder gar nicht“ zu bezeichnen.

Powershell kennt noch eine andere Art, um die Gültigkeit von Variablen zu definieren., nämlich mit Hilfe der Modifier **private**, **local**, **script**, **global**.

Scope allocation	Description
\$private:test = 1	The variable will be created only in the current scope and not passed to other scopes. Consequently, it can only be read and written in the current scope.
\$local:test = 1	Variables will be created only in the local scope. That is the default for variables that are specified without a scope. Local variables can be read from scopes originating from the current scope, but they cannot be modified.
\$script:test = 1	The variable is valid only in a script, but valid everywhere in it. Consequently, a function in a script can address other variables, which, while defined in a script, are outside the function.
\$global:test = 1	The variable is valid everywhere, even outside functions and scripts.

Bemerkung: Tafelbild

Tafelbild zu Scope



3.8 Typisiert vs. untypisiert

Die Werte, die einer Variable zugewiesen werden können, stellen häufig einen numerischen Wert oder einen Zeichenwert dar. Man spricht deshalb vom **Typ** einer Variable, bzw. dem sog. **Datentyp**. Dies kann z.B. eine Ganzzahl (int) oder eine Gleitkommazahl (double) sein.

Wichtig an dieser Tatsache ist, dass die Festlegung einer Variable auf einen bestimmten Typ Auswirkungen auf die weitere Benutzung hat. Insbesondere bei der Verbindung mit **Operatoren** erhält man manchmal überraschende Ausgaben, wie die obigen Bilder gezeigt haben.

Manche Programmiersprachen legen Wert darauf, dass die Variablen bei der Deklaration einen Datentyp **explizit** zugewiesen bekommen müssen (sog. typisierte Programmiersprachen, z.B. c#). Andere Programmiersprachen machen diese Zuweisung zu einem Datentyp **implizit**, d.h. ohne Zutun des Programmierers wird einer Variablen ein Datentyp zugewiesen.

Powershell lässt zunächst eine untypisierte Schreibweise zu; intern arbeitet es aber mit der typisierten Form der Variable.

Um den Datentyp einer Variablen zu erhalten, kann man hinter die Variable den Befehl GetType().Name stellen. Beispiel:

```
(12) .GetType() .Name
Int32
(10000000000000) .GetType() .Name
Int64
(12.5) .GetType() .Name
Double
(12d) .GetType() .Name
Decimal
("H") .GetType() .Name
String
(Get-Date) .GetType() .Name
DateTime
```

Powershell sucht den am besten passenden Datentyp automatisch heraus, was als sog. *weakly typed* bezeichnet wird. Dies kann auch negative Konsequenzen haben, weshalb man häufig explizit einen Datentyp angibt.

Viele Datentypen haben auch ihre eigenen Hilfsfunktionen, die nur bei der korrekten Zuweisung des Datentyps vorhanden sind.

Beispiel:

```
$date = "November 12, 2010"
$date

November 12, 2004

# als datetime bietet die Variable mehr Möglichkeiten

[datetime]$date = "November 12, 2004"
$date
Friday, November 12, 2004 00:00:00

$date.AddDays(60)
Tuesday, January 11, 2005 00:00:00

#auch xml-Dateien sind plausibel sehr einfach
# PowerShell stores a text in XML format as a string:
$t = "<servers><server name='PC1' ip='10.10.10.10' />" +
"<server name='PC2' ip='10.10.10.12' /></servers>"
$t
<servers><server name='PC1' ip='10.10.10.10' />
```

```
<server name='PC2' ip='10.10.10.12' /></servers>
# If you assign the text to a data type[xml], you'll
# suddenly be able to access the XML structure:
$xml = $t
$xml.servers
server
-----
{PC1, PC2}

$xml.servers.server
name          ip
---          --
PC1          10.10.10.10
PC2          10.10.10.12
# Even changes to the XML contents are possible:
$xml.servers.server[0].ip = "10.10.10.11"
$xml.servers
name          ip
---          --
PC1          10.10.10.11
PC2          10.10.10.12
# The result could be output again as text, including the
# modification:
$xml.get_InnerXML()
<servers><server name="PC1" ip="10.10.10.11" />
<server name="PC2" ip="10.10.10.12" /></servers>
```

3.9 Typ erzwingen

PowerShell kennt im Gegensatz zu C# keine Datentypen und ist damit untypisiert; man kann dieses Verhalten aber erzwingen, indem man den Datentyp in eckigen Klammern voranstellt.

```
PS> $a = "hello" # Implizites Zuweisen des Datentyps string zur Variable
[int]$b = 5 # Explizites Zuweisen des Datentyps int

$a.GetType();
IsPublic IsSerial Name          BaseType
-----  -----  -----
True     True      String       System.Object

$b.GetType();
IsPublic IsSerial Name          BaseType
-----  -----  -----
True     True      Int32       System.ValueType
```

Powershell kennt und benutzt die Datentypen des .NET-Frameworks. Eine Übersicht über gängige Datentypen und deren Eigenschaften finden Sie unter <http://msdn.microsoft.com/de-de/library/1dhd7f2x.aspx>

3.10 Aufruf von Befehlen über eine Variable

You can use the PowerShell call operator „&“. Here is a simple example:

```
$a = „Get-Process“ & $a
```

There is a caveat with using this since the call operator can only operate against a single command. If you were to modify the above like this:

```
$a = „Get-Process w*“ & $a
```

you will get an error. To get around this use Invoke-Expression as follows:

```
$a = „Get-Process w*“ |Invoke-Expression $a
```

See Get-Help Invoke-Expression -examples to get more examples on the usage of Invoke-Expression.

3.10.1 Powershell-Datentypen

Typ	Beschreibung
array	Liste von Werten
bool	Ja-Nein-Wert
char	Unicode Zeichen
byte	Integer 8 Bit
datetime	Datumswert
decimal	Dezimal
double	Gleitkomma
guid	Eindeutige Zahl
hashtable	
int 16/32/64	Ganzzahl
sbyte	
single	
string	Zeichenkette
timespan	Zeitspanne
xml	
....	

3.11 Zusammenfassung

- Variablen speichern jede Art von Information
- Sie beginnen immer mit dem „\$“-Zeichen. Anschließend können Sie aus alphanumerischen und Sonderzeichen (z.B. Underscore) bestehen
- Groß- und Kleinschreibung spielt keine Rolle
- „Konstante“ Variablen ändern ihren Wert nicht mehr
- **Powershell** hat schon selbst gewisse Variablen definiert (automatische Variablen)
- **Powershell** sorgt dafür, dass der Typ einer Variable zum Typ des zu speichernden Wertes passt (weakly typing)
- **Powershell** unterstützt Strong-Typing durch Voranstellen des Datentyps in eckigen Klammern []

- **Powershell** erstellt die Variablen jeweils innerhalb eines gewissen Gültigkeitsbereichs (local, private, script, global)
- An Umgebungsvariablen kann durch \$env: zugegriffen werden

Powershell-Datentypen

KAPITEL 4

Operatoren

Variablen werden häufig mit Hilfe von Operatoren weiterverarbeitet. Die gängigsten Operatoren sind die mathematischen Operatoren wie Addition, Multiplikation, Division,etc. Im Anhang finden Sie eine Übersicht *Mathematische Operatoren*.

4.1 Powershell-Variablen

Variable	Bedeutung
\$^	Contains the first token of the last line input into the shell
\$\$	Contains the last token of the last line input into the shell
\$_	The current pipeline object; used in script blocks, filters, Where-Object, ForEach-Object, and Switch
\$?	Contains the success/fail status of the last statement
\$Args	Used in creating functions requiring parameters
\$Error	If an error occurred, the error object is saved in the \$error variable.
\$Execution-Context	The execution objects available to cmdlets
\$foreach	Refers to the enumerator in a foreach loop
\$HOME	The user's home directory; set to %HOMEDRIVE% %HOMEPATH%
\$Input	Input is piped to a function or code block.
\$Match	A hash table consisting of items found by the -match operator
\$MyInvocation	Information about the currently executing script or command-line
\$PSHome	The directory where PS is installed
\$Host	Information about the currently executing host
\$LastExitCode	The exit code of the last native application to run
\$true	Boolean TRUE
\$false	Boolean FALSE
\$null	A null object
\$this	In the Types.ps1xml file and some script block instances, this represents the current object
\$OFS	Output Field Separator used when converting an array to a string
\$ShellID	The identifier for the shell. This value is used by the shell to determine the ExecutionPolicy and what profiles are run at Startup
\$StackTrace	Contains detailed stack trace information about the last error

4.2 Vergleichsoperatoren

Operator	klassisch	Beschreibung	Beispiel	Ergebnis
-eq,-ceq, -ieq	=	Gleichheit	10 -eq 15	\$true
-ne, -cne, -ice	<>	Ungleichheit	10 -ne 15	\$true
-gt,-cgt, -igt	>	Größer	10 -gt 15	\$false
-ge,-cge, -ige	>=	Größer gleich	10 -ge 15	\$false
-lt,-clt, -ilt	<	Kleiner	10 -lt 15	\$true
-le,-cle, -ile	<=	Kleiner gleich	10 -le 15	\$true
-contains [#]		Enthält	1,2,3 -contains 1	\$true
-notcontains		Nicht enthält	1,2,3 -notcontains 1	\$false
-is		Typgleichheit	\$feld -is [array]	\$true
-like		Wildcard		
-notlike		Wildcard		
-match		regulärer Ausdruck		
-notmatch		regulärer Ausdruck		

4.3 Logische Operatoren

Operator	Description	Left Value	Right Value	Result
-and	Both conditions must be met	True False False True	False True False True	False False False True
-or	At least one of the two conditions must be met	True False False True	False True False True	True True False True
-xor	One or the other condition must be met, but not both	True False False True	True False True False	False False True True
-not	Reverses the result	(not applicable)	True False	False True

4.4 Mathematische Operatoren

Es gelten folgende Voraussetzungen

```
PS> $a = 10
      $b = 15
```

Operator	klassisch	Beispiel	Ergebnis
Addition	+	\$a + \$b	25
Division	-	\$a - \$b	-5
Multiplikation	*	\$a * \$b	\$false
Modulo <u>[#]</u>	%	\$a % 3	1
Increment	++	\$a++	11
Decrement	--	\$b--	14

Operator	Description	example	result
+	Adds two values	5 + 4.5	9.5
		2gb + 120mb	2273312768
		0x100 + 5	261
		"Hello " + "there"	"Hello there"
-	Subtracts two values	5 - 4.5	0.5
		12gb - 4.5gb	8053063680
		200 - 0xAB	29
*	Multiplies two values	5 * 4.5	22.5
		4mb * 3	12582912
		12 * 0xC0	2304
		"x" * 5	"xxxxx"
/	Divides two values	5 / 4.5	1.111111111111111 1
		1mb / 30kb	34.13333333333333 3
		0xFFAB / 0xC	5454,25
%	Supplies the rest of division	5%4.5	0.5

KAPITEL 5

Diagramme der strukturierten Programmierung

Zur Darstellung von Sachverhalten der strukturierten Programmierung wurden vor vielen Jahren verschiedene Notationsformen entwickelt. Die wichtigsten sind der **Programmablaufplan** und das **Struktogramm**.

5.1 Programmablaufplan



5.2 Struktogramm

Ein Programm besteht aus einer Folge von Anweisungen, die häufig nicht nur sequenziell(hintereinander) ablaufen, sondern die wiederholt oder evtl. auch gar nicht ausgeführt werden sollen. Viele Problemstellungen sind umfangreich und kompliziert und erfordern deshalb eine systematische Vorarbeit. Beim Entwurf werden grafische Darstellungsmittel für die Logik des Programmablaufes verwendet.

Die Logik eines Programmablaufes kann mit den untenstehenden Konzepten realisiert werden.

- Anweisung
- Auswahl
- Wiederholung

5.2.1 Anweisung

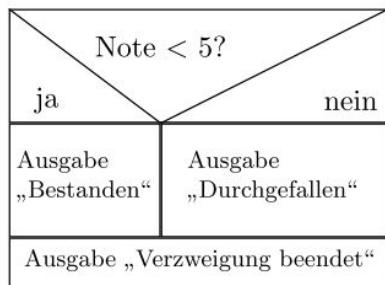
Die Anweisung ist der grundlegende Befehl innerhalb einer Programmiersprache. Im Struktogramm wird er durch ein Rechteck abgebildet. Durch die Folge von Anweisungen entsteht ein sequentieller Ablauf eines Programms.

5.2.2 Einseitige/Zweiseitige Auswahl

Ein Programm kann, abhängig von einem Kriterium, eine Entweder-oder-Entscheidung treuen, d.h. entweder einen bestimmten Programmteil „A“ ausführen oder einen anderen bestimmten Programmteil „B“. Der nötige Code lässt sich sehr intuitiv verstehen. Sehen wir uns als Beispiel den Code an, der, abhängig vom Wert der Variablen note, den Text „Bestanden“ oder „Durchgefallen“ ausgibt:

```
if ($note -lt 5) {
    Write-Host ("Bestanden");
} else {
    Write-Host ("Durchgefallen");
}
Write-Host ("Verzweigung beendet");
```

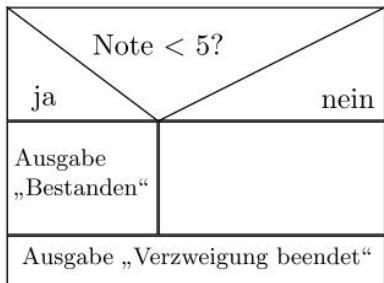
Zuvor steht ein neues Schlüsselwort: if (wenn). Dann folgt in Klammern das Kriterium: „Wenn note kleiner als 5 ist“. Wenn das der Fall ist, werden alle Anweisungen in dem folgenden, von geschweiften Klammern eingeschlossenen Block ausgeführt. Dann folgt ein weiteres Schlüsselwort else (sonst), gefolgt von einem weiteren Block, der ausgeführt wird, wenn note eben nicht kleiner als 5 ist. Danach ist die Verzweigung beendet, das heißt, der folgende Code wird auf jeden Fall ausgeführt und hängt nicht mehr von der if-Bedingung ab. Im Struktogramm sieht das so aus:



Man kann auch den else-Teil komplett weglassen. Wenn man zum Beispiel die Meldung „Durchgefallen“ nicht braucht, kann man schreiben:

```
if ($note -lt 5) {
    Write-Host("Bestanden");
}
Write-Host("Verzweigung beendet");
```

Das ist die sogenannte Einseitige Auswahl im Gegensatz zur Zweiseitigen Auswahl, die wir eben kennengelernt haben. Das Struktogramm hat jetzt folgendes Aussehen:



Wenn wir anders herum den „ja-Zweig“ (korrekt würde man sagen: if-Zweig) weglassen und nur den „nein-Zweig“ (korrekt: else-Zweig) implementieren wollen, wird es etwas schwieriger. Wir wandeln unser Beispiel so ab, dass nur der Text „Durchgefallen“ ausgegeben werden soll. Möglich wäre folgendes:

```
if ($note -lt 5) { //kein guter Stil
} else {
    Write-Host("Durchgefallen");
}
```

Das läuft zwar, ist aber kein guter Stil. Besser ist es, die Bedingung umzudrehen:

```
if ($note -ge 5) {
    Write-Host("Durchgefallen");
}
```

Kommen wir schließlich noch zur Bedingung, die in den Klammern steht. Was darin steht, ist das Ergebnis der Rechenoperation `note<5`. Die Rechenoperation ist eine sogenannte Vergleichsoperation und das Ergebnis ist ein Wahrheitswert. Ein Wahrheitswert kann nur die beiden Werte wahr und falsch annehmen.

Die önende und die schließende geschweifte Klammer darf in der if-Anweisung weggelassen werden. Dies ist jedoch immer schlechter Stil und kann leicht zu schwer auffindbaren Fehlern führen. Werden die Klammern weggelassen, besteht der if-Block aus der Zeile, die der if-Anweisung folgt und der else-Block besteht aus der Zeile, die der else-Anweisung folgt. Beispiel:

```
if ($wert%2 -eq 0)
    Write-Host("Die Zahl ist gerade");
```

Die Gefahr darin zeigt sich in folgendem Codeausschnitt:

```
if ($wert%2 -eq 0)
    Write-Host("Die Zahl ist gerade");
    Write-Host("Die Zahl ist durch 2 teilbar");
```

Entgegen dem Anschein wird die zweite WriteLine-Zeile auch bei ungeraden Zahlen ausgeführt, denn Java verwendet wegen der fehlenden geschweiften Klammern nur die erste WriteLine-Zeile für den if-Block. Dies ist ein nachträglich schwer zu ndender Fehler, der von vornherein vermieden werden kann, wenn man konsequent Klammern für den if und den else-Block setzt. Ein weiteres schwer aundbarer Fehler ist:

```
if ($wert%2 -eq 0);
    Write-Host("Die Zahl ist gerade");
```

Diesen Code muss man folgendermaßen interpretieren: Falls die Bedingung wahr ist, werden die Anweisungen bis zum nächsten Semikolon ausgeführt, also bis zum Semikolon am Ende der if-Zeile. Anschließend ist die if-Verweigung zu Ende. Die WriteLine-Anweisung wird also immer ausgeführt, gleichgültig ob wert gerade oder ungerade ist.

if-else-Kaskaden

In einem Sonderfall lässt man teilweise die geschweiften Klammern aber doch weg. Manchmal gibt es mehr als zwei Fälle, die unterschiedlich behandelt werden müssen. In diesem Fall schachtelt man mehrere if-Anweisungen ineinander und erhält die sogenannte if-else-Kaskade. Im nachfolgenden Beispiel bauen wir die Ausgabe einer Schulnote so weit aus, dass für jede Note ein individueller Text ausgegeben wird.

```
if ($note -eq 1) {
    Write-Host("sehr gut");
} else if ($note -eq 2) {
    Write-Host("gut");
} else if ($note -eq 3) {
    Write-Host("befriedigend");
} else if ($note -eq 4) {
    Write-Host("ausreichend");
} else if ($note -eq 5) {
    Write-Host("mangelhaft");
} else {
    Write-Host("Fehler im Programm");
}
```

5.2.3 Mehrseitige Auswahl

In allen neueren Sprachen, gibt es eine Verzweigung, die, abhängig von einer Integer-Variablen, einen von mehreren Programmblöcken anspringt. Das Notenprogramm, das im vorigen Kapitel mit einer if-else-Kaskade gelöst wurde, ist ein gutes Beispiel dafür. Das Struktogramm dazu ist:

		note					
		1	2	3	4	5	sonst
1	Ausgabe „sehr gut“	Ausgabe „gut“	Ausgabe „befriedi- gend“	Ausgabe „ausrei- chend“	Ausgabe „mangel- haft“	Ausgabe „Programm- fehler“	

switch-Anweisung

Die entsprechende Anweisung heißt in C# switch-Anweisung. In anderen Sprachen ist sie als case- oder select-Anweisung bekannt. Sie beginnt mit einer Zeile switch, gefolgt von der Variablen, deren Wert für die Verzweigung herangezogen wird:

```
switch ($note) {
```

Dann folgt für jede Note ein sogenannte case-Block. Die erste Zeile eines case- Blocks wird eingeleitet durch den Vergleichswert. Dann kommen die Anweisungen für den Block. Es gibt keine geschweiften Klammern. Ein Block wird mit dem Befehl break abgeschlossen.

```

switch(note) {
    1 {
        Write-Host("sehr gut");
        break;
    }
    2 {
        SWrite-Host("gut");
        break;
    }
    3 {
        Write-Host("befriedigend");
        break;
    }
    4 {
        Write-Host("ausreichend");
        break;
    }
    5 {
        Write-Host("mangelhaft");
        break;
    }
}
default{
    Write-Host("Fehler.");
}
}//switch

```

Am Ende der switch-Anweisung darf man noch einen sogenannten default-Block unterbringen, der immer dann ausgeführt wird, wenn keiner der vorigen case- Blöcke zutreend war. Man kann ihn auch weglassen. Dann wird statt dessen der switch-Block komplett übersprungen. Die switch-Anweisung hat ihre Tücken. Vor allem darf man das break am Ende nicht vergessen. Man kann das so verstehen: Die case- Zeilen sind An- sprungstellen. Das heißt, wenn jetzt zum Beispiel die Note gleich 2 ist, wird die Zeile mit case 2 angesprungen. Dann läuft das Programm Zeile für Zeile weiter. Wird ein break erreicht, springt das Programm aus dem switch-Block heraus. Haben wir jetzt beispielsweise das break nach case 2 vergessen, dann läuft das Programm einfach Zeile für Zeile weiter, bis der switch-Block zu Ende ist oder ein break erreicht wurde. In unserem Beispiel würde dann

gut befriedigend ausgegeben.

Diesen Eekt kann man durch geschickte Programmierung auch ausnutzen und Zweige für ganze Bereiche denieren. Im folgenden Beispiel wird bei den Noten 1-4 der Text „bestanden“ ausgegeben.

```

switch(note) {
    case 1:
    case 2:
    case 3:
    case 4: Write-Host("bestanden");
    break;
    case 5: Write-Host("mangelhaft");
    break;
    default: Write-Host("Fehler.");
}
}//switch

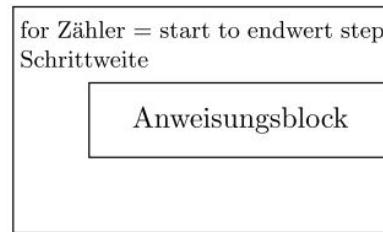
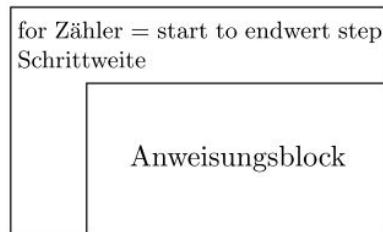
```

5.2.4 Wiederholung

Zählschleife

Die Zählschleife ist eine Schleifenart, bei der von Anfang an feststeht, wieviele Wiederholungen der Schleife ausgeführt werden. Es gibt einen Zähler (Laufvariable) der von einem Anfangswert bis zu einem Endwert läuft und sich bei

jedem Durchlauf um einen festen Betrag ändert. Das Struktogramm der Zählschleife ist:



Zählschleifen werden in Java mit dem Schlüsselwort *for* eingeleitet.

```

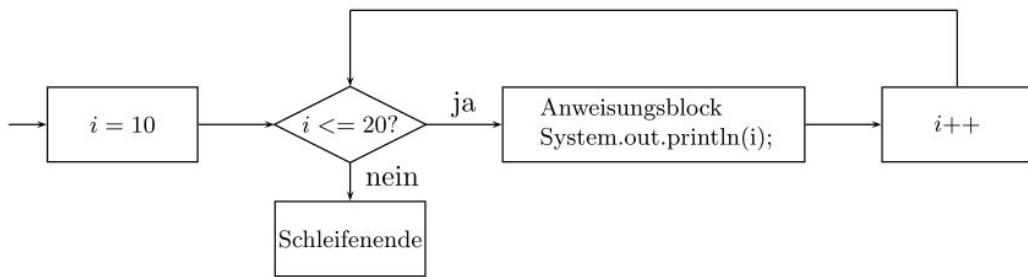
for ($i=10; $i<=20; i++) {
    Write-Host(i);
}

```

Dem Schlüsselwort folgt eine Parameterliste, die in einer runden Klammer zusammengefasst ist und aus 3 Teilen besteht, die jeweils durch ein Semikolon getrennt sind. Die 3 Teile sind:

1. Initialisierung der Laufvariablen.
2. Abbruchbedingung (kein Abbruch, solange die Bedingung erfüllt ist).
3. Veränderung der Laufvariablen.

Die Reihenfolge, in der die Teile abgearbeitet werden, veranschaulicht das folgende Flussdiagramm:



Änderung der Laufvariablen im Schleifenkörper

Es ist möglich, die Laufvariable im Schleifenkörper, also zwischen den geschweiften Klammern, zu ändern. Ein Beispiel dafür ist:

```

for ($i = 10; $i -gt 0; i++) {
    $i=$i-2;
    Write-Host($i);
}

```

Damit ist die Schleife aber keine reine Zählschleife mehr. In anderen Sprachen (z.B. Pascal) ist das auch verboten.

Deklaration der Laufvariablen in der Schleife

Es ist möglich, die Laufvariable im Schleifenkopf zu deklarieren:

```
for ($i=10; $i -gt 0; i--)
```

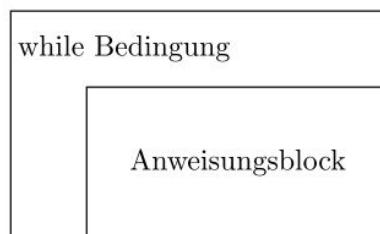
Dann ist die Laufvariable nur in der Schleife gültig und kann nach Beendigung der Schleife nicht mehr angesprochen werden. Diese Form ist die üblichste Form einer for-Schleife.

Schleife mit Anfangsabfrage (Kopfgesteuerte Schleife)

Bei manchen Schleifen steht zu Anfang die Anzahl der Durchläufe noch nicht fest. Es kann sein, dass es mehrere Abbruchbedingungen gibt oder dass die Laufvariable ihre Werte unvorhersehbar verändern kann. Hier benutzt man entweder die kopfgesteuerte oder die fußgesteuerte Schleife. Die kopfgesteuerte Schleife hat das Aussehen

```
while (Bedingung) {
    Anweisungs-Block
}
```

Das bedeutet, dass der Anweisungsblock ausgeführt wird, solange die Bedingung in der while-Zeile den Wert true ergibt. Das entsprechende Struktogramm hat das Aussehen:



Wir nehmen ein Countdown-Programm, dass von 100 rückwärts bis 10 zählt, aber alle durch 7 teilbaren Zahlen auslässt:

```
$i=100;
while ($i -gt 10) {
    Write-Host($i);
    $i--;
    if ($i%7 -eq 0) {
        $i--; //Durch 7 teilbare Zahlen überspringen
    }
}
```

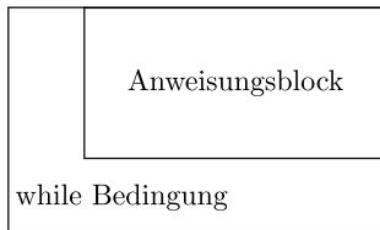
Schleife mit Endabfrage (Fußgesteuerte Schleife)

Die Schleife mit Endabfrage ähnelt der Schleife mit Anfangsabfrage. Allerdings wird die fußgesteuerte Schleife mindestens einmal durchlaufen, während demgegenüber die kopfgesteuerte Schleife gar nicht durchlaufen wird, wenn die Anfangsbedingung vor dem 1. Durchlauf nicht erfüllt ist. Die Schleife mit Endabfrage hat folgendes Aussehen:

```
do {
    //Anweisungsblock
} while (Bedingung)
```

Das Struktogramm ist:

Das Countdown-Beispiel hat hier folgendes Aussehen:



```
$i=10;
do {
    Write-Host($i);
    $i--;
} while ($i -gt 0);
```

In C und Java wird diese Schleife auch do-while-Schleife genannt, in Unterscheidung zur kopfgesteuerten while-Schleife. In Pascal spricht man stattdessen von einer repeat-until-Schleife. Da die fußgesteuerte Schleife sich immer mit einer kopfgesteuerten nachbilden lässt, besitzen manche Sprachen (z.B. Fortran, Python) keine fußgesteuerte Schleife.

continue

Innerhalb einer Schleife kann mit dem Befehl continue der aktuelle Schleifen- durchlauf abgebrochen werden, d.h. das Programm wird mit dem Beginn des nächsten Schleifendurchlaufs fortgesetzt. Die continue-Anweisung kann in allen Schleifenvarianten benutzt werden. Das folgende Countdown-Programm ist mit continue so abgewandelt, dass die Zahl 3 ausgelassen wird.

```
for ($i=10; $i -gt 0; $i--) {
    if ($i -eq 3) {
        continue;
    }
    Write-Host($i);
}
```

break

Der break-Befehl bewirkt, dass eine Schleife komplett abgebrochen wird. Das Programm

```
for ($i=10; $i -gt 0; $i--) {
    if ($i -eq 3) {
        break;
    }
    Write-Host($i);
}
```

zählt nur bis zur Zahl 4 herunter.

break und continue wirken sich nur auf die jeweilige Schleife aus, in der sie definiert wurden.

Mehrere verschachtelte Schleifen

Mehrere verschachtelte Schleifen sind möglich, wie am nachfolgenden Beispiel mehrerer verschachtelter for-Schleifen zu sehen ist. Es gibt ein Dreieck aus Sternen aus:

```
*
**
***
****
```

```
*****  
  
$max=5;  
for ($i=0; i -lt $max; $i++) {  
    for ($j = 0; $j -lt $i; $j++) {  
        Write-Host ("*");  
    }  
    Write-Host ();  
} //for i
```


KAPITEL 6

Bedingungen

Häufig wird man nicht einen *linearen* Quellcode schreiben, sondern in Abhängigkeit von bestimmten Bedingungen den einen oder den anderen Quellcode ausführen. Powershell kennt mehrere Möglichkeiten für das Prüfen von Bedingungen.

6.1 Einfache Vergleiche

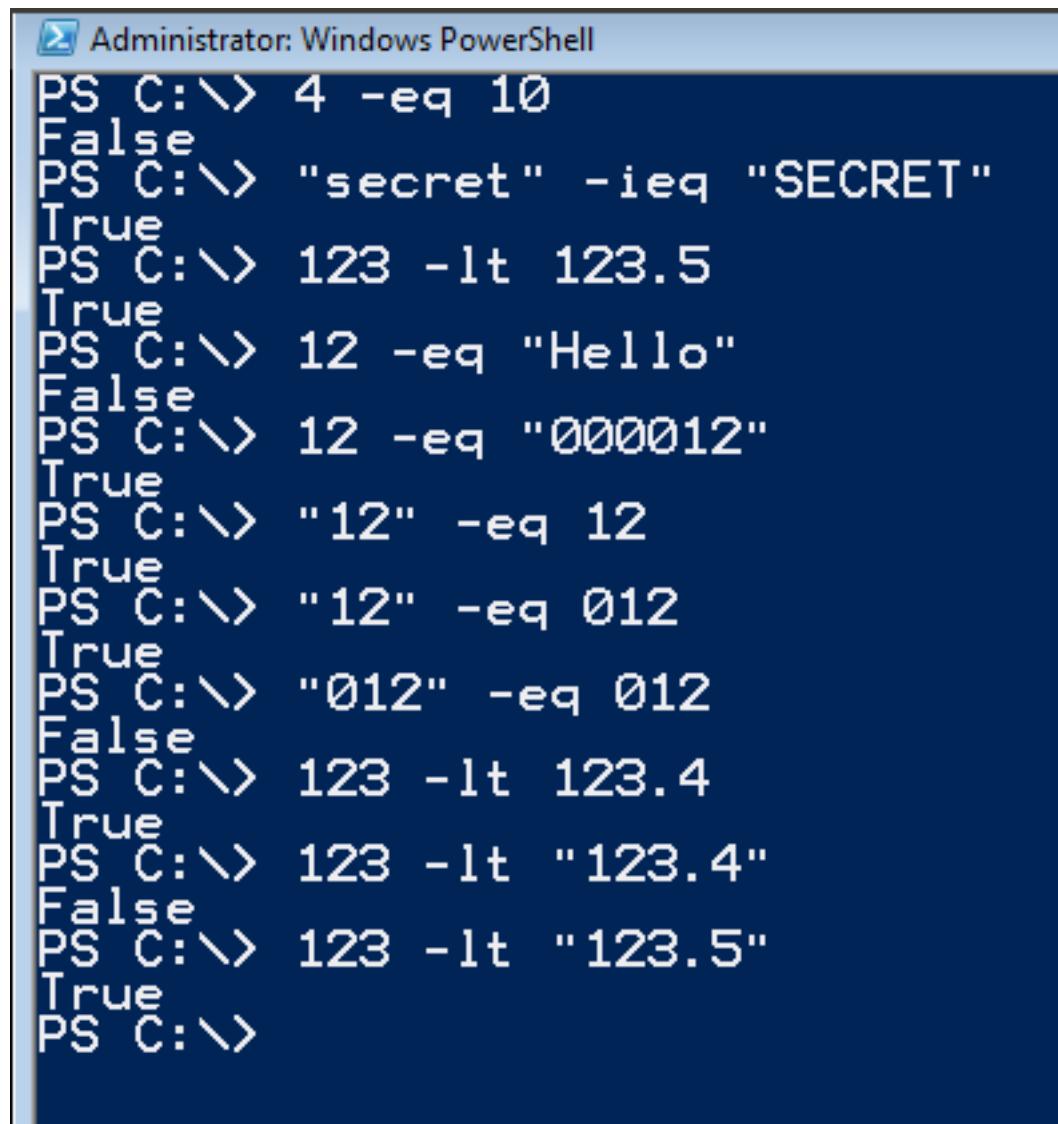
Mit Hilfe der Vergleichsoperatoren lässt sich eine Bedingung bereits in der Konsole vornehmen.

Im Gegensatz zu anderen Programmiersprachen verwendet PS bei der Formulierung von Vergleichsoperatoren eine der englischen Sprache verwandte Syntax und nicht die sonst verwendeten mathematischen Operationszeichen. Eine Übersicht finden sie hier [Vergleichsoperatoren](#).

Wie der Tabelle zu entnehmen ist, gibt es drei Varianten der Vergleichsoperatoren. Soll die Groß- und Kleinschreibung berücksichtigt werden, stellt man dem Operator ein „c“, ansonsten ein „i“ voraus. Wird dies weggelassen, ist case-insensitive eingestellt.

Beurteilen Sie die Ergebnisse der folgenden Vergleiche:

```
4 -eq 10
"secret" -ieq "SECRET"
123 -lt 123.5
12 -eq "Hello"
12 -eq "000012"
"12" -eq 12
"12" -eq 012
"012" -eq 012
123 -lt 123.4
123 -lt "123.4"
123 -lt "123.5"
```



```
Administrator: Windows PowerShell
PS C:\> 4 -eq 10
False
PS C:\> "secret" -ieq "SECRET"
True
PS C:\> 123 -lt 123.5
True
PS C:\> 12 -eq "Hello"
False
PS C:\> 12 -eq "000012"
True
PS C:\> "12" -eq 12
True
PS C:\> "12" -eq 012
True
PS C:\> "012" -eq 012
False
PS C:\> 123 -lt 123.4
True
PS C:\> 123 -lt "123.4"
False
PS C:\> 123 -lt "123.5"
True
PS C:\>
```

6.2 Umgekehrte Vergleiche

Ein Vergleich hat als Ergebnis immer entweder ein *true* oder ein *false*. Mit Hilfe des *-not* - Operators bzw. des *!* können diese Ergebnisse auch umgedreht werden.

```
$a = 10
$a -gt 5
True
-not ($a -gt 5)
False
# Shorthand: instead of -not "!" can also be used:
!($a -gt 5)
False
```

6.3 Verbinden mehrerer Vergleiche

Mit Hilfe der Verknüpfungsoperatoren `-and` bzw. `-or` können mehrere Vergleiche auch zu einem Gesamtvergleich verbunden werden.

```
( ($age -ge 18) -and ($sex -eq "m") )
```

:ref: *logischeOperatoren*

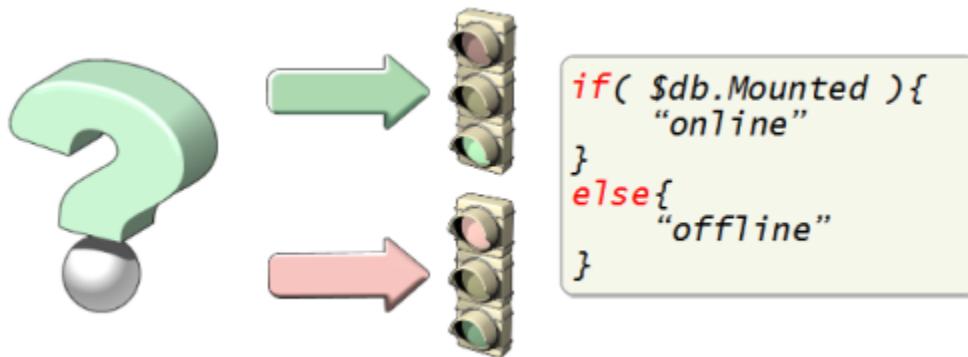
:ref: *auswahl_if_elseif_else*

6.4 Verzweigung

Die Vergleichsoperatoren geben zwar einen Hinweis auf True oder False, aber sie allein lassen keine Auswahlentscheidung des Programmiers zu. Deshalb finden die Vergleichsoperatoren ihren Einsatz innerhalb von Wenn-Dann-Statements, die eine Reaktion auf True oder False zulassen. Die Syntax dieser Anweisung ist wie folgt

```
If (condition) {
# If the condition applies,
# this code will be executed
}

$a = 11
If ($a -gt 10) { "$a is larger than 10" }
11 is larger than 10
```



```
If ($a -gt 10)
{
    "$a is larger than 10"
}
Else
{
    "$a is less than or equal to 10"
}
```

Mehrere Bedingungen können über das `ElseIf` - Statement beschrieben werden.

```
If ($a -gt 10)
{
    "$a is larger than 10"
}
Else
```

```
{  
    "$a is less than or equal to 10"  
}  
  
If ($a -gt 10)  
{  
    "$a is larger than 10"  
}  
ElseIf ($a -eq 10)  
{  
    "$a is exactly 10"  
}  
ElseIf ($a -ge 10)  
{  
    "$a is larger than or equal to 10"  
}  
Else  
{  
    "$a is smaller than 10"  
}
```

Die Bedingungen können natürlich noch in sich verschachtelt werden

The diagram features a blue header bar with the text "Using elseif with if/else". Below it is a white content area. On the left, there are three traffic light icons with arrows pointing right: a green one at the top, followed by two red ones. To the right of these icons is a code snippet in a light yellow box:

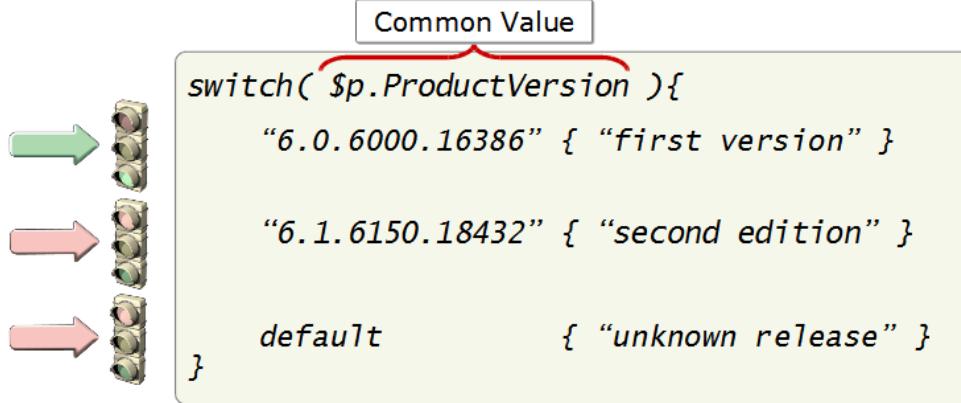
```
if( $p.ProductVersion -eq "6.0.6000.16386" ){  
}  
elseif( $p.ProductVersion -eq "6.1.6150.18432" ){  
}  
else{  
    "unknown release"  
}
```

At the bottom right of the content area is a blue navigation button with a play and square icon.

6.5 Auswahl

Um eine Variable auf verschiedene Werte hin abzuprüfen, ist neben dem if-elseif-Statement noch eine weitere Möglichkeit vorhanden. Sie kann allerdings nur auf **eine Variable** angewendet werden.

- When **if/elseif/else** expressions compare with a common value, a **switch** construct might be more concise
- The **switch** construct also has more powerful features



```

$value = 1
$switch($value)
{
    1 { "Number 1" }
    2 { "Number 2" }
    3 { "Number 3" }
}

Number 1

```

Als Standard-Vergleichsoperator wird `-eq` genommen. Es sind aber auch andere Vergleiche denkbar.

```

$value = 8
Switch ($value)
{
    # Instead of a standard value, a code block is used
    # that results in True for numbers smaller than 5:
    {$_-le 5} { "Number from 1 to 5" }
    # A value is used here; Switch checks whether this
    # value matches $value:
    6 { "Number 6" }
    # Complex conditions are allowed as they are here,
    # where -and is used to combine two comparisons:
    {(($_-gt 6) -and ($_-le 10))} { "Number from 7 to 10" }
}
Number from 7 to 10

```

Auch ein Mischen ist möglich:

```

$value = 8
Switch ($value)
{
    # The initial value (here it is in $value)
    # is available in the variable $_:
    6 { "Number 6" }
    {(($_-gt 6) -and ($_-le 10))}
        { "$_ is a number from 7 to 10" }
}

```

```
}
```

```
8 is a number from 7 to 10
```

6.5.1 Default

Falls keine der Bedingungen zutrifft, kann mit Hilfe des Schlüsselwortes **default** eine Klausel ausgeführt werden.

```
$value = 50
Switch ($value)
{
    {$_ -le 5} { "$_ is a number from 1 to 5" }
    6 { "Number 6" }
    {($_ -gt 6) -and ($_. -le 10))} { "$_ is a number from 7 to 10" }
    # The code after the next statement will be
    # executed if no other condition has been met:
    default { "$_ is a number outside the range from 1 to 10" }
}

50 is a number outside the range from 1 to 10
```

6.5.2 Break

Häufig kann es vorkommen, dass mehrere Vergleiche innerhalb einer switch-Anweisung gültig sind. **Powershell** würde dann mehrere Klauseln ausführen. Um dies zu verhindern, kann man jede Klausel mit Hilfe des Schlüsselwortes **break** versehen. Nach der ersten erfolgreichen Klausel beendet **Powershell** die Ausführung.

```
$value = 50
Switch ($value)
{
    50 { "the number 50" }
    {$_ -gt 10} {"larger than 10"}
    {$_ -is [int]} {"Integer number"}
}

The Number 50
Larger than 10
Integer number

$value = 50
Switch ($value)
{
    50 { "the number 50"; break }
    {$_ -gt 10} {"larger than 10"; break}
    {$_ -is [int]} {"Integer number"; break}
}
The number 50
```

6.5.3 String-Vergleiche

Da switch nur auf Gleichheit prüft, sind auch andere Datentypen vergleichbar.

```
$action = "sAVe"
Switch ($action)
{
    "save" { "I save..." }
    "open" { "I open..." }
    "print" { "I print..." }
    Default { "Unknown command" }
}
I save...

Groß- und Kleinschreibung kann man mit der Option -case eingeschaltet werden
```

```
$action = "sAVe"
Switch -case ($action)
{
    "save" { "I save..." }
    "open" { "I open..." }
    "print" { "I print..." }
    Default { "Unknown command" }
}
Unknown command
```

Wildcards können ebenfalls überprüft werden

```
$text = "IP address: 10.10.10.10"
Switch -wildcard ($text)
{
    "IP*" { "The text begins with IP: $_" }
    "*.*.*.*" { "The text contains an IP " +
    "address string pattern: $_" }
    "*dress*" { "The text contains the string " +
    "'dress' in arbitrary locations: $_" }
}
The text begins with IP: IP address: 10.10.10.10
The text contains an IP address string pattern:
IP address: 10.10.10.10
The text contains the string 'dress' in arbitrary
locations: IP address: 10.10.10.10
```

6.5.4 Reguläre Ausdrücke

6.5.5 switch auf Listen

Was mach folgender Code ?

```
$array = 1..5

Switch ($array)
{
    {$_ % 2} { "$_ is odd."}
    Default { "$_ is even."}
}
1      is      odd.
2      is      even.
3      is      odd.
```

4	is even.
5	is odd.

6.5.6 Zusammenfassung

Übersetzen Sie folgenden Text:

Intelligent decisions are based on conditions, which in the simplest form can be reduced to plain Yes or No answers. Using the comparison operators, you can formulate such conditions and can even combine these with the logical operators to form complex queries.

The simple Yes/No answers of your conditions determine whether particular PowerShell instructions should be carried out or not. In the simplest form, you can use the Where-Object cmdlet in the pipeline. It functions there like a filter, allowing only those results through the pipeline that correspond to your condition.

If you would like more control, or would like to execute larger code segments independently of conditions, use the If statement, which evaluates as many different conditions as you wish and, depending on the result, executes the allocated code. This is the typical „If-Then“ scenario: if certain conditions are met, then certain code segments will be executed.

An alternative to the If statement is the Switch statement: using it, you can compare a fixed initial value with various possibilities. Switch is the right choice when you want to check a particular variable against many different possible values.

6.6 Aufgaben

6.6.1 ZahlenRaten

Erstellen sie ein Skript, welches folgende Aufgabe erfüllt:

Es muss eine Meldung anzeigen, in der der Benutzer aufgefordert wird, eine Zahl zwischen 1 und 50 einzugeben. Das Skript muss die vom Benutzer eingegebene Zahl mit einer zufällig generierten Zahl vergleichen. Wenn die Zahlen nicht übereinstimmen, muss das Skript eine Meldung anzeigen, in der angegeben wird, ob die geratene Zahl zu hoch oder zu niedrig war, und der Benutzer aufgefordert wird, noch einmal zu raten.

Wenn der Benutzer richtig rät, muss das Skript die Zufallszahl sowie die Anzahl der Rateversuche anzeigen. An diesem Punkt ist das Spiel beendet, das Skript muss also auch beendet werden.

Siehe dazu auch: http://msdn.microsoft.com/de-de/library/system.random_members.aspx

6.6.2 Dateien kopieren

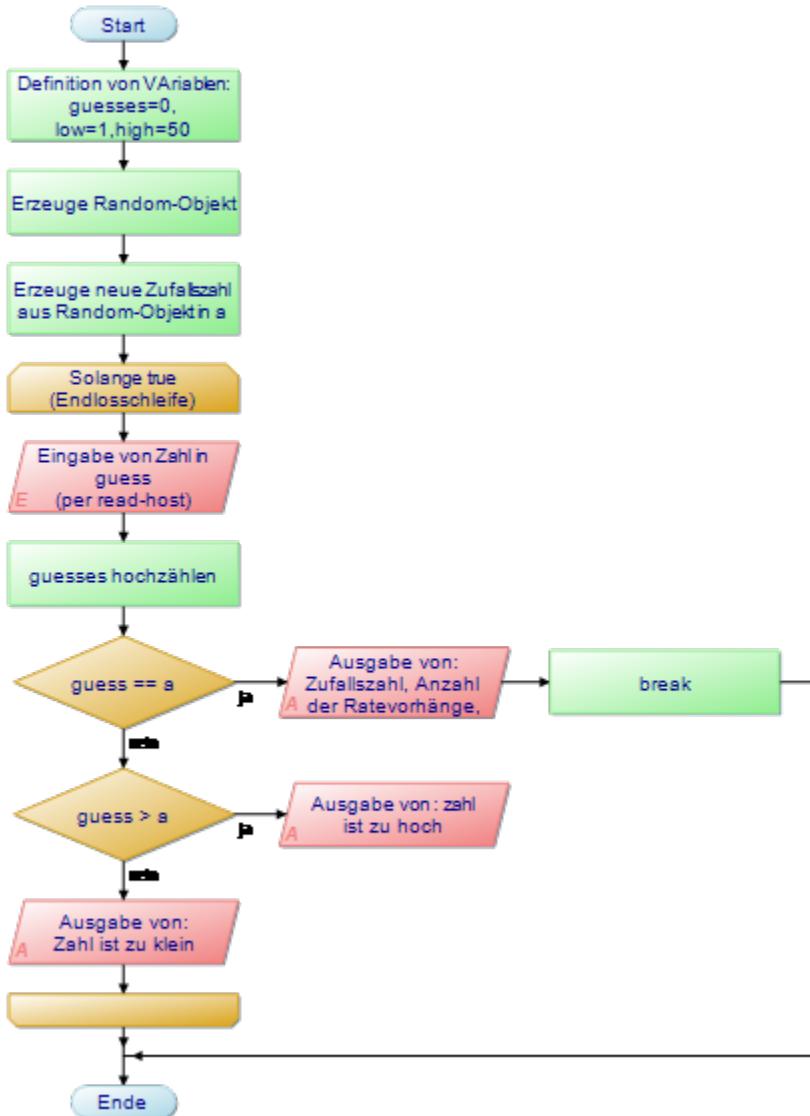
Dieses Skript soll folgende Aufgaben ausführen:

- Durchsuchen des Ordners „C:Scripts“ und von dessen Unterordnern.
- Durchsuchen jedes Ordners nach sämtlichen Textdateien (Dateien mit der Erweiterung „.txt“) und Prüfen des Erstellungsdatums jeder Datei.
- Kopieren jeder .txt-Datei, die mehr als 10 Tage zuvor erstellt wurde, in den Ordner „C:Old“.
- Ausgeben des Dateinamens (kein vollständiger Pfad, nur Dateiname) jeder kopierten Datei.
- Ausgeben der Anzahl der kopierten Dateien.

6.7 Lösungen

6.7.1 ZahlenRaten

Hauprogramm 1



```

$guesses = 0

$low = 1
$high = 50

$a = New-Object Random
$a = $a.Next($low,$high)

while ($true)
{
    $guess = read-host "Enter a number between 1 and 50: "
  
```

```
$guesses++  
  
if ($guess -eq $a)  
{  
    "Random Number: " + $guess  
    "Number of Guesses: " + $guesses  
    break  
}  
elseif ($guess -gt $a)  
{  
    "Too high"  
}  
else  
{  
    "Too low"  
}  
}
```

6.7.2 Dateien kopieren

```
foreach ($i in Get-ChildItem C:\Scripts -recurse)  
{  
    if ((-$i.CreationTime -lt $($Get-Date).AddDays(-10)) -and ($i.Extension -eq ".txt"))  
    {  
        #Copy-Item $i.FullName C:\old  
        $i.Name  
        $x = $x + 1  
    }  
}  
  
"Total Files: " + $x
```

KAPITEL 7

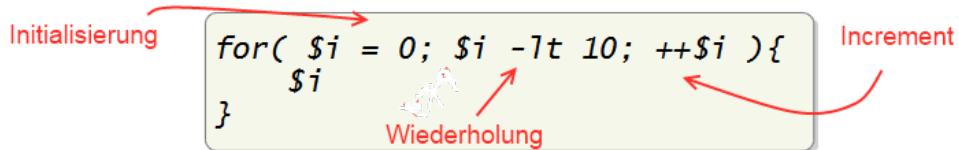
Schleifen

Überblick

- Do und While
- For
- Foreach
- Break/Continue
- Nested Loops und Labels

Neben der Abfrage von Bedingungen muss der Programmierer häufig durch Listen iterieren, so muss er z.B. häufig die Inhalte von Arrays ausgeben.

7.1 For-Schleife



```

"looking for processes with big VM"
$p = Get-Process
for( $i = $j = 0; $i -lt $p.count; ++$i ){
    if( $p[$i].VM -ge 200MB ){
        "{0}/{1} VM={2}" -f $p[$i].id, $p[$i].name,
        $p[$i].VM
        ++$j # note that we found one more
    }
}
"found {0} big processes out of {1}" -f $j, $i

```

Die einfachste Art des Schleifendurchlaufs ist der mittels des **for**-Statements, gerne auch Zählschleife genannt. Es hat 3 grundlegende Elemente:

Initialisierung: Der erste Ausdruck wird beim Beginn der Schleife ausgewertet.

Wiederholungsbedingung: Der zweite Ausdruck wird bei jedem neuen Schleifendurchlauf ausgewertet. Wenn der Ausdruck **true** ergibt, wird in die Schleife gesprungen.

Increment: Der 3. Ausdruck wird bei Vorhandensein nach jedem Schleifendurchlauf ausgeführt. Häufig wird hier ein Zähler inkrementiert. Er ist aber nicht grundsätzlich notwendig.

while	do/while	do/until
<pre> while(\$looping) { code } </pre>	<pre> do { code } while (\$looping) </pre>	<pre> do { code } until (\$done) </pre>

- In this demonstration, you will see how the **foreach** construct and ForEach-Object cmdlet are different, yet similar.

foreach Construct

```
$list = Get-ChildItem -Recurse
foreach ( $file in $list ){
    "{0} is {1} bytes"-f
    $file.name,$file.length
}
```

ForEach-Object Cmdlet

```
Get-ChildItem -Recurse | ForEach-Object {
    "{0} is {1} bytes" -f $_.name, $_.length
}
```

7.2 while, do - while

Do and While sind prinzipiell Endlosschleifen. Solange die Abbruchbedingung nicht true ergibt, werden beide Schleifen endlos durchlaufen.

```
Do {
    $input = Read-Host "Your homepage"
} While (!($input -like "www.*.*"))

$input = Read-Host "Your homepage"
while(!($input -like "www.*.*"))
{
    $input = Read-Host "Your homepage"
}
```

7.3 foreach

Die foreach-Anweisung iteriert über eine Liste von Objekten. Sie sollte nicht innerhalb einer Pipeline angewendet werden, weil sie dort auf das Ergebnis der Pipeline warten muss; innerhalb von Pipelines ist das foreach-Objekt die geeignete Wahl.

```
# Create your own array:
$array = 3,6,"Hello",12
# Read out this array element by element:
Foreach ($element in $array)
{
    "Current element: $element"
```

```
}
```

```
Current element: 3
```

```
Current element: 6
```

```
Current element: Hello
```

```
Current element: 12
```

7.4 Schleifen verlassen

Mit Hilfe des **break**-Statements können Schleifen vorzeitig verlassen werden. Eine eventuell noch zu durchlaufende Abbruchbedingung wird nicht mehr ausgewertet.

```
While ($true)
{
    $password = Read-Host "Enter password"
    If ($password -eq "secret") {break}

}

For ($i=1; $i -lt 4; $i++)
{
    $password = Read-Host "Enter password ($i. try)"
    If ($password -eq "secret") {break}
    If ($i -ge 3) { Throw "The entered password was incorrect." }
}
```

7.5 Schleifen fortführen

Mit Hilfe von **continue** können Sie den jeweils aktuellen Schleifendurchlauf beenden, ohne die Schleife selbst zu beenden.

Das folgende Beispiel gibt nur die Länge von Dateien aus; sollte es auf Ordner stoßen, werden diese sofort übersprungen.

```
Foreach ($entry in Dir $env:windir)
{
    # If the current element matches the desired type,
    # continue immediately with the next element:
    If (!$entry -is [System.IO.FileInfo])) { Continue }
    "File {0} is {1} bytes large." -f $entry.name, $entry.length
}
```

7.6 Zusammenfassung

The cmdlet `ForEach-Object` gives you the option of processing single objects of the PowerShell pipeline, such as to output the data contained in object properties as text or to invoke methods of the object.

`Foreach` is a similar type of loop whose contents do not come from the pipeline, but from an array or a collection. In addition, there are endless loops that iterate a code block until a particular condition is met.

The simplest type of such loops is `While`, in which continuation criteria are checked at the beginning of the loop. If you want to do the checking at the end of the loop, choose `Do...While`. The `For` loop is an extended `While` loop, because

it can count loop cycles and automatically terminate the loop after a designated number of iterations. This means that For is suited mainly for loops in which counts are to be made or which must complete a set number of iterations.

Do...While and While, on the other hand, are suited for loops that have to be iterated as long as the respective situation and running time conditions require it.

Finally, Switch is a combined Foreach loop with integrated conditions so that you can immediately implement different actions independently of the read element. Moreover, Switch can step through the contents of text files line by line and evaluate even log files of substantial size.

All loops can exit ahead of schedule with the help of Break and skip the current loop cycle with the help of Continue. In the case of nested loops, you can assign an unambiguous name to the loops and then use this name to apply Break or Continue to nested loops.

KAPITEL 8

Arrays und Hashs

Überblick

- Erzeugen von Arrays und Hash
- Ansprechen von Arrays und Hash
- Durchlaufen von Arrays und HAsh

Arrays erweitern den Grundgedanken einer einfachen Variable um den Aspekt des Speicherns mehrerer Werte unter einem Variablenamen.

Letzlich gibt **Powershell** bei mehrzeiligen Ausgaben immer einen Array zurück.

```
#die Ausgabe ist kein mehrzeiliger String, sondern ein Array von Strings
#jede Zeile ist ein Element des Arrays
$a = ipconfig
$a
Windows IP Configuration
Ethernet adapter LAN Connection
Media state
. . . . . : Medium disconnected

Connection-specific DNS Suffix:
Connection location IPv6 Address . . : fe80::6093:8889. ....
IPv4 address . . . . . : 192.168.1.35
Subnet Mask . . . . . : 255.255.255.0
Standard Gateway . . . . . : 192.168.1.1
```

Werte	Index
Windows IP Configuration	0
Ethernet adapter LAN Connection	1
Media state	2
..... : Medium disconnected	3
	4
Connection-specific DNS Suffix:	5
Connection location IPv6 Address . : fe80::6093:8889%25...	6
IPv4 address : 192.168.1.35	7
Subnet Mask : 255.255.255.0	8
Standard Gateway : 192.168.1.1	9

Write-Host: \$a[1] =====> Ethernet adapter ...

Da in diesem Beispiel lediglich Strings gespeichetr werden, bleibt eine Weiterverarbeitung nur mit Hilfe von String-verarbeitung möglich.

Echte Powershell-Cmdlets reichen jedoch Objekte in die Arays zurück, die sich deutlich einfacher und eleganter bearbeiten lassen. Näheres dazu später.

8.1 Erzeugen eigener Arrays

Wenn man Arrays nicht als Ergebnis erhält, so kann man sich recht einfach Arrays selbst erzeugen.

```
#verschiedene Methoden, um Werte in Arrays zu definieren
$array = 1,2,3,4
$array = 1..4
$array
1
2
3
4

PS C:\> $array = "Hello", "World", 1, 2, (Get-Date)
PS C:\> $array
Hello
World
1
2

Montag, 16. August 2010 22:18:17
```

8.2 Arrays mit einem oder keinem Element

```
$array = ,1
$array.Length
1

$array = @()
$array.Length
0
$array = @(12)
$array
12
$array = @(1,2,3,"Hello")
$array
```

8.3 Ansprechen von Array-Elementen

Die Inhalte des Arrays werden über den Index angesprochen

```
# Create your own new array:  
$array = -5..12  
# Access the first element:  
$array[0]  
-5  
# Access the last element (several methods):  
$array[-1]  
12  
$array[$array.Count-1]  
12  
$array[$array.length-1]  
12  
# Access a dynamically generated array that is not stored in a variable:  
(-5..12) [2]  
-3
```

8.4 Ansprechen mehrerer Elemente eines Arrays

Man kann mehrere Elemente eines Arrays durch eine kommaseparierte Liste von Indexwerten ansprechen. Das Ergebnis ist wiederum ein Array, der nur aus diesen Werten besteht.

```
PS C:\> $list = Dir  
PS C:\> $list[1,4,7,12]
```

Verzeichnis: C:\

Mode	LastWriteTime	Length	Name
----	-----	-----	-----
d----	20.07.2010	23:00	data
d----	31.07.2009	22:40	Dokumente und Einstellungen
d----	18.08.2009	20:54	inetpub
d----	14.07.2009	04:37	PerfLogs

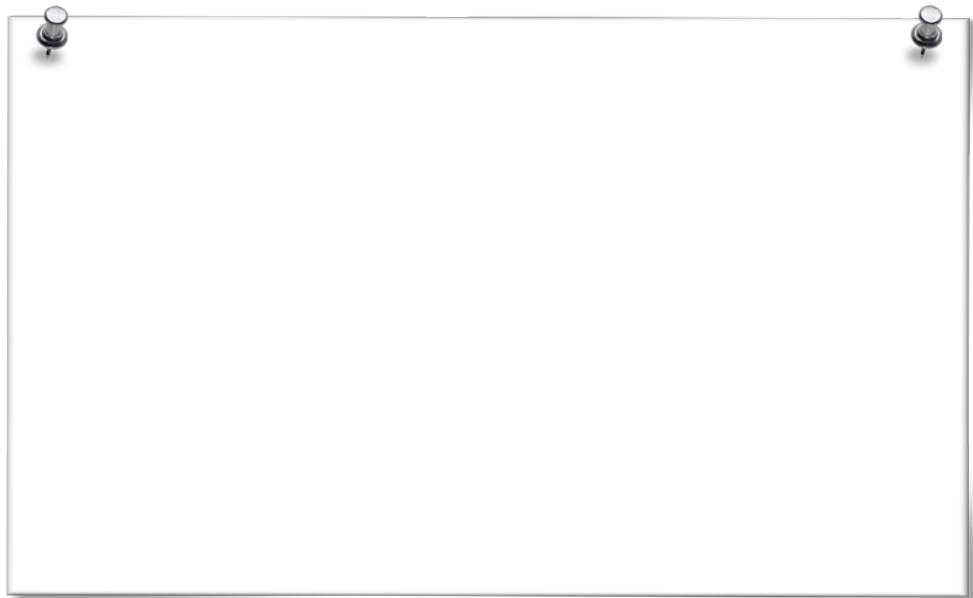
8.5 Hinzufügen und Entfernen von Elementen

Da **Powershell** die Länge von Arrays nicht ändern kann, muss beim Hinzufügen und Entfernen jeweils ein neuer Array angelegt werden und die Werte des alten Arrays in den neuen kopiert werden.

Das Hinzufügen erfolgt mit Hilfe des += - Operators.

```
PS C:\> $array = 1..5
PS C:\> $array += 6
PS C:\> $array
1
2
3
4
5
6
```

Das Hinzufügen von Arrays hat jedoch „teure“ Kopieraktionen im Hintergrund zur Folge



Da **Powershell** aus Arrays keine Werte entfernen kann, hilft nur ein (umständliches) Umkopieren des Alten in einen neuen Array.

```
PS C:\> $array = 1..5
PS C:\> $array += 6
PS C:\> $array
1
2
3
4
5
6
```

```
PS C:\> $array = $array[1..2] + $array[4..5]
PS C:\> $array
2
3
5
6
```

8.6 Kopieren von Arrays

Das Kopieren von Arrays erfolgt durch Zuweisen der alten Arrayvariable an eine neue Arrayvariable. Dabei muss jedoch beachtet werden, dass nicht die Werte kopiert werden, sondern lediglich die Referenzvariablen. Dies kann gewünschte/unerwünschte Nebeneffekte besitzen.

```
$array1 = 1,2,3
$array2 = $array1
$array2[0] = 99
$array1[0]
??
```

Um diesen Effekt nicht zu erhalten, müssen die Arrays mit Hilfe der **clone()**-Methode kopiert werden.

```
$array1 = 1,2,3
$array2 = $array1.Clone()
$array2[0] = 99
$array1[0]
??
```

Immer wenn in einer solchen Situation neue Elemente hinzugefügt werden, wird hinter den Kulissen der gleiche Ablauf stattfinden.

```
# Create array and store pointer to array in $array2:
$array1 = 1,2,3
$array2 = $array1
# Assign a new element to $array2. A new array is created in the process and stored
in $array2:
$array2 += 4
$array2[0]=99
# $array1 continues to point to the old array:
$array1[0]
1
```

8.7 Arrays by val/by ref

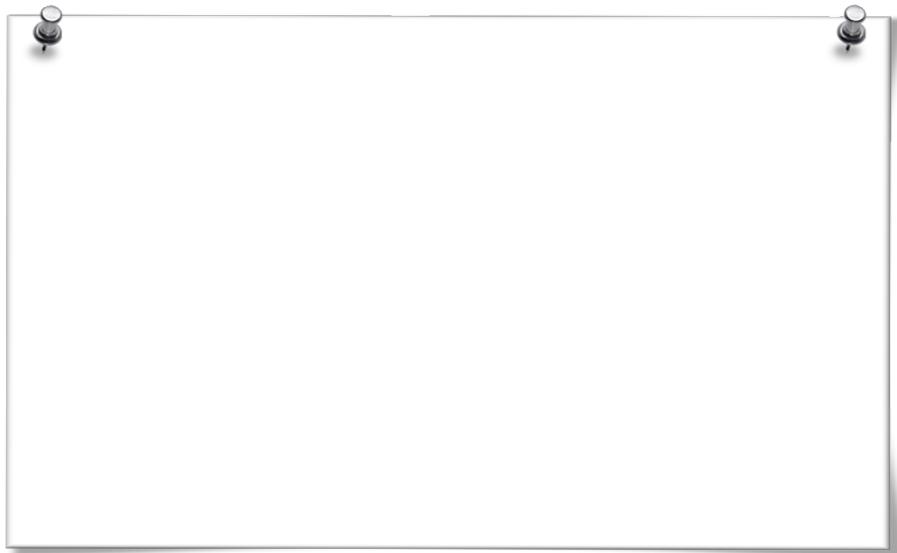
Verweise auf Arrays bzw. ein lesender Zugriff auf Einzelwerte haben führen zu unterschiedlichen Ergebnissen, wenn die Werte anschließend geändert werden.

Demo

```
$messwerte = 1,2,3,4,4,3,5,6
#indexpos   0 1 2 3 4 5 6 7
# 0 = Temp in Celsius
# 1 = Luftfeuchtigkeit in %
# 2 = Wassertemp

$kopie = $messwerte
$messwerte[1] = 20
$kopie[1]
$kopie[1]= 30
$messwerte[1]

$luftfeuchtigkeit = $messwerte[1]
$luftfeuchtigkeit = 40
```



By val:

Werden einzelne Werte aus einem Array in eine Variable kopiert, geschieht dies immer als echte Kopie. Die ausgelagerten Werte haben keinerlei Bezug mehr zu den Werten im Array.

By ref:

Wird der gesamte Array einer neuen Variable zugewiesen, so verweisen beide Variablen anschließend auf die gleiche Speicherstelle im Heap. Eine Änderung der Werte über eine Variable führt automatisch auch zu einer Veränderung in der anderen Variablen, weil beide auf die gleichen Speicherstellen im Heap zeigen.

8.8 Typisierte Arrays

Ähnlich wie bei Variablen können Arrays typisiert werden. Sie können dann nur noch bestimmte Datentypen aufnehmen.

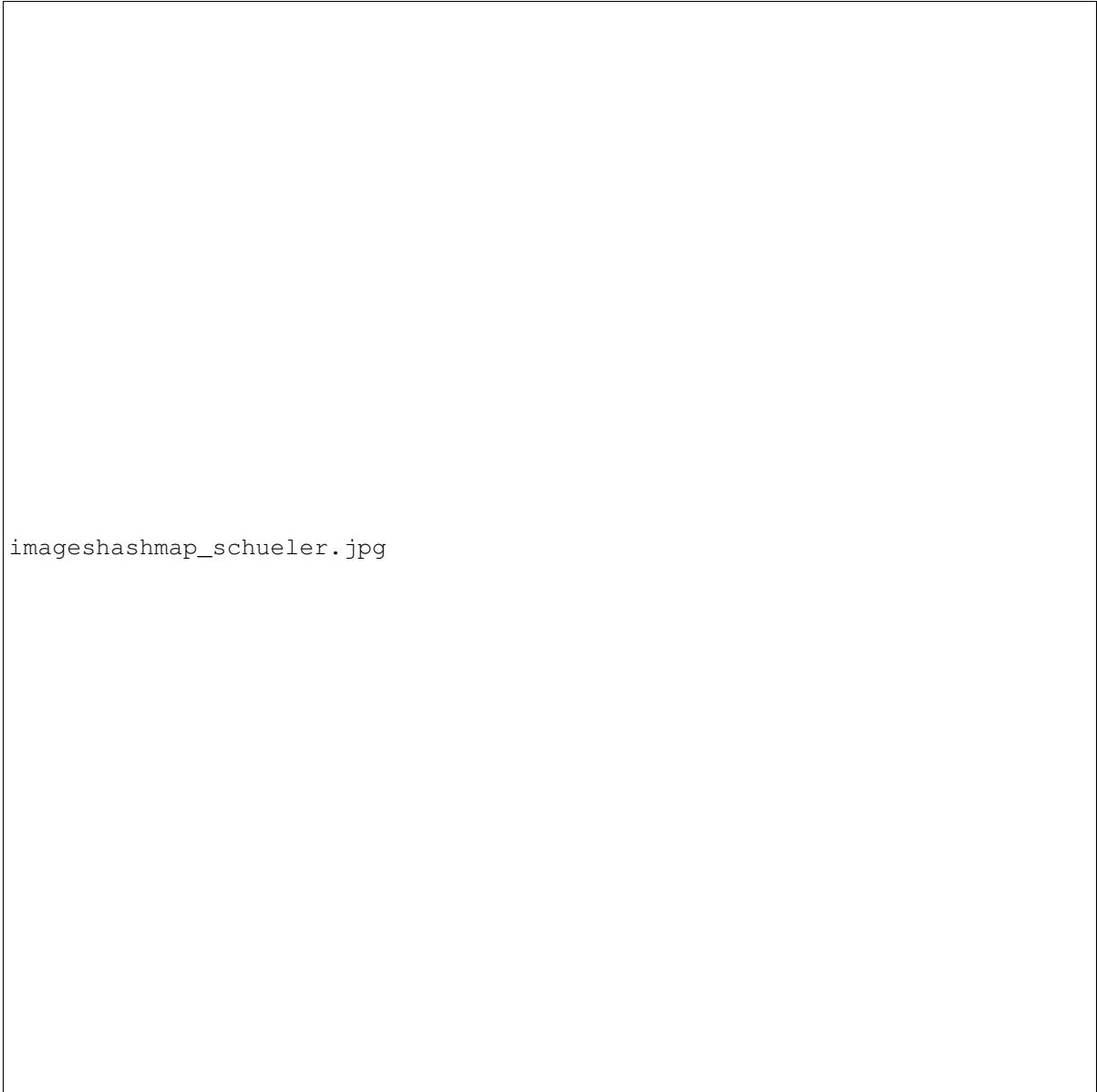
```
# Create a strongly typed array that can store whole numbers only:  
[int[]]$array = 1,2,3  
# Everything that can be converted into a number is allowed  
# (including strings):  
$array += 4  
$array += 12.56  
$array += "123"  
# If a value cannot be converted into a whole number, an error  
# will be reported:  
$array += "Hello"  
The value "Hello" cannot be converted into the type "System.Int32".  
Error: "Input string was not in a correct format."  
At line:1 char:6  
+ $array <<< += "Hello"
```

8.9 Zusammenfassung

Arrays can store as many separate elements as you like. Arrays assign a sequential index number to elements that always begin at 0. You create new arrays with @(Element1, Element2, ...). You can also leave out @() for arrays and only use the comma operator.

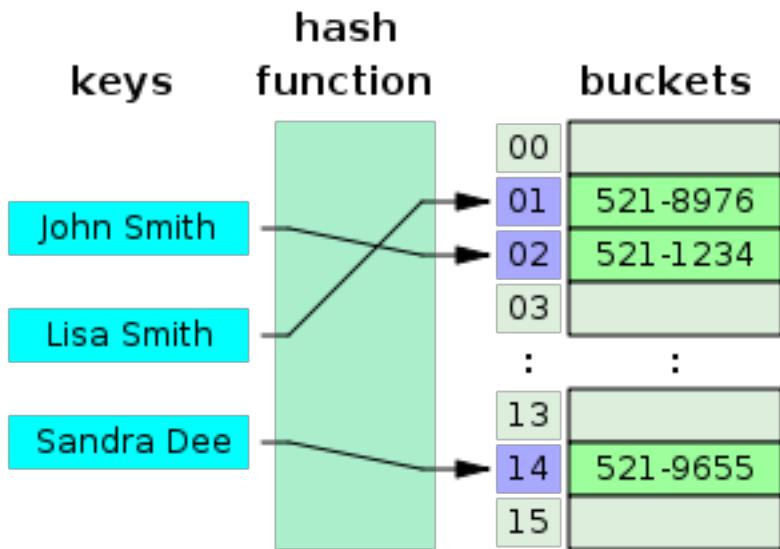
You can address single elements of an array or hash able by using square brackets. Specify either the index number (for arrays) or the desired element in the square brackets. Using this approach you can select and retrieve several elements at the same time.

8.10 Hash



imageshashmap_schueler.jpg

Hashmaps sind eine weitere Möglichkeit, Daten in einer strukturierten Form zu speichern. Im Gegensatz zum Array verwenden diese keinen Index zur Positionierung der Daten, sondern einen frei wählbaren Schlüssel, der durch eine Hashfunktion auf eine Indexposition umgerechnet wird.



8.10.1 Vorteile der Hashtable im Vergleich zum Array

- Freie Wahl des Schlüsselwertes
- Beliebig erweiterbar (ohne die „Verwaltungskosten“ des Array)
- Schnellere Suche, da direkter Zugriff über den Schlüssel

<http://powershell.com/cs/blogs/ebookv2/archive/2012/02/07/chapter-4-arrays-and-hashtables.aspx#using-hash-tables>

<http://technet.microsoft.com/en-us/library/ee692803.aspx>

<http://blogs.technet.com/b/heyscriptingguy/archive/2011/05/19/create-custom-objects-in-your-powershell-script.aspx>

<http://gallery.technet.microsoft.com/scriptcenter/925a517d-f400-4773-a79d-22ba9e91618e>

8.10.2 Zusammenfassende Übung

Ihre Firma möchte sicherstellen, dass sich in einem bestimmten Ordner der Windows-Server-Landschaft nur Dateien befinden, denen man *trauen* kann, d.h. die Integrität der Dateien ist sichergestellt. Der Einfachheit gehen wir davon aus, dass dieser Ordner keine Unterordner hat. Sie hat dazu einen garantieren sicheren Referenzordner als Vergleichsordner geschaffen, welcher alle Dateien des jeweiligen Ordners enthält. Andere Dateien sind nicht zugelassen; ebenso ist es wichtig, dass die Inhalte der Dateien gleich sind. Es handelt sich dabei sowohl um Textdateien als auch um Dateien im Binärformat (exe, dll, etc.). Sie können davon ausgehen, dass die Dateistruktur der Windows-Server identisch ist.

Realisieren Sie eine Lösung in Powershell.

Vertical bars
like this

siehe: <http://blog.ideri.com/?p=1382>

<https://www.windowspro.de/script/dateien-vergleichen-powershell>

<http://www.ct-systeme.com/ct/tipps/Seiten/MD5Hashes.aspx>

<https://www.windowspro.de/script/dateien-vergleichen-powershell>

http://www.tecchannel.de/server/windows/2054277/powershell_40_im_ueberblick/index3.html

<http://blog.ideri.com/?p=1382>

<http://blog.brianhartsock.com/2008/12/13/using-powershell-for-md5-checksums/>

Lösungsvorschlag

- Speichern der Servernamen in einer Textdatei, welche später per Schleife ausgelesen wird.
- Einlesen aller Dateien aus dem Referenzordner
- Einlesen aller Dateien aus dem Ordner des jeweiligen Rechners
- **Findet sich Rechnerdateiname in Referenzdateiname ==> alles gut**

ansonsten: Rechnerdatei in Quarantäneordner schieben zur weiteren Behandlung Ausgabe der Rechnerdatei in csv-Datei mit komplettem Pfad

- **Kontrolle über Hash-Wert** ==> gleicher Hashwert: Datei scheint integer zu sein ==> ungleicher Hashwert: Datei ist nicht integer;
==> verschieben in Quarantäne (s.o.)
- **Kontrolle des Inhalts:** Compare-Object der Powershell kan auf Inhaltsgleichheit prüForegroundColor Bei Un-
gleichheit verschieben in Quarantäne-Ordner (s.o.)
- Vergleich

KAPITEL 9

Funktionen

Häufig ist es sinnvoll, seinen Quellcode in abgeschlossenen Strukturen zu definieren. Die rein sequenzielle Abfolge wird ersetzt durch den Aufruf von Programmcontainern, die eine gewisse Funktionalität bereitstellen. Auf diese Funktionalität kann dann innerhalb des Skriptes immer wieder zugegriffen werden.

Folgende Elemente werden beinhaltet:

- Funktionskopf (Signatur)
- Rückgabewerter
- Parameterübergabe

9.1 Grundsätzlicher Aufbau

Funktionen sind selbst definierte Kommandos, die im Prinzip drei Aufgaben haben:

- Shorthand: Abkürzungen für einfache Befehle zur Arbeitserleichterung
- Combining: Funktionen können Arbeit erleichtern, indem Sie mehrere Arbeitsschritte zusammenfassen können
- Kapselung und Erweiterung:

Der grundlegende Aufbau einer Funktion ist immer gleich.

Nach dem Funktion-Statement folgt der Name der Funktion und anschließend der Powershell-Codeblock in geschweiften Klammern. Ein Beispiel:

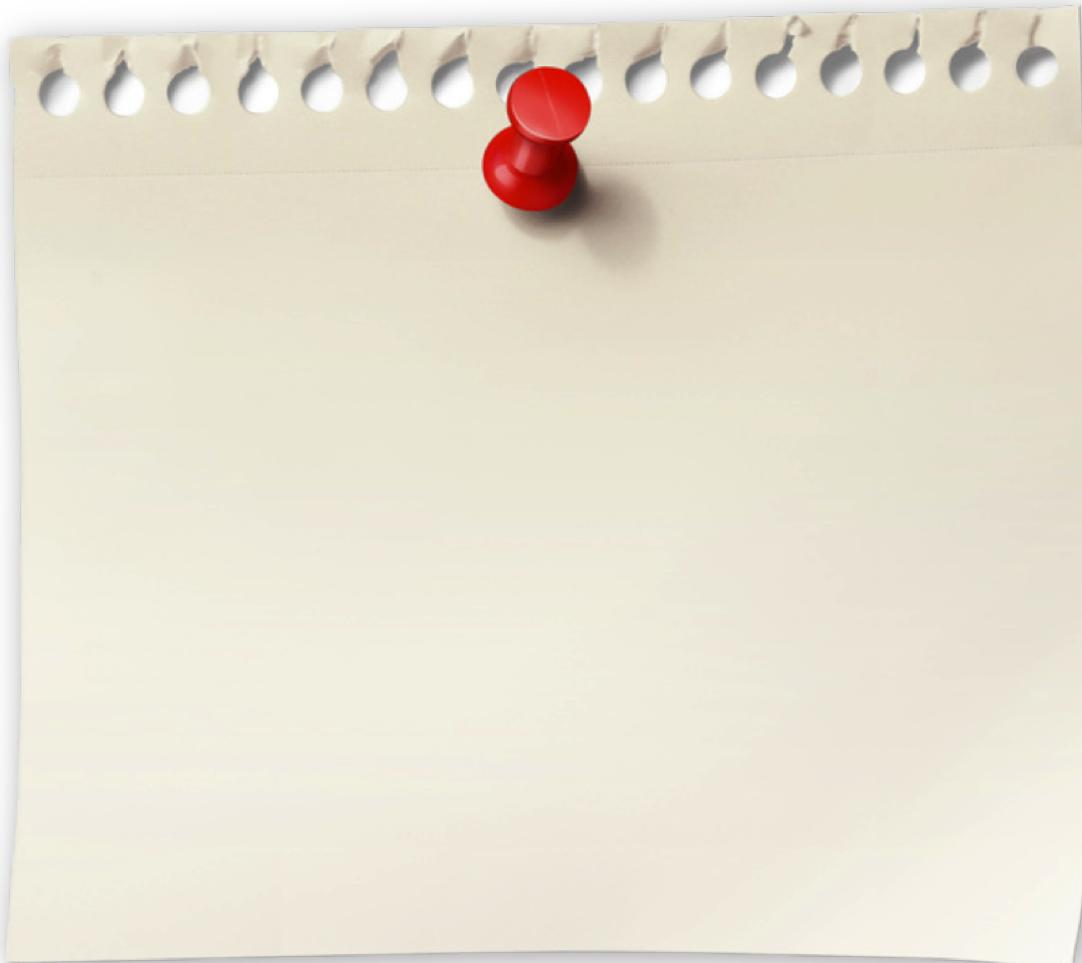
```
Function Cd.. { Cd .. }
Cd..

#Für immer wiederkehrende Aufgaben

Function myPing { ping.exe -w 100 -n 1 10.10.10.10 }
myPing
Pinging 10.10.10.10 with 32 bytes of data:
Reply from 88.70.64.1: destination host unreachable.
```

```
#mit einem Argument
Function myPing { ping.exe -w 100 -n 1 $args }
myPing www.microsoft.com
Pinging lbl1.www.ms.akadns.net [207.46.193.254] with 32 bytes of data:
Request timed out.
Ping statistics for 207.46.193.254:
Packets: Sent = 1, Received = 0, Lost = 1 (100% Loss),
```

Tafelbild



Gerade bei längeren Funktionen kann die Eingabe innerhalb einer Zeile etwas umständlich sein. Eingaben können in **Powershell** allerdings über mehrere Zeilen gehen. Dies erreicht man durch Drücken von Return, wenn die jeweils aktuelle Zeile noch nicht fertig geschrieben ist.

```

Function NextFreeDrive
{
    For ($x=67; $x -le 90; $x++)
    {
        $driveletter = [char]$x + ":"
        If (!(Test-Path $driveletter))
        {
            $driveletter
            break
        }
    }
}

```

Administrator: Windows PowerShell
PS C:\Users\steinam> Function nextFreeDrive
>> {
>> For(\$x=67;\$x -le 90; \$x++)
>> {
>> \$driveletter = [char]\$x + ":"
>> if(!(Test-Path \$driveletter))
>> {
>> \$driveletter
>> break
>> }
>> }
>> }
PS C:\Users\steinam> nextFreeDrive
D:
PS C:\Users\steinam> -

Spätestens jetzt wird der Einsatz eines Editors bzw. der grafischen Version der **Powershell** überlegenswert.

Eine Darstellung des gesamten Funktion kann man mit folgender Zeile erreichen

```
$function:NextFreeDrive
```

```

PS C:\Users\steinam> $function
PS C:\Users\steinam> $function:NextFreeDrive

For($x=67;$x -le 90; $x++)
{
$driveletter = [char]$x + ":"
if(!(Test-Path $driveletter))
{
$driveletter
break
}
}

```

Die Ausgabe einer Funktion in eine Textdatei kann man z.B. mit dem Out-File-Commandlet erreichen

```
$function:NextFreeDrive | Out-File -Encoding utf8 nextfreedrive.ps1
notepad .\nextfreedrive.ps1
```

```
$function:NextFreeDrive | Out-File -Encoding utf8 nextfreedrive.ps1
notepad .\nextfreedrive.ps1
```

```
For($x=67;$x -le 90; $x++)
{
$driveletter = [char]$x + ":"
if(!(Test-Path $driveletter))
{
$driveletter
```

9.2 Übergabe von Argumenten

Häufig will man einer Funktion Informationen mit übergeben, die diese dann weiterverarbeiten soll. Dies wird mit Hilfe von sog. **Argumenten** bewerkstelligt. Es gibt 4 verschiedene Arten der Argumentübergabe

Arbitrary arguments: die \$args variable (Array) beinhaltet alle Parameter, die einer Funktion übergeben werden. Damit ist sie eine gute Lösung, wenn eine flexible Anzahl von Parametern übergeben werden soll.

```
function add()
{
    $i = 0
    foreach($_ in $args)
    {
        $i = $i + $_
    }

    Write-Host $i
}
```

Named arguments: Eine Funktion kann auch pro Parameter einen Namen vergeben. Damit ist die Reihenfolge der Parameter beliebig, weil Sie über den Namen aufgelöst werden.

```
function add1($value1, $value2)
{
    Write-Host ($value1 + $value2)
}
add1 -value2 10 -value1 10
```

```
PS C:\Windows\system32> $function:add1
param($value1, $value2)
Write-Host ($value1 + $value2)

PS C:\Windows\system32> add1 -value2 10 -value1 10
20
PS C:\Windows\system32>
```

Predefined arguments: Parameter können mit default-Werten versehen sein. Falls der Benutzer keine eigenen Werte übergibt, werden die default-Werte genommen.

```
function add2($value1=10, $value2=20)
{
    Write-Host ($value1 + $value2)
}

add -value2 10 -value1 10
```

```
PS C:\Windows\system32> function add2($value1=10, $value2=20)
>>
>>
>>     Write-Host ($value1 + $value2)
>>
>>
PS C:\Windows\system32> add2
30
PS C:\Windows\system32> add2 -value2 5 -value1 10
15
PS C:\Windows\system32> _
```

Typed arguments: Parameter können mit einem bestimmten Datentyp definiert werden, um sicherzustellen, dass nur die Argumente aus „richtigen“ Datentypen bestehen.

```
function add([int] $value1, [int] $value2)
{
    $value1 + $value2
}
```

```

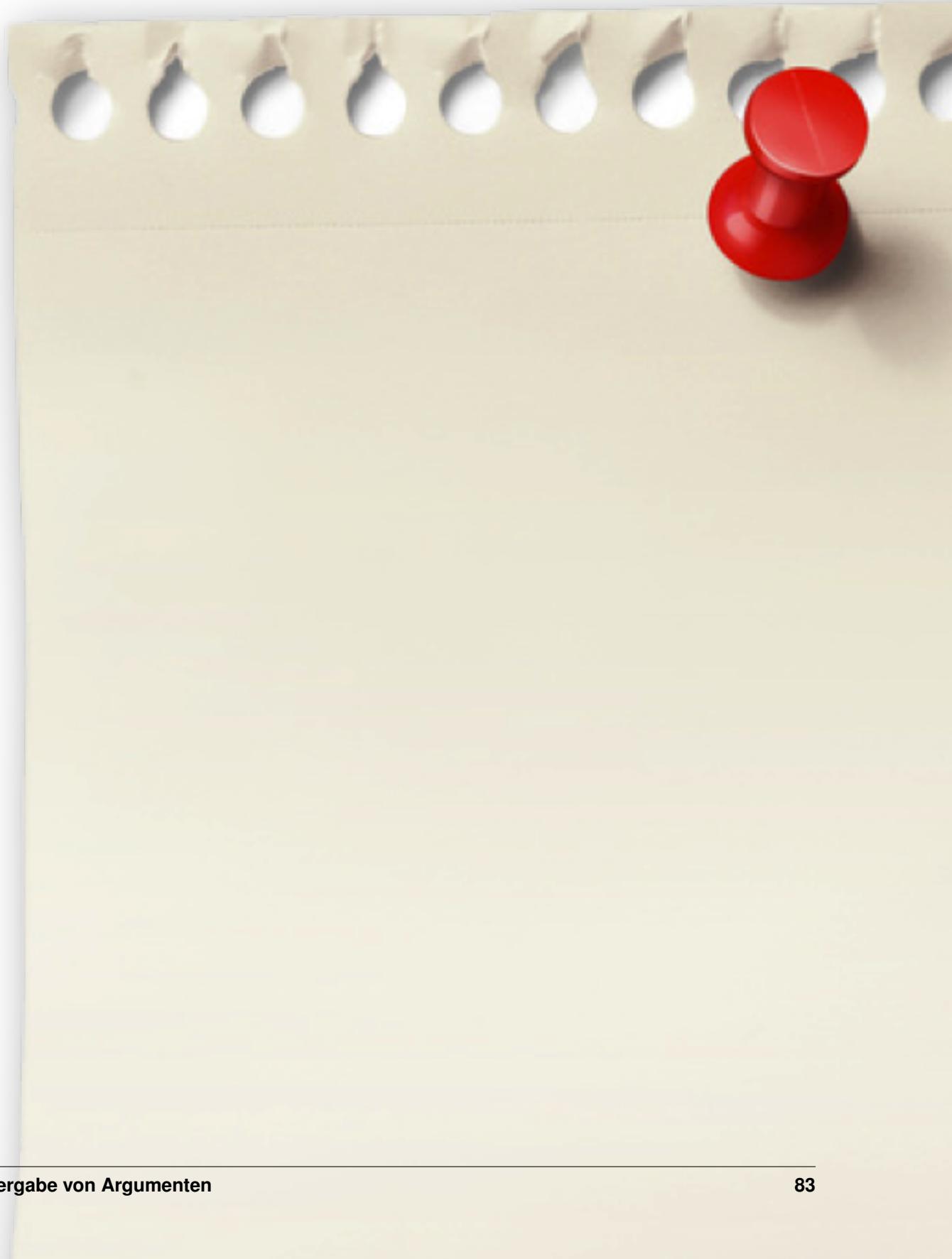
Administrator: Windows PowerShell
PS C:\Windows\system32> function add([int] $value1, [int] $value2)
>>> {
>>>     $value1 + $value2
>>> }
>>>
PS C:\Windows\system32> add hello world
add : Die Argumenttransformation für den Parameter "value1" kann nicht in den Typ "System.Int32" konvertiert werden. Fehler: "Die Eingabezeichen:1 Zeichen:4
Bei Zeile:1 Zeichen:4
+ add <<< hello world
    + CategoryInfo          : InvalidData: (:) [add], ParameterBindin
    + FullyQualifiedErrorId : ParameterArgumentTransformationError,a
PS C:\Windows\system32> add 1 2
3
PS C:\Windows\system32> add 1 "hello"
add : Die Argumenttransformation für den Parameter "value2" kann nicht in den Typ "System.Int32" konvertiert werden. Fehler: "Die Eingabezeichen:1 Zeichen:4
Bei Zeile:1 Zeichen:4
+ add <<< 1 "hello"
    + CategoryInfo          : InvalidData: (:) [add], ParameterBindin
    + FullyQualifiedErrorId : ParameterArgumentTransformationError,a
PS C:\Windows\system32> add "hello" 1
add : Die Argumenttransformation für den Parameter "value1" kann nicht in den Typ "System.Int32" konvertiert werden. Fehler: "Die Eingabezeichen:1 Zeichen:4
Bei Zeile:1 Zeichen:4
+ add <<< "hello" 1
    + CategoryInfo          : InvalidData: (:) [add], ParameterBindin
    + FullyQualifiedErrorId : ParameterArgumentTransformationError,a
PS C:\Windows\system32> add 1.4 1.9
3
PS C:\Windows\system32>
```

Special argument types: Neben herkömmlichen Datentypen können Parameter auch wie ein **switch**-Befehl funktionieren. Wenn der Parametername übergeben wird, dann hat der Parameter den Wert \$true, ansonsten \$false

```

function add($wert1, $wert2, [switch]$help)
{
    if($help)
    {
        Write-Host "Hilfe ausgeben"
    }
}

add -help
```

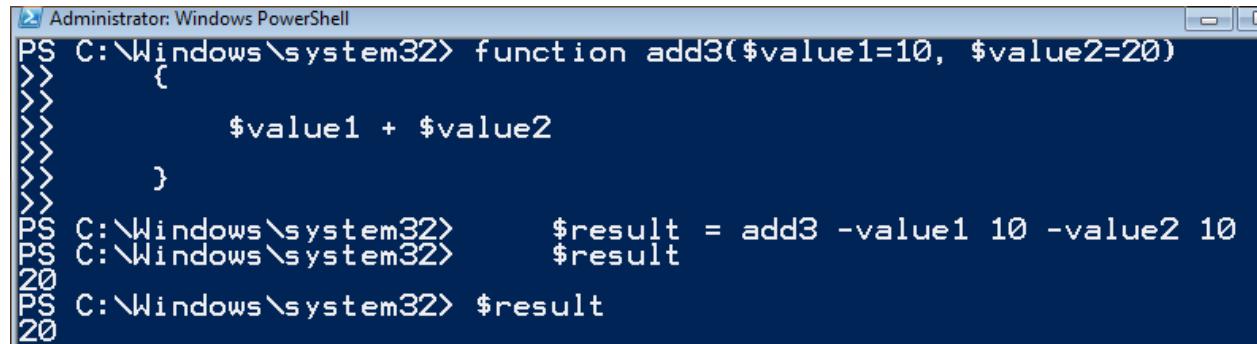


9.3 Rückgabewerte

In **Powershell** geben Funktionen nie nur einen Wert zurück sondern immer alles. Wenn man die Funktion lediglich aufruft, gibt die Funktion die Ausgabe über die Konsole aus. Die Ausgabe kann aber auch in einer Variablen gefangen werden.

```
function add3($value1=10, $value2=20)
{
    $value1 + $value2
}

$result = add3 -value1 10 -value2 10
$result
```



```
Administrator: Windows PowerShell
PS C:\Windows\system32> function add3($value1=10, $value2=20)
>>> {
>>>     $value1 + $value2
>>> }
PS C:\Windows\system32>      $result = add3 -value1 10 -value2 10
PS C:\Windows\system32>      $result
20
PS C:\Windows\system32> $result
20
```

Solange die Funktion nur einen Wert zurückliefert, ist der Rückgabetyp quasi eine Variable. Falls mehrere Ausgaben erfolgen würden, wird die Ausgabe in einen Array gekapselt. Dieser kann von der aufrufenden Seite aus beliebig angepackt werden.

```
function VAT([double]$amount=0)
{
    $amount * 0.19
}
# An interactively invoked function
# output results in the console:
VAT 130.67
24.8273
# But the result of the function can
# also be assign to a variable:
$result = VAT 130.67
$result
24.8273
# The result is a single number value
# of the "double" type:
$result.GetType().Name
Double

Function VAT([double]$amount=0)
{
    $factor = 0.19
    $total = $amount * $factor
    "Value added tax {0:C}" -f $total
    "Value added tax rate: {0:P}" -f $factor
}

The function returns two results:
```

```
VAT 130.67
Value added tax $24.83
Value added tax rate: 19.00%
# All results are stored in a single variable:
$result = VAT 130.67
$result
Value added tax $24.83
Value added tax rate: 19.00%
# Several results are automatically stored in an array:
$result.GetType().Name
Object[]
# You can get each separate result of the
# function by using the index number:
$result[0]
Value added tax $24.83
# The data type of the respective array element
# corresponds to the included data:
$result[0].GetType().Name
String
```

The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The command PS C:\> Function VAT([double]\$amount=0) is entered, followed by a multi-line function body. The command PS C:\> result = VAT 130.0 is then entered, but PowerShell returns an error message: "Die Benennung 'result' wurde nicht als Name eines Cmdlet, einer Funktion grammss erkannt. Überprüfen Sie die Schreibweise des Namens, oder ob der holen Sie den Vorgang." Below this, the command PS C:\> \$result = VAT 130.0 is shown again, followed by PS C:\> \$result, which outputs "Value added tax 24,70 ? Value added tax rate: 19,00%". The command PS C:\> \$result[0] is then entered, outputting "Value added tax 24,70 ?". Finally, PS C:\> \$result.GetType() is entered, showing a table:

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

9.4 Ablaufsteuerung

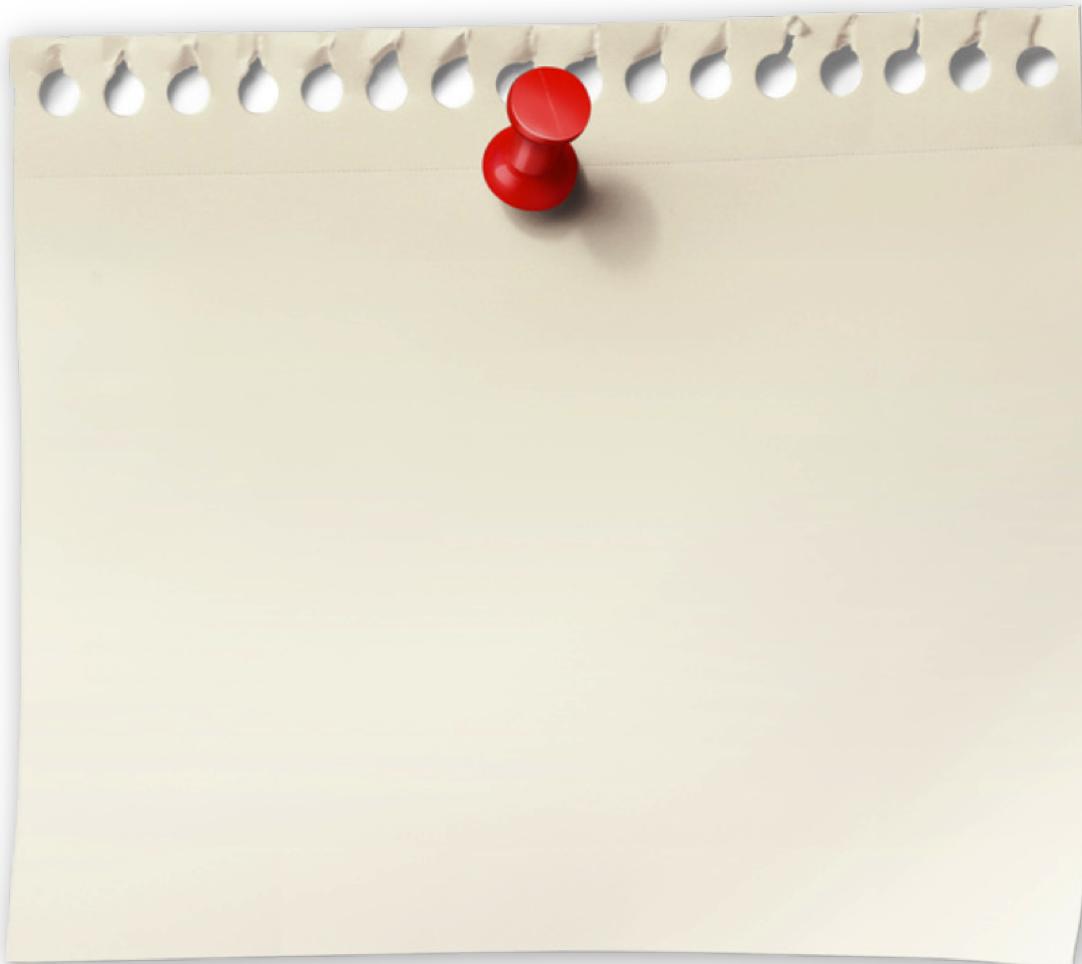
Das Aufteilen des Skriptes in Funktionen teilt den Quellcode in zwei Bereiche auf, nämlich den

- der Funktionalität und
- der Ablaufsteuerung

Da Funktionen zunächst innerhalb des Skriptes überhaupt keine Wirkung besitzen, werden sie erst durch die restlichen Codeteile, der Ablaufsteuerung, zum „Leben erweckt“. Die Verbindung zwischen beiden Teilen erfolgt durch die

- hereingereichten Parameter und die
- zurückgegebenen Ergebnisse.

Tafelbild



9.5 Return Statement

Das RETURN-Statement hat in der **Powershell** nicht die selbe Bedeutung wie in anderen Programmiersprachen, da Funktionen immer alles zurückgeben. Es wurde dennoch implementiert, vor allem aus 2 Gründen:

Style: Man folgte damit den Vorgaben anderer Programmiersprachen, die ebenfalls ein RETURN-Statement haben

Beenden einer Funktion: Return beendet eine Funktion; alle Ausgaben unterhalb des RETURN-Statements würden von der Funktion nicht zurückgegeben werden.

```
Function Add([double]$value1, [double]$value2)
{
```

```

# This time the function returns a whole
# series of oddly assorted results:
"Here the result follows:"
1
2
3
# Return also returns a further result:
$return $value1 + $value2

# This statement will no longer be executed
# because the function will exit when return is used:
"Another text"
}

Add 1 6
Here the result follows:
1
2
3
7
$result = Add 1 6
$result
Here the result follows:
1
2
3
7

```

9.6 Textausgeben verhindern

Häufig hat man bestimmte Ausgaben nur für Kontrollzwecke bzw. Debug-Ausgaben in eine Funktion integriert. Im Produktiveinsatz will man diese Ausgaben nicht sehen und benötigt sie auch nicht.

Folgendes Beispiel zeigt mögliche Alternativen:

```

Function Test
{
    "Calculation will be performed"
    $a = 12 * 10
    "Result will be emitted"
    "Result is: $a"
    "Done"
}
Test
Calculation will be performed
Result will be emitted
Result is: 120
Done

$result = Test
$result

```

Für diesen Zweck gibt es mehrere Möglichkeiten.

Write-Host benutzen: Das Cmdlet Write-Host gibt die Ausgabe sofort an die Konsole weiter

```
Function Test
{
    Write-Host "Calculation will be performed"
    $a = 12 * 10
    Write-Host "Result will be emitted"
    "Result is: $a"
    Write-Host "Done"
}
# This time your debugging reports will already
# be output when the function is executed:
$result = test
Calculation will be performed
Result will be emitted
Done
# The result will no longer include your debugging reports:
$result
Result is: 120
```

Debug-Ausgabe benutzen Mit Hilfe des Write-Debug-Cmdlets können Ausgaben nur zu Debug-Zwecken ausgegeben werden. Voraussetzung ist allerdings, dass man die globale Variabel \$DebugPreference auf „SilentlyContinue“ stellt

```
Function Test
{
    Write-Debug "Calculation will be performed"
    $a = 12 * 10
    Write-Debug "Result will be emitted"
    "Result is: $a"
    Write-Debug "Done"
}
# Debugging reports will remain completely
# invisible in the production environment:
$result = Test
# If you would like to debug your function,
# turn on reporting:
$DebugPreference = "Continue"
# Your debugging reports will now be output
# with the "DEBUG:" prefix and output in yellow:
$result = Test
DEBUG: Calculation will be performed
DEBUG: Result will be emitted
DEBUG: Done
# They are not contained in the result:
$result
Result is: 120
# Everything is running the way you wish;
# turn off debugging:
$DebugPreference = "SilentlyContinue"
$result = Test
```

Fehlermeldungen unterdrücken: Fehlermeldungen werden normalerweise immer sofort ausgegeben, was natürlich den Ablauf eines Skriptes stören kann. Auch hier gibt es wieder eine globale Variable, welche die Ausgabe von Fehlermeldungen steuern kann.

```
Function Test
{
    # Suppress all error messages from now on:
    $ErrorActionPreference = "SilentlyContinue"
```

```

Stop-Process -name "Unavailableprocess"
# Immediately begin outputting all error messages again:
$ErrorActionPreference = "Continue"
1/$null
}
# Error messages will be suppressed in certain
# areas but not in others:
$result = Test

```

9.7 Inline-Help von Funktionen

Um den Nutzer einer Funktion auch innerhalb der Powershell mit Informationen über die Funktion versorgen zu können, kann man sowohl Funktionen als auch das ganze Skript mit einer sog. **Inline-Help** ausstatten. Im Grunde handelt es sich dabei um eine Abfolge von Kommentarzeilen, die mit bestimmten Schlüsselworten versehen worden sind.

Details der Anwendung gibt es unter <http://technet.microsoft.com/en-us/library/dd819489.aspx>

```

<#
.SYNOPSIS
    Dieses Skript gibt eine Liste der nicht funktionierenden
    Hardware aus mit Hilfe von WMI.
.DESCRIPTION
    Per WMI werden zunächst die Systemdetails ermittelt
    und dann die Hardware mit Fehlern.
.NOTES
    File Name   : Get-BrokenHardware.ps1
    Author      : Karl Steinam - teetscher
    Requires    : PowerShell Version 2.0
.LINK
    Siehe auch:
        http://www.meineFirma.de
.EXAMPLE
    PSH [C:\foo]: Get-BrokenHardware.ps1
    Computer Details:
    Manufacturer: Dell Inc.
    Model:        Precision WorkStation T7400
    Service Tag:  6Y84C3J

    Hardware that's not working list
    Description: WD My Book Device USB Device
    Device ID:   USBSTOR\OTHER&VEN_WD&P\70332&1
    Error ID:    28
#>

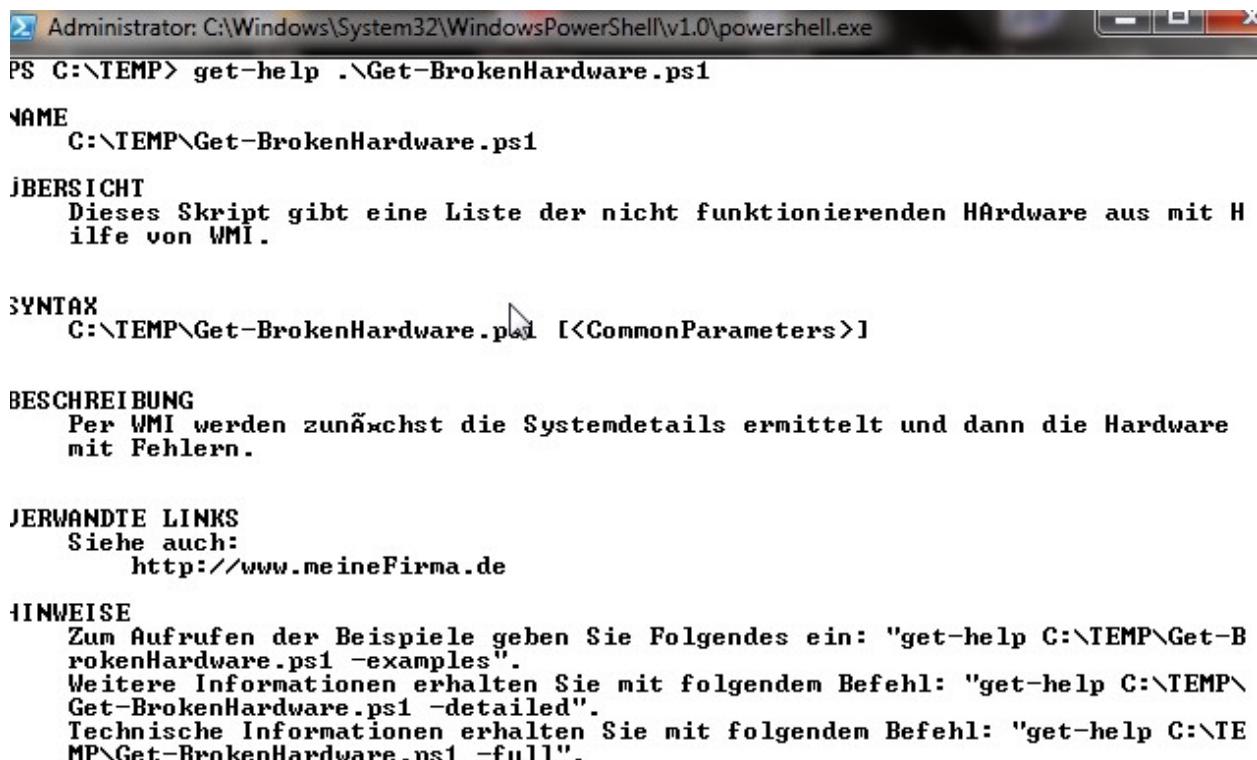
# Display Computer details
"Computer Details:"
$comp = gwmi Win32_ComputerSystem
"Manufacturer: {0}" -f $comp.Manufacturer
"Model:       {0}" -f $comp.Model
$computer2 = Get-WmiObject Win32_ComputerSystemProduct
"Service Tag: {0}" -f $computer2.IdentifyingNumber
"""

#Get hardware that is errored
"Hardware that's not working list"

```

```
$broken = Get-WmiObject Win32_PnPEntity | where ` 
{$_._ConfigManagerErrorCode -ne 0} `

#Display broken hardware
foreach ($obj in $broken){
    "Description: {0}" -f $obj.Description
    "Device ID: {0}" -f $obj.DeviceID
    "Error ID: {0}" -f $obj.ConfigManagerErrorCode
    ""
}
```



The screenshot shows a Windows PowerShell window titled "Administrator: C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe". The command "get-help .\Get-BrokenHardware.ps1" has been run. The output is as follows:

NAME
C:\TEMP\Get-BrokenHardware.ps1

BESCHREIBUNG
Dieses Skript gibt eine Liste der nicht funktionierenden Hardware aus mit Hilfe von WMI.

SYNTAX
C:\TEMP\Get-BrokenHardware.ps1 [**<CommonParameters>**]

BESCHREIBUNG
Per WMI werden zunächst die Systemdetails ermittelt und dann die Hardware mit Fehlern.

VERWANDTE LINKS
Siehe auch:
<http://www.meineFirma.de>

HINWEISE
Zum Aufrufen der Beispiele geben Sie Folgendes ein: "get-help C:\TEMP\Get-BrokenHardware.ps1 -examples".
Weitere Informationen erhalten Sie mit folgendem Befehl: "get-help C:\TEMP\Get-BrokenHardware.ps1 -detailed".
Technische Informationen erhalten Sie mit folgendem Befehl: "get-help C:\TEMP\Get-BrokenHardware.ps1 -full".

```
PS C:\TEMP> get-help .\Get-BrokenHardware.ps1 -examples
NAME
  C:\TEMP\Get-BrokenHardware.ps1
ÜBERSICHT
  Dieses Skript gibt eine Liste der nicht funktionierenden Hardware aus mit H
ilfe von WMI.

----- BEISPIEL 1 -----
C:\PS>PSH [C:\foo]: Get-BrokenHardware.ps1

Computer Details:
Manufacturer: Dell Inc.
Model: Precision WorkStation T7400
Service Tag: 6Y84C3J

Hardware that's not working list
Description: WD My Book Device USB Device
Device ID: USBSTOR\OTHER&VEN_WD&PROD_MY_BOOK_DEVICE&REV_1010\7&2A4E07C&0
&575532513130303732383932&1
Error ID: 28
```

Das nächste Beispiel fügt einer Funktion eine Inline-Help hinzu.

```
<#
    .SYNOPSIS
    Adds a file name extension to a supplied name.

    .DESCRIPTION
    Adds a file name extension to a supplied name.
    Takes any strings for the file name or extension.

    .PARAMETER Name
    Specifies the file name.

    .PARAMETER Extension
    Specifies the extension. "Txt" is the default.

    .INPUTS
    None. You cannot pipe objects to Add-Extension.

    .OUTPUTS
    System.String. Add-Extension returns a string
    with the extension or file name.

    .EXAMPLE
    C:\PS> extension -name "File"
    File.txt

    .EXAMPLE
    C:\PS> extension "File" "doc"
    File.doc

    .LINK
    http://www.fabrikam.com/extension.html

    .LINK
    Set-Item
#>
function Add-Extension2
```

```
{  
    param ([string]$Name, [string]$Extension = "txt")  
    $name = $name + "." + $extension  
    $name  
}
```

Je nach Laune kann die Inline-Help auch innerhalb der Funktion geschrieben werden.

9.8 Größeres Beispiel bzgl. Funktionen

Siehe:

<http://www.powershellpro.com/powershell-tutorial-introduction/powershell-scripting-with-wmi/>

<http://www.powershellpro.com/computernames-activedirectory/149/>

http://www.computerperformance.co.uk/powershell/powershell_example_basic.htm#Example_4:_PowerShell_Real-life_Task

<http://www.powershellpro.com/powershell-tutorial-introduction/powershell-scripting-with-wmi/>

<http://technobeanz.wordpress.com/2010/11/25/powershell-getting-inventory-details/>

Voraussetzung:

Installiertes CIM-Studio zur Abfrage des WMI

Im folgenden Beispiel sollen mit Hilfe von WMI verschiedene Informationen des lokalen Rechners abgerufen werden. In weiteren Schritten wird die Suche auf beliebige Rechner ausgeweitet. Im letzten Schritt werden Informationen des ActiveDirectory genutzt, um eine komfortable Liste der Rechner zu erhalten.

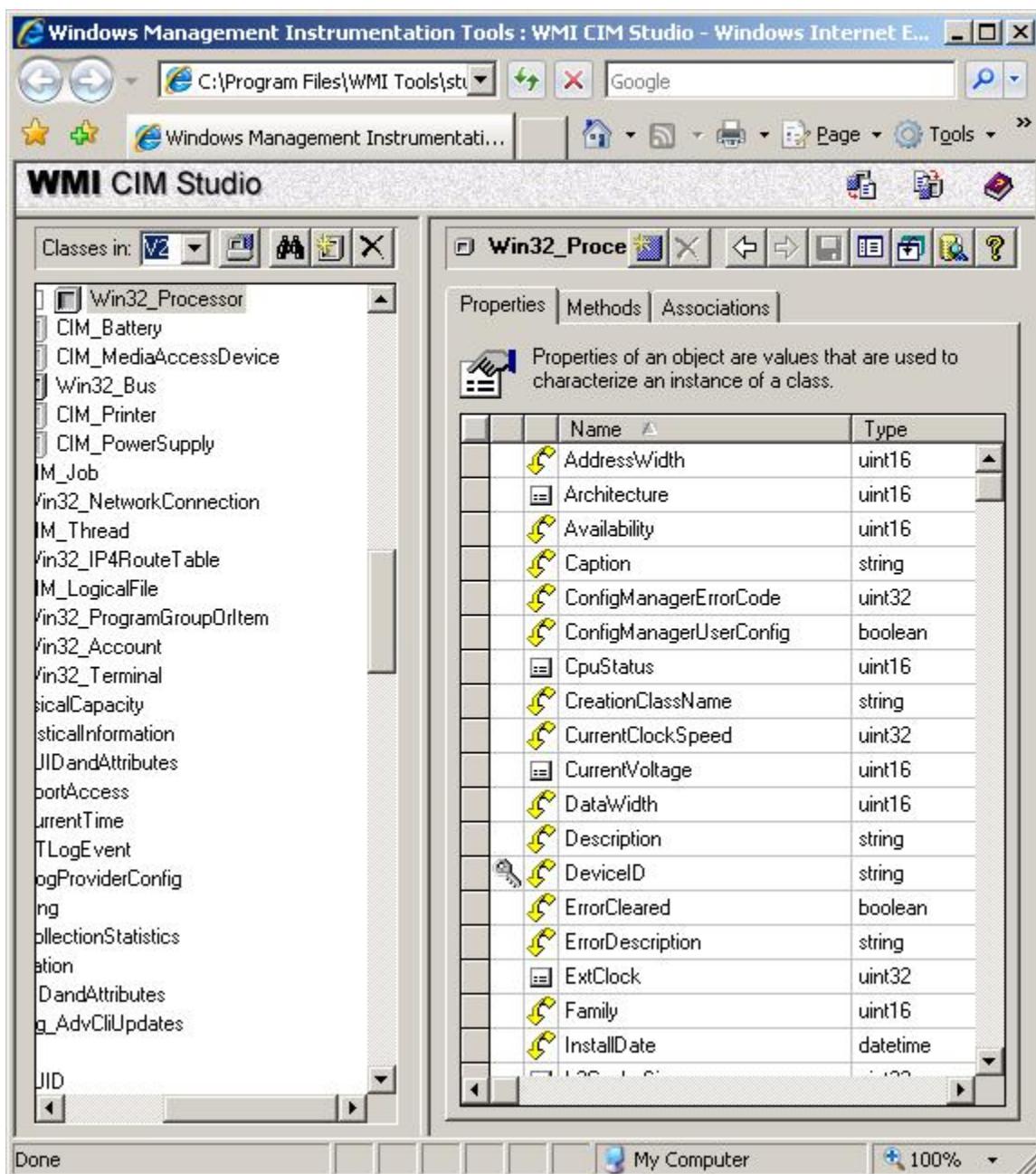
Scenario:

Ihr Chef will eine Inventur der Hardware aller Server und Rechner im Netzwerk. Insbesondere will er folgende Informationen.

- Machine manufacturer, model number, and serial number.
- BIOS information to determine if updates are required,
- OS type
- CPU information: Manufacturer, type, speed, and version.
- Amount of memory in each server.
- Disk information: Size, interface type, and media type.
- Network Information: IP settings and MAC address.

9.9 Benutzung von WMI

WMI ist im Grunde eine Datenbank von Informationen, das auf jeden Windows System vorhanden ist. Über den sog. WMI-Service kann man die jeweils gewünschten Daten abrufen.



Von Powershell aus ist die Vorgehensweise relativ einfach: :-)

```
Get-WmiObject -List -Namespace "root\CIMV2"
```

```

Windows PowerShell
Copyright (C) 2006 Microsoft Corporation. All rights reserved.

PS C:\MyScripts> Get-WmiObject -List -Namespace "root\cimv2"

__SystemClass
__Provider
__ProviderRegistration
__EventConsumerProviderRegistration
__ObjectProviderRegistration
__InstanceProviderRegistration
__NAMESPACE
__IndicationRelated
__EventConsumer
CmdTriggerConsumer
__EventGenerator
__IntervalTimerInstruction
__Event
MSFT_WmiSelfEvent
Msft_WmiProvider_ComServerLoadOper...
Msft_WmiProvider_LoadOperationEvent
Msft_WmiProvider_DeleteClassAsyncE...
Msft_WmiProvider_AccessCheck_Pre
Msft_WmiProvider_ExecQueryAsyncEve...
Msft_WmiProvider_NewQuery_Pre
Msft_WmiProvider_CancelQuery_Pre
Msft_WmiProvider_PutClassAsyncEven...
Msft_WmiProvider_ExecMethodAsyncEv...
Msft_WmiProvider_ComServerLoadOper...
Msft_WmiProvider_LoadOperationFail...
Msft_WmiProvider_ProvideEvents_Post
Msft_WmiProvider_CancelQuery_Post
Msft_WmiProvider_DeleteClassAsyncE...
Msft_WmiProvider_PutInstanceAsyncE...
Msft_WmiProvider_CreateInstanceEnum
Msft_WmiProvider_ExecQueryAsyncEve...
Msft_WmiProvider_ExecMethodAsyncEv...
MSFT_WmiRegisterNotificationSink
MSFT_WmiThreadpoolThreadCreated
MSFT_WmiFilterEvent
MSFT_WmiFilterDeactivated
MSFT_WmiConsumerProviderEvent
MSFT_WmiConsumerProviderSinkUnloaded
MSFT_WmiConsumerProviderSinkLoaded
MSFT_WmiCoreEvent
MSFT_WmiCoreLogoffEvent
MSFT_WMI_GenericNonCOMEvent
MSFT_NCPProvNewQuery
MSFT_NCPProvCancelQuery
    __thisNAMESPACE
    __Win32Provider
    __EventProviderRegistration
    __PropertyProviderRegistration
    __ClassProviderRegistration
    __MethodProviderRegistration
    __WmiMappedDriverNamespace
    __EventFilter
    EventViewerConsumer
    __FilterToConsumerBinding
    __TimerInstruction
    __AbsoluteTimerInstruction
    __ExtrinsicEvent
    Msft_WmiProvider_OperationEvent
    Msft_WmiProvider_InitializationOp...
    Msft_WmiProvider_OperationEvent_Pre
    Msft_WmiProvider_GetObjectAsyncEv...
    Msft_WmiProvider_CreateClassEnumA...
    Msft_WmiProvider.CreateInstanceEn...
    Msft_WmiProvider_DeleteInstanceAs...
    Msft_WmiProvider_PutInstanceAsync...
    Msft_WmiProvider_ProvideEvents_Pre
    Msft_WmiProvider_InitializationOp...
    Msft_WmiProvider_UnLoadOperationE...
    Msft_WmiProvider_OperationEvent_Post
    Msft_WmiProvider_GetObjectAsyncEv...
    Msft_WmiProvider_AccessCheck_Post
    Msft_WmiProvider_DeleteInstanceAs...
    Msft_WmiProvider_NewQuery_Post
    Msft_WmiProvider_PutClassAsyncEve...
    Msft_WmiProvider_CreateClassEnumA...
    MSFT_WmiEvent
    MSFT_WmiThreadpoolEvent
    MSFT_WmiThreadpoolThreadDeleted
    MSFT_WmiFilterActivated
    MSFT_WmiProviderEvent
    MSFT_WmiConsumerProviderLoaded
    MSFT_WmiConsumerProviderUnloaded
    MSFT_WmiCancelNotificationSink
    MSFT_WmiCoreTaskFailure
    MSFT_WmiCoreLogonEvent
    MSFT_NCPProvEvent
    MSFT_NCPProvAccessCheck
    MSFT_NCPProvClientConnected

```

Eine etwas kleinere Ausgabe gibt es, wenn man die Ausgabe auf bestimmte Inhalte beschränkt, in unserem Beispiel

```
Get-WmiObject Win32_ComputerSystem | Format-List *
```

```

Administrator: C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
Domain : WORKGROUP
DomainRole : 0
EnableDaylightSavingsTime : True
InfraredSupported : False
InitialLoadInfo :
InstallDate :
LastLoadInfo :
Manufacturer : ASUSTeK Computer INC.
Model : 1000HE
NameFormat :
NetworkServerModeEnabled : True
NumberOfLogicalProcessors : 2
NumberOfProcessors : 1
OEMLogoBitmap :
OEMStringArray : {To Be Filled By O.E.M., To Be Filled By O.E.M., To Be Filled By O.E.M., To Be Filled By O.E.M.}
PartOfDomain :
PauseAfterReset :
PCSystemType :
PrimaryOwnerContact :
PrimaryOwnerName : steinam
ResetCapability : 1
ResetCount : -1
ResetLimit : -1
Roles : {LM_Workstation, LM_Server, SQLServer, NT...}
SupportContactDescription :
SystemStartupDelay :
SystemStartupOptions :
SystemStartupSetting :
SystemType : x86-based PC
TotalPhysicalMemory : 2138300416
UserName : steinam-PC\steinam
WakeUpType : 6
Workgroup : WORKGROUP
Scope : System.Management.ManagementScope

```

Welche Informationen entsprechen den Anforderungen ?

- Manufacturer = Manufacturer property.
- Model Number = Model property.
- Serial Number = Nicht vorhanden; sie muss anderweitig gesucht werden
- Gesamtspeicher = TotalPhysicalMemory-Eigenschaft existiert in der Klasse und genügt unseren Ansprüchen.

Schritt 1: Machine manufacturer, model number, and serial number

```

1 #sets the computer to local
2 $strComputer = "."
3
4 #Creates a variable called $colItems which contains the WMI Object
5 $colItems = Get-WmiObject Win32_ComputerSystem -Namespace "root\CIMV2" -
6 ComputerName $strComputer
7
8 #Use a foreach loop to iterate the $colItems (collection).
9 #Store the properties information in the $objItem Variable.
10 foreach($objItem in $colItems) {
11     #Use the Write-Host cmdlet to output required property information
12     Write-Host "Computer Manufacturer: " $objItem.Manufacturer
13     Write-Host "Computer Model: " $objItem.Model
14     Write-Host "Total Memory: " $objItem.TotalPhysicalMemory "bytes"
15 }

```

Schritt 2: BIOS-Information

Die entsprechende Klasse im WMI ist Win32_BIOS

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

```
1 $strComputer = "."
2
3     $colItems = Get-WmiObject Win32_BIOS -Namespace "root\CIMV2" -ComputerName
4     ↳$strComputer
5     foreach($objItem in $colItems)
6     {
7         Write-Host "BIOS:"$objItem.Description
8         Write-Host "Version:"$objItem.SMBIOSBIOSVersion"."
9         $objItem.SMBIOSMajorVersion"."$objItem.SMBIOSMinorVersion
10        Write-Host "Serial Number:" $objItem.SerialNumber
11    }
```

Schritt 3: Übrige Informationen

OS TYPE:

```
1 $strComputer = "."
2
3     $colItems = Get-WmiObject Win32_OperatingSystem -Namespace "root\CIMV2" -
4     ↳Computername $strComputer
5
6     foreach($objItem in $colItems) {
7         Write-Host "Operating System:" $objItem.Name
8     }
```

CPU Info:

```
1 $strComputer = "."
2
3     $colItems = Get-WmiObject Win32_Processor -Namespace "root\CIMV2" -ComputerName
4     ↳$strComputer
5
6     foreach($objItem in $colItems) {
7         Write-Host "Processor:" $objItem.DeviceID $objItem.Name
8     }
```

DISK Info:

```
1 $strComputer = "."
2
3     $colItems = Get-WmiObject Win32_DiskDrive -Namespace "root\CIMV2" -ComputerName
4     ↳$strComputer
5
6     foreach($objItem in $colItems)
7     {
8         Write-Host "Disk:" $objItem.DeviceID
9         Write-Host "Size:" $objItem.Size "bytes"
10        Write-Host "Drive Type:" $objItem.InterfaceType
11        Write-Host "Media Type: " $objItem.MediaType
12    }
```

NETWORK Info:

```
1 $strComputer = "."
2
3     $colItems = Get-WmiObject Win32_NetworkAdapterConfiguration -Namespace
4     ↳"root\CIMV2"
```

```

4      -ComputerName $strComputer | where{$_.'IPEnabled' -eq "True"}
5
6      foreach($objItem in $colItems) {
7          Write-Host "DHCP Enabled:" $objItem.DHCPEnabled
8          Write-Host "Subnet Mask:" $objItem.IPSubnet
9          Write-Host "Gateway:" $objItem.DefaultIPGateway
10         Write-Host "MAC Address:" $objItem.MACAddress
11     }

```

Kapseln der Code-Schnipsel in Funktionen

Zur übersichtlicheren Handhabung werden wir die einzelnen Codeelemente in Funktionen kapseln. Alle Funktionen werden in die Datei ServerInventory.ps1 geschrieben.

```

1 Function SysInfo
2 {
3     $colItems = Get-WmiObject Win32_ComputerSystem ` 
4             -Namespace "root\CIMV2" -ComputerName $strComputer ` 
5
6     foreach($objItem in $colItems) {
7         Write-Host "Computer Manufacturer: " $objItem.Manufacturer
8         Write-Host "Computer Model: " $objItem.Model
9         Write-Host "Total Memory: " $objItem.TotalPhysicalMemory "bytes"
10    }
11 }
12
13 Function BIOSInfo {
14
15     $colItems = Get-WmiObject Win32_BIOS -Namespace "root\CIMV2" ` 
16             -Computername $strComputer ` 
17     foreach($objItem in $colItems) {
18         Write-Host "BIOS:$objItem.Description"
19         Write-Host "Version:$objItem.SMBIOSBIOSVersion."
20         $objItem.SMBIOSMajorVersion".$objItem.SMBIOSMinorVersion
21         Write-Host "Serial Number:$objItem.SerialNumber"
22    }
23 }
24
25
26 Function OSInfo
27 {
28     $colItems = Get-WmiObject Win32_OperatingSystem -Namespace ` "root\CIMV2` 
29             -Computername $strComputer ` 
30
31     foreach($objItem in $colItems) {
32         Write-Host "Operating System:" $objItem.Name
33    }
34 }
35
36 Function CPUInfo
37 {
38     $colItems = Get-WmiObject Win32_Processor -Namespace ` 
39             "root\CIMV2" -Computername $strComputer ` 
40
41     foreach($objItem in $colItems) {
42         Write-Host "Processor:" $objItem.DeviceID $objItem.Name
43    }
}

```

```

44 }
45
46
47     Function DiskInfo
48     {
49         $colItems = Get-WmiObject Win32_DiskDrive -Namespace ` "root\CIMV2" -
50             ComputerName $strComputer ` 
51
52         foreach($objItem in $colItems) {
53             Write-Host "Disk:" $objItem.DeviceID
54             Write-Host "Size:" $objItem.Size "bytes"
55             Write-Host "Drive Type:" $objItem.InterfaceType
56             Write-Host "Media Type: " $objItem.MediaType
57         }
58     }
59
60     Function NetworkInfo
61     {
62         $colItems = Get-WmiObject Win32_NetworkAdapterConfiguration ` -Namespace
63             "root\CIMV2" -ComputerName $strComputer | ` where{$_.IPEnabled -eq "True"}
64
65         foreach($objItem in $colItems) {
66             Write-Host "DHCP Enabled:" $objItem.DHCPEnabled
67             Write-Host "IP Address:" $objItem.IPAddress
68             Write-Host "Subnet Mask:" $objItem.IPSubnet
69             Write-Host "Gateway:" $objItem.DefaultIPGateway
70             Write-Host "MAC Address:" $objItem.MACAddress
71     }
72 }
```

In Powershell müssen die Funktionen vor der ersten Benutzung definiert worden sein. Der Aufruf des Funktionen erfolgt dann einfach **am Ende** des Skriptes.

```

1 #=====
2 #* SCRIPT BODY
3 #=====
4 #* Connect to computer
5 $strComputer = "."
6
7 #* Call SysInfo Function
8 Write-Host "System Information"
9 SysInfo
10 Write-Host
11
12 #* Call BIOSinfo Function
13 Write-Host "System BIOS Information"
14 BIOSInfo
15 Write-Host
16
17 #* Call OSInfo Function
18 Write-Host "Operating System Information"
19 OSInfo
20 Write-Host
21
22 #* Call CPUInfo Function
23 Write-Host "Processor Information"
24 CPUInfo
```

```

25 Write-Host
26
27 #* Call DiskInfo Function
28 Write-Host "Disk Information"
29 DiskInfo
30 Write-Host
31
32 #* Call NetworkInfo Function
33 Write-Host "Network Information"
34 NetworkInfo
35 Write-Host
36
37 #=====
38 #* END OF SCRIPT: [ServerInventory]
39 #=====

```

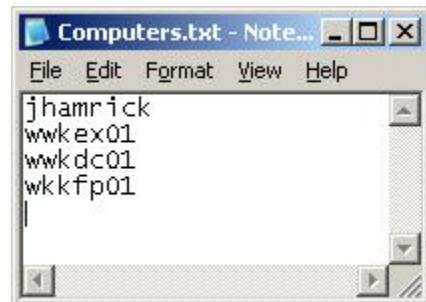
Abfrage eines anderen Computers

Zur Zeit fragt das Skript nur die Daten des eigenen Computers ab. Um auch remote arbeiten zu können, muss die Variable \$strComputer ersetzt werden.

```
$strComputer = Read-Host "Enter Computer Name"
Write-Host "Computer:" $strComputer
```

Optimierungen

Erzeugen Sie eine Textdatei und fügen Sie Computernamen ein. Erweiteren Sie das Skript so, dass es die Textdatei ausliest und den jeweiligen Computer abfragt.



```

1 #=====
2 #* SCRIPT BODY
3 #=====
4 #* Create and array from C:\MyScripts\Computers.txt
5
6 $arrComputers = get-Content -Path "C:\MyScripts\Computers.txt"
7
8 foreach ($strComputer in $arrComputers)
9 { #Function Calls go here
10
11     Write-Host "Computer Name:" $strComputer
12     Write-Host "===="
13
14     #* Call SysInfo Function
15     Write-Host "System Information"
16     SysInfo
17     Write-Host
18

```

```
19      #* Call BIOSinfo Function
20      Write-Host "System BIOS Information"
21      BIOSInfo
22      Write-Host
23
24      #* Call OSInfo Function
25      Write-Host "Operating System Information"
26      OSInfo
27      Write-Host
28
29      #* Call CPUInfo Function
30      Write-Host "Processor Information"
31      CPUInfo
32      Write-Host
33
34      #* Call DiskInfo Function
35      Write-Host "Disk Information"
36      DiskInfo
37      Write-Host
38
39      #* Call NetworkInfo Function
40      Write-Host "Network Information"
41      NetworkInfo
42      Write-Host "End of Report for $strComputer"
43      Write-Host "====="
44      Write-Host
45      Write-Host
46
47 } #End function calls.
```

Auslesen der Computer aus dem AD

Um sich die Eingabe der Computer in eine Textdatei zu ersparen, kann man auch innerhalb einer Windows-Domäne das ActiveDirectory befragen. Folgendes Skript liefert die Daten der Computernamen innerhalb des AD.

```
1      #GetPCNames.ps1
2
3      $strCategory = "computer"
4
5      $objDomain = New-Object System.DirectoryServices.DirectoryEntry
6
7      $objSearcher = New-Object System.DirectoryServices.DirectorySearcher
8      $objSearcher.SearchRoot = $objDomain
9      $objSearcher.Filter = ("(objectCategory=$strCategory)")
10
11     $colPropList = "name"
12     foreach ($i in $colPropList)
13     {
14         $objSearcher.PropertiesToLoad.Add($i)
15     }
16
17     $colResults = $objSearcher.FindAll()
18
19     foreach ($objResult in $colResults)
20     {
21         $objComputer = $objResult.Properties; $objComputer.name
22     }
```

Speichern Sie das Skript als GetPCNames.ps1, starten Sie es innerhalb ihrer Domäne und speichern Sie das Ergebnis in der Datei Computers.txt

```
. \GetPCNames.ps1 > "C:\MyScripts\Computers.txt"
```

9.10 Erweiterte Parameter

Validierung

Häufig ist es so, dass die Parameter einer Funktion einer Überprüfung bedürften, bevor diese innerhalb einer Funktion benutzt werden. Dies könnte z.B. sein

- Ist überhaupt ein Parameter übergeben worden
- Hat der Parameter die richtigen Werte
-

In der Regel führt dies dazu, dass der Quellcode innerhalb der Funktion durch Fehlerbehandlungscode erweitert wird, bevor man überhaupt die eigentliche Aufgabe löst.

Bsp:

Im unteren Beispiel soll eine Funktion nur bestimmte Werte für die Parameter akzeptieren sollen.

- Name: Tom, Dick, Jane
- Alter: Zwischen 21 und 65
- Übergebener Dateipfad soll vorhanden sein

<http://blogs.technet.com/b/heyscriptingguy/archive/2011/05/15/simplify-your-powershell-script-with-parameter-validation.aspx>

```
Function Foo
{
    Param(
        [String]
        $Name,
        [Int]
        $Age,
        [string]
        $Path
    )
    Process
    {
        If ("Tom","Dick","Jane" -NotContains $Name)
        {
            Throw "$($Name) is not a valid name! Please use Tom, Dick, Jane"
        }
        If ($age -lt 21 -OR $age -gt 65)
        {
            Throw "$($age) is not a between 21-65"
        }
        IF (-NOT (Test-Path $Path -PathType 'Container'))
        {
            Throw "$($Path) is not a valid folder"
        }
        # All parameters are valid so New-stuff"
        write-host "New-Foo"
    }
}
```

```
}
```

Das (noch nicht bekannte) **throw**-Statement gibt eine Fehlermeldung aus und beendet die Funktion.

Diese Vorgehensweise hat mehrere Nachteile:

- Ein Großteil des Skriptes ist der Fehlerbehandlung gewidmet.
- Die Validierung ist nur so gut wie der sie validierende Quellcode
- Die Fehlermeldungen müssen bei Änderungen immer wieder angepasst werden.

Aus all diesen Gründen wurden in der Powershell die Möglichkeit der **Parameter-Validierung** eingebaut. Zu den Parametern werden in eckigen Klammern weitere Informationen hinzugefügt, die von der PS zur Laufzeit ausgewertet werden können.

Besser gescripted, sieht der Quellcode wie folgt aus:

```
Function Foo
{
    Param(
        [ValidateSet("Tom", "Dick", "Jane")]
        [String]
        $Name,
        [ValidateRange(21, 65)]
        [Int]
        $Age,
        [ValidateScript({Test-Path $_ -PathType 'Container'})]
        [string]
        $Path
    )
    Process
    {
        write-host "New-Foo"
    }
}
```

Der Vorteil dieser Methode ist, dass die Fehlerbehandlung nun implizit von der PS vorgenommen wird, inkl. der Bereitstellung und Ausgabe der Fehlermeldungen. Insgesamt stehen über 11 Validierungsoptionen zur Verfügung; diese können unter <http://technet.microsoft.com/en-us/library/dd347600.aspx> bzw über **get-help about_functions** abgefragt werden.

Mandatory

Das Mandatory-Argument gibt an, dass ein Parameter erwartet wird. Falls kein Parameter übergeben wird, bricht die Funktion ab.

```
Function Foo
{
    Param
    (
        [parameter(Mandatory=$true)]
        [String[]]
        $ComputerName
    )
}
```

Häufig wird das Manadatory-Argument durch das Help-Message-Argument ergänzt, welches eine Fehlerbeschreibung zum nicht übergebenen Mandatory-Parameter hinzufügt.

Bsp:

```
Function Foo
{
    Param
    (
        [parameter(mandatory=$true,
                    HelpMessage="Enter one or more computer names separated by commas.
                    ")]
        [String[]]
        $ComputerName
    )
}
```

9.11 Funktionen in Profilen

<http://www.howtogeek.com/126469/how-to-create-a-powershell-profile/>

Häufig hat es ein Admin gerne, wenn beim Start der Powershell-Konsole bereits bestimmte Grundfunktionalitäten vorhanden sind. PS kennt hierfür das Konzept der Profildateien, die automatisch beim Start der PS mit geladen werden.

Zunächst sollte geprüft werden, ob es bereits eine Profildatei gibt. Dies kann man durch Ausgabe der UmgebungsvARIABLE \$Profile erreichen.

```
Test-Path $Profile
```

Falls noch keine Datei existiert kann diese mit jedem beliebigem Editor erzeugt werden, ansonsten auch mit

```
New-Item -Path $Profile -Type File -Force
```

In diese Datei kann jeder Befehl, CommandLet, Modul-Import eingesetzt werden. Folgendes Beispiel verdeutlicht die Möglichkeiten.

```
Set-ExecutionPolicy unrestricted
$ui = (Get-Host).UI.RawUI
$ui.ForegroundColor = "Black"
$ui.BackgroundColor = "Gray"
$ui.WindowTitle = $env:userdomain + "\\" + $env:username + "'s Powershell"

function prompt
{
    Write-Host "PS " -nonewline -foregroundcolor Red
    Write-Host $(get-location) "#" -nonewline
    return " "
}

$snapins = Get-PSSnapin -Registered
$snapins | Add-PSSnapin
Get-Module -ListAvailable | Import-Module
Get-PSSnapin | Format-Table -autosize PSVersion, Name
Get-Module | Format-Table -autosize ModuleType, Name
function ff ([string] $glob) { get-childitem -recurse -include $glob }
function osr { shutdown -r -t 5 }
function osh { shutdown -h -t 5 }
function rmd ([string] $glob) { remove-item -recurse -force $glob }
function whoami { (get-content env:\userdomain) + "\\" + `
```

```
(get-content env:\username); } `

function strip-extension ([string] $filename)
{
    [system.io.path]::getfilenamewithoutextension($filename)
}

function New-PSSecureRemoteSession
{
    param ($sshServerName, $Cred)
    $Session = New-PSSession $sshServerName -UseSSL ` 
    -Credential $Cred -ConfigurationName C2Remote `

    Enter-PSSession -Session $Session
}

function New-PSRemoteSession
{
    param ($shServerName, $Cred)
    $shSession = New-PSSession $shServerName ` 
    -Credential $Cred -ConfigurationName C2Remote `

    Enter-PSSession -Session $shSession
}

function prompt
{
    $promptText = "PS>";
    $wi = [System.Security.Principal.WindowsIdentity]::GetCurrent()
    $wp = new-object 'System.Security.Principal.WindowsPrincipal' $wi

    if ( $wp.IsInRole("Administrators") -eq 1 )
    {
        $color = "Red"
        $title = "***ADMIN** on " + (hostname);
    }
    else
    {
        $color = "Green"
        $title = hostname;
    }
    write-host $promptText -NoNewLine -ForegroundColor $color
    $host.UI.RawUI.WindowTitle = $title;
    return " "
}

function gotoprofile{set-location C:\Users\paul.cowan\Documents\WindowsPowerShell}

function gotodownloads{set-location C:\Users\paul.cowan\Downloads}
function openscratch{notepad C:\users\paul.cowan\desktop\scratch.txt}
function opencatscratch{cat C:\users\paul.cowan\desktop\scratch.txt}

set-alias notepad "C:\Program Files (x86)\Notepad++\notepad++.exe"
set-alias zip "C:\Program Files\7-Zip\7z.exe"
set-alias grep select-string
set-alias ssh New-PSSecureRemoteSession
set-alias sh New-PSRemoteSession
set-alias grep select-string
```

```
set-alias ssh New-PSSecureRemoteSession
set-alias sh New-PSRemoteSession
set-alias tr gototrunk
set-alias pr gotoprojects
set-alias profile gotoprofile
set-alias build gotobuild
set-alias downloads gotodownloads
set-alias nc gottencontinuity2
set-alias lead gotolead
set-alias live gotocurrent
set-alias live_build gotocurrentbuild
set-alias scratch openscratch
set-alias catscratch opencatscratch
set-alias ie "C:\Program Files\Internet Explorer\iexplore.exe"
set-alias c2 gotoc2
set-alias c2web "C:\projects\continuity2\ncontinuity2.web.sln"
```

9.12 Urls

<http://technet.microsoft.com/en-us/library/dd347600.aspx>

<http://blogs.technet.com/b/heyscriptingguy/archive/2011/05/22/use-powershell-to-make-mandatory-parameters.aspx>

<http://www.simple-talk.com/dotnet/.net-tools/down-the-rabbit-hole-a-study-in-powershell-pipelines,-functions,-and-parameters/>

<http://cleancode.sourceforge.net/wwwdoc/articles.html>

<http://powershell.isdecisions.com/>

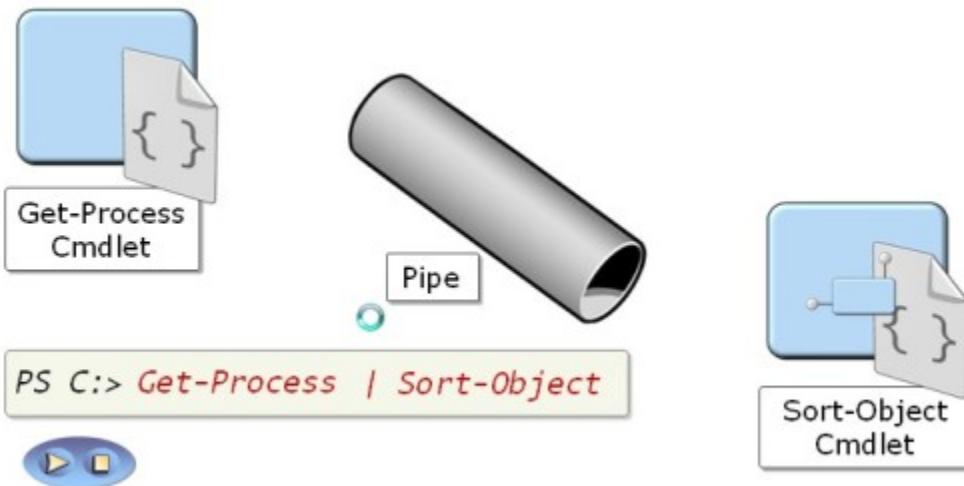
<http://blogs.technet.com/b/heyscriptingguy/archive/2012/05/21/understanding-the-six-powershell-profiles.aspx>

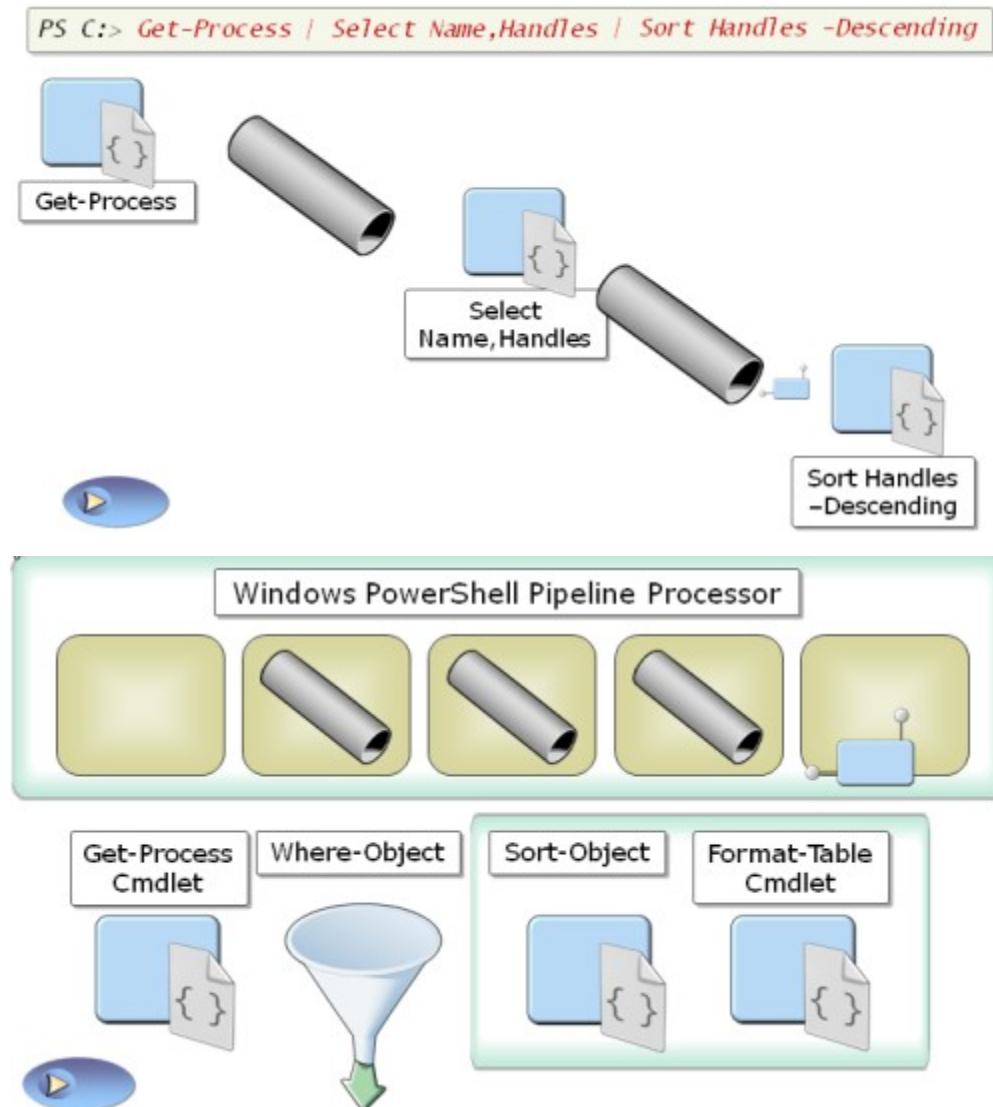
KAPITEL 10

Pipeline

Die Pipeline verbindet eine Vielzahl von Befehlen zusammen; das Ergebnis eines Befehls wird als Eingabewert für den nächsten Befehl verwendet, solange, bis das gewünschte Ergebnis erreicht ist.

test





Während es auch schon zu DOS-Zeiten das Konzept des Pipens gab und auch Unix-Shells heftig davon Gebrauch machen, gibt es doch einen gravierenden Unterschied zu diesen alten Implementierungen. Während in der alten Zeit Strings als Tauschformat benutzt wurden, verwendet die Powershell echte Objekte. Die Eigenschaften und Methoden der transportierten Objekte bleiben dabei erhalten.

```
Dir | Sort-Object Length | Select-Object Name, Length |
ConvertTo-HTML | Out-File report.htm
.\report.htm
```

Bis zuletzt bleiben die Ergebnisse Objekte, von denen relativ leicht bestimmte Eigenschaften (Name, Length) ausgegeben werden können

Zitat

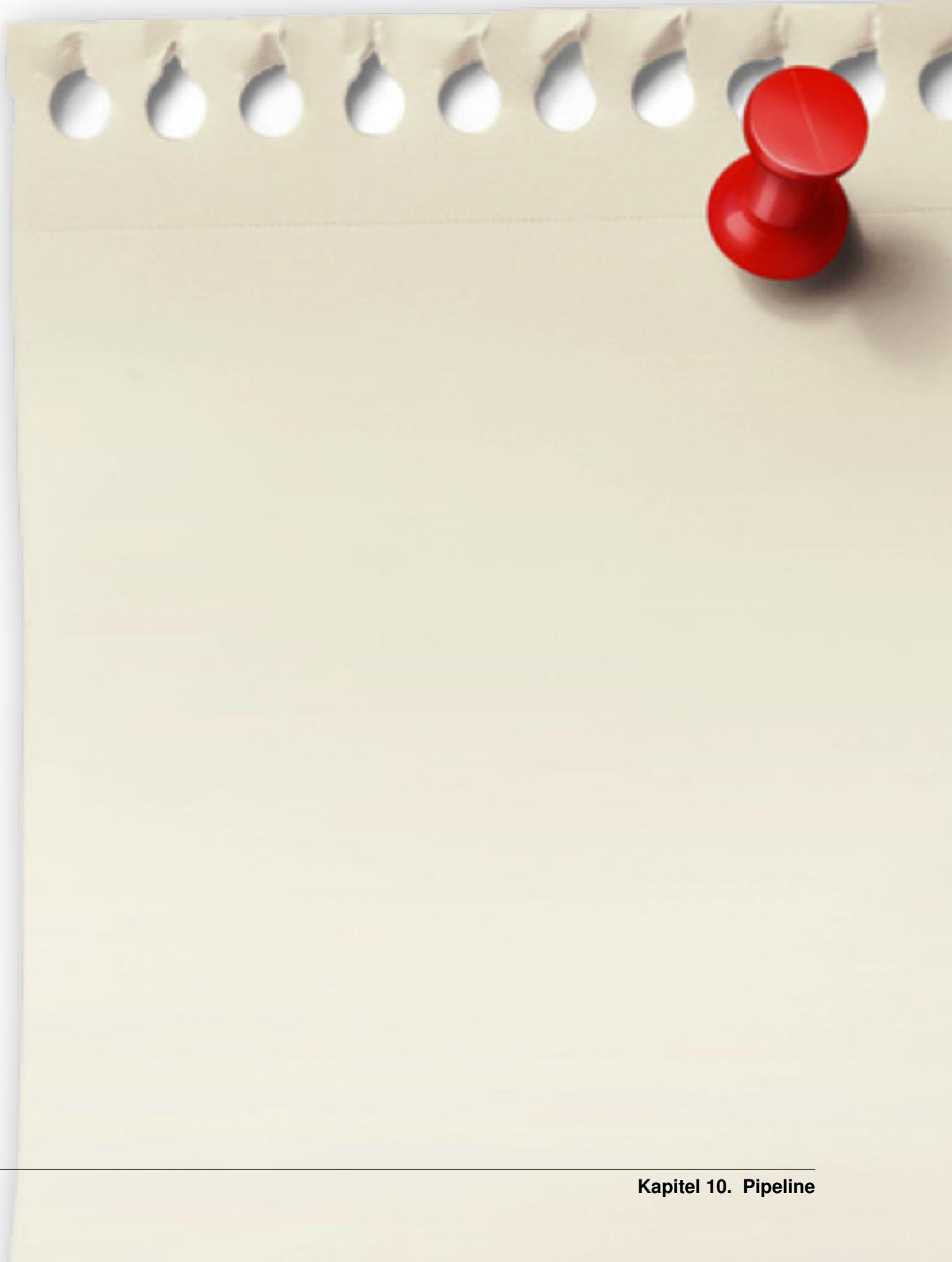
„What you see here is a true object-oriented pipeline so the results from a command remain rich objects. Only at the end of the pipeline will the results be reduced to text or HTML or whatever you choose for output. Take a look at Sort-Object. It sorts the directory listing by file size. If the pipeline had simply fed plain text into Sort-Object, you

would have had to tell Sort-Object just where the file size information was to be found in the raw text. You would also have had to tell Sort-Object to sort this information numerically and not alphabetically.

Not so here. All you need to do is tell Sort-Object which object property you want to sort. The object nature tells Sort-Object all it needs to know: where the information you want to sort is found, and whether it is numeric or letters.

You only have to tell Sort-Object which object property to use for sorting because PowerShell sends results as rich .NET objects through the pipeline. Sort-Object does all the rest automatically. Simply replace Length with another object property, such as Name or LastWriteTime, to sort according to these criteria. Unlike text, information in an object is clearly structured: this is a crucial PowerShell pipeline advantage.“

Aufgabe: Übersetzen Sie den englischen Text sinngemäß.



Das Konzept der Pipeline wird auf alle Ausgaben angewendet, auch wenn dies nicht immer explizit zu erkennen ist. Auf alle Ausgaben wird das Out-Default-Cmdlet hinzugefügt, welches die Ausgabe in Textform bewirkt

Ein DIR ist damit eigentlich ein DIR | Out-Default.

Folgende Cmdlets stehen zur Verfügung.

Cmdlet/Funktion	Description
<i>Compare-Object</i>	Compares two objects or object collections and marks their differences
<i>ConvertTo-Html</i>	Converts objects into HTML code
<i>Export-Clixml</i>	Saves objects to a file (serialization)
<i>Export-Csv</i>	Saves objects in a comma-separated values file
<i>ForEach-Object</i>	Returns each pipeline object one after the other
<i>Format-List</i>	Outputs results as a list
<i>Format-Table</i>	Outputs results as a table
<i>Format-Wide</i>	Outputs results in several columns
<i>Get-Unique</i>	Removes duplicates from a list of values
<i>Group-Object</i>	Groups results according to a criterion
<i>Import-Clixml</i>	Imports objects from a file and creates objects out of them (deserialization)

<i>Measure-Object</i>	Calculates the statistical frequency distribution of object values or texts
<i>more</i>	Returns text one page at a time
<i>Out-File</i>	Writes results to a file
<i>Out-Host</i>	Outputs results in the console
<i>Out-Host -paging</i>	Returns text one page at a time
<i>Out-Null</i>	Deletes results
<i>Out-Printer</i>	Sends results to printer
<i>Out-String</i>	Converts results into plain text
<i>Select-Object</i>	Filters properties of an object and limits number of results as requested
<i>Sort-Object</i>	Sorts results
<i>Tee-Object</i>	Copies the pipeline's contents and saves it to a file or a
	variable
<i>Where-Object</i>	Filters results according to a criterion

10.1 Streaming vs Blocking

Bei der Kombination per Pipeline kommt schnell die Frage auf, wann denn jetzt ein einzelner Befehl tatsächlich umgesetzt wird. Dies hängt vom jeweiligen Modus ab, in dem die Pipeline arbeitet.

Sequentieller Modus:

In diesem Modus werden die Befehle streng hintereinander ausgeführt, d.h. die Ergebnisse werden erst dann per Pipeline weitergereicht, wenn es komplett berechnet ist. Dieser Modus ist langsam und speicherintensiv, lässt sich aber nicht immer vermeiden, z.B. beim Sort-Object-Cmdlet. So sind im unteren Beispiel die ersten beiden Pipeline sequentiell, während das 3. Beispiel streaming-fähig ist.

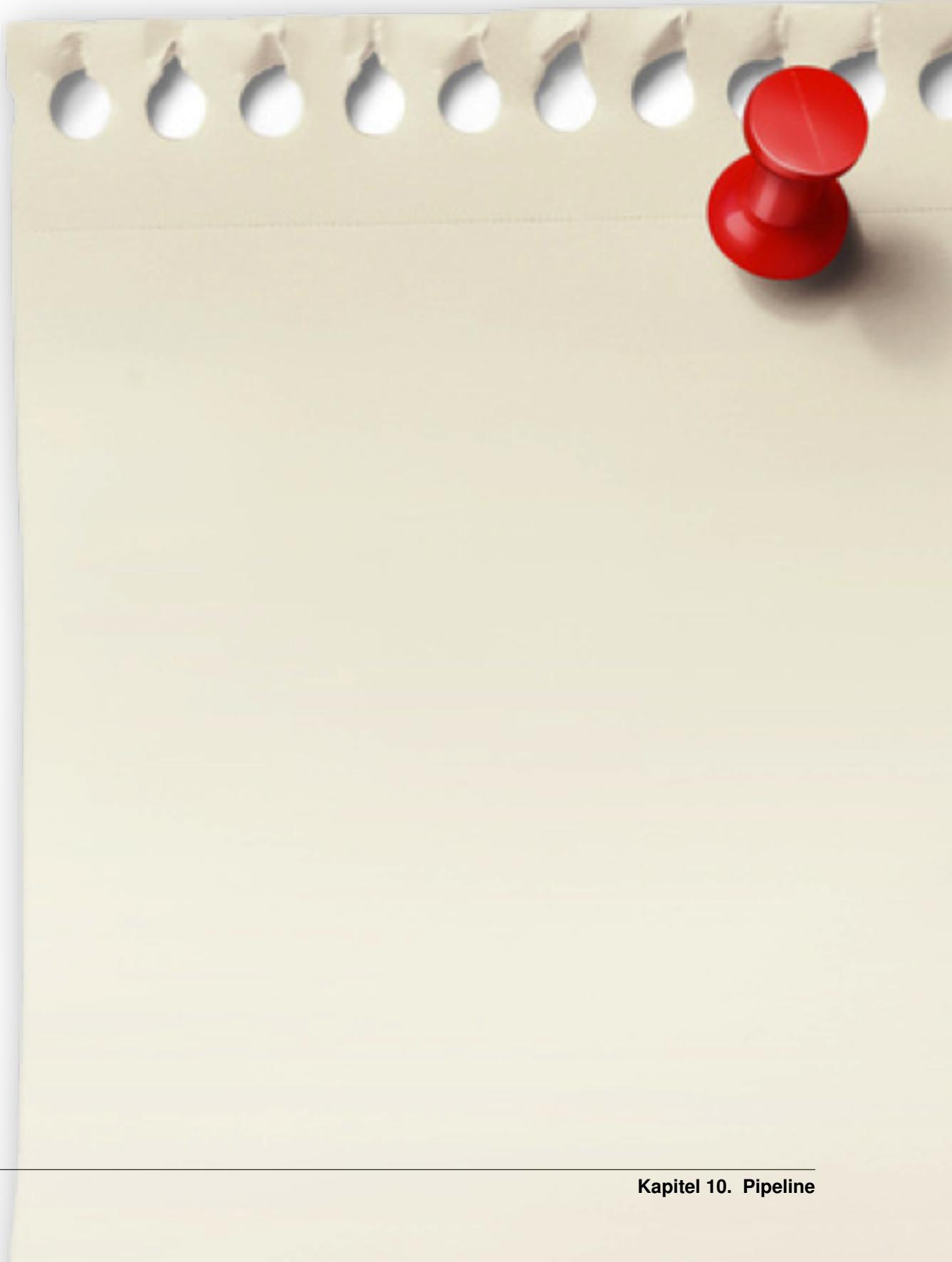
```
# Attention: danger!
Dir C:\ -recurse | Sort-Object

Dir C:\ -recurse | more

Dir c:\ -recurse | Out-Host -paging
```

Streaming Modus:

Hier wird jedes einzelne Ergebnis einer Berechnung sofort an den nächsten Befehl weitergereicht



10.2 Eigenschaften der Objekte

Um nicht nur die Standard-Eigenschaften der Objekte zu sehen, kann man sich mit Hilfe Format-Cmdlets einen Überblick über die Eigenschaften geben.

```
Dir | Format-Table *
Dir | Format-List *
```

Die Darstellung der Ausgabe kann mit Hilfe von 4 Format-Cmdlets erfolgen:

```
Get-Command -verb format
 CommandType
 -----
 Cmdlet
 Cmdlet
 Cmdlet
 Cmdlet
 Name
 -----
 Format-Custom
 Format-List
 Format-Table
 Format-Wide
 Definition
 -----
 Format-Custom [[-Property] <Object>...]
 Format-List [[-Property] <Object>...]
 Format-Table [[-Property] <Object>...]
 Format-Wide [[-Property] <Object>...]
```

10.3 Ausgabe

Die Ausgabe spezieller Eigenschaften erfolgt dann durch Hintereinanderstellen der jeweiligen Namen. Wildcards sind erlaubt.

```
PS C:\temp> Dir | Format-Table Name, Length
Name                                         Length
-----
FirefoxPortable
postkarten
Ausgabe.txt
autorun.inf
index.html
index_haas.html
ShellExec.exe
Zeungnisbemerkung_11FI2_2009
                                     2980
                                     35
                                     3682
                                     3498
                                     61440
                                     3123
```

```
Get-Process i* | Format-Table name, pe*64
```

Name	PeakPagedMemorySize64	PeakWorkingSet64
Idle	0	24576
inetinfo	6160384	11816960

Skriptblöcke und künstliche Eigenschaften sind ebenso möglich: Im unteren Beispiel ist `$_.Length` jeweils das aktuelle Objekt in der Pipeline.

Dir Format-Table Name, { [int] ((\$_.Length/1KB)) }	
PS C:\temp> Dir Format-Table Name, { [int] ((\$_.Length/1KB)) }	
Name	[int] ((\$_.Length/1KB))
-----	-----
FirefoxPortable	0
postkarten	0
Ausgabe.txt	3
autorun.inf	0
index.html	4
index_haas.html	3
ShelExec.exe	60
Zeungnisbemerkung_11FI2_2009	3

Oder man möchte wissen, wie alt die Dateien seit der letzten Veränderung sind. Dies liefert uns die Kombination der folgenden Befehle:

```
New-TimeSpan "01/01/2000" (Get-Date) //Zeitdifferenz zwischen heute und dem 1.1.2000

# gibt uns die Zeitdifferenz zwischen Heute und dem letzten Zugriff des jeweiligen
# Objektes
{ (New-TimeSpan $_.LastWriteTime (Get-Date)).Days }

# kombiniert alles zu einer schäf' Ätä€mÄfâ€ Äçâ, â„¢ÄfÈ'Äçâ, ÄÅ;Äfâ€šÃ, Ä¶nen Pipeline
Dir | Format-Table Name, Length, {(New-TimeSpan $_.LastWriteTime (Get-Date)).Days} -autosize
```

Name	Length	(New-TimeSpan \$_.LastWriteTime (Get-Date)).Days
FirefoxPortable		8
postkarten		31
Ausgabe.txt	2980	3
autorun.inf	35	63
index.html	3682	64
index_haas.html	3498	64
ShelExec.exe	61440	2197
Zeungnisbemerkung_11FI2_2009	3123	31

10.3.1 Ändern der Spaltenüberschriften

Die Spaltenüberschriften können geändert werden, wenn auch etwas umständlich

```
Dir | Format-Table Name, @ {Label="Größe"; Expression={[int]($_.Length/1KB)}} -AutoSize
```

```
Dir | Format-Table Name, Length, @{Label="Differenz"; Expression ={(New-TimeSpan $_.LastWriteTime (Get-Date)).Days}} -AutoSize
```

```
Dir | Format-Table Name, Length, @{Label="Länge"; Expression ={$_.Length}} -AutoSize
```

10.3.2 Optimieren der Spaltenbreiten

Wegen der Echtzeitausgabe kann Format-Table nicht wissen, wie breit eine Spalte zu wählen ist, um eine optimale Anzeige aller Werte zu erhalten. Man kann dies durch die Option **-auto** erhalten, verliert aber damit den Vorteil der Echtzeit.

```
column = @{Expression={ [int]($_.Length/1KB) }; Label="KB" }
Dir | Format-Table Name, $column -auto
Name
-----
output.htm
output.txt
backup.pfx
cmdlet.txt
KB
--
11
13
2
23
```

10.4 Sortieren und Gruppieren der Pipeline-Ergebnisse

Bemerkung: So that you can make good use of Sort-Object and all the other following cmdlets, you must also know which properties are available for the objects traveling through the pipeline. In the last section, you learned how to do that. Send the result of Dir to Format-List * first, then you'll see all properties and you can select one to use for subsequent sorting:

```
Dir | Format-List *
```

10.4.1 Sortieren

Das Sortieren erfolgt mit Hilfe des Sort-Object-Cmdlets. Falls nichts angegeben wird, sucht sich das Sort-Object-Cmdlet seine eigene Property zum Sortieren.

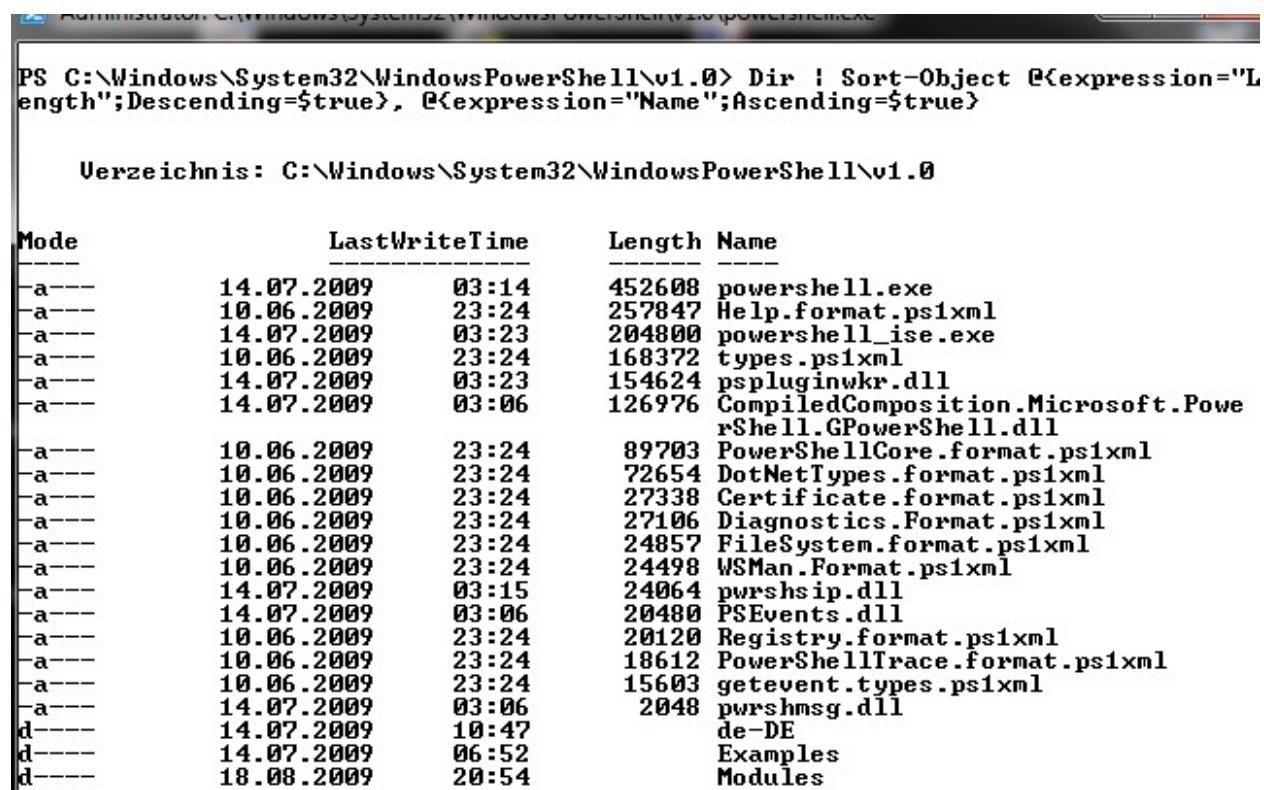
```
Dir | Sort-Object
```

Eigene Sortierkriterien überibt man mit der Property-Eigenschaft.

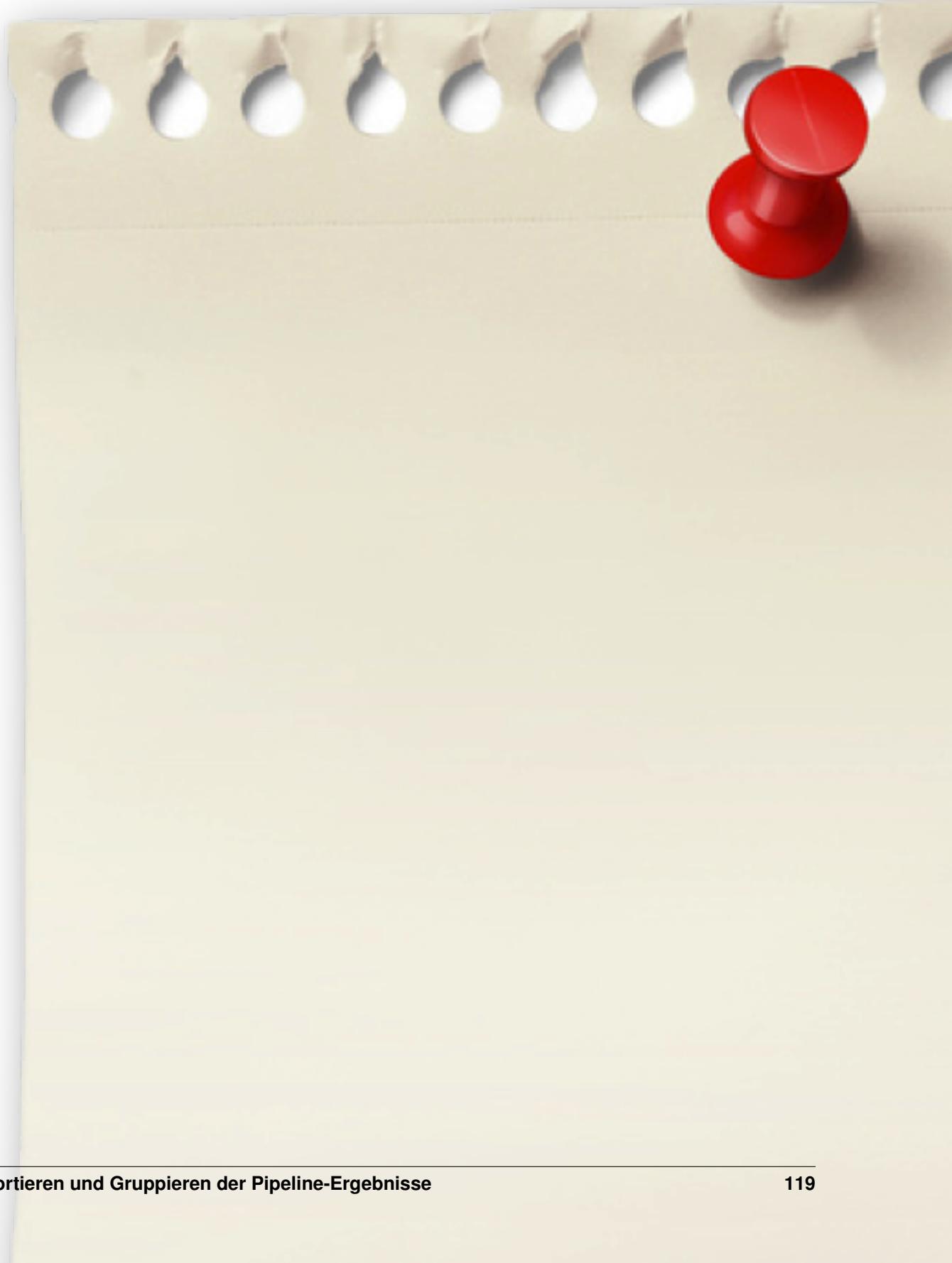
```
Dir | Sort-Object -property Length -descending
Dir | Sort-Object Extension, Name
Dir | Sort-Object Extension, Name -descending
#das geht nicht
```

```
Dir | Sort-Object Extension, Name -descending, -ascending
```

```
#wir müssen es an eine hash-table übergeben
Dir | Sort-Object @{expression="Length";Descending=$true}, @{expression="Name";
→Ascending=$true}
```



Mode	LastWriteTime	Length	Name
-a---	14.07.2009	03:14	452608 powershell.exe
-a---	10.06.2009	23:24	257847 Help.format.ps1xml
-a---	14.07.2009	03:23	204800 powershell_ise.exe
-a---	10.06.2009	23:24	168372 types.ps1xml
-a---	14.07.2009	03:23	154624 pspluginwkr.dll
-a---	14.07.2009	03:06	126976 CompiledComposition.Microsoft.PowerShell.GPowerShell.dll
-a---	10.06.2009	23:24	89703 PowerShellCore.format.ps1xml
-a---	10.06.2009	23:24	72654 DotNetTypes.format.ps1xml
-a---	10.06.2009	23:24	27338 Certificate.format.ps1xml
-a---	10.06.2009	23:24	27106 Diagnostics.Format.ps1xml
-a---	10.06.2009	23:24	24857 FileSystem.format.ps1xml
-a---	10.06.2009	23:24	24498 WSMan.Format.ps1xml
-a---	14.07.2009	03:15	24064 pwrshsip.dll
-a---	14.07.2009	03:06	20480 PSEvents.dll
-a---	10.06.2009	23:24	20120 Registry.format.ps1xml
-a---	10.06.2009	23:24	18612 PowerShellTrace.format.ps1xml
-a---	10.06.2009	23:24	15603 getevent.types.ps1xml
-a---	14.07.2009	03:06	2048 pwrshmsg.dll
d----	14.07.2009	10:47	de-DE
d----	14.07.2009	06:52	Examples
d----	18.08.2009	20:54	Modules



10.4.2 Gruppieren

Mit Hilfe des Group-Object_Cmdlets können Sie Objekte mit gleichen Eigenschaften nach Ihrer Anzahl gruppieren.

Das folgende Beispiel gibt die Anzahl der laufenden und gestoppten Dienste zurück.

```
Get-Service | Group-Object Status
```

```
PS C:\> Get-Service | Group-Object Status
```

Count	Name	Group
98	Stopped	{System.ServiceProcess.ServiceController, Sy...}
75	Running	{System.ServiceProcess.ServiceController, Sy...}

```
PS C:\>
```

```
$result = Get-Service | Group-Object Status
$result[0].Group

Dir | Group-Object Extension

Dir | Group-Object Extension | Sort-Object Count -descending

Count Name Group
----- ----
26 {data, docbook, Documents, Dokumente und Ein...
18 .log {start.log, 20100318180017.log, Ve...
9 .txt {ctapi_out_gr.txt, Ergebnis3.txt, ergebnisne...
5 .pdf ↪{Abschlussprüfung Winter 2002 - 2003.pdf, ...
an...
5 .jpg {last.jpg, presse_1.jpg, presse_2.jpg, press...
2 .out {err.out, start.out}
2 .zip {dvd_postkarten_16052010.zip, Stoffverteilun...
2 .tex {start.tex, syntax.tex}
2 .GDB {TEST10.GDB, TEST9.GDB}
1 .hnd {sf.hnd}
1 .ods {schueler.ods}
```

Group-Object kann nicht nur nach Eigenschaften gruppieren, sondern auch nach eigenen Ausdrücken. Das folgende Beispiel gruppiert alle Dateien, in Abhängigkeit, ob sie größer als 100 kByte oder kleiner sind.

```
PS C:\> Dir | Group-Object {$_.Length -gt 100KB}
```

Count	Name	Group
78	False	{cruisecontrol-bin-2.8.3, data, docbook, Doc...}
23	True	{Abschlussprüfung Winter 2002 - 2003.pdf, dv...}

Das folgende Beispiel gruppiert nach der Anzahl des Vorkommens des jeweils 1. Buchstabens.

```
PS C:\> Dir | Group-Object {$_.name.Substring(0,1).ToUpper()}
```

Count	Name	Group
3	C	{cruisecontrol-bin-2.8.3, config.sys, ctapi....}
10	D	{data, docbook, Documents, Dokumente und Ein...}

```

3      I          {inetpub, Intel, ihm_pruefung.pdf}
4      M          {Miranda IM, music, mymiktex, myPass.kdbx}
10     P          {PerfLogs, Program Files, ProgramData, Progr...
1      R          {RavenDB-Build-101}
13     S          {Sounds, steinam, schueler.ods, sf.hnd...}
8      T          {TEMP, totalcmd, test.ps1, TEST10.GDB...}
1      U          {Users}
3      W          {Windows, WiresharkPortable, wepkeys.txt}
2      .          {.emacs, .emacs~}
5      A          {Abschlussprüfung Winter 2002 - 2003.pdf, Ak...
3      E          {Ergebnis3.txt, ergebnisnew.txt, err.out}
2      H          {haas.txt, haas_lanig.txt}
1      K          {KOS_09_10.dav}
4      L          {last.jpg, LA_ansichten_booklet (2).pdf, LA_...

```

Wie man sieht, ist jede Zeile ein eigener Array mit allen Objekten der jeweiligen Gruppierung. Man könnte dies nutzen, um eine alphabetische Liste aller Objekte auszugeben.

```

Dir | Group-Object {$_.name.Substring(0,1).ToUpper()} | ForEach-Object { ($_
->Name)*7; "====="; $_.Group}

CCCCCCC
=====

Verzeichnis: C:\

Mode                LastWriteTime       Length Name
----              -----          -----
d----        06.06.2010    21:22            cruisecontrol-bin-2.8.3
-a---        10.06.2009    23:42           10 config.sys
-a---        01.04.2010    22:23           0 ctapi_out_gr.txt

DDDDDDD
=====
d----        20.07.2010    23:00            data
d----        21.08.2009    22:24            docbook
d----        25.07.2010    20:28            Documents
d----        31.07.2009    22:40            Dokumente und Einstellungen
d----        24.02.2010    08:58            download
d----        20.05.2010    18:09            dvd_postkarten_16052010
-a---        23.11.2009    21:27            264 dfsd.nsd
-a---        18.12.2009    21:20            7737 dienste.htm
-a---        16.05.2010    20:30            1665 dsd.aup
-a---        10.06.2010    08:30            289840446 dvd_postkarten_16052010.zip

IIIIIII
=====
d----        18.08.2009    20:54            inetpub
d----        18.08.2009    09:27            Intel
-a---        02.08.2009    21:35            165025 ihm_pruefung.pdf

MMMMMM

```

Wenn man die gruppierten Objekte selbst nicht benötigt, kann man mit Hilfe des **-noelement**-Parameters Speicher sparen.

```

Get-Process | Group-Object -property Company -noelement

Count Name
----- 
2 Apple Inc.

```

```
4
5
1 AVerMedia TECHNOLOGIES...
1 AVerMedia
3 Avira GmbH
32 Microsoft Corporation
2 Firebird Project
1 Google
1 Google Inc.
4 Intel Corporation
1 Irfan Skiljan
2 Sun Microsystems, Inc.
1 Tracker Software Produ...
1 C. Ghisler & Co.
1 http://tortoisessvn.net
```

10.5 Filtern

Mit Hilfe des Where-Objectes können die Objekte der Pipeline gefiltert werden.

```
PS C:\> Get-Service | Where-Object { $_.Status -eq "Running" } | more

Status      Name          DisplayName
----      ----          -----
Running    AntiVirSchedule...  Avira AntiVir Planer
Running    AntiVirService     Avira AntiVir Guard
Running    AppHostSvc        Anwendungshost-Hilfsdienst
Running    Apple Mobile De...  Apple Mobile Device
Running    AudioEndpointBu...  Windows-Audio-Endpunktterstellung
Running    Audiosrv          Windows-Audio
Running    AVerRemote         AVerRemote
Running    BFE                Basisfiltermodul
Running    BITS               Intelligenter Hintergrundübertragungsdienst
Running    Bonjour Service   Dienst "Bonjour"
Running    Browser            Computerbrowser
Running    CertPropSvc       Zertifikatverteilung
```

Das Cmdlet erwartet, dass man in geschweiften Klammern einen Powershell-Befehl übergibt. Das jeweils zu untersuchende Objekt kann über die Variable `$_.` angesprochen werden. Die Eigenschaft Status wird dann auf den Wert Running verglichen. Nur wenn dieses Objekt den entsprechenden Wert hat, wird es in die Pipeline gelassen.

Damit entspricht das Where-Objekt im Grunde der Formulierung einer Bedingung, die TRUE ergeben muss.

```
Get-WmiObject Win32_Service | ? { ($_.Started -eq $false) -and ($_.StartMode -eq "Auto
↪") } | Format-Table
```

```
PS C:\> Get-WmiObject Win32_Service | 
>> ? {($_.Started -eq $false) -and ($_.StartMode -eq "Auto") } | 
>> Format-Table

ExitCode Name ProcessId StartMode
----- ---- -----
1067 AVerScheduleService 0 Auto
0 clr_optimization... 0 Auto
0 gupdate 0 Auto
0 MMCSS 0 Auto
0 sppsvc 0 Auto
```

Das gleiche Ergebnis hätte man auch ohne Where-Objekt erhalten können, indem man WMI mit den korrekten Fragen füttert.

```
Get-WmiObject -query "select * from win32_Service where Started=false and StartMode=
→'Auto'" | Format-Table
```

```
PS C:\> Get-WmiObject -query "select * from win32_Service where ' 
>> Started=false and StartMode='Auto'" | Format-Table
>>

ExitCode Name ProcessId StartMode
----- ---- -----
1067 AVerScheduleService 0 Auto
0 clr_optimization... 0 Auto
0 gupdate 0 Auto
0 MMCSS 0 Auto
0 sppsvc 0 Auto
```

10.5.1 Begrenzen der Anzahl der Ausgabe

Select-Object kann neben der Ausgabe bestimmter Eigenschaften auch die Anzahl der ausgegebenen Datensätze begrenzen

```
# List the five largest files in a directory:
Dir | Sort-Object Length -descending | Select-Object -first 5

# List the five longest-running processes:
Get-Process | Sort-Object StartTime | Select-Object -last 5 | Format-Table_
→ProcessName, StartTime

# Alias shortcuts make the line shorter but also harder to read:
gps | sort StartTime -ea SilentlyContinue | select -last 5 | ft ProcessName, StartTime
```

10.6 Foreach-Objekt

Anstelle auf die Ergebnisse einer Pipeline zu warten, kann mit Hilfe des Foreach-Object-Cmdlets sofort auf jedes einzelne Objekt zugegriffen werden. Foreach-Object erwartet in geschweiften Klammern eine Anweisung, was es mit dem jeweiligen Objekt anstellen soll.

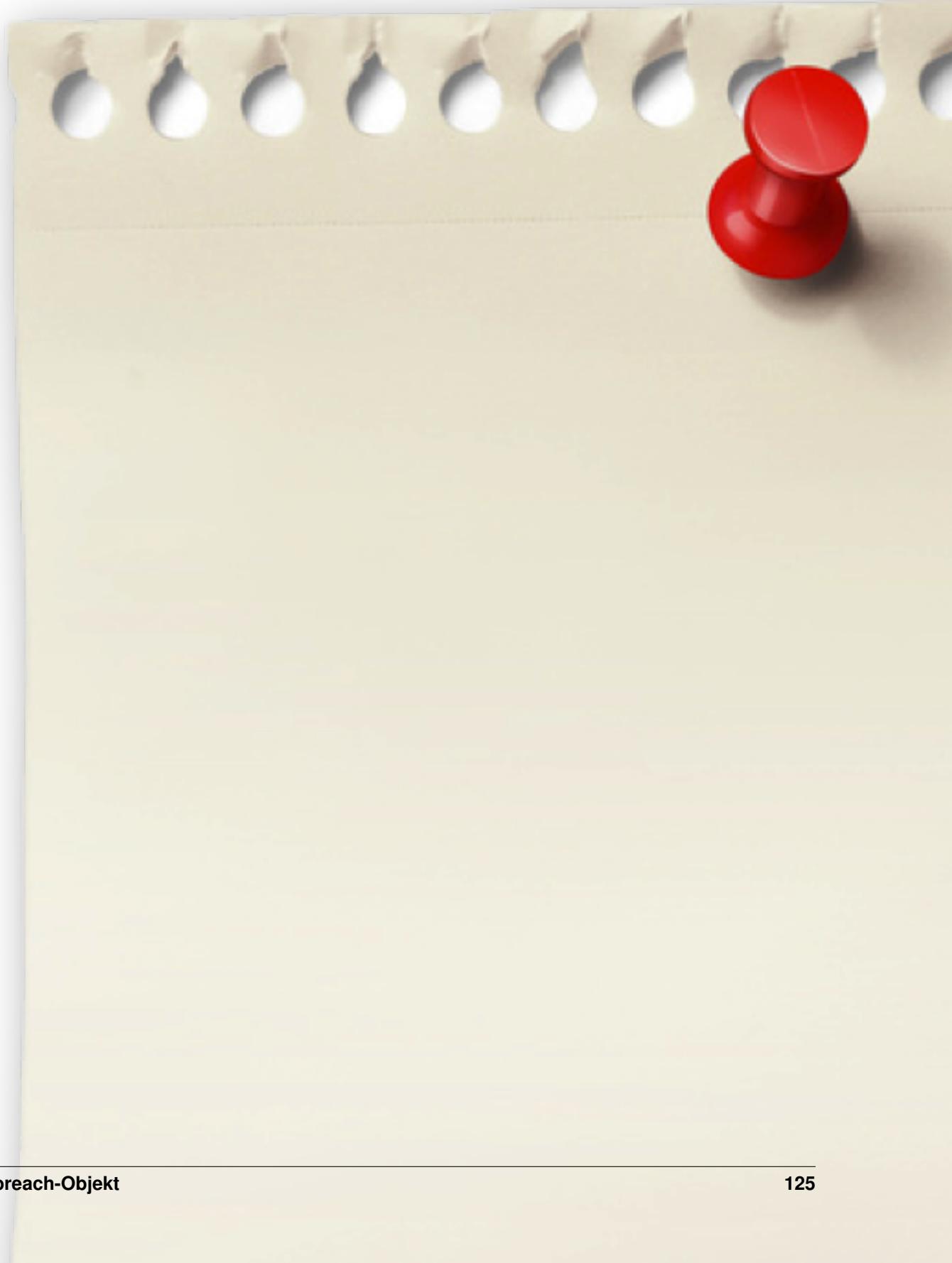
Foreach-Object kann wie das Where-Object als Filter verwendet werden. Dazu muss nur in der geschweiften Klammer eine Bedingung formuliert werden. Das Where-Object hat man eigentlich nur aus Begrenlichkeit geschaffen. Die folgenden Befehle haben alle das gleiche Ergebnis.

```
Get-Service | Where-Object { $_.Status -eq "Running" }  
Get-Service | ? { $_.Status -eq "Running" }  
Get-Service | ForEach-Object { if ($_.Status -eq "Running") { $_ } }  
Get-Service | % { if ($_.Status -eq "Running") { $_ } }
```

Bemerkung: ForEach-Object actually executes three script blocks, not just one. If you specify only one script block in braces after ForEach-Object, it will be executed once for every pipeline object. If you specify two script blocks, the first will be executed once and before the first pipeline object. If you specify three script blocks, the last will be executed once after the last pipeline object. The following will help you carry out initialization and tidying tasks or simply output initial and ending messages:

```
Get-Service | ForEach-Object {"Running services:"}{ if ($_.Status -eq "Running") { $_  
→ } }{"Done."}
```

The three script blocks of ForEach-Object actually correspond to the three script blocks begin, process, and end, which you'll examine in more detail later



10.7 Zwischenergebnisse

Während einer Pipeline-Operation können mit Hilfe des Tee-Objektes die Zustände von Pipeline-Operationen in Variablen zwischengespeichert werden.

```
Get-Process | Tee-Object -variable a1 | Select-Object Name, Description |  
Tee-Object -variable a2 | Sort-Object Name  
  
$a1  
$a2
```

10.8 Ausgabe von Pipeline-Ergebnissen

10.9 Funktionen und Pipeline

10.10 Pipeline oder Schleife

Häufig ist ein Problem sowohl mit Hilfe der Pipeline als auch mit Hilfe der „normalen“ strukturierten Programmierung möglich. Je nach Szenario kann es allerdings zu erheblichen Geschwindigkeitsunterschieden kommen. Folgendes Beispiel verdeutlicht den Zusammenhang.

```
#eine strukturierte Herangehensweise an das Problem  
#Schleife mit if  
#3 Minuten, 18 sec  
#$zeit = Get-Date  
##$result = Get-Content $env:windir\Windowsupdate.log  
#foreach($zeile in $result)  
#{  
#    if($zeile.Contains("WARNING"))  
#    {  
#        $line = $zeile.Split("`t")  
#        Write-Host ($line[0] + " " + $line[1])  
#        #Add-Content -Path "C:\ergebnisnew.txt" -value $zeile;  
#    }  
#}  
#$diff = $zeit - (Get-Date)  
  
#Write-Host $diff  
  
  
#Das gleiche nun mit Pipes  
#dauert 5 Min, 48 sec !!!  
#$zeit = Get-Date  
  
#Get-Content $Env:windir\WindowsUpdate.log | Select-String "WARNING" | Add-Content  
#→ "C:\Ergebnis3.txt"  
#$diff = $zeit - (Get-Date)  
  
  
#auch mit Pipes kann es schneller gehen  
  
#Write-Host $diff  
$zeit = Get-Date
```

```
Get-Content $Env:windir\WindowsUpdate.log -readCount 1000 | Select-String "WARNING" | 
    Add-Content "C:\Ergebnis3.txt"
$diff = $zeit - (Get-Date)
Write-Host $diff`
```

10.11 Pipeline/Filter und Funktionen

Funktionen können die Ergebnisse der Pipeline weiterverarbeiten. Es stellt sich lediglich die Frage, ob dies im langsamem *sequentiellen* oder schnelleren *streaming*-Modus vonstatten geht.

10.11.1 Sequentieller Modus

Im einfachsten Falle unterstützt eine Funktion nicht wirklich die Pipeline. Die Ergebnisse der Pipeline stehen in der automatischen Variable **\$input** zur Verfügung. Im einfachsten Falle gibt die Funktion einfach die Inhalte von **\$input** aus.

```
Function output
{
    $input
}
# The function, when invoked alone,
# will return nothing because no pipeline
# results are available:
output
# If you create an array in the pipeline,
# the function will output the array:
1,2,3 | output
1
2
3

# The function is completely indifferent to
# which type of data is in the pipeline:
Dir | output
```

Die Funktion soll nun aber abgeändert werden; alle Dateien mit der Endung .exe sollen in roter Farbe ausgegeben werden.

```
function MarkEXE
{
    # Note old foreground color
    $oldcolor = $host.ui.rawui.ForegroundColor
    # Inspect each pipeline element separately in a loop
    Foreach ($element in $input) {
        # If the name ends in ".exe", change the foreground color to red:
        If ($element.name.toLower().endsWith(".exe")) {
            $host.ui.Rawui.ForegroundColor = "red"
        }
        Else {
            # Otherwise, use the normal foreground color:
            $host.ui.Rawui.ForegroundColor = $oldcolor
        }
        # Output element
        $element
    }
}
```

```

}
# Finally, restore the old foreground color:
$host.ui.Rawui.ForegroundColor = $oldcolor
}

Measure-Command {Dir "c:\program files" -recurse | MarkEXE }

PS C:\Windows\system32> Measure-Command {Dir "c:\program files" ` -recurse | MarkEXE }
`->

Days          : 0
Hours         : 0
Minutes       : 0
Seconds       : 0
Milliseconds  : 706
..

```

Die Ausführung des Skripts dauert 706 msec.

10.11.2 Filter

Ersetzt man aber das Schlüsselwort **Function** durch das Wort **Filter**, dann ist die Ausgabe ca. um den Faktor 3 schneller.

```

PS C:\Windows\system32> Measure-Command {Dir "c:\program files" ` -recurse |_
`->MarkEXEFilter }

Days          : 0
Hours         : 0
Minutes       : 0
Seconds       : 0
Milliseconds  : 189
..

```

Da die \$input-Variable für Filter immer nur aus einem Wert besteht, macht ihr Einsatz innerhalb der Schleife wenig Sinn. Man kann Sie besser durch die Variable \$_ ersetzen.

```

Filter MarkEXEFilter2 {
    # Note old foreground color
    $oldcolor = $host.ui.rawui.ForegroundColor
    # The current pipeline element is in $_
    # If the name ends in ".exe", change
    # the foreground color to red:
    If ($_.name.ToLower().endsWith(".exe"))
    {
        $host.ui.Rawui.ForegroundColor = "red"
    }
    Else
    {
        # Otherwise, use the normal foreground color:
        $host.ui.Rawui.ForegroundColor = $oldcolor
    }
    # Output element
    $_
    # Finally, restore the old foreground color:
    $host.ui.Rawui.ForegroundColor = $oldcolor
}

```

```
}
```

```
PS C:\Windows\system32> Measure-Command {Dir "c:\program files" ` -recurse | 
→MarkEXEFilter2 } `

Days          : 0
Hours         : 0
Minutes       : 0
Seconds       : 1
Milliseconds  : 77
..
..
```

10.11.3 Generische Filter

Filter sind beim Einsatz von Pipelines den Funktionen vorzuziehen, weil Sie die Ergebnisse der Pipeline sofort verarbeiten können. Nachteilig kann sich aber auswirken, dass alle Codeblöcke eines Filters immer wieder aufgerufen werden.

In unserem Beispiel wird immer die alte Vordergrundfarbe gespeichert und nach dem Durchlauf wieder zurückgesetzt. Dies ist zeit- und ressourcenintensiv.

Das Verhalten einer Funktion kann aber an diese Situation angepasst werden, indem man innerhalb einer Funktion spezielle Bereiche definiert.

- begin: (Initialisierung: Wird einmal beim Beginn aufgerufen)
- process:
- end: Aufräumen; wird einmal am Ende aufgerufen

Ein Filter ist für die Powershell nichts anderes als eine Funktion mit einem einzigen **process**-Block.

Die optimale Codierung des obigen Problems lagert deswegen das Speichern und Zurücksetzen der Farbe in den begin- bzw. end-Block aus

```
Function MarkEXE {
    begin {
        # Note old foreground color
        $oldcolor = $host.ui.rawui.ForegroundColor
    }

    process {
        If ($_.name.toLowerCase().endsWith(".exe")) {
            $host.ui.Rawui.ForegroundColor = "red"
        }
        Else
        {
            $host.ui.Rawui.ForegroundColor = $oldcolor
        }
        # Output element
        $_
    }

    end {
        # Finally, restore the old foreground color:
        $host.ui.Rawui.ForegroundColor = $oldcolor
    }
}
```

```
}
```

Bemerkung: Das nächste Beispiel zeigt, dass ein Filter in Wirklichkeit eine normale Funktion mit einem process-Block ist.

```
filter Test { "Output: " + $_ }

Let's look now at the definition of the filter:
$function:Test
process {
"Output: " + $_
}
```

PowerShell hat die Filter-Anweisung in eine normale Funktion umgewandelt und den Code in einen process-Block gesetzt.

10.12 Zusammenfassung

PowerShell uses a pipeline for all command entries, which feeds the results of the preceding command directly into the subsequent command. The pipeline is active even when you enter only a single command because PowerShell always automatically adds the Out-Default cmdlet at the pipeline's end so that it always results in a two-member instruction chain.

Single command results are passed as objects. The cmdlets shown in Table 5.1 can filter, sort, compare, measure, expand, and restrict pipeline elements. All cmdlets accomplish this on the basis of object properties. In the process, the pipeline distinguishes between sequential and streaming modes. In streaming mode, command results are each collected, and then passed in mass onto the next command. Which mode you use depends solely on the pipeline commands used.

Output cmdlets dispose of output. If you specify none, PowerShell automatically uses Out-Host to output the results in the console. However, you could just as well send results to a file or printer. All output cmdlets convert objects into readable text while formatting cmdlets are responsible for conversion.

In addition to traditional output cmdlets, export cmdlets store objects either as comma-separated lists that can be opened in Excel or serialized in an XML format. Serialized objects can be comfortably converted back into objects at a later time. Because when exporting, in contrast to outputting, only plain object properties, without cosmetic formatting, are stored so that no formatting cmdlets are used.

10.13 Links

<http://thepowershellguy.com/blogs/posh/archive/2007/01/21/powershell-gui-scripblock-monitor-script.aspx>

Bemerkung: Als Beispiel für begin-process-end sowie pipeline kann man im nächsten Jahr eine Datenbankverbindung nehmen. Die Ergebnisse des dir-Befehls sollen in eine Datenbank geschrieben werden.

begin — öffnet die Verbindung process – führt die insert-Statements aus end – schließt die Datenbankverbindung nach dem letzten insert

a la : Dir -recurse | insertFileInDatabase

KAPITEL 11

Objekte in der PS

Zusammenfassung

Release 1.0

Datum 26.11.2017

Autor Steinam

Target Schüler FI SYS

status In Bearbeitung

vollständig 20 %

Downloads:

- AB Objekterzeugung.

<http://www.petri.co.il/custom-objects-windows-powershell-part-1.htm>

<https://www.powershellscripting.com/creating-custom-objects-windows-powershell-part-2.htm>

<https://www.powershellscripting.com/creating-custom-objects-in-windows-powershell-part-3.htm>

[creating-custom-objects-windows-powershell-part-4.htm](#)

463/powershell-eigene-objekte-erstellen-custom-objects

[powershell-v3-creating-objects-with-pscustomobject-its-usage](#)

07/24/using-add-member-in-a-powershell-one-liner-to-c

卷之三

<https://powertoe.wordpress.com/2014/04/26/you-know-p/>

[REDACTED]

<http://www.petri>.

<http://www.petri.co>

<http://www.petri.co.il/>

<http://www.admin-source.de/BlogDeu/>

<http://www.jonathanmedd.net/2011/09/>

<http://blogs.technet.com/b/gary/archive/2009/>

[properties.aspx?Redirected=true](#)

↳ [View on GitHub](#)

object-oriented-language-right/

11.1 Notwendigkeit von Objekten

Die bisherige Arbeit mit der PS legte den Wert auf Elemente der strukturierten Programmierung, z.B. Bedingungen, Schleifen, Funktionen; Werte wurden häufig nur in Form von Variablen bzw. Arrays übergeben.

Die Beschäftigung mit der Pipeline führte den Ansatz weiter fort, indem man nun von **Objekten** spricht, die man nun innerhalb der Pipeline weitergibt.

Objekte setzen die Vielzahl von Informationen zu einem einheitlichen Ganzen zusammen; sie verpacken die bisher in einzelnen Variablen gehaltenen Informationen zu einem sinnvollen Gesamtbild.

Goldene Regel der PowerShell

Eine Funktion oder ein Cmdlet sollte immer ein Objekt zurückgeben!

Wenn man kein fertiges Objekt hat und Daten aus verschiedenen Quellen zusammen sammelt, dann muss man sich eben ein eigenes Objekt zusammen bauen!

Das Erzeugen eines eigenen Objektes kann aus vielerlei Weise erfolgen.

11.2 Erzeugen von Objekten

Benutzen eines Hashes

Am einfachsten wäre es natürlich die Daten in einer Hashtable zu sammeln. Nur leider mögen einige Cmdlets keine Hashtables direkt verarbeiten!

Es ist aber leicht aus einer Hashtable ein Objekt zu erstellen.

```
# Hashtable mit Inhalt erstellen
$Hash= @{ Wert1 = 1;Wert2 = 2;Wert3 = 3}
$obj=New-Object -TypeName PSObject -Property $Hash

# Oder So: Ohne Umweg über die Variable der Hashtable:
$Hash=New-Object -TypeName PSObject -Property @{ Wert1=1;Wert2=2;Wert3=3}
```

PS 2.0: New-Object mit Add-Member

Dies ist der in PowerShell Version 2.0 vorgesehene Weg um PCCustomObjects zu erstellen.

Ein leeres Objekt erstellen und dann die NoteProperties nachträglich mit dem Cmdlet Add-Member anfügen:

```
# Neues leeres Objekt erstellen
$obj= New-object -TypeName PSObject

# Wert an das Objekt anfügen
Add-Member -InputObject $obj -Name Wert1 -Value 1 -MemberType NoteProperty
# Wert an das Objekt anfügen
Add-Member -InputObject $obj -Name Wert2 -Value 2 -MemberType NoteProperty
# Wert an das Objekt anfügen
Add-Member -InputObject $obj -Name Wert3 -Value 3 -MemberType NoteProperty
# CSV Export der Daten
Export-Csv -InputObject $obj -Path "C:\Temp\test.csv" -NoTypeInformation
```

Select-Object

```
$strObj="" | Select-Object Wert1,Wert2,Wert3
# Werte an die NoteProperties zuweisen

$strObj.Wert1 = "Hello"
$strObj.Wert2 = 2
```

```
$strObj.Wert3 = 3

$strObj.GetType()
$strObj | Get-Member -Name Wert2
```

New-Module –asCustomObject

Da alles in PowerShell ein Objekt ist, ist auch ein Modul “nur” ein Objekt mit Methoden (Function) und Feldern (Property).

Deshalb kann man mit dem Cmdlet New-Module sowie dem Parameter –asCustomObject und einem Scriptblock ein Objekt erstellen.

Der Vorteil hierbei ist, dass es viel leichter ist diesem Objekt neuen Methoden (Functions) oder neue Properties anzufügen.

Die bisher gezeigten NoteProperties können jeden Typen annehmen. Die Properties des Objektes, das über New-Modul erstellt wird, können stark Typisiert werden. So dass die Zuweisung eines falschen Wertes in einer Fehlermeldung enden.

```
$objekt = New-Module -AsCustomObject -ScriptBlock {
    [int]$Wert1=$null; # Objekt Property als Integer definieren
    Export-ModuleMember -Variable * # Variablen Public machen
}

$obj2 = New-Module -AsCustomObject -ScriptBlock {

    [int]$Wert1 = $null; # Objekt Property als Integer definieren
    [int]$Wert2 = $null; # Objekt Property als Integer definieren

    # Methode definieren
    Function Add {
        $Wert1+$Wert2 # Zahlen Addieren
    }

    Export-ModuleMember -Variable * -Function * # Variablen und Funktionen
    ↪Public machen
}

# Werte füllen
$obj2.Wert1 = 4
$obj2.Wert2 = 4
$obj2.Add() # Funktion ausführen
```

.NET-Code

```
# Eine Klasse mit C# Code und Add-Type definieren

add-type @"
using System;
public class myClass{
    public Double number=0;

    public Double Sqr()
    {
        return Math.Pow(number, 2.0);
    }
}@
"
```

```
# Das vorher definierte Objekt erstellen
$obj = New-Object myClass
# Der Property des Objektes einen Wert zuweisen
$obj.number = 5
# Methode des Objektes ausführen
$obj.Sqrt()
```

11.3 Einsatzmöglichkeiten

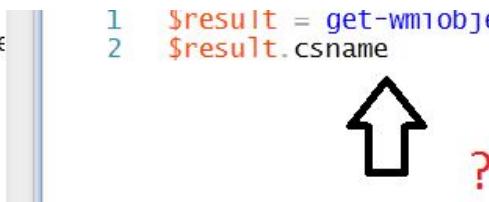
Objekte sind immer dann sinnvoll, wenn

- man die gegebenen Eigenschaften umbenennen/erweitern will
- wenn man mehrere Informationen zusammenfassen und weiterverwenden will

11.3.1 Erzeugen eines CustomObjektes aus einem vorhandenen Objekt

Im folgenden Beispiel will man die vorhandenen Eigenschaften eines zurückgegebenen Objektes in einer anderen Form darstellen.

```
PS C:\Users\admin> $re
$result.csname
STEINAM-LAPTOP
PS C:\Users\admin>
```



Die Eigenschaft **csname** ist wenig aussagekräftig. Diese und weitere Eigenschaften können durch folgendes Skript geändert werden.

```
get-wmiobject win32_operatingsystem | `>
Select @{Name="Computername";Expression={$_.CSName}} , `>
Version,@{Name="OS";Expression={$_.Caption}} , `>
@{Name="LastBoot";Expression={$_.Convert.ToDateTime($_.LastBootUpTime)}}`>
```

Der Code in einem Skriptblock kann beliebig sein. So kann die z.B. die LastBootUpTime auch schöner formatiert werden.

```
PS C:\Users\admin>
get-wmiobject win32_operatingsystem | select LastBootUpTime

LastBootUpTime
-----
20130408141608.125599+120
```

```
PS C:\Users\admin>
```

Die Rückgabe des Select-Commandlets ist ein echtes Objekt. Dies erkennt man an dem Nachschalten des Get-Member-CmdLets

```
get-wmiobject win32_operatingsystem | ` 
Select @{Name="Computername";Expression={$_.CSName}}, ` 
Version,@{Name="OS";Expression={$_.Caption}}, ` 
@{Name="LastBoot";Expression={$_.Convert.ToDateTime($_.LastBootUpTime)}} | get-member

` 

TypeName: Selected.System.Management.ManagementObject

Name      MemberType   Definition
----      -----      -----
Equals    Method      bool Equals(System.Object obj)
GetHashCode Method     int GetHashCode()
GetType   Method      type GetType()
ToString  Method      string ToString()
Computername NoteProperty System.String Computername=STEINAM-LAPTOP
LastBoot   NoteProperty System.DateTime LastBoot=08.04.2013 14:16:08
OS        NoteProperty System.String OS=Microsoft Windows 7 Enterprise
Version   NoteProperty System.String Version=6.1.7601
```

Mit diesem Objekt können nun weitere Dinge getan werden, z.B. das Sortieren nach der Bootzeit

```
get-wmiobject win32_operatingsystem -computername "localhost", "127.0.0.1" | ` 
Select @{Name="Computername";Expression={$_.CSName}}, ` 
Version,@{Name="OS";Expression={$_.Caption}}, ` 
@{Name="LastBoot";Expression={$_.Convert.ToDateTime($_.LastBootUpTime)}} | ` 
Sort LastBoot -Descending | format-table -autosize

Computername  Version  OS                               LastBoot
-----      -----  --                               -----
STEINAM-LAPTOP 6.1.7601 Microsoft Windows 7 Enterprise 08.04.2013 14:16:08
STEINAM-LAPTOP 6.1.7601 Microsoft Windows 7 Enterprise 08.04.2013 14:16:08
```

Der Nachteil des Verfahrens ist, dass das Original-Objekt verloren wurde.

Aufgabe

Auf welchem Rechner läuft der Druck-Spooler-Dienst. Zeigen Sie den Namen der Maschine sowie die Anzahl der benötigten Dienste.

Eine erste Annäherung:

```
PS C:\Users\admin> Get-Service Spooler | select machinename, RequiredServices  
  
MachineName          RequiredServices  
-----  
.                  { HTTP, RPCSS }
```

Falls die Druckspooler-Dienst auf dem eigenen Rechner(.) läuft, soll der Computername ausgegeben werden.

```
Get-Service Spooler | select @{Name="Computername";Expression={  
    if($_.Machinename -eq ".")  
    {  
        $env:COMPUTERNAME  
    }  
}}, RequiredServices  
  
Computername          RequiredServices  
-----  
STEINAM-LAPTOP        { HTTP, RPCSS }
```

Die Anzahl der benötigten Dienste muss noch ermittelt werden. Bisher werden nur die Dienstnamen ausgegeben. Informieren Sie sich dazu über das Measure-Cmdlet.

```
Get-Service Spooler | select @{Name="Computername";Expression={  
    if($_.Machinename -eq ".")  
    {  
        $env:COMPUTERNAME  
    }  
}},  
@{Name="Anzahl Dienste"; Expression= { ($.RequiredServices | Measure).Count }}, Name,  
→ DisplayName  
  
Computername          Anzahl Dienste Name          DisplayName  
-----  
STEINAM-LAPTOP        2 Spooler                 Druckwarteschlange
```

Auch hier geht das ursprüngliche Objekt verloren. Dies kann durch den nun folgenden Teil vermieden werden.

Hinzufügen neuer Objekteigenschaften mit Hilfe von Add-Member

Auch zu einem bestehenden Objekt können neue Eigenschaften hinzugefügt werden.

```
Get-Service Spooler | Add-Member -MemberType NoteProperty -Name "Foo" -Value 123 -  
→ PassThru
```

Mit Hilfe des **passthru**-Parameters wird die neue Eigenschaft mit in die Pipeline geworfen.

Beispiel

Informationen über logische Laufwerke per WMI

```
PS C:\> Get-WmiObject win32_logicaldisk
```

Ein Filter um nur lokale und removable Laufwerke zu sehen:

```
PS C:> Get-WmiObject win32_logicaldisk -Filter „drivetype=2 or drivetype=3“
```

Alright now that I have this I want to turn it into a nice table showing the name, freespace, and size, but freespace and size are in bytes, which I just cant stand looking at. Lets use Add-Member to create a couple of script properties and turn them into rounded GB.

```
PS C:> Get-WmiObject win32_logicaldisk -Filter "drivetype=2 or drivetype=3" | Add-
->Member -MemberType ScriptProperty -Name FreeSpaceinGB -Value {[math]::Round(($this.
->freespace / 1GB),2)} -PassThru | Format-Table Name,FreespaceinGB -AutoSize
```

Lastly I am going to use another add-member cmdlet to add a second property and get closer to my desired result. I still want to go another step, but first lets just look at the results with two properties added:

```
PS C:> Get-WmiObject win32_logicaldisk -Filter "drivetype=2 or drivetype=3" | Add-
->Member -MemberType ScriptProperty -Name FreeSpaceinGB -Value {[math]::Round(($this.
freespace / 1GB),2)} -PassThru | Add-Member -MemberType ScriptProperty -Name SizeinGB_
-> -Value {[math]::Round(($this.size / 1GB),2)} -PassThru | Format-Table name,
->FreespaceinGB,SizeinGB -AutoSize
```

Und jetzt noch den Anteil des freien Platzes in Prozent

```
Get-WmiObject win32_logicaldisk -Filter "drivetype=2 or drivetype=3" | Add-Member -
->MemberType ScriptProperty -Name FreeSpaceinGB -Value {[math]::Round(($this.
->freespace / 1GB),2)} -PassThru | Add-Member -MemberType ScriptProperty -Name_
->SizeinGB -Value {[math]::Round(($this.size / 1GB),2)} -PassThru | Add-Member -
->MemberTypeScriptProperty -Name FreespacePercent -Value {[math]::Round(([int64]$this.
->freespace / [int64]$this.size * 100),2)} -PassThru | Format-Table Name,
->FreespaceinGB,SizeinGB,FreespacePercent
```

Add-member hat die darunter liegenden .NET-Objekte nicht verändert, sondern um die Eigenschaften erweitert.

```
Get-WmiObject win32_logicaldisk -Filter "drivetype=2 or drivetype=3" | Add-Member -
->MemberType ScriptProperty -Name FreeSpaceinGB -Value {[math]::Round(($this.
->freespace / 1GB),2)} -PassThru | Add-Member -MemberType ScriptProperty -Name_
->SizeinGB -Value {[math]::Round(($this.size / 1GB),2)} -PassThru | Add-Member -
->MemberType ScriptProperty -Name FreespacePercent -Value {[math]::Round(([int64]
->$this.freespace / [int64]$this.size * 100),2)} -PassThru | Get-Member
```

11.4 Geschwindigkeitsvergleich

Unterlagen

- AB Objekterzeugung.

PowerShell v3 brings the possibility to create a custom object via

PS V3 bringt die Möglichkeit mit, eigene Objekte mit Hilfe von [pscustomobject] anzulegen.

BeispieL

```
$CustomObject2 = [pscustomobject]@{a=1; b=2; c=3; d=4}
$CustomObject2 | Format-List
```

Die Ausgabe von Get-Member zeigt, dass Note-Properties angelegt wurden

```
$CustomObject2 | Get-Member

TypeName: System.Management.Automation.PSCustomObject

Name      MemberType  Definition
----      -----      -----
Equals    Method     bool Equals(System.Object obj)
GetHashCode Method     int GetHashCode()
GetType   Method     type GetType()
ToString  Method     string ToString()
a        NoteProperty System.Int32 a=1
b        NoteProperty System.Int32 b=2
c        NoteProperty System.Int32 c=3
d        NoteProperty System.Int32 d=4
```

Warum sollte man diesen Weg wählen.

Im Vergleich zum Anlegen mit Hilfe von Hashtabellen bleibt die Einfügereihenfolge erhalten.

```
$HashTableOld = @{a=1; b=2; c=3; d=4}

$HashTableOld

Name          Value
----          ---
c              3
d              4
b              2
a              1
```

Der 2. Grund ist die deutlich erhöhte Geschwindigkeit zu allen anderen Möglichkeiten der Objekterzeugung. Folgendes Beispiel zeigt einen Geschwindigkeitsvergleich.

```
Write-Host "PS V2 Customobject"

$TestSelect = {
(0..5000) | ForEach-Object {$CustomObject = "" | Select-Object Name, ID
    $CustomObject.Name = "Test Name"
    $CustomObject.ID = $_
    $CustomObject
}
Measure-Command $TestSelect | Format-Table TotalSeconds -Autosize

Write-Host "new-object und add-member"

$TestAddMember = {
(0..5000) | ForEach-Object {
    $CustomObject = New-Object psobject
    $CustomObject | Add-Member -Name "Name" -MemberType NoteProperty -Value "Test"
    $CustomObject | Add-Member -Name "ID" -MemberType NoteProperty -Value $_
$CustomObject
}
Measure-Command $TestAddMember | Format-Table TotalSeconds -Autosize
```

```

Write-Host "New-Object und hash"

$TestProperty = {
    (0..5000) | ForEach-Object {New-Object psobject -Property @{Name = "Test Name
    ↪"; ID = $_.}}
}
Measure-Command $TestProperty | Format-Table TotalSeconds -AutoSize

Write-Host ".net-Objekte"

add-type @"
using System;
public class TestObject2{
    public string name = "";
    public int wert = 0;
}
"@
Measure-Command {
    0..5000 | ForEach-Object {
        $dotnet = New-Object TestObject2
        $dotnet.name = $_.ToString()
        $dotnet.wert = $_
        $dotnet
    } | Format-Table TotalSeconds -AutoSize
}

Write-Host "pscustomobjects"

$TestPSCustomObject = {
    (0..5000) | ForEach-Object {[pscustomobject]@{Name = "Test Name"; ID = $_.}}
}
Measure-Command $TestPSCustomObject | Format-Table TotalSeconds -AutoSize

```

Ein weiteres Skript vergleicht die Geschwindigkeit beim Erzeugen einer größeren Menge von Objekten anhand dreier unterschiedlicher Wege der Objekterzeugung.

```

[int[]]$objects = 1,2,5,10
[int]$cycles = 50000
$report = @()
ForEach ($obj in $objects) {
Switch ($obj) {
    1   {
        $temp = "" | Select Name, Objects,Cycles, Seconds
        Write-Host -fore Green "Beginning performance test for Add-Member custom_
        ↪objects using $cycles iterations and $($obj) objects"
        $run = $(Measure-Command {
            for($i = 0;$i -lt $cycles;$i++) {
                $a = new-Object psobject
                $a | add-member -membertype noteproperty -name test1 -
        ↪value "test1"
    })
    $report += $run
}
}
}

```

```

        } }).TotalSeconds
        Write-Host -ForegroundColor Cyan "$($run) seconds"
        $temp.Name = "Add-Member"
        $temp.Objects = $obj
        $temp.Cycles = $cycles
        $temp.seconds = $run
        $report += $temp
    }
2 {
    $temp = "" | Select Name, Objects,Cycles, Seconds
    Write-Host -fore Green "Beginning performance test for Add-Member custom_
objects using $cycles iterations and $($obj) objects"
    $run = $(Measure-Command {for($i = 0;$i -lt $cycles;$i++) {
        $a = new-Object psobject
        $a | add-member -membertype noteproperty -name test1 -value_
"test1"
        $a | add-member -membertype noteproperty -name test2 -value_
"test2"
    }}).TotalSeconds
    Write-Host -ForegroundColor Cyan "$($run) seconds"
    $temp.Name = "Add-Member"
    $temp.Objects = $obj
    $temp.Cycles = $cycles
    $temp.seconds = $run
    $report += $temp
}
5 {
    $temp = "" | Select Name, Objects,Cycles, Seconds
    Write-Host -fore Green "Beginning performance test for Add-Member custom_
objects using $cycles iterations and $($obj) objects"
    $run = $(Measure-Command {
        for($i = 0;$i -lt $cycles;$i++) {
            $a = new-Object psobject
            $a | add-member -membertype noteproperty -name test1 -
value "test1"
            $a | add-member -membertype noteproperty -name test2 -
value "test2"
            $a | add-member -membertype noteproperty -name test3 -
value "test3"
            $a | add-member -membertype noteproperty -name test4 -
value "test4"
            $a | add-member -membertype noteproperty -name test5 -
value "test5"
        }}).TotalSeconds
    Write-Host -ForegroundColor Cyan "$($run) seconds"
    $temp.Name = "Add-Member"
    $temp.Objects = $obj
    $temp.Cycles = $cycles
    $temp.seconds = $run
    $report += $temp
}
10 {
    $temp = "" | Select Name, Objects,Cycles, Seconds
    Write-Host -fore Green "Beginning performance test for Add-Member custom_
objects using $cycles iterations and $($obj) objects"
    $run = $(Measure-Command {for($i = 0;$i -lt $cycles;$i++) {
        $a = new-Object psobject
        $a | add-member -membertype noteproperty -name test1 -value_
"test1"
    }}).TotalSeconds
    Write-Host -ForegroundColor Cyan "$($run) seconds"
    $temp.Name = "Add-Member"
    $temp.Objects = $obj
    $temp.Cycles = $cycles
    $temp.seconds = $run
    $report += $temp
}

```

```

        $a | add-member -membertype noteproperty -name test2 -value_
"test2"
        $a | add-member -membertype noteproperty -name test3 -value_
"test3"
        $a | add-member -membertype noteproperty -name test4 -value_
"test4"
        $a | add-member -membertype noteproperty -name test5 -value_
"test5"
        $a | add-member -membertype noteproperty -name test6 -value_
"test6"
        $a | add-member -membertype noteproperty -name test7 -value_
"test7"
        $a | add-member -membertype noteproperty -name test8 -value_
"test8"
        $a | add-member -membertype noteproperty -name test9 -value_
"test9"
        $a | add-member -membertype noteproperty -name test10 -value_
"test10"
    }).TotalSeconds
    Write-Host -ForegroundColor Cyan "$($run) seconds"
    $temp.Name = "Add-Member"
    $temp.Objects = $obj
    $temp.Cycles = $cycles
    $temp.seconds = $run
    $report += $temp
}
}

Switch ($obj) {
1 {
    $temp = "" | Select Name, Objects, Cycles, Seconds
    Write-Host -fore Green "Beginning performance test for Select-Object custom_
objects using 50000 iterations and $($obj) objects"
    $run = $(Measure-Command {for($i = 0;$i -lt 50000;$i++) {
        $a = 1 |Select-Object test1,test2,test3,test4,test5,test6,test7,test8,
test9,test10
        $a.test1 = "test1"
    }}).TotalSeconds
    Write-Host -ForegroundColor Cyan "$($run) seconds"
    $temp.Name = "Select-Object"
    $temp.Objects = $obj
    $temp.Cycles = $cycles
    $temp.seconds = $run
    $report += $temp
}
2 {
    $temp = "" | Select Name, Objects, Cycles, Seconds
    Write-Host -fore Green "Beginning performance test for Select-Object custom_
objects using $cycles iterations and $($obj) objects"
    $run = $(Measure-Command {for($i = 0;$i -lt $cycles;$i++) {
        $a = 1 |Select-Object test1,test2,test3,test4,test5,test6,test7,test8,
test9,test10
        $a.test1 = "test1"; $a.test2 = "test2"
    }}).TotalSeconds
    Write-Host -ForegroundColor Cyan "$($run) seconds"
    $temp.Name = "Select-Object"
    $temp.Objects = $obj
    $temp.Cycles = $cycles
}
}
}

```

```

        $temp.seconds = $run
        $report += $temp
    }
5 {
    $temp = "" | Select Name, Objects,Cycles, Seconds
    Write-Host -fore Green "Beginning performance test for Select-Object custom_
objects using $cycles iterations and $($obj) objects"
    $run = $(Measure-Command {for($i = 0;$i -lt $cycles;$i++) {
        $a = 1 |Select-Object test1,test2,test3,test4,test5,test6,test7,test8,
test9,test10
        $a.test1 = "test1"; $a.test2 = "test2"; $a.test3 = "test3"; $a.test4 =
"test4"; $a.test5 = "test5"
    }}).TotalSeconds
    Write-Host -ForegroundColor Cyan "$($run) seconds"
    $temp.Name = "Select-Object"
    $temp.Objects = $obj
    $temp.Cycles = $cycles
    $temp.seconds = $run
    $report += $temp
}
10{
    $temp = "" | Select Name, Objects,Cycles, Seconds
    Write-Host -fore Green "Beginning performance test for Select-Object custom_
objects using $cycles iterations and $($obj) objects"
    $run = $(Measure-Command {for($i = 0;$i -lt $cycles;$i++) {
        $a = 1 |Select-Object test1,test2,test3,test4,test5,test6,test7,test8,
test9,test10
        $a.test1 = "test1"; $a.test2 = "test2"; $a.test3 = "test3"; $a.test4 =
"test4"; $a.test5 = "test5"
        $a.test6 = "test6"; $a.test7 = "test7"; $a.test8 = "test8"; $a.test9 =
"test9"; $a.test10 = "test10"
    }}).TotalSeconds
    Write-Host -ForegroundColor Cyan "$($run) seconds"
    $temp.Name = "Select-Object"
    $temp.Objects = $obj
    $temp.Cycles = $cycles
    $temp.seconds = $run
    $report += $temp
}
}

Switch ($obj) {
1 {
    $temp = "" | Select Name, Objects,Cycles, Seconds
    Write-Host -fore Green "Beginning performance test for Hash Table custom_
objects using $cycles iterations and $($obj) objects"
    $run = $(Measure-Command {for($i = 0;$i -lt $cycles;$i++) {
        New-Object PSObject -Property @{
            test1 = "test1"
        }
    }}).TotalSeconds
    Write-Host -ForegroundColor Cyan "$($run) seconds"
    $temp.Name = "HashTable"
    $temp.Objects = $obj
    $temp.Cycles = $cycles
    $temp.seconds = $run
    $report += $temp
}
}

```

```

2 {
    $temp = "" | Select Name, Objects,Cycles, Seconds
    Write-Host -fore Green "Beginning performance test for Hash Table custom_
objects using $cycles iterations and $($obj) objects"
    $run = $(Measure-Command {for($i = 0;$i -lt $cycles;$i++) {
        New-Object PSObject -Property @{
            test1 = "test1"; test2 = "test2"
        }
    }}).TotalSeconds
    Write-Host -ForegroundColor Cyan "$($run) seconds"
    $temp.Name = "HashTable"
    $temp.Objects = $obj
    $temp.Cycles = $cycles
    $temp.seconds = $run
    $report += $temp
}
5 {
    $temp = "" | Select Name, Objects,Cycles, Seconds
    Write-Host -fore Green "Beginning performance test for Hash Table custom_
objects using $cycles iterations and $($obj) objects"
    $run = $(Measure-Command {for($i = 0;$i -lt $cycles;$i++) {
        New-Object PSObject -Property @{
            test1 = "test1"; test2 = "test2"; test3 = "test3"; test4 = "test4"; test5_
-= "test5"
        }
    }}).TotalSeconds
    Write-Host -ForegroundColor Cyan "$($run) seconds"
    $temp.Name = "HashTable"
    $temp.Objects = $obj
    $temp.Cycles = $cycles
    $temp.seconds = $run
    $report += $temp
}
10{
    $temp = ""| Select Name, Objects,Cycles, Seconds
    Write-Host -fore Green "Beginning performance test for Hash Table custom_
objects using $cycles iterations and $($obj) objects"
    $run = $(Measure-Command {for($i = 0;$i -lt $cycles;$i++) {
        New-Object PSObject -Property @{
            test1 = "test1"; test2 = "test2"; test3 = "test3"; test4 = "test4";_
test5 = "test5";
            test6 = "test6"; test7 = "test7"; test8 = "test8"; test9 = "test9";_
test10 = "test10"
        }
    }}).TotalSeconds
    Write-Host -ForegroundColor Cyan "$($run) seconds"
    $temp.Name = "HashTable"
    $temp.Objects = $obj
    $temp.Cycles = $cycles
    $temp.seconds = $run
    $report += $temp
}

```

```

Switch ($obj) {
1 {
    $temp = "" | Select Name, Objects,Cycles, Seconds
    Write-Host -fore Green "Beginning performance test for psCustomObjects using
→$cycles iterations and $($obj) objects"
    $run = $(Measure-Command {for($i = 0;$i -lt $cycles;$i++) {
        [pscustomobject]@{Name = "Test Name"
    }}).TotalSeconds
    Write-Host -ForegroundColor Cyan "$($run) seconds"
    $temp.Name = "HashTable"
    $temp.Objects = $obj
    $temp.Cycles = $cycles
    $temp.seconds = $run
    $report += $temp
}
2 {
    $temp = "" | Select Name, Objects,Cycles, Seconds
    Write-Host -fore Green "Beginning performance test for Hash Table custom_
→objects using $cycles iterations and $($obj) objects"
    $run = $(Measure-Command {for($i = 0;$i -lt $cycles;$i++) {
        [pscustomobject]@{Name = "Test Name"; ID = $i
    }}).TotalSeconds
    Write-Host -ForegroundColor Cyan "$($run) seconds"
    $temp.Name = "HashTable"
    $temp.Objects = $obj
    $temp.Cycles = $cycles
    $temp.seconds = $run
    $report += $temp
}
5 {
    $temp = "" | Select Name, Objects,Cycles, Seconds
    Write-Host -fore Green "Beginning performance test for Hash Table custom_
→objects using $cycles iterations and $($obj) objects"
    $run = $(Measure-Command {for($i = 0;$i -lt $cycles;$i++) {
        New-Object PSObject -Property @{
            [pscustomobject]@{Name = "Test Name"; ID = 1; Test3="test3"; Test4 = "Test4
→"; Test5="Test5"
        }
    }}).TotalSeconds
    Write-Host -ForegroundColor Cyan "$($run) seconds"
    $temp.Name = "HashTable"
    $temp.Objects = $obj
    $temp.Cycles = $cycles
    $temp.seconds = $run
    $report += $temp
}
10{
    $temp = ""| Select Name, Objects,Cycles, Seconds
    Write-Host -fore Green "Beginning performance test for Hash Table custom_
→objects using $cycles iterations and $($obj) objects"
    $run = $(Measure-Command {for($i = 0;$i -lt $cycles;$i++) {
        [pscustomobject]@{
            test1 = "test1"; test2 = "test2"; test3 = "test3"; test4 = "test4";
→test5 = "test5"
            test6 = "test6"; test7 = "test7"; test8 = "test8"; test9 = "test9";
→test10 = "test10"
        }
    }}).TotalSeconds
    Write-Host -ForegroundColor Cyan "$($run) seconds"
    $temp.Name = "HashTable"
    $temp.Objects = $obj
    $temp.Cycles = $cycles
    $temp.seconds = $run
    $report += $temp
}
}

```

```
    } }).TotalSeconds
    Write-Host -ForegroundColor Cyan "$($run) seconds"
    $temp.Name = "HashTable"
    $temp.Objects = $obj
    $temp.Cycles = $cycles
    $temp.seconds = $run
    $report += $temp
}
}
}
$report
```


KAPITEL 12

Fehlerbehandlung

Je größer Skripts werden, desto größer ist die Wahrscheinlichkeit, dass sich Fehler in den Code einschleichen.

12.1 What-if

Um zu sehen, welche Effekte ein Skript hat, kann man es mit dem sog. What-if-Paramter versehen.

```
# What exactly would happen if Stop-Process
# ended all processes beginning with "c"?
Stop-Process -Name c* -WhatIf

PS C:\> Stop-Process -Name c* -WhatIf
WhatIf: Ausführen des Vorgangs "Stop-Process" für das Ziel "conhost (1012)".
WhatIf: Ausführen des Vorgangs "Stop-Process" für das Ziel "conhost (2000)".
WhatIf: Ausführen des Vorgangs "Stop-Process" für das Ziel "csrss (368)".
WhatIf: Ausführen des Vorgangs "Stop-Process" für das Ziel "csrss (428)".
```

What-IF kann auch in eigene Funktionen integriert werden.

```
function MapDrive([string]$driveletter, [string]$target, [switch]$whatif)
{
    If ($whatif)
    {
        Write-Host "WhatIf: creation of a network drive " +
        "with the letter ${driveletter}: at destination $target"
    }
    Else
    {
        New-PSDrive $driveletter FileSystem $target
    }
}

MapDrive k \\127.0.0.1\c$ -whatif
WhatIf: creation of a network drive + with the letter k: at destination \\127.0.0.1\c
```

12.2 Confirm

Mit Hilfe des -confirm - Parameters wird vor der Ausführung eines Befehls eine Sicherheitsabfrage formuliert.

```
Stop-Service a* -Confirm

Bestätigung
Möchten Sie diese Aktion wirklich ausführen?
Ausführen des Vorgangs "Stop-Service" für das Ziel "Anwendungserfahrung
(AeLookupSvc)".
[J] Ja [A] Ja, alle [N] Nein [K] Nein, keine [H] Anhalten [?] Hilfe
(Standard ist "J") :n

Bestätigung
Möchten Sie diese Aktion wirklich ausführen?
Ausführen des Vorgangs "Stop-Service" für das Ziel "Gatewaydienst auf
Anwendungsebene (ALG)".
[J] Ja [A] Ja, alle [N] Nein [K] Nein, keine [H] Anhalten [?] Hilfe
(Standard ist "J") :
```

Da manche Befehle „kritischer“ als andere Befehle sind, haben die Schöpfer der **Powershell** verschiedene Confirm-Level (Low, Medium, High, None) eingebaut und diese an bestimmte Befehle gebunden. So würde die Bestätigungs-abfrage beim Löschen einer User-Mail per exchange-cmdlet auch dann eine Sicherheitsabfrage hervorrufen, wenn der -confirm - Parameter nicht übergeben wird. Dies kann über die Variable \$ConfirmPreference gesteuert werden. Wenn diese auf „Low“ gesetzt wird, wird immer einer Bestätigung verlangt, bei „None“ findet keine bestätigung statt.

```
Calculator may be started and stopped without being called
# into question because Stop-Process is in the Medium category:
Calc
Stop-Process -Name calc
# If the default setting is changed from High to Low,
# PowerShell will automatically question every action:
$ConfirmPreference = "Low"
calc
Stop-Process -Name calc
```

12.3 ErrorAction

Auf einen Fehler kann Powershell mit verschiedenen Verhaltensweisen reagieren.

- SilentlyContinue: Einfach weitermachen
- Continue: Fehlermeldung anzeigen aber weitermachen
- Stop: Ausführung des Skriptes anhalten
- Inquire: Nachfragen, wie es sich verhalten soll.

Das Verhaltensweisen können mit Hilfe des Parameter *-ErrorAction* bzw. der globalen Variable *\$ErrorActionPreference* gesteuert werden.

```
Del "nosuchthing"; Write-Host "Done!"
```

```
Del "nosuchthing" -ErrorAction "SilentlyContinue"; Write-Host "Done!"  
$script:ErrorActionPreference = "Stop" #globales Anhalten bei Fehlern
```

12.4 Erkennen und Behandeln von Fehlern

Häufig will man auf Fehler auch reagieren können. Eine Möglichkeit ist es, den Fehlerstatus eines Programms auszuwerten. Dieser kann über die globale Variable `$?` abgefragt werden. Wenn ein Fehler aufgetreten ist, hat diese Variable den Wert **True**.

```
Del "nosuchthing" -ErrorAction "SilentlyContinue"  
If (!$?) { "Didn't work!"; break }; "Everything's okay!"  
  
Del "nosuchthing" -ErrorAction "SilentlyContinue"  
If (!$?) { "Error: $($error[0])"; break }; "Everything's okay!"  
Error: Cannot find path "u:\nosuchthing" because it does not exist.
```

Wenn Sie wissen wollen, welcher Fehler tatsächlich aufgetreten ist, kann mit Hilfe der `$Error`-Variable den tatsächlichen Fehler herausfinden. `$Error` ist ein Array, der an der Indexstelle 0 den jeweils aktuellsten Fehler hat

12.5 Benutzen von Traps

Als Alternative können sog. „Traps“ verwendet werden. Wenn bekannt ist, dass ein bestimmter Befehl eventuell nicht zur Laufzeit korrekt funktioniert, dann kann man dies mit Hilfe eines Traps definieren, was im Fehlerfall passieren soll.

```
Trap { "A dreadful error has occurred!" } 1/$null  
A dreadful error has occurred!  
Attempted to divide by zero.  
At line:1 char:53  
+ Trap { "A dreadful error has occurred!" } 1/$ <<< null
```

- Traps Require Unhandled Exceptions

Traps funktionieren nur mit Fehlern, die nicht von anderer Stelle „gehandelt“ werden. Das untere Statement gibt deshalb keine Fehlermeldung aus, da der Compiler den Teilungsversuch der beiden Konstanten selbst behandeln kann. Ebenfalls werden Commandlets häufig eine eigene Fehlerbehandlung haben. Diese kann man durch Setzen der `$ErrorAction` - Variable beeinflussen.

```
Trap { "A dreadful error has occurred!" } Del "nosuchthing"  
  
del : Der Pfad "C:\Users\steinam\nosuchthing" kann nicht gefunden werden, da er ↴  
nicht vorhan..  
In Zeile:1 Zeichen:42  
+ Trap { "A dreadful error has occurred!" } del "nosuchthing"  
+  
+ CategoryInfo : ObjectNotFound: (C:\Users\steinam\nosuchthing:String) ↴  
[Remove-...]  
+ FullyQualifiedErrorId : PathNotFound,Microsoft.PowerShell.Commands.RemoveItemCommand
```

```
Trap { "A dreadful error has occurred!" } `  
Del "nosuchthing" -ErrorAction "Stop"  
  
A dreadful error has occurred!  
Del : Der Pfad "C:\Users\steinam\nosuchthing" kann nicht gefun...
```

- Using Break and Continue to Determine What Happens after an Error
- Finding Out Error Details
- Error Records: Error Details
- Error Record by Redirection
- Table 11.3: Properties of an error record
- Error Record(s) Through the -ErrorVariable Parameter
- Error Records Through \$Error
- Error Record Through Traps

12.6 Try - catch -finally

<http://blogs.technet.com/b/heyscriptingguy/archive/2010/03/11/hey-scripting-guy-march-11-2010.aspx>

12.7 Exceptions verstehen

Fehler werden in modernen Programmiersprachen häufig mit dem neutralen Wort **Ausnahmen** bezeichnet. Beim Auftreten eines Fehlers wird eine Ausnahme ausgelöst, die von einer Stelle im Quellcode behandelt werden muss; ansonsten kommt es letztlich zu einer roten Fehlermeldung auf der Konsole.

Ausnahmen sind letztlich spezielle Fehlerklassen des .NET-Frameworks; für bestimmte Fehler gibt es jeweils spezielle Fehlerklassen.

In Abhängigkeit von bestimmten Fehler kann ein Skript nun unterschiedlich reagieren. Dies wird mit Hilfe des **try – catch – finally** - Konstruktes möglich.

12.8 Catching Errors in Functions and Scripts

- Stepping Through Code: Breakpoints
- Table 11.4: Settings for \$DebugPreference
- Table 11.5: Fine adjustments of the PowerShell console
- Tracing: Displaying Executed Statements
- Stepping: Executing Code Step-by-Step
- Summary

12.9 Links

<https://blogs.msdn.microsoft.com/kebab/2013/06/09/an-introduction-to-error-handling-in-powershell/>

KAPITEL 13

Aufgaben

13.1 Konsole

- Sie wollen, dass die Powershell automatisch im Administratormodus startet. Wie gehen Sie vor ?
- Beim Ausführen eines Skriptes erhalten Sie eine Fehlermeldung „Execution policy“. Welche Bedeutung hat diese Meldung und wie können Sie den Fehler abstellen
- Stellen Sie den Bildschirmhintergrund der Konsole auf Weiß und die Textfarbe auf Schwarz.
- Wie können Sie die letzten Befehle einfach wiederholen ?
- Was versteht man unter Tab-Vervollständigung
- Suchen Sie die Hilfe zu Powershell in Form einer Online-Dokumentation
- Wie lautet die Einstiegs-Website von Microsoft mit Informationen zur Powershell ?
- Erläutern Sie die Befehlzeichen | und >. Worin bestehen die Unterschiede ?
- Wie kann man in der Powershell mehrzeilige Befehle eingeben ?

13.2 Editormodus

Lösungen sind hier: loesung_powershell_editor:

13.3 Interaktivmodus

Lösungen sind hier: loesung_interaktiv:

- Welche Aliase gibt es für das Get-ChildItem
- Finden Sie mit Hilfe von Get-Help heraus, wofür das Cmdlet Stop-Process eingesetzt wird. Nutzen Sie die Hilfeinformationen, um mit Stop-Process alle Instanzen des Notepad-Editors zu schließen. Wie lautet der korrekte Befehl.
- Versuchen Sie wie im vergangenen Beispiel mit Stop-Process einen Prozess zu beenden, der gar nicht vorhanden ist, erhalten Sie eine Fehlermeldung. Wie kann man diese Fehlermeldung unterdrücken?
- Nutzen Sie das Wissen über die Parameter von Write-Host dazu, um einen weißen Text auf rotem Grund auszugeben.
- Wie kann man sich die Parameter eines Cmdlets per get-help anzeigen lassen.
- Windows hat verschiedene EventLogs, welche man mit Hilfe des Cmdlets **Get-Eventlog** prinzipiell abfragen kann. Aber wie erhält man die Namen der Eventlogs, die man einsehen will.

Versuchen Sie es mit Hilfe von **get-help get-eventlog** herauszufinden.

Listen Sie sämtliche Einträge des System-Eventlogs auf.

The screenshot shows the Windows Event Viewer window. On the left, the navigation pane is open, showing categories like System, Security, and Application. Under System, 'Administrative Ereignisse' is selected. The main pane displays a table of events from the System log. The table has columns for Ebene (Level), Datum und Uhrzeit (Date and Time), Quelle (Source), Ereignis-ID (Event ID), and Aufgabe (Task). Most events are 'Fehler' (Error) level, with timestamps ranging from April 5, 2010, to April 6, 2010. A specific event is selected, showing its details in a modal window at the bottom right. The modal window has tabs for 'Allgemein' (General) and 'Details'. It also includes a radio button for 'Angezeigte Ansicht' (Displayed View) and an 'XML-Ansicht' (XML View) button. Below these are sections for '+ System' and '- EventData'.

Ebene	Datum und Uhrzeit	Quelle	Ereignis-ID	Aufgabe
Fehler	05.04.2010 11:21:06	Google Update	20	Keine
Fehler	05.04.2010 10:21:06	Google Update	20	Keine
Fehler	05.04.2010 09:21:07	Google Update	20	Keine
Fehler	05.04.2010 09:20:55	VSS	8194	Keine
Fehler	05.04.2010 09:17:37	Dhcp-Client	1001	Adres
Fehler	05.04.2010 09:17:05	DistributedCOM	10016	Keine
Warnung	05.04.2010 09:15:30	Wininit	11	Keine
Warnung	04.04.2010 23:12:03	WLAN-AutoConfig	4001	Keine
Warnung	04.04.2010 22:29:15	MsiInstaller	1001	Keine
Warnung	04.04.2010 22:29:15	MsiInstaller	1004	Keine
Fehler	04.04.2010 22:21:07	Google Update	20	Keine

- Listen Sie nur die Ereignisse aus dem System-Ereignisprotokoll auf, die Fehler anzeigen. Schauen Sie sich dazu die Spalten an, die Get-EventLog liefert. Gibt es einen entsprechenden Parameter.
- Listen Sie alle Error-Einträge des System-Ereignisprotokolls der letzten 24 Stunden auf. Welche Parameter kennzeichnen den Zeitraum einschränken.
- Listen Sie alle Error-Einträge der letzten 24 Stunden auf, ohne ein konkretes Datum einzugeben zu müssen.
- Listen Sie alle Error-Einträge eines anderen Computers der letzten 24 Stunden auf, ohne ein konkretes Datum einzugeben zu müssen.

13.4 Variable_und_Datentyp

- Was ist das Ergebnis des folgenden Skriptes

```
[int]$a = -7  
[int]$b = 3  
Write-Host ($a/$b)  
Write-Host ([int]($a/$b))
```

- Welche Zeichen sind für das Erstellen eines Variablenamens möglich. Recherchieren Sie falls notwendig auch im Internet
- Welchen Unterschied macht es, ob sie als Kennzeichen für einen String das Zeichen “ oder das Zeichen , einsetzen.
- Wie kann man erfahren, welcher Datentyp eine Variable bzw. ein Wert hat
- Sie möchten einen String als Datum weiterverwenden, falls er ein reguläres Datum ist.
- Welche Vorteile hat das Casten eines Strings in ein DateTime-Objekt
- Welche Wertebereiche kann eine als int16 deklarierte Variable aufnehmen.
- Wie löscht man eine Variable innerhalb einer Powershell-Sitzung
- Welche Variablen sind beim Start von Powershell bereits vorhanden ? Wie kann man sie sichtbar machen
- Folgender Quellcode ist vorhanden

```
$a = "Hello"  
$b = 123
```

- Bestimmen Sie die Länge des Wertes der Variable \$a
- An welcher Stelle im String kommt der Buchstabe „e“ vor
- Tauschen Sie den Buchstaben „e“ durch den Buchstaben „a“

13.5 Operatoren

13.6 Aufgaben zu Arrays

Lösungen sind hier: loesung_powershell_arrays:

- Schreiben Sie eine Funktion, die als Parameter einen Dateipfad erwartet und anschließend den Inhalt des Pfades ausgibt.
- Wie überprüfen Sie, ob ein Array einen bestimmten Wert enthält
- Wie überprüfen Sie, ob zwei Arrays gleich sind
- Ein String ist eigentlich ein Array aus Elementen des Datentyps CHAR.

```
$test = "Hello"
$test[0].Gettype()

IsPublic IsSerial Name                                     BaseType
----- ----- 
True      True     Char                               System.
→ValueType
```

Die Ansprechen eines einzelnen Elementes ist auch mit \$test[1] problemlos möglich. Wie können Sie aber ein einzelnes Element eines bestehenden Strings mit Hilfe der Array-Indizierung ändern, also etwas so

```
$integerarray = 1,2,3
$integerarray[2] = 345 # geht

$test = "Hello"
$test[0] = "F"          #Fehlernachricht
```

- Überprüfen sie, ob ein bestimmter Wert in einem Array enthalten ist, z.B.

```
$arrColors = "blue", "red", "green", "yellow", "white", "pink", "orange",
→"turquoise"
```

Prüfen Sie, ob der Wert „green“ enthalten ist

- Haben wir Elemente, die mit „bl“ beginnen

```
$arrColors = "blue", "red", "green", "yellow", "white", "pink", "orange",
→"turquoise", "black"
```

- Sortieren sie den obigen Array alphabetisch

```
$arrColors = "blue", "red", "green", "yellow", "white", "pink", "orange",
→"turquoise", "black"
```

- Vergleichen Sie die Geschwindigkeit beim Erstellen und Suchen der folgenden Datenstrukturen: Array vs. Hashtable. Welches Ergebnis stellen Sie fest ?
- Sie besitzen eine Textdatei mit Familiennamen und Loginname der Firmenmitarbeiter. Sie umfasst 2000 Einträge und soll dazu hergenommen werden, um „vergessene“ Loginnamen zu suchen. Überlegen Sie sich, wie Sie diese Textdatei innerhalb von Powershell-Skripten nutzen können.
- Folgende Noten einer Schulaufgabe sind vorhanden:

2,3,3,2,4,5,4,3,2,1,6,4,3,2,5,4,1,2,3,6,5,4,3,2,1,4,3,2,1,5,4,4,4,3,2,3,4

Lösen Sie mit Hilfe eines Arrays die folgenden Aufgaben:

- Ermitteln Sie den Notendurchschnitt
- Ermitteln sie die Häufigkeit jeder Note (absolut/prozentual)
- Wie viele Noten liegen unter/über dem Schnitt
- Erstellen Sie eine sortierte Ausgabe in Form eines Write-Host-Statements

13.7 Bedingungen

Lösungen sind hier: loesung_powershell_bedingungen:

- Mit welchen Vergleichsoperatoren können Sie in Bedingungen arbeiten ?
- Folgende Variablen sind gegeben:

```
$user = 'steinam'  
$pass = 'password'
```

Formulieren Sie eine Bedingung in Powershell, die gleichzeitig nach einem korrekten Usernamen und Passwort fragt.

- Mit Hilfe von WMI können Sie feststellen, welche Rolle ein Computer innerhalb einer Domäne spielt. Es gibt 6 verschiedene Rollen:
 - 0 = Stand alone workstation
 - 1 = Member Workstation
 - 2 = Stand Alone Server
 - 3 = Member Server
 - 4 = Backup Domain Controller
 - 5 = Primary Domain Controller

Falls andere Werte zurückkommen, kann die Rolle nicht eindeutig definiert werden

Erstellen Sie ein Skript, welches die jeweilige Rolle in Textform ausgibt.

- Drucker auf verschiedenen Betriebssystemen ermitteln

Verschiedene Betriebssysteme nutzen manchmal unterschiedliche WMI Klassen und/oder Eigenschaften, so z.B. Windows XP, Windows 2003 und Windows 2000. Sie wollen die auf diesen Betriebssystemen verwendeten Drucker auflisten und benötigen ein universell einsetzbares Skript. Verwenden Sie ein if-Statement zum Ermitteln der korrekten OS-Version und verwenden Sie den dafür vorgesehenen Code.

Benutzen Sie die WMI-Klasse win32_OperatingSystem zum Ermitteln der OS-Version sowie win32_Printer auf XP und 2003 und win32_PrintJob auf Win2000-Systemen

- Schreiben Sie eine Anwendung, die nach Eingabe einer ganzen Zahl ausgibt, ob die Zahl gerade (restlos durch 2 teilbar, Modulo!) ist oder nicht.
- Schreiben Sie eine Anwendung, die nach Eingabe zweier Zahlen ausgibt, welche der beiden Zahlen die größere und welche die kleinere der beiden ist.
- Erweitern Sie die Anwendung um das Erkennen der Gleichheit der beiden Eingaben.
- Schreiben Sie eine Anwendung, die ermittelt, ob eine Kreditantrag aufgrund des Alters des Antragstellers in eine besondere Prüfung muss. Ist der Antragsteller nicht volljährig oder schon älter oder gleich 65 Jahre, soll eine Meldung ausgegeben werden. Falls das Alter dazwischen liegt, soll eine entsprechende Meldung ausgegeben werden.
- Schreiben Sie eine Anwendung, die zu einer eingegebenen Zahl den Kehrwert ausgibt. Der Wert muss auf Plausibilität überprüft werden!
- Schreiben Sie eine Anwendung, die zu einer Eingabe den Absolutwert ausgibt.
- Schreiben Sie eine Anwendung, die nach Eingabe zweier Zahlen vom Anwender die Summe, die Differenz, das Produkt und den Quotienten anfragt. Anschließend soll ausgegeben werden, welche Antwort falsch oder richtig

waren (im Fehlerfall mit Lösung) und wie viel Prozent der Antworten richtig waren. Vergessen Sie nicht die Plausibilitätsprüfung der Eingaben (Division!).

13.8 Schleifen

Lösungen sind hier loesung_powershell_schleifen:

- **Ping** Schreiben Sie eine Programm, welches Ihnen alle per PING erreichbaren Rechner eines Netzes ermittelt.
Das Programm soll alle Adressen von 1 bis 255 pingen. Wenn ein PING erfolgreich war, soll dies als Ausgabe angezeigt werden.

- **ZahlenRaten**

Erstellen sie ein Skript, welches folgende Aufgabe erfüllt: Es muss eine Meldung anzeigen, in der der Benutzer aufgefordert wird, eine Zahl zwischen 1 und 50 einzugeben. Das Skript muss die vom Benutzer eingegebene Zahl mit einer zufällig generierten Zahl vergleichen. Wenn die Zahlen nicht übereinstimmen, muss das Skript eine Meldung anzeigen, in der angegeben wird, ob die geratene Zahl zu hoch oder zu niedrig war, und der Benutzer aufgefordert wird, noch einmal zu raten.

Wenn der Benutzer richtig rät, muss das Skript die Zufallszahl sowie die Anzahl der Rateversuche anzeigen. An diesem Punkt ist das Spiel beendet, das Skript muss also auch beendet werden.

- **Dateien kopieren**

Diese Skripts sollen folgende Aufgaben ausführen:

- Durchsuchen des Ordners „C:Scripts“ und dessen Unterordnern.
- Durchsuchen jedes Ordners nach sämtlichen Textdateien (Dateien mit der Erweiterung .txt) und Prüfen des Erstellungsdatums jeder Datei.
- Kopieren/Verschieben jeder .txt-Datei, die mehr als 10 Tage zuvor erstellt wurde, in den Ordner „C:Old“.
- Ausgeben des Dateinamens (kein vollständiger Pfad, nur Dateiname) jeder kopierten Datei.
- Ausgeben der Anzahl der kopierten Dateien.

- **Fonts im System ermitteln**

Bei dieser Aufgabe möchten wir herausfinden, welche Schriftarten auf einem Computer installiert sind. Wir geben Ihnen einen Hinweis: Die Schriftarten sind in der Registrierung unter HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\NT\CurrentVersion gespeichert. Ihr Skript soll jedoch nicht alle Schriftarten auslesen, sondern nur die TrueType-Schriftarten. Wie können Sie eine TrueType-Schriftart von einer anderen Schriftart unterscheiden? Das ist leicht: In der Registrierung sind TrueType-Schriftarten durch den Ausdruck TrueType in Klammern direkt nach dem Schriftartnamen gekennzeichnet. Eine TrueType-Schriftart sieht also wie folgt aus: *Bauhaus 93 (TrueType)*

Zur Lösung dieser Aufgabe muss Ihr Skript Folgendes ausgeben:

- Die Namen aller TrueType-Schriftarten auf dem Computer.
- Die Anzahl der TrueType-Schriftarten auf dem Computer.
- Die Gesamtanzahl der Schriftarten auf dem Computer.

Das Ergebnis sollte in etwa so aussehen:

```
Lucida Bright (TrueType)
Lucida Bright Demibold (TrueType)
Lucida Bright Demibold Italic (TrueType)
Lucida Bright Italic (TrueType)
Lucida Calligraphy Italic (TrueType)
Lucida Fax Regular (TrueType)
```

```
TrueType: 419  
Total: 451
```

- **Schleifen optimieren**

Die Ausgabe des folgenden Powershell-Befehls dauert u.U. sehr lange. Verbessern Sie das Statement im Hinblick auf die Aufführungsgeschwindigkeit

```
# Foreach loop lists each element in a collection:  
Foreach ($element in Dir C:\ -recurse) { $element.name }
```

- **Dateien per ftp hochladen**

Sie sollen alle Dateien eines zu spezifizierenden Ordners auf einen ftp-server hochladen. Übergeben Sie den Ordnernamen auf der Kommandozeile,

- **Übersicht über laufende Dienste**

Verschaffen Sie sich einen Überblick über die laufenden Prozesse. Falls die Prozesse einen bestimmten Namen besitzen, stoppen Sie diese Prozesse bzw. geben das Wort „Gefunden <Dienstname> aus.

Erweitern Sie das Skript, indem Sie die laufenden Dienste in eine Datei schreiben, um sie später wie oben bereits geschehen, auszuwerten. Welche Schwierigkeiten können bei diesem Vorhaben auftreten.

- **Schätzen der Festplattenauslastung**

Wie lange wird der Speicherplatz einer Festplatte ausreichen, wenn ihr Inhalt jeden Monat um ca. 7,5% wächst. Die Platte hat eine Kapazität von 2 TBiT, eine Startbelegung von 100 MBiT

- **Berechnung des Notendurchschnitts und Notenverteilung eines Testes**

Ein Test ergab folgende Einzelnoten: 6,3,4,2,1,2,3,4,1,2,3,2,1,3,4,5,3,5,4,3,2,2,2,1,2,3

Berechnen Sie die Anzahl jeder Note sowie den Durchschnitt

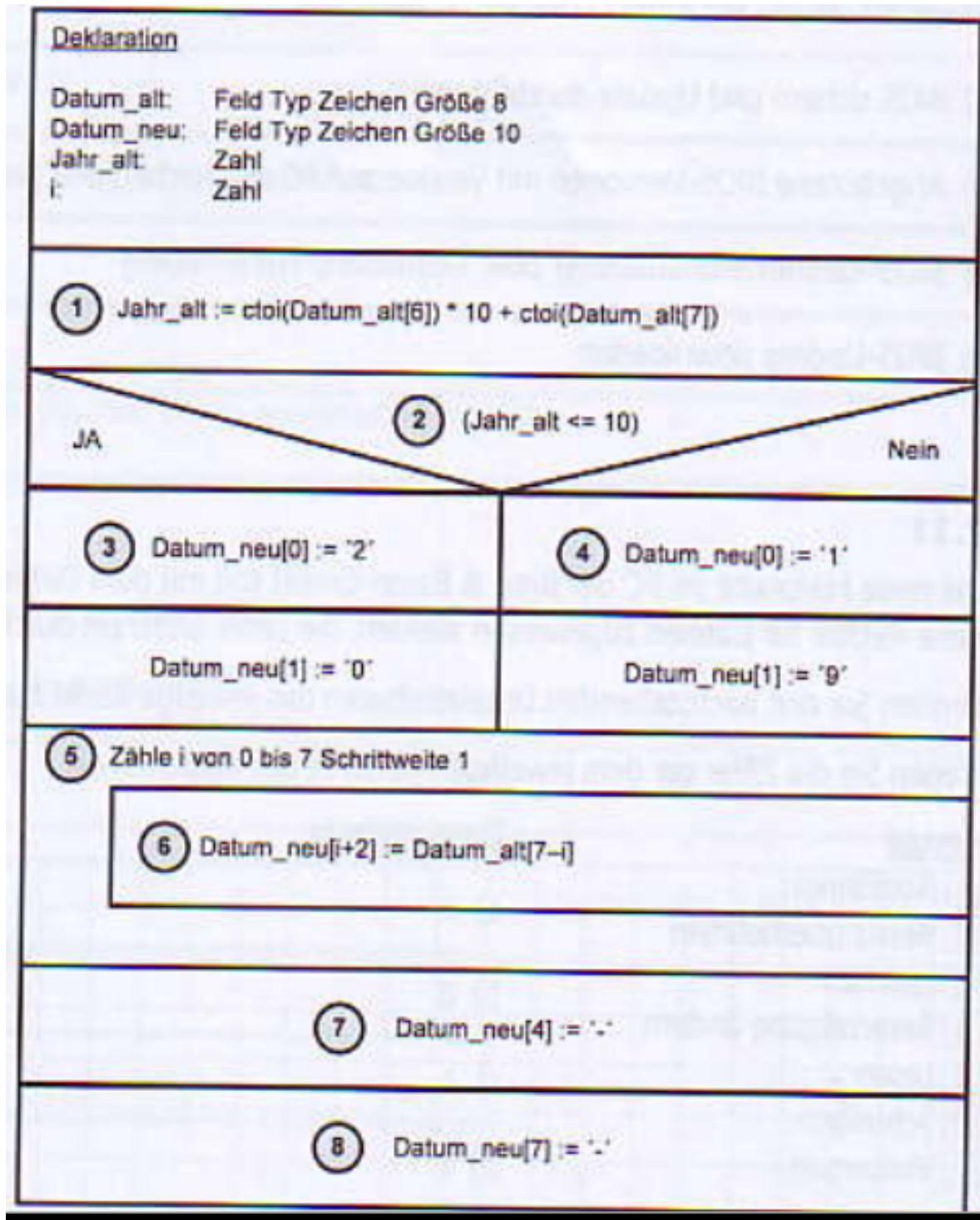
13.9 Struktogramm

loesung_struktogramm:

ZwiPruef: Frühjahr 2008

- Die alten Daten der Bims & Beton AG sollen in eine neue Datenbank übernommen werden. Das neue Datumsformat = JJJ-MM-TT, das alte TT.MM.JJ. (T=Tag, M=Monat, J=Jahr)

Die vorhandenen Daten reichen von 1940 bis 2010, daher sollen alle Jahreszahlen ≤ 10 mit einer „20“ und alle Jahreszahlen > 10 mit einer „19“ ergänzt werden. Dazu wurde nebenstehender Algorithmus entwickelt.

**Bemerkung:**

- Die Funktion ctoi wandelt ein n.Zeichen einer Zeichenkette in eine Zahl um
 - Die Zeichenkette beginnt mit dem Index 0
1. Eine der im Struktogramm mit 1,2,4,6 oder 7 gekennzeichneten Stellen stellt eine Bedingung dar. Welche ?

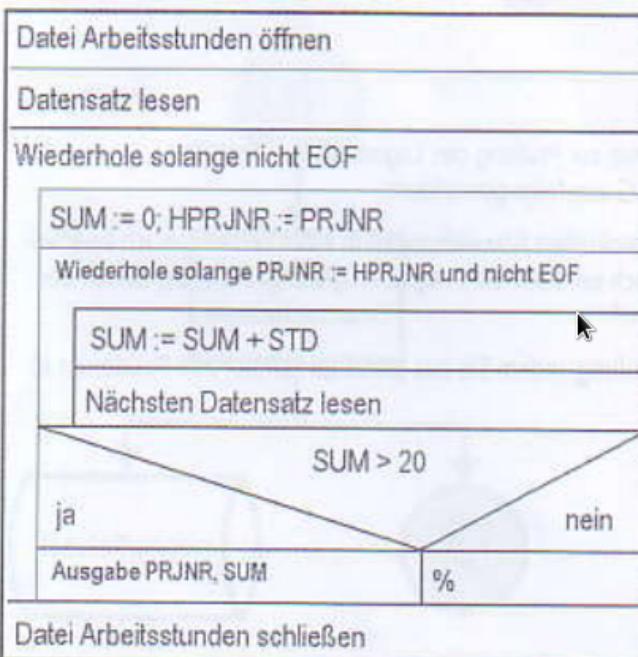
2. Das Struktogramm enthält an einer der mit 1,3,5,6 oder 8 gekennzeichneten Stellen einen Fehler.

- Viele der von der Softplus GmbH für die Maschinenbau AG entwickelten Programme enthalten kopfgesteuerte Schleifen. Welche Aussage ist richtig:
 1. Bei einer kopfgesteuerten Schleife wird die Anzahl der Durchläufe automatisch berechnet
 2. Eine kopfgesteuerte Schleife wird mindestens einmal durchlaufen
 3. Ein kopfgesteuerte Schleife kann nur durch eine Sprungsanweisung verlassen werden
 4. Wurde eine kopfgesteuerte Schleife einmal durchlaufen, wird die Bedingung nicht mehr geprüft
 5. Eine kopfgesteuerte Schleife wird nicht durchlaufen, wenn die Bedingung nicht erfüllt ist
- Die Softplus GmbH soll eine Software erstellen, mit der die für Projekte geleisteten Arbeitsstunden je Projekt ermittelt werden können. Die Arbeitsstunden für Projekte werden täglich in der Datei „Arbeitsstunden“ gespeichert, deren Datensätze nach Projektnummern **aufsteigend sortiert** sind. Es sollen die Projekte aufgelistet werden, für die insgesamt mehr als 20 Arbeitsstunden geleistet wurden. Ein entsprechendes Programmmodul wurde bereits nach unten stehendem Struktogramm erstellt. Ein Test zeigt, dass die Schleifenkopfbedingung „PRJNR := HPRJNR und nicht EOF“ nicht zu dem gewünschten Ergebnis führt.

Welche der folgenden Bedingungen ist geeignet ?

1. PRJNR = HPRJNR und nicht EOF
2. PRJNR = HPRJNR oder nicht EOF
3. PRJNR = HPRJNR oder EOF
4. PRJNR <> HPRJNR und nicht EOF
5. PRJNR <> HPRJNR oder nicht EOF
6. PRJNR <> HPRJNR oder EOF

Unterprogramm „Arbeitsstunden berechnen“
(Parameter: keine, Rückgabewerte: keine)



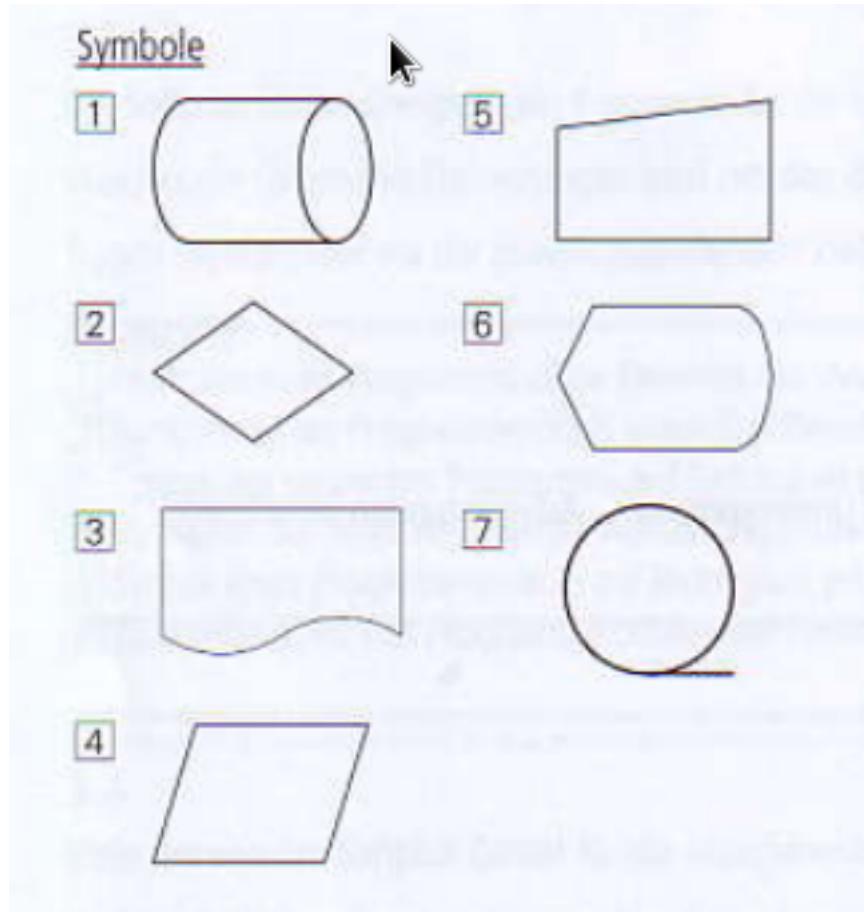
Hinweise:

PRJNR - Projektnummer
 HPRJNR - Hilfsvariable für Projektnummer
 EOF - Dateiende erreicht
 STD - Arbeitsstunde
 SUM - Summe der Arbeitsstunden

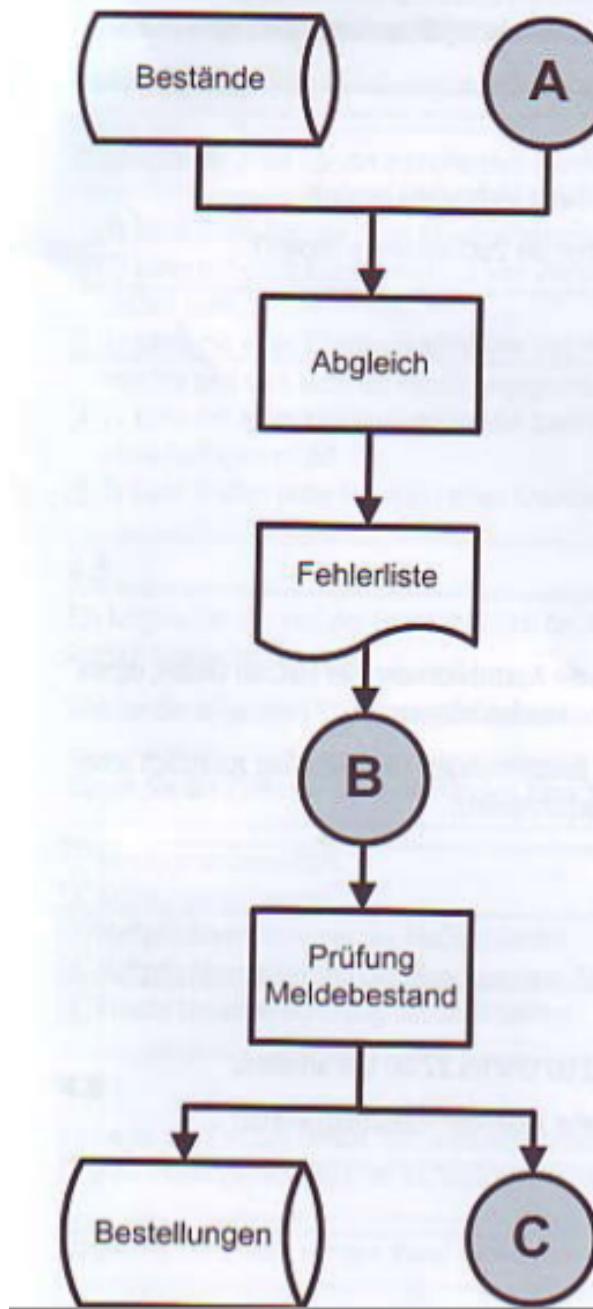
- Die Softplus GmbH soll für das Warenwirtschaftssystem der Maschinenbau AG ein Programm zur Prüfung des Lagerbestands erstellen. Der IST-„Ablauf der Lagerbestandsprüfung“ wird von einem Mitarbeiter der Maschinenbau AG wie folgt geschildert:

„Ein Programm gleicht wöchentlich die Bestandsdaten mit den Buchungsdaten ab und protokolliert Abweichungen in einer Fehlerliste. Ich bearbeite diese Fehlerliste und gebe die Korrekturen über die PC-Tastatur in das System ein. Danach ermittelt ein Programm die Lagerteile, bei denen der Meldebestand erreicht wurde, erzeugt Bestellungen im System und druckt Bestellschreiben.“

Vervollständigen Sie nebenstehenden Datenflussplan zum IST-Ablauf der Lagerbestandsprüfung, indem Sie das jeweilige Symbol den Positionen a) b) und c) zuordnen. Zur Bedeutung der Symbole informieren Sie sich bei wikipedia.



Datenflussplan der Lagerbestandsprüfung (unvollständig) zu Aufgabe 3.8



13.10 Funktionen

Lösungen sind hier: loesung_powershell_funktionen:

- Schreiben Sie eine Funktion, die als Parameter einen Pfad erwartet und anschließend den Inhalt des Pfades ausgibt. Lösung ist hier loesung_dir.
- Erweitern Sie die obige Funktion um eine rekursive Variante. Sollte der Pfad Unterordner besitzen, sollen auch diese ausgegeben werden. loesung_dir_rekursiv
- Erweitern Sie die obige Funktion: Es soll jetzt möglich sein, nach bestimmten Dateikennungen die Ausgabe zu gestalten. loesung_dir_rekursiv_extension
- Wie überprüfen Sie, ob ein Array einen bestimmten Wert enthält
- **Wie überprüfen Sie, ob zwei Arrays gleich sind** Folgende Funktion überprüft zwei Arrays auf Gleichheit

```
function AreArraysEqual($a1, $a2) {
    if ($a1 -isnot [array] -or $a2 -isnot [array]) {
        throw "Both inputs must be an array"
    }
    if ($a1.Rank -ne $a2.Rank) {
        return $false
    }
    if ([System.Object]::ReferenceEquals($a1, $a2)) {
        return $true
    }
    for ($r = 0; $r -lt $a1.Rank; $r++) {
        if ($a1.GetLength($r) -ne $a2.GetLength($r)) {
            return $false
        }
    }

    $enum1 = $a1.GetEnumerator()
    $enum2 = $a2.GetEnumerator()

    while ($enum1.MoveNext() -and $enum2.MoveNext()) {
        if ($enum1.Current -ne $enum2.Current) {
            return $false
        }
    }
    return $true
}
```

- Folgendes Powershell-Skript berechnet den Wochentag mit Hilfe eines eigenen Algorithmus. Verbessern Sie die Lesbarkeit des Skriptes, indem Sie die verschiedenen Teile des Quellcode mit Hilfe von Funktionen kapseln.

```
#Berechnung des Wochentags

[int]$t = Read-Host -prompt "Geben Sie einen Tag ein"
[int]$m = Read-Host -prompt "Geben Sie einen Monat ein"

$MerkeMonat = $m;

[int]$j = Read-Host -prompt "Geben Sie das Jahr ein"

if($m -le 2)
{
    $m += 10;
```

```

        $j -= 1;
    }
else
{
    $m -= 2;
}

[int]$c = $j/100;
[int]$y = $j%100;

[int]$h = (((26 * $m -2)/10) + $t + $y + $y/4 + $c/4 -2 * $c)%7

if ($h -lt 0)
{
    $h +=7;
}

[string]$Tag = "";
switch($h)
{
    0 { $Tag = "Sonntag"; break}
    1 { $Tag = "Montag"; break}
    2 { $Tag = "Dienstag"; break}
    3 { $Tag = "Mittwoch"; break}
    4 { $Tag = "Donnerstag";break}
    5 { $Tag = "Freitag"; break}
    6 { $Tag = "Samstag"; break}
}

Write-Host "Der " + $t +"." + $MerkeMonat + "." + $j + " ist ein " + $Tag

```

- Schreiben Sie eine Funktion, die ihnen die Größe des vorhandenen physikalischen und virtuellen RAMs ausgibt. Benutzen Sie zum Ermitteln der Größe die WMI-Klasse WIN32_OperatingSystem und deren Eigenschaften **FreePhysicalMemory** sowie **FreeVirtualMemory**. Als kleine „Zugabe“ möchten Sie die Angabe der Größe in verschiedenen Formaten (Byte, KByte, MByte) ermöglichen.
- Schreiben Sie eine Funktion zum Umbenennen eines Computernamens. Der Name soll entweder per Eingabeanforderung übergeben werden können oder aus einer Textdatei gelesen werden, falls keine Eingabe vorgenommen wird. Kommentieren Sie die Funktion über die Powershell-typische Hilfesyntax. Prüfen Sie innerhalb der Funktion, ob Sie überhaupt die administrativen Rechte für die Umbenennung haben. Zum Abschluss des Vorgangs müssen Sie einen Reboot vornehmen.
- **Powershell kann mit Hilfe des Commandlets Get-Eventlog die Ereignisanzeige von lokalen und remote-Computern anzeigen**
 - Wahl des Computers
 - Wahl des jeweiligen EventLogs
 - Wahl des jeweiligen Eintrags, z.B. Error, Warnig, Information

Informieren Sie sich mit Hilfe von get-help über die weiteren Möglichkeiten von Get-Eventlog

- Verändern von Attributen einer Datei

Sie haben als Administrator einige Dateien, auf die sie von Zeit zu Zeit zugreifen müssen. Um die Dateien vor versehentlichen Änderungen zu schützen, versehen Sie diese mit ReadOnly - Flag. Leider ist aber jetzt ein Bearbeiten dieser Dateien mühsam, weil Sie nun immer erst dieses Flag ändern müssen.

Schreiben Sie ein parametrisiertes Skript, welches alle Dateien eines Ordners ReadOnly bzw. Nicht-ReadOnly setzen kann. Schreiben Sie zusätzlich eine Funktion, die Ihnen alle Dateien mit ReadOnly-Flag auflistet. Eine Hilfe-Funktion soll Sie bei einer falschen Eingabe unterstützen.

- Ein Außendienstmitarbeiter erhält ein neues Firmen-Laptop. Auf seinem alten Laptop sind viele WLAN-Profile gespeichert und Sie suchen einen Weg, um diese Profile auf den neuen Rechner ohne manuelles Neuanlegen zu übernehmen. Ein Kollege erzählt Ihnen, dass Sie mit Hilfe des Befehls netsh Profile sowie exportieren als auch importieren können. - Informieren Sie sich über die Möglichkeiten des Befehls netsh - Schreiben Sie ein Skript, welches sowohl den Export als auch den Import der WLAN-Profile ermöglicht.
- Kommentieren Sie die Elemente des folgenden Quellcodes

- Aufruf von Remote-Desktop-Sitzungen

Erstellen Sie ein Skript, welches eine rdp-Verbindung unter Berücksichtigung verschiedener Optionen des mstsc-Befehls ermöglicht.

- Setzen Sie mit Hilfe der PS die IP-Adresse eines Rechners. Geben Sie auch die Daten für Gateway, SN-Mask, NIC, DNS-Server
- Mit Hilfe des Befehls net view \UNC-Name können Sie die Freigaben eines Rechners ermitteln. Schreiben Sie eine Funktion, die als Parameter einen UNC-Namen und einen speziellen Freigabetyp erwartet. Die Funktion soll dann die gefundenen Freigaben zurückgeben.
- Programmieren eines HangMan-Spiels

Okay, it is time to turn your attention back to the chapter's main game project, the PowerShell Hangman game. The PowerShell Hangman game is a word guessing game in which the player is challenged to guess a randomly selected secret word, a letter at a time. To win, the player must guess each letter in the word in 12 guesses or fewer.

The overall construction of the PowerShell Hangman game will be completed in 11 steps, as outlined here:

1. Create a new script file using the PowerShell script template.
2. Define and initialize game variables in the Initialization Section.
3. Define functions located in the Functions and Filters Section.
4. Prompt the player for permission to play the game.
5. **Create a loop to control overall gameplay.**
 - (a) Randomly select the game's secret word.
 - (b) Create a loop to control the collection and analysis of player input.
 - (c) Collect and validate player guesses.
 - (d) Display the result of each guess.
 - (e) Determine when the game is over.
 - (f) Challenge the player to play again.

13.11 Pipeline

Lösungen sind hier loesung_powershell_pipeline:

- You need a report of the top consumers of virtual memory and RAM on a server.
- You need a report of the largest files in a subtree.
- Die Ausgabe des folgenden Powershell-Befehls dauert u.U. sehr lange. Verbessern Sie das Statement im Hinblick auf die Ausführungsgeschwindigkeit

```
# Foreach loop lists each element in a collection:  
Foreach ($element in Dir C:\ -recurse) { $element.name }
```

- Welche Alias gibt es für das CmdLet get-process
- Welche Prozesse belegen mehr als 20 MByte
- Sortieren Sie das obige Ergebnis nach der Größe
- Geben Sie die Firma, den Produkt, Namen und Version der laufenden Prozesse aus
- Ermitteln Sie alle Word-Dateien im Verzeichnis h:Daten und seinen Unterverzeichnissen. Beschränken Sie die Ergebnismenge auf diejenigen Objekte, bei denen das Attribut Length größer ist als 40 000 Bytes. Geben Sie lediglich die Attribute Name und Length aus und sortieren Sie die Ausgabe nach der Länge. Exportieren Sie die Liste in eine .csv-Datei.
- Geben Sie alle Unterordner bis zu einer bestimmten Tiefe aus, ausgehend vom jeweils aktuellen Ordner
- Sie wollen prüfen, ob auf bestimmten Rechnern innerhalb ihres LANs Software bestimmter Hersteller vorhanden ist. Schreiben Sie ein entsprechendes Skript
- Erweitern Sie obiges Skript um die Abfrage nach bestimmten Softwarenamen bzw. Versionsständen.
- I needed to merge 365 CSV files that represent daily weather data sets into one CSV file that contains all the data accumulated during one year. Each of the daily CSV files had a header row. The yearly file should only have one. Filtering out rows is a perfect application of the PowerShell filter functions. Übersetzen Sie den englischen Text sinngemäß und schreiben Sie ein entsprechendes Skript.
- Suchen Sie die letzten 2000 Systemereignisse des Eventlogs, die vom Typ „Error“ sind
- Sie schreiben ein Powershell-Skript, welches eine eigene Ereignisquelle im Eventlog schafft und darunter seine Ereignisse schreibt.
- Schreiben Sie die Einträge der obigen Quelle in eine CSV-Datei. Es soll nur die jeweils letzte Stunde eingetragen werden
- A Real-life Example of Select-String

My practical problem was that I wanted to search for instances a particular string in a directory with hundreds of file. If a file had duplicate entries, then I needed to know.

The PowerShell problem is how to create a stream of the files, then compare each file with my string value. In the output I needed to know the filename.

To solve the problem, I employed four main commands, which you can see in the following script:

- Auflösung einer IP-Range in deren Namen

Erstellen Sie ein Skript, welches eine vordefinierte IP-Range durchläuft und zu jeder IP-Adresse bei Erreichbarkeit den Rechnernamen ausgibt.

Versuchen Sie dabei mögliche Fehleingaben durch Programmierlogik abzufangen

Speichern Sie anschließend die Kombination von IP-Adresse und Rechnername in einer CSV-Datei

Für die Implementierung können Sie folgende Funktionalität benutzen:

Test-Connection: Cmdlet, welches die Erreichbarkeit eines Rechners zurückgibt

[System.Net.Dns]::GetHostbyAddress(\$IPAddress) Funktionalität des .net-Frameworks, welches eine Namensauflösung für eine gegebene IP-Adresse vornimmt.

- Laptop oder Desktop

Sie sind die IT-Leiter einer kleinen mittelständischen Firma. Sie müssen Inventur machen und Sie brauchen dazu die Anzahl der zur Zeit sich im Netz befindlichen Laptop-und Desktop-Computer in ihrer Firma.

Um eine weitere Verarbeitung der Daten zu ermöglichen, sollte Sie den Computernamen, seine IP-Adresse, die Uhrzeit sowie die Aussage, ob es ein Laptop oder Desktop ist, speichern.

Sie planen folgende Vorgehensweise:

- Ihr Skript soll während des Login-Vorganges aufgerufen werden. Dies ist von Ihnen nicht zu programmieren.
- Das Skript sucht eine Textdatei auf einem Netzlaufwerk und schreibt die entsprechenden Daten in die Datei Es muss nicht darauf geachtet werden, dass evtl. ein gleichzeitiger Aufruf mehrerer Clients möglich wäre.
- Falls ihr Rechner in der Datei schon existiert, sollen die Daten aktualisiert werden
- Am Ende soll die Anzahl der Laptop- und Desktop-PC ermittelt und ausgegeben werden. Dies würde typischerweise in einem zweiten Skript stattfinden; sie können dies aber aus Vereinfachungsgründen auch in ihre vorhandene Lösung einbauen.

- Authentifizierung am Radius-Server

Eine Schule plant, seinen Schülern einen definierten Gast-Zugang zum Internet mit eingeschränkten Rechten (http) per WLAN anzubieten. Dazu müssen die Access-Points mit den jeweiligen Login-Daten der Gäste versehen sein. Dies wird an den Access-Points mit Hilfe von Textdateien (CSV) realisiert, die den Usernamen, Passwort sowie das Datum der Erstellung beinhalten. Neue Gäste sollen jederzeit zu den bestehenden Gastzugängen hinzugefügt werden können. Diese müssen dann in die Textdatei aufgenommen werden; die Textdatei wird anschließend auf dem Accesspoint überschrieben.

An den Login-Namen wird folgende Anforderung gestellt: Erster Buchstabe des Vornamens + Nachname

Das Passwort hat folgenden Aufbau:

gast + 4 stellige Zufallszahl

Das Datum soll im Format dd.mm.yyyy in der Textdatei gespeichert werden

Zur ersten Befüllung der Textdatei hat der Systemadministrator eine Textdatei mit dem Vornamen und Nachnamen der Schüler aus dem Active Directory extrahiert.

Aufgabe:

Erstellen Sie die Textdatei für die Authentifizierung am Access-Point.

13.12 Objekte

13.12.1 Aufgabe 1

Passen Sie das folgende Skript so an, dass die Informationen nicht lediglich per Write-Host ausgegeben werden, sondern in Form eines Objektes gespeichert werden.

```

Function SysInfo {
    $colItems = Get-WmiObject Win32_ComputerSystem ` 
    -Namespace "root\CIMV2" -ComputerName $strComputer

    foreach($objItem in $colItems) {
        Write-Host "Computer Manufacturer: " $objItem.Manufacturer
        Write-Host "Computer Model: " $objItem.Model
        Write-Host "Total Memory: " $objItem.TotalPhysicalMemory "bytes"
    }
}

Function BIOSInfo {
    $colItems = Get-WmiObject Win32_BIOS -Namespace "root\CIMV2" ` 
    -Computername $strComputer
    foreach($objItem in $colItems) {
        Write-Host "BIOS:"$objItem.Description
        Write-Host "Version:"$objItem.SMBIOSBIOSVersion"."
        $objItem.SMBIOSMajorVersion"."$objItem.SMBIOSMinorVersion
        Write-Host "Serial Number:" $objItem.SerialNumber
    }
}

Function OSInfo {
    $colItems = Get-WmiObject Win32_OperatingSystem -Namespace "root\CIMV2" - 
    Computername $strComputer

    foreach($objItem in $colItems) {
        Write-Host "Operating System:" $objItem.Name
    }
}

Function CPUInfo {
    $colItems = Get-WmiObject Win32_Processor -Namespace ` 
    "root\CIMV2" -Computername $strComputer

    foreach($objItem in $colItems) {
        Write-Host "Processor:" $objItem.DeviceID $objItem.Name
    }
}

Function DiskInfo {
    $colItems = Get-WmiObject Win32_DiskDrive -Namespace "root\CIMV2" ` - 
    ComputerName $strComputer
}

```

```

        foreach($objItem in $colItems) {
            Write-Host "Disk:" $objItem.DeviceID
            Write-Host "Size:" $objItem.Size "bytes"
            Write-Host "Drive Type:" $objItem.InterfaceType
            Write-Host "Media Type: " $objItem.MediaType
        }
    }

Function NetworkInfo
{
    $colItems = Get-WmiObject Win32_NetworkAdapterConfiguration ` -Namespace
    ↵"root\CIMV2" -ComputerName $strComputer | where{$_.IPEnabled -eq "True"}

    foreach($objItem in $colItems) {
        Write-Host "DHCP Enabled:" $objItem.DHCPEnabled
        Write-Host "IP Address:" $objItem.IPAddress
        Write-Host "Subnet Mask:" $objItem.IPSubnet
        Write-Host "Gateway:" $objItem.DefaultIPGateway
        Write-Host "MAC Address:" $objItem.MACAddress
    }
}

#*-----#
#* SCRIPT BODY
#*-----#
#* Connect to computer
$strComputer = "."

#* Call SysInfo Function
Write-Host "System Information"
SysInfo
Write-Host

#* Call BIOSinfo Function
Write-Host "System BIOS Information"
BIOSInfo
Write-Host

#* Call OSInfo Function
Write-Host "Operating System Information"
OSInfo
Write-Host

#* Call CPUInfo Function
Write-Host "Processor Information"
CPUInfo
Write-Host

#* Call DiskInfo Function
Write-Host "Disk Information"
DiskInfo
Write-Host

#* Call NetworkInfo Function
Write-Host "Network Information"

```

```
NetworkInfo  
Write-Host
```

13.12.2 Aufgabe 2

Ihn ihrer Firma wird ein Skript genutzt, welches die Aufrufe der einzelnen Webseiten pro Wochentag aus den Log-Files des Webservers erstellt und in einer Datei „weekly_Stats.csv“ speichert.

Die Struktur der Datei sieht wie folgt aus:

```
site_url, monday, tuesday, wednesday, thursday, friday, saturday, sunday  
index.php,20000,14324,23453,43213,56432,23451,12343  
login/login.php, 23432,34321,34567,43567,5432,43256,0  
login/logout.php,12345,22345,32456,4232,3453,23456,1234  
....
```

Insgesamt ist von ca 700 Zeilen in der Datei auszugehen.

Ihr Chef möchte nun als weitere Information für die jeweiligen urls den wöchentlichen Durchschnitt pro url berechnet bekommen und eine nach dem Durchschnitt sortierte Ausgabe in einer Textdatei speichern.

Erstellen Sie ein entsprechendes Skript. Benutzen Sie wenn möglich objektorientierte Ansätze sowie die Pipeline.

13.12.3 Aufgabe 3

Gegeben ist folgendes PS-Skript.

```
Function Get-NetworkConfiguration  
{  
    param (  
        [parameter(  
            ValueFromPipeline=$true,  
            ValueFromPipelineByPropertyName=$true,  
            Position=0)]  
        [Alias('__ServerName', 'Server', 'Computer', 'Name')]  
        [string[]]  
        $ComputerName = $env:COMPUTERNAME,  
        [parameter(Position=1)]  
        [System.Management.Automation.PSCredential]  
        $Credential  
    )  
    process  
    {  
        $WMIParameters = @{  
            Class = 'Win32_NetworkAdapterConfiguration'  
            Filter = "IPEnabled = 'true'"  
            ComputerName = $ComputerName  
        }  
  
        if ($Credential -ne $Null)  
        {  
            $WmiParameters.Credential = $Credential  
        }  
        foreach ($adapter in (Get-WmiObject @WMIParameters))  
        {  
            $OFS = ', '  
        }  
    }  
}
```

```

        Write-Host "Server: $($adapter.DNSHostName)"
        Write-Host "Adapter: $($adapter.Description)"
        Write-Host "IP Address: $($adapterIpAddress)"
        Write-Host "Subnet Mask: $($adapter.IPSubnet)"
        Write-Host "Default Gateway: $($adapter.DefaultIPGateway)"
        Write-Host "DNS Servers: $($adapter.DNSServerSearchOrder)"
        Write-Host "DNS Domain: $($adapter.DNSDomain)"
        Write-Host
    }
}

}

```

- Wählen Sie statt der Write-Host-Ausgabe einen objektorientierten Ansatz
- Sie wollen nicht alle Informationen ausgeben, sondern beispielsweise nur die IP-Adresse und die Subnet-Maske
- Können Sie mit diesem Skript mehrere Rechner abfragen ?
- Welche Rechner ihres Netzes benutzen den gleichen DNS-Server mit der IP-Adresse *.*.**.**. Eine Liste der Computer erhalten Sie mit Hilfe des CommandLets get-adcomputer
- Wie viele Rechner benutzen den gleichen Default-Gateway
- Übersetzen Sie den folgenden Text:

When you begin to output options from your functions, the reusability and extensibility options soar. Objects allow you to take advantage of the pipeline (which allows your objects to be the input to other commands), allow your commands to use incoming objects (via the Parameter attribute and ValueFromPipelineByPropertyName switch), and most importantly, allow you to leverage the formatting and output system that Windows PowerShell provides, which keeps you from having to design your own output options and saves your effort for the real work—getting your job done!

13.13 Fehlerbehandlung

13.14 Active Directory

- Sie planen das E-Learning-System Moodle (<http://www.moodle.org>) zu installieren. Um die Benutzerverwaltung zu vereinfachen planen Sie, die User aus dem ActiveDirectory ihrer Windows-Domäne als Benutzer in mysql anzulegen. Schreiben Sie ein entsprechendes Skript.

KAPITEL 14

PS und Objektorientierung

14.1 Objektorientierung

Objektorientierte Programmierung ist ein Programmierstil, der seit vielen Jahrzehnten als grundlegendes Programmierparadigma vorherrscht. **Alle** Programmiersprachen sind heutzutage objektorientiert; auch PowerShell. Leider hat man es bis zur Version 5 der Powershell dem Systemintegrator schwer gemacht, die gängigen objektorientierten Konzepte in der Powershell einzusetzen, weil einfach die sprachlichen Konstrukte in PowerShell nicht vorhanden waren.

Dies hat sich aber nun grundleg.

jhd ds jfds jf

KAPITEL 15

Powershell und Klassendiagramm

15.1 Vorwort

In der objektorientierten Programmierung ist das Klassendiagramm ein zentrales Diagramm der Modellierung. Viele Programmiersprachen lehnen sich eng an dieses Konzept an und bieten innerhalb ihrer Syntax eine entsprechende Umsetzung der jeweiligen Begriffe des Klassendiagramms an.

Dieses Skript ist deshalb zweigeteilt: Es werden sowohl die grundlegenden Begriffe der OOP bzgl. des Klassendiagramms erläutert als auch der Versuch unternommen, diese realitätsnah und praxisbezogen mit Hilfe von Powershell darzustellen.

Der exemplarische Einsatz der Programmiersprache Powershell hat dabei allerdings einen Nachteil. Powershell kann von seiner Syntax nicht alle Konzepte des KD 1:1 abbilden bzw. man muss dies über Mechanismen erreichen, die in der reinen Welt von java/php/c#/c++ eben nicht notwendig wären.

Da Powershell auf das .NET-Framework aufsetzt, ist prinzipiell jede Feature von OOP umsetzbar; häufig wird ein Skripter aber gar nicht die Notwendigkeit sehen, diese Feature einzusetzen zu wollen. Allerdings wird er häufig mit Objekten aus der .NET-Welt arbeiten und damit viele Elemente des Klassendiagramm nutzen (ohne es wissen zu müssen bzw zu wollen)

15.2 ERM - Klassendiagramm

Da in der 11. Klasse das Thema **Entity Relationship Model** bereits behandelt wurde, fällt ein Einstieg in die Begriffs-welt des Klassendiagrammes relativ leicht.

Einfache Definition Klassendiagramm

Ein Klassendiagramm ist ein Entity Relationship Modell, welches um Methoden erweitert wird.

Diese Aussage trifft in seiner Kürze des Kern und ist für einen Skripter völlig ausreichend. Das Klassendiagramm übernahm alle Konzepte des ER-Modells und fügte ihm einige Punkte hinzu. Diese wären beispielsweise

- Erweiterung der Klasse um Methoden
- Abstrakte Klasse / Interface
- Mehrfachvererbung
- Detailliertere Darstellung der Beziehungen zwischen Objekten

In der Summe sollte man deshalb folgende Begriffe der OOP/Klassendiagrammes kennen:

- Klasse / Objekt / Instanziierung
- Konstruktor/Destruktor
- Statische Attribute und Methoden
- Attribute / Methoden / Properties
- Geheimnisprinzip (public/private/protected/internal/....)
- **Vererbung**
 - Basisklasse / Abgeleitete Klasse
 - Einfach-, Mehrfachvererbung
 - Abstrakte Klasse / Interface
 - Überschreiben von Methoden/Polymorphie / Late vs. early Binding

-Beziehungen zwischen Klasse

- Assoziation
- Aggregation
- Komposition

15.3 Klasse / Objekt / Instanziierung

Die **Klasse** ist der grundlegende Begriff in der OOP.

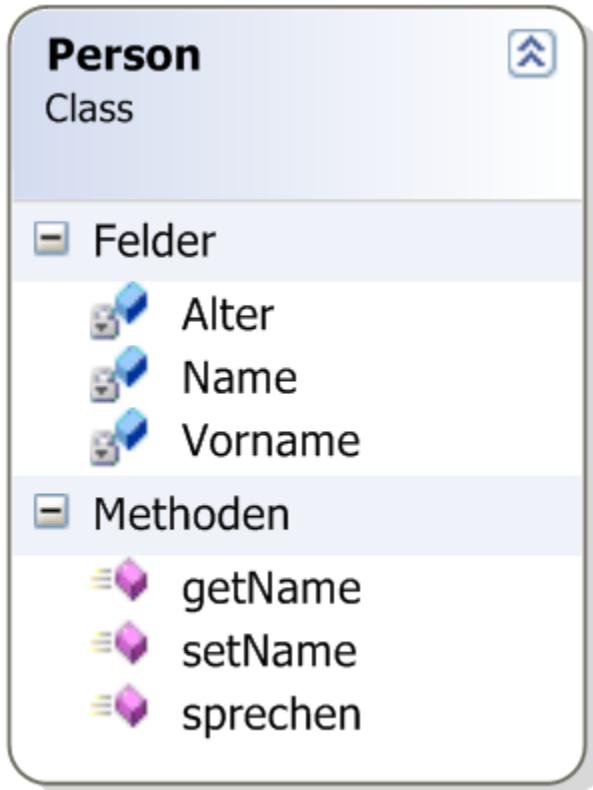
Über die Klasse definiert man zusammengehörende Informationen in einem gemeinsamen Container, der Klasse.

Die Klasse erhält einen Namen, die zu speichernden Informationen bezeichnet man als Attribute der Klasse.

Die Manipulation, d.h. das Schreiben, Lesen und Ändern der Daten, obliegt ebenfalls dem Verantwortungsbereich der Klasse und wird . Neben diesen Aufgaben kann eine Klasse noch weitere Fähigkeiten besitzen. Diese Fähigkeiten werden in der Klasse durch Methoden definiert.

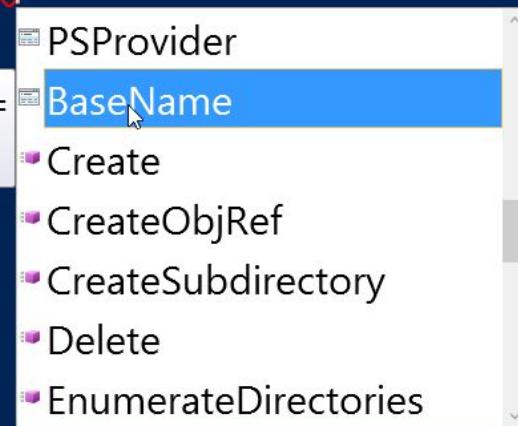
Eine Klasse besteht deshalb zumindest aus 3 Bereichen.

- Dem Klassennamen
- Der Liste der Attribute
- Der Liste der Methoden



Neben den wirklichen Fähigkeiten, im obigen Beispiel sprechen, muss eine Klasse häufig über Verwaltungsmethoden verfügen, die eine Manipulation der internen Attribute ermöglichen. Man nennt diese Methode häufig getter/setter-Methoden. Da ein Methodename lediglich einmal genutzt werden darf, muss man deshalb häufig 2 Methodennamen benutzen.

```
PS C:\Users\Karl> $result = Get-ChildItem C:\temp  
PS C:\Users\Karl> $datei = $result[0]  
PS C:\Users\Karl> $datei.|
```



Die Klasse definiert als Bauplan lediglich die Art der gehaltenen Informationen bzw. die Methoden, die zur Manipulation der Daten bzw. zur Funktionalität der Klasse notwendig sind. Doch wie kann man nun diese Klasse nutzen?

Da wir ja Informationen zu einem realen Gegenstand des Systems erheben wollen, müssen wir den Bauplan der Klasse einem realen Objekt zuordnen. Nur ein reales Objekt kann Daten speichern. Dieser Vorgang wird Instanziierung genannt und läuft in folgenden Schritten ab:

- Deklaration einer Variablen vom Typ der Klasse
- Instanziierung des Objektes mit Hilfe des new-Operators.

Der new-Operator erzeugt ein sog. Objekt der Klasse. Dieses Objekt ist einzigartig und ist nun in der Lage, die Informationen aufzunehmen und zu verarbeiten. Bei allen Instanziierungen wird immer ein Konstruktor aufgerufen. Er lautet wie der Name der Klasse und kann überladen sein, d.h. in verschiedenen Versionen existieren. Durch ihn ist das Objekt in der Lage, Zustände seiner Variablen bei der Erzeugung zu kontrollieren. Definiert man eigene Konstruktoren, so muss der parameterlose Standardkonstruktor ebenfalls angegeben werden, wenn man ihn zur Verfügung stellen will.

Erst nach dem Erzeugen kann man nun die Fähigkeiten des Objektes benutzen, d.h. man kann die Methoden der Klasse benutzen. Merke: Methoden werden auf Klassenebene definiert, aber auf Objektebene genutzt ! (Es gibt aber eine Ausnahme ! Welche ?)

Innerhalb der Klasse gibt es häufig eine spezielle Methode, den sog. Konstruktor. Sie hat den gleichen Namen wie die Klasse selbst und sie kann in verschiedenen Variationen vorliegen. Beim Erzeugen eines Objektes (s.u.) wird diese Methode als Erstes aufgerufen. Wird diese Methode nicht innerhalb der Klasse definiert, so wird ein sog. Standardkonstruktor benutzt. Er besteht lediglich aus dem Methodennamen ohne irgendwelche Parameter. Werden eigene Konstruktoren geschrieben, so muss der Standardkonstruktor explizit definiert werden, sonst ist er nicht mehr vorhanden.

In einem zusammenhängenden Stück Quellcode könnte eine Programmiersprache die oben dargestellten Konzepte wie folgt umsetzen.

```
class Person
{
    int mAlter;
    string mName;
    string mVorname;

    //Standardkonstruktor
    public Person()
    {

    }

    //Übeladener Konstruktor
    public Person(string _Name)
    {
        mName = _Name;
    }

    public void sprechen (string Aussage)
    {
        Console.WriteLine(Aussage);
    }

    public void setName(string _name)
    {
        mName = _name;
    }

    public string getName(string _name)
    {
        return Name;
    }
}
```

```

//Destruktor
//nicht nötig; wird beim Zerstören eines Objektes aufgerufen
~Person()
{
    Console.WriteLine("Aargh");
}
}

class Program
{

    public static void Main()
    {
        Person STE = new Person();
        Person SYC = new Person();
        Person SMA = new Person();
    }
}

```

15.4 Powershell und (Klasse/Attribut/Methode/Instanziierung)

Die Skriptsprache Powershell hat im Bezug auf die Konzepte „Klasse, Objekt, Attribut, Methode“ andere Umsetzungen gewählt. Insbesondere das Konzept der Klasse zur Definition von Eigenschaften u. Methoden ist unbekannt und es gibt mehrere Möglichkeiten, wie man Objekte erzeugen kann. Diesem werden dann beim Erzeugen die entsprechenden Attribute und Methoden zugewiesen.

```

# Neues leeres Objekt erstellen
$Obj= New-object -TypeName PSObject

# Wert an das Objekt anfügen
Add-Member -InputObject $Obj -Name Alter -Value 50 -MemberType NoteProperty
# Wert an das Objekt anfügen
Add-Member -InputObject $Obj -Name Name -Value "STE" -MemberType NoteProperty
# Wert an das Objekt anfügen
Add-Member -InputObject $Obj -Name Vorname -Value "Kurt" -MemberType NoteProperty

```

Powershell vermischt auf diese Weise die Definition der Klasse mit dem Erzeugen eines Objektes. Dies kann dadurch vermieden werden, dass man Module als Definition von Klassenstrukturen benutzt.

15.5 Geheimnisprinzip

So wie Objekte im realen Leben auch nicht jedes Geheimnis nach draußen preisgeben, so gilt dies auch in der OOP. Das Objekt sollte prinzipiell den Zustand seiner Attribute versteckt halten, d.h. keinen direkten Zugriff auf seine Attribute erlauben.

Durch das zur Verfügung stellen von entsprechenden getter/setter-Methoden bzw. Properties kann das Objekt den Zugriff auf seine Attribute kontrollieren.

Der generelle Zugriffsmöglichkeit auf Attribute und Methoden wird in der OOP über die **Sichtbarkeiten** definiert. Diese stehen vor dem Attribut bzw. der Methode und definieren den möglichen Zugriff von außerhalb auf die Attribute und Methoden. Gängige Definitionen von Sichtbarkeiten sind:

- public (+): Öffentlich sichtbar, von überall aufruf- und damit manipulierbar
- private (-): Nur innerhalb des Objektes selbst benutzbar
- protected/internal (#): Nur innerhalb des gleichen Namespaces bzw. über Vererbungsmechanismen sichtbar.

Des Weiteren kann in der PowerShell auch ein Gültigkeitsbereich für die Variablen festgelegt werden. Es gibt folgende Gültigkeitsbereiche: - Global: Überall sichtbar - Script: In allen Bereichen der Skriptdatei sichtbar - Local: Nur im aktuellen Bereich und Unterbereichen sichtbar - Private: Nur im aktuellen Bereich sichtbar Wird kein Gültigkeitsbereich angegeben für eine Variable gilt per Default „Local“ als Gültigkeitsbereich. Möchte man z.B. eine Variable haben die überall sichtbar ist geht das wie folgt: \$ global:z = "A",

http://www.computerperformance.co.uk/powershell/index_real_life.htm

KAPITEL 16

Indizes und Tabellen

- genindex
- modindex
- search

Stichwortverzeichnis

-not, 44
\$DebugPreference, 88
\$_, 120
\$errorActionPreference, 88
Ablaufsteuerung, 85
Add-Member, 130, 134
Add-Type, 131
Alias, 17, 97, 121
and, 45
Anlegen, 62
Argumentübergabe, 80
Arguments
 Arbitrary, 80
 Named, 80
 Predefined, 81
 Special, 82
 Typed, 81
Array, 13, 59, 62, 66, 70
asCustomObjects, 130
Bedingung, 53
break, 48, 58
case-sensitive, 30
cmdlet, 15, 30
condition, 53
confirm, 145
continue, 58
Datei, 97
Datentyp, 30
Debug-Write, 88
default, 48
do_while, 57
else, 53
Entfernen, 66
Fehlerbehandlung, 143
Filter, 120, 121, 217
first, 121
for, 55
foreach, 57
Foreach-Object, 121
foreach-object, 57
Format-Cmdlet, 113
Format-List, 113
Format-Table, 113
function, 77
Funktion, 77
Gültigkeit, 30
Group-Object, 117
Hash, 74, 136
Hashtable, 74
Help-Message, 103
Hinzufügen, 66
if, 53
if-else, 53
Increment, 56
Index, 66
Initialisierung, 56
Inline-Help, 89
Kopieren, 70
last, 121
Mandatory, 103
New-Module, 130
noelement, 119
notation, 97
Objekt, 128
 Add-Member, 130, 134
 Add-Type, 131
 Geschwindigkeit, 135
 New-Module, 130

psCustomObject, 135
Select-Object, 130
Operator, 30, 31
or, 45
Out-Default, 109

Parameter, 101
passThru, 134
Pipeline, 106, 217
 Blocking, 111
 sequentieller Modus, 111
 Streaming, 111
 streaming Modus, 112
psCustomObject, 135

Rückgabewert, 83
Random, 97
Return, 86

Scope, 30
Select-Object, 121, 130, 217
Sort-Object, 115, 217
switch, 47, 48

Tee-Object, 122
Test-Path, 30
Typisiert, 30
typisiert, 70

Umgebungsvariable, 30
Untypisiert, 30

Validierung, 101
Variable, 30
Vergleichsoperator, 43, 53

What_if, 145
Where-Object, 120, 217
while, 57
wildcard, 49
WMI, 92
Write-Host, 87

Zählschleife, 55