Windows PowerShell 3.0

Das Praxisbuch

SCHWICHTENBERG, MONADJEMI

3., aktualisierte Auflage

ADDISON-WESLEY

29

Anwendungsgebiete

Anleitungen und Beispiele zu zahlreichen Einsatzszenarien von Dateisystem und Diensten über Netzwerk und Hyper-V bis Datenbanken und Active Directory

977

Code-Beispiele

Auch zum Download: vom ersten PowerShell-Skript bis zur kommandozeilenbasierten Administration

www.IT-Visions.de

Dr. Holger Schwichtenberg

550

Commandlets

Zentrale PowerShell-Befehle werden in konkreten Anwendungsbeispielen vorgestellt

ALWAYS LEARNING

PEARSON

Bibliografische Information der Deutschen Nationalbibliothek

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über *http://dnb.d-nb.de* abrufbar.

10 9 8 7 6 5 4 3 2 1

15 14 13

ISBN 978-3-8273-3175-5 Print; 978-3-8273-3175-5 PDF

Einbandgestaltung: Kochan & Partner, München
Lektorat: Thomas Pohlmann
Korrektorat: Petra Kienle
Herstellung: Claudia Bäurle, cbaeurle@pearson.de
Satz: Nadine Krumm, mediaService, Siegen, www.mediaservice.tv
Druck und Verarbeitung: Drukarnia Dimograf, Bielsko-Biala
Printed in Poland

3 Einzelbefehle der PowerShell

Die PowerShell kennt folgende Arten von Einzelbefehlen:

- Commandlets (inkl. Funktionen)
- Aliase
- Ausdrücke
- Externe Befehle
- Dateinamen

3.1 Commandlets

Commandlet

Ein „normaler" PowerShell-Befehl heißt *Commandlet* (kurz: *Cmdlet*) oder *Funktion* (*Function*). Eine Funktion ist eine Möglichkeit, in der PowerShell selbst wieder einen Befehl zu erstellen, der funktioniert wie ein Commandlet. Da die Unterscheidung zwischen Commandlets und Funktionen aus Nutzersicht zum Teil akademischer Art ist, erfolgt hier zunächst keine Differenzierung: Das Kapitel spricht allgemein von Commandlets und meinte damit auch Funktionen.

Ein Commandlet besteht typischerweise aus drei Teilen:

- einem Verb,
- einem Substantiv und
- einer (optionalen) Parameterliste.

Verb und Substantiv werden durch einen Bindestrich „–" voneinander getrennt, die optionalen Parameter durch Leerzeichen. Daraus ergibt sich der folgende Aufbau:

```
Verb-Substantiv [-Parameterliste]
```

Die Groß- und Kleinschreibung ist bei den Commandlet-Namen nicht relevant.

Ein einfaches Beispiel ohne Parameter lautet:

```
Get-Process
```

Dieser Befehl liefert eine Liste aller laufenden Prozesse im System.

Ein zweites Beispiel ist:

```
Get-ChildItem
```

Dieser Befehl liefert Unterelemente des aktuellen Standorts auf. Meist ist der aktuelle Standort ein Dateisystempfad. In der PowerShell kann der aktuelle Standort aber auch in der Registrierungsdatenbank, dem Active Directory und vielen anderen (persistenten) Speichern liegen.

Ein drittes Beispiel ist

```
Get-Service
```

Dieser Befehl liefert alle Windows-Systemdienste.

Das waren alles Dienste, die Informationen liefern. Dienste, die Aktionen ausführen (z.B. Prozesse beenden, Dateien löschen, Dienste anhalten) kommen in der Regel nicht ohne Parameter aus, da sie sonst ja global alle Dateien löschen würden. Das ist absichtlich nicht implementiert. Solche Befehle kommen daher erst im nächsten Unterkapitel vor.

Die Tabulatorvervollständigung in der PowerShell-Konsole funktioniert bei Commandlets, wenn man das Verb und den Strich bereits eingegeben hat, z.B. `Export-` ↹. Auch Platzhalter kann man dabei verwenden. Die Eingabe `Get-?e*` ↹ liefert `Get-Help` ↹ `Get-Member` ↹ `Get-Service`. Andere Editoren wie das ISE bieten auch IntelliSense-Eingabeunterstützung für Commandlet-Namen an.

Commandlet-Parameter

Parameter

Durch Angabe eines Parameters können die Commandlets Informationen für die Befehlsausführung erhalten, z.B. ist bei `Get-Process` ein Filtern über den Prozessnamen möglich.

Durch

```
Get-Process  i*
```

werden nur diejenigen Prozesse angezeigt, deren Name auf das angegebene Muster (Name beginnt mit dem Buchstaben „i“) zutrifft:

Ein weiteres Beispiel für einen Befehl mit Parameter ist:

```
Get-ChildItem c:\daten
```

`Get-ChildItem` listet alle Unterobjekte des angegebenen Dateisystempfads (*c:\daten*) auf, also alle Dateien und Ordner unterhalb dieses Dateiordners.

Ein drittes Beispiel ist

```
Stop-Service BITS
```

Dieser Befehl führt eine Aktion aus: Der Windows-Hintergrundübertragungsdienst (BITS) wird angehalten.

Ein viertes Beispiel ist:

```
Remove-Item c:\temp\*.log
```

Dieser Befehl löscht alle Dateien mit der Dateinamenserweiterung „log“ aus dem Ordner c:\temp.

Anführungszeichen

Parameter werden als Zeichenkette aufgefasst – auch wenn sie nicht explizit in Anführungszeichen stehen. Die Anführungszeichen sind optional. Man muss Anführungszeichen um den Parameterwert nur dann verwenden, wenn Leerzeichen vorkommen, denn das Leerzeichen dient als Trennzeichen zwischen Parametern:

```
Get-ChildItem "C:\Program Files"
```

Parameterreihenfolge und -namen

Commandlets haben aber in der Regel nicht nur einen, sondern zahlreiche Parameter, die durch Position oder einen Parameternamen voneinander unterschieden werden. Ohne die Verwendung von Parameternamen werden vordefinierte Standardattribute belegt, d.h., die Reihenfolge ist entscheidend.

Beispiel: Auflisten von Dateien in einem Dateisystempfad, die eine bestimmt Datennamenserweiterung besitzen. Dies erfüllt der Befehl:

```
Get-ChildItem C:\temp *.doc
```

Wenn ein Commandlet mehrere Parameter besitzt, ist die Reihenfolge der Parameter entscheidend oder der Nutzer muss die Namen der Parameter mit angeben. Bei der Angabe von Parameternamen kann man die Reihenfolge der Parameter ändern:

```
Get-ChildItem -Filter *.doc -Path C:\temp
```

Alle folgenden Befehle sind daher gleichbedeutend:

```
Get-ChildItem C:\temp *.doc
Get-ChildItem -Path C:\temp -Filter *.doc
Get-ChildItem -Filter *.doc -Path C:\temp
```

Hingegen ist Folgendes falsch und funktioniert nicht wie gewünscht, weil die Parameter nicht benannt sind und die Reihenfolge falsch ist:

```
Get-ChildItem *.doc C:\temp
```

Diesen Versuch beantwortet die PowerShell mit einer Fehlermeldung („Das zweite Pfadfragment darf kein Laufwerk oder UNC-Name sein.“) in roter Schrift (siehe Abbildung).

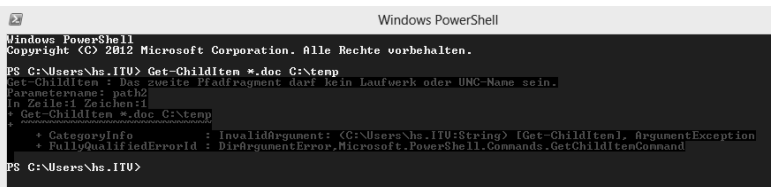


Abbildung 3.1 *Fehlermeldung bei falscher Parameterreihenfolge*

Schalter-Parameter

Schalter-Parameter (engl. Switch) sind Parameter, die keinen Wert haben. Durch die Verwendung des Parameternamens wird die Funktion aktiviert, z.B. das rekursive Durchlaufen durch einen Dateisystembaum mit `-recurse`:

```
Get-ChildItem h:\demo\PowerShell -recurse
```

Wenn man einen Schalter deaktivieren möchte, weil er im Standard aktiv ist oder weil man sehr explizit darauf hinweisen möchte, dass er nicht aktiv sein soll, kann man `$false` mit Doppelpunkt getrennt angeben, z.B.

```
Get-ChildItem h:\demo\PowerShell -recurse:$false
```

Berechnungen in Parametern

Parameter können berechnet, d.h. aus Teilzeichenketten zusammengesetzt sein, die mit einem Pluszeichen verbunden werden. (Dies macht insbesondere Sinn in Zusammenhang mit Variablen, die aber erst später in diesem Buch eingeführt werden.)

Der folgende Ausdruck führt jedoch nicht zum gewünschten Ergebnis, da auch hier das Trennzeichen vor und nach dem + ein Parametertrenner ist.

```
Get-ChildItem "c:\" + "Windows" *.dll -Recurse
```

Auch ohne die beiden Leerzeichen vor und nach dem + geht es nicht. In diesem Fall muss man durch eine runde Klammer dafür sorgen, dass die Berechnung erst ausgeführt wird:

```
Get-ChildItem ("c:\" + "Windows") *.dll -Recurse
```

Es folgt dazu noch ein Beispiel, bei dem Zahlen berechnet werden. Der folgende Befehl liefert den Prozess mit der ID 2900:

```
Get-Process -id (2800+100)
```

Weitere Beispiele

```
Get-Service -exclude "[k-z]*"
```

zeigt nur diejenigen Systemdienste an, deren Name nicht mit den Buchstaben „k“ bis „z“ beginnt.

Auch mehrere Parameter können der Einschränkung dienen. Der folgende Befehl liefert nur die Benutzereinträge aus einem bestimmten Active-Directory-Pfad. (Das Beispiel setzt die Installation der PSCX voraus.)

```
Get-ADObject -dis "LDAP://E02/ou=Geschäftsführung,
OU=www.IT-Visions.de,dc=IT-Visions,dc=local" -class user
```

Tabulatorvervollständigung klappt auch bei Parametern. Versuchen Sie einmal folgende Eingabe an der PowerShell-Konsole: `Get-ChildItem -` ⇥

An vielen Stellen sind Platzhalter bei den Parameterwerten erlaubt. **Platzhalter**

Eine Liste aller Prozesse, die mit einem „i“ anfangen, erhält man so:

```
Get-Process i*
```

Allgemeine Parameter

Es gibt einige Parameter, die in vielen (aber nicht allen) Commandlets vorkommen:

- `-Verbose`: Das Commandlet liefert eine detaillierte Bildschirmausgabe.
- `-Force`: Eine Aktion wird erzwungen, z.B. eine Datei wird mit Remove-Item gelöscht, obwohl die Datei einen Schreibschutz gesetzt hat. Ein weiteres Beispiel: `Remove-SmbShare` frag immer vor dem Löschen nach, wenn `-force` nicht gesetzt ist.
- `-Whatif` („Was wäre wenn“): Die Aktion wird nicht ausgeführt, es wird nur ausgegeben, was passieren würde, wenn man die Aktion ausführt. Das ist z.B. in einem Befehl mit Platzhaltern wie dem Folgenden sinnvoll, damit man weiß, welche Dienste nun gestoppt würden:
  ```
  get-service | where {$_.servicename -like "A*"}
  | foreach {stop-service $_.servicename -whatif}
  ```

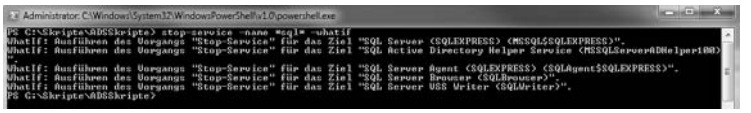


Abbildung 3.2 *Operationen mit Platzhaltern können schlimme Konsequenzen haben – whatif zeigt, welche Dienste betroffen wären.*

- `-Confirm`: Der Benutzer erhält eine Nachfrage für jede Änderungsaktion (siehe Bildschirmabbildung), z.B.
  ```
  get-service | where {$_.servicename -like "A*"}
  | foreach { stop-service $_.servicename -confirm }
  ```
  Innerhalb der Nachfrage kann der Benutzer in einen Suspend-Modus gehen, in dem er andere Befehle eingeben kann, z.B. um zu prüfen, ob er nun ja oder nein antworten will. Der Suspend-Modus wird mit drei Pfeilen >>> angezeigt und ist durch exit zu verlassen (siehe Bildschirmabbildung).

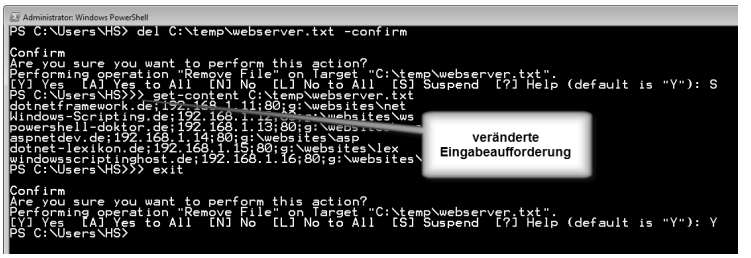


Abbildung 3.3 *Confirm und Suspend*

- `-ErrorAction` (abgekürzt `-ea`) und `–WarningAction` (`-wa`): Festlegung, wie ein Skript sich verhalten soll, wenn es auf einen Fehler trifft. Diese Parameter werden im *Kapitel 7 „PowerShell-Skriptsprache"* näher erklärt.

Leider gibt es bei den PowerShell-Commandlets, die gravierende Aktionen ausführen, einige Unterschiede im Grundverhalten und in der Verwendung der obigen Commandlets. Einige Commandlets führen im Standard die Aktion aus (z.B. `Remove-Item`). Andere Commandlets (z.B. `Remove-ADUser` und `Remove-SmbShare`) fragen immer nach vor dem Löschen. Das ist bei automatisierten Skripten natürlich unsinnig und daher gibt es auch eine Möglichkeit, diesen Commandlets das abzugewöhnen. Diese sieht jedoch oftmals verschieden aus. Bei `Remove-ADUser` muss man `–confirm:$false` als Parameter angeben; bei `Remove-SmbShare` ist es hingegen ein `-force`. Schade, dass Microsoft hier nicht einheitlich sein konnte.

Module

Import-Module

Schon seit PowerShell 2.0 sind die Commandlets und Funktionen in Modulen organisiert. Während der Benutzer in PowerShell 2.0 ein Modul noch explizit mit `Import-Module` aktivieren musste, bevor man die Befehle aus dem Modul nutzen konnte, erledigt dies die PowerShell 3.0 bei Bedarf automatisch (Module Auto-Loading). Sowohl Konsole als auch ISE zeigen alle verfügbaren Commandlets und Funktionen aller vorhandenen Module in der Vorschlagsliste und beim Aufruf von `Get-Command` bereits an. Der eigentliche Import des Moduls erfolgt dann beim ersten Aufruf eines Befehls aus einem Modul.

In PowerShell 3.0 sind auch alle Kernbefehle der PowerShell in Modulen organisiert, diese zeigt die folgende Tabelle.

Tabelle 3.1 *Die vier wichtigsten Module der PowerShell 3.0 mit Beispielen für Commandlets in diesem Modul*

Modul	Beispiele für Commandlets in diesem Modul
Microsoft.PowerShell.Diagnostics	Get-WinEvent, Get-Counter, Import-Counter, Export-Counter ...
Microsoft.PowerShell.Management	Add-Content, Clear-Content, Clear-ItemProperty, Join-Path, Get-Process, Get-Service ...
Microsoft.PowerShell.Security	Get-Acl, Set-Acl, Get-PfxCertificate, Get-Credential ...
Microsoft.PowerShell.Utility	Format-List, Format-Custom, Format-Table, Format-Wide, Where-Object ...

Prozessmodell

Die PowerShell erzeugt beim Start einen einzigen Prozess. In diesem Prozess laufen alle Commandlets. Dies ist ein Unterschied zum DOS-ähnlichen Windows-Kommandozeilenfenster, bei dem die ausführbaren Dateien (*.exe*) in eigenen Prozessen laufen.

> Mit STRG + C kann man einen laufenden Befehl abbrechen.

Namenskonventionen

Beachten Sie, dass bei den Commandlets das Substantiv im Singular steht, auch wenn eine Menge von Objekten abgerufen wird. Das Ergebnis muss nicht immer eine Objektmenge sein. Beispielsweise liefert

```
Get-Location
```

nur ein Objekt mit dem aktuellen Pfad.

Mit

```
Set-Location c:\windows
```

wechselt man den aktuellen Pfad. Diese Operation liefert gar kein Ergebnis.

> Die Groß- und Kleinschreibung der Commandlet-Namen und der Parameternamen ist irrelevant.

Verben

Gemäß der PowerShell-Konventionen soll es nur eine begrenzte Menge wiederkehrender Verben geben: `Get`, `Set`, `Add`, `New`, `Remove`, `Clear`, `Push`, `Pop`, `Write`, `Export`, `Select`, `Sort`, `Update`, `Start`, `Stop`, `Invoke` usw. Außer diesen Basisoperationen gibt es auch Ausgabekommandos wie `Out` und `Format`. Auch Bedingungen werden durch diese Syntax abgebildet (`Where-Object`).

3.2 Aliase

Namensersetzungen

Durch sogenannte Aliase kann die Eingabe von Commandlets verkürzt werden. So ist `ps` als Alias für `Get-Process` oder `help` für `Get-Help` vordefiniert. Statt `Get-Process i*` kann also auch geschrieben werden: `ps i*`.

Aliase auflisten

Durch `Get-Alias` (oder den entsprechenden Alias `aliases`) erhält man eine Liste aller vordefinierten Abkürzungen in Form von Instanzen der Klasse `System.Management.Automation.AliasInfo`.

Durch Angabe eines Namens bei `Get-Alias` erhält man die Bedeutung eines Alias:

```
Get-Alias pgs
```

Möchte man zu einem Commandlet alle Aliase wissen, muss man allerdings schreiben:

```
Get-Alias | Where-Object { $_.definition -eq "Get-Process " }
```

Dies erfordert schon den Einsatz einer Pipeline, die erst im nächsten Kapitel besprochen wird.

Tabelle 3.2 *Vordefinierte Aliase in der PowerShell 3.0*

Alias	Commandlet
%	ForEach-Object
?	Where-Object
ac	Add-Content
asnp	Add-PSSnapIn
cat	Get-Content
cd	Set-Location
chdir	Set-Location
clc	Clear-Content
clear	Clear-Host
clhy	Clear-History
cli	Clear-Item
clp	Clear-ItemProperty
cls	Clear-Host
clv	Clear-Variable
cnsn	Connect-PSSession
compare	Compare-Object
copy	Copy-Item
cp	Copy-Item
cpi	Copy-Item
cpp	Copy-ItemProperty
cvpa	Convert-Path
dbp	Disable-PSBreakpoint
del	Remove-Item
diff	Compare-Object

Alias	Commandlet
dir	Get-ChildItem
dnsn	Disconnect-PSSession
ebp	Enable-PSBreakpoint
echo	Write-Output
epal	Export-Alias
epcsv	Export-Csv
epsn	Export-PSSession
erase	Remove-Item
etsn	Enter-PSSession
exsn	Exit-PSSession
fc	Format-Custom
fl	Format-List
foreach	ForEach-Object
ft	Format-Table
fw	Format-Wide
gal	Get-Alias
gbp	Get-PSBreakpoint
gc	Get-Content
gci	Get-ChildItem
gcm	Get-Command
gcs	Get-PSCallStack
gdr	Get-PSDrive
ghy	Get-History
gi	Get-Item
gjb	Get-Job
gl	Get-Location
gm	Get-Member
gmo	Get-Module
gp	Get-ItemProperty
gps	Get-Process
group	Group-Object
gsn	Get-PSSession
gsnp	Get-PSSnapIn
gsv	Get-Service
gu	Get-Unique
gv	Get-Variable
gwmi	Get-WmiObject

Tabelle 3.2 *Vordefinierte Aliase in der PowerShell 3.0 (Forts.)*

Tabelle 3.2 *Vordefinierte Aliase in der PowerShell 3.0 (Forts.)*

Alias	Commandlet
h	Get-History
history	Get-History
icm	Invoke-Command
iex	Invoke-Expression
ihy	Invoke-History
ii	Invoke-Item
ipal	Import-Alias
ipcsv	Import-Csv
ipmo	Import-Module
ipsn	Import-PSSession
irm	Invoke-RestMethod
ise	PowerShell_ise.exe
iwmi	Invoke-WMIMethod
iwr	Invoke-WebRequest
kill	Stop-Process
lp	Out-Printer
ls	Get-ChildItem
man	help
md	mkdir
measure	Measure-Object
mi	Move-Item
mount	New-PSDrive
move	Move-Item
mp	Move-ItemProperty
mv	Move-Item
nal	New-Alias
ndr	New-PSDrive
ni	New-Item
nmo	New-Module
npssc	New-PSSessionConfigurationFile
nsn	New-PSSession
nv	New-Variable
ogv	Out-GridView
oh	Out-Host
popd	Pop-Location
ps	Get-Process
pushd	Push-Location

Alias	Commandlet
pwd	Get-Location
r	Invoke-History
rbp	Remove-PSBreakpoint
rcjb	Receive-Job
rcsn	Receive-PSSession
rd	Remove-Item
rdr	Remove-PSDrive
ren	Rename-Item
ri	Remove-Item
rjb	Remove-Job
rm	Remove-Item
rmdir	Remove-Item
rmo	Remove-Module
rni	Rename-Item
rnp	Rename-ItemProperty
rp	Remove-ItemProperty
rsn	Remove-PSSession
rsnp	Remove-PSSnapin
rujb	Resume-Job
rv	Remove-Variable
rvpa	Resolve-Path
rwmi	Remove-WMIObject
sajb	Start-Job
sal	Set-Alias
saps	Start-Process
sasv	Start-Service
sbp	Set-PSBreakpoint
sc	Set-Content
select	Select-Object
set	Set-Variable
shcm	Show-Command
si	Set-Item
sl	Set-Location
sleep	Start-Sleep
sls	Select-String
sort	Sort-Object
sp	Set-ItemProperty

Tabelle 3.2 *Vordefinierte Aliase in der PowerShell 3.0 (Forts.)*

Tabelle 3.2 Vordefinierte Aliase in der PowerShell 3.0 (Forts.)

Alias	Commandlet
spjb	Stop-Job
spps	Stop-Process
spsv	Stop-Service
start	Start-Process
sujb	Suspend-Job
sv	Set-Variable
swmi	Set-WMIInstance
tee	Tee-Object
trcm	Trace-Command
type	Get-Content
where	Where-Object
wjb	Wait-Job
write	Write-Output

Neue Aliase anlegen

Set-Alias, New-Aliase

Einen neuen Alias definiert der Nutzer mit `Set-Alias` oder `New-Alias`, z.B.:

```
Set-Alias procs Get-Process
New-Alias procs Get-Process
```

Der Unterschied zwischen `Set-Alias` und `New-Alias` ist marginal: `New-Alias` erstellt einen neuen Alias und liefert einen Fehler, wenn der zu vergebende Alias schon existiert. `Set-Alias` erstellt einen neuen Alias oder überschreibt einen Alias, wenn der zu vergebende Alias schon existiert. Mit dem Parameter `-description` kann man jeweils auch einen Beschreibungstext setzen.

Man kann einen Alias nicht nur für Commandlets, sondern auch für klassische Anwendungen vergeben, z.B.:

```
Set-Alias np notepad.exe
```

Beim Anlegen eines Alias wird nicht geprüft, ob das zugehörige Commandlet bzw. die Anwendung überhaupt existiert. Der Fehler würde erst beim Aufruf des Alias auftreten.

Man kann in Aliasdefinitionen keinen Parameter mit Werten vorbelegen. Möchten Sie zum Beispiel definieren, dass die Eingabe von „Temp" die Aktion „Get-ChildItem c:\Temp" ausführt, brauchen Sie dafür eine Funktion. Mit einem Alias geht das nicht.

```
Function Temp { Get-Childitem c:\temp }
```

Funktionen werden später (siehe *Kapitel 6 „PowerShell-Skripte"*) noch ausführlich besprochen. Die Windows PowerShell enthält zahlreiche vordefinierte Funktionen, z.B. `c:`, `d:`, `e:` sowie `mkdir` und `help`.

Die neu definierten Aliase gelten jeweils nur für die aktuelle Instanz der PowerShell-Konsole. Man kann die eigenen Alias-Definitionen exportieren mit `Export-Alias` und später wieder importieren mit `Import-Alias`. Als Speicherformate stehen das CSV-Format und das PowerShell-Skriptdateiformat (*.ps1* – siehe spätere Kapitel) zur Verfügung. Bei dem PS1-Format ist zum späteren Reimport der Datei das Skript mit dem Punktoperator (engl. „Dot Sourcing") aufzurufen.

	Dateiformat CSV	**Dateiformat .ps1**
Speichern	`Export-Alias c:\meinealias.csv`	`Export-Alias c:\meinealias.ps1 -as script`
Laden	`Import-Alias c:\meinealias.csv`	`. c:\meinealias.ps1`

Die Anzahl der Aliase ist im Standard auf 4096 beschränkt. Dies kann durch die Variable `$MaximumAliasCount` geändert werden.

Aliase für Eigenschaften

Aliase sind auch auf Ebene von Eigenschaften definiert. So kann man statt

```
Get-Process processname, workingset
```

auch schreiben:

```
Get-Process  name, ws
```

Diese Aliase der Attribute sind definiert in der Datei *types.ps1xml* im Installationsordner der PowerShell.

Abbildung 3.4
types.ps1xml

3.3 Ausdrücke

Mathematik Ebenfalls als Befehl direkt in die PowerShell eingeben kann man Ausdrücke, z.B. mathematische Ausdrücke wie

```
10* (8 + 6)
```

oder Zeichenkettenausdrücke wie

```
"Hello "+ " " + "World"
```

Microsoft spricht hier vom Expression Mode der PowerShell im Kontrast zum Command Mode, der verwendet wird, wenn man

```
Write-Output 10* (8 + 6)
```

aufruft.

Command Mode versus Expression Mode

Die PowerShell kennt zwei Verarbeitungsmodi für Befehle: einen Befehlsmodus (Command Mode) und einen Ausdrucksmodus (Expression Mode). Im Befehlsmodus werden alle Eingaben als Zeichenketten behandelt. Im Ausdrucksmodus werden Zahlen und Operationen verarbeitet. Als Faustregel gilt: Wenn eine Zeile mit einem Buchstaben oder den Sonderzeichen kaufmännisches Und (&), Punkt (.) oder Schrägstrich ("\") beginnt, dann ist die Zeile im Befehlsmodus. Wenn die Zeile mit einer Zahl, einem Anführungszeichen (" oder '), einer Klammer ("(") oder dem @-Zeichen („Klammeraffe“) beginnt, dann ist die Zeile im Ausdrucksmodus.

Befehls- und Ausdrucksmodus können gemischt werden. Dabei muss man in der Regel runde Klammern zur Abgrenzung verwenden. In einen Befehl kann ein Ausdruck durch Klammern eingebaut werden. Außerdem kann eine Pipeline mit einem Ausdruck beginnen. Die folgende Tabelle zeigt verschiedene Beispiele zur Erläuterung.

Tabelle 3.3 *Ausdrücke in der Windows PowerShell*

Beispiel	Bedeutung
`2+3`	Ein Ausdruck – die PowerShell führt die Berechnung aus und liefert 5.
`echo 2+3`	Ein reiner Befehl. „2+3“ wird als Zeichenkette angesehen und ohne Auswertung auf dem Bildschirm ausgegeben.
`echo (2+3)`	Ein Befehl mit integriertem Ausdruck. Auf dem Bildschirm erscheint 5.
`2+3 | echo`	Eine Pipeline, die mit einem Ausdruck beginnt. Auf dem Bildschirm erscheint 5.
`echo 2+3 | 7+6`	Eine unerlaubte Eingabe. Ausdrücke dürfen in der Pipeline nur als erstes Element auftauchen.
`$a = Get-Process`	Ein Ausdruck mit integriertem Befehl. Das Ergebnis wird einer Variablen zugewiesen.
`$a | Get-Process`	Eine Pipeline, die mit einem Ausdruck beginnt. Der Inhalt von `$a` wird als Parameter an `Get-Process` übergeben.
`Get-Process | $a`	Eine unerlaubte Eingabe. Ausdrücke dürfen in der Pipeline nur als erstes Element auftauchen.
"Anzahl der laufenden Prozesse: (Get-Process). Count"	Es ist wohl nicht das, was gewünscht ist, denn die Ausgabe ist: Anzahl der laufenden Prozesse: `(Get-Process).Count`
"Anzahl der laufenden Prozesse: $((Get-Process). Count)"	Jetzt ist die Ausgabe "Anzahl der laufenden Prozesse: 95", weil `$( ... )` einen Unterausdruck (Subexpression) einleitet und dafür sorgt, dass `Get-Process` ausgeführt wird.

3.4 Externe Befehle

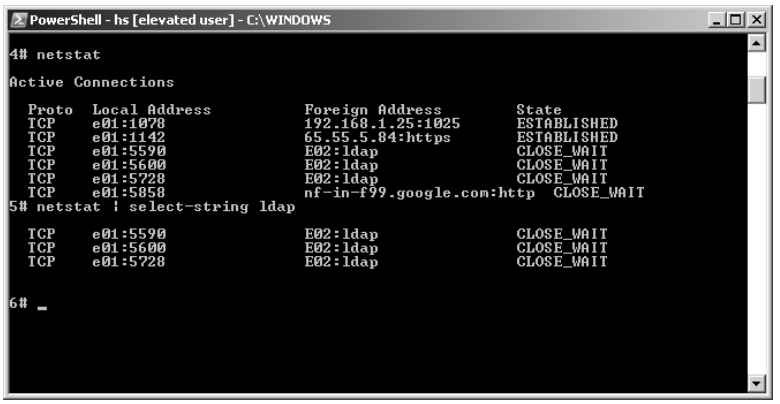
Windows-Anwendungen, DOS-Befehle, WSH-Skripte

Alle Eingaben, die nicht als Commandlets oder mathematische Formeln erkannt werden, werden als externe Anwendungen behandelt. Es können sowohl klassische Kommandozeilenbefehle (wie *ping.exe*, *ipconfig.exe* und *netstat.exe*) als auch Windows-Anwendungen ausgeführt werden.

Die Eingabe `c:\Windows\Notepad.exe` ist daher möglich, um den „beliebten" Windows-Editor zu starten. Auf gleiche Weise können auch WSH-Skripte aus der PowerShell heraus gestartet werden.

Die folgende Bildschirmabbildung zeigt den Aufruf von *netstat.exe*. Zuerst wird die Ausgabe nicht gefiltert. Im zweiten Beispiel kommt zusätzlich das Commandlet `Select-String` zum Einsatz, das nur die Zeilen ausgibt, die das Wort „LDAP" enthalten.

Abbildung 3.5
Ausführung von netstat



Wenn ein Leerzeichen im Pfad zu einer .exe-Datei vorkommt, dann kann man die Datei so nicht aufrufen (hier wird nach einem Befehl „T:\data\software\Windows" gesucht):

```
T:\data\software\Windows Tools\ImageEditor.exe
```

Auch die naheliegende Lösung der Verwendung von Anführungszeichen funktioniert nicht (hier wird die Zeichenkette ausgegeben):

```
"T:\data\software\Windows Tools\ImageEditor.exe"
```

Korrekt ist die Verwendung des kaufmännischen Und (`&`), das dafür sorgt, dass der Inhalt der Zeichenkette als Befehl betrachtet und ausgeführt wird:

```
& "T:\data\software\Windows Tools\ImageEditor.exe"
```

Grundsätzlich könnte es passieren, dass ein interner Befehl der PowerShell (Commandlet, Alias oder Function) genauso heißt wie ein externer Befehl. Die PowerShell warnt in einem solchen Fall nicht vor der Doppeldeutigkeit, sondern die Ausführung erfolgt nach folgender Präferenzliste:

- Aliase
- Funktionen
- Commandlets
- Externe Befehle

3.5 Dateinamen

Beim direkten Aufruf von Datendateien (z.B. `.doc`-Dateien) wird entsprechend den Windows-Einstellungen in der Registrierungsdatenbank die Standardanwendung gestartet und damit das Dokument geladen.

Dateinamen und Ordnerpfade müssen nur in Anführungszeichen (einfache oder doppelte) gesetzt werden, wenn sie Leerzeichen enthalten.

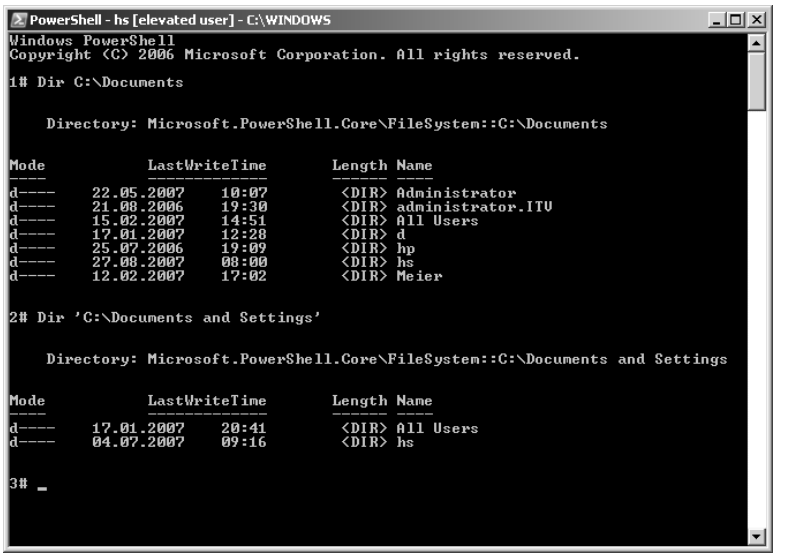


Abbildung 3.6
Anführungszeichen bei Pfadangaben

Copyright