

Windows PowerShell



*Eine Einführung in
Scripting Technologien für
Leute ohne echtes
Hintergrundwissen*

MICROSOFT SWITZERLAND

18 März 2007

Frank Koch (BERN)

Developer & Platform Evangelism

Windows PowerShell

Was kann man in einem solchen kurzen Werk erwarten Versuch eines Vorworts

Ziele beim Entwurf dieses Buchs

Im vorliegenden Buch „Windows Powershell“ finden Sie eine Einführung in die Windows PowerShell sowie Praxisbeispiele, um auch ohne umfangreiches Skript-Wissen schnell einen Einstieg in dieses Thema zu finden. Das Buch ist explizit nicht für die Profis unter den Skriptern gedacht; die umfangreiche Hilfe der Windows PowerShell sowie die inzwischen zahlreichen Internetforen und weitergehende Literatur bieten diesen Experten alles Nötige. Hingegen finden die Einsteiger unter Ihnen hier hoffentlich alles in einem Buch, um sich mit dem Thema Scripting einmal näher zu beschäftigen und hoffentlich schnell Spass an der Tatsache zu finden, den Computer auch ohne Maus bedienen zu können.

Das Buch bedient sich hierbei ausführlich bei vorhandenen Publikationen von Microsoft zur Windows PowerShell. Zielgruppengerecht stellt es diese neu zusammen, lässt vor allem die Theorie für den Anfang weg und bringt dafür viele Beispiele und kleine Aufgaben aus der Praxis, um den Spass an der Sache nicht zu kurz kommen zu lassen.

Wenn Sie diesen Text durchgearbeitet haben sollten und Sie die Windows PowerShell nun zu einem festen Bestandteil Ihres IT Alltags werden lassen wollen, lesen Sie daher ruhig auch die Originaldokumentation zur Windows PowerShell, welche sich bei der Installation von Windows PowerShell automatisch mit installiert:

- Erste Schritte mit Windows PowerShell
- Grundlagen von Windows PowerShell

Um von diesem Buch richtig profitieren zu können, sollten Sie den Zugang zu einem PC haben, an dem Sie die Übungen gleich selber ausprobieren können. Die einzige Voraussetzung ist ein PC mit installierter Windows PowerShell 1.0, welche ab Windows XP SP2 für Sie kostenlos zur Verfügung steht. Informationen zum Download und zur Installation finden Sie auf den folgenden Webseiten.

Weitere Informationsquellen im Internet

Die Einstiegsseite zur Windows PowerShell inklusive Download Link: www.microsoft.com/PowerShell
Hier finden Sie auch weitere Links zu sehr guten Webcasts, Büchern und weiteren Hilfeforen.

Die beste Blogseite zur Windows PowerShell ist <http://blogs.msdn.com/PowerShell/> Hier finden Sie alle Informationen zu Skripttechniken und praktischen Demos. Wirklich alles.

In der Schweiz finden Sie Informationen im ITPro Teamblog <http://blogs.technet.com/chITPro-DE>
Hier finden Sie auch Links zu deutschsprachigen Windows PowerShell Webcasts und Downloads zu den Übungen des Buchs im Archiv zum Monat März / April 2007.

Hilfreiche Tastaturbelegungen für die Schweizer Standardtastatur

ZEICHEN	TASTATURBELEGUNG	BEDEUTUNG
	ATGGR 7(NICHT: ALTGR1 = !)	AUSGABE EINES BEFEHLS WEITERLEITEN
`	SHIFT ^,DANACH LEERZEICHEN	BEFEHL AUF NÄCHSTER ZEILE FORTSETZEN
{	ALTGR Ä	BEGINN EINER BEFEHLSFOLGE (Z.B. NACH EINEM IF STATEMENT)
}	ALTGR \$	ENDE EINER BEFEHLSFOLGE (Z.B. BEIM IF STATEMENT)
[ALTGR Ü	TEILWEISE BEI OBJEKTEN NÖTIG
]	ALTGR !	TEILWEISE BEI OBJEKTEN NÖTIG
TAB	TABULATORASTE	VERVOLLSTÄNDIGT BEFEHLE, SOFERN NÖTIG BEISPIEL: GET-HE (TAB) ERGIBT GET-HELP

Die Windows PowerShell ist in Redmond entwickelt worden und ideal für das amerikanische Tastaturlayout angepasst. Einige häufig verwendete Tasten finden sich daher nur schwer auf der Schweizer Tastatur. Eine kleine Liste ist hier für Sie zusammengestellt.

Inhaltsverzeichnis

ZIELE BEIM ENTWURF DIESES BUCHS.....	2
WEITERE INFORMATIONSQUELLEN IM INTERNET.....	2
HILFREICHE TASTATURBELEGUNGEN FÜR DIE SCHWEIZER STANDARDTASTATUR	3
ERSTE EINDRÜCKE DER WINDOWS POWERSHELL.....	5
AUSGABEN WEITER VERWENDEN: “PIPING” VON OBJEKTEN.....	7
ERSTE ÜBUNGEN MIT WINDOWS POWERSHELL OBJEKTEN	8
DIE ARBEIT MIT PROZESSEN	8
DIE AUSGABE IN EINE TXT, CSV UND XML DATEI	9
AUSGABE IN FARBE	9
BEDINGUNGEN ÜBERPRÜFEN MIT DEM IF CMDLET	11
AUSGABE VON HTML	13
DIE ARBEIT MIT DATEIEN	15
INFORMATIONEN ZU OBJEKTEN MIT HILFE VON GET-MEMBER	16
LÖSCHEN VON DATEIEN	18
ANLEGEN VON ORDNERN	18
FALLS NOCH ZEIT IST.....	21
WINDOWS POWERSHELL ALS GENERISCHE OBJEKTVERARBEITUNGSMASCHINE	22
WMI OBJEKTE	22
ARBEITEN MIT .NET OBJEKTEN UND XML	24
ARBEITEN MIT COM OBJEKTEN	25
ARBEITEN MIT EVENTLOGS	28
LÖSUNGSSKRIPT ZU DEN AUFGABEN AUS DEM BUCH.....	29
WINDOWS POWERSHELL BEISPIELE – VON EINFACH BIS KOMPLEX	32
THEORETISCHE GRUNDLAGEN ZUR WINDOWS POWERSHELL.....	34
WINDOWS POWERSHELL – EINE KURZE EINFÜHRUNG.....	34
ZIELE BEIM ENTWURF VON WINDOWS POWERSHELL.....	34
VON TEXTEN, PARSERN UND OBJEKTEN.....	34
EINE NEUE SKRIPTSPRACHE	35
WINDOWS-BEFEHLE UND -DIENSTPROGRAMME	36
EINE INTERAKTIVE UMGEBUNG.....	36
SKRIPTUNTERSTÜTZUNG.....	36
CMD, WSCRIPT ODER POWERSHELL, MUSS ICH MICH ENTSCHEIDEN?.....	36
WINDOWS POWERSHELL 1.0.....	37
SICHERHEIT BEIM EINSETZEN VON SKRIPTEN	38

WINDOWS POWERSHELL IN DER PRAXIS

Windows PowerShell ist eine kostenlose Erweiterung für Windows XP Systeme und höher, welche unter <http://www.microsoft.com/powershell> von Microsoft heruntergeladen werden kann. Es setzt das .NET Framework 2.0 voraus, was - falls noch nicht installiert - einen eigenen Download und Installation verlangt. Windows PowerShell selber ist ein relativ kleiner Download von rund 1,5 MByte und kann problemlos auch automatisch im Hintergrund via Softwareverteilung installiert werden. Hierbei ist lediglich zu beachten, dass es für jede Windowsversion eine eigene Windows PowerShell Version gibt. Auch müssen Sie die jeweilige 32bit und 64bit Architektur berücksichtigen. Nach der Installation trägt sich Windows PowerShell ins Startmenü ein und steht ab dann durch den Aufruf der Verknüpfung oder der Eingabe von „PowerShell“ in der Befehlszeile von Windows zur Verfügung.

Erste Eindrücke der Windows PowerShell



Um einen ersten Eindruck zu sammeln starten Sie einmal Windows PowerShell und die klassischen Kommandozeile CMD aus dem Startmenü. Auf den ersten Blick sehen beide Shells - bis auf ihre charakteristische Farbe - sehr ähnlich aus:

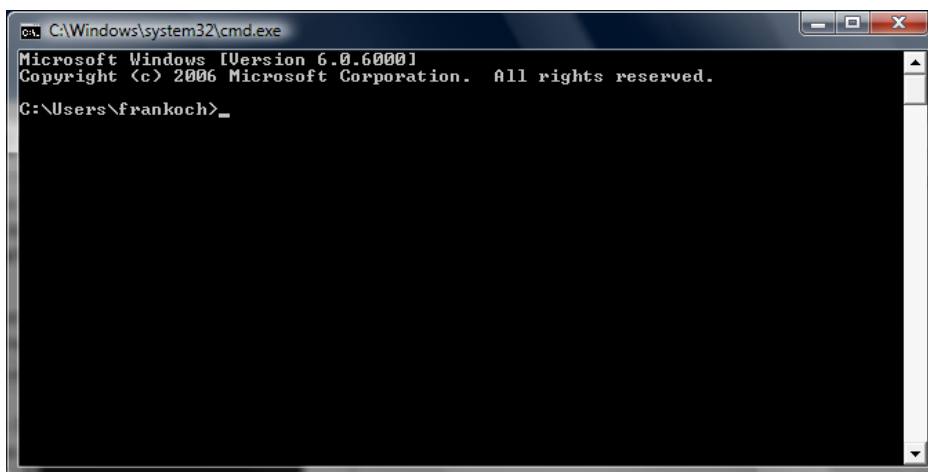


ABBILDUNG 1: DIE BEKANNTE KOMMANDOZEILE CMD

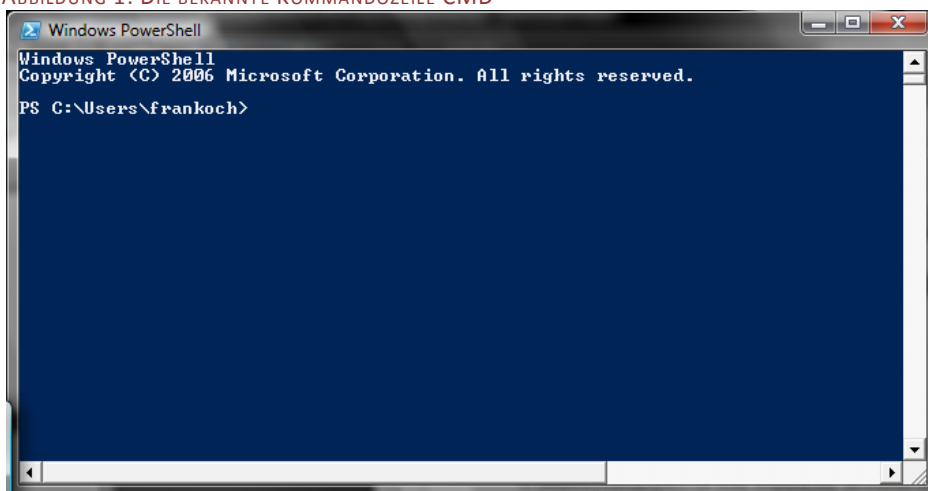


ABBILDUNG 2: DIE WINDOWS POWERSHELL

Und kein Wunder, nutzen doch beide Shells den gleichen „Eingabecontainer“. Das bedeutet leider auch, dass auch die Windows PowerShell unter der recht mässigen Unterstützung für Copy/Paste leidet wie schon die CMD seit Jahren. Etwas Erleichterung bringen die folgenden Tipps:

- mit der Maus den gewünschten Text markieren
- rechte Maustaste drücken (= kopieren)
- Cursor auf gewünschte Position bringen
- rechte Maustaste erneut drücken (= einfügen)



Versuchen Sie es selber einmal und kopieren Sie aus jeder Shell die erste Textzeile (wie z.B. „Copyright (c) 2006 Microsoft Corporation“) und fügen Sie diese gleich anschliessend wieder als „Befehlszeile“ ein. Stören Sie sich nicht an der Fehlermeldung, wenn Sie Enter drücken. Gewöhnen Sie sich an dieses Maustastenübungen, Sie werden es später noch oft brauchen.

Auch wenn der „Container“ gleich ist, so ist der Inhalt und Funktionsumfang bei beiden Shells sehr unterschiedlich. Am einfachsten findet man das heraus, wenn man sich die Online Hilfen anschaut. Ein erster Blick macht deutlich, dass Windows PowerShell ungleich viel mehr Funktionen bietet als die CMD; über 100 solcher Befehle, auch cmdlet (sprich „*Commandlet*“) genannt sind enthalten. Da in der CMD nur wenige Befehle direkt enthalten sind, kamen im Laufe der Zeit immer mehr Hilfsprogramme hinzu. Da jedes dieser CMD Programme seine eigene Syntax hat, muss ein erfahrender CMD Experte viele unterschiedliche Befehle und Befehlslogiken auswendig lernen. Für cmdlets ist die Syntax und Befehlslogik klar definiert; Windows PowerShell Befehle folgen einer gewissen Namensregel:

- Befehle in Windows PowerShell bestehen aus einem Verb und einem Substantiv (immer in Einzahl), getrennt durch einen Bindestrich. Befehle sind in Englisch gehalten. Ein Beispiel: *get-help* ist der Aufruf der Online Hilfe in Windows PowerShell Syntax
- Parameter werden mit einem vorgestellten – gekennzeichnet:
get-help -detailed
- Zur Vereinfachung hat man zusätzlich viele bekannte Befehle übernommen, um den Einstieg in Windows PowerShell leichter zu machen. Beispiel:
Neben *get-help* funktioniert auch *help* (Windows Klassiker) oder *man* (Unix Klassiker)



Lassen Sie sich die unterschiedlichen Hilfe-Texte der Shells einmal anzeigen. Geben Sie in jeder Shell den Befehl *Help* ein. Sie sehen die unterschiedliche Anzahl an dokumentierten Befehlen direkt in der jeweiligen Ausgabe.

Statt *help* oder *man* können Sie in Windows PowerShell auch *get-help* verwenden. Hierzu ist die Syntax wie folgt:

- *Get-help* gibt Hilfe zur Verwendung der Hilfe
- *Get-help ** listet alle Windows PowerShell Befehle auf
- *Get-help Befehl* listet die Hilfe zu dem jeweiligen Befehl auf
- *Get-help Befehl -detailed* listet detaillierte Hilfe mit Beispielen zum Befehl auf



Verwenden Sie den Hilfe-Befehl, um detaillierte Informationen zum Hilfe-Befehl zu bekommen: *get-help get-help -detailed*. Tipp: Verwenden Sie die TAB Taste, um Befehl automatisch zu vervollständigen. So vermeiden Sie Tippfehler.

Ausgaben weiter verwenden: "Piping" von Objekten

Wie schon erwähnt, ist die Windows PowerShell eine objektorientierte Shell. Das heisst, das Ein- und Ausgaben von Befehlen in der Regel Objekte sind. Da Menschen Objekte nicht lesen können, „übersetzt“ Windows PowerShell die Objekte für die Ausgabe auf dem Bildschirm in Text (Profis finden in der Hilfe zu Windows PowerShell sogar Befehle, um die Ausgabe an ihre Bedürfnisse hin anzupassen). Die Verbindung von Befehlen wird durch den sogenannten „Pipe“ Befehl dargestellt: |



Diese Verbindung können Sie nutzen, um Ihr eigenes Windows PowerShell Buch herzustellen: `Get-help * | get-help -detailed` macht das für uns: `get-help *` erzeugt uns die Liste der bekannten Befehle, die wir als Eingabe für den Befehl `get-help -detailed` verwenden. Die Ausgabe ist sehr umfangreich, Sie können Sie mit CTRL-C abbrechen.

Um das Ergebnis des eigenen „Hilfebuchs“ auch später verwenden zu können, lenken wir die Ausgabe am besten in eine Datei um, statt sie auf dem Bildschirm darzustellen. Windows PowerShell hat hierzu einen eigenen Befehl *Out-File*, bekannter ist vielleicht jedoch das Zeichen `>`.



Erstellen Sie nun Ihre „Buchdatei“ und geben Sie in der Windows PowerShell folgende Befehle ein: `Get-help * | get-help -detailed | out-file c:\Powershell-Hilfe.txt` oder auch `Get-help * | get-help -detailed > c:\PowerShell-Hilfe.txt`. Beachten Sie, dass Sie Schreibrechte auf dem Zielpfad (hier `c:\`) haben müssen.

Öffnen Sie anschliessend Ihre erstellte Hilfedatei in Notepad und verwenden Sie sie als OnlineHilfe für die späteren Übungen.

Sollten Sie später einmal nach einem Befehl suchen, so hilft einem auch hier der Befehl `get-help` weiter. Wollen Sie etwas sortieren, versuchen Sie einmal mit `get-help sort*` etwas Passendes zu finden. `Get-help` fängt nun an, im Befehlsschatz von der Windows PowerShell nach einem passenden Befehl zu suchen. Da alle Befehle mit einem Verb anfangen, können wir hier die Suche sehr einfach gestalten mit `get-help "passed english Verb"`. Falls Sie es nicht wissen sollten: das `*` Zeichen drückt dabei aus, dass nach dem eigentlichen Suchtext noch alles Mögliche kommen kann, wir es jedoch nicht wissen und daher mal alles zurückbekommen möchten, was damit anfängt. Solche Zeichen nennt man auch Wildcards.

Haben Sie einen Befehl angezeigt bekommen (hier wäre es wohl `sort-object` gewesen), so rufen Sie einfach anschliessend `get-help` erneut auf, diesmal jedoch mit dem einen Befehl und dem Parameter `-detailed`, um Beispiele für die Verwendung des Befehls zu bekommen:

`get-help sort-object -detailed.`

Mit dem Ergebnis sollten Sie in der Lage sein, Ihr Problem nun lösen zu können.

Erste Übungen mit Windows PowerShell Objekten

Wenn einem die Arbeit mit Objekten unbekannt ist, können folgende Übungen einem helfen, die umfangreichen Möglichkeiten dieser Welt besser zu verstehen. Objekte in der Programmierung sind dabei nichts Neues, jedoch gibt es im Scripting-Bereich kaum Vergleichbares. Bei mehr Interesse an der Arbeit mit Objekten findet sich umfangreiche Sekundärliteratur auch auf den MSDN Seiten von Microsoft unter <http://www.microsoft.com/switzerland/msdn/de/default.mspx> oder <http://msdn.microsoft.com>. Schauen wir uns die Arbeit mit Objekten doch am Beispiel des „process“-Objekts an. Sollte Ihnen „process“ kein Begriff sein, stellen Sie sich dies einfach als dasjenige vor, was Sie im Taskmanager auf Ihrem System aufgelistet bekommen. Bei mehr Interesse am Objekt „process“ hilft sonst auch wieder die MSDN Seite.

Die Arbeit mit Prozessen

Der Befehl *get-process* listet alle Prozesse auf Ihrem System auf. Diese Liste kann sehr lang sein. Um die Liste zu sortieren kann man ein weiteres cmdlet nutzen: *sort-object*. *Sort-object* kennt zwei Parameter *–descending* und *–ascending*. Letzteres ist der Standard und muss daher nicht explizit angegeben werden. Als Argument gibt man die Objekteigenschaften („property“ genannt) an, nach denen sortiert werden soll, z.B. die CPU Zeit.



A1: Ihre Aufgabe ist es nun, eine Liste aller Prozesse zu erzeugen und diese absteigend nach deren CPU Zeit zu sortieren. Alles was Sie dazu benötigen haben Sie bereits kennengelernt: *get-process*, *sort-object*, und die Pipe |. Hinweis: CPU ist kein Parameter für *sort-object*, sondern ein Argument, nachdem Sie sortieren wollen. Es bekommt also kein – Zeichen.

In der nächsten Übung wollen wir die Liste ein wenig einschränken, um sie handlicher zu machen. Hierzu verwenden wir den Befehl *select-object*. *Select-object* kennt mehrere Parameter (verwenden Sie *get-help* um diese herauszubekommen), wir brauchen aber nur *–first x* und *–last y*, um die ersten x oder die letzten y Objekte aus einer Liste zu erhalten, z.B. *select-object –first 5*. *Select-object* alleine funktioniert dabei nicht, sondern das cmdlet erwartet zwingend eine Eingabe, z.B. wieder aus einer Pipe.



A2: Erzeugen Sie die Liste der Top 10 Prozesse basierend auf deren CPU Zeit. Nehmen Sie dazu das Ergebnis von A1 und erweitern Sie es um den Befehl *select-object*. Für die optimale Lösung gibt es zwei Wege, je nachdem, wie die Liste sortiert werden soll. Versuchen Sie beide Wege zu finden. Hinweis: ein Weg verwendet den Parameter *–first*, der andere *–last*

Wir benutzen diese kleine Übung, um noch schnell Variablen einzuführen. Vereinfacht speichern Variablen alle möglichen Werte, dies können aber auch Objekte sein! Auch hier ist wieder auf Sekundärliteratur verwiesen, wenn man mehr zum Thema Variablen lernen möchte. Für uns ist es nur wichtig zu wissen, dass Variablen in PowerShell immer mit \$ anfangen müssen. Das Ergebnis der Aufgabe A2 kann man nun in einer Variablen speichern und so auf die Liste der Top 10 Prozesse zu diesem Zeitpunkt immer wieder zugreifen. Das erlaubt den Vergleich mit anderen Zeitpunkten, um vielleicht auszuwerten, was sich an einem System so ändert. Die Zuordnung ist dabei sehr einfach:

\$a = get-process | sort-object CPU -de...



A3: Weisen Sie der Variable \$P die verkürzte Prozessliste aus Aufgabe A2 zu. Hinweise: verwenden Sie die Cursortaste „Hoch“, um den letzten Befehl wieder hervorzurufen und bewegen Sie den Cursor mit *Home* zum Anfang der Zeile, um etwas hinzuzufügen.

```
PS C:\Users\frankoch> $P
```

Handles	NPM(K)	PM(K)	WS(K)	UM(M)	CPU(s)	Id	ProcessName
1774	60	91052	165624	459	2.956.31	7788	WINWORD
201	9	129796	128228	233	1.052.98	376	dwm
360	22	28120	6876	252	223.53	1724	POWERPNT
729	16	43664	29524	156	219.79	2628	sidebar
1036	37	51720	67436	278	154.19	776	explorer
2111	42	34624	44944	235	123.97	2588	communicator
891	19	18516	7332	196	43.91	8116	ONENOTE
188	6	5544	7992	81	33.56	2412	ipoint
732	14	65880	58956	241	18.02	6888	powershell
68	4	3340	3744	52	8.44	2440	TPwrMain

```
PS C:\Users\frankoch>
```

ABBILDUNG 3: AUSGABE DER VARIABLEN \$P

Die Ausgabe in eine TXT, CSV und XML Datei

Standardmässig gibt Windows PowerShell die Ergebnisse einer Befehlskette auf dem Bildschirm aus. Hierbei werden eventuelle Objekte in Text umgewandelt, damit Menschen Sie auch lesen können. Dazu dient der Befehl *Out-Host*. Da Windows PowerShell jedoch effizient sein will, wird dies automatisch unsichtbar hinzugefügt, wenn man es selber nicht macht. Statt *Out-host* gibt es noch Alternativen, schauen Sie selber nach mit *get-help out**.

Die Ausgabe in eine Textdatei kann man damit recht schnell erreichen: *out-file Dateiname* ist die Lösung. In vielen anderen Shells findet sich hierzu auch der Befehl *>* der daher auch in Windows PowerShell unterstützt wird. Neben der Ausgabe als Textdatei gibt es auch noch die Ausgabe als CSV Datei und als XML Datei. Wie *out-Host* sind es eigene cmdlets, die dies für Sie erledigen. Sie lauten *Export-CSV* und *Export-CliXML*, beide verlangen als Argument den Dateinamen. Und ja, Sie liegen richtig: wo ein Export da ein Import. Mit *Import-CSV* oder *Import-CliXML* können Sie auch wieder die Dateien einlesen.



A4: Nehmen Sie die Variable \$P aus A3 und speichern Sie deren Inhalt in eine Textdatei mit dem Namen „A4.txt“. Im nächsten Schritt speichern Sie den Inhalt von \$P in einer CSV Datei mit dem Namen „A4.CSV“ und schlussendlich das ganze nochmal in eine XML Datei „A4.XML“. Hinweis: der Befehl *>* ersetzt direkt die Pipe |, welche nur für echte cmdlets wie *Out-file*, *export-csv* etc nötig ist. Schauen Sie sich die Ergebnisse an, zur Not reicht dafür Notepad.

Ausgabe in Farbe

Manchmal möchte man Ausgaben betonen, um das Lesen einfacher zu machen. Dies kann man z.B. durch den Einsatz von Farbe erreichen. Der Befehl *Write-host* kennt dazu mehrere Parameter wie *-foregroundcolor* und *-backgroundcolor*. Was würde wohl folgendes bewirken:

Write-host „Rot auf Blau“ -foregroundcolor red -backgroundcolor blue

Sie ahnen es sicherlich. *Get-help write-host -detailed* listet Ihnen die möglichen Farben auf. Dazu gibt es auch vordefinierte Kombinationen: mit *write-warning „error“* erreichen Sie ebenfalls die nötige Aufmerksamkeit, versuchen Sie es einmal. Mit diesem Befehl könnte man nun alle Prozesse in Bunt ausgeben. Schöner wäre es jedoch, wenn man die Liste anhand von zusätzlichen Bedingungen einfärben könnte. Das wollen wir uns nun genauer anschauen. Dazu verwenden wir einfachheits- halber statt der Prozesse die Services Ihres Rechners. Sollten Sie nicht wissen, was Services sind, schlagen Sie dies bitte wieder in nach, z.B. auf den MSDN Seiten. Vereinfacht sind Services die Dinger, die Sie in der *Systemsteuerung / Verwaltungswerkzeuge / Services* aufgelistet bekommen. Das Schöne an ihnen ist, dass es sie im Zustand „running“ und „stopped“ gibt, was sich hervorragend für Farbausgaben nutzen lässt. Aber zunächst einmal schauen wir uns die Services mit dem cmdlet *get-service* an.



A5: Erzeugen Sie eine Liste aller Services und sortieren Sie diese nach deren Eigenschaft Status. Hinweis: Nehmen Sie die Lösung der Prozess-Sortierung nach CPU Zeit, verwenden Sie jedoch *get-service* und „status“ als Argument für *sort-object*.

Nun wollen wir die ganze Liste in roter Schrift ausgeben. *Write-host* sollte hier unser Freund sein. Leider funktioniert der Befehl *get-service | write-host -foregroundcolor red* nicht wie erhofft. *Write-host* ist leider nicht so nett wie die anderen cmdlets und nimmt eine Liste von Objekten, die dann richtig eingefärbt ausgegeben wird. Hierzu müsste *write-host* für jedes Objekt ja auch wissen, welche Eigenschaften des Objekts denn wie ausgegeben werden sollen. Wir müssen etwas *write-host* helfen. Als erstes arbeiten wir hierzu die Liste der Objekte Schritt für Schritt einzeln ab, was man mit einer sogenannten Schleife macht. Es gibt viele Schleifen, jede hat ihre Berechtigungen und Einsatzgebiete. Für unseren Zweck verwenden wir die *ForEach-Object* Schleife, welche durch eine Liste von Objekten geht und jedes Objekt einzeln abarbeitet. Innerhalb der Schleife spricht man das jeweilige Objekt unter dem Kürzel *\$_* an, eine „Willkür“ der PowerShell Entwickler. Eine Eigenschaft des Objekts wird entsprechend mit *\$_ .Name-der-property* angesprochen. Schauen wir uns das in einem Beispiel an:

```
Get-process | ForEach-Object { write-host $_.ProcessName $_.CPU }
```

Während *get-process* uns die ganze Liste der Prozesse ausgibt, sehen wir auf dem Beispiel nur noch den Namen und die CPU Zeit, da die erste Ausgabe an die Pipe (|) geht und danach *write-host* für jedes Objekt nur noch diese zwei Eigenschaften ausgibt. (Sollten Sie das Beispiel aufrufen, wundern Sie sich nicht: nicht jeder Prozess hat eine CPU Zeit, so dass unter Umständen eine Zeile auch nur aus dem Namen bestehen kann).



A6: Erzeugen Sie eine Liste der Services und geben Sie nur die Eigenschaften Name und Status aus. Verwenden Sie hierzu die grade beschriebene *ForEach* Schleife, auch wenn es andere Lösungen gibt.

Nun können wir die Möglichkeiten von *write-host* nutzen um die Ausgabe in Farbe durchzuführen.



A7: Erzeugen Sie eine Liste der Services und geben Sie nur die Eigenschaften Name und Status in einer Farbkombination Ihrer Wahl aus. Zeigen Sie die Farben Ihrem Nachbarn und vergleichen Sie, wer die schönere Kombination gewählt hat. Hinweis: Nehmen Sie Ihre Lösung von A6 und ergänzen Sie die Parameter *-foregroundcolor -backgroundcolor*.

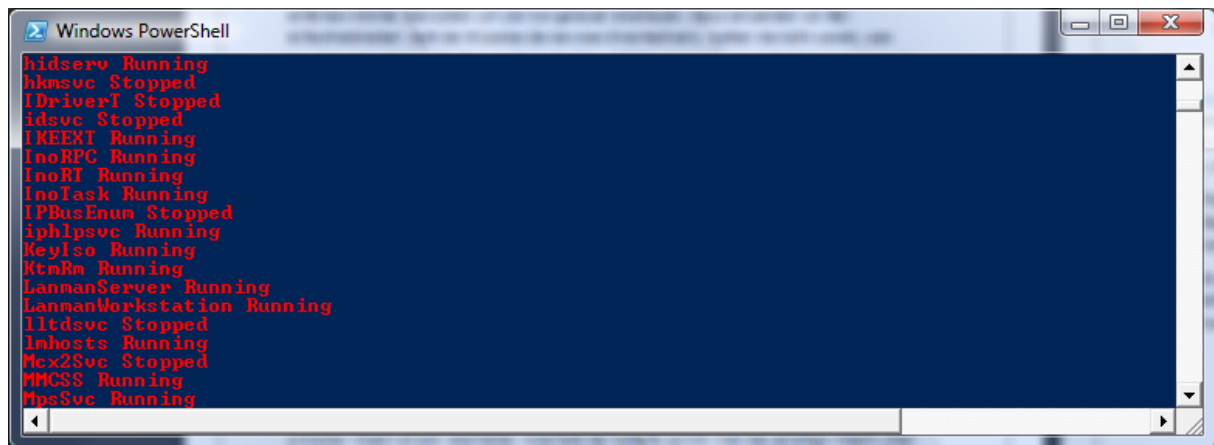


ABBILDUNG 4: AUSGABE VON SERVICENAMEN UND STATUS IN ROT

Bedingungen überprüfen mit dem *if* cmdlet

Als letztes fehlt uns jetzt nur noch, Bedingungen zu überprüfen. Auch hier gibt es viele Möglichkeiten für viele Einsatzszenarien. In dieser Einführung beschränken wir uns jedoch zunächst auf den IF Befehl. Die Syntax ist wahrscheinlich schon von anderen Stellen bekannt und ist sehr einfach:

```
If (Bedingung) {auszuführende Befehle}
elseif (Bedingung2) {auszuführende Befehle}
else{auszuführende Befehle}
```

Elseif ist dabei optional und nicht immer nötig. Sollen mehrere Befehle im {} Bereich ausgeführt werden, so kann man sie entweder mit dem Semikolon trennen oder pro Befehl eine neue Zeile benutzen. Windows PowerShell wartet einfach auf die } Klammer als Abschluss.

Zum Vergleichen kennt Windows PowerShell mehrere Vergleichsoperatoren. Sie fangen alle mit „-“ an und bestehen in der Regel aus einer englischen Abkürzung mit 2 Buchstaben: -eq steht z.B. für equals (gleich). Am wichtigsten für uns sind

-eq	Gleich
-match	„Entspricht“ (nicht ganz so streng wie gleich)
-ne	Ungleich
-notmatch	„entspricht nicht“
-gt -ge	Grösser als / Grösser oder gleich
-lt -le	Kleiner als / Kleiner oder gleich

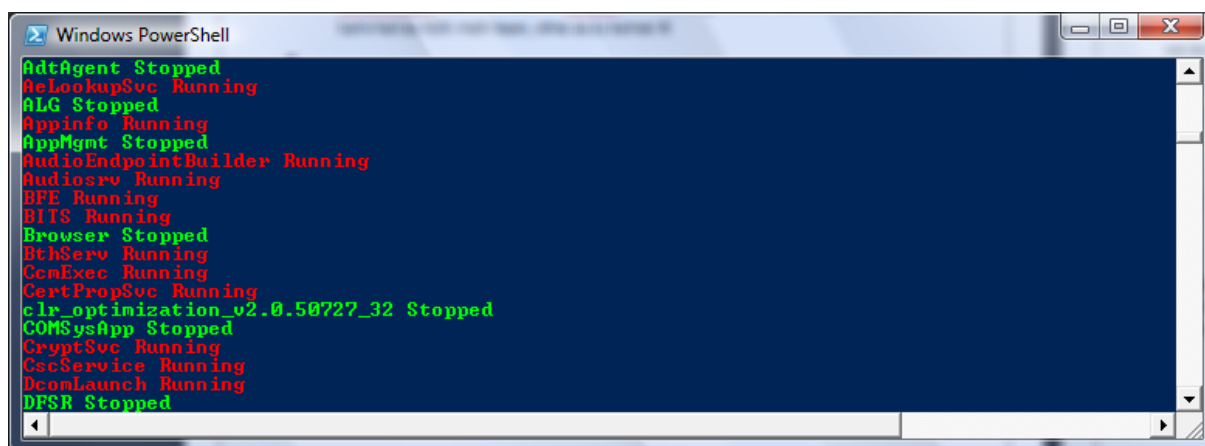
Die letzte Übung für diese Block kombiniert nun alle Punkte zu einer ersten Statusüberwachung eines Systems mit Powershell. Alle Services eines Systems werden zunächst sortiert nach ihrem Status und anschliessend in Farbe ausgegeben: Services mit dem Status „stopped“ in rot, Services mit dem Status „running“ in grün.



A8: Rufen Sie eine Liste aller Services auf. Sortieren Sie diese Liste nach dem Status und färben Sie die Ausgabe entweder in Rot oder Grün, je nachdem, ob der Status des jeweiligen Service „stopped“ oder „running“ ist. Hinweis: Verwenden Sie zunächst *sort-object* von der vorherigen Übungen. Verwenden Sie dann die *Foreach* Schleife, aber statt nur *write-host* zu nehmen, bauen Sie eine *If* Abfrage ein. Den Status eines Service bekommen Sie wie gewohnt mit `$_status`, die möglichen Werte sind „stopped“ oder „running“. Zur Syntax: Die *If* Bedingung kommen in runde Klammern (), der Ausgabebefehl in {} Klammern. Ignorieren Sie die theoretischen Möglichkeiten und verzichten Sie einfachheitshalber auf die *Elseif* Abfrage. Vergessen Sie nicht die Schlussklammer } von *ForEach*! Wenn Sie am Ende in einer >> Zeile stehen, schliessen Sie diese mit 2x Return ab, um die umgebrochenen Zeilen auszuführen.

Führen Sie die Lösung noch einmal aus, diesmal ohne das cmdlet *sort-object*. Verwenden Sie hierzu die auch die Cursortasten um nicht alles noch einmal eingeben zu müssen.

Foreach-Object kann man auch mit *ForEach* abkürzen. Es geht auch noch kürzer, aber dann kann man es nicht mehr lesen, ohne das Abkürzungszeichen zu kennen. Daher bleiben wir zunächst bei *ForEach-Object* oder *ForEach*.



```
Windows PowerShell
AdtAgent Stopped
AeLookupSvc Running
ALG Stopped
Appinfo Running
AppMgmt Stopped
AudioEndpointBuilder Running
Audiosrv Running
BFE Running
BITS Running
Browser Stopped
BthServ Running
CcmExec Running
CertPropSvc Running
clr_optimization_v2.0.50727_32 Stopped
COMSysApp Stopped
CryptSvc Running
CscService Running
DcomLaunch Running
DFSR Stopped
```

ABBILDUNG 5: FARBIGE AUSGABE DER SERVICES

Ausgabe von HTML

Zur Überwachung von Servern kann das Beispiel A8 schon etwas helfen. Schön wäre es nun, wenn die Ausgabe leichter weiterverwendet werden kann. Die Ausgabe als CSV und XML haben Sie schon kennengelernt. Es gibt jedoch noch eine weitere Ausgabe, die manchmal sinnvoll sein kann: die Ausgabe als HTML. Das cmdlet *convertto-html* ist dafür zuständig. Die Ausgabe erfolgt dabei nicht direkt in eine Datei sondern wie bei den anderen cmdlets auch, so dass man direkt mit einer Pipe den HTML Text noch bearbeiten kann. Am Ende sollte man den Text dann in eine Datei umlenken, damit man sie z.B. mit einem Webbrowser leichter betrachten kann. Anhand einer Reihe von kleinen Beispielen werden wir einige Möglichkeiten von *convertto-html* ausloten.



A9 Konvertieren Sie die Ausgabe von *get-service* in HTML Verwenden Sie dazu das cmdlet *convertto-html*, welches direkt mit einer List von Objekten arbeiten kann. Hinweis: die umfangreiche Ausgabe können Sie mit CTRL-C abbrechen.



A10 Leiten Sie diesmal die Ausgabe am Ende mit den bekannten Befehlen in die Datei „\A10.html“ um. Betrachten Sie die Datei. Hinweis: Sie können mit *invoke-item .\a10.html* direkt aus Powershell Ihren default-Browser mit der Ausgabedatei starten. Vergessen Sie jedoch nicht den korrekten Pfad zu A10.html mit einzugeben! Wenn es Ihnen lieber ist, öffnen Sie die Datei mit dem Dateexplorer.

ConvertTo-HTML erlaubt die Einschränkung der Ausgabe, so dass die Liste nicht zu unübersichtlich wird. Hierzu wird *ConvertTo-HTML* einfach mit der Liste der auszugebenden Objekteigenschaften aufgerufen, z.B. ... | *convertto-html -property name, status* .



A11: Aufbauend auf A10: Erzeugen Sie eine schönere Webseite und listen Sie lediglich den Namen und den Status aller Services auf. Sie können die Ausgabe auch vor der Konvertierung nach dem Status sortieren. Hinweis: Ihre Befehlszeile besteht damit aus 4 Befehlen:
Liste alle Services, sortiere nach Status, konvertieren in HTML, Ausgabe in eine Datei.

Da *ConvertTo-HTML* einen HTML Text erzeugt, kann dieser mittels HTML Kenntnissen angepasst werden. Dies ist dann nicht mehr Aufgabe von Windows PowerShell, kann hiermit jedoch unterstützt werden. Versuchen Sie den folgenden Code zu verstehen, bevor Sie ihn in die Windows PowerShell kopieren oder abtippen und ausführen lassen:

```
get-service | ConvertTo-Html -Property Name,Status | foreach {  
    if ($_ -like "*<td>Running</td>*") {$_ -replace "<tr>", "<tr bgcolor=green>"}  
    else {$_ -replace "<tr>", "<tr bgcolor=red>"}} > .\get-service.html
```

Die Ausgabedatei sollte etwa so aussehen wie unten dargestellt. Prinzipiell funktioniert das Beispiel wie die write-host Ausgabe, jedoch werden hier die einzelnen Zeilen der HTML Datei umgeschrieben: dem HTML Befehl für die Tabellenspalte wird die Hintergrundfarbe bgcolor Green oder bgcolor red angefügt. Da nicht jeder HTML kann, wurde das Beispiel ausnahmsweise mit dem Lösungscode direkt angegeben.

Name	Status
AdtAgent	Stopped
AeLookupSvc	Running
ALG	Stopped
Appinfo	Running
AppMgmt	Stopped
AudioEndpointBuilder	Running
Audiosrv	Running
BFE	Running
BITS	Running
Browser	Stopped
BthServ	Running
CcmExec	Running
CertPropSvc	Running
clr_optimization_v2.0.50727_32	Stopped
COMSysApp	Stopped

ABBILDUNG 6: FARBIGE HTML AUSGABE DURCH HTML MANIPULATION

Die Arbeit mit Dateien

In den nächsten Übungen werden wir mit Dateien arbeiten. Wenn Sie dieses Buch im Rahmen eines Workshops von mir bekommen haben, so melden Sie sich jetzt beim Instrukteur, um einen Memory-Stick mit einigen Testdateien zu erhalten. Wenn Sie die Übungen alleine durchführen, so legen Sie sich selber einen solchen Übungsordner an. Hierfür kopieren sie einige unterschiedliche Dateien (vielleicht um die 40) in einen Ordner. Wenn Sie nichts anderes finden, können Sie auch Dateien aus Ihrem Internetcache nehmen. Stellen Sie lediglich sicher, dass Sie mindestens zwei unterschiedliche Dateitypen nehmen, je mehr desto besser.

Arbeiten mit Dateien in Windows PowerShell ist sehr einfach. Gängige Befehle wie `DIR` oder `ls` können direkt verwendet werden. Lediglich beim `cd` muss man auf die zwingende Lücke danach achten: statt `cd..` muss es nun `cd ..` heissen!

Windows PowerShell fasst auch bei Dateien alles als Objekte auf. Die Grösse einer Datei kann so direkt abgefragt werden und muss nicht aus einem String herausgeschnitten werden. Zusätzlich kennt Windows PowerShell nicht nur das „klassische“ Dateisystem. Mittels dem cmdlet *get-psdrive* können Sie sich alle momentanen Lokationen anzeigen lassen, die Windows Powershell Ihnen direkt zum Zugriff anbietet. Laufwerke werden dabei wie gewohnt am Ende mit dem Doppelpunkt (:) angesprochen.



Listen Sie alle Windows PowerShell Laufwerke auf. Wechseln Sie (`cd`) zu dem Laufwerk `HKLM:`. Geben Sie *cd software* ein. Geben Sie *dir* ein. Wo befinden Sie sich gerade? Wechseln Sie ins ENV: „Laufwerk“. Listen Sie den Inhalt mittels *ls* auf, als wäre es ein klassischer Unix Ordner. Wechseln Sie als letztes in das CERT: Laufwerk und geben Sie auch hier den Inhalt mit dem cmdlet *get-childitem* an.

Sie sehen, in der Windows PowerShell ist nahezu kein Platz Ihres Rechners vor Ihnen sicher. Die verwendeten Befehle *dir*, *ls* und *get-childitem* sind dabei überall gleichberechtigt, Sie können daher verwenden, was Ihnen am liebsten ist. Um die Windows PowerShell Syntax zu wahren, werde ich vor allem mit *get-childitem* arbeite. Noch kurz etwas zur Registry. Wenn Sie ein wenig in der Registry umhergewandert sind, so werden Sie vielleicht festgestellt haben, dass Sie mit *dir* und *cd* zwar gut zu Recht kommen, die eigentlichen Registry-Werte jedoch nicht zu Gesicht bekommen. Das liegt daran, dass die Registry Werte eine Eigenschaft (Property) des Registry-Objekts sind, so wie Grösse oder letztes Zugriffsdatum die Eigenschaften einer Datei. Um die Werte der Registry zu entlocken, benötigen Sie den Befehl *get-ItemProperty*. Dieser Befehl listet Ihnen nun alle Registry-Werte wie gewünscht auf. Mehr dazu wie immer in der Hilfe der Windows PowerShell.

Um die Arbeit mit den Testdateien einfacher zu machen, erzeugen wir ein neues „Laufwerk“ in der Windows PowerShell, welches auf den eigentlichen Ordner zeigt. Hierzu brauchen wir den Befehl *new-psdrive*.



Erzeugen Sie ein neues Laufwerk durch Eingabe des Befehls wie folgt. Passen Sie am Ende den Pfad auf Ihren Ordner mit den Übungsdateien an:

New-PSdrive -name FK -psprovider FileSystem -root c:\pfad-auf-Ordner

Wechseln Sie dann mit *cd FK*: in das Verzeichnis und überprüfen Sie, dass Sie im richtigen Ordner sind. Lagen Sie falsch, können sie mit *Remove-PSDrive FK* das Laufwerk wieder entfernen und es erneut versuchen.

Die folgenden Übungen sollen Ihnen wieder die prinzipiellen Möglichkeiten der Windows PowerShell zeigen. Sie können auch hier nur an der Oberfläche kratzen, zeigen aber schön, was man prinzipiell machen kann. Passen Sie zur Not die jeweiligen Dateiendungen aus den Beispielen Ihren eigenen Übungsdateien an.



Wechseln Sie zum Übungslaufwerk: `cd fk:` (In PowerShell Syntax wäre dies übrigens `set-location fk:`) Listen Sie den Inhalt auf mit `get-childitem`. Blenden Sie temporäre Dateien aus: `get-childitem * -exclude *.tmp, *.temp`



B1: Geben Sie ausschliesslich den Dateinamen (name) und Dateilänge (length) aus, jedoch keine temporären Dateien mit der Endung temp oder tmp. Hinweis: Verwenden Sie die gleiche Technik wie bei der Ausgabe der Prozesse und Services.

```

Name                                                    Length
-----
Annual Meeting - Finance.ppt                          108032
Annual Meeting - Keynote.ppt                          65024
Annual Meeting - Travel.ppt                           43520
Annual Revenue Report.xls                             32768
arrow 0.png                                             1806
arrow 1.png                                             1946
arrow 2.png                                             2100
arrow 3.png                                             2371
arrow 4.png                                             2604
arrow 5.png                                             2756
arrow 6.png                                             3088
arrow 7.png                                             3303
AW Administration Best Practices.doc                  192512
AW Administrative Training.ppt                       107008
AW Reviewing Training.ppt                            127488
Book1.xlsx                                             8835
Corporate Management Guidelines.doc                   192000
curved arrow 1.png                                    3163
curved arrow 2.png                                    4163
curved arrow 3.png                                    5822
curved arrow 4.png                                    7277
curved arrow 5.png                                    14915
curved arrows circle cycle 2 down.png                11208
curved arrows circle cycle 2 up.png                  11297
curved arrows circle cycle.png                       20173
curved arrows down circle cycle 2 faded.png         11961
curved arrows up circle cycle 2 faded.png            12142
Customer Base.xls                                    29696
double headed arrow 0b faded.png                    3835
double headed arrow 0b.png                           3647
double headed arrow 1 faded.png                      4161
double headed arrow 1.png                             3885
double headed arrow 1b faded.png                     4697
double headed arrow 1b.png                           4392
  
```

ABBILDUNG 7: AUSGABE VON DATEINAME & LÄNGE OHNE TMP DATEIEN

Um den Tippaufwand zu minimieren, bietet Windows PowerShell verschiedene Abkürzungsmöglichkeiten. Rufen Sie hierzu `get-alias / sort-object definition` auf und Sie erhalten eine Liste aller denkbaren Schreibformen und „Dialekte“ für Befehle. Zusätzlich können für Parameter nur so viele Buchstaben verwendet werden, bis eindeutig ein Parameter festgelegt ist. Aus

`Get-childitem * -exclude *.tmp / select-object name, length`

wird so `ls* -ex *.tmp / select n*, le*`



B2: Sortieren Sie die Dateien aufsteigend nach ihrer Grösse (length), dann nach ihrem Namen. Hinweis: Verwenden Sie die gleiche Vorgehensweise wie bei den Prozessen.

Informationen zu Objekten mit Hilfe von `get-member`

Mit dem Befehl `get-member` erhält man eine Übersicht über alle Objekteigenschaften und Funktionen. Hierzu übergibt man ein Objekt via der Pipe an `get-member`. Man kann auch eine Liste von gleichen Objekten übergeben, ohne dass `get-member` Probleme bekommt.



B3: Erstellen Sie eine Liste aller möglichen Eigenschaften einer Datei mit Hilfe des cmdlet *get-member*. Sortieren Sie danach die Dateien nach dem letzten Zugriffsdatum. Hinweis: Verwenden Sie das Ergebnis der *get-member* Funktion und „raten“ Sie die passende Eigenschaft aus der Liste der Properties.

Der Befehl *Group-Object* kann eine Liste von Objekten in Gruppen einteilen. Hierfür muss als Argument eines der Objekteigenschaften verwendet werden. *Get-service / group-object status* generiert somit eine neue Liste mit 2 (oder mehr) Einträgen. Netterweise wird auch gleich die Anzahl der Services im jeweiligen Status angezeigt:

```
PS FK:\PSHDownload\Dateien> get-service | group-object status
```

Count	Name	Group
58	Stopped	{AdtAgent, ALG, AppMgmt, Browser...}
95	Running	{AeLookupSvc, Appinfo, AudioEndpointBuilder, Audiosrv...}

```
PS FK:\PSHDownload\Dateien>
```

ABBILDUNG 8: ERGEBNIS DES GROUP-OBJECT CMDLET



B4: Gruppieren Sie die Ausgabe Ihrer Dateien nach Ihren Dateieendungen (Extension). Sortieren Sie anschliessend auch diese Ausgabe anhand der Anzahl der Dateien mit der jeweiligen Endung. Hinweis: Geben Sie die Dateien aus, gruppieren Sie, dann sortieren Sie die neue Ausgabenliste nach der Anzahl. Hierfür verwenden Sie als Argument *count*.

Neben *get-member* gibt es noch ein weiteres sinnvolles cmdlet für die Informationsbeschaffung von Objekten: *measure-object*. Auch wenn wir die umfangreichen Möglichkeiten von *measure-object* hier nicht ausloten können, so können wir zumindest an einigen Beispielen dessen Möglichkeiten erahnen. Versuchen Sie einmal das Ergebnis von folgender Befehlskette zu verstehen:

get-childitem | measure-object length -average -sum -maximum -minimum

Wahrscheinlich verstehen Sie die Ausgabe bereits nach einigem Hinschauen. Das Ganze funktioniert auch mit Wildcards und den normalen Windows PowerShell cmdlet Pipes.



B5: Ermitteln Sie die Gesamtgrösse aller TMP Dateien. In einer 2. Aufgabe geben Sie diesmal NUR die Gesamtgrösse aus. Hinweis: Nach Ihrem ersten Versuch nehmen Sie die cmdlets und klammern Sie diese mit () ein. Nach der Ausführung wiederholen Sie die Befehle mit dem Zusatzbefehl *get-member* um alle Eigenschaften der Ausgabe sich anzeigen zu lassen (Sie erinnern sich? Windows PowerShell arbeitet mit Objekten und wandelt sie nur für uns zur Bildschirmausgabe in Texte um!). Finden Sie die Property Ihrer Klammerausgabe (), welche wohl der Eigenschaft Summe entsprechen könnte. Erinnern Sie sich an die ForEach-Schleife und wie dort die die Property „Status“ angesprochen wurde? Genau: *object.status*. Hier wollen wir aber die Summe statt dem Status.

Löschen von Dateien

Zum Löschen einer Datei bringt Windows PowerShell ebenfalls alle nötigen Befehle mit. Durch das cmdlet *Remove-item* können Sie sich nicht nur von Dateien entledigen. Es arbeitet analog zu *get-childitem*. Nun wäre es eine gute Idee, zunächst eine Sicherungskopie Ihres Übungsordners zu machen. Sollten Sie ausversehen zu viel löschen, können Sie so jederzeit wieder von vorne anfangen.



B6: Löschen Sie alle TMP Dateien mittels *remove-item*.

Manchmal möchte man auch Dateien löschen, die vielleicht grösser als ein Schwellwert sind. Hierbei hilft einem das cmdlet *where-object*. Wie der *IF* Befehl kann man eine Bedingung festlegen, die Objekte aus einer Liste erfüllen müssen, um ausgewählt zu werden. Schauen wir uns das Beispiel der Services an. Mittels

```
Get-service | where-object {$_.status -eq „stopped“}
```

erhalten wir nur die gestoppten Services.



B7: Löschen Sie nun alle Dateien, welche grösser als 2 MB sind. 2 MB entsprechen dabei vereinfacht 2000000 Byte. Hinweis: tasten Sie sich an das finale Skript heran. Erstellen Sie erst die Liste aller Dateien und schränken Sie diese nach Grösse (*...length -gt 2000000*) ein. Sie erhalten so eine neue Liste, welche Sie mit einer neuen Schleife abarbeiten sollten um nur den Dateinamen (*\$_fullname*) auszugeben. Diesen Namen brauchen Sie bei der Verwendung im finalen Durchlauf für *remove-item*. Sie können zwischendurch auch mit Variablen arbeiten, wenn Sie Ihre Eingabezeilen nicht unnötig lang werden lassen wollen!

Die Grösse 2 MB müssen Sie übrigens nicht unbedingt in 2000000 angeben (was ja eh nur eine grobe Näherung ist). Besser wäre die Nutzung direkt von 2MB als Grössenangabe, was Windows PowerShell problemlos akzeptiert. Sie können sich sogar ausrechnen lassen, was 512KB + 512KB sind. Für Rechenaufgaben brauchen Sie keine speziellen cmdlets, Sie geben die Aufgabe direkt ein.

Anlegen von Ordnern

Versuchen wir nun, etwas Ordnung in das Dateichaos zu bringen. Dazu werden wir für jeden Dateityp einen eigenen Unterordner erstellen und die passenden Dateien anschliessend dahin verschieben. Hierzu brauchen wir das cmdlet zur Erstellung eines neuen „Items“¹ im Dateisystem: *new-item*. Es nimmt als Argument den Namen an und als Parameter den Typ, z.B. *directory* für ein Verzeichnis. Das Anlegen eines neuen Verzeichnisses „Test“ kann man damit so durchführen:

```
New-item .\test -type directory
```

Um Ihnen das Leben noch einfacher zu machen, schauen wir uns noch einmal den Sort Befehl an: *Get-Service | Sort-Object status* kennen Sie, versuchen Sie aber mal

```
get-service | sort-object status -unique
```

¹ Die Dateisysteme, wie FAT oder NTFS sind keine echten Objektorientierten Systeme. Daher verwenden wir den Befehl *new-item* statt *new-object*. In Zukunft mag das einmal anders sein.

Dies gibt nur jeweils einen Vertreter pro Status zurück. Versuchen Sie es einmal. Damit haben Sie alles was Sie brauchen, um die Verzeichnisse zu erstellen.



B8: Erstellen Sie für jede Dateiendung einen eigenen Unterordner in Ihrem Übungsverzeichnis. Hinweis: Erstellen Sie eine Liste aller Dateien und selektieren Sie danach im 2. Befehl ausschliesslich die Eigenschaft „Endung“. Wenden Sie nun darauf die Sortierung an mit dem `-unique` Parameter und Sie erhalten eine Liste aller Dateiendungen, jede aber nur 1x. Wenn Sie soweit sind, können Sie diese Liste einer Variablen zuweisen und zum nächsten Schritt gehen: Mit einer Schleife gehen Sie durch diese Objektliste und erzeugen jeweils ein Verzeichnis mit dem Namen der Endung (`.extension`). Denken Sie daran, dass dies ein ganzer Pfadname sein muss, also inklusive mindestens eines `\` Zeichens! Falls Sie Probleme mit dem Pfad haben sollten, versuchen Sie einmal (`„.\Neu“+$_.Extension`) als Argument. Vergessen Sie nicht die Typ-Angabe (`directory`) am Ende für die Erstellung eines Verzeichnisses!

Das schlussendliche Verschieben der Dateien übernimmt der Befehl `move-item`. Als Argumente nimmt das cmdlet den ganzen Namen des Ausgangsobjekts an und die Pfadangabe des Endverzeichnisses, z.B. `move-item .\test.txt .\neuer-Ordner`



B9: Verschieben Sie alle Dateien aus dem Testordner in die erstellten Unterordner. Hinweis: Die Liste `get-childitem` des Ausgangsordners umfasst nun auch die neu erstellten Unterordner, diese müssen nun ausselektiert werden. Erzeugen Sie also wiederum eine Liste aller Items (schauen Sie die Liste diesmal zunächst genau an). Schränken Sie die Liste ein, der Vergleichsoperator (`...type -notmatch „d“`) kann hierbei helfen. Dann brauchen Sie eine Schleife für die Liste, die nun nur noch Dateien beinhaltet. Die Abschlussschleife ist dann wieder einfach: Finden Sie pro Objekt den richtigen Zielordner anhand der Dateiendung und verschieben Sie die Datei in diesen Ordner. Variablen zum Speichern von Zwischenergebnissen sind immer hilfreich.

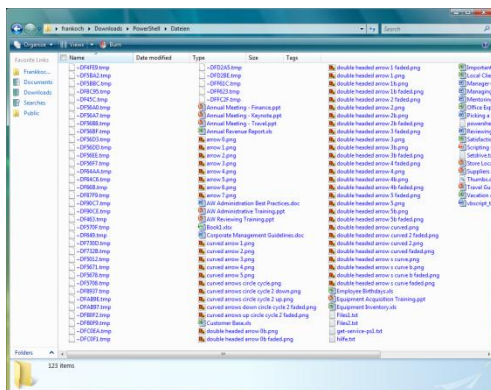


ABBILDUNG 9: DATEIORDNER VOR DER SORTIERUNG

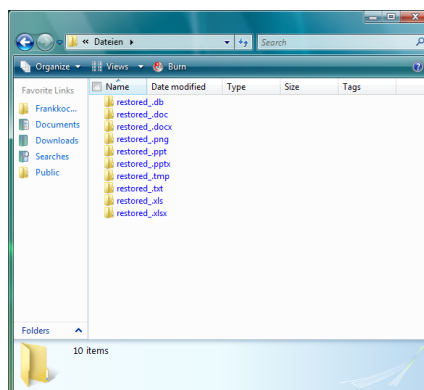


ABBILDUNG 10: DATEIORDNER NACH DER SORTIERUNG

Am Ende geben wir nun noch einmal die ganzen Dateien aus. Der Ausgangspfad sollte nun leer sein, die Unterordner sind jedoch prall gefüllt. Die Ausgabe *get-childitem -recurse* gibt sie uns ebenfalls aus. Speichern wir die Ausgabe in einer TXT Datei um sie in Notepad zu analysieren.



B10: Geben Sie den Inhalt des Übungsordners inklusive aller Unterordner in eine Textdatei aus und speichern Sie diese unter dem Namen FinalOutput.txt.



B11: Um das Leben noch einfacher zu machen, kann noch für jede Word-Datei die Eigenschaft Readonly zurückgesetzt werden. Gehen Sie dazu in das Unterverzeichnis für doc Dateien und rufen sie alle Objekte auf. Die zu setzende Objekteigenschaft lautet IsReadOnly und muss auf 0 (Numerische Null) gesetzt werden. Hinweis: Verenden Sie 2 Befehle: erzeuge Sie die Liste aller Objekte, dann verwenden Sie eine Schleife durch die Objektliste wie schon so oft in diesen Übungen.

Windows PowerShell hilft auch bei den ACL, den Sicherheitsvorgaben. Mit *get-acl* und *set-acl* können Sie diese sehr einfach von einem Objekt auf ein anderes Objekt übertragen oder auch neu erzeugen. Die sprengt jedoch den Rahmen des Workshops, schlagen Sie hierfür bitte in der Online-Hilfe nach.

```

dir . -notepad
File Edit Format View Help

Directory: Microsoft.PowerShell.Core\FileSystem::C:\users\frankoch\downloads\powershell\dateien\restored_db

Mode                LastWriteTime         Length Name
----                -
-a---         12.11.2008         06:06         84992 Thumbs.db

Directory: Microsoft.PowerShell.Core\FileSystem::C:\users\frankoch\downloads\powershell\dateien\restored_doc

Mode                LastWriteTime         Length Name
----                -
-a---         24.08.2005        18:51        192512 Aw Administration Best Practices.doc
-a---         08.07.2005        17:42        130200 corporate management guidelines.doc
-a---         08.07.2005        17:42        194560 Manager-employee conduct.doc
-a---         08.07.2005        17:42        195024 Managing a store.doc
-a---         08.07.2005        17:42        185344 mentoring new managers.doc
-a---         08.07.2005        17:42        195160 Pricing a store location.doc
-a---         08.07.2005        17:42        193024 reviewing employees.doc

Directory: Microsoft.PowerShell.Core\FileSystem::C:\users\frankoch\downloads\powershell\dateien\restored_img

Mode                LastWriteTime         Length Name
----                -
-a---         18.03.2007        12:37        134718 vbscript_to_powershell.docx

Directory: Microsoft.PowerShell.Core\FileSystem::C:\users\frankoch\downloads\powershell\dateien\restored_png

Mode                LastWriteTime         Length Name
----                -
-a---         20.08.2005         00:13         1806 arrow 0.png
-a---         20.08.2005         00:13         1840 arrow 1.png
-a---         20.08.2005         00:13         1880 arrow 2.png
-a---         20.08.2005         00:13         2271 arrow 3.png
-a---         20.08.2005         00:12         2604 arrow 4.png
-a---         20.08.2005         00:12         2756 arrow 5.png
-a---         20.08.2005         00:12         3008 arrow 6.png
-a---         20.08.2005         00:12         3193 arrow 7.png
-a---         20.08.2005         00:13         3161 curved arrow 1.png
-a---         20.08.2005         00:13         4163 curved arrow 2.png
-a---         20.08.2005         00:13         3822 curved arrow 3.png
-a---         20.08.2005         00:13         2777 curved arrow 4.png
-a---         20.08.2005         00:13         14915 curved arrow 5.png
-a---         20.08.2005         00:13         11208 curved arrow circle cycle 2 down.png
-a---         20.08.2005         00:13         11207 curved arrow circle cycle 2 up.png
-a---         20.08.2005         00:13         28193 curved arrow circle cycle 2 faded.png
-a---         20.08.2005         00:13         12181 curved arrow down circle cycle 2 faded.png
-a---         20.08.2005         00:13         12145 curved arrow up circle cycle 2 faded.png
-a---         20.08.2005         00:12         1815 double headed arrow 0b faded.png
-a---         20.08.2005         00:12         3847 double headed arrow 0b.png
-a---         20.08.2005         00:12         4161 double headed arrow 1 faded.png
-a---         20.08.2005         00:12         3885 double headed arrow 1.png
-a---         20.08.2005         00:12         4697 double headed arrow 1b faded.png
-a---         20.08.2005         00:12         4392 double headed arrow 1b.png
-a---         20.08.2005         00:12         4312 double headed arrow 2 faded.png
-a---         20.08.2005         00:12         4031 double headed arrow 2.png

```

ABBILDUNG 11: AUSGABE DES NEUEN INHALTSVERZEICHNIS MIT UNTERORDNER

Falls noch Zeit ist...

Gehen wir noch einmal zurück zu den ganz am Anfang beschriebenen cmdlets *export-csv* und *export-CliXML*. Wenn noch vorhanden, nehmen Sie Ihre Variable *\$P*, ansonsten geben Sie bitte folgende Zeile in der Windows PowerShell ein:

```
$p = get-process
```

Schreiben Sie nun die Variable in eine CSV Datei und eine CliXML Datei:

```
$p | export-CSV .\test.csv
```

```
$p | export-CliXML .\test.xml
```

Jetzt importieren Sie diese Werte in zwei neue Variable:

```
$p1 = import-csv .\test.csv
```

```
$p2 = import-Clixml .\test.xml
```



C1: Errechnen Sie nun den Mittelwert der CPU Belastung, den Maximalwert und den Minimalwert. Hinweis: Verwenden Sie den Befehl *measure-object* wie in den Beispielen oben. Tun Sie dies getrennt für die 3 Variablen *\$p*, *\$p1* und *\$p2*.



C2 Nun sortieren Sie die Variablen, die ja eine Liste von Einträgen umfassen, nach der CPU Zeit und selektieren die ersten 5. Tun Sie dies wieder für alle 3 Variablen getrennt *\$p*, *\$p1* *\$p2*. Sind alle Ergebnisse wirklich gleich? Welche Variable weicht unter Umständen von den anderen ab? Was stimmt hier nicht?

Die Lösung ist relativ einfach. Beim Export in die CSV Datei und dem anschliessenden Import verliert Windows PowerShell den Informationskontext zu den jeweiligen Werten. Damit werden bei der anschliessenden Sortierung die Zahlen zu Strings und 8.0 ist auf einmal vor 800. Bei den XML Dateien werden diese Informationen mit gespeichert und so auch nach dem Import wieder sauber sortiert. Für die Berechnung des Mittelwerts etc. ist das kein Problem, hier ist 8.0 + 800 auch wieder 808.0 und somit ist die Auswertung für *Measure-Object* korrekt, für die Sortierung jedoch nicht. Beachten sie dies beim Speichern Ihrer Variablen; XML kann das sichere Format sein!

DAS ARBEITEN MIT WEITEREN OBJEKTEN

Windows PowerShell als generische Objektverarbeitungsmaschine

Mit der Windows PowerShell müssen Sie nicht ausschliesslich mit den eigenen Objekten arbeiten, sondern Sie haben Zugriff auf die ganze Welt der Objekte inklusive WMI, .NET oder auch COM. Dies sind jeweils Bereiche für sich, die eine eigene umfangreiche Schulung erlauben. Sie finden zu jedem der genannten Themen umfangreiche Sekundärliteratur. Wir können hier noch nicht einmal an der Oberfläche kratzen sondern beschränken uns jeweils nur auf ein kleines Beispiel, um hoffentlich Ihre Neugierde auf mehr zu wecken.

WMI Objekte

WMI Objekte kennen Sie vielleicht aus dem Windows Scripting Host WSH und VBScript. Wenn nicht, lassen Sie sich einfach überraschen, aber erwarten Sie bitte keine tiefergehenden Erklärungen zu WMI. Wir konzentrieren uns ausschliesslich auf die Windows PowerShell Bezüge.

Ein WMI Objekt erzeugt man in der Windows PowerShell mit dem eigenen cmdlet *get-wmiobject*. Dies zeigt auch die Wichtigkeit von WMI an sich. Gehen Sie in die Windows PowerShell und geben Sie folgenden Befehl ein:

```
Get-WMIObject -Class Win32_Computersystem
```

Sie erhalten einige Grundlegende Informationen zu Ihrem System. Im Gegensatz zu VBScript oder anderen Sprachen erspart Ihnen Windows PowerShell aufwendige Syntax und reduziert Ihre Eingabe auf das absolut Notwendige. Sie brauchen lediglich

- das cmdlet *get-WMIObject* um anzudeuten, dass Sie nun mit WMI arbeiten wollen
- die jeweilige WMI Klasse mit der Sie arbeiten wollen, z.B. *-class Win32_Computersystem*

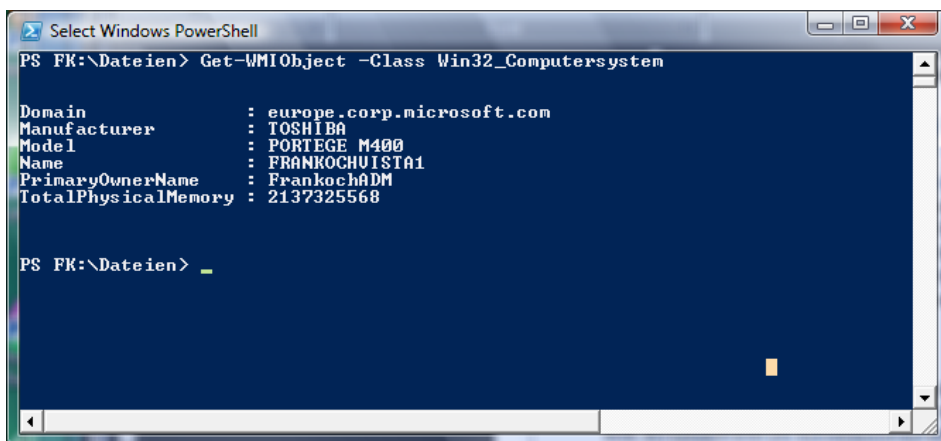


ABBILDUNG 12: AUSGABE DES WMI OBJEKTS WIN32_COMPUTERSYSTEM

Die Ausgabe ist natürlich wieder nur eine kleine Untermenge an allen Daten des Objekts. Verwenden Sie die bekannten Windows PowerShell Befehle *get-member* um sich die Liste an Eigenschaften ausgeben zu lassen.



D1: Zeigen Sie die Eigenschaft Benutzername von Ihrem System an. Hinweis: Verwenden Sie das erste angegebene WMI Beispiel und ermitteln Sie die passende Eigenschaft für den Benutzer.

WMI ist wie gesagt eine Welt für sich. Hier kann man nicht nur Informationen auslesen, sondern auch schreiben. Dies geht sogar über das Netz hinweg auf Fremdsysteme, sofern Sie sich dort authentisieren können. Um die Welt von WMI zu erkunden, gibt es eigene graphische Objekt-Browser wie das CIM Studio. Die Literatur zu WMI zeigt Ihnen hierzu alle Informationen auf. Einige kleine Beispiele möchte ich Ihnen jedoch nicht vorenthalten. Die folgenden WMI Objektklassen bieten oft hilfreiche Informationen im IT Alltag:

Auflisten von Informationen zu Ihrem Desktop

```
get-wmiobject -class win32_desktop -computername .
```

Der Punkt ist Bestandteil des Befehls und gibt an, dass Sie den lokalen Rechner meinen. Ansonsten geben Sie einen anderen Rechnernamen an (server1, server4.mycompany.ch ...)

Informationen zu Ihrem Systembios:

```
Get-wmiobject -class win32_bios
```

Meinen Sie Ihren Rechner, so ist die Angabe „-computername.“ nicht zwingend.

Auflistung der installierten Hotfixe

```
Get-wmiobject -class win32_QuickfixEngineering      oder anders dargestellt:
```

```
Get-wmiobject -class win32_Quickfixengineering -Property HotFixID |
```

```
Select-object -property HotFixID
```

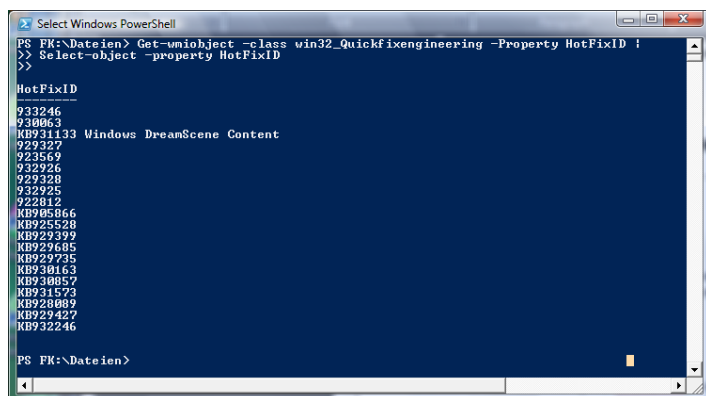


ABBILDUNG 13: AUSGABE DER HOTFIXLISTE NUR ALS KB NUMMERN

Das Schreiben von Daten mittels WMI Objekten verläuft analog zum Umgang mit Objekten in Windows PowerShell. Rufen Sie das jeweilige Objekt mit *Get-member* auf, um die Eigenschaften und Methoden zu erfahren. Dort steht auch, ob dies lediglich auslesbare Werte sind (*get*) oder auch schreibbare Werte sind (*set*). Die Eigenschaft *Visible* aus den späteren COM Beispielen ist so ein Wert, der auslesbar und schreibbar ist.

Arbeiten mit .NET Objekten und XML

Ebenso eindrucksvoll wie mit WMI sind die Möglichkeiten mit .NET und XML. Im Anhang finden Sie hierzu zwei Beispiele. Das eine ruft eine Webseite im Internet auf, verbindet sich an den RSS Feed der Seite und liest alle Topics des RSS Feeds sowie deren genaue URL aus, Ihr eigener RSS Reader sozusagen. Das erste Beispiel ist hierbei eindrucksvoll kurz und wird zu Demonstrationszwecken angegeben. Eine genaue Erläuterung muss entfallen, um den Rahmen des Buchs nicht zu sprengen. In dem wirklich fantastischen Windows PowerShell Buch „Windows PowerShell in Action“ von Bruce Payette, einem der Väter der Windows PowerShell, finden Sie für beide Skripte die ausführlichen Erläuterungen.

```
([xml](new-object net.webclient).DownloadString(  
    "http://blogs.msdn.com/powershell/rss.aspx")).rss.channel.item |  
format-table title,link
```

Ja, Sie sehen richtig, das ganze sind grade einmal zwei Zeilen Skript.

Die Syntax ist wieder ähnlich zu WMI. Ignorieren wir zunächst die Klammer [XML], dann steht dort ein leicht zu lesendes cmdlet *new-object*. Hiermit erzeugen Sie ein neues Objekt. Da wir den Typ direkt angeben (*net.webclient*), weiss Windows PowerShell auch gleich, um was es sich handelt: ein .NET Objekt vom Typ *Webclient*.

Von diesem Objekt nutzen wir die Methode *DownloadString(url)*, welche von der angegebenen URL die Ausgabe (die „Webseite“ sozusagen) ausliest.

Der Rest ist „Mogelei“: da wir wissen, was sich hinter der URL versteckt, überprüfen wir das Ganze nicht sondern gehen davon aus, dass es sich um einen RSS Feed handelt und sprechen damit das Objekt mit für RSS Feeds bekannte Eigenschaften an. Wenn Sie eine andere URL eingeben muss das Skript daher unwiderruflich fehlschlagen, es sei denn, es handelt sich per Zufall wieder um einen RSS Feed.

Das *Format-table* ist wieder ein ganz normales cmdlet und gibt einfach eine Liste von Objekten als Tabelle aus. Die gewünschten Spalten kann man dazu direkt angeben wie hier Titel (*title*) und URL (*link*) des RSS Feeds.

Das zweite Beispiel finden Sie im Anhang, dort als Beispiel 4 aufgelistet. Auch dieses Skript verwendet die Möglichkeit, .NET Objekte in Windows PowerShell Skripte einzubinden. In diesem Beispiel wird eine Windows Fensterapplikation verwendet. Kopieren Sie das Beispiel einmal in Ihre Windows PowerShell und starten Sie es. Das Ergebnis ist hoffentlich beeindruckend genug, um Lust auf mehr zu bekommen. Informationen zur .NET Programmierung und zu .NET Objekten finden Sie entweder auf den MSDN Webseiten von Microsoft oder in einem der vielfältigen Bücher über .NET.

Arbeiten mit COM Objekten

Als letztes Beispiel möchte ich noch kurz auf die Möglichkeiten mit COM Objekten eingehen. Hierzu verwende ich ein Excel-Beispiel. Sollten Sie auf Ihrem Rechner kein Excel installiert haben, so können Sie auch das alternative InternetExplorer-Demo durchspielen. Jedoch verzichte ich auch hier auf eine ausführliche Einleitung zu COM. Hierzu gibt es so viele Bücher auf dem Markt, dass von mir nun wirklich nichts mehr hinzuzufügen ist.

Als erstes müssen wir Windows PowerShell mitteilen, dass wir mit einem COM Objekt arbeiten möchten. Damit wir später weniger tippen müssen, ordnen wir es gleich einer Variablen zu. So haben wir unser eigenes COM Objekt der gewünschten Art. Für die Nutzung von COM Objekten gibt es nicht wie für WMI Objekte ein eigenes cmdlet, sondern wir benutzen das generische cmdlet für neue Objekte: *new-object*.

```
$a = New-Object -comobject Excel.Application
```

Als Argument geben wir mit, welches COM Objekt wir wollen. Hier ist es Excel, ausgewählt mit dem Argument *Excel.Application*. Wie Sie herausfinden können, welche COM Objekte auf Ihrem System vorhanden sind und wie sie genau heissen wird hier verschwiegen. Vielleicht schauen Sie dazu einmal auf den MSDN Seiten von Microsoft vorbei oder verwenden eines der vielen Bücher zu COM. Neben Excel als Sammelstelle für Daten, um einfach Berichte daraus zu erzeugen, könnte im Alltag eines IT Administrators vielleicht auch Visio zur graphischen Darstellung von Systemwerten für die Automatisierung mit Windows PowerShell interessant sein. Bei MS Press finden Sie hierzu einige Bücher, die die COM Objekte von Visio sehr gut beschreiben. Aber kommen wir zurück zu unserem Excel Beispiel.

Um Daten aufzunehmen, braucht Excel nun ein Workbook. Da wir hier nun die Welt von COM betreten haben, müssen wir auch die Syntax von COM verwenden. Windows PowerShell hilft uns aber, das ganze so einfach wie möglich zu halten. Die Syntax können wir uns mit *\$a | get-member* erklären lassen. Auch bei COM Objekten listet *get-member* ganz brav alle Eigenschaften und Methoden auf. Erwarten Sie nur einfach mehr von denen bei einer Applikation wie Excel als bei einem kleinen Objekt, welche wir bisher betrachtet haben. Die Methode *Workbooks.Add()* erstellt uns nun ein neues Workbook, Sie finden aber auch Methoden zum Laden eines existierenden Workbook. Hierbei müssten Sie dann den Pfad angeben.

```
$b = $a.Workbooks.Add()
```

Eventuell erhalten Sie eine Fehlermeldung in der Form „Error: 0x80028018 (-2147647512) Description: Old Format or Invalid Type Library“ beim Anlegen des Excel Workbooks. Der Microsoft Knowledge Base Artikel 320369 beschreibt das Problem. In der Regel tritt dieser Fehler dann auf, wenn Excel in einer anderen Sprache installiert ist (z.B. Englisch) als die „Regionalen Einstellung“ des darunter liegenden Windows (z.B. Deutsch (Schweiz)). Dies ist ein Bug im Excel Objekt. Zur Zeit der Fertigstellung des Buchs gab es noch keinen Fix von Microsoft für diesen Bug. Als Workaround setzen Sie in diesem Fall für die Testzwecke in der Systemsteuerung Ihre Regional Settings auf English-US und starten Sie die Windows PowerShell erneut. Anschliessend sollte das Beispiel für Excel problemlos funktionieren.

Zu einem Workbook gehören automatisch auch Worksheets. Wir wählen nun das erste davon aus

```
$c = $b.Worksheets.Item(1)
```

Wollen wir in das Worksheet schreiben, so tun wir das nicht auf das Sheet sondern in eine Zelle im Excelsheet. Daher müssen wir die gewünschte Zelle genau angeben:

```
$c.Cells.Item(1,1) = "Windows PowerShell rocks!"
```

Das war es schon. Unser Eintrag in Excel ist fertig. Sie glauben mir nicht? Ok, wie wäre es denn, wenn wir Excel nun mal erscheinen lassen und sie nachschauen können? Der Befehl

```
$a.Visible = $True
```

setzt die Eigenschaft „Sichtbar“ (visible) unseres Excelobjekts auf den Wert „wahr“ (\$true). Und schon erscheint wie aus dem Nichts unser erzeugtes Excel und zeigt unseren Eintrag.

Neben der Eigenschaft „Sichtbar“ können wir auch Methoden (Funktionen) von Excel aufrufen und für uns nutzen. Sie erraten sicherlich, was folgende Zeile macht:

```
$b.SaveAs(".\Test.xls")
```

Und falls Sie sich wundern: wir sichern hier nicht Excel, sondern natürlich das entsprechende Workbook aus dem Excel. Denn wir sind ja an der XLS Datei interessiert, nicht am Programm.

Natürlich ist diese Demo wieder ein kleiner Betrug. Ich habe Ihnen nicht gross gezeigt, wie Sie an die Informationen kommen, um Excel zu manipulieren. Woher weiss man, dass die Excel Arbeitsmappe „Workbook“ heisst? Und wie kommt man auf den Befehl „SaveAs“ und dessen Syntax? All das ist in den Welten von COM enthalten, jede COM Informationsquellen können Sie hierfür anzapfen und einfach in Windows PowerShell nutzen, *get-member* hilft natürlich ebenfalls. Da es hier lediglich um das Tools Windows PowerShell geht, beschränke ich mich auf die Nutzung der Informationen, nicht um die Quelle der Informationen. Am Ende sollten wir übrigens noch aufräumen und Excel schliessen, wenn nicht schon manuell passiert:

```
$a.Quit()
```

Falls Sie Lust haben, versuchen Sie diese kleine Aufgabe zu lösen:



D2: Erstellen Sie eine Liste aller Services und tragen Sie den Namen der Services sowie deren Status in ein Excel Sheet ein. Hinweis: Verwenden Sie das hier gezeigte Beispiel zur Erstellung eines Excel Objekts und der Zuweisung in eine Variable. Um die Zeile im Excelsheet zu adressieren, verwenden wir eine eigene Variable *\$i*. Nehmen Sie das Skript zur Farbausgabe der Services und ersetzen Sie die Farbausgabe durch einen *\$c.Cells.Item(\$i,1)* Eintrag. Vergessen Sie nicht, *\$i* nach jeder Zeile hochzuzählen, z.B. mit *\$i = \$i + 1*. Mehrere Befehle können Sie in einer Zeile mit dem Semikolon ; trennen. Speichern Sie das Ergebnis aus Excel in einer XLS Datei, aber bitte automatisch von Ihrem Skript aus, nicht manuell aus dem Dateimenü in Excel.

Als Ergebnis werden Sie sehen, dass der Status der Services in Excel als Zahl dargestellt wird. Windows PowerShell ist so nett, uns diese Zahl durch einen lesbareren Text wie „running“ oder „stopped“ bei der Ausgabe auf dem Bildschirm gleich zu ersetzen.

Service Name	Service Status
AdtAgent	1
AeLookupSvc	4
ALG	1
AppInfo	1
AppMgmt	1
AudioEndpointBuilder	4
AudioSrv	4
BFE	4
BITS	4
Browser	1
BthServ	4
CcmExec	4
CertPropSvc	4
clr_optimization_v2.0.50727_32	1
COMSysApp	1
CryptSvc	4
CscService	4
DcomLaunch	4
DFSR	1
Dhcp	4
Dnscache	4
dot3svc	1

ABBILDUNG 14: AUSGABE DER SERVICENAMEN UND STATUS IN EXCEL 2007. ZUSÄTZLICH WURDE DIE BEDINGTE FORMATIERUNG VON EXCEL 2007 VERWENDET, UM DEN STATUSWERT STATT EINER ZAHL ALS EIN ICON DARZUSTELLEN.

Sollten Sie kein Excel auf Ihrem Rechner haben, so können Sie eine andere Übung nutzen. Hierfür verwenden Sie den Internetexplorer. Statt Daten in eine Zelle einzutragen, surfen wir auf eine Webseite. Eine Kombination mit dem RSS Reader von oben ist dann nur noch ein kleiner Schritt: Lesen Sie die RSS Feeds automatisch, suchen Sie im Titel nach Schlüsselwörter, und rufen Sie anschliessend die Webseiten automatisch auf. Bequemes Surfen von der Couch war noch nie einfacher.

Das Skript beginnt wie das Excelskript mit dem Aufruf von *new-object*:

```
$ie = new-object -comobject InternetExplorer.application
```

Auch hier ist das neue Objekt zunächst unsichtbar, aber wie bei Excel können wir es ganz einfach hervorholen:

```
$ie.Visible = $True
```

Die Möglichkeiten, die uns das InternetExplorer Objekt bietet, erfahren wir ebenfalls mit dem cmdlet *get-member*:

```
$ie | get-member
```

Um es einfach zu halten, surfen wir auf eine bekannte Webseite:

```
$ie.Navigate(http://www.microsoft.com/powershell)
```

Falls Sie Lust haben, kombinieren Sie doch dieses Skript mit dem RSS Reader. Hier erhalten Sie eine Menge von interessanten Links und Titeln. Nehmen Sie die Liste von Titeln, schauen Sie nach Schlagwörtern und bei einem Treffer surfen Sie auf die Seite. Die Suche nach Schlagwörtern in der Liste von Titeln haben wir jedoch bisher nicht betrachtet, die Windows PowerShell Hilfe muss dazu verwendet werden. In diesem Buch bleiben wir Ihnen die Lösung zu diesem Problem jedoch schuldig.

Arbeiten mit Eventlogs

Zum Abschluss der praktischen Übungen dieses Buches gehen wir noch auf das Eventlog ein. Windows PowerShell erlaubt den Zugriff auf das Eventlog über vielfältige Verfahren. So bringen WMI, .NET als auch COM Objekte hierfür umfangreiche Befehle mit. Aber auch in der Windows PowerShell finden sich bereits einige cmdlets, um den Alltag zu bestreiten. Das wichtigste cmdlet ist hierbei *get-eventlog*.

Der Befehl *Get-EventLog -list* zeigt einem alle Eventlogs des jeweiligen Systems an. Um auf ein bestimmtes Eventlog zugreifen zu können, kann man das bereits vorgestellte cmdlet *where-object* verwenden. Der Zugriff auf das Systemlog erfolgt damit wie folgt:

```
Get-EventLog -list | Where-Object {$_.logdisplayname -eq "System"}
```

Es geht aber natürlich auch einfacher, diese Zeile Code wäre sonst ungewohnt lang für den Alltag als Windows PowerShell Anwender. Möchte man daher Einträge aus dem Eventlog auswerten, so ruft man einfach *get-eventlog* mit dem Namen des gewünschten Eventlogs auf. Die Ausgabe kann unter Umständen sehr lang werden und kann jederzeit mit Ctrl-C abgebrochen werden. Möchte man nur die neusten 20 Einträge haben, so hilft einem der Parameter *-newest 20* über die Runden. Die 20 kann natürlich durch eine beliebige Zahl ersetzt werden.

```
Get-EventLog system -newest 3
```

```
Get-EventLog system -newest 3 | Format-List
```

Die Events stehen anschliessend natürlich der gewohnten Windows PowerShell Verarbeitung zur Verfügung und können so sortiert und gruppiert werden.



E1: Suchen Sie nach dem Namen des Windows PowerShell Eventlogs. Gruppieren Sie die Eventeinträge nach ihrer EventID und sortieren Sie diese anschliessend nach ihrem Namen. Listen Sie in einer weiteren Aufgabe nur die Events mit der ID 403 auf. Hinweis: Sollte der Eventlogname ein Leerzeichen beinhalten, so setzen Sie den ganzen Namen in Anführungszeichen „mein Eventlog“.



E2: Sortieren Sie die 15 neuesten Einträge des System Eventlogs absteigend nach ihrer Event ID. Hinweis: Lesen Sie das Buch noch einmal, wenn Sie die Lösung nicht direkt hinschreiben können.

LÖSUNGEN DER ÜBUNGSAUFGABEN

Lösungsskripte zu den Aufgaben aus dem Buch

A1

```
get-process | sort-object CPU
```

A2a

```
get-process | sort-object CPU -descending | select-object -first 10
```

```
get-process | sort-object CPU | select-object -last 10
```

A3

```
$P = get-process | sort-object CPU -descending | select-object -first 10
```

A4

```
$P > .\A4.txt
```

```
$P | export-csv .\a4.csv
```

```
$P | export-CliXML .\a4.xml
```

A5

```
get-service | sort-object status
```

A6

```
get-service | foreach-object{ write-host $_.name $_.status}
```

A7

```
get-service | foreach-object{ write-host -f yellow -b red $_.name $_.status}
```

Statt `-f` können Sie auch `-foregroundcolor` und statt `-b` auch `-backgroundcolor` schreiben

A8

```
get-service | foreach-object{
```

```
if ($_.status -eq "stopped") {write-host -f green $_.name $_.status}`
```

```
else{ write-host -f red $_.name $_.status}}
```

A9

```
Get-service | convertto-html
```

A10

```
Get-service | convertto-html > .\a10.html
```

A11

```
Get-service | sort-object status | convertto-html name, status > .\a10.html
```

B1

Get-ChildItem * -Exclude *.tmp | Select-Object name, length

B2

Get-ChildItem * -Exclude *.tmp | Select-Object name, length | sort-object length, name

B3

Get-childitem | get-member

B4

get-childitem | Group-Object extension | sort-object count

B5

(Get-childitem .*.tmp | measure-object length -sum).sum

B6

Remove-item .*.tmp

B7

Get-childitem | where-object {\$_.length -gt 2000000} | foreach-object {remove-item \$_.fullname}

B8

get-childitem | select-object extension | sort-object extension -unique `

| foreach-object {new-item (".\Neu" + \$_.extension) -type directory}

B9

get-childitem | where-object {\$_.mode -notmatch "d"} | foreach-object {\$b= ".\Neu" + `

\$_extension; move-item \$_.fullname \$b}

B10

Get-childitem -recurse > .\FinalOutput.txt

B11

get-childitem *.doc | foreach-object {\$_.Isreadonly = 0}

C1

```
$p | measure-object CPU -min -max -average
```

C2

```
$p | sort-object CPU -Descending | Select-Object -first 5
```

D1

```
(Get-WMIObject -Class Win32_Computersystem).username
```

D2

```
$a = New-Object -comobject Excel.Application
```

```
$a.Visible = $True
```

```
$b = $a.Workbooks.Add()
```

```
$c = $b.Worksheets.Item(1)
```

```
$c.Cells.Item(1,1) = "Service Name"
```

```
$c.Cells.Item(1,2) = "Service Status"
```

```
$i = 2
```

```
Get-service | foreach-object{ $c.cells.item($i,1) = $_.Name; $c.cells.item($i,2) = $_.status; $i=$i+1}
```

```
$b.SaveAs("C:\Users\frankoch\Downloads\Test.xls")
```

```
$a.Quit()
```

E1

```
Get-EventLog "Windows PowerShell" | Group-Object eventid | Sort-Object Name
```

```
Get-EventLog "Windows PowerShell" | Where-Object {$_.EventID -eq 403}
```

E2

```
Get-EventLog system -newest 5 | Sort-Object eventid -descending
```


ANHANG

SCRIPTBEISPIELE

Windows PowerShell Beispiele – von einfach bis komplex

Die folgenden Skripte können Sie einfach kopieren und direkt in der Windows PowerShell ausführen. Sie zeigen die prinzipiellen Möglichkeiten, aber auch, dass eine kurze Einleitung nicht in der Lage ist, das volle Spektrum der Windows PowerShell abdecken zu können. Markieren Sie dazu alle Textzeilen unter dem ersten > Zeichen bis zum Kommentar (diesen dann nicht mehr). Diese Übungen stammen vor allem aus dem sehr empfehlenswerten Buch „Windows PowerShell in action“ von Bruce Payette, einem der Väter der Windows PowerShell. In diesem Buch sind die Beispiele auch ausführlicher erläutert, falls Sie noch mehr Informationen hierzu wünschen.

Beispiel 1: Direkte Ausgabe eines Strings

Das kürzeste Hello-World Programm :>

```
"hello world"
```

Kommentar:

Windows PowerShell kann Strings direkt erkennen und gibt Sie anschliessend aus.

Beispiel 2: Logfile Analyse

Erstellen einer Liste aller Log-Dateien im "Windir" Verzeichnis, durchsuchen der Log-Dateien nach dem Wort „Error“, Ausgabe des Logdateinamens und der Zeile mit dem Fehler:>

```
dir $env:windir\*.log | select-string -List error | format-table path,linenumber –autosize
```

Kommentar:

Eventuell kann es Fehlermeldungen aufgrund von Zugriffsberechtigungen etc. geben, die Sie ignorieren sollten.

Beispiel 3: Der eigene RSS Reader

Aufrufen einer Webseite, auslesen des RSS Channels, Darstellen der RSS Beiträge und deren URLs:>

```
([xml](new-object net.webclient).DownloadString(  
    "http://blogs.msdn.com/powershell/rss.aspx")).rss.channel.item | format-table title,link
```

Kommentar:

Die Verbindung zur Webseite kann anfangs etwas dauern.

Beispiel 4: Einbindung von Windows Fenstern in Skripte

Erstellen einer eigenen WinForm zur grafischen Ausgabe von Infos etc:>

```
[void][reflection.assembly]::LoadWithPartialName("System.Windows.Forms")

$form = new-object Windows.Forms.Form

$form.Text = "My First Form"

$button = new-object Windows.Forms.Button

$button.text="Push Me!"

$button.Dock="fill"

$button.add_click({$form.close()})

$form.controls.add($button)

$form.Add_Shown({$form.Activate()})

$form.ShowDialog()
```

Kommentar:

Zum Beenden einfach auf das neu erstellte Fenster klicken (evtl. von der Windows PowerShell verdeckt).

ANHANG

DIE THEORIE ZUR WINDOWS POWERSHELL

Theoretische Grundlagen zur Windows PowerShell

Windows Powershell – eine kurze Einführung

Microsofts bisherige Versuche mit Kommandozeilen-Shells waren eher unerfreulich. Die alte `command.com` mag für die ersten Versionen von MS DOS adäquat gewesen sein, doch den wachsenden Betriebssystemfunktionen hielt sie nicht stand. Die mit Windows NT eingeführte `cmd.exe` Shell stellte da schon deutlich mehr Möglichkeiten bereit. Im Vergleich mit gängigen Unix-Shells wie der Bash zieht Microsofts Kommandozeile aber deutlich den Kürzeren.

Das hat Microsoft nun komplett geändert. Mit der Windows PowerShell (vormals Monad Shell, MSH) möchte der Software-Konzern Administratoren auch unter Windows eine Shell an die Hand geben, mit der sie ganz nach Herzenslust Scripts schreiben und ihr System kontrollieren können. Dabei verfolgt die PowerShell ein von Grund auf anderes Konzept, als das bei textorientierten Shells wie der Bash oder auch `cmd.exe` der Fall ist.

Ziele beim Entwurf von Windows PowerShell

Windows PowerShell ist eine neue Windows-Befehlszeilenshell, die speziell für Systemadministratoren entwickelt wurde. Die Shell umfasst eine interaktive Eingabeaufforderung und eine Skriptumgebung, die einzeln oder zusammen verwendet werden können. Im Gegensatz zu den meisten Shells, die Text akzeptieren und zurückgeben, basiert Windows PowerShell auf einem Objektmodell, zur Verfügung gestellt durch das .NET Framework 2.0. Diese grundlegende Änderung in der Umgebung ermöglicht völlig neue Tools und Verfahren für die Verwaltung und Konfiguration von Windows.

Mit Windows PowerShell wird das Konzept von Cmdlets eingeführt (ausgesprochen „Command-let“). Ein Cmdlet ist ein einfaches Befehlszeilentool, das in der Shell integriert ist und eine einzelne Funktion ausführt. Obwohl Sie Cmdlets einzeln verwenden können, wird deren Leistungsfähigkeit besonders sichtbar, wenn Sie diese einfachen Tools kombiniert einsetzen, um komplexe Aufgaben auszuführen. Windows PowerShell enthält mehrere Hundert Basis-Cmdlets, und Sie können zusätzlich eigene Cmdlets schreiben und an andere Benutzer weitergeben.

Wie viele andere Shells bietet Windows PowerShell Zugriff auf das Dateisystem des Computers. Darüber hinaus haben Sie über Windows PowerShell-Anbieter genauso leicht Zugriff auf andere Datenspeicher, z. B. auf die Registrierung und auf Speicher für digitale Signaturzertifikate.

Von Texten, Parsern und Objekten

Die PowerShell arbeitet vollständig objektorientiert. Anders als bei üblichen Shells handelt es sich bei den Ergebnissen jedes Kommandos der PowerShell also nicht um einen Text, sondern um ein *Objekt* (dazu gleich mehr). Ähnlich wie bei bisher bekannten Shells gibt es allerdings eine *Pipeline*, in der die Resultate der einzelnen Befehle weitergereicht und weiterverarbeitet werden können. Nur sind die Eingabewerte sowie die Resultate eben Objekte und keine Texte.

Die Objekte der PowerShell unterscheiden sich nicht von denen eines C++- oder C#-Programms. Sie können sich ein Objekt als Dateneinheit mit Eigenschaften (Merkmale) und Methoden vorstellen. Methoden sind Aktionen, die für das Objekt ausgeführt werden können.

Wenn Sie beispielsweise einen Dienst oder *Service* in Windows PowerShell abrufen, verwenden Sie eigentlich ein Objekt, das diesen Dienst darstellt. Wenn Sie Informationen über einen Dienst anzeigen, zeigen Sie die Eigenschaften des zugehörigen Dienstobjekts an. Und wenn Sie einen Dienst starten, d. h. die **Status**-Eigenschaft des Dienstes auf **started** festlegen, verwenden Sie eine Methode des Dienstobjekts. Mit zunehmender Erfahrung werden Sie die Vorteile der Objektverarbeitung besser verstehen und bewusst mit den Objekten arbeiten.

Das objektorientierte Konzept der PowerShell macht das von Unix-Shells bekannte dauernde Parsen (Analysieren / Auswerten) von textbasierten Informationen mit all seinen Fehlermöglichkeiten überflüssig. Zur Verdeutlichung folgendes Beispiel:

Angenommen, Sie hätten gerne eine Liste aller Prozesse, die mehr als 100 Handles verbrauchen. Mit einer traditionellen Linux-Shell würden Sie dazu das Kommando zur Anzeige aller Prozesse (`ps -A`) aufrufen. Das Kommando liefert dann eine Textliste zurück. In jeder Zeile stehen dabei Informationen über einen Prozess, durch Leerzeichen voneinander getrennt. Diese Zeilen würde man dann mit einem der gängigen Tools parsen, die Prozess-ID ausschneiden und über diese mit einem weiteren Programm die Handle-Anzahl abfragen. Das textbasierte Ergebnis dieser Abfrage würde man erneut parsen, die relevanten Zeilen herausfiltern und den relevanten Text schliesslich anzeigen.

Je nachdem, wie gut das Ausschneiden und Filtern der Information aus den Textlisten gelöst ist, klappt dieser Ansatz mehr oder weniger zuverlässig. Ändert sich zum Beispiel die Breite einer Spalte der Ausgabe durch zu lange Prozessnamen, kann es schon schief gehen.

Die PowerShell verwendet einen grundlegend anderen Ansatz. Hier startet man ebenfalls das Kommando `get-process`, das alle laufenden Prozesse vom Betriebssystem liefert. Nur werden sie als ein *Listenobjekt* („Liste von Objekten“) von Prozessobjekten geliefert. Diese Objekte kann man dann auf ihre Eigenschaften hin untersuchen und direkt abfragen – es müssen also keinerlei Textzeilen untersucht und in Spalten aufgeteilt werden. Dies werden wir uns gleich in der Praxis noch etwas genauer anschauen.

Eine neue Skriptsprache

Für Windows PowerShell wird keine der bestehenden Sprachen, sondern eine eigene Sprache verwendet. Dafür gibt es folgende Gründe:

- Für Windows PowerShell wurde eine Sprache für die Verwaltung von .NET-Objekten benötigt.
- Die Sprache musste komplexe Aufgaben unterstützen, ohne einfache Aufgaben unnötig kompliziert zu machen.
- Die Sprache musste den Konventionen anderer bei der .NET-Programmierung verwendeter Sprachen wie C# entsprechen.

Jede Sprache hat nun seine eigenen Befehle. Bei Windows PowerShell hat man darauf geachtet, dass diese Befehle alle einer gewissen Logik beim Aufbau und der Benennung folgen. Ein *Cmdlet* ist ein

spezialisierte Befehl, der Objekte in Windows PowerShell bearbeitet. Sie können Cmdlets an ihrem Namen erkennen: ein Verb und ein Substantiv, immer in Einzahl, getrennt durch einen Bindestrich (-), beispielsweise *Get-Help*, *Get-Process* und *Start-Service*. In Windows PowerShell sind die meisten Cmdlets sehr einfach und sind für die Verwendung zusammen mit anderen Cmdlets vorgesehen. So werden z. B. mit Get-Cmdlets nur Daten abgerufen, mit Set-Cmdlets nur Daten erzeugt oder geändert, mit Format-Cmdlets nur Daten formatiert und mit Out-Cmdlets nur Ausgaben an ein angegebenes Ziel geleitet.

Windows-Befehle und -Dienstprogramme

Gewisse Befehle hat man sich im Laufe der Zeit angeeignet. Eine neue Sprache, die dies nicht berücksichtigte, würde schnell störend wirken und so auf keine grosse Akzeptanz stossen. Sie können in Windows PowerShell daher auch herkömmliche Windows-Befehlszeilenprogramme ausführen und Windows-Programme starten, die über eine grafische Benutzeroberfläche verfügen, beispielsweise Editor (Notepad) und Rechner (Calculator). Ausserdem können Sie wie in **Cmd.exe** Textausgaben von Programmen erfassen und diese Texte in der Shell verwenden. Auch wenn Befehle wie *dir*, *ls* oder *cd* nicht der offiziellen Syntax von Windows PowerShell folgen, so funktionieren sie doch auch weiterhin und können getrost weiterverwendet werden.

Eine interaktive Umgebung

Wie andere Shells unterstützt Windows PowerShell eine vollständig interaktive Umgebung. Wenn Sie einen Befehl an der Eingabeaufforderung eingeben, wird der Befehl verarbeitet, und die Ausgabe wird im Shellfenster angezeigt. Sie können die Ausgabe eines Befehls an eine Datei oder einen Drucker oder mithilfe des Pipelineoperators (|) an einen anderen Befehl senden.

Skriptunterstützung

Wenn Sie immer wieder bestimmte Befehle oder Befehlsfolgen ausführen, empfiehlt es sich, die Befehle nicht einzeln an der Eingabeaufforderung einzugeben, sondern in einer Datei zu speichern und diese Befehlsdatei auszuführen. Eine solche Datei mit Befehlen wird als Skript bezeichnet.

Neben der interaktiven Oberfläche bietet Windows PowerShell auch eine vollständige Skriptunterstützung. Sie können ein Skript ausführen, indem Sie den Namen des Skripts an der Eingabeaufforderung eingeben. Die Dateierweiterung für Windows PowerShell-Skripts lautet **.ps1**, die Angabe der Datennamenerweiterung ist aber optional.

Obwohl Skripts sehr nützlich sind, können Sie gleichzeitig auch zur Verbreitung von böartigem Code verwendet werden. Daher können Sie über die Sicherheitsrichtlinie in Windows PowerShell (auch als Ausführungsrichtlinie bezeichnet) festlegen, welche Skripts ausgeführt werden dürfen und ob diese über eine digitale Signatur verfügen müssen. Um unnötige Risiken zu vermeiden, ist es in keiner der Ausführungsrichtlinien in Windows PowerShell zulässig, ein Skript durch Doppelklicken auf dessen Symbol auszuführen, wie Sie es z.B. mit den guten alten .bat, .cmd oder .vbs Dateien können.

CMD, WScript oder Powershell, muss ich mich entscheiden?

Ab Windows XP stehen Ihnen alle drei genannten Scripting-Shells zur Verfügung: die gute alte CMD Shell, der Windows Scripting Host für Ihre VB oder J-Skripte sowie neu nun auch die Windows PowerShell. Und keine Angst, Sie müssen sich nicht entscheiden oder befürchten, dass kurz über lang eine der Shells verschwindet. Auch in den aktuellen Windows Varianten wie Vista oder Longhorn Server finden Sie noch alle 3 Shells gleichberechtigt nebeneinander. Sie können also nach eigenem

Geschmack das verwenden, was Ihnen am liebsten ist. Oder Sie verwenden die Shell, die am besten für Ihre Aufgabe passt. Wenn Sie bisher wenig Skripte geschrieben haben, dann könnte es für Sie der richtige Zeitpunkt sein, direkt mit Windows PowerShell anzufangen und so gleich auf die neueste und bequemste Technologie zu setzen. Anhand der Praxiskapitel haben Sie oder werden Sie noch sehen, wie leicht das ist und wie mächtig einfach Skripte sein können, ohne erst dicke Wälzer zu lesen und Bücher aus den 60er und 70er Jahren des letzten Millenniums zum Thema Batchprogrammierung hervorzukramen.

Windows PowerShell 1.0

Auch wenn Windows PowerShell eine 1.0 Version darstellt, so kann die Qualität des Produkts vollkommen überzeugen und der Einsatz in der Praxis mit gutem Gewissen empfohlen werden. Dies liegt vor allem daran, dass Windows PowerShell eigentlich nur eine neue Art zur Verwendung des bekannten und bewährten .NET Framework 2.0 ist. Somit merkt man dem Produkt die geringe Versionsnummer nicht an.

Lediglich beim Funktionsumfang kann man feststellen, dass noch nicht alles Gewünschte in Windows PowerShell enthalten ist. So kann Windows PowerShell zwar schon heute direkt auf .Dateisysteme, Eventlogs, Registrierung (Registry), .NET, WMI und ADSI Schnittstellen und Objekte zugreifen, ein echtes, eigenes Remoting fehlt jedoch noch. Das heisst der Zugriff auf andere Rechner funktioniert so gut, wie es die bekannten WMI oder .NET Befehle erlauben. Für die Zukunft könnte man sich hier vielleicht auch die Nutzung neuer Webservice Schnittstellen für Management vorstellen, welche im Laufe 2006 zertifiziert worden sind. Bis dahin kann man aber das Erprobte von heute wie WMI und .NET erst einmal nutzen.

ANHANG

SKRIPTE & SICHERHEIT

Sicherheit beim Einsetzen von Skripten

Mit dem Windows Scripting Host (WSH) hatte Microsoft mit Windows 2000 eine neue mächtige Scriptengine eingeführt. Diese Engine war so mächtig, dass sie schnell als neue Angriffsfläche für Virenautoren genutzt wurde. Unbedarfte Anwender erhielten schon bald erste Emails mit der Versprechung schöner Bilder, jedoch bekam man beim Öffnen des Anhangs nichts zu sehen sondern der Anhang entpuppte sich als VBScript, welches dem System mächtig zusetze. Windows Powershell versucht hier sein Bestes, diesen Bedrohungen entgegenzuwirken.

So führt in der Grundeinstellung Windows PowerShell überhaupt erst keine Skripte aus. Das muss explizit vom Systemadministrator freigeschaltet werden. Die Freischaltung erlaubt dabei unterschiedliche Abstufungen, welche alle mit der Signierung von Skripten zusammenhängen. Zusätzlich ist die Endung der Windows PowerShell (PS1) mit Notepad verknüpft. Selbst wenn Ihre Umgebung Skripte zulässt, würde ein unbedarfter Doppelklick auf ein Attachment oder eine Datei immer lediglich Notepad öffnen und einem den Quelltext anzeigen. Auch verlangt PowerShell immer die explizite Pfadangabe „.\“ für Dateien, die direkt aus dem aktuellen Verzeichnis aufgerufen werden sollen. So kann verhindert werden, dass man der Windows PowerShell einen Befehl unterschiebt, den man dort gar nicht erwartet hat.

Um Skripte auszuführen, muss man die Sicherheit der Windows PowerShell anpassen. Hierzu gibt es die zwei cmdlets *get-executionpolicy* und *set-executionpolicy*. Mit *get-executionpolicy* fragt man die aktuellen Einstellungen ab. Es gibt dabei die vier Stufen:

Policy Wert	Beschreibung
Restricted (Default)	Keine Skripte werden ausgeführt
Allsigned	Nur signierte Skripte werden ausgeführt
RemoteSigned	Lookal erstellte Skripte sind erlaubt, aber andere Skripte müssen signiert sein
Unrestricted	Jedes Skript wird ausgeführt

Um die Einstellung zu ändern, muss ein Systemadministrator zum Beispiel den Befehl

Set-ExecutionPolicy RemoteSigned

aufrufen. Mehr zu dem Thema finden Sie auch in der Dokumentation zur Windows PowerShell.

SCHLUSSWORTE

Ich möchte meiner Frau Petra für ihre Geduld und Liebe danken. Sie musste auf manche gemeinsame Wochenenden und Feierabende mit mir verzichten, die ich vorm Rechner statt mir ihr verbrachte.

Dieses Buch wäre ohne die Entwickler der Windows PowerShell nicht erschienen, daher gilt mein Dank auch Ihnen. Besonders natürlich Bruce Payette, der mit seinem Buch „Windows PowerShell in action“ die letzten Anregungen für das erste Manuskript brachte. Lesen Sie es selber, es inspiriert!

Nicht alle Teile des Buchs stammen von mir, vor allem im Theorieteil stammt der meiste Inhalt von den Veröffentlichungen der Microsoft MSDN Seiten oder der Windows PowerShell Hilfe. Hier finden Sie weitergehende Informationen, welche ich Ihnen nahelege zu lesen.

Und wenn Sie noch Anregungen oder Feedback zu diesem Buch haben, schreiben Sie mir einfach per Email Ihre Gedanken. Ich werde vielleicht nicht auf alles eingehen können, freue mich aber sehr über jede Form von konstruktiver Kritik als auch Lob: frankoch@microsoft.com

PowerShell Cheat Sheet

Wichtige Befehle

Um Hilfe zu irgendeinem *cmdlet* zu bekommen: `get-help`
`Get-Help Get-Service`
Liste aller verfügbaren *cmdlets* : `get-command`
`Get-Command`
Alle Eigenschaften und Methoden eines Objekts: `get-member`
`Get-Service | Get-Member`

Setzen der Security Policy

Lesen und Ändern der Sicherheit durch `Get-ExecutionPolicy` und `Set-ExecutionPolicy`
`Get-ExecutionPolicy`
`Set-ExecutionPolicy remotesigned`

Ausführen eines Skripts

`powershell.exe -noexit &"c:\myscript.ps1"`

Variablen

Fangen mit `$` an
`$a = 32`
Typvorgabe geht:
`[int]$a = 32`

Felder

Initialisierung
`$a = 1,2,4,8`
Abfrage:
`$b = $a[3]`

Funktionen

Parameter durch Leerschlag getrennt.
Rückgabewert ist optional.

```
function sum ([int]$a,[int]$b)
{
    return $a + $b
}
sum 4 5
```

Konstanten

Werden ohne `$` kreiert:
`Set-Variable -name b -value 3.142 -option constant`
Und mit `$` abgefragt:
`$b`

Objekte verwenden

Um eine Instanz eines COM Objekts zu generieren:
`New-Object -comobject <ProgID>`
`$a = New-Object -comobject "wscript.network"`
`$a.username`

Um eine Instanz eines .Net Framework Objekts zu generieren (Parameter können übergeben werden wenn nötig): `New-Object -type <.Net Object>`
`$d = New-Object -Type System.DateTime 2006,12,25`
`$d.get_DayOfWeek()`

Ausgabe an Konsole

Variable Name
`$a`
oder
`Write-Host $a -foregroundcolor "green"`

Benutzereingaben verwenden

`Read-Host` liest Benutzereingaben
`$a = Read-Host "Enter your name"`
`Write-Host "Hello" $a`

Kommandozeilenargumente

Müssen mit Leerschlag übergeben werden
`myscript.ps1 server1 benp`
Im Skript mit Feld `$args` benutzbar
`$servername = $args[0]`
`$username = $args[1]`

Sonstiges

Zeilenumbruch: ``` (Shift ^ + Leerschlag)
`Get-Process | Select-Object name, ID`
Kommentare: `#`
`# code here not executed`
Mehrere Befehle in eine Zeile:
`$a=1;$b=3;$c=9`
`$a=1;$b=3;$c=9`
Ausgabe weitergeben: `|` (AltGr 7)
`Get-Service | Get-Member`

Do While Schleife

Führt Schleife aus SOLANGE Bedingung erfüllt

```
$a=1  
Do {$a; $a++}  
While ($a -lt 10)
```

Do Until Schleife

Führt Schleife aus BIS Bedingung erfüllt

```
$a=1  
Do {$a; $a++}  
Until ($a -gt 10)
```

For Schleife

Befehle wiederholen in definierter Anzahl

```
For ($a=1; $a -le 10; $a++)  
{ $a }
```

ForEach - Schleife bei Gruppe von Objekten

Arbeitet Gruppe von Objekten ab:

```
Foreach ($i in Get-Childitem c:\windows)  
{ $i.name; $i.creationtime }
```

If Bedingung

Führt Code aus bei Erfüllung einer Bedingung

```
$a = "white"  
if ($a -eq "red")  
    {"The colour is red"}  
elseif ($a -eq "white")  
    {"The colour is white"}  
else  
    {"Another colour"}
```

Switch Bedingung

Weitere Möglichkeit, um Code auszuführen bei Erfüllung einer Bedingung

```
$a = "red"  
switch ($a)  
{  
    "red" {"The colour is red"}  
    "white" {"The colour is white"}  
    default {"Another colour"}  
}
```

Aus einer Datei lesen

Get-Content erzeugt ein Feld aus den Zeilen (Objekte!) der Datei. Danach Foreach Schleife verwenden:

```
$a = Get-Content "c:\servers.txt"  
foreach ($i in $a)  
{ $i }
```

In eine einfache Datei schreiben

Verwende Out-File oder > für einfache Textdateien

```
$a = "Hello world"  
$a | out-file test.txt  
Oder > benutzen:  
.\test.ps1 > test.txt
```

In eine HTML Datei schreiben

ConvertTo-Html verwenden UND danach >

```
$a = Get-Process  
$a | Convertto-Html -property Name,Path,Company > test.htm
```

Eine CSV Datei erzeugen

Export-Csv verwenden und Select-Object um Ausgabe zu filtern

```
$a = Get-Process  
$a | Select-Object Name,Path,Company | Export-Csv -path test.csv
```