Das Standard-werk

Für PowerShell 2.0 - 5.0

5.0

PowerShell

Windows-Automation für Einsteiger und Profis

Dr. Tobias Weltner

O'REILLY®

Dr. Tobias Weltner

PowerShell 5.0 – Windows-Automation für Einsteiger und Profis

O'REILLY®

Dr. Tobias Weltner

Lektorat: Ariane Hesse
Korrektorat: Sibylle Feldmann
Satz: mediaService, www.mediaservice.tv
Herstellung: Susanne Bröckelmann
Umschlaggestaltung: Michael Oreal, www.oreal.de, unter Verwendung eines Fotos von Valerie Loiseleux / iStock. by Getty Images
Druck und Bindung: Druckerei C.H. Beck, www.becksche.de

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.

ISBN:
Print 978-3-96009-009-0
PDF 978-3-96010-033-1
ePub 978-3-96010-034-8
mobi 978-3-96010-035-5

1. Auflage 2016

Dieses Buch erscheint in Kooperation mit O'Reilly Media, Inc. unter dem Imprint »O'REILLY«. O'REILLY ist ein Markenzeichen und eine eingetragene Marke von O'Reilly Media, Inc. und wird mit Einwilligung des Eigentümers verwendet.

Wieblinger Weg 17
69123 Heidelberg

5 4 3 2 1 0

Inhalt

Kapitel 4
Anwendungen und Konsolenbefehle

In diesem Kapitel:

Ausführlich werden in diesem Kapitel die folgenden Aspekte erläutert:

- **Anwendungen:** PowerShell kann Anwendungen direkt starten. Zum Start eines Anwendungsprogramms muss der absolute oder relative Pfadname angegeben werden, es sei denn, die Anwendung liegt in einem der Ordner, die in der Umgebungsvariablen `$env:path` festgelegt sind. Steht der Pfadname in Anführungszeichen, muss er mit dem Call-Operator (`&`) aufgerufen werden. Über `Start-Process` lassen sich Anwendungsprogramme außerdem mit vielen zusätzlichen Optionen starten.
- **Konsolenbefehle:** Das Textergebnis von Konsolenbefehlen (z. B. `ipconfig.exe`) kann direkt Variablen zugewiesen werden. Den numerischen »Error Level« des zuletzt ausgeführten Konsolenbefehls findet man in `$LASTEXITCODE`. Liefert ein Konsolenbefehl kommaseparierte Informationen, kann PowerShell diese mit `ConvertFrom-CSV` in strukturierte Objekte verwandeln.
- **Argumente:** Es hängt von der jeweiligen Anwendung ab, ob Benutzerargumente als Gesamttext oder als Array einzelner Texte erwartet werden. Interaktiv erfragte Benutzerein-

gaben können einem Konsolenbefehl auch über die Pipeline übergeben werden, um den Befehl unbeaufsichtigt ausführen zu können.

- **Einschränkungen in der ISE:** Konsolenbefehle, die während der Ausführung Benutzereingaben erfordern, können im ISE-Editor nicht ausgeführt werden (es sei denn, die Argumente werden über die Pipeline übergeben). Deutsche Umlaute und Sonderzeichen gehen bei der Ausgabe von Konsolenbefehlen in der ISE verloren.
- **Fremde Prozesse steuern:** Die Cmdlets aus der Familie `Process` können auf alle laufenden Prozesse zugreifen, ihre Einstellungen ändern, sie beenden oder auch auf die Beendigung der Prozesse warten. Eine Liste der Cmdlets erhält man mit `Get-Command -Noun Process`.

In einer perfekten Welt wären alle Automationsprobleme mit Cmdlets lösbar, und dieses Buch wäre jetzt zu Ende. Ist es aber nicht. Ein kurzer Blick auf die Fülle der noch vor Ihnen liegenden Kapitel nährt den Verdacht, dass Cmdlets allein wohl doch nicht genügen, um die Welt zu retten. Es gibt einfach (noch) nicht genügend davon.

Cmdlets sind deshalb nicht das einzige Mittel, um Aufgaben zu lösen. Eine andere Gruppe von Problemlösern sind die vielfältigen Windows-Programme und Konsolenbefehle wie zum Beispiel *ipconfig.exe*, *robocopy.exe* oder *icacls.exe*, die über viele Jahre in der klassischen Befehlskonsole *cmd.exe* ihren Dienst verrichtet haben – und das auch weiterhin tun. PowerShell diskriminiert solche Befehle nicht, sondern heißt sie willkommen und führt sie gleichberechtigt genau wie Cmdlets aus:

```
PS> ipconfig

Windows-IP-Konfiguration

Ethernet-Adapter Ethernet:

   Verbindungsspezifisches DNS-Suffix: Speedport_W_921V_1_17_000
   Verbindungslokale IPv6-Adresse  . : fe80::e1a2:d0c:f7fc:f49c%12
   IPv4-Adresse  . . . . . . . . . . : 10.0.2.15
   Subnetzmaske  . . . . . . . . . . : 255.255.255.0
   Standardgateway . . . . . . . . . : 10.0.2.2
(…)

PS> ping 127.0.0.1

Ping wird ausgeführt für 127.0.0.1 mit 32 Bytes Daten:
Antwort von 127.0.0.1: Bytes=32 Zeit<1ms TTL=128
Antwort von 127.0.0.1: Bytes=32 Zeit<1ms TTL=128
(…)
```

Die Ergebnisse von Konsolenbefehlen lassen sich genau wie bei Cmdlets in Variablen speichern und auswerten:

```
PS> $info = ipconfig
PS> $info -like '*IPv4*'
   IPv4-Adresse  . . . . . . . . . . : 10.154.240.127
```

Und auch fensterbasierte Anwendungen dürfen ebenso direkt aufgerufen und gestartet werden, wodurch sich Windows-Funktionen, etwa die *Systemsteuerung* oder der *Geräte-Manager* (Abbildung 4.1), von PowerShell aus ohne Umwege durch verschlungene Menüs direkt öffnen lassen (immer vorausgesetzt, man weiß, wie der passende Befehl heißt):

```
PS> notepad
PS> control
PS> devmgmt
PS> wscui
PS> lpksetup
```

Programme starten

PowerShell startet externe Programme unbürokratisch, wenn Sie den Namen des Programms angeben:

```
PS> regedit
PS> tracert www.microsoft.com
PS> driverquery
```

Handelt es sich um eine Windows-Anwendung, öffnet sie ihr Fenster, und PowerShell setzt einfach seine Arbeit fort. Ist es dagegen eine Konsolenanwendung, teilt sie sich das Ausgabefenster mit PowerShell, und PowerShell wartet, bis die Konsolenanwendung wieder beendet ist.

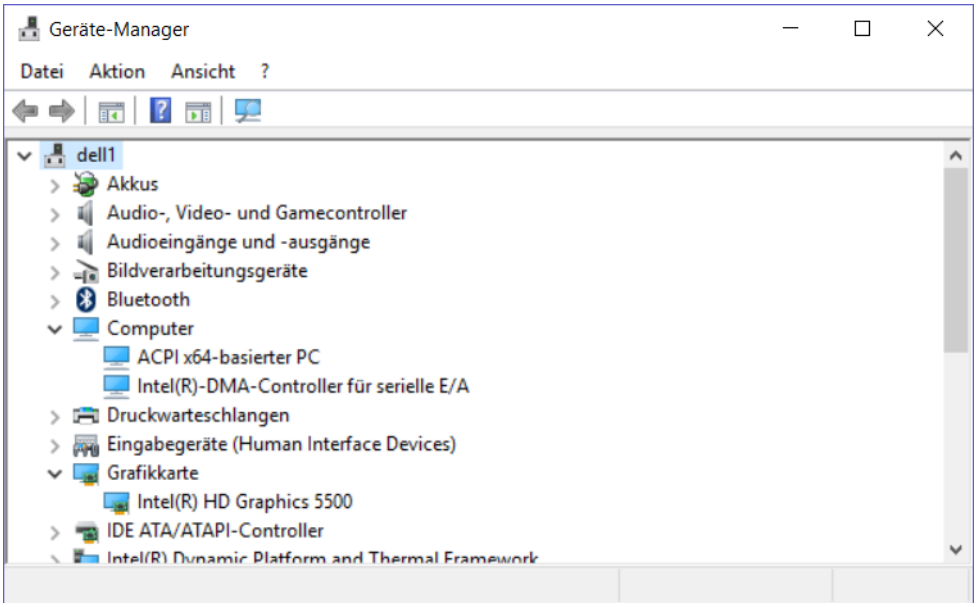

Abbildung 4.1: Mit Befehlen wie »devmgmt« einen schnellen Zugriff auf Systemdialoge des Betriebssystems erhalten.

Manche Programme lassen sich von PowerShell überraschenderweise jedoch nicht starten, obwohl sie nachweislich vorhanden sind:

```
PS> iexplore

iexplore : Die Benennung "iexplore" wurde nicht als Name eines Cmdlet, einer Funktion, einer
Skriptdatei oder eines ausführbaren Programms erkannt. Überprüfen Sie die Schreibweise des Namens,
oder ob der Pfad korrekt ist (sofern enthalten), und wiederholen Sie den Vorgang.
(…)
```

Damit PowerShell ein Programm finden kann, muss es sich in einem derjenigen Ordner befinden, die in der Umgebungsvariablen `$env:Path` aufgelistet sind. Nur die durchsucht PowerShell automatisch:

```
PS> $env:Path -split ';'
```

Liegt das Programm woanders, müssen Sie PowerShell schon verraten, wo genau. Dazu geben Sie den absoluten oder relativen Pfadnamen an, zum Beispiel so:

```
PS> & 'C:\Program Files\Internet Explorer\iexplore.exe'
PS> & 'C:\Program Files\Internet Explorer\iexplore.exe' www.powertheshell.com
```

Tipp

Nutzen Sie die Autovervollständigung, um Pfadnamen einzugeben:

```
PS> c:\pro⇆
PS> & 'C:\Program Files'\int⇆
PS> & 'C:\Program Files\Internet Explorer'\iexp⇆
PS> & 'C:\Program Files\Internet Explorer\iexplore.exe'
```

Die Autovervollständigung findet nicht nur die Pfadbestandteile (drücken Sie mehrmals ⇆, um alle Auswahlmöglichkeiten zu sehen), sie achtet auch automatisch darauf, Pfadnamen in Anführungszeichen zu setzen, wenn darin Sonderzeichen wie Leerzeichen vorkommen. Weil ein Pfadname in Anführungszeichen zu reinem Text wird, würde PowerShell ihn nun allerdings nicht mehr als Befehl verstehen und einfach den Text ausgeben:

```
PS> 'C:\Program Files\Internet Explorer\iexplore.exe'
C:\Program Files\Internet Explorer\iexplore.exe
```

Deshalb stellt die Autovervollständigung außerdem noch den Call-Operator `&` vor den Text, wie etwas weiter oben zu sehen. Er sorgt dafür, dass der Text von PowerShell als Befehl verstanden wird – als hätten Sie ihn direkt eingegeben. Können Sie nachvollziehen, was in diesem (zugegebenermaßen leicht skurrilen) Beispiel geschieht?

```
PS> $a = 'not'
PS> $b = 'epa'
PS> $c = 'D'
PS> & "$a$b$c"
```

Geben Sie einfach den Text ohne den Call-Operator aus. Dann wird sicher klarer, warum PowerShell den Windows-Editor gestartet hat:

```
PS> "$a$b$c"
notepaD
```

Wird ein Text in doppelte Anführungszeichen gefasst, ersetzt PowerShell alle darin vorkommenden Variablen durch ihren Inhalt. Die einzelnen Textbruchstücke werden so zu `notepaD` zusammengefügt, und der Call-Operator führt diesen Befehl aus. Die Groß-/Kleinschreibung wird von PowerShell dabei grundsätzlich ignoriert.

Auf Dauer ist die Eingabe langer Pfadnamen natürlich keine Lösung. Einfacher geht es auf eine der folgenden Arten: Sie könnten den Pfadnamen des Programms beispielsweise in einer eigenen Variablen speichern und diese dann mit dem Call-Operator aufrufen:

```
PS> $ie = 'C:\Program Files\Internet Explorer\iexplore.exe'
PS> & $ie www.powertheshell.com
```

Oder Sie legen einen neuen Alias auf den Programmpfad an:

```
PS> Set-Alias -Name ie -Value 'C:\Program Files\Internet Explorer\iexplore.exe'
PS> ie www.powertheshell.com
```

Schließlich könnten Sie auch den Ordner, in dem sich das Programm befindet, in die Umgebungsvariable `$env:Path` aufnehmen:

```
PS> $env:Path += ';C:\Program Files\Internet Explorer\'
PS> iexplore www.powertheshell.com
```

Alle drei Varianten – Variable, Alias und Umgebungsvariable – wirken sich allerdings nur in der aktuellen PowerShell-Sitzung aus. Wer länger etwas von diesen Änderungen haben möchte, sollte sie im Rahmen eines Profilskripts ausführen.

Optionen für den Programmstart festlegen

Haben Sie besondere Wünsche für den Programmstart, dann greifen Sie zu `Start-Process` und starten das Programm mit diesem Cmdlet. Es liefert viele optionale Parameter, mit denen Sie Sonderwünsche festlegen können.

Warten, bis ein Programm wieder beendet ist

Die folgende Zeile öffnet den Windows-Editor *synchron*. PowerShell wartet also so lange, bis der Editor geschlossen wird, bevor der Befehlsprompt zurückkehrt:

```
PS> Start-Process -FilePath notepad -Wait
```

Bei Konsolenanwendungen wartet PowerShell normalerweise ohnehin, bis der Konsolenbefehl seine Arbeit erledigt hat. Möchten Sie einen Konsolenbefehl *asynchron* ausführen, ihn also in seinem eigenen Fenster sich selbst überlassen und nicht auf ihn warten, gehen Sie folgendermaßen vor:

```
PS> Start-Process -FilePath systeminfo
```

Ein zweites Konsolenfenster öffnet sich, und darin wird der Befehl `systeminfo` parallel zu PowerShell ausgeführt. Sobald `systeminfo` fertig ist, schließt sich das Fenster – zusammen mit allen Hoffnungen, an die Resultate des Befehls zu gelangen. Die sind jetzt nämlich ebenfalls weg. Konsolenbefehle sollten also nur dann in einem Extrafenster parallel ausgeführt werden, wenn sie lediglich etwas eigenverantwortlich erledigen sollen, aber keine Ergebnisse an PowerShell zurückliefern müssen.

Programme unter anderem Benutzernamen ausführen

Wollen Sie ein Programm im Namen eines anderen Benutzers ausführen, greifen Sie zu `-Credential`. Die folgende Zeile startet den Windows-Editor als Benutzer *testfirma/testuser*:

```
PS> Start-Process -FilePath notepad.exe -WorkingDirectory C:\ -Credential testfirma/testuser
-LoadUserProfile
```

Ein Anmeldedialog erscheint, in den das passende Kennwort eingegeben wird. Danach startet Notepad unter dem Namen des angegebenen Benutzers.

Profitipp

Wann immer Sie `Start-Process` mit `-Credential` einsetzen, werden zwei andere Parameter essenziell: `-LoadUserProfile` lädt zusätzlich das Benutzerprofil des angegebenen Anwenders. Ohne das Benutzerprofil funktionieren manche Programme nicht. Außerdem legt `-WorkingDirectory` fest, in welchem Ordner das Programm startet. Wählen Sie einen Ordner aus, auf den der angegebene Benutzer auch tatsächlich Zugriffsrechte hat. Andernfalls wird nämlich Ihr augenblicklicher Ordner als Arbeitsverzeichnis verwendet, und die Chancen stehen sehr gut, dass der Anwender darauf nun keinerlei Zugriffsrechte hat oder das Laufwerk dieses Ordners noch nicht einmal sieht (falls es ein persönliches Netzwerklaufwerk ist). In beiden Fällen würde der Aufruf scheitern, und das Programm könnte nicht gestartet werden.

`Start-Process` bietet noch viele weitere Parameter, mit denen Sie zum Beispiel kontrollieren, wie eine Windows-Anwendung ihr Fenster anzeigt und ob die Anwendung Administratorrechte anfordern soll (Abbildung 4.2). Diese Zeile startet den Windows-Editor in einem maximierten Fenster mit Administratorrechten:

```
PS> Start-Process -FilePath Notepad -WindowStyle Maximized -Verb Runas
```

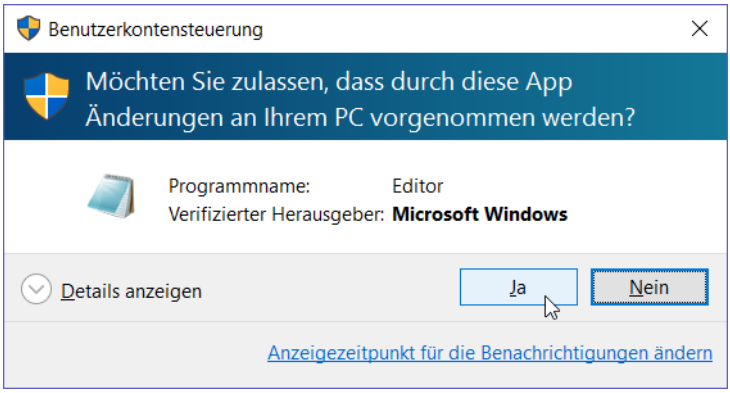

Abbildung 4.2: Programme von PowerShell aus mit vollen Administratorrechten starten.

`Start-Process` kann Ihnen mit `-PassThru` auch das Prozessobjekt zurückliefern, sodass Sie die Kontrolle über den gestarteten Prozess behalten und ihn später zum Beispiel jederzeit wieder schließen könnten. Diese Zeilen öffnen Notepad für genau fünf Sekunden und schließen es dann wieder:

```
PS> $prozess = Start-Process -FilePath Notepad -PassThru
PS> Start-Sleep -Seconds 5
PS> Stop-Process -InputObject $prozess
```

Nicht unterstützte Konsolenbefehle im ISE-Editor

Die meisten Konsolenbefehle verrichten ohne weitere Rückfragen ihren Dienst. Solche Konsolenbefehle können im ISE-Editor problemlos ausgeführt werden. Sobald ein Konsolenbefehl allerdings Rückfragen stellt, ergibt sich in der ISE ein Problem. Die ISE ist keine Konsolenanwendung. Damit sie dennoch Konsolenanwendungen ausführen kann, hält sie sich ein verstecktes Konsolenfenster, das man höchstens bei der Ausführung des ersten Konsolenbefehls kurz aufflackern sieht, bevor die ISE es sofort wieder versteckt.

Konsolenbefehle werden darin ausgeführt. Ihre Ergebnisse werden automatisch in den Vordergrund transportiert und in der ISE angezeigt. Dieser Mechanismus erlaubt aber keine interaktiven Ein- und Ausgaben. Die ISE kann also keine spontanen Tastatureingaben des Anwenders an die versteckte Konsole durchleiten und stellt auch keine Hinweismeldungen eines Konsolenprogramms dar.

In der Praxis ist das kein allzu großes Problem, weil interaktive Konsolenbefehle selten sind. Starten Sie dennoch einen in der ISE – beispielsweise `choice.exe` –, bleibt die ISE hängen. Der Konsolenbefehl wartet vergeblich auf die erwarteten Anwendereingaben. Führen Sie `choice.exe` dagegen in der PowerShell-Konsole aus, funktioniert alles einwandfrei.

Damit Sie in der ISE möglichst nicht in solch unangenehme Situationen geraten, unterhält diese eine Sperrliste. Programme, die in diese Sperrliste eingetragen sind, können in der ISE nur gestartet werden, wenn ihnen Argumente mitgegeben werden. Ohne Argumente – hier unterstellt die ISE dann einen interaktiven Aufruf – führt der Start zu einer Fehlermeldung. `$PSUnsupportedConsoleApplications` enthält diese Sperrliste:

```
PS> $psUnsupportedConsoleApplications
wmic
wmic.exe
cmd
cmd.exe
diskpart
diskpart.exe
edit.com
netsh
netsh.exe
nslookup
nslookup.exe
PowerShell
PowerShell.exe
```

Ohne weitere Argumente würde `nslookup` zu einer Fehlermeldung führen, denn ohne Argumente würde dieser Konsolenbefehl tatsächlich interaktiv nach Aufträgen fragen. Dasselbe gilt für Konsolenbefehle wie `wmic` oder `cmd`:

```
PS> nslookup

"nslookup" kann nicht gestartet werden. Interaktive Konsolenanwendungen werden nicht unterstützt.
Verwenden Sie das Start-Process-Cmdlet oder "PowerShell.exe starten" im Menü "Datei" zum Ausführen
der Anwendung. Verwenden Sie $psUnsupportedConsoleApplications zum Anzeigen/Ändern der Liste
blockierter Konsolenanwendungen, oder rufen Sie die Onlinehilfe auf.
```

```
PS> wmic

"wmic" kann nicht gestartet werden. Interaktive Konsolenanwendungen werden nicht unterstützt.
Verwenden (…)

PS> cmd

"cmd" kann nicht gestartet werden. Interaktive Konsolenanwendungen werden nicht unterstützt.
Verwenden (…)
```

Mit Argumenten aufgerufen, funktionieren die gleichen Konsolenbefehle hingegen einwandfrei auch in der ISE, weil dann keine interaktiven Eingaben nötig sind. Die notwendigen Eingaben wurden nun ja als Argument übergeben:

```
PS> nslookup www.powertheshell.com

Server:  speedport.ip
Address:  192.168.2.1

Name:    www.powertheshell.com
Address:  173.254.71.70
PS> wmic os get version
Version

6.2.9200
PS> cmd.exe /c dir %WINDIR%
(…)
```

Dieses Konzept ist allerdings nur ein Workaround. Erstens ist die Sperrliste niemals vollständig (sie enthielt ja beispielsweise nicht `choice.exe`), und zweitens können auch Konsolenbefehle, die mit Argumenten aufgerufen werden, nachträglich interaktiv werden – und dann in der ISE für sonderbare Situationen sorgen.

Die Sperrliste lässt sich leicht ergänzen, wenn Sie finden, dass weitere Konsolenbefehle ausgeschlossen gehören (speichern Sie diese Anweisungen in Ihrem Profilskript, wenn sie dauerhaft wirken sollen):

```
PS> $psUnsupportedConsoleApplications.Add('choice')
PS> $psUnsupportedConsoleApplications.Add('choice.exe')
```

Problematischer schon sind Konsolenbefehle, die nachträglich – oder nur gelegentlich – interaktiv nachfragen. Rufen Sie beispielsweise `systeminfo.exe` auf, gelingt dies lokal einwandfrei. Sie würden zwar die Statusmeldungen des Konsolenbefehls nicht sehen (führen Sie den Befehl zum Vergleich in der PowerShell-Konsole aus, um den Unterschied zu erleben), aber die Ergebnisse erscheinen wie erwartet trotzdem.

```
PS> systeminfo.exe
```

Auch remote könnten Sie `systeminfo.exe` aufrufen, jedenfalls dann, wenn das Zielsystem erreichbar ist und Sie darauf Administratorrechte besitzen:

```
PS> systeminfo.exe /S testserver
```

Haben Sie indes keine Administratorrechte, würde `systeminfo.exe` nun nach Ihrem Benutzernamen und/oder Kennwort fragen. Der Befehl würde also plötzlich nachträglich interaktiv, und weil die ISE weder die Frage nach dem Kennwort anzeigt noch etwaige Eingaben Ihrerseits an den Befehl zurückmeldet, scheint alles so, als würde die ISE hängen.

Führen Sie hier erneut den Befehl zum Vergleich in der PowerShell-Konsole aus. Und genau das ist auch der allgemeine Ratschlag, falls ein Konsolenbefehl in der ISE nicht wie geplant funktioniert: Testen Sie den Aufruf in einer PowerShell-Konsole, um zu prüfen, ob interaktive Ein- oder Ausgabewünsche zum Problem geführt haben.

Profitipp

Die besondere Architektur der ISE mit der versteckten Konsole ist der Grund für ein weiteres Phänomen: Liefert ein Konsolenbefehl deutsche Umlaute oder andere Sonderzeichen zurück, fehlen diese in der ISE mitunter oder werden durch falsche Zeichen ersetzt. Schuld ist hier das Encoding, mit dem die ISE die Ergebnisse von der versteckten Konsole in die eigene Anwendung kopiert.

Wenn Sie in der ISE beispielsweise die folgende Zeile ausführen, erscheinen die Betriebssysteminformationen im GridView, aber Umlaute werden durch fehlerhafte Zeichen ersetzt:

```
PS> systeminfo.exe /FO CSV | ConvertFrom-CSV | Out-GridView
```

Mit einem kleinen Trick kann das Problem behoben werden: Zunächst wird die ISE mit einem einfachen Konsolenbefehl gezwungen, die versteckte Konsole anzulegen, sollte sie noch nicht vorhanden sein. Danach wird das Encoding der versteckten Konsole so geändert, dass deutsche Umlaute korrekt angezeigt werden. Nun funktioniert `systeminfo.exe` einwandfrei auch mit deutschen Umlauten:

```
# sicherstellen, dass eine versteckte ISE-Konsole vorhanden ist:
$null = cmd.exe /c echo

# Konsolen-Encoding korrigieren:
[Console]::OutputEncoding = [System.Text.Encoding]::UTF8

# deutsche Umlaute erscheinen korrekt:
systeminfo.exe /FO CSV | ConvertFrom-CSV | Out-GridView
```

Listing 4.1: ISE-Konsolen-Encoding für deutsche Sonderzeichen einstellen.

Argumente an Anwendungen übergeben

Auch Anwendungen – insbesondere aber Konsolenbefehle – akzeptieren Argumente, mit denen Sie ähnlich wie mit Cmdlets Wünsche an den Befehl übermitteln. Welche Argumente eine Anwendung unterstützt, weiß nur die Anwendung selbst.

Hilfe für Konsolenbefehle anzeigen

Die allermeisten Anwendungen unterstützen den Parameter /?, mit dem man sich die unterstützten Parameter anzeigen lassen kann. Schauen Sie sich beispielsweise den Konsolenbefehl `systeminfo.exe` an, der Teil von Windows ist:

```
PS> systeminfo /?

SYSTEMINFO [/S System [/U Benutzername [/P [Kennwort]]]] [/FO Format] [/NH]

Beschreibung:
    Mit diesem Programm wird die Betriebssystemkonfiguration für
```

```
    einen lokalen bzw. Remotecomputer, inklusive Service Packs, angezeigt.

Parameterliste:
    /S      System           Bestimmt das Remotesystem mit dem die Verbindung
                             hergestellt werden soll.
    /U      [Domäne\]Benutzer Bestimmt den Benutzerkontext unter dem
                             der Befehl ausgeführt werden soll.
    /P      [Kennwort]       Bestimmt das Kennwort für den zugewiesenen
                             Benutzerkontext. Bei Auslassung, wird dieses
                             angefordert.
    /FO     format           Bestimmt das Format in dem die Ausgabe
                             angezeigt werden soll.
                             Gültige Werte: "TABLE", "LIST", "CSV".
    /NH                      Bestimmt, dass der "Spalten-Header" in der
                             Ausgabe nicht angezeigt werden soll.
                             Nur für Formate TABLE und CSV.
    /?                       Zeigt diese Hilfe an.

Beispiele:
    SYSTEMINFO
    SYSTEMINFO /?
    SYSTEMINFO /S System
    SYSTEMINFO /S System /U Benutzer
    SYSTEMINFO /S System /U Domäne\Benutzer /P Kennwort /FO TABLE
    SYSTEMINFO /S System /FO LIST
    SYSTEMINFO /S System /FO CSV /NH
```

Die Beispiele am Ende des Hilfetexts können genau so wie angegeben in PowerShell verwendet werden. Das ist allerdings nicht immer der Fall. Welche Fallstricke es bei der Angabe von Argumenten gibt, schauen wir uns als Nächstes an.

Beispiel: Lizenzstatus von Windows überprüfen

Auch viele Skripte geben mit dem Parameter /? Hilfestellung oder zeigen automatisch Informationen über die vorrätigen Parameter aus, wenn beim Aufruf falsche oder keine Parameter angegeben wurden. Hinter `slmgr` verbirgt sich zum Beispiel ein VBScript, das Teil von Windows ist und die Windows-Lizenzen verwaltet:

```
PS> Get-Command -Name slmgr

CommandType     Name                                               ModuleName
-----------     ----                                               ----------
Application     slmgr.vbs

PS> slmgr

Ungültige Kombination von Befehlszeilenparametern.

Windows-Software-Lizenzverwaltungstool
Syntax: slmgr.vbs [Computername [Benutzerkennwort]] [<Option>]
           Computername: Name des Remotecomputers (Standard: lokaler Computer)
           Benutzer:     Konto mit erforderlichen Rechten für Remotecomputer
           Kennwort:     Kennwort für das vorherige Konto

Globale Optionen:
/ipk <Product Key>
    Product Key installieren (ersetzt den vorhandenen Key)
```

```
/ato [Aktivierungs-ID]
    Windows aktivieren
/dli [Aktivierungs-ID | All]
    Lizenzinformationen anzeigen (Standard: aktuelle Lizenz)
/dlv [Aktivierungs-ID | All]
    Detaillierte Lizenzinformationen anzeigen (Standard: aktuelle Lizenz)
/xpr [Aktivierungs-ID]
    Ablaufdatum für aktuellen Lizenzstatus

Erweiterte Optionen:
/cpky
    Product Key aus Registrierung löschen (verhindert Offenlegungsangriffe)
(…)

PS> slmgr /dli

Name: Windows(R), Professional edition
Beschreibung: Windows(R) Operating System, RETAIL channel
Teil-Product Key: HMFDH
Lizenzstatus: Lizenziert
```

Falls `slmgr` seine Hilfetexte nicht in die Konsole ausgibt, sondern als Extrafenster anzeigt, liegt das an der Festlegung des Programms, das für VBScript zuständig ist und diese Skripte ausführt (Abbildung 4.3). Als Vorgabe ist dies nämlich *wscript.exe*, also der fensterbasierte Script Host von VBScript. Hier erscheinen mangels Konsole alle Ausgaben als Fenster, was auf Dauer reichlich lästig ist und zudem einer engeren Zusammenarbeit zwischen VBScript und PowerShell im Weg steht.

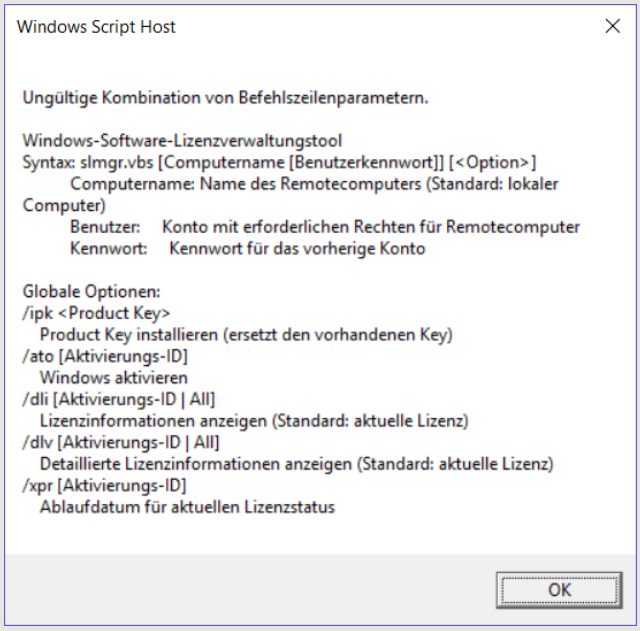

Abbildung 4.3: Wenn VBScripts Meldungen in Fenstern anzeigen, fehlt eine wichtige Grundeinstellung.

Besser ist es, VBScript mit dem konsolenbasierten Scripthost *cscript.exe* zu assoziieren, wozu nur ein einziger Befehlsaufruf nötig ist (Administratorrechte vorausgesetzt):

```
PS> wscript //H:Cscript
```

Jetzt landen die Ausgaben des VBScript in der PowerShell-Konsole. Damit *cscript.exe* auch noch von der Anzeige der störenden Copyright-Meldung absieht, schicken Sie diesen Befehl hinterher:

```
PS> wscript
```

Ein Dialogfeld öffnet sich, in dem Sie das Kontrollkästchen *Logo anzeigen ...* deaktivieren und auf *OK* klicken. Diese Einstellungen gelten übrigens dauerhaft, das heißt so lange, bis Sie mit `wscript //H:Wscript` wieder zum alternativen ursprünglichen VBScript-Host zurückschalten.

Anders als bei Konsolenanwendungen und Skripten sind die unterstützten Argumente bei Windows-Anwendungen meist un(ter)dokumentiert und erfordern etwas Kaffee und engagierte Google-Recherche im Internet.

Argumentübergabe kann scheitern

Leider kommen Ihre Argumente nicht immer unbeschadet bei der Anwendung an, weil der PowerShell-Parser die Eingabe zuerst erhält. Erst wenn er die Eingabe begutachtet hat, leitet er sie nach eigenem Ermessen an die Anwendung. Dabei kann es zu Missverständnissen kommen, zum Beispiel wenn Ihre Argumente Sonderzeichen enthalten, die bei PowerShell eine besondere Bedeutung haben. Möchten Sie zum Beispiel den Windows-Explorer beauftragen, einen bestimmten Ordner anzuzeigen und darin eine Datei zu markieren, funktioniert in der klassischen Eingabeaufforderung `cmd.exe` (der aus dem *Ausführen*-Dialog, den Sie mit ⊞+R öffnen) dieser Befehl ganz ausgezeichnet:

```
explorer /select,c:\windows\system32\calc.exe
```

Geben Sie dagegen den gleichen Befehl in PowerShell ein, öffnet sich zwar auch der Windows-Explorer, er zeigt aber weder den angegebenen Ordner an, noch wird darin irgendeine Datei markiert:

```
PS> explorer /select,c:\windows\system32\calc.exe
```

Hinweis

Dieses Problem ist in PowerShell 5/Windows 10 inzwischen behoben.

Für den Parser ist alles, was Sie eingeben, PowerShell-Code. Das Komma legt bei PowerShell stets ein Array an. In Wirklichkeit wird `explorer.exe` also ein Textarray mit zwei Elementen übergeben, und weil der Windows-Explorer nur ein Argument erwartet, präsentiert er seine Standardansicht. Häufig kann man solche Probleme schon dadurch lösen, dass man die Argumente vom Parser fernhält, indem man sie in einfache Anführungszeichen setzt (und damit zu statischem Text macht, dessen Inhalt der Parser nicht anrührt):

```
PS> explorer '/select,c:\windows\system32\calc.exe'
```

Noch ein Weg, den Parser auszuschalten, ist `Start-Process`, mit dem die Argumente für ein Programm über einen separaten Parameter angegeben werden können. Auch hier werden die Argumente dann als Text übergeben:

```
PS> Start-Process -FilePath explorer.exe -ArgumentList '/select,c:\windows\system32\calc.exe'
```

Schließlich kann man den Parser auch ausdrücklich anweisen, die Finger vom Code zu lassen, indem man (ab PowerShell 3.0) den besonderen Parameter `--%` einsetzt. Sobald der Parser auf diesen Parameter trifft, ignoriert er den Rest der Zeile und verarbeitet den Teil so, wie er ist:

```
PS> explorer --% /select,c:\windows\system32\calc.exe
```

Weil das so ist, dürfen Sie nun allerdings in dem Teil nach `--%` keine Variablen mehr verwenden, denn dieser Teil wird jetzt konsequent wörtlich verstanden. Wer beispielsweise die Datei *PowerShell.exe* im Windows-Explorer hervorheben möchte, kommt nicht mehr auf diese Weise zum Ziel:

```
PS> explorer --% /select,$PSHOME\PowerShell.exe
```

Doppelte Anführungszeichen funktionieren dagegen:

```
PS> explorer "/select,$PSHOME\PowerShell.exe"
```

Welche Verpackungsart die beste ist, hängt vom jeweiligen Fall ab.

Texteingaben an Konsolenbefehle senden

Manche Konsolenbefehle erwarten zur Bestätigung bestimmte Tastendrücke oder Eingaben und können deshalb schlecht oder gar nicht unbeaufsichtigt eingesetzt werden. Der Befehl `format.com` zum Formatieren eines Laufwerks gehört dazu.

Achtung

Das Formatieren eines Laufwerks ist nicht gerade eine beiläufige Angelegenheit, und formatiert man aus Versehen das falsche Laufwerk, ist der Abend gelaufen. Für Automationslösungen gilt ganz besondere Vorsicht. Setzen Sie daher die eigentlich sinnvollen Sicherheitsabfragen nicht ohne Not außer Kraft und fragen Sie sich gegebenenfalls lieber, ob die eine oder andere Aufgabe überhaupt vollautomatisch durchgeführt werden sollte. Die folgenden Beispiele formatieren das Laufwerk *I:* mit der Laufwerkbezeichnung *Volume*. Passen Sie die Angaben gegebenenfalls an Ihre Umgebung an. Achten Sie aber insbesondere darauf, dass sich auf dem Laufwerk keine wichtigen Daten befinden, denn die werden gleich gelöscht.

```
format i: /FS:NTFS /Q
Der Typ des Dateisystems ist NTFS.
Geben Sie die aktuelle Volumebezeichnung für Laufwerk I: ein:
```

Zunächst werden Sie aus Sicherheitsgründen aufgefordert, manuell die aktuelle Laufwerkbezeichnung des Laufwerks einzugeben, um abzusichern, dass Sie das richtige Laufwerk meinen. Die Laufwerkbezeichnung eines Laufwerks wird im Windows-Explorer neben dem Laufwerkbuchstaben genannt. Im folgenden Beispiel heißt die Datenträgerbezeichnung *Volume*.

Um diese Eingabe automatisiert vorzunehmen, legt man den Eingabetext vor Aufruf des Befehls in die Pipeline. So landet der Text im Tastaturpuffer. Sobald ein Konsolenbefehl eine Frage hat, schaut dieser in den Tastaturpuffer, und wenn dort schon etwas liegt, wird dieser Text als Eingabe akzeptiert.

```
"Volume" | format i: /FS:NTFS /Q
Der Typ des Dateisystems ist NTFS.
Geben Sie die aktuelle Volumebezeichnung für Laufwerk I: ein:
ACHTUNG: ALLE DATEN AUF DEM
FESTPLATTENLAUFWERK I: GEHEN VERLOREN!
Formatierung durchführen (J/N)?
ACHTUNG: ALLE DATEN AUF DEM
FESTPLATTENLAUFWERK I: GEHEN VERLOREN!
Formatierung durchführen (J/N)? PS>
```

Der Befehl akzeptiert die mitgelieferte Laufwerkbezeichnung zwar, doch anschließend folgt eine weitere Sicherheitsabfrage, bei der Sie `J` eingeben müssen, damit die Formatierung tatsächlich gestartet wird. Die PowerShell-Pipeline kann beliebig viele Zusatzinformationen an den folgenden Befehl liefern. Wie das geschieht, haben Sie in den vorangegangenen Beispielen schon gesehen: Verwenden Sie ein Komma, um die Einzelinformationen in einem Array zu verpacken:

```
"Volume", "J" | Format i: /FS:NTFS /Q
Der Typ des Dateisystems ist NTFS.
Geben Sie die aktuelle Volumebezeichnung für Laufwerk I: ein:
ACHTUNG: ALLE DATEN AUF DEM
FESTPLATTENLAUFWERK I: GEHEN VERLOREN!
Formatierung durchführen (J/N)? Formatieren mit Schnellformatierung 14999 MB
Volumebezeichnung (32 Zeichen, EINGABETASTE für keine)? Struktur des Dateisystems wird erstellt.
Formatieren beendet.
      14,6 GB Speicherplatz insgesamt.
      14,6 GB sind verfügbar.
```

Das Beispiellaufwerk *I:* wurde nun erfolgreich unbeaufsichtigt formatiert, weil die erforderlichen Bestätigungen vorab in die Pipeline gelegt und so an den folgenden Befehl weitergereicht wurden.

Profitipp

Die Automation von Konsolenanwendungen wie `format.com` über die PowerShell-Pipeline ist nur ein Beispiel für ihre Flexibilität, nicht aber unbedingt in jedem Szenario der sinnvollste Weg. Zwar können Sie über die Pipeline Informationen an native Konsolenanwendungen weiterreichen, haben aber keinen Einfluss darauf, ob und wie diese Informationen vom Befehl weiterverarbeitet werden. Reagiert dieser anders als geplant und erfordert andere Eingaben, kann der Aufruf scheitern. Auf einem englischen System würde `format.com` beispielsweise zur Bestätigung nicht `J`, sondern `Y` erwarten.

Ergebnisse von Anwendungen weiterverarbeiten

Einzelne externe Programme aufzurufen, kann allein für sich schon durchaus nützlich sein, aber wenn Sie externe Programme in Skriptlösungen einbetten wollen, haben Sie vielleicht auch Interesse daran, die Ergebnisse dieser Programme in PowerShell zu empfangen und dort sinnvoll weiterzuverarbeiten.

Error Level auswerten

Konsolenbasierte Programme liefern meist einen numerischen Rückgabewert, den Error Level (Fehlerstufe). Was die zurückgemeldete Zahl bedeutet, bestimmt natürlich der Autor des Programms, und PowerShell liefert diese Zahl in der Variablen `$LASTEXITCODE` zur weiteren Auswertung an den Aufrufer – also Sie – zurück.

Möchten Sie zum Beispiel herausfinden, ob eine bestimmte IP-Adresse oder Webseite in Ihrem Netzwerk erreichbar ist, können Sie diese Adresse mit `ping.exe` »anpingen«, was man sich ein wenig so vorstellen kann wie das Echolot aus der Schifffahrt, mit dem sich zum Beispiel U-Boote orten lassen. Wird das ausgesendete Signal an der angegebenen IP-Adresse »reflektiert« und kommt zu Ihnen zurück, wissen Sie nicht nur, dass es die IP-Adresse gibt, sondern auch, wie lange das Signal für die Reise gebraucht hat (im Gegensatz zu U-Booten können Sie daraus allerdings nicht die Entfernung des Remotecomputers ableiten, sondern höchstens die Qualität und Übertragungsgeschwindigkeit Ihres Netzwerks):

```
PS> ping www.tagesschau.de

Ping wird ausgeführt für a1838.g.akamai.net [62.154.232.146] mit 32 Bytes Daten:
Antwort von 62.154.232.146: Bytes=32 Zeit=44ms TTL=60
Antwort von 62.154.232.146: Bytes=32 Zeit=34ms TTL=60
Antwort von 62.154.232.146: Bytes=32 Zeit=30ms TTL=60
Antwort von 62.154.232.146: Bytes=32 Zeit=29ms TTL=60

Ping-Statistik für 62.154.232.146:
    Pakete: Gesendet = 4, Empfangen = 4, Verloren = 0
    (0% Verlust),
Ca. Zeitangaben in Millisek.:
    Minimum = 29ms, Maximum = 44ms, Mittelwert = 34ms
```

Zwar könnten Sie den von `ping` gelieferten Text nun untersuchen und daraus entnehmen, ob die angegebene Adresse erreichbar ist oder nicht. Einfacher ist es allerdings häufig, den (normalerweise unsichtbaren) numerischen Rückgabewert des Konsolenbefehls zurate zu ziehen. Bei `ping` lautet er 0, falls eine Antwort empfangen wurde, ansonsten 1.

```
PS> $LASTEXITCODE
0
```

Sind Sie nur am Rückgabewert eines Befehls interessiert, nicht aber an seiner Textausgabe, können Sie diese zum Beispiel an die besondere Variable `$null` weiterleiten, die alles, was man ihr übergibt, sofort wieder vergisst:

```
PS> ping.exe 10.10.10.10 -n 1 -w 500 > $null
PS> "Antwort erhalten (0) oder nicht (1): $LASTEXITCODE"
```

Wirkliche Begeisterungsstürme wird das allein noch nicht auslösen, denn noch fehlen Ihnen die Mittel, um daraus Hunderte oder Tausende von Webseiten oder IP-Adressen automatisiert anzupingen. Auch die Aussagekraft des Rückgabewerts ist nur so gut wie der Befehl, von dem er stammt, denn `ping` meldet auch dann freudig eine empfangene Antwort, wenn diese gar nicht vom adressierten Computer stammt, sondern lediglich von einem Router, der meldet, dass die IP-Adresse nicht in seinem Einzugsgebiet liegt:

```
PS> ping 169.254.1.2

Ping wird ausgeführt für 169.254.1.2 mit 32 Bytes Daten:
Antwort von 10.0.2.15: Zielhost nicht erreichbar.
```

```
(…)
Ping-Statistik für 169.254.1.2:
    Pakete: Gesendet = 4, Empfangen = 4, Verloren = 0
    (0% Verlust),

PS> $LASTEXITCODE
0
```

Außerdem verwenden viele Computer taktische Tarnkappen und antworten erst gar nicht auf den ausgesendeten Ping, um potenziellen Hausierern die Geschäftsgrundlage zu entziehen:

```
PS> ping www.microsoft.com

Ping wird ausgeführt für lb1.www.ms.akadns.net [64.4.11.42] mit 32 Bytes Daten:
Zeitüberschreitung der Anforderung.
(…)
Ping-Statistik für 64.4.11.42:
    Pakete: Gesendet = 4, Empfangen = 0, Verloren = 4
    (100% Verlust),

PS> $LASTEXITCODE
1
```

Fragen an Benutzer stellen mit choice.exe

Ob der numerische Rückgabewert eines Konsolenbefehls Ihnen helfen kann, ist also eine Einzelfallentscheidung. Nützlich ist er zum Beispiel bei `choice.exe`, einem (interaktiven) Konsolenbefehl, der dem Anwender Fragen stellt. Über dessen Parameter `/?` erhalten Sie eine Übersicht seiner Parameter (Abbildung 4.4).

Achtung

Erinnern Sie sich? Weil `choice.exe` interaktiv arbeitet, funktioniert er leider nicht in der ISE und kann nur in der PowerShell-Konsole eingesetzt werden.

Der folgende Aufruf fragt den Anwender etwa, ob er den Computer neu starten möchte (`/C` legt die erlaubten Antworten fest und `/M` die Frage an den Anwender), und gibt ihm für die Entscheidungsfindung 10 Sekunden Zeit (`/T`). Genügt das nicht, um den Anwender zu einer Reaktion zu bewegen – antwortet er also nicht –, wird die Default-Antwort (`/D`) genommen, in diesem Fall vorsichtshalber die Antwort `N` für »Nein«.

```
PS> choice /C JN /T 10 /D N /M "Wollen Sie den Computer neu starten?"
Wollen Sie den Computer neu starten? [J,N]?N

PS> $LASTEXITCODE
2

PS> choice /C JN /T 10 /D N /M "Wollen Sie den Computer neu starten?"
Wollen Sie den Computer neu starten? [J,N]?J

PS> $LASTEXITCODE
1
```

Der Befehl `choice.exe` startet den Computer in Wirklichkeit natürlich nicht neu, denn er stellt nur (beliebige) Fragen – besorgniserregend scheint eher zu sein, dass nirgendwo angezeigt wird, welche Auswahl der Benutzer getroffen hat. Diese ist nämlich unsichtbar und wird, wie Sie sich hoffentlich gerade denken, durch den Error Level in `$LASTEXITCODE` gemeldet. Die zurückgelieferte Zahl steht für die mit `/C` angegebene Auswahlmöglichkeit: eine 1 also für die erste Auswahlmöglichkeit und eine 2 für die zweite.

Tipp

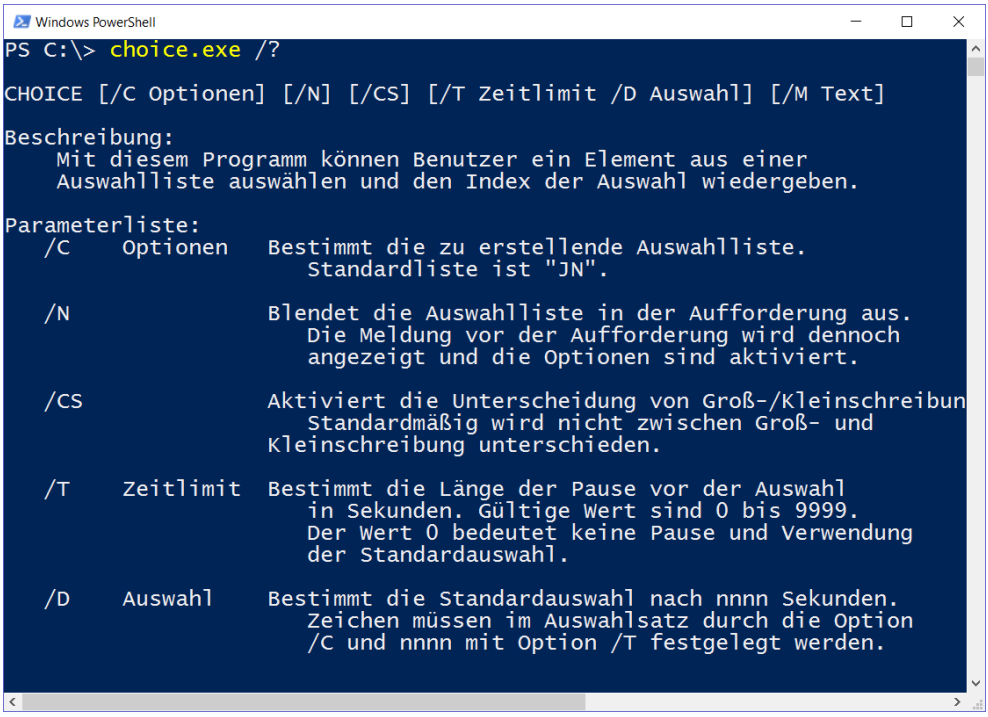

Abbildung 4.4: Interaktive Konsolenbefehle wie choice.exe funktionieren nur in echten Konsolenfenstern.

Damit PowerShell auf die Ergebnisse eines anderen Befehls reagiert, also zum Beispiel wirklich den Computer neu startet, wenn der Anwender auf J drückt, benötigen Sie sogenannte Bedingungen. Dass Bedingungen nicht wirklich kompliziert sind, zeigt ihr Einsatz in diesem kleinen Skript:

```
choice /C JN /T 10 /D N /M "Wollen Sie den Computer neu starten?"
If ($LASTEXITCODE -eq 1) { Restart-Computer -WhatIf }
```

Vorsicht: Dieses Skript würde nun *wirklich* den Computer neu starten, wenn Sie die passende Antwort geben (zumindest dann, wenn Sie im Code hinter `Restart-Computer` den Simulationsmodus `-WhatIf` entfernen). Denken Sie daran: Da `choice.exe` als interaktiver Konsolenbefehl in ISE nicht funktioniert, wäre ein Skript, das ihn einsetzt, in ISE auch nicht ausführbar. Häufig ist das nicht weiter schlimm, weil ISE in der Regel nur für die Entwicklung von PowerShell-Code eingesetzt wird, der in freier Wildbahn später in einer echten PowerShell-Konsole läuft. Trotzdem sind Inkompatibilitäten etwas, das man besser vermeiden sollte.

Rückgabetext auswerten

PowerShell kann den Ergebnistext eines Konsolenbefehls oder Skripts empfangen und auswerten – zumindest dann, wenn der Text in der PowerShell-Konsole ausgegeben wird. Erscheint der Text anderswo, zum Beispiel in einem separaten Konsolenfenster oder Dialogfeld, kommt PowerShell an solchen Text nicht heran.

```
PS> whoami.exe
w8ps\tobias

PS> $username = whoami.exe
PS> $username
w8ps\tobias

PS> "Angemeldeter User: $username"
Angemeldeter User: w8ps\tobias
```

Etwas größer ist die Herausforderung, wenn ein Befehl mehrere Zeilen Text ausgibt:

```
PS> driverquery

Modulname    Anzeigename            Treibertyp    Linkdatum
============ ====================== ============ ====================
1394ohci     OHCI-konformer 1394-Ho Kernel       26.07.2012 04:26:46
3ware        3ware                  Kernel       08.03.2012 21:33:45
ACPI         Microsoft ACPI-Treiber Kernel       26.07.2012 04:28:26
acpiex       Microsoft ACPIEx Drive Kernel       26.07.2012 04:25:57
acpipagr     ACPI-Prozessoraggregat Kernel       26.07.2012 04:27:16
AcpiPmi      ACPI-Energieanzeigetre Kernel       26.07.2012 04:27:33
(…)
```

Speichern Sie dieses Ergebnis in einer Variablen, wird daraus ein Array, und Sie greifen auf die einzelnen Zeilen über eckige Klammern zu. Das nächste Beispiel fischt sich die Zeilen 4 und 5 sowie die letzte Zeile heraus, denn die Nummerierung der Zeilen beginnt bei 0, und negative Indizes zählen von hinten:

```
PS> (driverquery) -like '*net*'

bO6bdrv      Broadcom NetXtreme II  Kernel       14.05.2012 23:42:24
ebdrv        Broadcom NetXtreme II  Kernel       13.05.2012 17:32:42
IPNAT        IP Network Address Tra Kernel       26.07.2012 04:23:01
Ndu          Windows Network Data U Kernel       26.07.2012 04:23:41
NetBIOS      NetBIOS Interface      File System  26.07.2012 04:28:19
NetBT        NetBT                  Kernel       26.07.2012 04:24:26
srvnet       srvnet                 File System  26.07.2012 04:23:17
tdx          NetIO-Legacy-TDI-Suppo Kernel       26.07.2012 04:24:58
```

Die folgende Zeile listet alle laufenden Prozesse auf, die im Namen des gerade angemeldeten Benutzers laufen:

```
PS> (qprocess) -like ">$env:USERNAME*"

>tobias                console             1   2056  taskhostex.exe
>tobias                console             1   2152  explorer.exe
>tobias                console             1   2404  livecomm.exe
(…)
```

Ob so eine Filterung überhaupt nötig ist, steht auf einem anderen Blatt. Wie sich herausstellt, kann `qprocess.exe` über seine Argumente bereits nach Benutzernamen filtern:

```
PS> qprocess /?

Zeigt Informationen über Vorgänge an.

QUERY PROCESS [* | Prozess-ID | Benutzername | Sitzungsname | /ID:nn |
              Programmname]
  [/SERVER:Servername]

  *                   Zeigt alle sichtbaren Prozesse an.
  Prozess-ID          Zeigt Prozesse anhand der Prozess-ID an.
  Benutzername        Zeigt alle Prozesse an, die zum Benutzer gehören.
  Sitzungsname        Zeigt alle Prozesse der Sitzung an.
  /ID:nn              Zeigt alle Prozesse der Sitzung "nn" an.
  Programmname        Zeigt alle dem Programm zugeordnete Prozesse an.
  /SERVER:Servername Der abzufragende Remotedesktop-Hostserver.

PS> qprocess $env:USERNAME

 BENUTZERNAME          SITZUNGSNAME        ID    PID  ABBILD
>tobias                console              1   2056  taskhostex.exe
>tobias                console              1   2152  explorer.exe
>tobias                console              1   2404  livecomm.exe
(…)
```

Hinweis

Natürlich müssen Sie Konsolenbefehle wie **`qprocess.exe`** nicht einsetzen, wenn es für den verfolgten Zweck auch komfortablere Cmdlets wie **`Get-Process`** gibt. Auf die Nuancen kommt es an: **`qprocess.exe`** liefert den Eigentümer eines Prozesses sowie die Sitzung, in der der Prozess läuft – das leistet **`Get-Process`** nicht.

Allerdings könnte alternativ auch der WMI-Dienst zurate gezogen werden. Er liefert ebenfalls laufende Prozesse, und mit ein paar Kniffen, die den bisher besprochenen Rahmen zugegebenermaßen noch sprengen, gelangen Sie an die Besitzer der Prozesse:

```
#requires -Version 2

Get-WmiObject -Class Win32_Process | Foreach-Object {
    $owner = $_.GetOwner()
    if ($owner.ReturnValue -eq 0)
    {
        $owner = '{0}\{1}' -f $owner.Domain, $owner.User
    }
    else
    {
        $owner = $null
    }
    $_ | Add-Member -MemberType NoteProperty -Name Owner -Value $owner -PassThru
} | Select-Object -Property Name, Owner, ProcessId |
Out-GridView
```

Listing 4.2: Prozesse mit Prozess-Owner ermitteln.

Rückgabetexte in Objekte verwandeln

Leider liefern Konsolenbefehle lediglich unstrukturierten Text zurück, aus dem Sie sich danach mühsam die benötigten Informationen extrahieren müssen. Cmdlets sind hier klar im Vorteil – liefern sie doch strukturierte Informationen, die klar in einzeln ansprechbare Spalten untergliedert sind.

Dabei bedarf es häufig gar nicht viel Aufwand, auch die Ergebnisse von Konsolenbefehlen in echte Objekte zu verwandeln. Viele Konsolenbefehle unterstützen nämlich die Ausgabe im (strukturierten) CSV-Format (*Comma Separated Values*). PowerShell verwandelt CSV-Daten mithilfe von `ConvertFrom-CSV` bequem in Objekte.

```
PS> driverquery /FO CSV

"Modulname","Anzeigename","Treibertyp","Linkdatum"
"1394ohci","OHCI-konformer 1394-Hostcontroller","Kernel ","26.07.2012 04:26:46"
"3ware","3ware","Kernel ","08.03.2012 21:33:45"
"ACPI","Microsoft ACPI-Treiber","Kernel ","26.07.2012 04:28:26"
"acpiex","Microsoft ACPIEx Driver","Kernel ","26.07.2012 04:25:57"
(…)

PS> driverquery /FO CSV | ConvertFrom-CSV

Modulname             Anzeigename
---------             -----------
1394ohci              OHCI-konformer 1394-Hostcontroller
3ware                 3ware
ACPI                  Microsoft ACPI-Treiber
acpiex                Microsoft ACPIEx Driver
acpipagr              ACPI-Prozessoraggregatortreiber
AcpiPmi               ACPI-Energieanzeigetreiber
acpitime              Treiber für ACPI Wake Alarm
ADP80XX               ADP80XX
AFD                   Treiber für zusätzliche WinSock-Funktionen
(…)

PS> driverquery /FO CSV | ConvertFrom-CSV | Out-GridView
```

Die Informationen sind jetzt in einzelne Spalten untergliedert, genau wie bei objektorientierten Ergebnissen, die von Cmdlets stammen. Ein Klick auf eine Spaltenüberschrift sortiert das Ergebnis nun auch.

Wenn Sie nicht gerade unter enormem Zeitdruck stehen, sollten Sie an dieser Stelle zur Kaffeemaschine spurten, sich einen ausreichenden Vorrat schwarzes Gold sichern und dann mit den gerade vorgestellten Möglichkeiten experimentieren. Es lohnt sich! Hier erhalten Sie für den Anfang eine Reihe weiterer Konsolenbefehle, die alle den Parameter `/FO CSV` unterstützen und also kommaseparierte Informationen zurückliefern:

```
PS> whoami /groups /fo CSV | ConvertFrom-CSV | Out-GridView
PS> tasklist /FO CSV | ConvertFrom-CSV | Out-GridView
PS> schtasks /FO CSV | ConvertFrom-CSV | Out-GridView
PS> systeminfo /FO CSV | ConvertFrom-CSV | Out-GridView
PS> getmac /FO CSV  | ConvertFrom-CSV | Out-GridView
PS> openfiles /Query /S [NameEinesRemotecomputers] /FO CSV /V | ConvertFrom-CSV | Out-GridView
```

Wandeln Sie auch die Rohergebnisse dieser Befehle um und lassen Sie sie im grafischen Fenster anzeigen. Zuständig sind offensichtlich immer wieder dieselben beiden Befehle: `ConvertFrom-CSV` und `Out-GridView`. Deshalb sollten Sie sich etwas näher mit den Möglichkeiten beschäftigen, die diese beiden Cmdlets bieten. Werfen Sie einen Blick in ihre Hilfe:

```
PS> Get-Help -Name ConvertFrom-CSV -ShowWindow
```

Dann nämlich werden Sie auch mit Praxisproblemen wie diesem fertig:

```
PS> driverquery /V /FO CSV | ConvertFrom-CSV
ConvertFrom-CSV : Das Element "Status" ist bereits vorhanden.
```

Diese Fehlermeldung taucht (auf deutschen Systemen) auf, sobald Sie `driverquery` mit seinem Parameter `/V` auffordern, besonders ausführliche Informationen auszuspucken. Vielleicht haben Sie schon einen Verdacht, was schiefgelaufen sein könnte, und ein Blick auf die Spaltenüberschriften bestätigt: `driverquery` hat zwei Spalten genau denselben Namen zugewiesen. Konkret kommt `Status` ungeschickterweise doppelt vor. `ConvertFrom-CSV` braucht aber eindeutige Spaltennamen:

```
PS> $ergebnis = driverquery /V /FO CSV
PS> $ergebnis[0]
"Modulname","Anzeigename","Beschreibung","Treibertyp","Startmodus","Status","Status",
"Beenden annehmen","Anhalten annehmen","Ausgelagerter Pool (Bytes)","Code(Bytes)","BSS(Bytes)",
"Linkdatum","Pfad","Init(Bytes)"
```

Gegen diese Namensgebung können Sie wenig unternehmen. Offensichtlich haben die Übersetzer die englischen Spaltennamen `State` und `Status` freizügig auf gleiche Weise übersetzt. Eine Möglichkeit der Problemlösung gibt es aber doch: Entfernen Sie die Spaltennamen, die `driverquery` liefert, nachträglich und ersetzen Sie sie kurzerhand durch Ihre eigenen.

Damit lässt sich nicht nur das Problem der doppelten Spaltennamen beheben. Gleichzeitig gewinnen Sie die Freiheit, Spalten so zu nennen, wie Sie wollen. Das ist nicht nur kosmetisch schön (störende Sonderzeichen wie Klammern lassen sich aus den Originalspaltennamen tilgen), sondern auch ein wichtiger Schritt zu kulturneutralen Daten (Daten also, die unabhängig von den Ländereinstellungen des Computers immer die gleichen Spaltennamen tragen). So könnten Sie vorgehen:

```
PS> $spalten =
'Name','DisplayName','Description','Type','Startmode','State','Status','AcceptStop','AcceptPaus
e','PagedPool','Code','BSS','LinkDate','Path','Init'
PS> driverquery /V /FO CSV | Select-Object -Skip 1 | ConvertFrom-CSV -Header $spalten |
Out-GridView
```

Mit `Select-Object -Skip 1` entfernen Sie die erste Zeile des Ergebnisses, also die Originalspaltennamen. Stattdessen definieren Sie in der Variablen `$spalten` Ihre eigenen Spaltennamen und müssen dabei nur genauso viele (eindeutige) Namen angeben, wie es Spalten gibt. Ihre neuen Spaltennamen übergeben Sie dann mit dem Parameter `-Header` an `ConvertFrom-CSV`.

Schon funktioniert der Befehl auch auf deutschen Systemen und liefert nun länderunabhängig einheitliche Spaltennamen, die noch dazu keine Sonderzeichen mehr enthalten. `ConvertFrom-CSV` ist also eine äußerst vielseitige Allzweckwaffe, um Texte mit eindeutigen Trennzeichen in echte PowerShell-Objekte zu verwandeln. Dabei muss das Trennzeichen keineswegs ein Komma sein, und wie Sie gerade gesehen haben, sind auch Spaltenüberschriften nicht unbedingt erforderlich, weil Sie sie mit `-Header` nachliefern können.

So lassen sich mit verblüffend geringem Aufwand sogar handelsübliche Textprotokolldateien parsen. Im Windows-Ordner liegt beispielsweise die Datei *windowsupdate.log*, die Buch führt über sämtliche automatischen Updates, die das Betriebssystem anfordert, empfängt und installiert (allerdings nicht mehr bei Windows 10). Sie zu lesen, ist kein Spaß, aber zumindest fällt dabei auf, dass die Einzelinformationen durch Tabulatoren voneinander getrennt werden.

Um die rohen Textinformationen dieser Datei zu parsen, teilen Sie `ConvertFrom-CSV` also nur mit, dass das Trennzeichen diesmal nicht das Komma ist, sondern der Tabulator (ASCII-Code 9), und wie die einzelnen Spalten heißen sollen:

```
PS> $spalten = 'Datum', 'Uhrzeit', 'Code1', 'Code2', 'Kategorie', 'Meldung', 'Details', 'Code3',
'Code4', 'Code5', 'Code6', 'Code7', 'Code8', 'Quelle', 'Status', 'Mode', 'Produkt'
PS> $tab = [Char]9
PS> $Path = "$env:windir\windowsupdate.log"
PS> Get-Content $Path -Encoding UTF8 | ConvertFrom-CSV -Delimiter $tab -Header $spalten |
Out-GridView
```

Nur wenige Zeilen sind dafür nötig, und diese lassen sich an sehr viele Szenarien anpassen. Ändern Sie dazu die Spaltennamen (und die Anzahl der Spalten), das verwendete Trennzeichen und den Pfadnamen, und schon lassen sich auch ganz andere textbasierte Protokolldateien nach diesem Muster parsen.

Sind die Rohdaten erst einmal ins PowerShell-Format konvertiert, können Sie die Daten nicht nur im Fenster von `Out-GridView` filtern oder per Klick auf eine Spalte sortieren. Jetzt stehen Ihnen auch sämtliche PowerShell-Cmdlets zur Verfügung, um die Daten zu filtern, zu analysieren und gezielt bestimmte Spalten auszugeben. Diese Cmdlets lernen Sie in Kapitel 5 kennen. Dass es sich lohnt, sich auf dieses Kapitel zu freuen, soll das nächste Beispiel demonstrieren.

```
PS> $spalten = 'Datum', 'Uhrzeit', 'Code1', 'Code2', 'Kategorie', 'Meldung', 'Details', 'Code3',
'Code4', 'Code5', 'Code6', 'Code7', 'Code8', 'Quelle', 'Status', 'Mode', 'Produkt'
PS> $tab = [Char]9
PS> $Path = "$env:windir\windowsupdate.log"
PS> Get-Content $Path -Encoding UTF8 | ConvertFrom-CSV -Delimiter $tab -Header $spalten |
Where-Object Quelle | Select-Object -Property Datum, Uhrzeit, Quelle, Status, Mode, Produkt |
Out-GridView
```

Es wählt mit `Select-Object` nur noch die Spalten aus, die wirklich interessant sind, und sorgt mit `Where-Object` dafür, dass lediglich die Zeilen berücksichtigt werden, in deren Spalte `Quelle` ein Wert steht. Das Ergebnis ist ein stark bereinigtes Protokoll, das jetzt nur noch die wichtigen Aktionen der Windows Update-Funktion anzeigt.

Sogar schwierige Fälle lassen sich mit der hier gezeigten Technik lösen. Der vorhin schon erwähnte Befehl `qprocess` etwa liefert alle laufenden Prozesse, aber anders als das Cmdlet `Get-Process` verrät `qprocess` auch den Benutzernamen und die Anmeldesitzung, was zum Beispiel bei der Terminalserververwaltung wichtig sein kann:

```
PS> qprocess
 BENUTZERNAME          SITZUNGSNAME        ID    PID  ABBILD
>tobias                console              1   2056  taskhostex.exe
>tobias                console              1   2152  explorer.exe
>tobias                console              1   2404  livecomm.exe
(…)
```

Das Problem bei diesem Ergebnis ist aber, dass die einzelnen Spalten nicht durch Trennzeichen abgegrenzt sind, sondern feste Spaltenbreiten verwenden. `ConvertFrom-CSV` kann solche Informationen nicht aufsplitten. Feste Spaltenbreiten bedeuten andererseits, dass ein Großteil der Spalte durch Leerzeichen aufgefüllt ist. Mit dem Operator `-replace` könnte der Text also passend gemacht werden. Dazu werden alle Textstellen, die mindestens aus zwei Leerzeichen bestehen, durch ein einzelnes Komma ersetzt:

```
PS> (qprocess) -replace '\s{2,}', ','
 BENUTZERNAME,SITZUNGSNAME,ID,PID,ABBILD
>tobias,console,1,2056,taskhostex.exe
>tobias,console,1,2152,explorer.exe
>tobias,console,1,2404,livecomm.exe
```

Der Einsatz von `-replace` entspricht also quasi dem Parameter `/FO CSV`, der von manchen Befehlen angeboten wird. Wo er fehlt, kann man sich jetzt mit `-replace` behelfen und die Ergebnisse doch noch erfolgreich an `ConvertFrom-CSV` senden:

```
PS> (qprocess) -replace '\s{2,}',','  | ConvertFrom-CSV | Out-GridView -Title "Laufende
Prozesse"
```

Auch hier steht es Ihnen natürlich frei, wie eben gezeigt zusätzlich die Spaltennamen zu verändern.

Rückgabetext analysieren

Vielleicht möchten Sie auch bloß das Ergebnis eines Konsolenbefehls analysieren, um daraus bestimmte Schlüsse zu ziehen. Wie könnte man dem Ergebnis von `ping.exe` beispielsweise entnehmen, ob ein Zielsystem geantwortet hat oder nicht?

```
PS> $ergebnis = ping www.tagesschau.de -n 1 -w 1000
PS> $ergebnis

Ping wird ausgeführt für a1838.g.akamai.net [217.89.105.154] mit 32 Bytes Daten:
Antwort von 217.89.105.154: Bytes=32 Zeit=26ms TTL=60

Ping-Statistik für 217.89.105.154:
    Pakete: Gesendet = 1, Empfangen = 1, Verloren = 0
    (0% Verlust),
Ca. Zeitangaben in Millisek.:
    Minimum = 26ms, Maximum = 26ms, Mittelwert = 26ms
```

Sie könnten beispielsweise nach dem Stichwort `Antwort` suchen, aber dann wäre Ihr Code auf deutsche Systeme beschränkt. Die Zeichenfolge `0%` würde anzeigen, dass alle abgesendeten Pakete empfangen wurden, aber dann würden auch Routerantworten als erfolgreich betrachtet.

Sie sehen also, dass die Analyse und sorgfältige Auswahl des richtigen Suchkriteriums knifflig sind. Wenn das Zielsystem erreichbar ist, gibt `ping` stets aus, *wie lange* der Ping unterwegs war. Antwortete das Zielsystem nicht oder gab es eine abschlägige Antwort vom Router, fehlt diese Angabe. Gesucht werden also Geschwindigkeitsangaben im Rückgabetext, und zwar so, dass die Ländereinstellungen keine Rolle spielen. Gefunden werden soll folglich eine Zeile, in der die im Folgenden fett hervorgehobenen Bereiche vorkommen, wobei die Zahl natürlich beliebig gehalten sein muss:

```
    Minimum = 26ms, Maximum = 26ms, Mittelwert = 26ms
```

Somit lautet die Fragestellung:

Kommt in einer Zeile des Rückgabetexts von `ping.exe` *ein Textmuster mindestens zweimal vor, das aus einem Leerzeichen, einem Gleichheitszeichen, einem weiteren Leerzeichen, einer mehrstelligen Zahl und der Zeichenfolge »ms,« besteht?*

Muster beschreibt man mit sogenannten »regulären Ausdrücken« – Steckbriefe, die beschreiben, was Sie suchen. Dazu bieten reguläre Ausdrücke drei Fahndungsmöglichkeiten: Platzhalter (die festlegen, welche Art von Information Sie suchen, also beispielsweise Leerzeichen oder Zahlen), Quantifizierer (die festlegen, wie oft ein Muster vorkommt, also beispielsweise wie viele Stellen eine Zahl haben darf) und Anker (die feste Bestandteile suchen, zum Beispiel einen Wortanfang oder einen festen Text wie `ms,`). Das Muster für die gestellte Aufgabe sieht so aus:

```
PS> $muster = '(.*?\s=\s\d{1,8}ms,\s){2}'
```

Typischerweise verursachen reguläre Ausdrücke beim Erstkontakt Panikattacken, die aber üblicherweise nach Lektüre des zehnten Kapitels wieder abflauen. Für den Moment genügt es, zu wissen, dass dieses Muster genau das gesuchte Textmuster identifizieren kann. Um zu prüfen, ob das Muster in einer Zeile des Rückgabetexts vorkommt, setzten Sie den Operator `-match` ein:

```
PS> $ergebnis -match $muster
    Minimum = 26ms, Maximum = 26ms, Mittelwert = 26ms
```

Wie Sie sehen, funktioniert die Sache erstaunlich gut. Der Operator `-match` fischt aus den Textzeilen nur diejenigen heraus, die dem Muster entsprechen. Damit ist die Prüfung jetzt leicht: Es ist nur noch festzustellen, wie viele Zeilen `-match` zurückgeliefert hat. Sind es 0 Zeilen, war der Ping nicht erfolgreich. Ist es genau eine Zeile, war er erfolgreich und hat eine Antwort vom Zielsystem empfangen. Die Anzahl der Zeilen, die `-match` zurückgibt, findet sich in der Eigenschaft `Count`, denn das Ergebnis von `-match` ist ein Array. Jedes Array teilt in dieser Eigenschaft mit, wie viele Elemente es aufweist:

```
PS> $zeilen = $ergebnis -match $muster
PS> $zeilen.Count
1
```

Damit lässt sich jetzt eine kleine Funktion namens `Test-Online` erstellen, die intern das gute alte `ping.exe` einsetzt, um zu prüfen, ob ein System antwortet. Das Ergebnis ist stets ein einfach auszuwertendes `$true` oder `$false`:

```
function Test-Online($URL=$env:COMPUTERNAME)
{
  $muster = '(.*?\s=\s\d{1,8}ms,\s){2}'
  $zeilen = (ping.exe $URL -n 1 -w 500) -match $muster
  (($zeilen.Count -gt 0) -and ($zeilen -ne $false))
}
```

Laufende Programme steuern

Zur Verwaltung von Programmen liefert PowerShell eine kleine Familie von Cmdlets mit, die alle das Substantiv `Process` tragen:

```
PS> Get-Command -Noun Process

CommandType     Name                                               Version    Source
-----------     ----                                               -------    ------
Cmdlet          Debug-Process                                      3.1.0.0    Mic...
Cmdlet          Get-Process                                        3.1.0.0    Mic...
Cmdlet          Start-Process                                      3.1.0.0    Mic...
Cmdlet          Stop-Process                                       3.1.0.0    Mic...
Cmdlet          Wait-Process                                       3.1.0.0    Mic...
```

Feststellen, ob ein Prozess läuft

Möchten Sie wissen, ob ein bestimmter Prozess läuft, greifen Sie zu `Get-Process` und suchen den Prozess. So finden Sie heraus, wie viele Instanzen des Prozesses laufen. Das folgende Beispiel prüft, ob der Notepad-Editor ausgeführt wird und, falls ja, wie viele Instanzen laufen:

```
#requires -Version 1
$name = 'notepad'
$prozesse = Get-Process -Name $name -ErrorAction SilentlyContinue
$anzahl = $prozesse.Count
$läuft = $anzahl -gt 0

if ($läuft)
{
  "Es werden $anzahl Instanzen von $name ausgeführt."
}
else
{
  "$name läuft nicht."
}
```

Listing 4.3: Herausfinden, ob ein bestimmter Prozess ausgeführt wird.

Sie können den Prozessnamen in `$name` ändern, um ein anderes Programm zu überprüfen. Ändern Sie die Variable zum Beispiel in `excel`, wenn Sie wissen möchten, ob Microsoft Excel ausgeführt wird.

Auf einen Prozess warten

Soll PowerShell warten, bis ein bestimmter Prozess beendet ist, greifen Sie zu `Wait-Process`. Die folgende Zeile wartet maximal 20 Sekunden darauf, dass sämtliche Instanzen von Microsoft Excel geschlossen werden.

```
#requires -Version 2

# auf Microsoft Excel warten:
Wait-Process -Name excel -Timeout 10 -ErrorAction SilentlyContinue -ErrorVariable err

# Fehlermeldung auswerten:
```

```
if ($err.FullyQualifiedErrorId -eq 'ProcessNotTerminated,Microsoft.Power-
Shell.Commands.WaitProcessCommand')
{
  'Excel läuft immer noch.'
}
elseif ($err.FullyQualifiedErrorId -eq 'NoProcessFoundForGivenName,Microsoft.Power-
Shell.Commands.WaitProcessCommand')
{
  'Excel lief gar nicht.'
}
else
{
  'Excel wurde beendet.'
}
```

Listing 4.4: Warten, bis Microsoft Excel beendet wurde.

Das Skript kann eine von drei Meldungen ausgeben: Lief überhaupt kein Microsoft Excel, wird `Excel lief gar nicht.` ausgegeben. Wurde Excel nicht innerhalb von 20 Sekunden beendet, meldet das Skript `Excel läuft immer noch.` Wurde Excel innerhalb des Timeouts beendet, lautet die Meldung `Excel wurde beendet.`

Welche Situation vorliegt, erkennt das Skript an der Fehlermeldung, die von `Wait-Process` ausgegeben wurde. Mit `-ErrorAction SilentlyContinue` wurde die Fehlermeldung zwar unsichtbar gemacht, aber durch `-ErrorVariable err` in der Variablen `$err` gespeichert. Dort kann sie ausgewertet werden. Die Eigenschaft `FullyQualifiedErrorId` liefert die eindeutige Fehler-ID, auf die das Skript dann reagiert.

Einstellungen laufender Prozesse ändern

PowerShell kann die Priorität eines Prozesses im laufenden Betrieb ändern und damit kontrollieren, wie viel Rechenzeit dem Prozess zur Verfügung gestellt wird. Das funktioniert prinzipiell für jeden Prozess, auf den Sie mit `Get-Process` zugreifen können, ist aber insbesondere für PowerShell selbst interessant.

Wenn PowerShell ein Skript ausführt, verwendet es dafür so viel CPU-Leistung, wie es bekommen kann. Führen Sie also ein aufwendiges Skript aus, kann dadurch ein gesamter CPU-Kern mit Volllast belegt werden. Oft ist die Bearbeitung eines Skripts aber gar nicht so wichtig. Möchten Sie also CPU-Last einsparen und ein Skript lieber etwas gemütlicher ausführen, reduzieren Sie die Priorität des PowerShell-Prozesses vorübergehend etwas.

In der Variablen `$pid` finden Sie stets die Prozess-ID des aktuellen PowerShell-Prozesses. Die folgende Zeile liefert also immer den eigenen PowerShell-Prozess.

```
PS> $prozess = Get-Process -ID $PID
```

Dessen Eigenschaften lassen sich dann verändern. Das folgende Skript ermittelt rekursiv eine Liste mit allen Logfiles aus dem Windows-Ordner und schaltet dafür die Prozesspriorität vorübergehend auf `BelowNormal`. Alle anderen normalen Prozesse erhalten Vorrang vor dem PowerShell-Prozess, sodass das Skript andere Prozesse nicht beeinflusst.

```
#requires -Version 1

# Priorität verringern
$prozess = Get-Process -id $pid
```

```
$prozess.PriorityClass = 'BelowNormal'

$liste = Get-ChildItem -Path $env:windir -Filter *.log -Recurse -ErrorAction SilentlyContinue |
Select-Object -ExpandProperty Fullname

# Priorität wiederherstellen
$prozess.PriorityClass = 'Normal'
```

Listing 4.5: Priorität der PowerShell vorübergehend verringern.

Die folgenden Einstellungen sind für `PriorityClass` erlaubt: `Normal`, `Idle`, `High`, `RealTime`, `BelowNormal`, `AboveNormal`. Sie können die Priorität der PowerShell also auch erhöhen. Das allerdings kann das Betriebssystem aus dem Takt bringen: Es reagiert danach eventuell nur noch hakelig oder zeitweise gar nicht mehr. `Idle` dagegen würde das PowerShell-Skript nur noch ausführen, wenn gerade kein anderes Programm CPU-Zeit benötigt.

Hinweis

Falls Sie verwundert feststellen, dass sich die Ausführungszeit eines Skripts bei verschiedenen Prioritäten gar nicht nennenswert ändert, arbeiten Sie vielleicht mit einem sehr gut ausgestatteten System. Selbst bei der Einstellung `Idle` läuft das Skript nicht spürbar langsamer, wenn die CPU ohnehin nur Däumchen dreht und nicht ausgelastet ist.

Die meisten modernen Computer verfügen über Multicore-Prozessoren, die also aus mehreren logischen Einzelprozessoren bestehen. Verfügt ein Computer über mehr als einen Prozessor, kann er mehrere Aufgaben gleichzeitig ausführen, indem die Aufgaben auf die unterschiedlichen Prozessoren verteilt werden. Welchem Prozessor ein Prozess zugeordnet werden kann, verrät die Eigenschaft `ProcessorAffinity`:

```
PS> $prozess.ProcessorAffinity
15
```

Das Ergebnis ist eine Bitmaske, wobei jedes Bit für einen Prozessor steht. Indirekt können Sie darüber nebenbei herausfinden, über wie viele Prozessoren ein Computer verfügt, weil Prozesse wie Notepad als Vorgabe allen Prozessoren zugewiesen werden. Lautet das Ergebnis also 1, steht nur ein Prozessor zur Verfügung. Ist das Ergebnis 15 (binär: 1111), stehen vier Prozessoren zur Verfügung. Wollen Sie einen Prozess an einen bestimmten Prozessor binden, weisen Sie diesem die passende Bitmaske zu. Der folgende Aufruf würde Notepad mit dem Wert 4 (binär: 0100) exklusiv an Prozessor 3 binden:

```
PS> $prozess.ProcessorAffinity = 4
```

Allerdings kassieren Sie erwartungsgemäß eine Fehlermeldung, wenn Sie versuchen, einen Prozess an einen nicht vorhandenen Prozessor zu binden (was besonders diejenigen betrifft, deren Computer nicht über einen Multicore-Prozessor bzw. über mehrere Prozessoren verfügt und die deshalb auch keine Auswahlmöglichkeiten haben).

Die folgende Zeile liefert die Anzahl der logischen Prozessoren Ihres Systems:

```
PS> [Environment]::ProcessorCount
4
```

Prozesse vorzeitig abbrechen

Muss ein Prozess abgebrochen werden, kann dafür `Stop-Process` eingesetzt werden, was allerdings relativ ungehobelt vonstattengeht: Der Prozess wird sofort und ohne weitere Rückfragen beendet. Daten, die ein Anwender möglicherweise noch nicht gespeichert hat, gehen dabei verloren.

Ein höflicherer Weg bei Windows-Anwendungen ist, den Prozess zunächst nur aufzufordern, sich selbst zu beenden. Dem Prozess bleibt damit die Freiheit, dem Anwender noch anzubieten, seine Daten in Sicherheit zu bringen. Zuständig für diese Aufforderung ist die Methode `CloseMainWindow()`, die jedes Prozessobjekt unterstützt, das ein eigenes Anwendungsfenster betreibt. Der folgende Code öffnet zum Beispiel einen neuen Windows-Editor und speichert das zugehörige Prozess-Objekt in einer Variablen:

```
PS> $prozess = Start-Process -FilePath notepad -PassThru
```

Geben Sie nun beliebigen Text in den Editor ein, ohne ihn zu speichern. Danach fordern Sie den Prozess auf, sich zu schließen:

```
PS> $null = $prozess.CloseMainWindow()
```

Weil `CloseMainWindow()` zurückmeldet, ob es die Aufforderung an den gewünschten Prozess senden konnte oder nicht, wird diese im Augenblick unwichtige Randnotiz noch kurz in `$null` gespeichert, also vernichtet. Die Sache funktioniert: Es erscheint tatsächlich im Editor die übliche Nachfrage, ob der Anwender seine Daten speichern will, und anschließend beendet sich der Prozess. Allerdings hat der Anwender ein Schlupfloch: Klickt er auf *Abbrechen*, wird der Prozess nicht beendet. Ein Skript würde deshalb nach einer großzügigen Karenzzeit nachprüfen, ob der Prozess auch wirklich beendet wurde, und falls nicht, mit `Stop-Process` nachhelfen:

```
PS> $prozess.CloseMainWindow()
PS> $prozess | Wait-Process -Timeout 30 -ErrorAction Ignore
PS> $prozess | Stop-Process
```

`Wait-Process` gibt dem Anwender hier also maximal 30 Sekunden Zeit, ungesicherte Arbeiten zu speichern. Wenn der Prozess danach noch vorhanden ist, beendet `Stop-Process` ihn ohne Rücksicht auf Datenverluste.

Testaufgaben

Die folgenden Aufgaben helfen Ihnen dabei, zu kontrollieren, ob Sie die Inhalte dieses Kapitels bereits gut verstanden haben oder vielleicht noch etwas vertiefen sollten. Gleichzeitig lernen Sie viele weitere und teils spannende Anwendungsbeispiele sowie die typischen Fallstricke kennen.

Aufgabe: Können Sie sich vorstellen, was die folgende Zeile bewirkt?

```
PS> $env:Path += ";."
```

Lösung: Mit dieser Anweisung wird der Umgebungsvariablen `%Path%` Text hinzugefügt. Die Zeile fügt separiert durch ein Semikolon einen weiteren Ordner der Liste der globalen Ordner hinzu. In diesem Fall allerdings handelt es sich nicht um einen bestimmten absoluten Ordnerpfad, sondern um einen relativen Pfad: Der Punkt (`.`) steht für den aktuellen Ordner. Durch

diese Änderung führt PowerShell alle Befehle, die sich im aktuellen Ordner befinden, direkt und ohne relativen oder absoluten Pfad aus.

Wechseln Sie zum Beispiel in den Ordner mit den Windows-Zubehörprogrammen, können Sie anschließend `wordpad` eingeben und damit das Textverarbeitungsprogramm WordPad starten. Ohne die Erweiterung der `%Path%`-Umgebungsvariablen hätten Sie mindestens den relativen Pfad `.\wordpad` angeben müssen:

```
PS> cd "$env:ProgramFiles\Windows NT\Accessories"
PS> wordpad
```

Aufgabe: Ändern Sie die Umgebungsvariable `%Path%` so, dass Sie künftig WordPad durch Eingabe seines Namens starten können.

Lösung: Weil sich `wordpad.exe` nicht in einem der Ordner befindet, die in der Umgebungsvariablen `%Path%` aufgelistet sind, weiß PowerShell nicht, wo das Programm zu finden ist. Deshalb muss der Pfadname seines Ordners dieser Variablen hinzugefügt werden:

```
PS> $env:Path += ";$env:ProgramFiles\Windows NT\Accessories"
```

Danach kann WordPad jederzeit durch den Befehl `wordpad` gestartet werden. Falls Sie nicht wissen, in welchem Ordner sich ein bestimmtes Programm befindet, öffnen Sie bis inklusive Windows 7 das Startmenü und suchen das Programm darin. Haben Sie es gefunden, genügt ein Rechtsklick und ein anschließender Klick auf *Eigenschaften*. Im *Eigenschaften*-Dialogfeld wird der Pfadname zum Programm genannt. In Windows 8 suchen Sie dagegen im Startbildschirm am besten nach dem Programm, indem Sie die ersten Zeichen des Namens eintippen (das funktioniert tatsächlich, obwohl zunächst kein Suchfeld sichtbar ist – dieses wird nach dem ersten Tastendruck automatisch eingeblendet). Sobald das Programm erscheint, klicken Sie mit der rechten Maustaste auf den Treffer und dann am Bildschirm unten auf *Speicherort öffnen*, woraufhin der Windows-Explorer in dem entsprechenden Verzeichnis gestartet wird. Der exakte Pfad wird erst dann sichtbar, wenn in die Adresszeile ganz rechts geklickt wird. Darüber kann der Pfad dann auch bequem per Zwischenablage kopiert werden.

Aufgabe: Starten Sie die Defragmentierung des Laufwerks *C:* mit dem Konsolenbefehl `defrag.exe`. Tipp: Hilfe zu diesem Nicht-PowerShell-Befehl erhalten Sie über `defrag /?`. Wie kann man nach Abschluss des Programms herausfinden, ob die Defragmentierung erfolgreich war?

Lösung: Der korrekte Aufruf zur Defragmentierung des Laufwerks *C:* lautet (seit Windows 7):

```
PS> defrag.exe C: /U
```

Allerdings erfordert dieser Befehl volle Administratorrechte. Verfügen Sie nur über eingeschränkte Rechte, starten Sie PowerShell als Administrator (etwa per Rechtsklick auf eine PowerShell-Verknüpfung und Klick auf *Als Administrator ausführen*). Die Analyse und Defragmentierung selbst kann sehr lange dauern. Während dieser Zeit ist die PowerShell-Konsole blockiert. Möchten Sie den Befehl vorzeitig abbrechen, drücken Sie Strg+C.

Das Ergebnis des Befehls wird über einen Zahlenwert gemeldet, den PowerShell in der Variablen `$LASTEXITCODE` zurückliefert. Brechen Sie `defrag.exe` vorzeitig ab, lautet der Rückgabewert beispielsweise `1223`. Was genau die Rückgabewerte einzelner Befehle bedeuten, hängt vom jeweiligen Befehl ab. Nur ein Rückgabewert ist weitgehend standardisiert: `0` steht für erfolgreichen Abschluss.

Aufgabe: Beenden Sie alle laufenden Instanzen von Internet Explorer. Sie kennen dazu zwei Varianten: eine zuverlässige und eine freundliche. Setzen Sie beide Varianten ein. Fallen Ihnen Unterschiede auf? Tipp: Wie verhält sich der Internet Explorer beim anschließenden Neustart?

Lösung: Der Prozessname von Internet Explorers lautet `iexplore`. Falls Sie den Prozessnamen nicht kennen, rufen Sie `Get-Process` auf, um sich alle laufenden Prozesse und ihre Prozessnamen anzeigen zu lassen. Um alle Instanzen des Internet Explorer sofort zu beenden, verwenden Sie `Stop-Process`:

```
PS> Stop-Process -Name iexplore
```

Weil das beendete Programm keine Gelegenheit hat, kontrolliert beendet zu werden, können dabei nicht nur ungesicherte Daten abhandenkommen, sondern auch andere Nebenwirkungen auftreten. Der Internet Explorer geht beim nächsten Start möglicherweise davon aus, dass er abgestürzt ist, und bietet an, die letzte Browsersitzung wiederherzustellen.

Beenden Sie den Internet Explorer dagegen auf freundliche Weise, geschieht dasselbe, als wenn der Benutzer das Fenster des Internet Explorer regulär schließen würde. In den Standardvorgaben fragt der Internet Explorer jetzt nach, ob Sie wirklich alle Registerkarten schließen wollen (sofern mehr als eine geöffnet ist). Die folgende Zeile funktioniert nur in PowerShell 3.0 (und auch nur dann fehlerfrei, wenn tatsächlich mindestens eine Instanz des Internet Explorer geöffnet ist):

```
PS> (Get-Process -Name iexplore -ErrorAction SilentlyContinue).CloseMainWindow()
```

Aufgabe: Sie möchten mithilfe von `Start-Process` den Registrierungs-Editor mit einem maximierten Fenster öffnen, aber der Befehl scheint nicht immer zu funktionieren:

```
PS> Start-Process regedit -WindowStyle Maximized
```

Die Fenstergröße ändert sich unter Umständen nicht. Warum?

Lösung: `regedit` ist eine Single Instance-Anwendung, die nicht mehrmals parallel gestartet werden kann. Läuft sie bereits, bringt `Start-Process` das Fenster der Anwendung lediglich in den Vordergrund. Die Fenstergröße wird nicht geändert, denn das geschieht nur, wenn `Start-Process` eine Anwendung auch tatsächlich startet.

Aufgabe: Sie möchten mit dem Befehl `diskpart.exe` eine neue virtuelle Festplatte erstellen. Wie das interaktiv funktioniert, wissen Sie bereits. Wie kann man eine neue virtuelle Festplatte mithilfe der PowerShell-Pipeline automatisiert und unbeaufsichtigt erstellen?

Lösung:

```
$command= @"
create vdisk file="$path"
maximum=$maximum
type=$type
select vdisk file="$path"
attach vdisk create partition primary
assign letter=$letter
format quick label="$label"
"@$command | DiskPart
```

Aufgabe: Sie haben mit `Start-Process` gespielt und wollten eigentlich den Registrierungs-Editor synchron starten, sodass PowerShell wartet, bis die Anwendung wieder geschlossen wird. Allerdings kann es sein, dass `Start-Process` den Parameter `-Wait` ignoriert und eine Fehlermeldung auswirft:

```
PS> Start-Process regedit -Wait; "Fertig!"

Start-Process : Zugriff verweigert
Fertig!
```

```
PS> Get-Process regedit

Handles  NPM(K)    PM(K)      WS(K) VM(M)   CPU(s)     Id ProcessName
-------  ------    -----      ----- -----   ------     -- -----------
     72       9     5364       8912    78            5376 regedit

PS> Stop-Process -Name regedit

Stop-Process : Der Prozess "regedit (5376)" kann aufgrund des folgenden Fehlers nicht beendet
werden: Zugriff verweigert
```

Auch gelingt es nicht, eine laufende Instanz des Registrierungs-Editors mit `Stop-Process` zu beenden. Warum?

Lösung: Dieses Verhalten ist typisch für Programme, die besondere Rechte anfordern. Verfügt Ihre PowerShell-Konsole nicht über volle Administratorrechte, fordert `regedit` diese beim Start kurzerhand selbst an und besitzt danach mehr Rechte als die PowerShell-Konsole. Weil weniger privilegierte Anwendungen nicht auf höher privilegierte Anwendungen zugreifen dürfen, bricht folglich der Kontakt zwischen PowerShell und dem gestarteten Registrierungs-Editor ab. PowerShell kann weder den Status des Programms prüfen (weswegen `-Wait` scheitert) noch das Programm mittels `Stop-Process` beenden. Möchten Sie solche Probleme vermeiden, sorgen Sie dafür, dass es zwischen PowerShell und anderen Programmen nicht zu Rechteunterschieden kommt. Starten Sie die PowerShell-Konsole beispielsweise von vornherein mit vollen Administratorrechten. In diesem Fall unterbleibt beim Start von `regedit` die Rechteerhöhung, und die Befehle führen nicht länger zu Zugriffsverletzungen.