Kapitel 2
Cmdlets – die PowerShell-Befehle

In diesem Kapitel:

Ausführlich werden in diesem Kapitel die folgenden Aspekte erläutert:

- **Cmdlets:** PowerShell-Befehle werden »Cmdlets« (sprich: »Commandlets«) genannt. Sie können nur innerhalb der PowerShell ausgeführt werden. Ihr Name besteht immer aus einem Verb, einem Bindestrich und einem Nomen. Mit `Get-Command` und `Show-Command` lassen sich Cmdlets suchen und finden. Welche Cmdlets zur Verfügung stehen, hängt nicht nur von der eingesetzten PowerShell-Version ab, sondern auch vom Betriebssystem und der installierten zusätzlichen Softwareausstattung.
- **Parameter:** Alle Informationen, die ein Cmdlet vom Anwender benötigt, erhält es über Parameter. Parameter beginnen stets mit einem Bindestrich. Gibt man im ISE-Editor hinter einem Cmdlet ein Leerzeichen und dann einen Bindestrich an, öffnet sich eine Liste der Parameter. Parameter können optional oder zwingend sein. Wird ein zwingender Parameter nicht angegeben, fragt PowerShell nach.

- **Aliase:** Cmdlets können über Aliasnamen angesprochen werden. So bildet PowerShell Befehle anderer Shells ab (beispielsweise `dir` oder `ls`). Die Liste der Aliasnamen erhält man mit `Get-Alias`.
- **Hilfe:** Klickt man in der ISE auf ein Cmdlet und drückt dann F1, öffnet sich ein Hilfefenster und erklärt die Funktionsweise des Cmdlets. Dahinter steckt das Cmdlet `Get-Help`, dessen Aufruf man nach Druck auf F1 in der Konsole der ISE sieht. Über diesen Befehl kann die Hilfe auch in der PowerShell-Konsole geöffnet werden.
- **Fehlermeldungen:** Wie sich ein Cmdlet im Fehlerfall verhalten soll, bestimmt der Parameter `-ErrorAction`. Er gehört zu den »Common Parameters«, die jedes Cmdlet unterstützt.

PowerShell ist zwar eine Automationssprache, mit der man beliebige Aufgaben beschreiben und lösen kann, damit Sie aber nicht jedes Mal von vorn beginnen müssen, enthält PowerShell bereits Tausende fix und fertiger Problemlösungen. Diese Problemlösungen werden »Cmdlets« genannt (sprich: »Commandlet«), und in diesem Kapitel geht es ausschließlich darum, wie man mit diesen Cmdlets umgeht.

Im Verlauf der nächsten Kapitel erfahren Sie dann Schritt für Schritt, wie man auch andere Probleme lösen kann, für die es (noch) keine fertigen Cmdlets gibt, und sogar, wie man völlig eigene neue PowerShell-Befehle hinzufügen kann, die dann genau das tun, was Sie gerade brauchen. Doch bereits die mitgelieferten Cmdlets sind ein spannendes Gebiet mit vielfältigen Einsatzmöglichkeiten, wie Sie gleich an vielen Praxisbeispielen sehen werden.

Alles, was Sie über Cmdlets wissen müssen

Die PowerShell-Befehle werden »Cmdlet« genannt. Es sind eigenständige, fertige Problemlösungen, die aber auf PowerShell aufbauen. Deshalb funktionieren Cmdlets nur innerhalb der PowerShell, und deshalb heißen diese Befehle eben auch Cmdlet und nicht Command: So wie es ein Treibstoff sparendes »Winglet« beim Ferienflieger nicht ohne dazugehörenden »Wing« gibt, an dem es befestigt wird, kann auch ein Cmdlet nicht außerhalb der PowerShell ausgeführt werden – daher also das »let« am Ende des Namens.

Ein Cmdlet enthält ausschließlich diejenige Fachkompetenz, die zur Lösung einer bestimmten Aufgabe nötig ist. Um all den allgemeinen übrigen Rest – das Ein- und Ausgeben von Informationen, Hilfestellung, die Übergabe von Parametern oder auch die Behandlung von Fehlern – kümmert sich ein Cmdlet nicht. Diese Dinge werden für alle Cmdlets zentral von der PowerShell bereitgestellt.

Das macht die Entwicklung von Cmdlets besonders einfach, und auch Sie als Nutzer dürfen sich freuen, denn die Bedienung der Cmdlets ist sehr konsistent. Kennen Sie sich mit einem Cmdlet aus, können Sie dieses Grundwissen auf alle anderen Cmdlets übertragen.

- **Namensgebung:** Alle Cmdlets tragen einen Doppelnamen. Der erste Namensteil ist ein Verb, also eine Tätigkeit wie zum Beispiel `Get`. Dieses Verb verrät, was das Cmdlet tun wird. Der zweite Teil ist ein Substantiv (Nomen), also ein Tätigkeitsbereich wie zum Beispiel `Service`. Er verrät, worauf sich das Cmdlet auswirkt, und ist sozusagen der Familienname. Beide Namensteile sind üblicherweise in Englisch und im Singular (Einzahl).

- **Parameter:** Hinter dem Cmdlet-Namen können zusätzliche Informationen folgen, mit denen Sie dem Cmdlet dann genauer erklären, was es genau für Sie erledigen soll. Diese Zusatzinformationen werden »Argumente« genannt und vom Cmdlet über einen passenden »Parameter« erfragt. Die meisten Parameter sind optional, also freiwillig, manche aber auch zwingend. Ohne solche Parameter kann das Cmdlet seine Arbeit nicht beginnen. Parameter sind der einzige Weg, wie Sie die Funktionsweise eines Cmdlets steuern und anpassen.
- **Hilfe:** Das PowerShell-Hilfesystem kann für alle Cmdlets ausführliche Hilfestellung liefern, die erklärt, was das Cmdlet genau für Arbeiten durchführt, welche Parameter es unterstützt und wofür die Parameter gut sind. Häufig enthält die Hilfe auch konkrete Codebeispiele, mit denen man experimentieren kann und die als Ausgangspunkt für eigene Aufrufe dienen können. Dazu müssen die ergänzenden Hilfeinformationen aber zuerst, wie im letzten Kapitel beschrieben, mit `Update-Help` aus dem Internet heruntergeladen worden sein. Steht keine Hilfe für ein Cmdlet zur Verfügung, zeigt das PowerShell-Hilfesystem zumindest die verfügbaren Parameter und die genaue »Syntax« des Cmdlet-Aufrufs an – die formale Beschreibung, wie das Cmdlet aufgerufen werden kann.
- **Autovervollständigung:** PowerShell unterstützt Sie mit den Autovervollständigungsfunktionen aus dem letzten Kapitel. Auf Wunsch vervollständigt PowerShell Cmdlet-Namen, die Parameternamen eines Cmdlets und in einigen Fällen sogar die Argumente, die Sie einem Parameter zuweisen können.

Hinweis

Ausnahmslos jedes Cmdlet stammt aus einem sogenannten »Modul«. Einige Module bringt PowerShell selbst mit, und andere Module werden von Betriebssystemen oder installierter Software hinzugefügt. Das erklärt, warum es bei Windows 8 mehr Cmdlets gibt als bei Windows 7 und bei Windows 10 wiederum mehr: Hier fügen die Betriebssysteme jeweils zusätzliche Module mit weiteren Cmdlets hinzu. Je moderner Ihr Windows ist, desto umfassender ist seine PowerShell-Unterstützung. Module sind allerdings im Augenblick nicht wichtig. Sie lesen später mehr darüber. Aktuell genügt es vollauf, sich mit den Cmdlets zu beschäftigen, die bereits vorhanden sind.

Cmdlets für eine Aufgabe finden

Cmdlets nehmen Ihnen Arbeit ab, denn der Autor des Cmdlets hat bereits für Sie eine Lösung für eine bestimmte Aufgabe programmiert. Ohne Cmdlets müssten Sie das selbst tun und zum Programmierer werden. Einen guten PowerShell-Skripter zeichnet also seine Fähigkeit aus, die notwendigen Befehle schnell zu finden und wiederzuverwenden, anstatt unnötig das Rad jeweils neu zu erfinden und selbst umfangreiche Skripte zu schreiben.

Damit die Suche nach Cmdlets kurzweilig für Sie wird, werden Sie sofort praktische Probleme lösen. Sie erhalten jeweils eine Aufgabe, die mit PowerShell erledigt werden soll. Danach suchen (und finden) Sie auf verschiedenen Wegen das passende Cmdlet und setzen es ein.

Suche nach Tätigkeit oder Tätigkeitsbereich

»Ein Computer stürzt häufiger ab. Um die Ursache zu ergründen, sollen die letzten 15 Fehlerereignisse aus dem Ereignisprotokoll des Systems ausgelesen werden.«

Um diese Aufgabe zu lösen, benötigen Sie ein Cmdlet, das Einträge aus einem Ereignisprotokoll lesen kann. Um ein solches zu finden, erinnern Sie sich an die Namensgebung aller Cmdlets: Ihr Name besteht jeweils aus einem Verb (einer Tätigkeit) und einem Substantiv (Nomen, Tätigkeitsbereich).

Überlegen Sie also zuerst, was das Cmdlet eigentlich leisten soll. Es soll Ihnen Informationen liefern. Das Verb hierfür lautet »Get«. Anfangs wird Ihnen das noch etwas willkürlich vorkommen (warum heißt es zum Beispiel nicht »List« oder »Dump«?), aber schnell werden Sie erkennen, dass die Verben der Cmdlets streng reglementiert sind. Haben Sie einmal entdeckt, dass »Get« für die Informationsbeschaffung zuständig ist, wird das auch bei allen weiteren Cmdlets so sein.

Überlegen Sie dann, was für Informationen Sie erhalten wollen. Es sollen Einträge aus dem Ereignislogbuch von Windows sein. Der zweite Namensteil eines Cmdlets ist in aller Regel ein englisches Wort, und es wird im Singular angegeben. Was also heißt »Systemlogbuch« auf Englisch?

Falls Sie nicht gleich auf »EventLog« kommen, macht das nichts, denn wenn Sie eine Cmdlet-Suche starten, dürfen Sie mit »*« Jokerzeichen benutzen, können also auch nur Wortteile zur Fahndung ausschreiben und hätten damit mit »*Log*« oder »*Event*« ebenfalls Erfolg.

Zunächst aber brauchen Sie jemanden, den Sie nach Cmdlets fragen können. Derjenige ist selbst ein Cmdlet und heißt `Get-Command`. Wenn Sie allerdings nicht mit Ergebnissen überschüttet werden wollen, rufen Sie `Get-Command` besser nicht allein auf, sondern geben Ihren Steckbrief mit. Legen Sie also mit den Parametern `-Verb` und `-Noun` fest, wonach Sie überhaupt suchen:

```
PS> Get-Command -Verb Get -Noun *Event* ↵

CommandType Name                                Version Source
----------- ----                                ------- ------
Function    Get-NetEventNetworkAdapter          1.0.0.0 NetEventPacketCapture
Function    Get-NetEventPacketCaptureProvider   1.0.0.0 NetEventPacketCapture
Function    Get-NetEventProvider                1.0.0.0 NetEventPacketCapture
Function    Get-NetEventSession                 1.0.0.0 NetEventPacketCapture
Function    Get-NetEventVmNetworkAdapter        1.0.0.0 NetEventPacketCapture
Function    Get-NetEventVmSwitch                1.0.0.0 NetEventPacketCapture
Function    Get-NetEventWFPCaptureProvider      1.0.0.0 NetEventPacketCapture
Cmdlet      Get-Event                           3.1.0.0 Microsoft.PowerShell.Utility
Cmdlet      Get-EventLog                        3.1.0.0 Microsoft.PowerShell.Management
Cmdlet      Get-EventSubscriber                 3.1.0.0 Microsoft.PowerShell.Utility
Cmdlet      Get-WinEvent                        3.0.0.0 Microsoft.PowerShell.Diagnostics
```

Welche Cmdlets auf Ihrem Computer gefunden werden, hängt davon ab, welches Betriebssystem und welche PowerShell-Version Sie verwenden, aber `Get-EventLog` sollte immer darunter sein. Die Spalte `Source` meldet, dass dieses Cmdlet aus dem Modul `Microsoft.PowerShell.Management` stammt, und alle Cmdlets, die aus Modulen stammen, die mit `Microsoft.PowerShell` beginnen, gehören zum Standardumfang von PowerShell.

Die übrigen Module, beispielsweise `NetEventPacketCapture`, sind nicht Teil von PowerShell. Sie *können* zwar, *müssen* aber nicht vorhanden sein. Solche Zusatzmodule werden von zusätzlich installierter Software bereitgestellt. Das Modul `NetEventPacketCapture` beispielsweise ist Bestandteil von Windows 10, aber nicht Teil irgendeiner PowerShell-Version.

Hinweis

Wenn Sie eben genau hingeschaut haben, werden Sie bemerken: `Get-Command` liefert nicht nur Befehle vom Typ `Cmdlet` zurück, sondern es sind unter Umständen auch solche vom Typ `Function` dabei. Im Augenblick kann Ihnen das herzlich egal sein. Beide funktionieren aus Anwendersicht genau gleich.

Der Unterschied liegt im Innenleben dieser Befehle. Während ein Cmdlet immer binär vorliegt, also zum Beispiel als Teil einer Systemdatei, sind Functions Befehle, die mit PowerShell-Bordmitteln hergestellt wurden. Der Quellcode einer Function besteht also aus reinem PowerShell-Code und kann (etwas später in diesem Buch) nicht nur ausgeforscht, sondern auch geändert werden.

Parameter erkunden

Woher weiß man eigentlich, dass `Get-Command` die Parameter `-Verb` und `-Noun` unterstützt? Wenn Sie den Befehl im ISE-Editor eingegeben haben, kennen Sie schon eine Antwort: Der ISE-Editor unterstützt Sie bei der Eingabe durch IntelliSense-Menüs (Abbildung 2.1). Da Parameternamen immer mit einem Bindestrich beginnen, müssen Sie im ISE-Editor hinter einem Cmdlet-Namen lediglich ein Leerzeichen und dann einen Bindestrich eingeben, und schon öffnet sich ein Kontextmenü mit allen infrage kommenden Parametern.

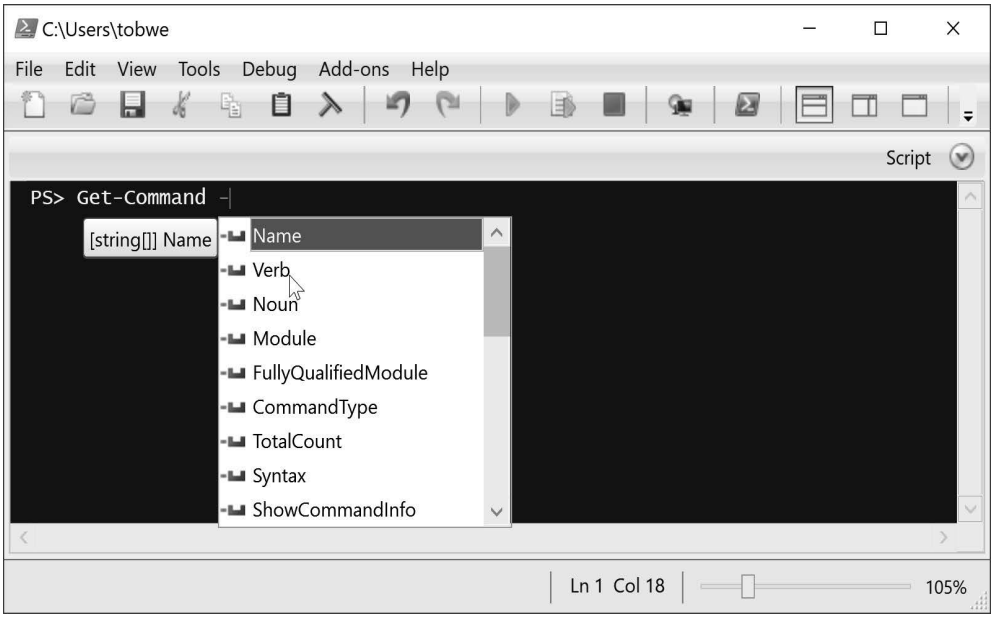

Abbildung 2.1: Parametervervollständigung im ISE-Editor.

In der PowerShell-Konsole funktioniert diese Hilfestellung auch, nur nicht automatisch. Hier müssten Sie nach Eingabe des Bindestrichs auf [⇆] drücken, und das auch noch mehrfach, bis der gewünschte Parameter erscheint.

Alternativ können Sie in der Konsole auf [Strg]+[Leertaste] drücken. Zumindest bei PowerShell 5 reagiert die Konsole darauf und versucht, das IntelliSense-Menü der ISE nachzubilden (Abbildung 2.2).

```
PS> Get-Command -Strg+Leertaste
```

Mit den Pfeiltasten kann man sich dann den gewünschten Parameter aussuchen.

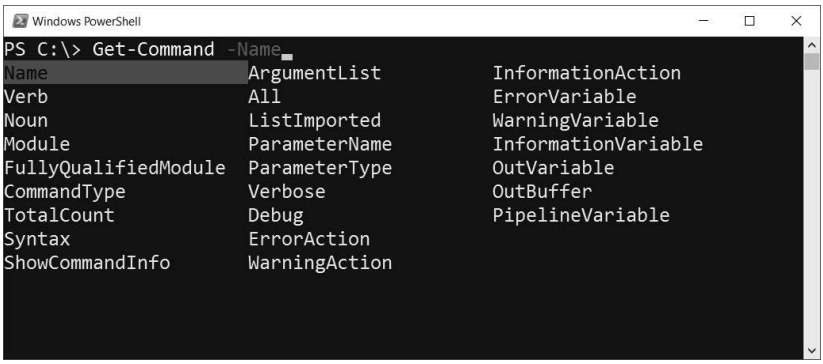

Abbildung 2.2: Parameterauflistung in der PowerShell-Konsole (PowerShell 5.0).

Hilfestellung zu Cmdlets abrufen

Sollten Sie sich nicht sicher sein, ob ein gefundenes Cmdlet auch tut, was Sie sich von ihm versprechen, werfen Sie einen Blick in die Hilfe. Falls Ihre Befehlssuche also zum Beispiel mehrere Ergebnisse geliefert hat und Sie unschlüssig sind, worin der Unterschied zwischen `Get-Event`, `Get-EventLog` und `Get-WinEvent` besteht, ziehen Sie die Hilfe zurate.

Am einfachsten gelingt dies im ISE-Editor: Hier klicken Sie einfach auf das fragliche Cmdlet und drücken F1. Sofort öffnet sich ein separates Fenster und erklärt einiges zum angeklickten Cmdlet (Abbildung 2.3). Wer genau hinschaut, entdeckt: Der entschlossene Druck auf F1 hat lediglich den Text genommen, in dem sich der Eingabecursor befand, und dann das Cmdlet `Get-Help` mit dem Parameter `-ShowHelp` aufgerufen.

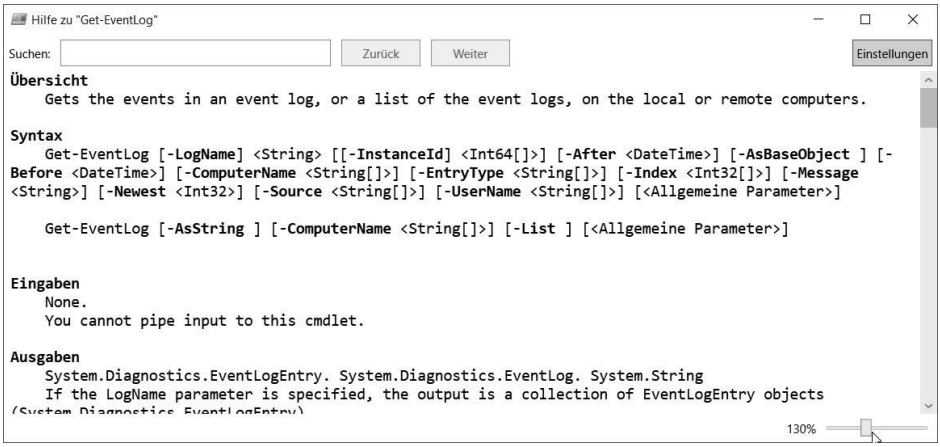

Abbildung 2.3: Hilfeinformationen zu einem Cmdlet abrufen.

Hinweis

Falls das Hilfefenster im Bereich *Übersicht* keine Beschreibung liefert und auch sonst im Wesentlichen nur die Befehlssyntax anzeigt, haben Sie die PowerShell-Hilfedateien bisher nicht aus dem Internet heruntergeladen. Blättern Sie in diesem Fall noch einmal ins letzte Kapitel zurück und holen Sie diesen Schritt nach.

Wieder etwas gelernt: `Get-Help` liefert immer die Hilfe zu dem Cmdlet (oder der Funktion), dessen Namen Sie angeben. Der Parameter `-ShowHelp` zeigt die Hilfe in einem Extrafenster an (ohne ihn erscheinen die Hilfeinformationen im Konsolenfenster).

Und das funktioniert überall: Zwar unterstützt nur der ISE-Editor den praktischen Trick mit F1, aber am Ende des Tages produziert auch dieser lediglich den notwendigen `Get-Help`-Aufruf, und der funktioniert auch in der normalen PowerShell-Konsole. Damit kennen Sie nun schon die beiden wichtigsten PowerShell-Cmdlets:

- **Bei völliger Ahnungslosigkeit:** Beauftragen Sie `Get-Command`, nach passenden Cmdlets zu suchen. Beschreiben Sie mit dem Parameter `-Verb` die gesuchte Tätigkeit und mit dem Parameter `-Noun` den gewünschten Tätigkeitsbereich (alles in Englisch und im Zweifelsfall in Einzahl). Wildcards (*) sind erlaubt.
- **Bedienungsanleitung abrufen:** Klicken Sie in der ISE auf ein Cmdlet und drücken Sie F1 oder geben Sie `Get-Help` von Hand ein und dahinter den Namen des Befehls, über den Sie etwas herausfinden wollen. Das funktioniert prima, jedenfalls dann, wenn es sich bei dem Befehl um ein PowerShell-Cmdlet oder eine Funktion handelt, denn nur diese werden vom PowerShell-Hilfesystem unterstützt – Sie sollten aber, wie im ersten Kapitel beschrieben, die PowerShell-Hilfe heruntergeladen haben. Andernfalls sind die Hilfetexte eher kurz gehalten und ohne Beispielcode.

Jedes Cmdlet liefert übrigens mithilfe des Parameters `-?` ebenfalls eine Kurzfassung der Hilfe:

```
PS> Get-EventLog -?
```

Cmdlet eingeben und Befehl ausführen

Jetzt, da Sie ein Cmdlet gefunden haben, das die Aufgabe lösen kann, soll es gleich eingesetzt werden. Das ist völlig unproblematisch, sogar auf Produktivsystemen: Da das Cmdlet das Verb `Get` trägt (und nicht etwa aggressivere Varianten wie `Stop` oder `Remove`), ist es gutartig und wird das System niemals verändern. Cmdlets mit dem Verb `Get` lesen nur.

Natürlich tippen Sie nicht den vollständigen Cmdlet-Namen ein. Als guter PowerShell-Skripter sind Sie faul beziehungsweise effizient. Es genügt, `Get-Eve` einzugeben und danach die Eingabe durch zweimaliges Drücken auf ⇆ zu vervollständigen. Das ist nicht bloß ein Tribut an die Tippfaulheit, sondern eine wichtige Sofortkontrolle: Falls die Autovervollständigung nicht funktioniert, stimmt etwas nicht mit der Eingabe.

Wenn Sie nur `Get-EventLog` eingeben und dann entschlossen auf ↵ drücken, erscheint eine Nachfrage:

```
PS> Get-EventLog
Cmdlet Get-EventLog an der Befehlspipelineposition 1
Geben Sie Werte für die folgenden Parameter an:
LogName:
```

Vor dem blinkenden Eingabecursor steht `LogName:`, und PowerShell will Ihnen damit auf eine etwas ruppige, aber nicht unfreundliche Art zu Verstehen geben, dass der Parameter `-LogName` nicht freiwillig war. Sie müssen `Get-EventLog` also schon noch verraten, in *welches* Logbuch Sie eigentlich schauen wollen. Deshalb bietet PowerShell Ihnen die Möglichkeit, die fehlende Information nachzureichen.

Können Sie zwar – clever wäre das aber nicht. Brechen Sie in solch einem Fall lieber mit Strg+C ab und drücken Sie ↑. Nun steht Ihr ursprünglicher Befehl wieder in der Eingabezeile, und Sie können den fehlenden Parameter hinzufügen.

Wenn Sie das tun, fällt Ihnen spätenstens nach Eingabe des Parameters `-LogName` eine weitere angenehme Hilfestellung auf: PowerShell autovervollständigt – zumindest im ISE-Editor – auch die Argumente für `-LogName` (Abbildung 2.4). Sie sehen im IntelliSense-Menü also alle Logbücher, die es auf Ihrem Computer gibt und die `Get-EventLog` für Sie untersuchen kann.

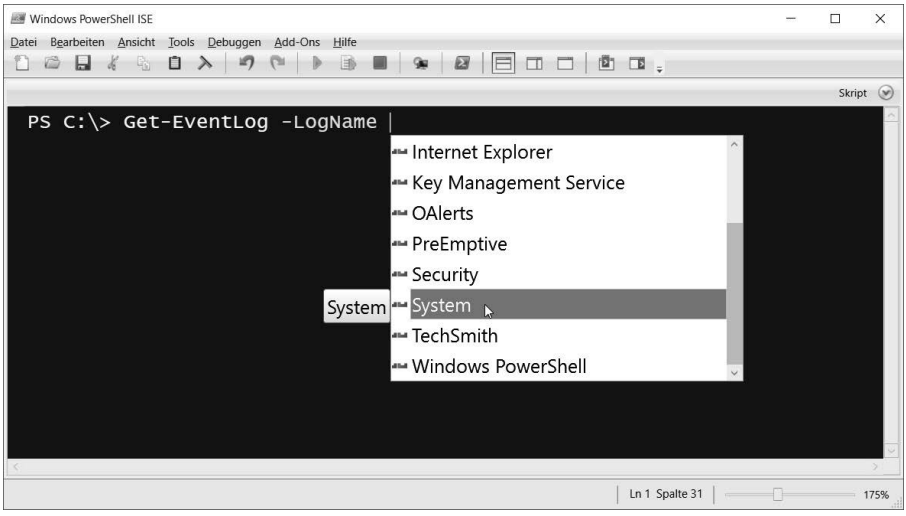

Abbildung 2.4: ISE schlägt passende Argumente automatisch vor.

Wählen Sie *System* aus und drücken Sie ↵.

```
PS> Get-EventLog -LogName System

   Index Time          EntryType   Source                 InstanceID Message
   ----- ----          ---------   ------                 ---------- -------
   30047 Sep 28 20:26  Information Microsoft-Windows...          12 Vom Pro...
   30046 Sep 28 20:26  Information Microsoft-Windows...           1 Das Sys...
   30045 Sep 28 20:26  Information Microsoft-Windows...          12 Vom Pro...
   30044 Sep 28 20:26  Information BTHUSB                 1074069522 Windows...
   30043 Sep 28 20:26  Information Microsoft-Windows...          12 Vom Pro...
   30042 Sep 28 20:26  Information Microsoft-Windows...          12 Vom Pro...
   30041 Sep 28 20:26  Information Microsoft-Windows...          12 Vom Pro...
   30040 Sep 28 20:26  Information Microsoft-Windows...         131 Die Bes...
   30028 Sep 28 20:23  Information Microsoft-Windows...          12 Vom Pro...
(...)
```

Auch hier zeigt sich die gesprächige Natur der meisten Cmdlets: Sie liefern eher zu viele als zu wenige Ergebnisse – es sei denn, Sie werden konkreter und legen über die Parameter genauer

fest, was Sie eigentlich wollen. Interessieren Sie sich beispielsweise nur für die letzten 15 Fehler, wären dies die passenden Parameter:

```
PS> Get-EventLog -LogName System -Newest 15 -EntryType Error
```

So erhalten Sie die 15 aktuellsten Einträge – und auch nur die vom Typ `Error`.

Profitipp

Ohne Parameter liefern die meisten Cmdlets erst einmal zu viele Informationen. Möchten Sie die Ergebnisse eines Cmdlets einschränken, schauen Sie sich die Ergebnisse doch mal genauer an: Sie sind fast immer in Spalten unterteilt. Um nun also nach dem Inhalt einer Spalte zu filtern, suchen Sie nach einem Parameter, der so heißt wie die Spalte. Dem übergeben Sie das Filterkriterium, also das, wonach Sie suchen.

Im Beispiel von eben verrät der Inhalt der Spalte `EntryType`, um was für ein Ereignis es sich handelt. Um die Ausgabe auf Fehler zu beschränken, ist also der Parameter `-EntryType Error` der passende. Manche Parameter unterstützen auch mehrere Argumente, die dann kommasepariert angegeben werden. Diese Zeile findet die letzten 20 Fehler und Warnungen:

```
PS> Get-EventLog -LogName System -EntryType Error, Warning -Newest 20
```

Andere Parameter unterstützen Platzhalterzeichen. Diese Zeile findet alle Fehler und Warnungen, in deren `Message`-Teil das Wort »Dienst« vorkommt:

```
PS> Get-EventLog -LogName System -EntryType Error, Warning -Message *Dienst*

   Index Time          EntryType   Source                InstanceID Message
   ----- ----          ---------   ------                ---------- -------
   29999 Sep 28 17:29  Error       bowser                3221233475 Der Hau...
   29969 Sep 26 10:21  Error       bowser                3221233475 Der Hau...
   28965 Sep 24 10:28  Error       Service Control M...  3221232483 Das Zei...
   (…)
```

Nicht für alle Spalten gibt es gleichnamige Parameter, mit denen man sie filtern könnte. Es liegt ganz im Ermessen des Entwicklers eines Cmdlets, für welche Ergebnisspalten er einen Parameter zur Filterung vorsieht. Zurzeit sind Sie dem Cmdlet-Entwickler noch ausgeliefert und können nur das durchführen, was das Cmdlet und seine Parameter anbieten. Etwas später werden Sie auch in der Lage sein, die Ergebnisse mit eigenen Mitteln nach beliebigen Kriterien zu filtern.

Herzlichen Glückwunsch, Sie haben soeben die erste Aufgabe mit PowerShell gemeistert! Die Euphorie wird indes vielleicht noch durch den miesepetrigen Kommentar Ihres Kollegen getrübt, der einwirft, dass man die Meldungen der soeben ermittelten Ereignislogbücher ja gar nicht richtig lesen könne – was bei näherer Betrachtung nicht von der Hand zu weisen ist. Die Konsolenbreite reicht nicht aus, um alle Informationen anzuzeigen. Ausgerechnet die interessante Spalte `Message` ist abgeschnitten.

Wer das nicht so gut findet, kann die Ergebnisse aber auch mehrzeilig untereinander schreiben oder in ein separates Ausgabefenster leiten. `Get-EventLog` liefert nämlich so wie alle übrigen Cmdlets mit dem Verb `Get` lediglich Informationen. Wie diese dargestellt werden, ist dem Cmdlet egal, und wenn sich niemand sonst darum kümmert, stellt eben die Konsole die Informationen dar – und schneidet überschüssiges Material ausgesprochen hemdsärmelig einfach ab.

Alternativ könnten Sie die Informationen aber auch an andere Cmdlets leiten, wie zum Beispiel `Out-Printer` (um sie zu drucken und dem vorlauten Kollegen um die Nase zu wedeln), `Out-File` (um sie in eine Datei zu schreiben) oder `Out-GridView` (um sie in einer Art Mini-Excel-Fenster anzuzeigen, wobei keine Informationen mehr abgeschnitten zu werden brauchen).

Zuständig dafür ist ein senkrechter Strich, das »Pipeline«-Symbol (|) (AltGr+< oder Strg+Alt+<). Dabei handelt es sich streng genommen um einen Operator, der den Ausgabekanal des vorangegangenen Befehls direkt mit dem Eingabekanal des nachfolgenden Befehls verbindet:

```
PS> Get-EventLog -LogName System -EntryType Error,Warning -Message *Dienst* | Format-Table -Wrap
PS> Get-EventLog -LogName System -EntryType Error,Warning -Message *Dienst* | Out-GridView
```

Befehlsergebnisse weiterverarbeiten

Wenn Sie Befehle in PowerShell ausführen – egal ob es sich um Anwendungen wie *ping.exe* handelt oder um PowerShells eigene Cmdlets –, haben Sie stets die folgenden Möglichkeiten:

- **Sie führen den Befehl einfach aus:** Der Befehl verrichtet seine Arbeit und gibt mögliche Ergebnisse direkt in der PowerShell-Konsole aus. Das gilt nicht nur für Befehlsergebnisse, sondern auch für literale Informationen wie zum Beispiel Texte in Anführungszeichen:

```
PS C:\> ping.exe 127.0.0.1 -n 1

Pinging 127.0.0.1 with 32 bytes of data:
Reply from 127.0.0.1: bytes=32 time<1ms TTL=128

Ping statistics for 127.0.0.1:
    Packets: Sent = 1, Received = 1, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 0ms, Maximum = 0ms, Average = 0ms
PS C:\> Get-EventLog -LogName System -EntryType Error -Newest 3

   Index Time          EntryType   Source                 InstanceID
   ----- ----          ---------   ------                 ----------
   15468 Nov 07 13:04  Error       DCOM                       10016
   15450 Nov 07 09:37  Error       NetBT                  3221229793
   15448 Nov 07 09:37  Error       DCOM                       10016


PS C:\> "Hallo Welt"
Hallo Welt
```

- **Sie leiten die Befehlsergebnisse mit der Pipeline (|) an einen anderen Befehl weiter:** Alles, was in die Konsole ausgegeben wird, kann auch an andere Befehle weitergereicht werden. `Out-GridView` ist beispielsweise ein Universal-Cmdlet, das die Ergebnisse in einem Extrafenster anzeigt (grafische Elemente einschließlich dieses Fensters werden nur auf dem Windows-Betriebssystem unterstützt). Alternativ könnten Sie aber auch `Out-Printer` (zum Ausdrucken) oder `Out-File` (zum Speichern in einer Textdatei) wählen:

```
PS C:\> ping.exe 127.0.0.1 -n 1 | Out-GridView
PS C:\> Get-EventLog -LogName System -EntryType Error -Newest 3 | Out-GridView
PS C:\> "Hallo Welt" | Out-GridView
```

- **Sie speichern das Ergebnis in einer Variablen:** In der Variablen befindet sich anschließend das, was ansonsten in die Konsole ausgegeben worden wäre. Sie könnten den Inhalt der Variablen anschließend beliebig oft ausgeben oder an andere Befehle weiterreichen:

```
PS C:\> $ergebnis = ipconfig
PS C:\> $ergebnis

Windows IP Configuration

Wireless LAN adapter WiFi:

   Connection-specific DNS Suffix  . :
   Link-local IPv6 Address . . . . . : fe80::ca6:c542:daf1:5cc%14
   Autoconfiguration IPv4 Address. . : 169.254.5.204
   Subnet Mask . . . . . . . . . . . : 255.255.0.0
   Default Gateway . . . . . . . . . :


PS C:\> $ergebnis -like '*IPv4*'
   Autoconfiguration IPv4 Address. . : 169.254.5.204

PS C:\> $ergebnis = Get-EventLog -LogName System -EntryType Error -Newest 3
PS C:\> $ergebnis

   Index Time          EntryType   Source                 InstanceID
   ----- ----          ---------   ------                 ----------
   15468 Nov 07 13:04  Error       DCOM                       10016
   15450 Nov 07 09:37  Error       NetBT                  3221229793
   15448 Nov 07 09:37  Error       DCOM                       10016


PS C:\> $ergebnis | Out-GridView

PS C:\> $ergebnis = "Hallo Welt"
PS C:\> $ergebnis
Hallo Welt
PS C:\> $ergebnis * 5
Hallo WeltHallo WeltHallo WeltHallo WeltHallo Welt
```

Mit ISE nach Cmdlets suchen

»Benötigt werden die Lottozahlen der nächsten Woche, also sechs Zahlen zwischen 1 und 49, bei denen keine Zahl doppelt vorkommen darf.«

Das gesuchte Cmdlet für diese Aufgabe soll Zufallszahlen generieren. Das englische Wort für *Zufall* lautet *Random*. Diesmal soll das Cmdlet mit einem besonderen Assistenten gesucht werden. Dazu öffnen Sie ISE, falls Sie nicht schon damit arbeiten. Dann drücken Sie Strg+F1.

Ein Fenster öffnet sich (und wer genau hinsieht, erkennt, dass die Tastenkombination selbst gar kein Fenster öffnet, sondern lediglich in der ISE-Konsole den Befehl `Show-Command` abgesetzt hat. Es ist also eigentlich dieser Befehl, der das Fenster öffnet, und der kann auch direkt aufgerufen werden, also auch in der klassischen PowerShell-Konsole).

Geben Sie das Suchwort `Random` ins Feld *Name* ein. Noch während Sie das Suchwort eintippen, wird die Liste der verfügbaren Cmdlets ausgedünnt, und schnell kristallisiert sich heraus, dass `Get-Random` das gesuchte Cmdlet sein muss.

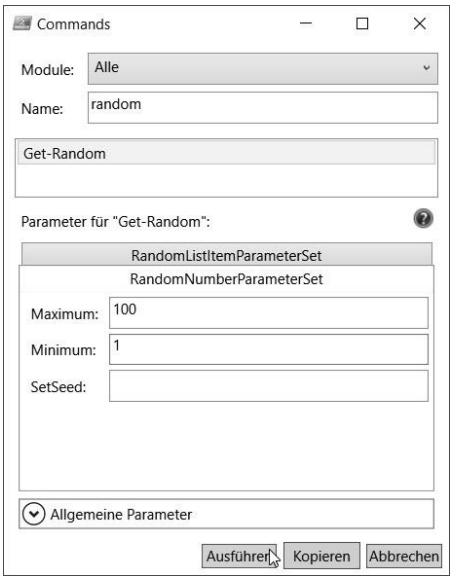

Abbildung 2.5: Cmdlets finden und mit Parametern ausstatten.

Das Fenster kann aber noch mehr und ist Ihnen dabei behilflich, das Cmdlet mit Parametern zu füttern. Dazu klicken Sie auf *Get-Random*. Im unteren Teil des Fensters sehen Sie nun alle Parameter, die das Cmdlet unterstützt, und können den Parametern Werte zuweisen (Abbildung 2.5). Klicken Sie auf *Ausführen*, wird der Befehl einschließlich der festgelegten Parameter direkt in die Konsole eingetragen und ausgeführt:

```
PS C:\> Get-Random -Maximum 100 -Minimum 1
59
```

Eine Zufallszahl haben Sie nun, und sie liegt auch im angegebenen Wertebereich. Für Lottozahlen müsste dieser nun noch etwas eingeschränkt werden. Allerdings fragt sich, wie man sechs Zufallszahlen bekommt, die noch dazu nicht doppelt vorkommen dürfen.

Rufen Sie deshalb die Eingabehilfe mit `Show-Command` noch einmal auf, aber diesmal gezielt für den Befehl `Get-Random`:

```
PS> Show-Command Get-Random
```

Die Suchelemente im Fenster fallen jetzt weg, und am oberen Rand treten die Registerkarten deutlicher zutage, die Sie beim ersten Aufruf vielleicht ganz übersehen haben (Abbildung 2.6).

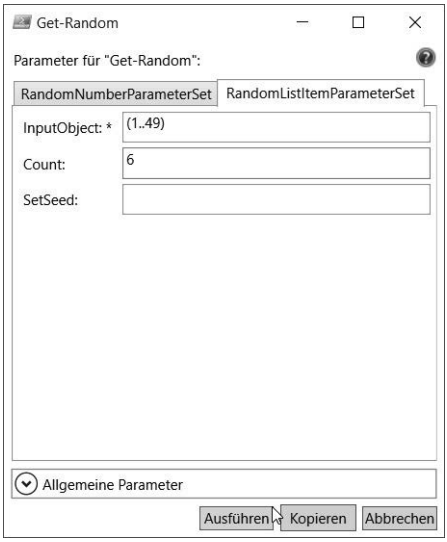

Abbildung 2.6: Ein anderes »Parameterset« eines Cmdlets verwenden.

Die Namen auf den Registerkarten sind nicht besonders hilfreich, aber sobald Sie die zweite Registerkarte namens *RandomListItemParameterSet* anklicken, zeigt `Get-Random` ganz andere Parameter. Darunter ist auch einer, der `Count` heißt. Vielversprechend!

Maximal- und Minimalwerte kann man hier allerdings nicht festlegen. Stattdessen fällt der Parameter `InputObject` auf, der zudem mit einem Sternchen als zwingend erforderlich gekennzeichnet ist. Ihm weist man die Zahlen zu, die in das digitale »Lottoziehgerät« gelegt werden sollen.

Entweder kochen Sie sich einen Kaffee und geben die möglichen Lottozahlen dann in aller Seelenruhe als (relativ lange) kommaseparierte Liste ein, oder Sie erinnern sich an das vorangegangene Kapitel. Mit `..` liefert PowerShell Zahlenreihen. `1..49` erzeugt also die Zahlen 1 bis 49. Damit auch wirklich diese Zahlenreihe (und nicht etwa der Ausdruck `1..49` selbst) in das Ziehgerät gelangt, setzen Sie den Ausdruck in runde Klammern. Vielleicht erinnern Sie sich noch an die entsprechende Passage aus dem letzten Kapitel: Runde Klammern funktionieren bei PowerShell genauso wie in der Mathematik: PowerShell wertet zuerst das aus, was in den runden Klammern steht, und fährt dann mit dem Ergebnis des Ausdrucks fort.

Ein Klick auf *Ausführen* generiert den kompletten Befehlsaufruf, der danach in der Konsole erscheint und die Lottozahlen generiert. Falls es wirklich die der nächsten Ziehung sind (und Sie daran teilnehmen), schauen Sie sich bei Gelegenheit das Cmdlet `Send-MailMessage` an. Jedenfalls sind es aber wie gefordert sechs, und keine kommt doppelt vor. Mission erfüllt.

```
PS> Get-Random -InputObject (1..49) -Count 6
32
17
33
14
30
41
```

Hinweis

Nun gut, vollkommen intuitiv war der Name des Parameters `-InputObject` nicht, und dass er das digitale Lottoziehgerät füllt, war nirgends beschrieben. Auch die runden Klammern um `(1..49)` waren keine Selbstverständlichkeit. Deshalb bietet das Fenster eine kleine unscheinbare Schaltfläche mit einem Fragezeichen darauf. Klickt man diese an, öffnet sich ein Extrafenster mit allen Detailinformationen zum Cmdlet und seinen Parametern. Sogar Beispielcode liefert es, der spätestens jetzt klarstellt, wie die Parameter eingesetzt werden.

Dialoggestützte Parametereingabe

In der vorherigen Aufgabe hatten Sie mit `Get-EventLog` Fehlereinträge im Systemereignisprotokoll gefunden. Das könnten Sie auch dialoggestützt tun:

```
PS> Show-Command Get-EventLog
```

Das Dialogfeld zeigt nun auch an, dass es für `Get-EventLog` ganz ähnlich wie eben bei `Get-Random` ebenfalls einen zweiten Parametersatz namens `List` gibt. Wechseln Sie zu diesem Parametersatz und aktivieren hier beispielsweise das Kontrollkästchen *List*, sieht der generierte Befehl so aus:

```
PS> Get-EventLog -List

  Max(K) Retain OverflowAction       Entries Log
  ------ ------ --------------       ------- ---
  20.480      0 OverwriteAsNeeded      1.077 Application
  20.480      0 OverwriteAsNeeded          0 HardwareEvents
     512      7 OverwriteOlder             0 Internet Explorer
  20.480      0 OverwriteAsNeeded          0 Key Management Service
                                           Security
  20.480      0 OverwriteAsNeeded      1.099 System
  15.360      0 OverwriteAsNeeded      1.071 Windows PowerShell
```

`Get-EventLog` kann also zweierlei durchführen: entweder die Einträge eines bestimmten Protokolls auflisten oder die Namen aller vorhandenen Ereignisprotokolle nennen. Jede Funktion wird über einen eigenen Parametersatz abgebildet, und insgesamt verhält sich `Get-EventLog` so wie die meisten Cmdlets: Es ist »schmal, aber tief«, kann also genau einen sehr speziellen Themenbereich abdecken, diesen dafür aber gründlich.

Genau diese Erkenntnisse hätten Sie sogar bereits der Hilfe entnehmen können, denn die Syntax darin beschreibt alle diese Dinge auf eine recht knappe, aber sehr eindeutige Weise. Die Syntax für `Get-Random` sieht zum Beispiel so aus:

```
PS> Get-Random -?

NAME
    Get-Random

ÜBERSICHT
    Gets a random number, or selects objects randomly from a collection.


SYNTAX
    Get-Random [[-Maximum] [<Object>]] [-InformationAction {SilentlyContinue |
    Stop | Continue | Inquire | Ignore | Suspend}] [-InformationVariable
    [<System.String>]] [-Minimum [<Object>]] [-SetSeed [<Int32>]]
```

```
    [<CommonParameters>]

    Get-Random [-InputObject] <Object[]> [-Count [<Int32>]]
    [-InformationAction {SilentlyContinue | Stop | Continue | Inquire | Ignore
    | Suspend}] [-InformationVariable [<System.String>]] [-SetSeed [<Int32>]]
    [<CommonParameters>]
```

Tatsächlich listet die Syntax den Befehl **Get-Random** zweimal auf, jeweils mit unterschiedlichen Parametern. Das sind die sogenannten *Parametersätze*. Es gibt den Befehl in Wirklichkeit also zweimal, und je nachdem, welche Parameter Sie verwenden, verhält er sich anders. Der erste Parameter bildet das Ziehen eines zufälligen Werts ab. Der zweite ist das Ziehen ohne Zurücklegen.

Alles, was in der Syntax nicht in eckigen Klammern steht, ist Pflicht. Im zweiten Parametersatz muss also mindestens der Wert für den Parameter **-InputObject** angegeben werden. Alles andere ist freiwillig.

Und auch bei **Get-EventLog** beschreibt die Syntax den Befehl auffallend vollständig:

```
PS> Get-EventLog -?

NAME
    Get-EventLog

ÜBERSICHT
    Gets the events in an event log, or a list of the event logs, on the local
    or remote computers.


SYNTAX
    Get-EventLog [-LogName] <String> [[-InstanceId] <Int64[]>] [-After
    <DateTime>] [-AsBaseObject] [-Before <DateTime>] [-ComputerName
    <String[]>] [-EntryType <String[]>] [-Index <Int32[]>] [-Message <String>]
    [-Newest <Int32>] [-Source <String[]>] [-UserName <String[]>]
    [<CommonParameters>]

    Get-EventLog [-AsString] [-ComputerName <String[]>] [-List]
    [<CommonParameters>]
```

Hier ist im ersten Parametersatz mindestens der Wert für den Parameter **-LogName** zwingend erforderlich. Den Parameternamen selbst braucht man nicht unbedingt anzugeben. Auch der Parameter **-InstanceID** lässt sich so entschlüsseln: Weder der Parameter noch sein Wert ist zwingend. Gibt man den Wert an, muss es sich um Daten vom Typ »Int64« handeln (ganze Zahlen). Und weil hinter dem Datentyp **[]** steht, kann es auch ein Array sein, also zum Beispiel viele kommaseparierte Werte.

Der folgende Aufruf ist also vollkommen legal (und würde aus dem Systemlogbuch alle Events mit den IDs 10 bis 200 auflisten:

```
PS> Get-EventLog System (10..200)
```

Die Syntax verrät, dass der Parameter **-EntryType** ebenfalls mehrere Werte haben darf. Der Datentyp »String« wird mit **[]** abgeschlossen. Wenn Sie also nicht nur Fehler, sondern vielleicht auch Warnungen auslesen möchten, wäre dieser Aufruf erlaubt und würde die neuesten zehn Fehler oder Warnungen ausgeben:

```
PS> Get-EventLog -LogName System -EntryType Error, Warning -Newest 10
```

Mit der Hilfe nach Cmdlets suchen

Dass beinahe alle Cmdlets über eigene Hilfedateien verfügen, haben Sie bereits erlebt. Jedes Cmdlet unterstützt den Parameter `-?`, mit dem man eine Kurzhilfe einschließlich der Befehlssyntax abrufen kann. Voraussetzung dafür ist also, dass man den Namen des gesuchten Cmdlets bereits kennt (und dass Sie, wie im letzten Kapitel gezeigt, die Hilfeinhalte mit `Update-Help` aus dem Internet heruntergeladen haben).

Die Hilfe kann aber auch Befehle für Sie finden, die Sie noch nicht kennen. Bevor Sie erfahren, wie das funktioniert, schauen Sie sich zunächst an, wie die Hilfe bei Cmdlets funktioniert, die Sie schon kennen. Hinter der Hilfe steckt das Cmdlet `Get-Help`, und sobald Sie mit dem Parameter `-?` die Kurzhilfe eines Cmdlets abrufen, verrät diese am Ende, mit welchen weiteren Befehlen Sie noch mehr Informationen erhalten können:

```
PS> Get-Process -?

NAME
    Get-Process
    (...)

HINWEISE
    Zum Aufrufen der Beispiele geben Sie Folgendes ein: "get-help Get-Process
    -examples".
    Weitere Informationen erhalten Sie mit folgendem Befehl: "get-help
    Get-Process -detailed".
    Technische Informationen erhalten Sie mit folgendem Befehl: "get-help
    Get-Process -full".
    Geben Sie zum Abrufen der Onlinehilfe Folgendes ein: "get-help Get-Process
-online"
```

Wer sich also für die Praxisbeispiele zu einem Cmdlet interessiert, verwendet `Get-Help` mit dem Parameter `-Examples`:

```
PS> Get-Help -Name Get-Process -Examples
```

Damit die vielen Informationen nicht an Ihnen vorbeisausen, sondern seitenweise angezeigt werden, ersetzen Sie `Get-Help` durch `help`. Jetzt wird stets nur eine Bildschirmseite gefüllt, und erst wenn Sie mit Lesen fertig sind, blättert ein Druck auf Leertaste zur nächsten Seite um. Mit Strg+C kann man die Ausgabe vorzeitig abbrechen.

Tipp

Wer die Hilfe zu einem Cmdlet lieber parallel in einem separaten Fenster anzeigen möchte, setzt `-ShowWindow` ein.

Durch die Volltextsuche finden Sie Informationen schnell: Geben Sie ein Stichwort ins *Suchen*-Feld am oberen Fensterrand ein, werden alle Vorkommnisse gelb markiert. Mit dem Schieberegler am rechten unteren Fensterrand lässt sich die Schriftgröße genau wie in ISE stufenlos anpassen. Außerdem kann über die Schaltfläche *Einstellungen* in der rechten oberen Ecke die Anzeige auf bestimmte Inhalte begrenzt werden. Aktivieren Sie darin beispielsweise nur die Kontrollkästchen *Syntax* und *Beispiele*, erhalten Sie eine Kurzübersicht über die Parameter, die ein Cmdlet unterstützt, sowie die Praxisbeispiele, die das Cmdlet im Einsatz demonstrieren.

Leider enthält Ihnen das Hilfefenster einige Hilfeinformationen vor (jedenfalls dann, wenn Sie die detaillierte Hilfe vorher mit `Update-Help` heruntergeladen haben). Die vollständige Hilfe mit allen Details lässt sich nur ohne Extrafenster mit dem Parameter `-Full` abrufen:

```
PS> Get-Help -Name Get-Process -Full
```

Immerhin könnten Sie die Hilfetexte beinahe genauso einfach in die Zwischenablage kopieren und von dort direkt in Ihre Lieblingstextverarbeitung einfügen, sie ausdrucken und als Bettlektüre verwenden:

```
PS> Get-Help -Name Get-Process –Examples | clip.exe
```

Unbekannte Befehle suchen

Um gänzlich unbekannte Cmdlets aufzuspüren, übergeben Sie `Get-Help` anstelle eines bestimmten Cmdlet-Namens einfach ein Suchwort. Möchten Sie zum Beispiel wissen, welche Cmdlets Windows-Dienste steuern, verwenden Sie als Suchwort `service`:

```
PS> Get-Help -Name service

Name                              Category  Module                    Synopsis
----                              --------  ------                    --------
Get-Service                       Cmdlet    Microsoft.PowerShell.M... Gets the servic...
New-Service                       Cmdlet    Microsoft.PowerShell.M... Creates a new W...
New-WebServiceProxy               Cmdlet    Microsoft.PowerShell.M... Creates a Web s...
Restart-Service                   Cmdlet    Microsoft.PowerShell.M... Stops and then ...
Resume-Service                    Cmdlet    Microsoft.PowerShell.M... Resumes one or ...
Set-Service                       Cmdlet    Microsoft.PowerShell.M... Starts, stops, ...
Start-Service                     Cmdlet    Microsoft.PowerShell.M... Starts one or m...
Stop-Service                      Cmdlet    Microsoft.PowerShell.M... Stops one or mo...
Suspend-Service                   Cmdlet    Microsoft.PowerShell.M... Suspends (pause...
```

Prompt listet `Get-Help` alle Cmdlets auf, in deren Hilfethema das Suchwort gefunden wurde. Weil `Get-Help` im Gegensatz zu `Get-Command` Zugriff auf die detaillierten Hilfetexte zu den einzelnen Cmdlets hat, erscheint in der Spalte `Synopsis` auch gleich zuvorkommenderweise die Kurzbeschreibung zu den einzelnen Cmdlets. Leider ist ausgerechnet diese Spalte wegen Platzmangels nicht vollständig lesbar. Sie haben schon einige Möglichkeiten kennengelernt, das Problem abgeschnittener Spalten zu beheben. Leiten Sie das Ergebnis zum Beispiel an `Format-Table` oder `Out-GridView` weiter.

Kann `Get-Help` nur ein einziges infrage kommendes Cmdlet finden, zeigt es sofort dessen Hilfe an.

`Get-Help` kann Cmdlets auch auf andere Weise suchen. Interessieren Sie sich zum Beispiel für alle Cmdlets, die einen bestimmten Parameter wie `-ComputerName` unterstützen (und also höchstwahrscheinlich remotefähig sind), setzen Sie `Get-Help` mit dem Parameter `-Parameter` ein:

```
PS> Get-Help -Name * -Parameter ComputerName

Name                              Category  Module                    Synopsis
----                              --------  ------                    --------
Invoke-Command                    Cmdlet    Microsoft.PowerShell.Core Runs c...
New-PSSession                     Cmdlet    Microsoft.PowerShell.Core Create...
Connect-PSSession                 Cmdlet    Microsoft.PowerShell.Core Reconn...
Receive-PSSession                 Cmdlet    Microsoft.PowerShell.Core Gets r...
(…)
```

Get-Help kann zudem allgemeine Hilfethemen durchsuchen, die nicht für ein bestimmtes Cmdlet gelten, sondern Informationen zu allgemeinen PowerShell-Themen anbieten. Möchten Sie beispielsweise mehr zu Operatoren erfahren, suchen Sie nach dem Stichwort operator:

```
PS> Get-Help -Name operator

Name                              Category  Module                    Synopsis
----                              --------  ------                    --------
about_Arithmetic_Operators        HelpFile                            Descri...
about_Assignment_Operators        HelpFile                            Descri...
about_Comparison_Operators        HelpFile                            Descri...
about_Logical_Operators           HelpFile                            Descri...
about_Operators                   HelpFile                            Descri...
about_Operator_Precedence         HelpFile                            Lists ...
about_Type_Operators              HelpFile                            Descri...
```

Diesmal erhalten Sie die Namen sämtlicher allgemeiner Hilfethemen (die Spalte Category meldet hierfür diesmal HelpFile und nicht Cmdlet), die Operatoren beschreiben.

Um Hilfestellung zu einem der speziellen Themen zu bekommen, geben Sie den Namen der Hilfe (oder einen eindeutigen Teil davon) an und verwenden am besten help anstelle von Get-Help, um mit [Leertaste] bequem seitenweise durchblättern zu können:

```
PS> help -Name about_Comparison_Operators
PS> help -Name Comparison
```

Da die allgemeinen Hilfethemen mit about_ beginnen und der Category HelpFile entsprechen, könnten Sie alle diese Themen auch auf einem der folgenden beiden Wege auflisten:

```
PS> Get-Help -Name about_*
PS> Get-Help -Category HelpFile
```

Befehl	Hilfedatei	Beschreibung
help compari [↵]	*about_Comparison_Operators*	Vergleichsoperatoren
help wildcard [↵]	*about_Wildcards*	Platzhalterzeichen
help special [↵]	*about_Special_Characters*	Sonderzeichen
help regular [↵]	*about_Regular_Expressions*	reguläre Ausdrücke
help redir [↵]	*about_Redirection*	Umleitung
help quot [↵]	*about_Quoting_Rules*	Anführungszeichen
help parsing [↵]	*about_Parsing*	Befehlsparsing
help escape [↵]	*about_Escape_Characters*	Textsonderzeichen
help _common [↵]	*about_CommonParameters*	allgemeine Parameter

Tabelle 2.1: Schnellabruf ausgewählter PowerShell-Themenkomplexe.

Mit Parametern Wünsche formulieren

Erst wenn Sie dem Cmdlet mit Parametern genauer mitteilen, was Sie eigentlich wollen, schöpfen Cmdlets ihr wahres Potenzial aus. Parameter sind übrigens der *einzige* Weg, das Verhalten eines Cmdlets zu beeinflussen. Während also ein Mensch üblicherweise sieben Sinne hat, um seine Umwelt wahrzunehmen, haben Cmdlets nur einen: ihre Parameter.

Falls ein Cmdlet ohne Angabe von Parametern bereits ungefähr das vollbringt, was Sie wollen, aber eben noch nicht ganz genau, schauen Sie sich seine Parameter genauer an, zum Beispiel mithilfe der Autovervollständigung oder der Hilfe zum Cmdlet. Sehr häufig findet sich der passende Parameter, damit das Cmdlet die gestellte Aufgabe noch besser lösen kann.

Parameter wecken das volle Potenzial der Cmdlets

Was Parameter aus einem unscheinbaren Cmdlet herausholen können, zeigt das Beispiel von `Get-Date`. Ohne Parameter liefert es das aktuelle Datum und die Uhrzeit und wirkt nicht unbedingt spektakulär:

```
PS> Get-Date

Montag, 10. September 2012 11:03:40
```

Cmdlets sind »schmal, aber tief«, also Spezialisten für ein bestimmtes Fachgebiet, das sie dann bis in alle Ecken und Winkel beleuchten. Das gilt auch für `Get-Date`. Es ist Ihr Universalbefehl für alle Aufgaben rund um Datum und Zeit. Welche Lösungen `Get-Date` anbietet, wird einzig durch seine Parameter gesteuert. Einen anderen Weg gibt es nicht. Die Autovervollständigung, die Hilfe oder IntelliSense in ISE zeigen diese Parameter an. Falls `Get-Date` also Ihre Aufgabe meistern kann, muss es dafür einen oder mehrere passende Parameter geben.

Alle folgenden Aufgaben können mit `Get-Date` und seinen Parametern gelöst werden. Widerstehen Sie möglichst dem Drang, nach der jeweiligen Aufgabe sofort die Lösung zu lesen. Ich kann Sie daran zwar augenscheinlich nicht hindern, aber nachhaltiger ist, die Lösung zuerst selbst zu suchen. Als Hilfsmittel haben Sie bereits die Cmdlet-Hilfe (`Get-Help`) kennengelernt, die Ihnen zahlreiche Beispiele und alle Beschreibungen zu unbekannten Parametern liefert:

```
PS> help -Name Get-Date -Examples              # zeigt die Codebeispiele für Get-Date an
PS> help -Name Get-Date -Parameter DisplayHint # ruft die Hilfe für den Parameter
                                               # "DisplayHint" ab
PS> help -Name Get-Date -ShowWindow            # zeigt die gesamte Hilfe zu Get-Date in
                                               # einem durchsuchbaren Extrafenster an
```

Nur das Datum oder nur die Zeit ausgeben

»Get-Date soll nur das aktuelle Datum ausgeben, aber nicht die Zeit (oder umgekehrt).«

Der Parameter `-DisplayHint` sorgt dafür, dass `Get-Date` nur das Datum, nur die Uhrzeit oder beides zur Ausgabe bringt:

```
PS> Get-Date -DisplayHint Date
Montag, 10. September 2012

PS> Get-Date -DisplayHint Time
11:29:37
```

Welche Werte der Parameter `-DisplayHint` akzeptiert, zeigt die Hilfe an:

```
PS> Get-Help -Name Get-Date -Parameter DisplayHint

-DisplayHint [<DisplayHintType>]
    Determines which elements of the date and time are displayed.
```

```
    Valid values are:

    -- Date: displays only the date

    -- Time: displays only the time

    -- DateTime: displays the date and time

    DateTime is the default. This parameter does not affect the DateTime
    object that Get-Date gets.
```

Sie können die erlaubten Werte auch der Fehlermeldung entnehmen, die Sie erhalten, wenn Sie einen unsinnigen Wert angeben:

```
PS> Get-Date -DisplayHint Unsinn
Get-Date : Der Parameter "DisplayHint" kann nicht gebunden werden. Der Wert
"Unsinn" kann nicht in den Typ "Microsoft.PowerShell.Commands.DisplayHintType"
konvertiert werden. Fehler: "Der Bezeichner "Unsinn" kann keinem gültigen
Enumeratornamen zugeordnet werden. Geben Sie einen der folgenden
Enumeratornamen an, und wiederholen Sie den Vorgang:
Date, Time, DateTime"
```

In ISE werden die erlaubten Werte sogar als IntelliSense-Menü angezeigt.

Den Wochentag eines bestimmten Datums errechnen

»Geben Sie den heutigen Wochentag (oder den Wochentag eines beliebigen anderen Datums) aus. Stellen Sie zum Beispiel fest, an welchem Tag Sie geboren wurden und ob Sie möglicherweise ein Sonntagskind sind.«

Der Parameter `-Format` legt fest, in welchem Format `Get-Date` das Datum und die Uhrzeit ausgibt. Die Hilfe zum Parameter `-Format` sagt dazu:

```
PS> Get-Help -Name Get-Date -Parameter Format

-Format [<String>]
    Displays the date and time in the Microsoft .NET Framework format
    indicated by the format specifier. Enter a format specifier. For a list of
    available format specifiers, see "DateTimeFormatInfo Class" in the MSDN
    (Microsoft Developer Network) library at
    http://go.microsoft.com/fwlink/?LinkId=143638.

    When you use the Format parameter, Windows PowerShell gets only the
    properties of the DateTime object that it needs to display the date in the
    format that you specify. As a result, some of the properties and methods
    of DateTime objects might not be available.
```

Die gültigen Formatbezeichner werden zwar nicht aufgeführt, dafür aber ein Link zu einer Webseite: *http://go.microsoft.com/fwlink/?LinkId=143638*. Sie listet im unteren Teil eine umfangreiche Tabelle mit den Platzhalterzeichen der einzelnen Datums- und Zeitinformationen auf. Alternativ blättern Sie vor zu Kapitel 8 und schauen sich den Abschnitt über den Formatierungsoperator »`-f`« an. Dort finden Sie die Tabellen mit allen erlaubten Platzhalterzeichen.

Hinweis

Die Groß- und Kleinschreibung dieser Platzhalterzeichen ist wichtig. Das Platzhalterzeichen `m` steht zum Beispiel für Minuten, während `M` den Monat repräsentiert – ein nicht ganz unerheblicher Unterschied.

Der Wochentag wird vom Platzhalterzeichen `d` (für day) repräsentiert:

```
PS> Get-Date -Format d
10.09.2012
PS> Get-Date -Format dd
10
PS> Get-Date -Format ddd
Mo
PS> Get-Date -Format dddd
Montag
```

Je mehr Platzhalterzeichen Sie also angeben, desto ausführlicher wird die Ausgabe, und `dddd` meldet schließlich den vollständigen Wochentag. Das funktioniert genau so mit den übrigen Platzhalterzeichen, die die Webseite auflistet. Sie können auch kombiniert werden, um beispielsweise einen Zeitstempel für Dateinamen zu generieren:

```
PS> Get-Date -Format 'yyyy-MM-dd HH-mm-ss-fff'
2012-09-10 11-47-20-375
```

Als Vorgabe verwendet `Get-Date` das aktuelle Datum und die aktuelle Uhrzeit. Möchten Sie ein anderes Datum (wie zum Beispiel Ihren Geburtstag) verwenden, geben Sie dieses Datum mit dem Parameter `-Date` an, und zwar am besten im kulturneutralen Format, das unabhängig von bestimmten regionalen Ländereinstellungen immer richtig interpretiert wird. Der Wochentag des 5. September 1968 berechnet sich also so:

```
PS> Get-Date -Format dddd -Date 1968-09-05
Donnerstag
```

Die aktuelle Kalenderwoche anzeigen

»Ermitteln Sie die aktuelle Kalenderwoche für das heutige Datum oder ein beliebiges anderes Datum.«

Mit dem Parameter `-UFormat` lässt sich die Ausgabe ganz ähnlich wie mit `-Format` speziell formatieren. Allerdings unterstützt `-UFormat` eine andere Liste von Platzhaltern, zu denen auch die Kalenderwoche gehört. Die Hilfe schreibt dazu:

```
PS> Get-Help -Name Get-Date -Parameter UFormat

-UFormat [<String>]
    Displays the date and time in UNIX format. For a list of the format
    specifiers, see the Notes section.

    When you use the UFormat parameter, Windows PowerShell gets only the
    properties of the DateTime object that it needs to display the date in the
    format that you specify. As a result, some of the properties and methods
    of DateTime objects might not be available.
```

Die erlaubten Platzhalterzeichen werden im Abschnitt `HINWEISE` der Hilfe angezeigt. Um diesen Abschnitt zu sehen, muss die Hilfe komplett angezeigt werden:

```
PS> Get-Help -Name Get-Date –Full

(…)
HINWEISE
        (…)
        Uformat Values:
```

```
The following are the values of the UFormat parameter. The format for
the command is:
Get-Date -UFormat %<value>
For example, Get-Date -UFormat %d

Date-Time:
Date and time - full
(default) (Friday, June 16, 2006 10:31:27 AM)
c     Date and time - abbreviated (Fri Jun 16 10:31:27 2006)
Date:
D    Date in mm/dd/yy format (06/14/06)
x    Date in standard format for locale (09/12/07 for English-US)
Year:
C   Century (20 for 2006)
Y   Year in 4-digit format (2006)
y   Year in 2-digit format (06)
G   Same as 'Y'
g   Same as 'y'
Month:
b   Month name - abbreviated (Jan)
B   Month name - full (January)
h   Same as 'b'
m   Month number (06)
Week:
W  Week of the year (00-52)
V   Week of the year (01-53)
U   Same as 'W'
Day:
a   Day of the week - abbreviated name (Mon)
A   Day of the week - full name (Monday)
u   Day of the week - number (Monday = 1)
d   Day of the month - 2 digits (05)
e   Day of the month - digit preceded by a space ( 5)
j    Day of the year - (1-366)
w   Same as 'u'
Time:
p   AM or PM
r   Time in 12-hour format (09:15:36 AM)
R   Time in 24-hour format - no seconds (17:45)
T   Time in 24 hour format (17:45:52)
X   Same as 'T'
Z   Time zone offset from Universal Time Coordinate (UTC) (-07)
Hour:
H   Hour in 24-hour format (17)
I    Hour in 12 hour format (05)
k   Same as 'H'
l    Same as 'I' (Upper-case I = Lower-case L)
Minutes & Seconds:
M   Minutes (35)
S   Seconds (05)
s   Seconds elapsed since January 1, 1970 00:00:00 (1150451174.95705)
Special Characters:
n   newline character (\n)
t   Tab character (\t)
```

Die aktuelle Kalenderwoche liefert der Platzhalter `%V`:

```
PS> Get-Date -UFormat %V
```

Und falls Sie die Kalenderwoche eines anderen Datums benötigen, geben Sie das Datum wieder mit dem Parameter `-Date` an. Die Kalenderwoche des 8. Juli 1967 bestimmen Sie so:

```
PS> Get-Date -UFormat %V -Date '2015-11-24'
47
```

Hinweis

Etwas gelogen ist das schon. Es gibt mehrere Definitionen dazu, was eine »Kalenderwoche« eigentlich ist und wann sie beginnt. Die deutsche Definition der Kalenderwoche stimmt nicht mit der Definition des Unix-Formats überein. Der 24. November 2015 lag in Deutschland in Kalenderwoche 48 und nicht 47, wie es `Get-Date` im letzten Beispiel berechnete.

Die Kalenderwoche in Deutschland ist nach ISO 8601 so definiert:

- Kalenderwochen haben sieben Tage, beginnen an einem Montag und werden über das Jahr fortlaufend nummeriert.
- Die Kalenderwoche 1 eines Jahres ist diejenige, die den ersten Donnerstag enthält.

Der Grund für die Diskrepanz bei der berechneten Kalenderwoche: In den USA beginnt die Woche mit dem Sonntag, in Europa dagegen mit dem Montag. Um die Kalenderwoche nach europäischen Maßstäben korrekt zu berechnen, bleibt nur der Rückgriff auf die Low-Level-Systemfunktionen, die normalerweise vor Ihnen abgeschirmt im Inneren der Cmdlets ablaufen und die erst sehr viel später in diesem Buch genauer besprochen werden:

```
# Kalenderwoche dieses Datums berechnen:
PS> $datum = Get-Date -Date '2015-11-24'

# Kalenderwoche (USA)
PS> (Get-Culture).Calendar.GetWeekOfYear($datum,'FirstFourDayWeek','Sunday')
47

# Kalenderwoche (Europa)
PS> (Get-Culture).Calendar.GetWeekOfYear($datum,'FirstFourDayWeek','Monday')
48
```

Das Datum vor 48 Stunden berechnen

»Berechnen Sie das Datum und die Uhrzeit von vor genau 48 Stunden und rufen Sie damit die Fehlerereignisse und Warnungen der letzten 48 Stunden aus dem Systemereignisprotokoll ab.«

Es findet sich kein Parameter, mit dem eine bestimmte Zeitmenge vom aktuellen Datum abgezogen werden kann. Ein Blick in die Beispiele des Cmdlets zeigt aber ein interessantes Codebruchstück:

```
PS> Get-Help -Name Get-Date –Examples

(...)
-------------------------- EXAMPLE 5 -------------------------

    PS C:\>$a = Get-Date
    PS C:\>$a.IsDaylightSavingTime()
    True
    (…)
```

Offensichtlich ist es möglich, das Ergebnis von `Get-Date` in einer Variablen zu speichern und danach mit einem Punkt (.) auf weitere Befehle zuzugreifen. Jedenfalls spricht das Beispiel auf diese Weise den Befehl `IsDaylightSavingTime()` an, der offenbar feststellt, ob das Datum in die Sommerzeit fällt.

Sie sehen daran einerseits, dass die Codebeispiele der Hilfe keinerlei Rücksicht nehmen auf unsere Videospiellevels in diesem Buch und unter Umständen auch Techniken zeigen, die Sie noch gar nicht kennengelernt haben. Andererseits können Sie solche Beispiele durchaus aufgreifen und mit dem bereits gesammelten Wissen kombinieren.

Sie wissen schon, dass PowerShell eine Autovervollständigung besitzt, die in ISE sogar häufig automatisch als IntelliSense-Menü angezeigt wird. Wenn Sie das Codebeispiel in ISE eingeben, werden Sie schnell feststellen, wie man herausfindet, dass das von `Get-Date` gelieferte Ergebnis einen Befehl namens `IsDaylightSavingTime()` enthält (und welche sonstigen Befehle es noch gibt).

Das IntelliSense-Menü zeigt nicht nur den Befehl `IsDaylightSavingTime` an, sondern auch andere Befehle, die die Zeichen `is` enthalten, die Sie eingegeben haben. Verdächtig interessant ist zum Beispiel `AddMilliseconds` (der ebenfalls `is` enthält, nur eben nicht am Anfang).

Löschen Sie die Zeichen `is` wieder, sodass hinter der Variablen `$datum` nur noch ein Punkt steht, zeigt die Liste sämtliche Befehle an. Sollte sich das IntelliSense-Menü schon wieder geschlossen haben, drücken Sie Strg+Leertaste, um es erneut zu öffnen.

Geben Sie nun `Add` ein, zeigt das IntelliSense-Menü alle Befehle mit diesem Schlüsselbegriff, und `AddDays` ist genau das, wonach Sie suchen. In der QuickInfo dahinter steht, wie der Befehl eingesetzt wird: Er erwartet eine Zahl mit Nachkommastellen (Datentyp `double`) und liefert ein neues Datum zurück (Datentyp `datetime`).

Zwar befinden wir uns hier definitiv nicht mehr im PowerShell-Videospiellevel 1, aber wirklich schwierig ist der neue Befehl trotzdem nicht; kennt man ihn erst, kann man damit künftig jederzeit schnell und einfach relative Datumsangaben berechnen lassen:

```
PS> $datum = Get-Date
PS> $datum.AddHours(-48)

Samstag, 8. September 2012 12:24:29
```

Und wer sich aus dem ersten Kapitel noch an die Bedeutung der runden Klammern erinnert, kann das auch in einer einzelnen Zeile ganz ohne Variablen formulieren:

```
PS> (Get-Date).AddHours(-48)

Samstag, 8. September 2012 12:24:29
```

Damit lassen sich jetzt auch die Fehler und Warnungen der letzten 48 Stunden auslesen, unabhängig davon – und ohne hardcodiertes Datum –, wann dieser Code ausgeführt wird:

```
PS> Get-EventLog -LogName System -EntryType Error,Warning -After (Get-Date).AddHours(-48)
```

Hinweis

Wenn die Zeile keine Ergebnisse liefert, gab es vielleicht gar keine Fehler und Warnungen in den letzten 48 Stunden. Um das zu überprüfen, wiederholen Sie den Aufruf einfach ohne den Parameter `-After`.

Drei universelle Parametertypen

Alle Parameter eines Cmdlets lassen sich auf drei grundlegende Parametertypen zurückführen. Ganz gleich also, welche Parameter ein Cmdlet unterstützt: Jeder dieser Parameter lässt sich eindeutig einem der drei Parametertypen in Tabelle 2.2 zuordnen.

Parametername	Argument	Typ
`-ParameterName`	*Wert*	benannter Parameter
`-Parameter`		Switch-Parameter
	Wert	positionaler Parameter

Tabelle 2.2: Die drei grundsätzlichen PowerShell-Parametertypen.

Im Grunde handelt es sich bei den drei Parametertypen und die drei denkbaren Kombinationsmöglichkeiten aus Parametername und Argument.

Benannte Parameter

Am häufigsten begegnet Ihnen der »benannte Parameter«: Er ist immer ein Schlüssel/Wort-Paar. Dieses Paar besteht aus dem Parameternamen, der stets am Bindestrich vor seinem Namen erkannt werden kann, und dem ihm zugewiesenen Wert, also seinem »Argument«.

Benannte Parameter bieten ein »explizites Binding«: Weil Sie den Parameternamen selbst vor Ihr Argument schreiben, legen Sie unmissverständlich fest, an welchen Parameter Ihr Argument gehen soll. Sie überlassen diese wichtige Entscheidung also nicht irgendwelchen Automatismen, und das führt zu robusterem, schnellerem und zudem noch besser lesbarem Code.

Im folgenden Befehlsaufruf finden sich zwei benannte Parameter. Dem Parameter `-LogName` wird das Argument `System` zugewiesen und dem Parameter `-EntryType` das Argument `Error`:

```
PS> Get-EventLog -LogName System -EntryType Error
```

Dieser Code ist selbstbeschreibend, also gut lesbar, weil durch den vorangestellten Parameternamen klar ist, was die Argumente bedeuten. Die Reihenfolge der Parameter spielt bei benannten Parametern ebenfalls keine Rolle. Entsprechend leistet dieser Befehlsaufruf mit anderer Parameterreihenfolge genau dasselbe wie das zurückliegende Beispiel:

```
PS> Get-EventLog -EntryType Error -LogName System
```

Dieses Grundprinzip findet sich bei allen Cmdlet-Aufrufen immer wieder. Der folgende Befehl listet alle Protokolldateien aus dem Windows-Ordner auf und setzt dazu erneut zwei benannte Parameter ein:

```
PS> Get-ChildItem -Path c:\windows -Filter *.log
```

Switch-Parameter

Manche Parameter sollen nur bestimmte Funktionalitäten ein- oder ausschalten. Anstatt einem Parameter dabei die Werte `$true` oder `$false` zuzuweisen, verwendet man stattdessen einen Switch-Parameter. Er funktioniert ähnlich wie ein Lichtschalter: Gibt man ihn an, gilt die Funktion als eingeschaltet, andernfalls ausgeschaltet. Switch-Parameter besitzen also kein folgendes Argument, sondern werden nur angegeben oder weggelassen.

Wenn Sie also die Protokolldateien nicht nur im Windows-Ordner suchen möchten, sondern auch in allen seinen Unterordnern, schalten Sie die Rekursion mit dem Switch-Parameter `-Recurse` ein:

```
PS> Get-ChildItem -Path c:\windows -Filter *.log -Recurse
```

Verzichten Sie auf die Angabe des Parameters, wird entsprechend ohne Rekursion nur im angegebenen Ordner gesucht.

Hinweis

Es kann durchaus sein, dass dieses Beispiel (genau wie einige der folgenden) neben den gewünschten Resultaten auch Fehlermeldungen ausgeben. Ein Grund dafür könnte sein, dass die Rekursion auch Unterordner untersuchen will, für die Sie gar keine Zugriffsberechtigungen besitzen. Sie erfahren gleich, wie störende Fehlermeldungen unterdrückt werden können. Ignorieren Sie die roten Fehlermeldungen einstweilen einfach freundlich.

Damit auch versteckte Dateien gefunden werden, fügen Sie den Switch-Parameter `-Force` hinzu:

```
PS> Get-ChildItem -Path c:\windows -Filter *.log -Recurse -Force
```

Weil Switch-Parameter genau wie benannte Parameter eindeutig benannt sind, spielt bei ihnen die Reihenfolge ebenfalls keine Rolle.

Switch-Parameter gibt es bei vielen Cmdlets. `Get-Process` listet normalerweise alle laufenden Prozesse auf:

```
PS> Get-Process
```

Geben Sie den Switch-Parameter `-FileVersionInfo` an, schaltet das Cmdlet in einen anderen Modus und zeigt jetzt die den Prozessen zugrunde liegenden Dateien (samt ihren Versionen) an.

```
PS> Get-Process -FileVersionInfo
```

Auch bei `Get-EventLog` ist Ihnen bereits ein Switch-Parameter begegnet: Der Parameter `-List` schaltet das Cmdlet in den Listenmodus, bei dem nicht mehr die Einträge eines bestimmten Ereignisprotokolls ausgegeben werden, sondern die Namen der vorhandenen Ereignisprotokolle:

```
PS> Get-EventLog -List
```

Positionale Parameter

Neben dem »expliziten Binding« gibt es auch das »implizite Binding«, das immer dann stattfindet, wenn Sie einem Cmdlet nur Ihre Argumente übergeben, ohne durch Parameterangabe selbst festzulegen, an welche Parameter Ihre Argumente gebunden werden sollen. Möglich ist das nur bei manchen Parametern. Die Parameter müssen also positionale Argumente akzeptieren und eine definierte Position besitzen.

Beim »impliziten Binding« übernimmt PowerShell dann die Aufgabe, das Argument an einen Parameter zu übergeben (oder zu »binden«). Wie bei allen Aufgaben, die Sie aus der Hand geben und an andere delegieren, verlieren Sie dabei die Kontrolle und müssen sich darauf verlassen, dass PowerShell Ihr Argument an den richtigen (nämlich den von Ihnen gewünschten) Parameter übergibt. Deshalb können positionale Parameter leicht zu Fehlern führen, denn manchmal ist PowerShell anderer Meinung als Sie. Außerdem sind solche Befehle schlechter

lesbar, weil ohne Angabe des Parameternamens nicht mehr klar ersichtlich ist, welche Bedeutung ein bestimmtes Argument überhaupt hat.

Gedacht sind positionale Parameter für erfahrene PowerShell-Anwender und auch nur für Code, der keine lange Lebensspanne hat – also für Direkteingaben beispielsweise, wenn es schnell gehen muss. Positionale Parameter haben jedenfalls in Skripten absolut nichts zu suchen (wenngleich sie dort nicht ausdrücklich verboten sind): Skripte müssen auch nach Wochen und Monaten noch lesbar und nachvollziehbar sein.

Hier sind Beispiele für Befehlsaufrufe, die positionale Parameter einsetzen:

```
PS> Get-Service spooler
PS> Get-ChildItem c:\windows *.log
PS> Get-EventLog System
```

Positionale Parameter sind grundsätzlich immer nur eine besondere Form eines benannten Parameters. Entsprechend können positionale Parameter *immer* in benannte Parameter umgewandelt werden. Die drei Aufrufe hätten also auch mit benannten Parametern formuliert werden können:

```
PS> Get-Service -Name Spooler
PS> Get-ChildItem -Path c:\windows -Filter *.log
PS> Get-EventLog -LogName System
```

Was die Frage aufwirft, woher PowerShell überhaupt weiß, welchen Parametern ein positionales Argument zugeordnet werden soll. Wie leitet PowerShell also ab, dass das unbenannte Argument `Spooler` ausgerechnet an den Parameter `Name` gebunden werden soll?

Dazu kann jedes Cmdlet seinen Parametern Positionsnummern zuteilen. Bei `Get-Service` trägt der Parameter `Name` die Position 1. Ihm wird also das erste unbenannte Argument zugewiesen:

```
PS> Get-Help -Name Get-Service -Parameter Name

-Name <String[]>
(…)
    Erforderlich?                   false
    Position?                       1
    Standardwert                    All services
    Pipelineeingaben akzeptieren?true (ByPropertyName, ByValue)
    Platzhalterzeichen akzeptieren?true
```

Tipp

Die Hilfe zum Parameter verrät Ihnen nebenbei auch, ob ein Parameter zwingend nötig ist. Muss ein Parameter eingegeben werden, wie der Parameter **`-LogName`** bei **`Get-EventLog`**, steht hinter **`Erforderlich?`** der Wert **`true`**:

```
PS> Get-Help -Name Get-EventLog -Parameter LogName

-LogName <String>
    (…)
    Erforderlich?                   true
    Position?                       1
    (…)
```

Sie haben auch schon gesehen, was geschieht, wenn Sie einen erforderlichen Parameter nicht angeben: Das Cmdlet fragt dann kurzerhand nach.

Bei `Get-ChildItem` trägt der Parameter `Path` die Position 1 und der Parameter `Filter` die Position 2. Gibt man zwei unbenannte Argumente also genau in dieser Reihenfolge an, werden sie an die richtigen Parameter gebunden.

```
PS> Get-Help -Name Get-ChildItem -Parameter Path

-Path <String[]>
    (…)
    Position?                   1
    (…)

PS> Get-Help -Name Get-ChildItem -Parameter Filter

-Filter <String>
    (…)
    Position?                   2
    (…)
```

Welche Parameter überhaupt eine Position tragen (und welche Position das ist), verrät auch die Syntax eines Cmdlets, und zwar für alle Parameter auf einen Blick:

```
PS> Get-ChildItem -?

NAME
    Get-ChildItem

ÜBERSICHT
    Gets the items and child items in one or more specified locations.


SYNTAX
    Get-ChildItem [[-Path] <String[]>] [[-Filter] <String>] [-Exclude
    <String[]>] [-Force] [-Include <String[]>] [-Name] [-Recurse]
    [-UseTransaction [<SwitchParameter>]] [<CommonParameters>]

    Get-ChildItem [[-Filter] <String>] [-Exclude <String[]>] [-Force]
    [-Include <String[]>] [-Name] [-Recurse] -LiteralPath <String[]>
    [-UseTransaction [<SwitchParameter>]] [<CommonParameters>]
```

Positionale Parameter erkennt man daran, dass sie mit eckigen Klammern als »optional« gekennzeichnet sind, also allein in eckigen Klammern stehen, ohne dass sich ihr Argument mit in dieser (innersten) eckigen Klammerebene befindet.

Parameter ohne Argument, also Switch-Parameter, sind nie positional.

Wie sich herausstellt, sind längst nicht alle Parameter mit einer Position versehen. Bei `Get-ChildItem` sind nur die Parameter `-Path` und `-Filter` positional verwendbar. Alle übrigen Parameter müssen benannt werden, wenn man sie verwenden will.

Bei `[-Exclude <String[]>]` etwa ist `-Exclude` nicht allein eingeklammert, auch das Argument `<String[]>` ist noch in der gleichen eckigen Klammerebene enthalten. Also erfüllt der Parameter nicht die Voraussetzungen eines positionalen Parameters.

Alle Parameter eines Cmdlets listet die Hilfe übrigens auf, indem für den Parameternamen das Platzhalterzeichen `*` angegeben wird:

```
PS> Get-Help Get-Process -Parameter *
```

Spätestens jetzt leuchtet ein, dass Parameter, die keine Position zugewiesen bekommen haben, in der Hilfe als `named` (also *benannt*) bezeichnet werden.

Tipp

Weil positionale Parameter nur eine Abkürzung für schnelle Eingaben im hektischen Alltag sind, werden nur lediglich am häufigsten benötigten Parameter üblicherweise mit einer Position versehen. Selbst wenn Sie sich also aus Stilgründen gegen die Verwendung positionaler Parameter entscheiden, identifizieren Sie bei einem unbekannten Cmdlet auf diese Weise schnell seine wichtigsten Parameter. Die allerwichtigsten Parameter eines Cmdlets sind stets mit einer Position versehen und/oder zwingend erforderlich.

Zwingend erforderliche Parameter lassen sich in der Syntax mit etwas Übung ebenfalls identifizieren:

```
PS> Get-EventLog -?

NAME
    Get-EventLog

ÜBERSICHT
    Gets the events in an event log, or a list of the event logs, on the local
    or remote computers.


SYNTAX
    Get-EventLog [-LogName] <String> [[-InstanceId] <Int64[]>] [-After
    <DateTime>] [-AsBaseObject] [-Before <DateTime>] [-ComputerName
    <String[]>] [-EntryType <String[]>] [-Index <Int32[]>] [-Message <String>]
    [-Newest <Int32>] [-Source <String[]>] [-UserName <String[]>]
    [<CommonParameters>]

    Get-EventLog [-AsString] [-ComputerName <String[]>] [-List]
    [<CommonParameters>]
```

Tatsächlich gibt es im Beispiel nur einen Pflichtparameter, nämlich `-Logname`. Es darf Sie an dieser Stelle nicht irritieren, dass der Parameter `-Logname` selbst in eckigen Klammern steht. Sein Argument ist entscheidend, und dieses Argument steht nicht in eckigen Klammern. Das Argument muss also angegeben werden.

Common Parameter – allgemeine Parameter für alle Cmdlets

Cmdlets regeln nur die speziellen Dinge, für die sie erfunden wurden. Alle allgemeinen Aufgaben, die sämtliche Cmdlets gleichermaßen betreffen, werden von PowerShell erledigt. Deshalb unterstützen Cmdlets neben ihren eigenen individuellen Parametern zusätzlich eine Reihe sogenannter »Common Parameter«.

Die Common Parameter werden im Hilfethema `about_commonParameters` beschrieben, das man sich am besten in einem separaten Fenster anzeigen lässt:

```
PS> help about_CommonParameters -ShowWindow
```

Allgemeiner Parameter	Typ	Beschreibung
-Verbose	Switch	Zeigt Informationen an, die eine Funktion oder ein Cmdlet über Write-Verbose ausgibt. Ohne diesen Parameter werden diese Ausgaben normalerweise unterdrückt.
-Debug	Switch	Zeigt Debug-Meldungen an, die eine Funktion oder ein Cmdlet mit Write-Debug ausgibt, und fragt dann nach, ob die Ausführung unterbrochen werden soll.
-ErrorAction	Wert	Legt fest, wie sich das Cmdlet bei einem Fehler verhalten soll. Erlaubte Werte: Continue: Fehler melden und fortsetzen (Vorgabe) SilentlyContinue: keinen Fehler melden, weitermachen Suspend (nur in Workflows erlaubt): hält den Workflow im Fehlerfall an, sodass der Workflow nach Analyse der Fehlerursache fortgesetzt werden kann Stop: Fehler melden und abbrechen Ignore: keinen Fehler melden, aber Fehler in $error protokollieren und weitermachen Inquire: nachfragen Die Vorgabe wird mit der Variablen $ErrorActionPreference festgelegt und ist normalerweise auf Continue eingestellt.
-ErrorVariable	Wert	Fehlermeldungen in der angegebenen Variablen protokollieren. Dieser Parameter verlangt den *Namen* einer Variablen (ohne vorangestelltes Dollarzeichen).
-OutBuffer	Wert	Legt fest, wie viele Ergebnisobjekte anfallen müssen, bevor diese en bloc an das nächste Cmdlet einer Pipeline weitergegeben werden. Normalerweise wird jedes Ergebnis einzeln sofort an den nächstfolgenden Pipeline-Befehl weitergereicht.
-OutVariable	Wert	Name einer Variablen, in der das Ergebnis des Cmdlets gespeichert werden soll. Nützlich, wenn das Ergebnis eines Cmdlets sowohl in einer Variablen gespeichert als auch in die Konsole ausgegeben werden soll: `$ergebnis = Get-ChildItem` Weisen Sie das Ergebnis zusätzlich einer Variablen zu, wird es in die Konsole ausgegeben und in einer Variablen gespeichert: `Get-ChildItem -OutVariable ergebnis`
-PipelineVariable	Wert	Speichert das aktuelle Pipeline-Ergebnis in einer Variablen, damit es von verschachtelten Schleifen nicht überschrieben wird: `1..254 | Foreach-Object -PipelineVariable A { $_ } \|` `Foreach-Object { 1..254 } \|` `Foreach-Object { "192.168.$A.$_" }` Entspricht diesem klassischen Ansatz: `1..254 \| Foreach-Object {$A = $_; $_ } \|` `Foreach-Object { 1..254 } \|` `Foreach-Object { "192.168.$A.$_" }`
-WarningAction	Wert	Bestimmt, was mit Warnungen geschehen soll, die ein Cmdlet oder eine Funktion mit Write-Warning ausgibt. Erlaubte Werte: Continue: Warnung ausgeben und fortsetzen (Vorgabe) Stop: Warnung ausgeben und abbrechen SilentlyContinue: Warnung unterdrücken, fortfahren Inquire: nachfragen Die Vorgabe wird mit der Variablen $WarningPreference festgelegt und ist normalerweise auf Continue eingestellt.

Tabelle 2.3: Allgemeine Parameter, die für (fast) alle Cmdlets gelten.

Allgemeiner Parameter	Typ	Beschreibung
-WarningVariable	Wert	Name einer Variablen, in der Warnungsmeldungen eines Cmdlets oder einer Funktion gespeichert werden, die mit Write-Warning ausgegeben wurden.
-WhatIf	Switch	Simuliert einen Befehl nur, ohne ihn wirklich auszuführen. Dieser Switch-Parameter wird nur von Cmdlets unterstützt, die tatsächlich Änderungen am System vornehmen würden.
-Confirm	Switch	Fragt für jede Aktion eines Befehls zuerst nach, bevor der Befehl Änderungen vornimmt. Dieser Switch-Parameter steht nur bei Cmdlets zur Verfügung, die Änderungen am System vornehmen.

Tabelle 2.3: Allgemeine Parameter, die für (fast) alle Cmdlets gelten. (Forts.)

Fehlermeldungen unterdrücken

Cmdlets enthalten grundsätzlich einen Fehlerhandler, der die meisten Fehler abfängt und dann entscheidet, was geschehen soll. Die Vorgabe hierfür stammt aus der Variablen $ErrorActionPreference und lautet normalerweise Continue: Das Cmdlet gibt die Fehlermeldung in Rot aus und fährt dann fort.

Bevorzugen Sie im Fehlerfall eine andere Maßnahme, verwenden Sie den Parameter -ErrorAction und geben dahinter die gewünschte Aktion an:

Aktion	Beschreibung
SilentlyContinue	Fehlermeldung unterdrücken und fortfahren. Die Fehlermeldung wird in der Variablen $error protokolliert.
Ignore	Fehlermeldung unterdrücken und fortfahren. Die Fehlermeldung wird nicht protokolliert (ab PowerShell Version 3.0).
Stop	Fehlermeldung ausgeben und anhalten.
Continue	Fehlermeldung ausgeben und fortfahren (die Vorgabe).
Inquire	Nachfragen, welche Aktion durchgeführt werden soll.
Suspend	Nur für Workflows: Workflow anhalten. Der Workflow kann später fortgesetzt werden.

Tabelle 2.4: Mögliche Cmdlet-Aktionen im Fehlerfall.

Die gleichen Aktionen wie die aus Tabelle 2.4 können auch der Variablen $ErrorActionPreference zugewiesen werden und gelten dann für alle Cmdlets, bei denen Sie nicht ausdrücklich mit dem Parameter -ErrorAction eine Auswahl getroffen haben.

Möchte man Fehlermeldungen kurzerhand unterdrücken, weil man sicher weiß, dass die Fehlermeldungen keine Bedeutung für die Aufgabe haben, die man zu lösen hat, kann das Argument SilentlyContinue eingesetzt werden:

```
PS> Get-Process -FileVersionInfo -ErrorAction SilentlyContinue
```

Es funktioniert: Der Befehl liefert Informationen, aber verzichtet auf störende rote Fehlermeldungen für Prozesse, auf die er nicht zugreifen kann. Gleiches gilt, wenn Sie mit Get-ChildItem rekursiv nach Dateien suchen und dabei mögliche Zugriffsverletzungen auf Unterordner ignorieren möchten:

```
PS> Get-ChildItem -Path c:\windows -Filter *.ps1 -Recurse -Force -ErrorAction SilentlyContinue
```

Mit `Stop` kann das Verhalten im Fehlerfall aber auch verschärft werden: Das Cmdlet bricht dann beim ersten Fehler die Arbeit ab.

Unvollständige Parameternamen

Parameternamen müssen nicht vollständig angegeben werden, solange das, was Sie angeben, eindeutig ist. Die folgende Zeile ist also erlaubt, weil `Get-ChildItem` nur einen Parameter kennt, der mit dem Buchstaben `r` beginnt:

```
PS> Get-ChildItem C:\Windows *.dll -r
```

Auch diese Zeile ist erlaubt:

```
PS> Get-ChildItem -pa C:\Windows -fi *.dll -r
```

Kürzen Sie Parameter so stark, dass sie nicht mehr eindeutig zugeordnet werden können, quittiert PowerShell das mit einer roten Fehlermeldung, in der die mehrdeutigen Parameternamen genannt werden:

```
PS> Get-ChildItem -pa C:\Windows -f *.dll -r

Get-ChildItem : Der Parameter kann nicht verarbeitet werden, da der Parametername "f" nicht
eindeutig ist. Mögliche Übereinstimmungen:  -Filter -Force.
```

Wie sich zeigt, gibt es zwei Parameter, die mit *f* beginnen, sodass `-f` nicht eindeutig ist.

Unvollständige Parameternamen sollten indes der Vergangenheit angehören. Inzwischen ist die Autovervollständigung von PowerShell so leistungsfähig, dass es Sie nur einen Druck auf ⇆ kostet, um Parameternamen automatisch ausschreiben zu lassen.

Parameter mit Aliasnamen abkürzen

Häufig benötigte Parameter können darüber hinaus mit sogenannten Aliasnamen versehen sein. Aliasnamen sind zusätzliche Kurznamen, unter denen man Parameter alternativ ansprechen kann.

Sie haben schon den Parameter `-ErrorAction` kennengelernt, mit dessen Hilfe man Fehlermeldungen unterdrücken kann. Der Aliasname dieses Parameters lautet `-ea` (was man bei PowerShell 2.0 noch auswendig wissen musste, weil die Aliasnamen der PowerShell-Parameter in der Hilfe normalerweise nicht verraten werden. Aber ab PowerShell 3.0 lassen sie sich per Befehl ermitteln. Sie erfahren gleich, wie).

Die folgenden beiden Zeilen haben also identische Wirkung und sorgen dafür, dass alle Notepad-Instanzen geschlossen werden. Läuft kein Notepad, wird keine Fehlermeldung mehr ausgegeben:

```
PS> Stop-Process -Name Notepad -ErrorAction SilentlyContinue

PS> Stop-Process -Name Notepad -ea SilentlyContinue
```

Profitipp

Tatsächlich kann man den letzten Aufruf noch sehr viel kürzer fassen, wenn man will. Mit dem folgenden Aufruf findet man heraus, ob es für das Cmdlet `Stop-Process` kürzere Aliasnamen gibt:

```
PS> Get-Alias -Definition Stop-Process

CommandType     Name
-----------     ----
Alias           kill -> Stop-Process
Alias           spps -> Stop-Process
```

Außerdem kann der Parametername verkürzt werden. Damit ergibt sich dieser Aufruf:

```
PS> kill -n Notepad -ea SilentlyContinue
```

Schließlich kann anstelle der Konstanten `SilentlyContinue` auch dessen zugrunde liegender Zahlenwert angegeben werden, was den Aufruf zwar noch kürzer, dafür dann aber beinahe unleserlich macht:

```
PS> kill -n Notepad -ea 0
```

Die Aliasnamen eines Parameters sind normalerweise gut versteckt. Möchten Sie trotzdem wissen, über welche Aliasnamen ein Parameter angesprochen werden kann, setzen Sie die folgende etwas kryptische Zeile Code ein (aber beschweren Sie sich bitte nicht, dass darin Techniken vorkommen, die den aktuellen Videospiellevel übersteigen und erst in den folgenden Kapiteln erklärt werden). Sie liefert eine Übersicht der Parameter von `Get-ChildItem` und der jeweils zugewiesenen Aliasnamen – jedenfalls dann, wenn Sie PowerShell 3.0 verwenden:

```
PS> (Get-Command -Name Get-ChildItem).Parameters.Values | Select-Object -Property Name, Aliases

Name                Aliases
----                -------
Path                {}
LiteralPath         {PSPath}
Filter              {}
(…)
```

Konflikte bei Parameternamen

Übrigens können Sie die Parametererkennung auch ausdrücklich abschalten. Nötig ist das nur in dem seltenen Fall, dass ein Argument genauso lautet wie ein Parametername und deshalb ausdrücklich nicht als Parameter verstanden werden soll. Möchten Sie also unbedingt mit `Write-Host` den Text `-BackgroundColor` ausgeben, käme es normalerweise zu einem Konflikt:

```
PS> Write-Host -BackgroundColor

Write-Host : Fehlendes Argument für den Parameter "BackgroundColor". Geben Sie einen Parameter vom
Typ "System.ConsoleColor" an, und versuchen Sie es erneut.
```

Hier könnten Sie die Parametererkennung mit zwei aufeinanderfolgenden Bindestrichen (`--`) ausdrücklich ausschalten. Alles, was diesen beiden Zeichen folgt (bis zum Zeilenende), wird nicht länger als Parameter erkannt:

```
PS> Write-Host -- -BackgroundColor
-BackgroundColor
```

Wirklich notwendig ist das allerdings nicht. Es würde auch genügen, den Text einfach in Anführungszeichen zu setzen:

```
PS> Write-Host "-BackgroundColor"
-BackgroundColor
```

Neue Cmdlets aus Modulen nachladen

Cmdlets sind grundsätzlich in Modulen beheimatet. Das gilt auch für die Basis-Cmdlets, die PowerShell selbst mitbringt. Um zu sehen, aus welchen Modulen ein Cmdlet eigentlich stammt, fragen Sie einfach `Get-Command`:

```
PS> Get-Command -CommandType Cmdlet

CommandType Name                                ModuleName
----------- ----                                ----------
Cmdlet      Add-Computer                        Microsoft.PowerShell.Management
Cmdlet      Add-Content                         Microsoft.PowerShell.Management
Cmdlet      Add-History                         Microsoft.PowerShell.Core
Cmdlet      Add-Member                          Microsoft.PowerShell.Utility
Cmdlet      Add-PSSnapin                        Microsoft.PowerShell.Core
Cmdlet      Add-Type                            Microsoft.PowerShell.Utility
Cmdlet      Checkpoint-Computer                 Microsoft.PowerShell.Management
Cmdlet      Clear-Content                       Microsoft.PowerShell.Management
Cmdlet      Clear-EventLog                      Microsoft.PowerShell.Management
Cmdlet      Clear-History                       Microsoft.PowerShell.Core
(…)
```

In der Spalte `ModuleName` wird nun das Modul genannt, das das jeweilige Cmdlet beherbergt. PowerShell bringt diese Module mit:

Name	PS	Bedeutung
`BitsTransfer`	2	Zugriff auf den »Background Intelligent Transfer Service« (BITS)
`CimCmdlets`	3	Cmdlets der zweiten Generation für den Zugriff auf WMI-Informationen.
`ISE`	3	Erweiterungsbefehle für den ISE-Editor.
`Microsoft.PowerShell.Archive`	5	Unterstützung für ZIP-Archive.
`Microsoft.PowerShell.Core`	2	Basis-Cmdlets der PowerShell.
`Microsoft.PowerShell.Diagnostics`	2	Cmdlets zu Ereignisprotokollen und Performance-Countern.
`Microsoft.PowerShell.Host`	2	Unterstützung für Start-/Stopp-Transcript.
`Microsoft.PowerShell.Management`	2	Cmdlets für die Verwaltung des Computers.
`Microsoft.PowerShell.ODataUtils`	5	Unterstützung für Open Data Protocol.
`Microsoft.PowerShell.Security`	2	Cmdlets für Signaturen, NTFS-Berechtigungen und Anmeldeinfos.
`Microsoft.PowerShell.Utility`	2	Cmdlets zur Formatierung von Ergebnissen.
`Microsoft.WSMan.Management`	2	Cmdlets zur Verwaltung von Remotezugriffen über WSMan.
`PackageManagement`	5	Installation und Update von Softwarepaketen.
`Pester`	5	Unterstützung für Unit-Tests (Open Source).
`PowerShellGet`	5	Ermöglicht Suche, Nachladen und Aktualisieren von Modulen aus öffentlichen und eigenen Sammlungen. Dieses Modul kann bei älteren PowerShell-Versionen kostenfrei aus dem Internet nachgeladen werden.
`PSDesiredStateConfiguration`	4	Desired State Configuration-Unterstützung.
`PSReadline`	5	Modernisiert die Codedarstellung in der PowerShell-Konsole und fügt beispielsweise Color-Coding hinzu.

Tabelle 2.5: PowerShell-Module im Standardlieferumfang von PowerShell 5.

Name	PS	Bedeutung
PSScheduledJob	3	Cmdlets zur Verwaltung geplanter Aufgaben.
PSScriptAnalyzer	5	Analysiert Skripte und Einhaltung von Best Practices.
PSWorkflow, PSWorkflowUtility	3	Cmdlets zur Arbeit mit Workflows.
TroubleshootingPack	2	Cmdlets zur Arbeit mit Problemlöse-Assistenten.

Tabelle 2.5: PowerShell-Module im Standardlieferumfang von PowerShell 5. (Forts.)

Alle Cmdlets, die aus einem der in Tabelle 2.5 genannten Module entstammen, stehen in jeder PowerShell 5 zur Verfügung. Cmdlets, die aus anderen als den hier genannten Modulen stammen, sind dagegen mit gewisser Vorsicht zu genießen. Bevor Sie solche Cmdlets in eigenen Skriptlösungen einsetzen, sollten Sie sicherstellen, dass sie später auch auf dem gewünschten Zielsystem zur Verfügung stehen.

Insbesondere ab Windows 8 und Server 2012 bringt das Windows-Betriebssystem selbst Tausende nützlicher Cmdlets mit. Sie sind Teil des Betriebssystems und bei älteren Betriebssystemen wie Windows 7 nicht nachrüstbar.

Die aktuell geladenen Module zeigt `Get-Module`. Wenn Sie den Parameter `-ListAvailable` angeben, werden auch alle übrigen verfügbaren Module angezeigt. PowerShell findet automatisch alle Module, die sich an einem der üblichen Speicherorte für Module befinden – was die Frage aufwirft, um welche Orte es sich handelt. Die Umgebungsvariable `$env:PSModulePath` listet diese Orte auf:

```
PS> $env:PSModulePath -split ';'
C:\Users\[UserName]\Documents\WindowsPowerShell\Modules
C:\Program Files\WindowsPowerShell\Modules
C:\WINDOWS\system32\WindowsPowerShell\v1.0\Modules\
```

Neue Module automatisch nachladen

Sofern sich ein Modul in einem der Ordner befindet, die in `$env:PSModulePath` genannt werden, dürfen Sie die darin enthaltenen Cmdlets sofort einsetzen. Sie brauchen zusätzliche Module nicht selbst zu laden. Das erledigt PowerShell ganz automatisch.

Das Betriebssystem Windows ist beispielsweise inzwischen fast vollständig durch PowerShell-Cmdlets verwaltbar.

Die Verwaltung der Netzwerkkarten ist damit beispielsweise ein Kinderspiel, denn für fast alle Fragestellungen gibt es jetzt die entsprechenden Cmdlets:

```
PS> Get-NetAdapterStatistics

Name                            ReceivedBytes ReceivedUnicastPackets SentBytes
----                            ------------- ---------------------- ---------
Ethernet                            430599313                 352480   4545107

PS> Get-NetAdapterAdvancedProperty

Name                    DisplayName                 DisplayValue
----                    -----------                 ------------
Ethernet                Flusssteuerung              Rx- & Tx-aktiviert
Ethernet                Interruptüberprüfung        Aktiviert
```

```
Ethernet                  IPv4-Prüfsummenabladung        Rx- & Tx-aktiviert
Ethernet                  Großes Paket                   Deaktiviert
Ethernet                  Abladung großer Sendungen (... Aktiviert
Ethernet                  Priorität & VLAN               Priorität & VLAN ak...
Ethernet                  Empfangspuffer                 256
Ethernet                  Übertragungsrate und Duplex... Automatische Aushan...
Ethernet                  TCP-Prüfsummenabladung (IPv4)  Rx- & Tx-aktiviert
Ethernet                  Übertragungspuffer             512
Ethernet                  UDP-Prüfsummenabladung (IPv4)  Rx- & Tx-aktiviert
Ethernet                  Adaptives Inter-Frame-Spacing  Aktiviert
Ethernet                  Interruptdämpfungsrate         Adaptiv
Ethernet                  Lokal verwaltete Adresse       --
Ethernet                  Anzahl der zusammengefügten... 128

PS> Get-NetIPAddress

IPAddress         : fe80::e1a2:d0c:f7fc:f49c%12
InterfaceIndex    : 12
InterfaceAlias    : Ethernet
AddressFamily     : IPv6
Type              : Unicast
PrefixLength      : 64
PrefixOrigin      : WellKnown
SuffixOrigin      : Link
AddressState      : Preferred
ValidLifetime     : Infinite ([TimeSpan]::MaxValue)
PreferredLifetime : Infinite ([TimeSpan]::MaxValue)
SkipAsSource      : False
PolicyStore       : ActiveStore
```

Wie praktisch neue Cmdlets aus weiteren Modulen sein können, zeigt das folgende Beispiel, das sich die Möglichkeit des BITS-Diensts zunutze macht, um auch größere Dateien aus dem Internet herunterzuladen, hier zum Beispiel ein NASA-HD-Video (Größe: 567 MB):

```
$url = 'http://s3.amazonaws.com/akamai.netstorage/anon.nasa-global/NASAHD/CX_Mission/CX_Missi
on_Short3b_HDweb.wmv'
Start-BitsTransfer -Source $url -Destination $env:temp\video1.wmv -DisplayName 'Nasa-Video'
Invoke-Item -Path $env:temp\video1.wmv
```

Listing 2.1: NASA-Video herunterladen und anzeigen.

Sobald das Video heruntergeladen ist, startet der Windows Media Player bzw. das in Windows entsprechend registrierte Standardprogramm, spielt es ab und beschert wundervolle Ansichten auf unseren Planeten (Abbildung 2.7).

Abbildung 2.7: Ein NASA-Video: von PowerShell heruntergeladen und abgespielt.

Der Download wird im Vordergrund durchgeführt und läuft nur, solange auch PowerShell ausgeführt wird. Genauso gut hätten Sie den Download aber auch still im Hintergrund unabhängig von der PowerShell-Sitzung und über viele Tage verteilt ausführen lassen können – einfach nur durch den zusätzlichen Parameter `-Asynchronous`:

```
Start-BitsTransfer -Source 'http://s3.amazonaws.com/akamai.netstorage/anon.nasa-global/NASAHD/CX_
Mission/CX_Mission_Short3b_HDweb.wmv' -Destination $env:tmp\video2.wmv -Asynchronous


JobId             DisplayName      TransferType     JobState         OwnerAccount
-----             -----------      ------------     --------         ------------
1365d9d6-938c-... BITS Transfer    Download         Connecting       DEMO5\w7-pc9
```

Anschließend könnte PowerShell beendet und der Computer sogar heruntergefahren oder neu gestartet werden. Der Download wird dann bei Bedarf fortgesetzt. Wenn es Sie interessiert, können Sie mit `Get-BitsTransfer` nachschauen, wie weit der Download fortgeschritten ist:

```
PS> Get-BitsTransfer

JobId             DisplayName      TransferType     JobState         OwnerAccount
-----             -----------      ------------     --------         ------------
1365d9d6-938c-... BITS Transfer    Download         Transferred      DEMO5\w7-pc9
```

Nutzen Sie den asynchronen Modus, muss der Download mit `Complete-BitsTransfer` abgeschlossen werden – vorher stehen die heruntergeladenen Dateien nicht zur Verfügung. Verwenden Sie zum Beispiel eine Zeile wie diese, um alle Downloads abzuschließen:

```
PS> Get-BitsTransfer | Complete-BitsTransfer
```

Die neuen Befehle des Moduls machen deutlich, dass mit jeder neuen Erweiterung auch Ihre Möglichkeiten wachsen, ohne dass sehr viel neues Wissen dazu nötig wäre. Die Cmdlets aus dem Modul `BitsTransfer` jedenfalls folgen genau den gleichen Regeln wie alle übrigen Cmdlets, die Sie schon kennengelernt haben.

Alias: Zweitname für Cmdlets

PowerShell baut Brücken in die Vergangenheit und nutzt dazu sogenannte »historische Aliase«. Diese helfen Anwendern, die bereits früher mit der Windows-Eingabeaufforderung oder in Unix-Shells gearbeitet haben oder heute noch damit arbeiten, schnell den passenden Befehl zu finden. Aliase kennen Sie aus Actionthrillern: »Clark Kent alias Superman«. Es sind also nur Zweitnamen, Illusionen. Dank dieser Aliase funktionieren viele alte Befehle in PowerShell auf den ersten Blick fast wie früher:

```
PS> dir c:\windows
PS> md c:\newfolder
PS> ren c:\newfolder ordner_neu
PS> del c:\ordner_neu
```

Aliase sind keine neuen Befehle

Katerstimmung entwickelt sich höchstens, sobald Sie versuchen, mit diesen »alten« Befehlen handfest zu werden und komplexere Dinge anzustellen. Der folgende Befehl lieferte in klassischen Befehlskonsolen beispielsweise alle **.log*-Dateien aus dem Windows-Ordner und seinen Unterordnern, bei PowerShell dagegen eine rote Fehlermeldung:

```
PS> dir c:\windows /S

dir : Das zweite Pfadfragment darf kein Laufwerk oder UNC-Name sein.
Parametername: path2
```

Der Grund: `dir` ist gar kein eigenständiger Befehl und entspricht schon gar nicht dem alten `dir` aus einer normalen Windows-Eingabeaufforderung. `dir` ist lediglich ein Verweis auf das PowerShell-Cmdlet, das dem alten Befehl am nächsten kommt, nämlich `Get-ChildItem`.

Aliase können also dabei helfen, das zuständige neue Cmdlet zu finden. Danach allerdings müssen Sie sich mit diesem und seinen Parametern auseinandersetzen:

```
PS> dir c:\windows -Recurse -ErrorAction SilentlyContinue
```

Das muss nicht lästig sein, denn im Gegenzug erhalten Sie dafür natürlich wieder das komfortable IntelliSense (Abbildung 2.8).

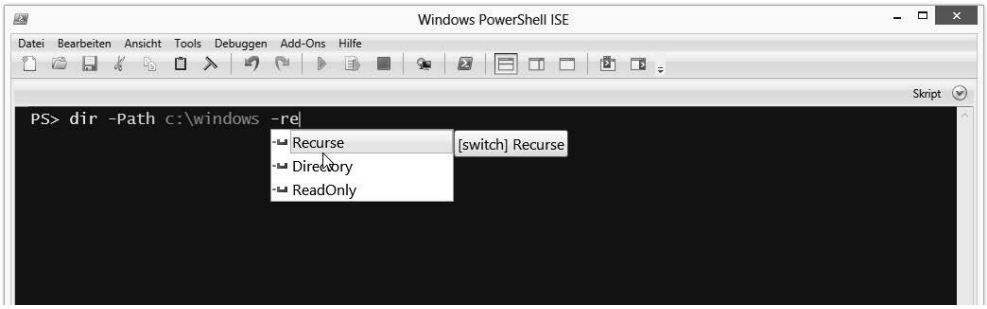

Abbildung 2.8: Aliase, die auf Cmdlets verweisen, liefern die gleiche umfangreiche IntelliSense-Unterstützung.

Befehlstypen ermitteln

Möchten Sie alle Aliase sehen, die PowerShell mitbringt, verwenden Sie `Get-Alias` (oder das Laufwerk *Alias:*). `Get-Alias` kann Ihnen mit dem Parameter `-Definition` auch Aliase heraussuchen, die auf einen bestimmten Befehl verweisen. Die folgende Zeile findet alle Aliase für das Cmdlet `Get-ChildItem`:

```
PS> Get-Alias -Definition Get-ChildItem

CommandType     Name
-----------     ----
Alias           dir -> Get-ChildItem
Alias           gci -> Get-ChildItem
Alias           ls -> Get-ChildItem
```

Im Zweifelsfall deckt `Get-Command` auf, um was für einen Befehlstyp es sich jeweils handelt. So finden Sie zum Beispiel heraus, welche Befehlsarten hinter Aliasnamen in Wirklichkeit stecken:

```
PS> Get-Command -Name ipconfig, echo, dir, notepad, control, devmgmt, wscui, lpksetup

CommandType     Name
-----------     ----
Application     ipconfig.exe
Alias           echo -> Write-Output
Alias           dir -> Get-ChildItem
Application     notepad.exe
Application     control.exe
Application     devmgmt.msc
Application     wscui.cpl
Application     lpksetup.exe
```

Wie sich zeigt, sind `echo` und `dir` in Wirklichkeit Aliase (Verweise) auf die Cmdlets `Write-Output` und `Get-ChildItem`. Alle übrigen Befehle sind vom Typ `Application`, also eigenständige Programme mit der Dateierweiterung *.exe*.

Während Befehle vom Typ `Application` vollkommen autark sind und deshalb bei PowerShell exakt genauso funktionieren wie anderswo, richtet sich das Verhalten der Aliase nach dem Befehl, auf den sie in Wirklichkeit verweisen. Weil `dir` in Wirklichkeit auf das Cmdlet `Get-ChildItem` verweist und Sie also in Wirklichkeit Letzteres aufrufen, wenn Sie `dir` verwenden, gelten für `dir` dieselben Regeln, als hätten Sie `Get-ChildItem` geschrieben. Sie dürfen daher nicht die

Parameter des alten Befehls `dir` einsetzen, sondern müssen ausschließlich auf die Parameter des Cmdlets `Get-ChildItem` zurückgreifen.

Klassische cmd.exe-Interpreterbefehle sind Cmdlets

Warum hat PowerShell Befehle wie `dir` und `echo` überhaupt in Aliase verwandelt und mit eigenen Cmdlets implementiert – und nicht einfach so gelassen, wie sie waren? Befehle wie `ipconfig` und `ping` funktionieren in PowerShell doch ebenfalls noch genauso wie früher.

Wenn ein Befehl eine eigenständige Anwendung ist, so wie `ipconfig` und `ping`, ändert sich ihr Verhalten in keiner Weise. PowerShell ruft analog zur Eingabeaufforderung ja lediglich das entsprechend Programm auf. Die Befehle `dir` und `echo` (sowie einige weitere) waren hingegen nie eigenständige Anwendungen. Auch schon zu »DOS-Zeiten« (also vor sehr langer Zeit) gab es keine Anwendung wie *dir.exe* oder *echo.exe*. Stattdessen waren `dir` und `echo` Teil des alten Befehlszeileninterpreters und können über diesen heute noch eingesetzt werden – auch von PowerShell aus:

```
PS> cmd.exe /c dir c:\windows
 Volume in Laufwerk C: hat keine Bezeichnung.
 Volumeseriennummer: 18D6-E089

 Verzeichnis von c:\windows

15.08.2012  20:41    <DIR>          .
15.08.2012  20:41    <DIR>          ..
26.07.2012  10:13    <DIR>          addins
26.07.2012  10:12    <DIR>          AppCompat
26.07.2012  12:27    <DIR>          apppatch
26.07.2012  09:22    <DIR>          assembly
(…)
```

Weil heute aber PowerShell der neue Befehlszeileninterpreter ist und nicht mehr *cmd.exe*, fallen alle integrierten alten Konsolenbefehle aus *cmd.exe* weg und wurden deshalb von PowerShell mit den eigenen Mitteln – also als Cmdlets – neu erfunden.

Da sich die meisten Befehle in der klassischen *cmd.exe* mit Dateihandling beschäftigt haben, kann man sich auch einen Großteil der historischen Aliase auf diese Weise sichtbar machen und erfährt auf einen Blick, wie die neuen Cmdlets für diese alten Befehle heißen – jedenfalls wenn man weiß, dass die Cmdlets für das Dateihandling in der Regel in ihrem Substantiv die Schlüsselwörter `Item`, `Content` oder `Location` tragen:

```
PS> Get-Alias -Definition *-Item*, *-Content*, *-Location*

CommandType     Name
-----------     ----
(…)
Alias           copy -> Copy-Item
(…)
Alias           del -> Remove-Item
Alias           erase -> Remove-Item
(…)
Alias           move -> Move-Item
(…)
Alias           rd -> Remove-Item
Alias           ren -> Rename-Item
(…)
```

```
Alias           rmdir -> Remove-Item
(…)
Alias           cat -> Get-Content
(…)
Alias           type -> Get-Content
Alias           cd -> Set-Location
Alias           chdir -> Set-Location
```

PowerShell verwendet Aliase übrigens nicht nur, um erfahrenen Anwendern den Umstieg zu PowerShell zu versüßen. Auch PowerShell-Anwender greifen im hektischen Alltag gern mal zu diesen Kurznamen. `gps` ist beispielsweise viel schneller gezückt als `Get-Process`, wenn man kurz die laufenden Prozesse zu überprüfen hat.

Eigene Aliase anlegen

Sie dürfen auch gern eigene Aliase anlegen, wenn Sie möchten. Dazu setzen Sie `Set-Alias` ein. Die folgende Zeile legt einen neuen Alias namens `edit` an, der den Windows-Editor startet. Künftig startet also *notepad.exe*, sobald Sie den Befehl `edit` eingeben.

```
PS> Set-Alias -Name edit -Value notepad
```

Allerdings gilt Ihr neuer Alias nur in der aktuellen PowerShell-Sitzung, die ihn umgehend wieder vergisst, sobald Sie sie schließen. Eigene Aliase ergeben erst dann richtig Sinn, wenn Sie persönliche Einstellungen mithilfe eines Profilskripts speichern. Wie das funktioniert, erfahren Sie in Kapitel 5.

Der Nutzen eigener Aliase ist ohnehin generell begrenzt. Denn PowerShell-Skripte, die von Ihnen definierte Aliase verwenden, funktionieren nicht auf anderen Computern. Eigene Aliase sind deshalb nur für PowerShell-Profis gedacht, die einen Großteil ihrer Zeit mit der interaktiven PowerShell-Konsole verbringen und ihre *ganz persönliche Befehlsumgebung* etwas tippfreundlicher gestalten möchten.

Es gibt noch einen Haken: Eigene Aliase können sogar das System beeinträchtigen. Denn sie tragen die höchste Befehlspriorität und gewinnen bei Namensgleichheiten immer. Mit Aliasen kann man also (gewollt oder nicht) andere Befehle schachmatt setzen oder »verbiegen«. Die Befehle verrichten dann plötzlich etwas ganz anderes. Die folgenden beiden Zeilen setzen den Befehl `ping` außer Kraft und starten stattdessen den Windows-Editor:

```
PS> Set-Alias -Name ping -Value notepad
PS> Set-Alias -Name ping.exe -Value notepad
```

Aliase sind im Übrigen reine Befehlsersetzungen. Die Befehlsparameter kann man nicht beeinflussen oder erweitern. Der Alias verhält sich daher in puncto Parameter exakt so wie das Original. Nur der reine Befehlsname kann mit einem Alias abgekürzt werden. Wer mehr will, sollte unauffällig zu Kapitel 11 vorblättern.

Zusammenfassend ist also festzustellen: Die in PowerShell integrierten Aliase sind praktisch und dürfen bedenkenlos eingesetzt werden, um Tipparbeit zu sparen oder in alter Gewohnheit mit klassischen Befehlsnamen zu arbeiten. Neue Aliase kann man zwar auf Wunsch mit `Set-Alias` oder `New-Alias` hinzufügen, aber sinnvoll ist das indes häufig nicht. Spätestens wenn Sie damit beginnen, PowerShell-Skripte zu schreiben, sollten Sie Aliase ausmustern und stattdessen ausschließlich die Originalbefehle verwenden.