O'REILLY®

5. Auflage
Für PowerShell 7 und
Windows PowerShell 5

PowerShell
kurz & gut

O'Reillys Taschenbibliothek

Thorsten Butz

5. AUFLAGE

PowerShell

kurz & gut

Thorsten Butz

O'REILLY®

Thorsten Butz

Lektorat: Alexandra Follenius
Fachgutachten: Holger Voges
Korrektorat: Sibylle Feldmann, *www.richtiger-text.de*
Satz: III-satz, *www.drei-satz.de*
Herstellung: Stefanie Weidner
Umschlaggestaltung: Karen Montgomery, Michael Oréal, *www.oreal.de*

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

ISBN:
Print 978-3-96009-145-5
PDF 978-3-96010-405-6
ePub 978-3-96010-406-3
mobi 978-3-96010-407-0

5. Auflage 2021

Wieblinger Weg 17
69123 Heidelberg

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

Hinweis:
Dieses Buch wurde auf PEFC-zertifiziertem Papier aus nachhaltiger Waldwirtschaft gedruckt. Der Umwelt zuliebe verzichten wir zusätzlich auf die Einschweißfolie.

Schreiben Sie uns:
Falls Sie Anregungen, Wünsche und Kommentare haben, lassen Sie es uns wissen: kommentar@oreilly.de.

5 4 3 2 1 0

Inhalt

Über dieses Buch

Dieses Buch vermittelt Ihnen die zentralen Konzepte der PowerShell. Unabhängig von der spezifischen Version und dem darunterliegenden Betriebssystem erlernen Sie anhand vieler Beispiele notwendiges und sinnvolles Wissen über Cmdlets, über die Objektorientierung der Shell, das Pipelining, das Erstellen von Funktionen und Skripten, über Fernabfragen (Remoting), den Umgang mit Webservices und das Erweitern des Funktionsumfangs.

In Kapitel 2, *Hallo PowerShell!*, erhalten Sie zu Beginn einen komprimierten Überblick über die zentralen Konzepte der PowerShell. Zahlreiche Codebeispiele, die Sie sowohl unter Linux als auch unter Windows oder macOS, in PowerShell 7 oder in der Windows PowerShell 5 ausführen können, ermöglichen Ihnen einen schnellen Einstieg in das Thema.

Bei näherer Betrachtung werden Sie aber auch feststellen, dass insbesondere die kleinen, auf den ersten Blick unscheinbaren Unterschiede zwischen den PowerShell-Versionen und den unterstützten Betriebssystemen ein enormes Fehlerpotenzial mit sich bringen. Aus diesem Grund werden wir in Kapitel 20, *Neuerungen in PowerShell 7*, ausführlich die neuen Features der PowerShell 7 erläutern, was jenen Leserinnen und Lesern den Umstieg erleichtern wird, die schon über gute

(Windows-)PowerShell-Kenntnisse verfügen. Starten Sie mit diesem Kapitel, wenn Sie mögen.

PowerShell 7 ist, im Gegensatz zur Windows PowerShell 5, kein fester Bestandteil von Windows. PowerShell 7 ist an manchen Stellen der Windows PowerShell 5 voraus, an einigen Stellen ist sie funktional eingeschränkt. Gewusst wie, kann man fast alle diese Beschränkungen aufheben und mittels Fernabfragen und einer integrierten Kompatibilitätsschicht das Beste aus beiden Welten zusammenführen. Wo immer es erforderlich ist, werde ich Sie explizit auf Unterschiede und Inkompatibilitäten hinweisen. Im Kern erlernen und vertiefen Sie den Umgang mit der PowerShell, und zwar gänzlich unabhängig von der spezifischen Version und Ihrem favorisierten Betriebssystem.

Im PowerShell-Universum gibt es viele weitere spannende Entwicklungen, die wir in diesem Buch nur kurz streifen wollen, allen voran der ebenfalls plattformunabhängige Editor *Visual Studio Code* (VSCode) und die neue Applikation *Windows Terminal*.

Unter dem nachfolgenden Link finden Sie ein Repository, in dem Sie die im Buch behandelten Codebeispiele herunterladen können. Aufgrund der beschränkten Zeichenzahl, die der Druck in einer Zeile erlaubt, sind viele Codebeispiele auf eine Weise umbrochen worden, die in der PowerShell zu Fehlern führen würde. Zugunsten der Lesbarkeit habe ich auf die grundsätzlich mögliche Entwertung des Zeilenumbruchs (mit dem Gravis/Backtick) verzichtet. Schauen Sie bitte in das bereits erwähnte Repository – dort sind alle Beispiele syntaktisch korrekt umbrochen und kommentiert. Des Weiteren finden Sie dort zu den einzelnen Kapiteln weiterführende Beispiele und Aktualisierungen:

https://github.com/thorstenbutz/kurzundgut

Das Wichtigste ist, wie so oft, eigene Erfahrungen zu sammeln.

KAPITEL 1

Microsofts (R)Evolution

Seit der Veröffentlichung der *Windows PowerShell* im Jahr 2006 hat die objektorientierte Shell aus dem Hause Microsoft eine erstaunliche Metamorphose durchlaufen. Ursprünglich exklusiv für die Windows-Plattform konzipiert, entwickelte sie sich dort innerhalb weniger Jahre zur Lingua franca der Systemadministration. Microsoft, vormals auf einfache grafische Bedienkonzepte fokussiert, stellte zunehmend die Automation der Infrastruktur in den Mittelpunkt. Automation und Scripting erfuhr einen für das Microsoft-Ökosystem ungeahnten Aufschwung.

Mit einem Wechsel an der Spitze erfindet sich Microsoft im Jahr 2014 vollständig neu: Der klassische Softwarevertrieb verliert unter der Führung von Satya Nadella immer weiter an Bedeutung, in den Mittelpunkt rückt ein komplexes, Abonnement-gestütztes Dienstleistungsmodell, das den Betrieb eigener Rechenzentren in Unternehmen (»On Premise«) Stück für Stück obsolet machen soll. So wird aus dem klassischen Outlook-Exchange-Gespann das Cloud-Abonnement Microsoft 365 (vormals Office 365).

Unbeobachtet von der breiten Öffentlichkeit wandelt sich Microsoft intern vom Linux- und Open-Source-skeptischen Unternehmen zum Unterstützer quelloffener Lösungen. Schon 2013 geht Microsoft mit der Firma Docker eine strategische Partnerschaft ein, 2018 wird Microsoft Platinum-Sponsor der *Linux Foundation* und erwirbt im

selben Jahr die populäre Softwareverwaltungsplattform GitHub. Unter dem Dach seiner Cloud-Plattform *Azure* finden sich Linuxbasierte Anwendungen gleichrangig zu Windows-spezifischen Angeboten. Für Android- und iOS-Smartphones bietet Microsoft Software an, das Eigengewächs Windows Phone wird hingegen 2015 aufgegeben.

Lässt man diese (R)Evolution Revue passieren, überrascht es wenig, dass die Weiterentwicklung der Windows PowerShell nach zehn Jahren eingestellt wurde. An ihre Stelle tritt 2018 mit Version 6 eine in wesentlichen Teilen neu entwickelte PowerShell (ursprünglich »PowerShell Core« genannt), die neben Windows auch Linux und macOS unterstützt. Diese neue Generation der PowerShell wird nach wie vor federführend von Microsoft-Mitarbeitern entwickelt, ist aber quelloffen auf der bereits erwähnten Plattform GitHub jedem Interessierten zugänglich und steht unter der MIT-Lizenz, einer anerkannten FOSS-Lizenz (*Free and Open Source Software*). Zahlreiche Personen und Organisationen tragen seither zur Weiterentwicklung der Shell bei.

Das lange Leben der Windows PowerShell 5

Die PowerShell basiert wesentlich auf Komponenten des .NET Framework. Diese mächtige Sammlung von Programmbibliotheken und Schnittstellen (zum Beispiel zur Anbindung von Datenbanken, dem Dateisystem oder der Netzwerkkommunikation) existiert in zwei Editionen:

- *.NET Framework für Windows-Betriebssysteme*
  Diese traditionelle .NET-Variante wird gern als »FullCLR« umschrieben, da sie alle Komponenten des

Frameworks bereitstellt. Die Abkürzung CLR steht für *Common Language Runtime*, was im Kern die Ausführungsschicht des .NET Framework beschreibt.

- *.NET Framework Core für Windows, Linux und macOS*
  Im Funktionsumfang (noch) eingeschränkte Neuentwicklung, auch »CoreCLR« genannt. Konzipiert als Nachfolger des klassischen .NET Framework.

Mit dem Neustart der Entwicklung hin zu einer plattformübergreifenden Shell war der Wechsel auf das *.NET Framework Core* verbunden. Die Entwickler mussten große Teile des Codes neu schreiben und dennoch zahlreiche funktionale Einschränkungen in Kauf nehmen, da der Funktionsumfang der CoreCLR stark eingeschränkt war. Diese neue Generation der PowerShell (Version 6+) ist eine installierbare Anwendung. Anders als die etablierte Windows PowerShell 5, wie sie in Windows 10 und in Windows Server ab Version 2016 integriert ist, ist sie kein Bestandteil des Betriebssystems und ist kein Teil der Lizenzvereinbarung zwischen dem Hersteller und dem Kunden.

Microsoft gewährt Geschäftskunden bis zu zehn Jahre (unter gewissen Umständen sogar noch länger) Unterstützung beim Betrieb von Windows Server und Windows 10. Werden Fehler gefunden, stellt der Hersteller Updates für das Betriebssystem bereit. Hierzu gehört nach wie vor die Windows PowerShell 5. Mit jedem neuen *Long Term Servicing*-Release von Windows verlängert sich die garantierte Unterstützung für die alte Generation somit um weitere zehn Jahre – bis zu dem Tag, an dem die Windows PowerShell 5 aus dem Betriebssystem entfernt wird.

Das .NET Framework Core hat einen wesentlich kürzeren Lebenszyklus als die Betriebssysteme aus dem gleichen Haus. Die Core Edition wird bestenfalls für drei Jahre mit Updates versorgt, anschließend ist man gezwungen, auf die nachfolgende Version zu wechseln. Sie ahnen sicherlich, in welchem Dilemma sich aufgrund dieser Umstände Microsofts PowerShell-Team befindet: Die PowerShell ist ab Version 6 nicht mehr Teil des Windows-Betriebssystems, aus Gründen der Kompatibilität verteilt das Windows-Team aber noch auf Jahre hinaus die Windows PowerShell 5, die aller Voraussicht nach jedoch keinerlei Funktionsupdate mehr erfahren wird. Sie verbleibt als Artefakt der Vergangenheit im Fokus der Anwender, da sie mit dem Betriebssystem ausgeliefert wird. Neue Versionen müssen dahin gehend heruntergeladen und bereitgestellt werden, bestehende Skripte müssen auf ihre Kompatibilität hin getestet werden – und das Ganze vor dem Hintergrund, dass PowerShell 6 und 7 in wesentlichen Teilen funktionsreduziert sind im Vergleich zu ihrem Urahn. Neue Versionen der PowerShell müssen also sehr überzeugende Verbesserungen bieten, um den Umstieg auf die aktuelle Generation zu beschleunigen.

Weiterführende Informationen zum .NET Core Lifecycle:

https://docs.microsoft.com/powershell/scripting/powershell-support-lifecycle

https://dotnet.microsoft.com/platform/support/policy/dotnet-core

Dieser Zwiespalt zwischen den Generationen spielt für den Einstieg in die PowerShell jedoch eine untergeordnete Rolle. Dieses Buch stellt die grundlegenden Ideen und Konzepte der PowerShell in den Mittelpunkt. Wo es sinnvoll oder erforderlich ist, erläutere ich versionsabhängige Unterschiede im Detail. Konzentrieren Sie sich auf die für

Sie bedeutsamen Anwendungsgebiete und erweitern Sie Ihre Fähigkeiten im Umgang mit der Shell. Ihr Know-how ist versionsunabhängig.

KAPITEL 2

Hallo PowerShell!

Die PowerShell ist ein ursprünglich von Microsoft entwickeltes plattformübergreifendes Werkzeug zur Automation und Konfiguration der IT-Infrastruktur, bestehend aus einer Skriptsprache und einem Kommandozeileninterpreter, der *Shell*. PowerShell ist eine objektorientierte Skriptsprache, die die Klassen und Datentypen des .NET Framework mit einer benutzerfreundlichen Befehlssyntax zugänglich macht.

Die PowerShell steht sowohl im Quelltext als auch kompiliert für zahlreiche Betriebssysteme auf den Projektseiten zum Download bereit: *https://github.com/PowerShell/PowerShell*.

Seit Version 6 steht der Code unter der *MIT-License*. Details zur Installation und den betriebssystemspezifischen Unterschieden finden Sie in Kapitel 3, *Installieren und Aktualisieren der PowerShell*.

```
Get-Uptime

Get-FileHash -Path 'helloworld.ps1'

Test-Connection -TargetName 'www.oreilly.de' -TcpPort 80
```

PowerShell-Befehle sind idealerweise auch für Einsteiger gut les- und erinnerbar. Sogenannte Cmdlets, wie `Get-`

`Uptime` im Beispiel oben, bestehen prinzipiell aus einem Verb (`Get`) und einem Nomen (`Uptime`) und sind exklusiv in der PowerShell ausführbar. Die Shell unterstützt den Anwender darüber hinaus mithilfe der Autovervollständigung beim (Wieder-)Finden der gewünschten Befehle, Parameter und Argumente.

```
Get-Uptime -since

Get-Process -Name 'pwsh'

Compress-Archive -Path '~/*' -DestinationPath 'backup.zip'
```

Einem Cmdlet folgen gegebenenfalls Parameter, gut erkennbar an dem vorangestellten Bindestrich, und Argumente. Im Beispiel oben verwenden wir die Tilde ~, die in der PowerShell (und vielen anderen Shells) das Heimatverzeichnis repräsentiert. In vielen Fällen unterstützt Sie die Shell mithilfe der Tabulatortaste nicht nur beim Vervollständigen des Cmdlet-Namens und dessen Parametern, auch Argumente können in einigen Fällen automatisch vervollständigt werden.

Hinweis

Um Ihnen Werte anbieten zu können, die Sie mithilfe der Autovervollständigung als Argument übergeben, muss der PowerShell-Interpret gültige Werte bereits ermitteln, bevor Sie diese Werte eintippen. Im Beispiel `Get-Process -Name 'pwsh'` kann die PowerShell Ihnen `'pwsh'` anbieten, da Sie im Hintergrund die Liste aktiver Prozesse bereits abgerufen hat. Das Gleiche wird mit `Compress-Archive -Path '~/*' -DestinationPath 'backup.zip'` nicht funktionieren – woher sollte das System wissen, wie Sie Ihre Zip-Datei nennen möchten?

Cmdlets sind das Herzstück der PowerShell. Die Ausführung von Programmcode ist aber nicht auf Cmdlets beschränkt. Sie können ebenfalls die althergebrachten

Kommandozeilenprogramme (englisch *Native Commands*) verwenden.

```
nslookup www.oreilly.de

ping www.oreilly.de
```

Ebenso sind einfache (Rechen-)Operationen unmittelbar ausführbar.

```
42 + 23

60 * 60

7 % 2

'Power' + 'Shell'
```

Betrachtet man die PowerShell ein wenig technischer, stößt man schnell auf das bereits eingangs erwähnte .NET Framework.

```
[System.Net.Dns]::GetHostByName('www.oreilly.de')

[System.Net.Sockets.TcpClient]::new('188.40.159.226',80)
```

Das .NET Framework ist Microsofts zentrale Sammlung von Klassenbibliotheken und Schnittstellen zur Entwicklung von Software in Hochsprachen wie C#. Mittels PowerShell erlangt der Anwender Zugriff auf diese Sammlung Zehntausender Programmbausteine, wodurch sich der Einsatzbereich der Shell explosionsartig erweitert. Mehr zu der speziellen Syntax in diesem Kontext erfahren Sie im Abschnitt »Rückgriff auf das .NET Framework« auf Seite 115.

Das LEGO®-Prinzip

Als Klassenbibliothek bezeichnet man eine Sammlung von Standard-implementierungen häufig benötigter Datenstrukturen und Algorithmen beispielsweise zum Dateizugriff, zur Netzwerkkommunikation, zum Zugriff auf Datenbanken oder zur Darstellung grafischer Benutzeroberflächen. Diese Softwarebausteine werden typischerweise vom Softwareentwickler zu einem größeren Ganzen zusammengefügt und mit eigener Programmlogik ergänzt. Schlicht ausgedrückt, kann man dies mit dem sehr erfolgreichen Spielprinzip eines dänischen Spielzeugherstellers vergleichen: Einzelne Bausteine lassen sich auf eine vorgegebene Weise zusammenstecken und erlauben so sehr einfach das Erschaffen komplexer Gegenstände aus Standardelementen.

Was ist ein Cmdlet?

Im Idealfall finden Sie in der PowerShell ein sogenanntes *Cmdlet*, das die von Ihnen gewünschte Aufgabe erfüllen kann. Angenommen, Sie suchten nach einer Aufzählung aller aktiven Prozesse.

```
Get-Process
```

Nehmen wir weiter an, dass Sie nun im Anschluss einen ganz bestimmten Prozess suchen und beenden wollen.

```
Get-Process -Name 'pwsh'
```

```
Stop-Process -Name 'pwsh'
```

Vergleichen wir diese Codebeispiele mit dem nachfolgenden Listing.

```
[System.Diagnostics.Process]::GetProcesses()

[System.Diagnostics.Process]::GetProcessesByName('pwsh').kill()
```

Ganz offenkundig handelt es sich um die gleiche Funktionalität: Zunächst fragen Sie nach aktiven Prozessen, im Anschluss beenden Sie einen spezifischen Prozess. Dies ist kein Zufall, es handelt sich im Inneren um denselben Code. Ein Cmdlet, hier `Get-Process`, ist oftmals nur eine benutzerfreundliche Hülle, ein Frontend für eine Softwareroutine, die unabhängig von der PowerShell implementiert wurde.

Der Begriff Cmdlet (gesprochen Commandlet) ist ein Kunstwort, es beschreibt ein Kommando, das einzig in der PowerShell ausführbar ist und der benutzerfreundlichen Verb-Nomen-Syntax folgt. Allgemein spricht man von einem *Befehl* (englisch *Command*), wenn die genaue Unterscheidung (Cmdlet, Applikation …) irrelevant ist oder man mehrere Befehle zu einer komplexen Anweisung zusammenführt, zum Beispiel mittels Pipelining. Ein Befehl kann im allgemeinen Sprachgebrauch praktisch alles sein, was den Computer dazu bringt, eine Aktion auszuführen.

Cmdlets im engeren Sinn werden in einer .NET-Sprache wie C# entwickelt und in **.dll*-Dateien[1] übersetzt. Sie werden auch binäre Cmdlets genannt. Zusammen mit einem beschreibenden Manifest, einer **.psd1*-Datei und gegebenenfalls weiteren Ressourcen wie Hilfedateien werden diese als sogenannte *Module* bereitgestellt.

```
Get-Command -Name 'Get-Process' | Select-Object -property

  CommandType, Name, Version, Source, DLL

Get-Command -CommandType 'Cmdlet'
```

Zu welchem Modul ein Cmdlet gehört, erkennen Sie in der Ausgabe von `Get-Command` (siehe Beispiel oben) an der Eigenschaft `Source`. So können Sie unter anderem nachvollziehen, welche Cmdlets ähnliche Aufgaben erfüllen.

Binären Cmdlets gleichgestellt sind sogenannte *Advanced Functions*, wie etwa `Compress-Archive`. Advanced Functions sind in der Sprache PowerShell implementiert, ihre Funktionsweise können Sie jederzeit im Quelltext nachvollziehen.

```
Get-Command -Name 'Compress-Archive'

Get-Command -Name 'Compress-Archive' |

  Select-Object -ExpandProperty Definition

Get-Command -CommandType 'Function'
```

Begriffsverwirrung

In Version 1 der PowerShell existierten ausschließlich binäre Cmdlets, sie wurden in einer .NET-Hochsprache programmiert und übersetzt/kompiliert. Mit Einführung der flexibleren Advanced Functions in PowerShell-Version 2 wurde das Kunstwort Cmdlet gleichsam zum Oberbegriff, was bis zum heutigen Tag mancherorts für Verwirrung sorgt. So erweckt die Ausgabe von `Get-Command` den Eindruck, nur binäre Cmdlets seien *echte* Cmdlets. Technisch ist dies weitgehend irrelevant. Aus Sicht des Anwenders sind jedoch das einprägsame Benennungskonzept, die Objektorientierung und das Pipelining die entscheidenden Vorteile gegenüber Applikationen wie

ipconfig.exe (Windows) oder *ip* bzw. *ifconfig* (Linux, macOS). Und dies umfasst mehr als nur die binären Cmdlets, sodass ich in diesem Buch Cmdlet als Oberbegriff verwenden werde.

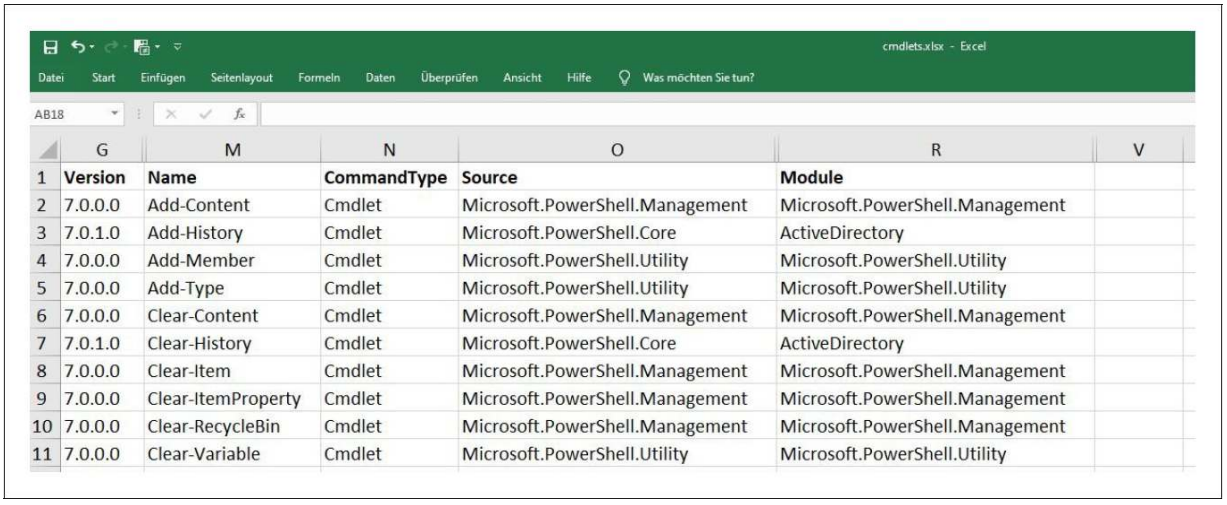
Objektorientierung und Pipeline

Ich habe die Pipeline bereits im vorhergehenden Abschnitt in einigen Beispielen verwendet. Lassen Sie uns diese Beispiele aufgreifen und erweitern.

```
Get-Command -Name 'Get-Process'

Get-Command -Name 'Get-Process' | Select-Object -Property *
```

Wenn Sie den ersten Befehl ausführen, wird Ihnen unmittelbar auffallen, dass eine interessante Information fehlt: Binäre Cmdlets sind in DLL-Dateien gespeichert. In welcher? Das erfahren wir mit der Standardausgabe des `Get-Command`-Befehls zunächst nicht. Stellen Sie sich für einen Moment vor, dass ein Freund für Sie eine Liste aller PowerShell-Kommandos mithilfe einer Tabellenkalkulation erstellt hat, wie in Abbildung 2-1 zu sehen, und Ihnen diese Liste als **.xlsx*- oder **.ods*-Datei zukommen lässt.



	G	M	N	O	R	V
1	**Version**	**Name**	**CommandType**	**Source**	**Module**	
2	7.0.0.0	Add-Content	Cmdlet	Microsoft.PowerShell.Management	Microsoft.PowerShell.Management	
3	7.0.1.0	Add-History	Cmdlet	Microsoft.PowerShell.Core	ActiveDirectory	
4	7.0.0.0	Add-Member	Cmdlet	Microsoft.PowerShell.Utility	Microsoft.PowerShell.Utility	
5	7.0.0.0	Add-Type	Cmdlet	Microsoft.PowerShell.Utility	Microsoft.PowerShell.Utility	
6	7.0.0.0	Clear-Content	Cmdlet	Microsoft.PowerShell.Management	Microsoft.PowerShell.Management	
7	7.0.1.0	Clear-History	Cmdlet	Microsoft.PowerShell.Core	ActiveDirectory	
8	7.0.0.0	Clear-Item	Cmdlet	Microsoft.PowerShell.Management	Microsoft.PowerShell.Management	
9	7.0.0.0	Clear-ItemProperty	Cmdlet	Microsoft.PowerShell.Management	Microsoft.PowerShell.Management	
10	7.0.0.0	Clear-RecycleBin	Cmdlet	Microsoft.PowerShell.Management	Microsoft.PowerShell.Management	
11	7.0.0.0	Clear-Variable	Cmdlet	Microsoft.PowerShell.Utility	Microsoft.PowerShell.Utility	

Abbildung 2-1: PowerShell-7-Cmdlets in einer Tabellenkalkulation

Wenn Sie genau hinschauen, werden Sie entdecken, dass offenkundig einige Spalten ausgeblendet wurden (die Tabellenspalten springen von G nach M etc.). Vermutlich hat unser gedachter Freund die Tabelle übersichtlich gestalten wollen und aus diesem Grund einige vermeintlich weniger relevante Informationen ausgeblendet.

Im Wesentlichen geschieht in der PowerShell exakt das Gleiche. Wollen Sie wirklich alle Eigenschaften sehen, übergeben Sie mithilfe des Pipeline-Operators `|` das Ergebnis des ersten Befehls `Get-Command -Name 'Get-Process'` an einen zweiten Befehl, hier `Select-Object`, der ihnen dann auf Wunsch alle oder eine ganz bestimmte Auswahl an Eigenschaften ausgibt.

```
Get-Command -Name 'Get-Process' | Select-Object -Property *

Get-Command -Name 'Get-Process' |

  Select-Object -Property CommandType, Name, Version,

    Source, DLL
```

Betrachten Sie einige weitere Beispiele – Sie werden das immer gleiche Muster erkennen, mit dem Sie die Ausgabe eines Befehls an Ihre Bedürfnisse anpassen können.

```
Get-Uptime

Get-Uptime | Select-Object -property *

Get-Uptime | Select-Object -property TotalHours


Get-Process -Name 'pwsh'
```

```
Get-Process -Name 'pwsh' | Select-Object -Property *

Get-Process -Name 'pwsh' |
  Select-Object -Property Processname, ID, Path
```

PowerShell-Cmdlets sind für sich allein oftmals keine sonderlich mächtigen Werkzeuge. Ihre Wirkung entwickeln sie in aller Regel in der Kombination verschiedener Befehle.

```
Get-Process | Sort-Object -Property Handles -Descending |
  Select-Object -First 10

Get-Process -name 'pwsh' | Stop-Process -confirm

Get-ChildItem -Path $home -File -Recurse |
  Group-Object -Property Extension |
    Sort-Object -Property Count -Descending

Get-ChildItem -Path $home -File -Recurse |
  Measure-Object -Property Length -AllStats

Find-Module -Tag 'PSEdition_Core' | Measure-Object |
  Select-Object -ExpandProperty Count
```

Im Listing oben suchen wir zunächst nach aktiven Prozessen, sortieren diese absteigend, sodass die ressourcenhungrigsten Prozesse am Anfang unsere Liste stehen, und lassen uns schließlich die ersten zehn Prozesse ausgeben.

Anschließend suchen wir mit dem Befehl `Get-ChildItem` nach Dateien (ähnlich `dir` oder `ls`) im Heimatverzeichnis des Anwenders und in allen darunterliegenden Ordnern. Die gefundenen Dateien werden nach Dateityp (Extension) gruppiert und in der Reihenfolge ihrer Häufigkeit ausgegeben. Da die PowerShell von klein nach groß sortiert, drehen wir die Reihenfolge abschließend mit dem Parameter `-descending` um. Im zweiten Beispiel lassen wir die PowerShell den benötigten Speicherplatz für diese Dateien berechnen. Grundlage hierfür ist die Eigenschaft `Length`, die etwas unglücklich benannt ist. Sie gibt die Dateigröße in Byte an.

Schließlich suchen wir mithilfe von `Find-Module -tag PSEdition_ Core` in der PowerShell Gallery nach Modulen, die zur PowerShell ab Version 6 kompatibel sind. Die gefundenen Module werden gezählt, die Anzahl wird als reiner Zahlenwert ausgegeben. Sie erfahren mehr darüber im Abschnitt »Die PowerShell Gallery« auf Seite 94.

Haben Sie es gemerkt? PowerShell-Befehle sind leicht lesbar. Die oftmals selbsterklärende Benennung und das wiederkehrende Prinzip der Übergabe von strukturierten Daten an einen nachfolgenden Befehl macht die PowerShell gleichzeitig leicht benutzbar und sehr mächtig. Ein Eckpfeiler der PowerShell-Philosophie ist die Übergabe von strukturierten und typisierten Daten.

Sie haben bereits `Get-ChildItem` kennengelernt, um Dateien und Ordner zu finden. Lassen Sie uns eine einzelne Datei betrachten, das geeignetere Cmdlet dazu ist `Get-Item` (obgleich Sie auch mit `Get-ChildItem` zum Ziel kämen), siehe Abbildung 2-2.

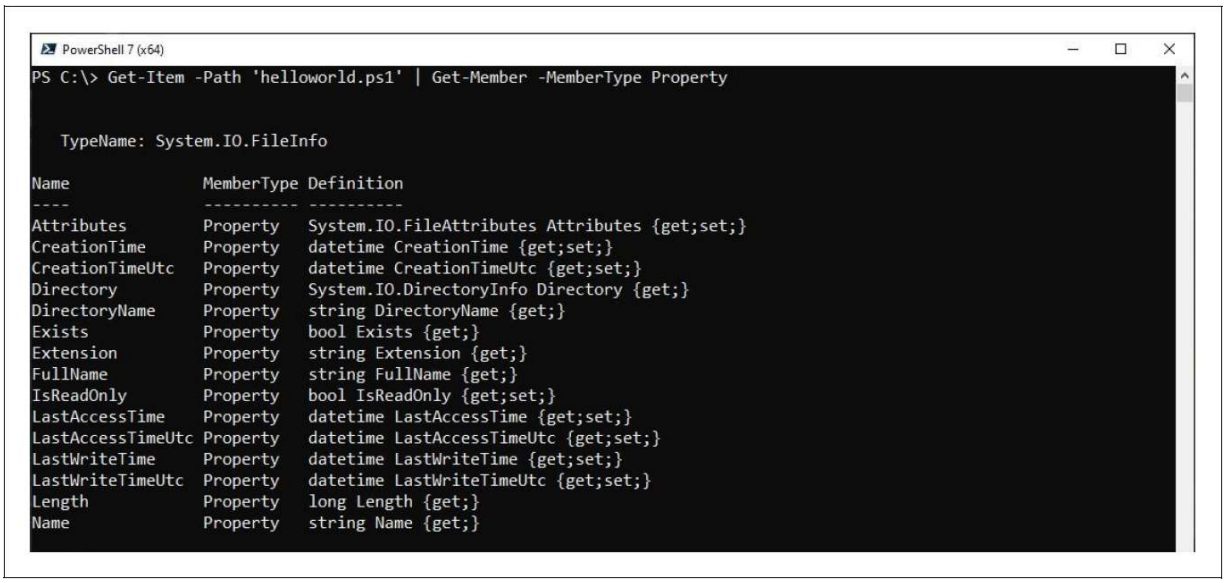


Abbildung 2-2: Die Eigenschaften einer Datei untersuchen

`Get-Member` gibt Aufschluss über den Datentyp (`System.IO.FileInfo`), den das Cmdlet `Get-Item` erzeugt. Anstatt Informationen über die konkrete Datei ausgeben zu lassen, übermitteln wir über die Pipeline das Ergebnis an `Get-Member` und fragen außerdem die korrespondierenden Eigenschaften (Properties) der Klasse ab. Über die Pipeline werden die Daten gemäß dieser definierten Struktur übergeben.

Wenn Sie sich für den Zeitpunkt interessieren, an dem die Beispieldatei erstellt wurde, müssen Sie diese Information nicht aus einer willkürlichen Ausgabe herausfiltern, Sie fordern einfach diese und gegebenenfalls weitere interessante Eigenschaften an. Die Standardausgabe ist im Prinzip nur ein Vorschlag der PowerShell-Entwickler.

```
Get-Item -Path 'helloworld.ps1' |

  Select-Object -Property CreationTime, Fullname


Get-ChildItem -Path $home -file -Recurse |
```