

PowerShell 7 und Windows PowerShell 5

Das Praxisbuch

» Hier geht's
direkt
zum Buch

DIE LESEPROBE

5

Objektorientiertes Pipelining

Ihre Mächtigkeit entfaltet die PowerShell erst durch das objektorientierte Pipelining, also durch die Weitergabe von strukturierten Daten von einem Commandlet zum anderen.



HINWEIS: Dieses Kapitel setzt ein Grundverständnis des Konzepts der Objektorientierung voraus. Wenn Sie diese Grundkenntnisse nicht besitzen, lesen Sie bitte zuvor im Anhang den Crashkurs „Objektorientierung“ sowie den Crashkurs „.NET Framework“ oder vertiefende Literatur.

■ 5.1 Befehlsübersicht

Die folgende Tabelle zeigt eine Übersicht der wichtigsten Commandlets, die Basisoperationen auf Pipelines ausführen. Diese Commandlets werden in den folgenden Kapiteln genau besprochen.

Tabelle 5.1 Übersicht über die wichtigsten Pipelining-Commandlets

Commandlet (mit Aliasen)	Bedeutung
Where-Object (where, ?)	Filtern mit Bedingungen
Select-Object (select)	Abschneiden der Ergebnismenge vorne/hinten bzw. Reduktion der Attribute der Objekte. Auch: Eliminieren von Duplikaten
Sort-Object (sort)	Sortieren der Objekte
Group-Object (group)	Gruppieren der Objekte
Foreach-Object { \$_... } (%)	Schleife über alle Objekte. Der Befehlsblock { ... } wird für jedes Objekt in der Pipeline einmal ausgeführt.
Get-Member (gm)	Ausgabe der Metadaten (Reflection)
Measure-Object (measure)	Berechnung: -min -max -sum -average
Compare-Object (compare, diff)	Vergleichen von zwei Objektmengen

■ 5.2 Pipeline-Operator

Für eine Pipeline wird – wie auch in Unix-Shells üblich und in der normalen Windows-Konsole möglich – der vertikale Strich „|“ (genannt „Pipe“ oder „Pipeline Operator“) verwendet.

```
Get-Process | Format-List
```

bedeutet, dass das Ergebnis des `Get-Process`-Commandlets an `Format-List` weitergegeben werden soll. Die Standardausgabeform von `Get-Process` ist eine Tabelle. Durch `Format-List` werden die einzelnen Attribute der aufzulistenden Prozesse untereinander statt in Spalten ausgegeben.

Die Pipeline kann beliebig lang sein, d.h., die Anzahl der Commandlets in einer einzigen Pipeline ist nicht begrenzt. Man muss aber jedes Mal den Pipeline-Operator nutzen, um die Commandlets zu trennen.

Ein Beispiel für eine komplexere Pipeline lautet:

```
Get-ChildItem w:\daten -r -filter *.doc  
| Where-Object { $_.Length -gt 40000 }  
| Select-Object Name, Length  
| Sort-Object Length  
| Format-List
```

`Get-ChildItem` ermittelt alle Microsoft-Word-Dateien im Ordner `w:\daten` und in seinen Unterordnern. Durch das zweite Commandlet (`Where-Object`) wird die Ergebnismenge auf diejenigen Objekte beschränkt, bei denen das Attribut `Length` größer ist als 40000. `$_` ist dabei der Zugriff auf das aktuelle Objekt in der Pipeline. Der Ausdruck `$.Length -gt 40000` ruft aus dem aktuellen Objekt die Eigenschaft `Length` ab und vergleicht, ob diese größer (`-gt`) als 40000 ist. `Select-Object` beschneidet alle Attribute aus `Name` und `Length`. Durch das vierte Commandlet in der Pipeline wird die Ausgabe nach dem Attribut `Length` sortiert. Das letzte Commandlet schließlich erzwingt eine Listendarstellung.

Nicht alle Aneinanderreihungen von Commandlets ergeben einen Sinn. Einige Aneinanderreihungen sind auch gar nicht erlaubt. Die Reihenfolge der einzelnen Befehle in der Pipeline ist nicht beliebig. Keineswegs kann man im obigen Befehl die Sortierung hinter die Formatierung setzen, weil nach dem Formatieren zwar noch ein Objekt existiert, dieses aber einen Textstrom repräsentiert. `Where-Object` und `Sort-Object` könnte man vertauschen; aus Gründen des Ressourcenverbrauchs sollte man aber erst einschränken und dann die verringerte Liste sortieren. Ein Commandlet kann aus vorgenannten Gründen erwarten, dass es bestimmte Arten von Eingabeobjekten gibt. Am besten sind aber Commandlets, die jede Art von Eingabeobjekt verarbeiten können.

Eine automatische Optimierung der Befehlsfolge wie in der Datenbankabfrage SQL gibt es bei PowerShell nicht.

Seit PowerShell-Version 3.0 hat Microsoft für den Zugriff auf das aktuelle Objekt der Pipeline zusätzlich zum Ausdruck `$_` den Ausdruck `$PSItem` eingeführt. `$_` und `$PSItem` sind synonym. Microsoft hat `$PSItem` eingeführt, weil einige Benutzer das Feedback gaben, dass `$_` zu (Zitat) „magisch“ sei.



ACHTUNG: Die PowerShell erlaubt beliebig lange Pipelines und es gibt auch Menschen, die sich einen Spaß daraus machen, möglichst viel durch eine einzige Befehlsfolge mit sehr vielen Pipes auszudrücken. Solche umfangreichen Befehlsfolgen sind aber meist für andere Menschen extrem schlecht lesbar. Bitte befolgen Sie daher den folgenden Ratschlag: Schreiben Sie nicht alles in eine einzige Befehlsfolge, nur weil es geht. Teilen Sie besser die Befehlsfolgen nach jeweils drei bis vier Pipe-Symbolen durch den Einsatz von Variablen auf (wird in diesem Kapitel auch beschrieben!) und lassen Sie diese geteilten Befehlsfolgen dann besser als PowerShell-Skripte ablaufen (siehe das Kapitel „PowerShell-Skripte“).

■ 5.3 .NET-Objekte in der Pipeline

Objektorientierung ist die herausragende Eigenschaft der PowerShell: Commandlets können durch Pipelines mit anderen Commandlets verbunden werden. Anders als Pipelines in Unix-Shells tauschen die Commandlets der PowerShell keine Zeichenketten, sondern typisierte .NET-Objekte aus. Das objektorientierte Pipelining ist im Gegensatz zum in den Unix-Shells und in der normalen Windows-Shell (*cmd.exe*) verwendeten zeichenkettenbasierten Pipelining nicht abhängig von der Position der Informationen in der Pipeline.

Ein Commandlet kann auf alle Attribute und Methoden der .NET-Objekte, die das vorhergehende Commandlet in die Pipeline gelegt hat, zugreifen. Die Mitglieder der Objekte können entweder durch Parameter der Commandlets (z. B. in `Sort-Object Length`) oder durch den expliziten Verweis auf das aktuelle Pipeline-Objekt (`$_`) in einer Schleife oder Bedingung (z. B. `Where-Object { $_.Length -gt 40000 }`) genutzt werden.

In einer Pipeline wie

```
Get-Process | Where-Object { $_.name -eq "iexplore" } | Format-Table ProcessName, WorkingSet64
```

ist das dritte Commandlet daher nicht auf eine bestimmte Anordnung und Formatierung der Ausgabe von vorherigen Commandlets angewiesen, sondern es greift über den sogenannten Reflection-Mechanismus (den eingebauten Komponentenerforschungsmechanismus des .NET Frameworks) direkt auf die Eigenschaften der Objekte in der Pipeline zu.



HINWEIS: Genau genommen bezeichnet Microsoft das Verfahren als „Extended Reflection“ bzw. „Extended Type System (ETS)“, weil die PowerShell in der Lage ist, Objekte um zusätzliche Eigenschaften anzureichern, die in der Klassendefinition gar nicht existieren.

Im obigen Beispiel legt `Get-Process` ein .NET-Objekt der Klasse `System.Diagnostics.Process` für jeden laufenden Prozess in die Pipeline. `System.Diagnostics.Process` ist eine Klasse aus der .NET-Klassenbibliothek. Commandlets können aber jedes beliebige .NET-Objekt in die Pipeline legen, also auch einfache Zahlen oder Zeichenketten, da es in .NET

keine Unterscheidung zwischen elementaren Datentypen und Klassen gibt. Eine Zeichenkette in die Pipeline zu legen, wird aber in der PowerShell die Ausnahme bleiben, denn der typisierte Zugriff auf Objekte ist wesentlich robuster gegenüber möglichen Änderungen als die Zeichenkettenauswertung mit regulären Ausdrücken.

Deutlicher wird der objektorientierte Ansatz, wenn man als Attribut keine Zeichenkette heranzieht, sondern eine Zahl. `WorkingSet64` ist ein 64 Bit langer Zahlenwert, der den aktuellen Speicherverbrauch eines Prozesses repräsentiert. Der folgende Befehl liefert alle Prozesse, die aktuell mehr als 20 Megabyte verbrauchen:

```
Get-Process | Where-Object { $_.WorkingSet64 -gt 20*1024*1024 }
```

Anstelle von `20*1024*1024` hätte man auch das Kürzel „20MB“ einsetzen können. Außerdem kann man `Where-Object` mit einem Fragezeichen abkürzen. Die kurze Variante des Befehls wäre dann also:

```
ps | ? { $_.ws -gt 20MB }
```

Wenn nur ein einziges Commandlet angegeben ist, dann wird das Ergebnis auf dem Bildschirm ausgegeben. Auch wenn mehrere Commandlets in einer Pipeline zusammengeschaltet sind, wird das Ergebnis des letzten Commandlets auf dem Bildschirm ausgegeben. Wenn das letzte Commandlet keine Daten in die Pipeline wirft, erfolgt keine Ausgabe.

5.4 Pipeline Processor

Für die Übergabe der .NET-Objekte zwischen den Commandlets sorgt der *PowerShell Pipeline Processor* (siehe folgende Grafik). Die Commandlets selbst müssen sich weder um die Objektweitergabe noch um die Parameterauswertung kümmern.

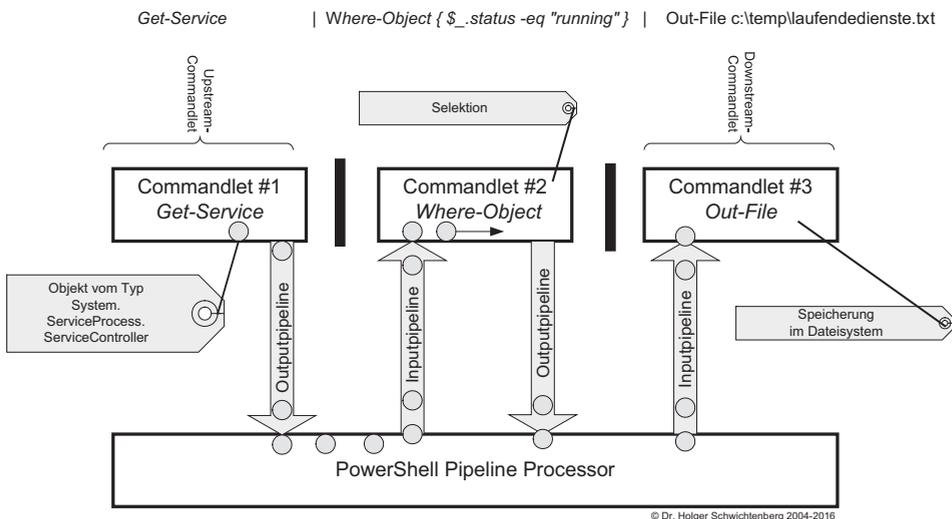


Bild 5.1 Der Pipeline Processor befördert die Objekte vom Upstream-Commandlet zum Downstream-Commandlet. Die Verarbeitung ist in der Regel asynchron.

Wie das obige Bild schon zeigt, beginnt ein nachfolgendes Commandlet mit seiner Arbeit, sobald es ein erstes Objekt aus der Pipeline erhält. Das Objekt durchläuft die komplette Pipeline. Erst dann wird das nächste Objekt vom ersten Commandlet abgeholt. Man nennt dies „Streaming-Verarbeitung“. Streaming-Verarbeitung ist schneller als die klassische sequentielle Verarbeitung, weil die folgenden Commandlets in der Pipeline nicht auf vorhergehende warten müssen.



HINWEIS: Intern arbeitet die einem Thread, d. h. es findet keine parallele Verarbeitung mehrerer Befehle statt. Erst seit PowerShell 7.0 gibt es mit dem Parameter `-parallel` bei `Foreach-Command` eine einfache Möglichkeit, jedes Objekt in einem eigenen Thread zu verarbeiten.

Aber nicht alle Commandlets beherrschen die asynchrone Streaming-Verarbeitung. Commandlets, die alle Objekte naturgemäß erst mal kennen müssen, bevor sie überhaupt ihren Zweck erfüllen können (z. B. `Sort-Object` zum Sortieren und `Group-Object` zum Gruppieren), blockieren die asynchrone Verarbeitung.



HINWEIS: Es gibt auch einige Commandlets, die zwar asynchron arbeiten könnten, aber leider nicht so programmiert wurden, um dies zu unterstützen.

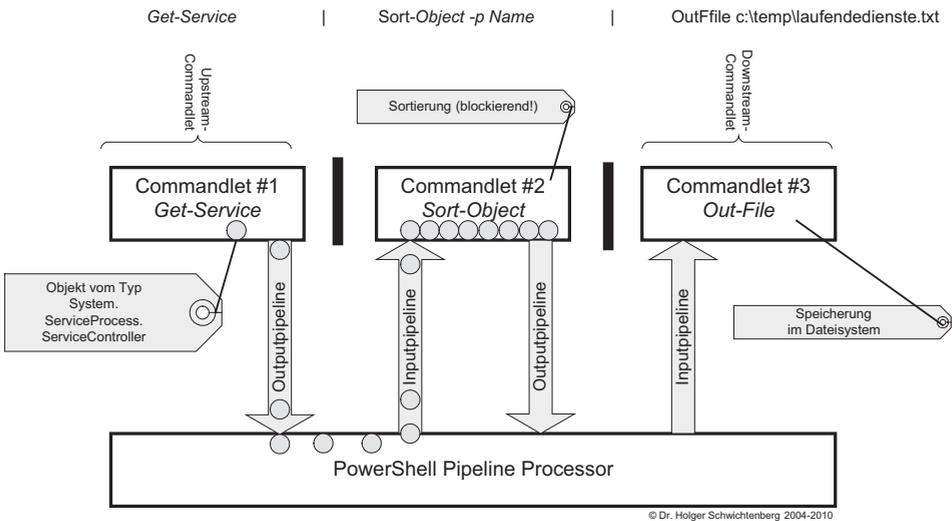


Bild 5.2 `Sort-Object` blockiert die direkte Weitergabe. Erst wenn alle Objekte angekommen sind, kann das Commandlet sortieren.

Auch bei Commandlets, die Streaming-Verarbeitung unterstützen kann der PowerShell-Nutzer mit dem allgemeinen Parameter `-OutBuffer` (abgekürzt `-ob`), das jedes Commandlet anbietet, dafür sorgen, dass eine bestimmte Anzahl von Objekten angesammelt wird bevor eine Weitergabe an das nachfolgende Commandlet erfolgt.

Im Standard beginnt die Ausgabe der Ordner- und Dateinamen sofort:

```
dir c:\ -Recurse | ft name
```

In diesem Fall passiert lange nichts, bevor die Ausgabe beginnt:

```
dir c:\ -Recurse -OutBuffer:100000 | ft name
```

■ 5.5 Pipelining von Parametern

Die Pipeline kann jegliche Art von Information befördern, auch einzelne elementare Daten. Einige Commandlets unterstützen es, dass auch die Parameter aus der Pipeline ausgelesen werden. Der folgende Pipeline-Befehl führt zu einer Auflistung aller Windows-Systemdienste, die mit dem Buchstaben „I“ beginnen.

```
"i*" | Get-Service
```

Die folgende Abbildung zeigt einige Parameter des Commandlets Get-Service. Diese Liste erhält man durch den Befehl `Get-Help Get-Service -Parameter *`.

```
-Include <string[]>
  Retrieves only the specified services. The value of this parameter qualifies the Name parameter. Enter a name element or pattern, such as "s*". Wildcards are permitted.

  Required?                false
  Position?                named
  Default value            named
  Accept pipeline input?   false
  Accept wildcard characters? false

-InputObject <ServiceController[]>
  Specifies ServiceController objects representing the services to be retrieved. Enter a variable that contains the objects, or type a command or expression that gets the objects. You can also pipe a service object to Get-Service.

  Required?                false
  Position?                named
  Default value            named
  Accept pipeline input?   true (ByValue)
  Accept wildcard characters? false

-Name <string[]>
  Specifies the service names of services to be retrieved. Wildcards are permitted. By default, Get-Service gets all of the services on the computer.

  Required?                false
  Position?                1
  Default value            1
  Accept pipeline input?   true (ByValue, ByPropertyName)
  Accept wildcard characters? true

-RequiredServices [(SwitchParameter)]
  Gets only the services that this service requires.

  This parameter gets the value of the ServicesDependedOn property of the service. By default, Get-Service gets all services.

  Required?                false
  Position?                named
  Default value            False
  Accept pipeline input?   false
  Accept wildcard characters? false
```

Bild 5.3 Hilfe zu den Parametern des Commandlets Get-Service

Interessant sind die mit Pfeil markierten Stellen. Nach „Accept pipeline Input“ kann man jeweils nachlesen, ob der Parameter des Commandlets aus den vorhergehenden Objekten in der Pipeline „befüttert“ werden kann.

Bei „-Name“ steht ByValue und ByPropertyName. Dies bedeutet, dass der Name sowohl das ganze Objekt in der Pipeline sein darf als auch Teil eines Objekts.

Im Fall von

```
"BITS" | Get-Service
```

ist der Pipeline-Inhalt eine Zeichenkette (ein Objekt vom Typ String), die als Ganzes auf Name abgebildet werden kann.

Es funktioniert aber auch folgender Befehl, der alle Dienste ermittelt, deren Name genauso lautet wie der Name eines laufenden Prozesses:

```
Get-Process | Get-Service -ea silentlycontinue | ft name
```

Dies funktioniert über die zweite Option (ByPropertyName), denn Get-Process liefert Objekte des Typs Process, die ein Attribut namens Name haben. Der Parameter Name von Get-Service wird auf dieses Name-Attribut abgebildet.

Beim Parameter -InputObject ist hingegen nur „ByValue“ angegeben. Hier erwartet Get-Service gerne Instanzen der Klasse ServiceController. Es gibt aber keine Objekte, die ein Attribut namens InputObject haben, in dem dann ServiceController-Objekte stecken.

Zahlreiche Commandlets besitzen einen Parameter -InputObject, insbesondere die allgemeinen Verarbeitungs-Commandlets wie Where-Object, Select-Object und Measure-Object, die Sie im nächsten Kapitel kennenlernen werden. Der Name -InputObject ist eine Konvention.

```
PS P:\> Get-Help Where-Object -Parameter *
-FilterScript <scriptblock>
  Specifies the script block that is used to filter the objects. Enclose the
  script block in braces < > .
  Required?                true
  Position?                1
  Default value
  Accept pipeline input?   false
  Accept wildcard characters? false

-InputObject <psobject>
  Specifies the objects to be filtered. You can also pipe the objects to Where-Object.
  Required?                false
  Position?                named
  Default value
  Accept pipeline input?   true <ByValue>
  Accept wildcard characters? false
```

Bild 5.4 Parameter des Commandlets Where-Object

Leider geht es nicht bei allen Commandlets so einfach mit der Parameterübergabe. Man nehme zum Beispiel das Commandlet Test-Connection, das prüft, ob ein Computer per Ping erreichbar ist.

Der normale Aufruf mit Parameter ist:

```
Test-Connection -computername Server123
```

oder ohne benannten Parameter

```
Test-Connection Server123
```

Nun könnte man auf die Idee kommen, hier den Computernamen genau so zu übergeben, wie den Namen bei Get-Service. Allerdings liefert "Server123" | Test-Connection den Fehler: *"The input object cannot be bound to any parameters for the command either because the command does not take pipeline input or the input and its properties do not match any of the parameters that take pipeline input."*

Warum das nicht geht, kann man in der Hilfe zum Parameter ComputerName des Commandlets Test-Connection erkennen. Dort steht, dass ComputerName nur als „ByPropertyName“ akzeptiert wird und nicht wie beim Parameter Name beim Commandlet Get-Service auch „ByValue“. Das bedeutet also, dass man erst ein Objekt mit der Eigenschaft ComputerName konstruieren und dann übergeben muss:

```
New-Object psobject -Property @{Computername="Server123"} | Test-Connection
```

Das funktioniert zwar, ist aber hässlich und umständlich. Warum Test-Connection und einige andere Commandlets die Eingaben nicht „ByValue“ unterstützen, wusste übrigens das PowerShell-Entwicklungsteam auf Nachfrage auch nicht zu beantworten. Die Schuld liegt hier vermutlich bei dem einzelnen Entwickler bei Microsoft, der die Commandlets implementiert hat.

```
-ComputerName <string[]>
Required?                true
Position?                0
Accept pipeline input?   true (ByPropertyName)
Parameter set name       (All)
Aliases                  CN, IPAddress, __SERVER, Server, Destination
Dynamic?                 false
```

Bild 5.5 Hilfe zum Parameter ComputerName des Commandlets Test-Connection

■ 5.6 Pipelining von klassischen Befehlen

Grundsätzlich dürfen auch klassische Kommandozeilenanwendungen in der PowerShell verwendet werden. Wenn man einen Befehl wie netstat.exe oder ping.exe ausführt, dann legen diese eine Menge von Zeichenketten in die Pipeline: Jede Ausgabezeile ist eine Zeichenkette.

Diese Zeichenketten kann man sehr gut mit dem Commandlet Select-String auswerten. Select-String lässt nur diejenigen Zeilen die Pipeline passieren, die auf den angegebenen regulären Ausdruck zutreffen.



TIPP: Die Syntax der regulären Ausdrücke in .NET wird im Kapitel „PowerShell-Skriptsprache“ noch etwas näher beschrieben werden.

In dem folgenden Beispiel werden nur diejenigen Zeilen der Ausgabe von `netstat.exe` gefiltert, die einen Doppelpunkt gefolgt von den Ziffern 59 und zwei weiteren Ziffern enthalten. Die Hervorhebung der Treffer durch Negativschrift gibt es erst seit PowerShell 7.0.

```

pwsh
PS X:\> netstat | select-string ":59\d\d" -case
TCP    127.0.0.1:5905           E60:49675           ESTABLISHED
TCP    127.0.0.1:5905           E60:49677           ESTABLISHED
TCP    127.0.0.1:5905           E60:49678           ESTABLISHED
TCP    127.0.0.1:5905           E60:57897           ESTABLISHED
TCP    127.0.0.1:5905           E60:57898           ESTABLISHED
TCP    127.0.0.1:5905           E60:57899           ESTABLISHED
TCP    127.0.0.1:5905           E60:57920           ESTABLISHED
TCP    127.0.0.1:5905           E60:57921           ESTABLISHED
TCP    127.0.0.1:49675          E60:5905            ESTABLISHED
TCP    127.0.0.1:49677          E60:5905            ESTABLISHED
TCP    127.0.0.1:49678          E60:5905            ESTABLISHED
TCP    127.0.0.1:57897          E60:5905            ESTABLISHED
TCP    127.0.0.1:57898          E60:5905            ESTABLISHED
TCP    127.0.0.1:57899          E60:5905            ESTABLISHED
TCP    127.0.0.1:57920          E60:5905            ESTABLISHED
TCP    127.0.0.1:57921          E60:5905            ESTABLISHED
PS X:\>

```

Bild 5.6
Einsatz von Select-String zur Filterung von Ausgaben klassischer Kommandozeilenwerkzeuge

Ein weiteres Beispiel ist das Filtern der Ausgaben von `ipconfig.exe`. Der nachfolgende Befehl liefert nur die Zeilen zum Thema IPv4:

```
ipconfig.exe /all | select-string IPv4
```

```

pwsh
PS X:\> ipconfig.exe /all | select-string IPv4
IPv4 Address. . . . . : 192.168.1.60(Preferrred)
IPv4 Address. . . . . : 172.17.24.113(Preferrred)
PS X:\>

```

Bild 5.7
Abbildung: Ausführung des obigen Befehls

Es gibt aber leider klassische Kommandozeilenbefehle, die inhaltliche Informationen über Farben statt über Texte transportieren. Ein schlechtes Beispiel ist hier:

```
git branch -a
```

Der Befehl `git branch -a` liefert eine Liste aller Git-Banches in einem lokalen Git-Repository als farblich verschieden markierte Textzeilen.

```

T:\CC2 [master =>] git branch -a
* master
remotes/GITHUB/Feature1
remotes/GITHUB/master
remotes/GITHUB/F2
remotes/GITHUB/Feature1
remotes/GITHUB/Feature2
remotes/GITHUB/Feature3
remotes/GITHUB/HEAD -> GITHUB/master
remotes/GITHUB/master

```

Eine schwarze Ausgabe (erste beide Zeilen) bedeutet, dass es für den Remote-Branch auch einen lokalen Branch gibt. Eine rote Ausgabe (Zeile 3 bis 8, hier im Buch aufgrund des Schwarz-Weiß-Drucks leider nicht zu sehen) bedeutet dabei, dass ein Remote-Branch noch kein lokales Äquivalent besitzt.

Man kann diesen Befehl zwar in der PowerShell ausführen und sieht dort auch die Farben. Aber eine Weiterverarbeitung per Pipeline mit dem Ziel „Lege einen lokalen Branch an für alle Branches, die lokal noch nicht existieren“, ist nicht möglich.

Man kann lediglich `git branch` für alle ausführen. Hierbei muss man nicht nur filtern, sondern auch mit `Trim()` die Leerzeichen zu Beginn eliminieren:

```
git branch -a | ? { $_ -like "*remotes*" -and $_ -notlike "*HEAD*" } | % { git branch --track ${remote#origin/} $_.Trim() }
```

oder

```
git branch -a | sls -pattern "remotes" | sls -pattern "HEAD" -NotMatch | % { git branch --track ${remote#origin/} $_.Line.Trim() }
```

Man bekommt aber immer eine Fehlermeldung für die schon existierenden lokalen Branches.

```
T:\CC2 [master =>] git branch -a | ? { $_ -like "*remotes*" -and $_ -notlike "*HEAD*" } | % { git branch --track ${remote#origin/} $_.Trim() }
fatal: A branch named 'remotes/GITHUB/Feature1' already exists.
fatal: A branch named 'remotes/GITHUB/master' already exists.
Branch 'remotes/GITHUB/F2' set up to track local branch 'master'.
fatal: A branch named 'remotes/GITHUB/Feature1' already exists.
Branch 'remotes/GITHUB/Feature2' set up to track local branch 'master'.
Branch 'remotes/GITHUB/Feature3' set up to track local branch 'master'.
fatal: A branch named 'remotes/GITHUB/master' already exists.
```

■ 5.7 Zeilenumbrüche in Pipelines

Wenn sich ein Pipeline-Befehl über mehrere Zeilen erstrecken soll, kann man dies auf mehrere Weisen bewerkstelligen:

- Man beendet die Zeile mit einem Pipe-Symbol `[]` und drückt EINGABE. PowerShell-Standardkonsole und PowerShell-ISE-Konsole erkennen, dass der Befehl noch nicht abgeschlossen ist, und erwarten weitere Eingaben. Die Standardkonsole zeigt dies auch mit `>>>` an.
- Man kann am Ende einer Zeile mit einem Gravis `[`]`, ASCII-Code 96, bewirken, dass die nächste Zeile mit zum Befehl hinzugerechnet wird (Zeilenumbruch in einem Befehl). Das funktioniert in allen PowerShell-Hosts und auch in PowerShell-Skripten.

```
PS T:\> Get-Process p* | Sort-Object WorkingSet |
>> Format-Table id,name,WorkingSet

    Id Name                WorkingSet
    -- --                -
10828 powershell          92942336
15340 powershell_ise     220946432
 1804 powershell          83664896
  404 powershell          76177408

PS T:\> _
```

Bild 5.8
Zeilenumbruch nach
Pipeline-Symbol

■ 5.8 Schleifen

Ein wichtiges Commandlet ist

```
Foreach-Object { $_... }
```

Alias:

```
% { $_... }
```

Foreach-Object führt eine Schleife (Iteration) über alle Objekte in der Pipeline aus. Der Befehlsblock { ... } wird für jedes Objekt in der Pipeline einmal ausgeführt. Das jeweils aktuelle Objekt, das an der Reihe ist, erhält man über die eingebaute Variable `$_`. `$_` ist die Abkürzung für `$PSItem`. Beide Schreibweisen haben die gleiche Funktion.

5.8.1 Notwendigkeit für Foreach-Object

Der Einsatz von Foreach-Object ist in Pipelines nicht notwendig, wenn das nachfolgende Commandlet die Objekte des vorherigen Commandlets direkt verarbeiten kann.

Beispiele:

```
Get-ChildItem Bu* | Remove-Item
Get-Service BI* | Start-Service
Get-Process chrome | Stop-Process
```

Gleichwohl könnte man in diesen Fällen Foreach-Object einsetzen, was den Befehl aber verlängert:

```
Get-ChildItem Bu* | Foreach-Object { Remove-item $_.FullName }
Get-Service BI* | Foreach-Object { Start-Service $_ }
Get-Process chrome | Foreach-Object { Stop-Process $_ }
```

Es liegt an den Eigenarten des jeweiligen Commandlets, ob sie als Standardparameter das gesamte Objekt (`$_`) oder eine bestimmte Eigenschaft (`$_.Fullname`) erwarten.

In manchen Situationen ist der Einsatz von Foreach-Object aber auch nicht möglich, denn man will mit Sort-Object die ganze Menge sortieren und nicht jedes Objekt einzeln:

```
"----- richtig:"
Get-Service x* | Sort-Object name
"----- falsch:"
Get-Service x* | Foreach-Object { Sort-Object $_.Name }
```

Schließlich gibt es Fälle, in denen Foreach-Object zwingend eingesetzt werden muss. Dies gilt insbesondere, wenn das nachfolgende Commandlet die Objekte nicht verarbeiten kann. Zudem quittiert die PowerShell diesen Befehl

```
Get-Service BI* | Write-Host $_.DisplayName -ForegroundColor yellow
```

mit dem Laufzeitfehler „The input object cannot be bound to any parameters for the command either because the command does not take pipeline input or the input and its properties do not“.

Richtig ist:

```
Get-Service BI* | foreach-object { Write-Host $_.DisplayName -ForegroundColor Yellow }
```

Ebenso ist Foreach-Object notwendig, wenn mehrere Befehle (also ganzer Befehlsblock) ausgeführt werden sollen. Befehlsblöcke werden in den Kapiteln „PowerShell-Skripte“ und „PowerShell-Skriptspache“ erläutert.

```
Get-Service BI* | foreach-object {  
    if ($_.Status -eq "Stopped")  
    {  
        Write-Host "Beendet Dienst " $_.DisplayName -ForegroundColor Yellow  
        Start-Service $_  
    }  
    else  
    {  
        Write-Host "Starte Dienst " $_.DisplayName -ForegroundColor Yellow  
        Stop-Service $_  
    }  
}
```

5.8.2 Parallelisierung mit Multithreading

In PowerShell 1.0 bis 6.2 erfolgt die Ausführung im Hauptthread der PowerShell, d. h., die einzelnen Durchläufe erfolgen nacheinander. Seit PowerShell 7.0 kann man mit dem Parameter `-parallel` die Ausführung auf verschiedene Threads parallelisieren (via Multithreading), sodass bei längeren Operationen in Summe das Ergebnis schneller vorliegt.



ACHTUNG: Die Multithreading hat immer einigen Overhead. Die Parallelisierung lohnt sich nur bei länger dauernden Operationen. Bei kurzen Operationen ist der Zeitverlust durch die Erzeugung und Vernichtung der Threads höher als der Zeitgewinn durch die Parallelisierung.

Das folgende Beispiel zeigt zwei Varianten der Abfrage, ob die Software „Classic Shell“ auf drei verschiedenen Computern installiert ist. Bei der ersten Variante ohne `-parallel` wird die leider etwas langwierige Abfrage der WMI-Klasse `Win32_Product` auf den drei Computern nacheinander in dem gleichen Thread ausgeführt. Bei der zweiten Variante mit `-parallel` wird die Abfrage parallel in drei verschiedenen Threads gestartet! Die Parallelisierung ist erst möglich seit PowerShell 7.0.



TIPP: Die Nummer des Threads fragt man ab mit der .NET-Klasse `Thread: [System.Threading.Thread]::CurrentThread.ManagedThreadId`

Listing 5.1 [\PowerShell\1_Basiswissen\Pipelining\Schleifen.ps1]

```

Write-Host "# ForEach-Object ohne -parallel" -ForegroundColor Yellow
"E27","E29","E44" | ForEach-Object {
    "Abfrage bei Computer $_ in Thread $($([System.Threading.Thread]::CurrentThread.
ManagedThreadId)"
    $e = Get-CimInstance -Class Win32_
Product -Filter "Name='Classic Shell'" -computername $_
    if ($e -eq $null) { "Kein Ergebnis bei $_!"}
    else { $e }
}
Write-Host ""
Write-Host " # ForEach-Object mit -parallel" -ForegroundColor Yellow
"E27","E29","E44" | ForEach-Object -parallel {
    "Abfrage bei Computer $_ in Thread $($([System.Threading.Thread]::CurrentThread.
ManagedThreadId)"
    $e = Get-CimInstance -Class Win32_
Product -Filter "Name='Classic Shell'" -computername $_
    if ($e -eq $null) { "Kein Ergebnis bei $_!"}
    else { $e }
}
# ohne Read-
Host würde das Skript die später eingehenden Ergebnisse nicht mehr anzeigen!
read-host

```

```

# ForEach-Object ohne -parallel
Abfrage bei Computer E27 in Thread 19
Name          Caption          Vendor          Version          IdentifyingNumber          PSComputerName
----          -
Classic Shell Classic Shell    Ivosoft         4.1.0            {849C8587-03D6-4143-9AF9-DAE88FD... E27
Abfrage bei Computer E29 in Thread 19
Classic Shell Classic Shell    Ivosoft         4.1.0            {849C8587-03D6-4143-9AF9-DAE88FD... E29
Abfrage bei Computer E44 in Thread 19
Kein Ergebnis bei E44!

# ForEach-Object mit -parallel
Abfrage bei Computer E27 in Thread 88
Abfrage bei Computer E29 in Thread 94
Abfrage bei Computer E44 in Thread 96
Kein Ergebnis bei E44!
Classic Shell Classic Shell    Ivosoft         4.1.0            {849C8587-03D6-4143-9AF9-DAE88FD... E29
Classic Shell Classic Shell    Ivosoft         4.1.0            {849C8587-03D6-4143-9AF9-DAE88FD... E27

```

Bild 5.9 Parallelität bei Foreach-Object in PowerShell 7

Die Anzahl der Threads, die Foreach-Object nutzen soll, kann man mit dem Parameter `-ThrottleLimit` begrenzen:

```

1..20 | ForEach-Object -parallel {
    Write-Host "Objekt #$_ in Thread $($([System.Threading.Thread]::CurrentThread.
ManagedThreadId)"
    sleep -Seconds 2 } -ThrottleLimit 5

```

■ 5.9 Zugriff auf einzelne Objekte aus einer Menge

Es ist möglich, gezielt einzelne Objekte über ihre Position (Index) in der Pipeline anzusprechen. Die Positionsangabe ist in eckige Klammern zu setzen und die Zählung beginnt bei 0. Der Pipeline-Ausdruck ist in runde Klammern zu setzen.

Beispiele:

Der erste Prozess:

```
(Get-Process)[0]
```

Der dreizehnte Prozess:

```
(Get-Process)[12]
```

Alternativ kann man dies auch mit `Select-Object` unter Verwendung der Parameter `-First` und `-Skip` ausdrücken:

```
(Get-Process i* | Select-Object -first 1).name
(Get-Process i* | Select-Object -skip 12 -first 1).name
```



HINWEIS: Während `(Get-Date)[0]` in PowerShell vor Version 3.0 zu einem Fehler führt („Unable to index into an object of type System.DateTime.“), weil `Get-Date` keine Menge liefert, ist der Befehl seit PowerShell-Version 3.0 in Ordnung und liefert das gleiche Ergebnis wie `Get-Date`, da die PowerShell seit Version 3.0 ja aus Benutzersicht ein einzelnes Objekt und eine Menge von Objekten gleich behandelt. `(Get-Date)[1]` liefert dann natürlich kein Ergebnis, weil es kein zweites Objekt in der Pipeline gibt.

Die Positionsangaben kann man natürlich mit Bedingungen kombinieren. So liefert dieser Befehl den dreizehnten Prozess in der Liste der Prozesse, die mehr als 20 MB Hauptspeicher brauchen:

```
(Get-Process | where-object { $_.WorkingSet64 -gt 20mb } )[12]
```

```
PS C:\Windows\System32> (get-process)[0]
Handles NPM(K) PM(K) WS(K) UM(M) CPU(s) Id ProcessName
-----
20      2   1968   2664   17    0,03  2784 cmd

PS C:\Windows\System32> (get-process)[12]
Handles NPM(K) PM(K) WS(K) UM(M) CPU(s) Id ProcessName
-----
69      9   1484   4196   41    0,03  2100 dlpwdnt

PS C:\Windows\System32> (get-process | where-object { $_.WorkingSet64 -gt 20mb } )[12]
Handles NPM(K) PM(K) WS(K) UM(M) CPU(s) Id ProcessName
-----
685     29  53924  59544  291   34,39  4984 powershell

PS C:\Windows\System32>
```

Bild 5.10 Zugriff auf einzelne Prozessobjekte

■ 5.10 Zugriff auf einzelne Werte in einem Objekt

Manchmal möchte man nicht ein komplettes Objekt bzw. eine komplette Objektmenge verarbeiten, sondern nur eine einzelne Eigenschaft.

Oben wurde bereits gezeigt, wie man mit den Format-Commandlets wie `Format-Table` auf einzelne Eigenschaften zugreifen kann:

```
Get-Process | Format-Table ProcessName, WorkingSet64
```

Hat man nur ein einzelnes Objekt in Händen, geht das ebenfalls:

```
(Get-Process)[0] | Format-Table ProcessName, WorkingSet64
```

`Format-Table` liefert aber immer eine bestimmte Ausgabe, eben in Tabellenform mit Kopfzeile.

5.10.1 Punkt-Operator

Wenn man wirklich nur den Inhalt einer bestimmten Eigenschaft eines Objekts haben möchte, so verwendet man den in objektorientierten Sprachen üblichen Punkt-Operator, d. h., man trennt das Objekt und die abzurufende Eigenschaft durch einen Punkt (Punktnotation).

Beispiele:

```
(Get-Process)[0].ProcessName
```

Die Ausgabe ist eine einzelne Zeichenkette mit dem Namen des Prozesses.

```
(Get-Process)[0].WorkingSet64
```

Die Ausgabe ist eine einzelne Zahl mit der Speichernutzung des Prozesses.

Mit den Einzelwerten kann man weiterrechnen, z. B. errechnet man so die Speichernutzung in Megabyte:

```
(Get-Process)[0].WorkingSet64 / 1MB
```

```
PS C:\Windows\System32> <get-process>[0] | Format-Table ProcessName, WorkingSet64
-----
ProcessName                                     WorkingSet64
-----
cmd                                             2727936
PS C:\Windows\System32> <get-process>[0].ProcessName
cmd
PS C:\Windows\System32> <get-process>[0].WorkingSet64
2727936
PS C:\Windows\System32> <get-process>[0].WorkingSet64 / 1MB
2.6015625
PS C:\Windows\System32>
```

Bild 5.11 Ausgabe zu den obigen Beispielen

Weitere Anwendungsfälle seien am Beispiel `Get-Date` gezeigt. `Date`, `TimeOfDay`, `Year`, `Day`, `Month`, `Hour` und `Minute` sind einige der zahlreichen Eigenschaften der Klasse `DateTime`, die `Get-Date` liefert.

```

PowerShell
PS T:\> Get-Date
Donnerstag, 7. April 2022 17:57:44
PS T:\> (Get-Date).Date
Donnerstag, 7. April 2022 00:00:00
PS T:\> (Get-Date).TimeOfDay
Days           : 0
Hours          : 17
Minutes        : 58
Seconds        : 0
Milliseconds   : 396
Ticks          : 646803968929
TotalDays      : 0,748615704778935
TotalHours     : 17,9667769146944
TotalMinutes   : 1078,00661488167
TotalSeconds   : 64680,3968929
TotalMilliseconds : 64680396,8929
PS T:\> (Get-Date).Year
2022
PS T:\> (Get-Date).Month
4
PS T:\> (Get-Date).Day
7
PS T:\> (Get-Date).Hour
17
PS T:\> (Get-Date).Minute
58
PS T:\> |

```

Bild 5.12

Zugriff auf einzelne Werte aus dem aktuellen Datum/der aktuellen Zeit

5.10.2 Null-Werte

Zu beachten ist, dass PowerShell-Objekte, wie in objektorientierten Sprachen üblich, den Null-Wert (in PowerShell: `$null`) annehmen können mit der Interpretation, dass ein Objekt nicht vorhanden ist. Anders als in den meisten objektorientierten Sprachen führt die Anwendung des Punkt-Operators auf Null-Werte aber nicht zwangsläufig zu einem Laufzeitfehler. Die PowerShell ist sehr tolerant:

- Wenn man einen Null-Wert ausgibt, bekommt man keine Ausgabe.
- Wenn man in der Pipeline auf einen Null-Wert den Punkt-Operator anwendet, wird der Laufzeitfehler unterdrückt und man erhält keine Ausgabe.

Die PowerShell ist aber nicht in allen Fällen gegenüber der Anwendung des Punkt-Operators auf Variablen mit Wert `$null` tolerant (siehe folgende Abbildung).

```

pwsh
PS X:\> (Get-Process).Count
255
PS X:\> (Get-Process)[2000]
PS X:\> (Get-Process)[2000].Processname
PS X:\> (Get-Process)[2000].WorkingSet64
PS X:\> (Get-Process)[2000].WorkingSet64 / 1MB
0
PS X:\> $nichtinitialisierteVariable -eq $null
True
PS X:\> $nichtinitialisierteVariable.Length
0
PS X:\> $nichtinitialisierteVariable.ToUpper()
InvalidOperation: You cannot call a method on a null-valued expression.
PS X:\> $nichtinitialisierteVariable.Substring(10,5)
InvalidOperation: You cannot call a method on a null-valued expression.
PS X:\> _

```

Bild 5.13

Null-Werte in der PowerShell

5.10.3 Einzelne Werte aus allen Objekten einer Objektmenge

Wenn man einen einzelnen Wert aus allen Objekten aus einer Objektmenge ausgeben wollte, so konnte man das bis PowerShell 2.0 nur über ein nachgeschaltetes Foreach-Object lösen, wobei innerhalb von Foreach-Object mit `$_` auf das aktuelle Objekt der Pipeline zu verweisen war:

```
Get-Process | Foreach-Object { $_.Name }
```

Das geht seit PowerShell-Version 3.0 wesentlich prägnanter und eleganter:

```
(Get-Process).Name
```

Oder

```
(Get-Process).WorkingSet
```

Weiterhin muss man Foreach-Object anwenden für eine kombinierte Ausgabe:

```
Get-Process | Foreach-Object { $_.Name + ": " + $_.WorkingSet }
```

Mancher könnte denken, dass

```
(Get-Process).Name + ":" + (Get-Process).WorkingSet
```

auch als Schreibweise möglich wäre. Das liefert aber weder optisch noch inhaltlich ein korrektes Ergebnis, denn die Prozessliste wird zweimal abgerufen und könnte sich in der Zwischenzeit geändert haben!

■ 5.11 Methoden ausführen

Der folgende PowerShell-Pipeline-Befehl beendet alle Instanzen des Internet Explorers auf dem lokalen System, indem das Commandlet `Stop-Process` die Instanzen des betreffenden Prozesses von `Get-Process` empfängt.

```
Get-Process iexplore | Stop-Process
```

Die Objekt-Pipeline der PowerShell hat noch weitere Möglichkeiten: Gemäß dem objektorientierten Paradigma haben .NET-Objekte nicht nur Attribute, sondern auch Methoden. In einer Pipeline kann der Administrator daher auch die Methoden der Objekte aufrufen. Objekte des Typs `System.Diagnostics.Process` besitzen zum Beispiel eine Methode `Kill()`. Der Aufruf dieser Methode ist in der PowerShell gekapselt in der Methode `Stop-Process`.

Wer sich mit dem .NET Framework gut auskennt, könnte die `Kill()`-Methode auch direkt aufrufen. Dann ist aber eine explizite `ForEach`-Schleife notwendig. Die Commandlets iterieren automatisch über alle Objekte der Pipeline, die Methodenaufrufe aber nicht.

```
Get-Process iexplore | Foreach-Object { $_.Kill() }
```

Durch den Einsatz von Aliasen geht das auch kürzer:

```
ps | ? { $_.name -eq "iexplore" } | % { $_.Kill() }
```

Und seit PowerShell-Version 3.0 kann man auf das Foreach-Object bzw. % verzichten, also

```
(Get-Process iexplore).Kill()
```

oder

```
(ps iexplore).Kill()
```

schreiben.

Der Einsatz der Methode Kill() diene hier nur zur Demonstration, dass die Pipeline tatsächlich Objekte befördert. Eigentlich ist die gleiche Aufgabe besser mit dem eingebauten Commandlet Stop-Process zu lösen.



ACHTUNG: Vergessen Sie beim Aufruf von Methoden nicht die runden Klammern, auch wenn die Methoden keine Parameter besitzen. Ohne die Klammern erhalten Sie Informationen über die Methode, es erfolgt aber kein Aufruf.

```
PS C:\Users\hs.ITU> Get-Process notepad | foreach < $_.kill >

MemberType           : Method
OverloadDefinitions  : <System.Void Kill()>
TypeNameOfValue      : System.Management.Automation.PSMethod
Value                 : System.Void Kill()
Name                  : Kill
IsInstance            : True

MemberType           : Method
OverloadDefinitions  : <System.Void Kill()>
TypeNameOfValue      : System.Management.Automation.PSMethod
Value                 : System.Void Kill()
Name                  : Kill
IsInstance            : True
```

Runde
Kammern ()
fehlen

Bild 5.14

Folgen des verges-
senen Klammern-
paars

Dies funktioniert aber nur dann gut, wenn es auch Instanzen des Internet Explorers gibt. Wenn alle beendet sind, meldet Get-Process einen Fehler. Dies kann das gewünschte Verhalten sein. Mit einer etwas anderen Pipeline wird dieser Fehler jedoch unterbunden:

```
Get-Process | Where-Object { $_.Name -eq "iexplore" } |  
Stop-Process
```

Die zweite Pipeline unterscheidet sich von der ersten dadurch, dass das Filtern der Prozesse aus der Prozessliste nun nicht mehr von Get-Process erledigt wird, sondern durch ein eigenes Commandlet mit Namen Where-Object in der Pipeline selbst durchgeführt wird. Where-Object ist toleranter als Get-Process in Hinblick auf die Möglichkeit, dass es kein passendes Objekt gibt.

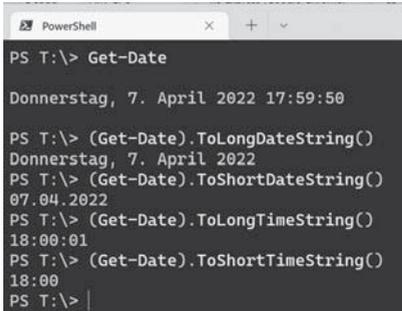
ps ist ein Alias für Get-Process, Kill für Stop-Process. Außerdem hat Get-Process eine eingebaute Filterfunktion. Um alle Instanzen des Internet Explorers zu beenden, kann man also statt

```
Get-Process | Where-Object { $_.Name -eq "iexplore" } |  
Stop-Process
```

auch schreiben:

```
ps -name "iexplore" | kill
```

Weitere Beispiele für die Aufrufe von Methoden seien am Beispiel von `Get-Date` gezeigt, das ja nur ein Objekt der Klasse `DateTime` liefert. Die Klasse `DateTime` bietet zahlreiche Methoden an, um Datum und Zeit auf bestimmte Weise darzustellen, z.B. `GetShortDateString()`, `GetLongDateString()`, `GetShortTimeString()` und `GetLongTimeString()`. Die Ausgaben zeigt die folgende Abbildung.



```
PowerShell
PS T:\> Get-Date
Donnerstag, 7. April 2022 17:59:50
PS T:\> (Get-Date).ToLongDateString()
Donnerstag, 7. April 2022
PS T:\> (Get-Date).ToShortDateString()
07.04.2022
PS T:\> (Get-Date).ToLongTimeString()
18:00:01
PS T:\> (Get-Date).ToShortTimeString()
18:00
PS T:\> |
```

Bild 5.15
Ausgaben der Methoden der Klasse `DateTime`

■ 5.12 Analyse des Pipeline-Inhalts

Drei der größten Fragestellungen bei der praktischen Arbeit mit der PowerShell sind:

- Wie viele Objekte sind in der Pipeline? (Das wurde schon zuvor in diesem Kapitel erörtert.)
- Welchen Typ haben die Objekte, die ein Commandlet in die Pipeline legt?
- Welche Attribute und Methoden haben diese Objekte?

Die Hilfe der Commandlets ist hier nicht immer hilfreich. Bei `Get-Service` kann man zwar lesen:

```
OUTPUTS
System.ServiceProcess.ServiceController
```

Bei anderen Commandlets aber heißt es nur wenig hilfreich:

```
OUTPUTS
Object
```

In keinem Fall sind in der PowerShell-Benutzerdokumentation (siehe <https://docs.microsoft.com/en-us/powershell/> und das Commandlet `Get-Help`) die Attribute und die Methoden der resultierenden Objekte genannt. Diese findet man nur in der .NET API-Dokumentation [<https://docs.microsoft.com/de-de/dotnet/api/>].

Im Folgenden werden zwei hilfreiche Commandlets sowie zwei Methoden und zwei Eigenschaften aus dem .NET Framework vorgestellt, die im Alltag helfen, zu erforschen, was man in der Pipeline hat:

- Count und Length
- ToString()
- GetType()
- Get-PipelineInfo
- Get-Member

5.12.1 Anzahl der Objekte in der Pipeline mit Count und Length

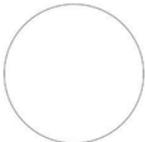
Viele Commandlets legen ganze Mengen von Objekten in die Pipeline (z. B. `Get-Process` eine Liste der Prozesse und `Get-Service` eine Liste der Dienste). Bei einer Objektmenge kann man, wie oben bereits gezeigt, mit `Where-Object` filtern. Das Ergebnis kann ein Objekt, kein Objekt oder eine Menge von Objekten sein.

Es kann aber auch sein, dass ein Commandlet, das normalerweise eine Menge von Objekten liefert, im konkreten Fall (z. B. bei Einsatz eines filternden Parameters) nur ein einzelnes Objekt liefert (z. B. `Get-Process idle`). In diesem Fall liefert die PowerShell dem Benutzer nicht eine Liste mit einem Objekt, sondern direkt das ausgepackte Objekt.

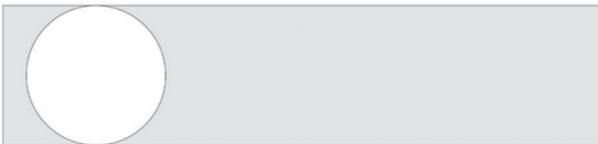
Einige Commandlets legen aber immer nur einzelne Objekte in die Pipeline. Ein Beispiel dafür ist `Get-Date`, das ein einziges Objekt des Typs `System.DateTime` in die Pipeline legt. Ruft man z. B. `Get-Date` ohne Weiteres auf, werden das aktuelle Datum und die aktuelle Zeit ausgegeben.

Zu differenzieren ist, ob die Pipeline ein Objekt direkt enthält oder eine Menge, die aus einem Objekt besteht (siehe Abbildung).

Pipeline mit einem Einzelobjekt



Pipeline mit einer Menge (ein `Object[]`), die nur ein Objekt enthält



Pipeline mit einer Menge (ein `Object[]`), die drei Objekte enthält



Bild 5.16
Einzelobjekt versus Menge

Bis Version 2.0 der PowerShell war es so, dass man eine Liste durch Zugriff auf `Count` oder `Length` nach der Anzahl der Elemente fragen konnte, nicht aber ein einzelnes Objekt.

Das war also erlaubt:

```
(Get-Process).Count
```

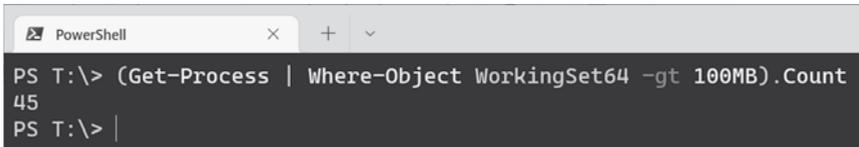
Das führte aber zu keinem Ergebnis:

```
(Get-Process idle).Count
(Get-Date).Count
```

Seit PowerShell-Version 3.0 ist dieser Unterschied (in den meisten Fällen) aufgehoben, man kann auch bei Einzelobjekten Count und Length abfragen, und die PowerShell liefert dann eben bei Einzelobjekten eine "1" zurück. Allerdings schlägt die Eingabehilfe der PowerShell-Konsole und der PowerShell ISE weiterhin weder Count noch Length als Möglichkeit vor!

Praxislösung: Wie viele Prozesse gibt es, die mehr als 100 MB Hauptspeicher (RAM) verbrauchen?

```
(Get-Process | Where-Object WorkingSet64 -gt 100MB).Count
```



```
PS T:\> (Get-Process | Where-Object WorkingSet64 -gt 100MB).Count
45
PS T:\> |
```

Bild 5.17 Aufruf von Count für eine Pipeline

Es gibt aber (mindestens) einen Fall, in denen Count auf einem Einzelobjekt nicht funktioniert. Dieser Fall, der nicht dokumentiert, mir aber in der Praxis ausgefallen ist, ist ein einzelnes PSCustomObject in der Pipeline. Es kann sicherlich weitere solcher nicht-dokumentierter Fälle geben. Wenn Sie Fälle kennen, schreiben Sie mir bitte!

Das folgende Beispiel zeigt auch, wie Sie diese Anomalie umgehen: Mit einem vorangestellten Komma macht man aus dem Einzelobjekt (System.Management.Automation.PSCustomObject) eine Menge mit einem Objekt (System.Object[]) mit einem System.Management.Automation.PSCustomObject).

Listing 5.2 [\PowerShell\1_Basiswissen\Pipelining\Pipelining.ps1]

```
$prozesse = Get-Process | select -First 1
Write-Host "Anzahl Prozesse: " $prozesse.Count # 1

$zahlen = 123
Write-Host "Anzahl Zahlen: " $zahlen.Count # 1

$firma1 = [PSCustomObject]@{
    Firma   = "www.IT-Visions.de"
    Ort     = "Essen"
}

Write-Host "Anzahl Firmen: " $firma1.Count # geht nicht! $null
$firma1.GetType().FullName # System.Management.Automation.PSCustomObject
if ($firma1.Count -eq $null) { Write-Warning "Count ist null!" }
```

```
# Workaround für Anomalie: Das vorangestellte Komma macht aus dem Einzelobjekt eine
Menge mit einem Objekt.
$firmen = , $firma1
$firmen.GetType().FullName # System.Object[]
Write-Host "Anzahl Firmen: " $firmen.Count # 1
```



TIPP: Ob die Pipeline ein Einzelobjekt oder eine Menge enthält, können Sie über den Aufruf von `Count` oder `Length` nicht zuverlässig feststellen. Hierzu müssen Sie das der PowerShell zu Grunde liegende .NET fragen, aus welcher Klasse die Pipeline stammt. Dies erfolgt durch den Aufruf `.GetType().FullName`. Wenn dieser Aufruf `System.Object[]` liefert, ist der Inhalt ein „Array von Objekten“, also eine Menge. Die geschweiften Klammern bedeuten in .NET ein „Array“ (Menge).

```
# Einzelobjekt
$pipeline = 1
$pipeline.GetType().FullName # System.Int32
# Menge
$pipeline = 1,2
$pipeline.GetType().FullName # System.Object[]
```

Sie lernen dies im Detail noch im Kapitel „Verwendung von .NET-Klassen“.

5.12.2 Methode GetType()

Da jede PowerShell-Variablen eine Instanz einer .NET-Klasse ist, besitzt jedes Objekt in der Pipeline die Methode `GetType()`, die es von der Mutter aller .NET-Klassen (*System.Object*) erbt. `GetType()` liefert ein *System.Type*-Objekt mit zahlreichen Informationen. Meistens interessiert man sich nur für den Klassennamen, den man aus `FullName` (mit Namensraum) oder `Name` (ohne Namensraum) auslesen kann. `GetType()` ist eine Methode, und daher muss der Pipeline-Inhalt in runden Klammern stehen.

Beispiele zeigt die folgende Abbildung:

```
PS X:\> (Get-Date).GetType()
-----
IsPublic IsSerial Name                                     BaseType
-----
True     True     DateTime                                               System.ValueType

PS X:\> (Get-Process).GetType()
-----
IsPublic IsSerial Name                                     BaseType
-----
True     True     Object[]                                               System.Array

PS X:\> (Get-Process)[0].GetType()
-----
IsPublic IsSerial Name                                     BaseType
-----
True     False    Process                                                System.ComponentModel.Component

PS X:\> (Get-Process)[0].GetType().Name
Process
PS X:\> (Get-Process)[0].GetType().FullName
System.Diagnostics.Process
PS X:\> |
```

Bild 5.18 Einsatz von `GetType()`

Erläuterung: „Name“ ist der Name der Klasse, zu der die Objekte in der Pipeline gehören. „BaseType“ ist der Name der Oberklasse. .NET unterstützt Vererbung, d. h., eine Klasse kann von einer anderen erben (höchstens von einer anderen Klasse; Mehrfachvererbung gibt es nicht!). Dies ist für die PowerShell jedoch zumeist irrelevant und Sie können diese Information ignorieren.

Bei `Get-Date()` ist ein `DateTime`-Objekt in der Pipeline. Der zweite Aufruf liefert nur die Information, dass eine Menge von Objekten in der Pipeline ist. Bei der Anwendung von `GetType()` auf eine Objektmenge in der Pipeline kann man leider noch nicht den Typ erkennen. Hintergrund ist, dass in einer Pipeline Objekte verschiedener Klassen sein können. Der dritte Aufruf, bei dem gezielt ein Objekt (das erste) herausgenommen wird, zeigt dann wieder an, dass es sich um `Process`-Objekte handelt. Den ganzen Klassennamen inklusive des Namensraums bekommt man nur, wenn man explizit die Eigenschaft `FullName` abfragt.

5.12.3 Methode ToString()

Jedes .NET-Objekt bietet die Methode `ToString()`, weil diese Methode von der Basisklasse aller .NET-Klassen `System.Object` an alle Klassen vererbt wird. Das Standardverhalten von `ToString()` ist, dass der Name der Klasse geliefert wird, zu der das Objekt gehört. Das heißt, dass die Ausgabe für alle Instanzen der Klasse gleich ist.

Nur wenige Klassen überschreiben die Implementierung und liefern eine Zeichenkette, die tatsächlich den Inhalt des Objekts wiedergibt. Manchmal wird der Name des Objekts alleine (z. B. bei den Instanzen der Klasse `System.Diagnostics.Process`, die das Commandlet `Get-Process` liefert), manchmal der Name der Klasse mit dem Objektname geliefert (z. B. bei den Instanzen der Klasse `System.Service.ServiceController`, die das Commandlet `Get-Service` liefert).

Listing 5.3 [Basiswissen\Pipelining\ToString.ps1]

```
(Get-Service).ToString() # System.Object[]
(Get-Service w*)[0].ToString() # W32Time
(Get-Process w*)[0].ToString() # System.Diagnostics.Process (wininit)
(Get-Host)[0].ToString() # System.Management.Automation.Internal.Host.InternalHost
(Get-Date).ToString() # liefert aktuelles Datum
```



HINWEIS: Die Konvertierung in den Klassennamen ist das Standardverhalten, das von `System.Object` geerbt wird, und dieses Standardverhalten ist leider auch üblich, da sich die Entwickler der meisten .NET-Klassen bei Microsoft nicht die „Mühe“ gemacht haben, eine sinnvolle Zeichenkettenrepräsentanz zu definieren.

`ToString()` ist üblicherweise **keine** Serialisierung des kompletten Objektinhalts, sondern im besten Fall nur der „Primärschlüssel“ des Objekts. Theoretisch kann eine .NET-Klasse bei `ToString()` alle Werte liefern. Das macht aber fast keine .NET-Klasse. Bei vielen .NET-Klassen liefert `ToString()` nur den Klassennamen.

Ob `ToString()` eine sinnvolle Ausgabe liefert, hängt von der jeweiligen Klasse ab. Der Autor dieses Buchs und auch Sie als Nutzer haben darauf keinen Einfluss für die Klassen, die Microsoft und andere geschrieben haben. Sie können darauf nur in den Klassen Einfluss nehmen, die Sie selbst schreiben.

```

pwsh
PS X:\> (Get-service a*) | foreach {$_.tostring()}
AarSvc_abb5e
AcronisActiveProtectionService
AcrSch2Svc
AdobeARMSvc
afcdpsrv
A3Router
ALG
AMD External Events Utility
AntiVirusKit Client
AppHostSvc
AppIDSvc
AppInfo
AppMgmt
AppReadiness
AppVClient
AppXSvc
aspnet_state
AssignedAccessManagerSvc
AudioEndpointBuilder
Audiosrv
autotimesvc
AVKProxy
AVKWctl
AxInstSV
PS X:\> (Get-Process a*) | foreach {$_.tostring()}
System.Diagnostics.Process (AcroRd32)
System.Diagnostics.Process (AcroRd32)
System.Diagnostics.Process (afcdpsrv)
System.Diagnostics.Process (anti_ransomware_service)
System.Diagnostics.Process (ApplicationFrameHost)
System.Diagnostics.Process (armsvc)
System.Diagnostics.Process (atieclxx)
System.Diagnostics.Process (atiesrxx)
System.Diagnostics.Process (audiodg)
System.Diagnostics.Process (AVKProxy)
System.Diagnostics.Process (AVKWctlx64)
PS X:\> (Get-Host).ToString()
System.Management.Automation.Internal.Host.InternalHost
PS X:\>

```

Bild 5.19

Anwendung von ToString() auf Instanzen verschiedener Klassen

5.12.4 Get-PipelineInfo

Das Commandlet Get-PipelineInfo aus den PowerShell Extensions von *www.IT-Visions.de* liefert drei wichtige Informationen über die Pipeline-Inhalte:

- Anzahl der Objekte in der Pipeline (die Objekte werden durchnummeriert)
- Typ der Objekte in der Pipeline (ganzer Name der .NET-Klasse)
- Zeichenkettenrepräsentation der Objekte in der Pipeline

```

Windows PowerShell
PS T:\> Get-ChildItem T:\Daten\ | Get-PipelineInfo

Count TypeName          String
-----
1 System.IO.DirectoryInfo Kunden
2 System.IO.DirectoryInfo Webservices
3 System.IO.FileInfo     dienste.csv
4 System.IO.FileInfo     links.txt
5 System.IO.FileInfo     LinksToCheck-Error.txt.lnk
6 System.IO.FileInfo     webserver.txt

PS T:\>

```

Bild 5.20

Get-PipelineInfo liefert Informationen, dass sich in dem Dateisystemordner elf Objekte befinden. Davon sind sieben Unterordner (Klasse DirectoryInfo) und vier Dateien (Klasse FileInfo).

Das Stichwort Zeichenkettenrepräsentation (Spalte „String“ in der Abbildung) ist erklärungsbedürftig: Dies ist die Zeichenkettenrepräsentation mit ToString()

5.12.5 Get-Member

Das eingebaute Commandlet Get-Member (Alias: gm) ist sehr hilfreich: Es zeigt den .NET-Klassennamen für die Objekte in der Pipeline sowie die Attribute und Methoden dieser Klasse. Für Get-Process | Get-Member ist die Ausgabe so lang, dass man dazu zwei Bildschirmabbildungen braucht.



HINWEIS: Wenn sich mehrere verschiedene Objekttypen in der Pipeline befinden, werden die Mitglieder aller Typen ausgegeben, gruppiert durch die Kopfsektion, die mit „TypeName:“ beginnt.

```

PowerShell
PS T:\> Get-Process | Get-Member

TypeName: System.Diagnostics.Process

Name                MemberType          Definition
-----
Handles             AliasProperty      Handles = Handlecount
Name                AliasProperty      Name = ProcessName
NPM                 AliasProperty      NPM = NonpagedSystemMemorySize64
PM                  AliasProperty      PM = PagedMemorySize64
SI                  AliasProperty      SI = SessionId
VM                  AliasProperty      VM = VirtualMemorySize64
WS                  AliasProperty      WS = WorkingSet64
Parent              CodeProperty       System.Object Parent{get=GetParentProcess;}
Disposed            Event               System.EventHandler Disposed(System.Object, System.EventArgs)
ErrorDataReceived  Event               System.Diagnostics.DataReceivedEventHandler ErrorDataReceived(System.Object, ...
Exited              Event               System.EventHandler Exited(System.Object, System.EventArgs)
OutputDataReceived Event               System.Diagnostics.DataReceivedEventHandler OutputDataReceived(System.Object, ...
BeginErrorReadLine Method               void BeginErrorReadLine()
BeginOutputReadLine Method               void BeginOutputReadLine()
CancelErrorRead    Method               void CancelErrorRead()
CancelOutputRead   Method               void CancelOutputRead()
Close               Method               void Close()
CloseMainWindow    Method               bool CloseMainWindow()
Dispose             Method               void Dispose(), void IDisposable.Dispose()
Equals              Method               bool Equals(System.Object obj)
GetHashCode         Method               int GetHashCode()
GetLifetimeService Method               System.Object GetLifetimeService()
GetType             Method               type GetType()
InitializeLifetimeService Method               System.Object InitializeLifetimeService()
Kill                Method               void Kill(), void Kill(bool entireProcessTree)
Refresh             Method               void Refresh()
Start               Method               bool Start()
ToString            Method               string ToString()
WaitForExit         Method               void WaitForExit(), bool WaitForExit(int milliseconds)
WaitForExitAsync   Method               System.Threading.Tasks.Task WaitForExitAsync(System.Threading.CancellationTo...
WaitForInputIdle   Method               bool WaitForInputIdle(), bool WaitForInputIdle(int milliseconds)
__NounName          NoteProperty        string __NounName=Process
BasePriority         Property             int BasePriority {get;}
Container            Property             System.ComponentModel.IContainer Container {get;}
EnableRaisingEvents Property             bool EnableRaisingEvents {get;set;}
ExitCode            Property             int ExitCode {get;}
ExitTime            Property             datetime ExitTime {get;}
Handle              Property             System.IntPtr Handle {get;}
HandleCount         Property             int HandleCount {get;}
HasExited           Property             bool HasExited {get;}
Id                  Property             int Id {get;}
MachineName         Property             string MachineName {get;}
MainModule          Property             System.Diagnostics.ProcessModule MainModule {get;}
MainWindowHandle    Property             System.IntPtr MainWindowHandle {get;}
MainWindowTitle     Property             string MainWindowTitle {get;}
MaxWorkingSet       Property             System.IntPtr MaxWorkingSet {get;set;}
MinWorkingSet       Property             System.IntPtr MinWorkingSet {get;set;}
Modules             Property             System.Diagnostics.ProcessModuleCollection Modules {get;}
NonpagedSystemMemorySize Property             int NonpagedSystemMemorySize {get;}
NonpagedSystemMemorySize64 Property             long NonpagedSystemMemorySize64 {get;}

```

Bild 5.21 Teil 1 der Ausgabe von Get-Process | Get-Member

```

NonpagedSystemMemorySize64 Property long NonpagedSystemMemorySize64 {get;}
PagedMemorySize Property int PagedMemorySize {get;}
PagedMemorySize64 Property long PagedMemorySize64 {get;}
PagedSystemMemorySize Property int PagedSystemMemorySize {get;}
PagedSystemMemorySize64 Property long PagedSystemMemorySize64 {get;}
PeakPagedMemorySize Property int PeakPagedMemorySize {get;}
PeakPagedMemorySize64 Property long PeakPagedMemorySize64 {get;}
PeakVirtualMemorySize Property int PeakVirtualMemorySize {get;}
PeakVirtualMemorySize64 Property long PeakVirtualMemorySize64 {get;}
PeakWorkingSet Property int PeakWorkingSet {get;}
PeakWorkingSet64 Property long PeakWorkingSet64 {get;}
PriorityBoostEnabled Property bool PriorityBoostEnabled {get;set;}
PriorityClass Property System.Diagnostics.ProcessPriorityClass PriorityClass {get;set;}
PrivateMemorySize Property int PrivateMemorySize {get;}
PrivateMemorySize64 Property long PrivateMemorySize64 {get;}
PrivilegedProcessorTime Property timespan PrivilegedProcessorTime {get;}
ProcessName Property string ProcessName {get;}
ProcessorAffinity Property System.IntPtr ProcessorAffinity {get;set;}
Responding Property bool Responding {get;}
SafeHandle Property Microsoft.Win32.SafeHandles.SafeProcessHandle SafeHandle {get;}
SessionId Property int SessionId {get;}
Site Property System.ComponentModel.ISite Site {get;set;}
StandardError Property System.IO.StreamReader StandardError {get;}
StandardInput Property System.IO.StreamWriter StandardInput {get;}
StandardOutput Property System.IO.StreamReader StandardOutput {get;}
StartInfo Property System.Diagnostics.ProcessStartInfo StartInfo {get;set;}
StartTime Property datetime StartTime {get;}
SynchronizingObject Property System.ComponentModel.ISynchronizeInvoke SynchronizingObject {get;set;}
Threads Property System.Diagnostics.ProcessThreadCollection Threads {get;}
TotalProcessorTime Property timespan TotalProcessorTime {get;}
UserProcessorTime Property timespan UserProcessorTime {get;}
VirtualMemorySize Property int VirtualMemorySize {get;}
VirtualMemorySize64 Property long VirtualMemorySize64 {get;}
WorkingSet Property int WorkingSet {get;}
WorkingSet64 Property long WorkingSet64 {get;}
PSConfiguration PropertySet PSConfiguration {Name, Id, PriorityClass, FileVersion}
PSResources PropertySet PSResources {Name, Id, Handlecount, WorkingSet, NonPagedMemorySize, PagedMem...
CommandLine ScriptProperty System.Object CommandLine {get=...
Company ScriptProperty System.Object Company {get=$this.Mainmodule.FileVersionInfo.CompanyName;}
CPU ScriptProperty System.Object CPU {get=$this.TotalProcessorTime.TotalSeconds;}
Description ScriptProperty System.Object Description {get=$this.Mainmodule.FileVersionInfo.FileDescript...
FileVersion ScriptProperty System.Object FileVersion {get=$this.Mainmodule.FileVersionInfo.FileVersion;}
Path ScriptProperty System.Object Path {get=$this.Mainmodule.FileName;}
Product ScriptProperty System.Object Product {get=$this.Mainmodule.FileVersionInfo.ProductName;}
ProductVersion ScriptProperty System.Object ProductVersion {get=$this.Mainmodule.FileVersionInfo.ProductVe...

PS T:\> |

```

Bild 5.22 Teil 2 der Ausgabe von Get-Process | Get-Member

Die Ausgabe zeigt, dass aus der Sicht der PowerShell eine .NET-Klasse sieben Arten von Mitgliedern hat:

1. Method (Methode)
2. Property (Eigenschaft)
3. PropertySet (Eigenschaftssatz)
4. NoteProperty (Notizeigenschaft)
5. ScriptProperty (Skripteigenschaft)
6. CodeProperty (Codeeigenschaft)
7. AliasProperty (Aliaseigenschaft)



HINWEIS: Von den oben genannten Mitgliedsarten sind nur „Method“ und „Property“ tatsächliche Mitglieder der .NET-Klasse. Alle anderen Mitgliedsarten sind Zusätze, welche die PowerShell mittels des sogenannten Extended Type System (ETS) dem .NET-Objekt hinzugefügt hat.

Die Ausgabe von `Get-Member` kann man verkürzen, indem man nur eine bestimmte Art von Mitgliedern ausgeben lässt. Diese erreicht man über den Parameter `-MemberType` (kurz: `-m`). Der folgende Befehl listet nur die Properties auf:

```
Get-Process | Get-Member -MemberType Properties
```

Außerdem ist eine Filterung beim Namen möglich:

```
Get-Process | Get-Member *set*
```

Der obige Befehl listet nur solche Mitglieder der Klasse *Process* auf, deren Name das Wort „set“ enthält.

5.12.6 Methoden (Mitgliedsart Method)

Methoden (Mitgliedsart Method) sind Operationen, die man auf dem Objekt aufrufen kann und die eine Aktion auslösen, z. B. beendet `Kill()` den Prozess. Methoden können aber auch Daten liefern oder Daten in dem Objekt verändern.



ACHTUNG: Beim Aufruf von Methoden sind immer runde Klammern anzugeben, auch wenn es keine Parameter gibt. Ohne die runden Klammern erhält man Informationen über die Methode, man ruft aber nicht die Methode selbst auf.

5.12.7 Eigenschaften (Mitgliedsart Property)

Eigenschaften (Mitgliedsart Property) sind Datenelemente, die Informationen aus dem Objekt enthalten oder mit denen man Informationen an das Objekt übergeben kann, z. B. `MaxWorkingSet`.



ACHTUNG: In PowerShell 1.0 sah die Aussage von `Get-Member` noch etwas anders aus (siehe nächste Abbildung). Man sieht dort, dass es zu jedem Property zwei Methoden gibt, z. B. `get_MaxWorkingSet()` und `set_MaxWorkingSet()`. Die Ursache dafür liegt in den Interna des .NET Frameworks: Dort werden Properties (nicht aber sogenannte Fields, eine andere Art von Eigenschaften) durch ein Methodenpaar abgebildet: eine Methode zum Auslesen der Daten (genannt „Get-Methode“ oder „Getter“), eine andere Methode zum Setzen der Daten (genannt „Set-Methode“ oder „Setter“). Einige Anfänger störte die „Aufblähung“ der Liste durch diese Optionen. Seit PowerShell 2.0 zeigte `Get-Member` die Getter-Methoden (`get_`) und Setter-Methoden (`set_`) nur noch an, wenn man den Parameter `-force` verwendet.

```

Administrator: C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
PS C:\Windows\System32\WindowsPowerShell\v1.0> Get-Process | Get-Member

TypeName: System.Diagnostics.Process

Name      MemberType      Definition
-----
Handles   AliasProperty   Handles = Handlecount
Name      AliasProperty   Name = ProcessName
NPM       AliasProperty   NPM = NonpagedSystemMemorySize
PM        AliasProperty   PM = PagedMemorySize
VM        AliasProperty   VM = VirtualMemorySize
WS        AliasProperty   WS = WorkingSet
Disposed  Event            System.EventHandler Disposed(System.Object, System.EventArgs)
ErrorDataReceived Event            System.Diagnostics.DataReceivedEventHandler ErrorDataReceived(System.Object, System.EventArgs)
Exited    Event            System.EventHandler Exited(System.Object, System.EventArgs)
OutputDataReceived Event            System.Diagnostics.DataReceivedEventHandler OutputDataReceived(System.Object, System.EventArgs)
BeginErrorReadLine Method           System.Void BeginErrorReadLine()
BeginOutputReadLine Method           System.Void BeginOutputReadLine()
CancelErrorRead Method          System.Void CancelErrorRead()
CancelOutputRead Method          System.Void CancelOutputRead()
Close     Method          System.Void Close()
CloseMainWindow Method         bool CloseMainWindow()
CreateObjRef Method         System.Runtime.Remoting.ObjRef CreateObjRef(type requestedType)
Dispose   Method          System.Void Dispose()
Equals    Method          bool Equals(System.Object obj)
GetHashCode Method         int GetHashCode()
GetLifetimeService Method         System.Object GetLifetimeService()
GetType   Method          type GetType()
InitializeLifetimeService Method         System.Object InitializeLifetimeService()
Kill      Method          System.Void Kill()
Refresh   Method          System.Void Refresh()
Start     Method          bool Start()
ToString  Method          string ToString()
WaitForExit Method         bool WaitForExit(int milliseconds), System.Void WaitForExit()
WaitForInputIdle Method         bool WaitForInputIdle(int milliseconds), bool WaitForInputIdle()
__NounName NoteProperty   System.String __NounName=Process
BasePriority Property        System.Int32 BasePriority {get;}
Container Property        System.ComponentModel.IContainer Container {get;}
EnableRaisingEvents Property        System.Boolean EnableRaisingEvents {get;set;}
ExitCode  Property        System.Int32 ExitCode {get;}
ExitTime  Property        System.DateTime ExitTime {get;}
Handle     Property        System.IntPtr Handle {get;}
HandleCount Property        System.Int32 HandleCount {get;}
HasExited Property        System.Boolean HasExited {get;}
Id         Property        System.Int32 Id {get;}
MachineName Property        System.String MachineName {get;}
MainModule Property        System.Diagnostics.ProcessModule MainModule {get;}
MainWindowHandle Property        System.IntPtr MainWindowHandle {get;}
MainWindowTitle Property        System.String MainWindowTitle {get;}
MaxWorkingSet Property        System.IntPtr MaxWorkingSet {get;set;}
MinWorkingSet Property        System.IntPtr MinWorkingSet {get;set;}
Modules   Property        System.Diagnostics.ProcessModuleCollection Modules {get;}

```

Bild 5.23 Anzeige der Getter und Setter in PowerShell 1.0

Fortgeschrittene Benutzer bevorzugen die Auflistung der Getter und Setter. Man kann erkennen, welche Aktionen auf einem Property möglich sind. Fehlt der Setter, kann die Eigenschaft nicht verändert werden (z.B. `StartTime` bei der Klasse `Process`). Fehlt der Getter, kann man die Eigenschaft nur setzen. Dafür gibt es kein Beispiel in der Klasse `Process`. Dieser Fall kommt auch viel seltener vor, wird aber z.B. bei Kennwörtern eingesetzt, die man nicht wiedergewinnen kann, weil sie nicht im Klartext, sondern nur als Hash-Wert abgespeichert werden.

Für den PowerShell-Nutzer bedeutet die Existenz von Gettern und Settern, dass er zwei Möglichkeiten hat, Daten abzurufen. Über die Eigenschaft (Property):

```
Get-Process | Where-Object { $_.name -eq "iexplore" } | Foreach-Object
{ $_.PriorityClass }
```

oder die entsprechende "Get"-Methode:

```
Get-Process | Where-Object { $_.name -eq "iexplore" } | Foreach-Object
{ $_.get_PriorityClass() }
```

Analog gibt es für das Schreiben die Option über die Eigenschaft:

```
Get-Process | Where-Object { $_.name -eq "iexplore" } | Foreach-Object
{ $_.PriorityClass = "High" }
```

oder die entsprechende „Set“-Methode:

```
Get-Process | Where-Object { $_.name -eq "iexplore" } | Foreach-Object
{ $_.set_PriorityClass("High") }
```



TIPP: Auch hier kann man wieder grundsätzlich die verkürzte Schreibweise seit PowerShell-Version 3.0 anwenden, also:

```
(Get-Process | Where-Object { $_.name -eq "iexplore" }).PriorityClass
(Get-Process | Where-Object { $_.name -eq "iexplore" }).get_
PriorityClass()
(Get-Process | Where-Object { $_.name -eq "iexplore" }).set_
PriorityClass("High")
```

Syntaktisch nicht erlaubt ist aber:

```
(Get-Process | Where-Object { $_.name -eq "iexplore" }).PriorityClass =
"High"
```

Hier geht nur die o. g. Schreibweise mit `Foreach-Object`.

5.12.8 Eigenschaftssätze (PropertySet)

Eigenschaftssätze (PropertySet) sind eine Zusammenfassung einer Menge von Eigenschaften unter einem gemeinsamen Dach. Beispielsweise umfasst der Eigenschaftssatz `psResources` alle Eigenschaften, die sich auf den Ressourcenverbrauch eines Prozesses beziehen. Dies ermöglicht es, dass man nicht alle diesbezüglichen Eigenschaften einzeln nennen muss, sondern schreiben kann:

```
Get-Process | Select-Object psResources | Format-Table
```

Die Eigenschaftssätze gibt es nicht im .NET Framework; sie sind eine Eigenart der PowerShell und definiert in der Datei `types.ps1xml` im Installationsordner der PowerShell.

```
PS T:\> Get-Process | Select-Object psResources | Format-Table
```

Name	Id	HandleCount	WorkingSet	PagedMemorySize	PrivateMemorySize	VirtualMemorySize	TotalProcessorTime
AcroRd32	7264	696	146747392	123883520	123883520	380379136	00:00:03.5312500
AcroRd32	13724	449	26161152	11595776	11595776	150568960	00:00:00.1562500
armsvc	4572	143	6991872	1921024	1921024	65519616	
atieclxx	3252	225	10227712	2854912	2854912	109641728	
atiesrxx	3084	140	6004736	1875968	1875968	42979328	
audiogd	7244	210	14209024	8101888	8101888	64712704	00:00:00.6250000
AVKProxy	4652	388	4210688	7131136	7131136	127819776	
AVKctlx64	2880	1649	186978304	171806720	171806720	413323264	
backgroundTaskHost	11324	214	16220160	4587520	4587520	111869952	00:00:00.0468750
CCC	2104	924	7221248	80384000	80384000	908488704	00:00:02.8125000
chrome	608	401	94724096	65105920	65105920	1025171456	00:00:01.8593750
chrome	1840	204	9650176	2387968	2387968	105435136	00:00:00.0468750
chrome	2024	1578	181137408	117514240	117514240	524996608	00:00:09.1718750
chrome	10272	289	37580800	24600576	24600576	810311680	00:00:01.1718750
chrome	11124	285	33673216	21262336	21262336	799825920	00:00:00.2968750
chrome	11532	144	10096640	2252800	2252800	98193408	00:00:00.0156250
chrome	12232	318	39460864	24760320	24760320	814440448	00:00:00.3906250
chrome	13792	578	66674688	72155136	72155136	483831808	00:00:01.5468750
chrome	13796	290	40976384	29523968	29523968	812408832	00:00:00.3593750
chrome	14252	287	37462016	24379392	24379392	806641664	00:00:00.3750000
conhost	9420	229	15302656	4157440	4157440	113057792	00:00:00.2656250
csrss	628	746	5365760	2158592	2158592	61370368	
csrss	736	606	5709824	10141696	10141696	72417280	
dashost	5632	271	14905344	4575232	4575232	69120000	
dllhost	9320	247	33275904	24334336	24334336	393596928	00:00:00.2656250
dllhost	10648	172	11730944	4620288	4620288	354037760	00:00:00.1093750
DSAService	4556	614	41193472	26361856	26361856	237490176	
DSATray	604	493	44281856	38748160	38748160	320065536	00:00:00.4687500
dwm	1388	712	135815168	148025344	148025344	429637632	
explorer	7380	2580	130813952	61431808	61431808	563249152	00:00:10.7812500
ExpressTray	12992	988	70295552	56795136	56795136	409272320	00:00:01.0625000
fontdrvhost	536	46	4747264	2023424	2023424	63488000	
fontdrvhost	548	46	11591680	4321280	4321280	148639744	
GarminService	4564	1274	70078464	45420544	45420544	318988288	
GdAgentSrv	4696	616	34365440	7983104	7983104	141025280	
GdAgentUi	10968	295	1474560	3985408	3985408	116932608	00:00:00.0781250
GDScan	2008	735	43450368	692846592	692846592	831651840	
GoodSync-v10	12896	391	279642112	267563008	267563008	439402496	00:00:48.9218750
GoogleCrashHandler	6268	154	995328	2105344	2105344	67391488	
GoogleCrashHandler64	1420	136	741376	1953792	1953792	70037504	
gs-server	6692	381	17182720	9486336	9486336	106037248	
Idle	0	0	8192	53248	53248	65536	
ieplere	2456	630	39108608	13930496	13930496	211484672	00:00:00.5000000
ieplere	4536	649	62017536	35028992	35028992	305123328	00:00:00.3281250
IpOverUsbSvc	4444	262	13094912	8650752	8650752	128581632	

Bild 5.24 Verwendung des Eigenschaftssatzes „psResources“

```
<PropertySet>
  <Name>PSConfiguration</Name>
  <ReferencedProperties>
    <Name>Name</Name>
    <Name>Id</Name>
    <Name>PriorityClass</Name>
    <Name>FileVersion</Name>
  </ReferencedProperties>
</PropertySet>
<PropertySet>
  <Name>PSResources</Name>
  <ReferencedProperties>
    <Name>Name</Name>
    <Name>Id</Name>
    <Name>HandleCount</Name>
    <Name>WorkingSet</Name>
    <Name>NonPagedMemorySize</Name>
    <Name>PagedMemorySize</Name>
    <Name>PrivateMemorySize</Name>
    <Name>VirtualMemorySize</Name>
    <Name>Threads.Count</Name>
    <Name>TotalProcessorTime</Name>
  </ReferencedProperties>
</PropertySet>
```

Bild 5.25

Definition der Eigenschaftssätze für die Klasse *System.Diagnostics.Process* in *types.psxml*

5.12.9 Notizeigenschaften (NoteProperty)

Notizeigenschaften (NoteProperties) sind zusätzliche Datenelemente, die nicht dem .NET-Objekt entstammen, sondern welche die PowerShell-Infrastruktur hinzugefügt hat. Im Beispiel der Ergebnismenge des Commandlets `Get-Process` ist dies `__NounName`, der einen Kurznamen der Klasse liefert. Andere Klassen haben zahlreiche Notizeigenschaften. Notizeigenschaften gibt es nicht im .NET Framework; sie sind eine Eigenart der PowerShell.



HINWEIS: Man kann einem Objekt zur Laufzeit eine Notizeigenschaft hinzufügen, siehe das Kapitel „*Dynamische Objekte*“.

5.12.10 Skripteigenschaften (ScriptProperty)

Eine **Skripteigenschaft (ScriptProperty)** ist eine berechnete Eigenschaft, also eine Information, die nicht im .NET-Objekt selbst gespeichert ist. Dabei muss die Berechnung nicht notwendigerweise eine mathematische Berechnung sein; es kann sich auch um den Zugriff auf die Eigenschaften eines untergeordneten Objekts handeln. Der Befehl

```
Get-Process | Select-Object name, product | where { $_.product -ne "" -and $_.product -ne $null }
```

listet alle Prozesse mit den Produkten auf, zu denen der Prozess gehört (siehe folgende Abbildung). Dies ist gut zu wissen, wenn man auf seinem System einen Prozess sieht, den man nicht kennt und von dem man befürchtet, dass es sich um einen Schädling handeln könnte.



TIPP: Nicht zu allen Prozessen bekommt man eine Produktinfo. Manchmal liefert die Eigenschaft `$null`, manchmal eine leere Zeichenkette. Die obige Bedingung schließt beides aus.

Die Information über das Produkt steht nicht in dem Prozess (Windows listet diese Information im Taskmanager ja auch nicht auf), aber in der Datei, die den Programmcode für den Prozess enthält. Das .NET Framework bietet über die `MainModule.FileVersionInfo.ProductName` einen Zugang zu dieser Information. Anstelle des Befehls

```
Get-Process | Select-Object name, Mainmodule.FileVersionInfo.ProductName
```

bietet Microsoft durch die Skripteigenschaft eine Abkürzung an. Diese Abkürzung ist definiert in der Datei `types.ps1xml` im Installationsordner der PowerShell.

```

PowerShell
PS T:\> Get-Process | Select-Object name, product | where { $_.product -ne "" -and $_.product -ne $null }

Name                                     Product
----                                     -
Acrobat                                  Adobe Acrobat DC
Acrobat                                  Adobe Acrobat DC
ApplicationFrameHost                    Microsoft Windows Operating System
Code                                     Visual Studio Code
Code                                     Visual Studio Code
Code                                     Visual Studio Code
Code                                     Visual Studio Code
CompPkgSrv                               Microsoft Windows Operating System
conhost                                  Microsoft Windows Operating System
CyberProtectHomeOfficeMonitor           Acronis Cyber Protect Home Office
devenv                                   Microsoft Visual Studio
dllhost                                  Microsoft Windows Operating System
dllhost                                  Microsoft Windows Operating System
eWallet                                  eWallet
explorer                                 Microsoft Windows Operating System
filezilla                                FileZilla
git-credential-manager-core             git-credential-manager-core
HelpPane                                 Microsoft Windows Operating System
iisexpress                               Internet Information Services
iisexpressstray                          Microsoft Web Platform Extensions
ITV_CRMTools                             ITV CRM
ITV_CRMTools                             ITV CRM
laclient                                  Logitech Analytics Client
LCDCLock                                  Logitech GamePanel Software
LCDCountdown                             Logitech GamePanel Software
LCDMedia                                  Logitech GamePanel Software
LCDMovieViewer                           Logitech GamePanel Software
LCDPictureViewer                         Logitech GamePanel Software
LCDPOP3                                   Logitech GamePanel Software
LCDRSS                                    Logitech GamePanel Software
LCDWebCam                                 Logitech GamePanel Software
LCore                                     Logitech Gaming Framework
LockApp                                   Microsoft Windows Operating System
logitech_discord                         Logitech G Discord Applet

```

Bild 5.26 Auflistung der berechneten Eigenschaft „Product“

```

615 |<<ScriptProperty>
616 |<<Name>Product</Name>
617 |<<GetScriptBlock>$this.Mainmodule.FileVersionInfo.ProductName</GetScriptBlock>
618 |</ScriptProperty>

```

Bild 5.27 Definition einer Skripteigenschaft in der types.psxml

Skripteigenschaften gibt es nicht im .NET Framework; sie sind eine Eigenart der PowerShell. Man kann einem Objekt zur Laufzeit eine Skripteigenschaft hinzufügen, siehe das Kapitel „Dynamische Objekte“.

5.12.11 Codeeigenschaften (Code Property)

Eine **Codeeigenschaft (CodeProperty)** entspricht einer Script Property, allerdings ist der Programmcode nicht als Skript in der PowerShell-Sprache, sondern als .NET-Programmcode hinterlegt.

5.12.12 Aliaseigenschaft (AliasProperty)

Eine **Aliaseigenschaft (AliasProperty)** ist eine verkürzte Schreibweise für ein Property. Dahinter steckt keine Berechnung, sondern nur eine Verkürzung des Namens. Beispielsweise ist `WS` eine Abkürzung für `WorkingSet`. Auch die Aliaseigenschaften sind in der Datei `types.ps1xml` im Installationsordner der PowerShell definiert. Aliaseigenschaften sind ebenfalls eine PowerShell-Eigenart.

5.12.13 Hintergrundwissen: Adapted Type System (ATS)/ Extended Type System (ETS)

Als **Extended Type System (ETS)** bezeichnet Microsoft die Möglichkeit, .NET-Klassen in der PowerShell um Klassenmitglieder zu erweitern, ohne im klassischen Sinne der Objektivierung von diesen Klassen zu erben.

Als **Adapted Type System (ATS)** bezeichnet Microsoft die grundsätzliche Anpassung von .NET-Klassen aus der .NET-Klassenbibliothek auf die Bedürfnisse von PowerShell-Benutzern. Wie bereits dargestellt, zeigt die PowerShell für viele .NET-Objekte mehr Mitglieder an, als eigentlich in der .NET-Klasse definiert sind. In einigen Fällen werden aber auch Mitglieder ausgeblendet.

Die Ergänzung von Mitgliedern per ATS wird verwendet, um bei einigen .NET-Klassen, die Metaklassen für die eigentlichen Daten sind (z. B. `ManagementObject` für WMI-Objekte, `ManagementClass` für WMI-Klassen, `DirectoryEntry` für Einträge in Verzeichnisdiensten und `DataRow` für Datenbankzeilen), die Daten direkt ohne Umweg dem PowerShell-Nutzer zur Verfügung zu stellen.

Mitglieder werden ausgeblendet, wenn sie in der PowerShell nicht nutzbar sind oder es bessere Alternativen durch die Ergänzungen gibt.

In der Dokumentation nimmt das PowerShell-Entwicklungsteam dazu wie folgt Stellung:

- Some .NET objects are "meta" objects (for example: WMI Objects, ADO objects, and XML objects) whose members describe the data they contain. However, in a scripting environment it is the contained data that is most interesting, not the description of the contained data. ETS resolves this issue by introducing the notion of Adapters that adapt the underlying .NET object to have the expected default semantics.
- Some .NET Object members are inconsistently named, provide an insufficient set of public members, or provide insufficient capability. ETS resolves this issue by introducing the ability to extend the .NET object with additional members.

Bild 5.28 Quelle: <https://docs.microsoft.com/en-us/powershell/scripting/developer/ets/overview>

Dies heißt im Klartext, dass das PowerShell-Team mit der Arbeit des Entwicklungsteams der .NET-Klassenbibliothek nicht ganz zufrieden ist.

Das ATS verpackt grundsätzlich jedes Objekt, das von einem Commandlet in die Pipeline gelegt wird, in ein PowerShell-Objekt des Typs `PSObject`. Die Implementierung der Klasse `PSObject` entscheidet dann, was für die folgenden Commandlets und Befehle sichtbar ist.

Diese Entscheidung wird beeinflusst durch verschiedene Instrumente:

- PowerShell-Objektadapter, die für bestimmte Typen wie `ManagementObject`, `ManagementClass`, `DirectoryEntry` und `DataRow` implementiert wurden,
- die Deklarationen in der `types.ps1xml`-Datei,
- in den Commandlets hinzugefügte Mitglieder,
- mit dem Commandlet `Add-Member` hinzugefügte Mitglieder.

Die folgende Tabelle zeigt die .NET-Klassen, die im Standard per ATS verändert werden:

Tabelle 5.2 .NET-Klassen mit ATS

PowerShell-Wrapper	.NET Framework-Klasse
WMI Class	System.Management.ManagementClass
WMI Object	System.Management.ManagementObject
ADSI Object	System.DirectoryServices.DirectoryEntry
ADO.NET DataRowView	System.Data.DataRowView
ADO.NET DataRow	System.Data.DataRow
XML	System.Xml.XmlNode
PSObject	System.Management.Automation.PSObject
PSMemberSet	System.Management.Automation.PSMemberSet
COM Object	System.__ComObject
.NET Object	System.Object

■ 5.13 Filtern

Nicht immer will man alle Objekte weiterverarbeiten, die ein Commandlet liefert. Einschränkungskriterien sind Bedingungen (z. B. nur Prozesse, bei denen der Speicherbedarf größer ist als 10 000 000 Byte) oder die Position (z. B. nur die fünf Prozesse mit dem größten Speicherbedarf). Zur wertabhängigen Einschränkung verwendet man das Commandlet `Where-Object` (Alias `where`).

```
Get-Process | Where-Object {$_.ws -gt 10000000 }
```

Einschränkungen über die Position definiert man mit dem `Select-Object` (in dem nachfolgenden Befehl für das oben genannte Beispiel ist zusätzlich noch eine Sortierung eingebaut, damit die Ausgabe einen Sinn ergibt):

```
Get-Process | Sort-Object ws -desc | Select-Object -first 5
```

Analog dazu sind die kleinsten Speicherfresser zu ermitteln mit:

```
Get-Process | Sort-Object ws -desc | Select-Object -last 5
```

Mit `Select-Object` kann man auch eine Teilmenge aus der Mitte auswählen, indem man am Beginn einige Elemente mit `-Skip` überspringt:

```
Get-Process | Sort-Object ws -desc | Select-Object -skip 5 -first 5
```

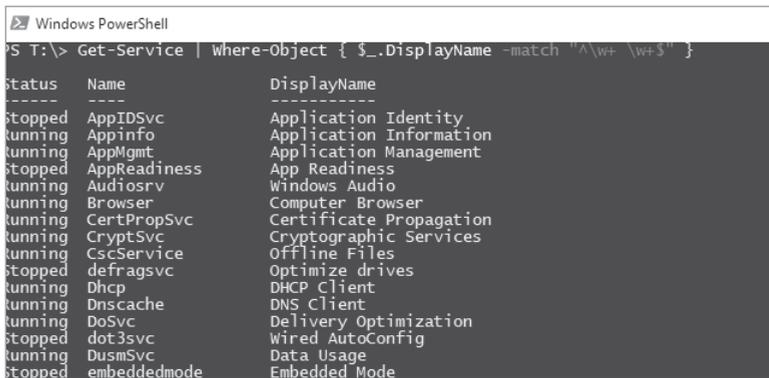
5.13.1 Operatoren

Etwas gewöhnungsbedürftig ist die Schreibweise der Vergleichsoperatoren: Statt `>=` schreibt man `-ge` (siehe folgende Tabelle). Die Nutzung regulärer Ausdrücke ist möglich mit dem Operator `-Match`.

Dazu zwei **Beispiele**:

1. Der folgende Ausdruck listet alle Systemdienste, deren Beschreibung aus zwei durch ein Leerzeichen getrennten Wörtern besteht.

```
Get-Service | Where-Object { $_.DisplayName -match "\w+ \w+$" }
```



```

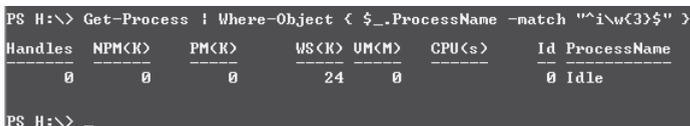
Windows PowerShell
PS T:\> Get-Service | Where-Object { $_.DisplayName -match "\w+ \w+$" }
-----
status Name           DisplayName
-----
Stopped AppIDSvc         Application Identity
Running Appinfo         Application Information
Running AppMgmt      Application Management
Stopped AppReadiness App Readiness
Running Audiosrv    Windows Audio
Running Browser     Computer Browser
Running CertPropSvc Certificate Propagation
Running CryptSvc    Cryptographic Services
Running CscService  Offline Files
Stopped defragSvc   Optimize drives
Running Dhcp         DHCP Client
Running Dnscache    DNS Client
Running DoSvc       Delivery Optimization
Stopped dot3Svc      Wired AutoConfig
Running DismSvc     Data Usage
Stopped embeddedmode Embedded Mode

```

Bild 5.29 Ausgabe zu obigem Beispiel

2. Der folgende Ausdruck listet alle Prozesse, deren Namen mit einem "i" starten und danach aus drei Buchstaben bestehen.

```
Get-Process | Where-Object { $_.ProcessName -match "^i\w{3}$" }
```



```

PS H:\> Get-Process | Where-Object { $_.ProcessName -match "^i\w{3}$" }
-----
Handles NPM(K) PM(K) WS(K) UM(K) CPU(s) Id ProcessName
-----
0 0 0 24 0 0 0 Idle

```

Bild 5.30
Ausgabe zu obigem
Beispiel

Tabelle 5.3 Vergleichsoperatoren der PowerShell

Vergleich unter Ignorierung der Groß-/Kleinschreibung	Vergleich unter Berücksichtigung der Groß-/Kleinschreibung	Bedeutung
-lt / -ilt	-clt	Kleiner
-le / -ile	-cle	Kleiner oder gleich
-gt / -igt	-cgt	Größer
-ge / -ige	-cge	Größer oder gleich
-eq / -ieq	-ceq	Gleich
-ne / -ine	-cne	Nicht gleich
-like / -ilike	-clike	Ähnlichkeit zwischen Zeichenketten, Einsatz von Platzhaltern (* und ?) möglich
-notlike / -inotlike	-cnotlike	Keine Ähnlichkeit zwischen Zeichenketten, Einsatz von Platzhaltern (* und ?) möglich
-match / -imatch	-cmatch	Vergleich mit regulärem Ausdruck
-notmatch / -inotmatch	-cnotmatch	Stimmt nicht mit regulärem Ausdruck überein
-is		Typvergleich, z. B. (Get-Date) -is [DateTime]
-in -contains		Ist enthalten in Menge
-notin -notcontains		Ist nicht enthalten in Menge

Tabelle 5.4 Logische Operatoren in der PowerShell-Sprache

Logischer Operator	Bedeutung
-not oder !	Nicht
-and	Und
-or	Oder

5.13.2 Vereinfachte Schreibweise von Bedingungen seit PowerShell 3.0

Microsoft hat versucht, die Schreibweise von Bedingungen nach Where-Object seit PowerShell-Version 3.0 zu vereinfachen.

Die Bedingung

```
Get-Service | where-object { $_.status -eq "running" }
```

kann der Nutzer seitdem vereinfacht schreiben als

```
Get-Service | where-object status -eq "running".
```

Dass auch

```
Get-Service | where-object -eq status "running"
```

und

```
Get-Service | where-object status "running" -eq
```

zum gleichen Ergebnis führen, wirkt befremdlich.

Allerdings funktioniert die neue Syntaxform nur in den einfachsten Fällen. Bei der Verwendung von `-and` und `-or` ist die Verkürzung nicht möglich.

So sind folgende Befehle **nicht** erlaubt:

```
Get-Process | Where-Object Name -eq "iexplore" -or name -eq "Chrome" -or name -eq "Firefox" | Stop-Process
```

```
Get-Service | where-object status -eq running -and name -like "a*"
```

Korrekt muss es heißen:

```
Get-Process | Where-Object { $_.Name -eq "iexplore" -or $_.name -eq "Chrome" -or $_.name -eq "Firefox" } | Stop-Process
```

```
Get-Service | where-object { $_.status -eq "running" -and $_.name -like "a*" }
```

Grund für das Versagen bei komplexeren Ausdrücken ist, dass Microsoft die Syntaxvereinfachung über die Parameter abgebildet hat. So wird in der einfachsten Form `-eq` als Parameter von `where-object` betrachtet. Microsoft hätte da lieber den Parser grundsätzlich überarbeiten sollen.

5.13.3 Where()-Methode seit PowerShell 4.0

In PowerShell hat Microsoft eine Optionen für das Filtern von Pipelines eingebaut, die sich vor allem an fortgeschrittene PowerShell-Nutzer richtet bzw. an Softwareentwickler, die die PowerShell nutzen. Alternativ zum Commandlet `Where-Object` kann man nun auch mit einer `Where()`-Methode filtern. Anstelle von

```
Get-Service a* | where status -eq "stopped"
```

oder

```
Get-Service a* | Where-Object { $_.status -eq "stopped" }
```

Ist nun auch diese Syntax möglich:

```
(Get-Service a*).Where({ $_.status -eq "stopped"})
```

Dabei ist die Eingabemenge, die auch eine Pipeline mit mehreren Commandlets sein kann, zu klammern.

Man kann auch mehrere Bedingungen verketteten:

```
(Get-Service).Where({($_.name.startswith("a") -or $_.name.startswith("A")) -and $_.status -eq "stopped"})
```

Soweit bietet die Methode Where() nichts, was das Commandlet Where-Object nicht auch könnte - nur in anderer Syntax.

Interessant sind die weiteren Optionen. Man kann bei der Where()-Methode einen weiteren Parameter angeben: Default, First, Last, SkipUntil, Until, Split. Dieser Parameter muss als Zeichenkette übergeben werden.

Beispiele:

```
# Alle, bis Bedingung erfüllt
(1..10).Where({ $_ -eq 5}, 'Until')
# Nur das erste Objekt, das Bedingung erfüllt, also 6
(1..10).Where({ $_ -gt 5}, 'First')
# Nur das letzte Objekt, das Bedingung erfüllt, also 10
(1..10).Where({ $_ -gt 5}, 'Last')
```

Sehr spannend ist die Möglichkeit, eine Menge mit Where() im Modus 'Split' in zwei Teilmengen zu teilen und als Ergebnis des Befehls direkt zwei Ausgabevariablen zu erhalten:

```
# Teile eine Menge von Zahlen in zwei Teile
$kleiner,$groesser = (Get-Random -max 49 -Count 7).Where({ $_ -lt 30}, 'Split')
"# Zahlen < 5"
$kleiner
"# Zahlen >= 5"
$groesser
```



HINWEIS: Dieses Beispiel setzt PowerShell 7.0 oder höher voraus, da der Parameter -count bei Get-Random erst in PowerShell 7 eingeführt wurde.

```
# Zahlen < 5
27
11
17
29
15
# Zahlen >= 5
40
36
```

Bild 5.31
Gespaltene Ausgabe der Zufallszahlen

Auch komplexe Objekte kann man so mit Where() im Modus 'Split' in Teilmengen aufteilen:

```
# Teile die Dienste in zwei Teilmengen
$Running,$Stopped = (Get-Service a*).Where({$_ .Status -eq 'Running'}, 'Split')
$Running
$Stopped
```

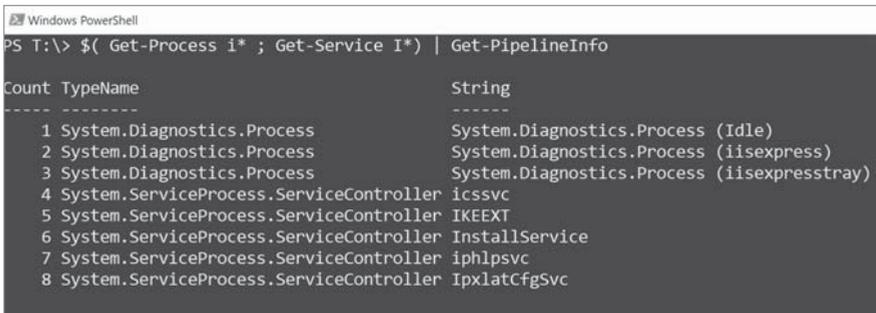
■ 5.14 Zusammenfassung von Pipeline-Inhalten

Die Menge der Objekte in der Pipeline kann heterogen sein, d. h. verschiedenen .NET-Klassen angehören. Dies ist zum Beispiel automatisch der Fall, wenn man `Get-ChildItem` im Dateisystem ausführt: Die Ergebnismenge enthält sowohl `FileInfo`- als auch `DirectoryInfo`-Objekte.

Man kann auch zwei Befehle, die beide Objekte in die Pipeline senden, zusammenfassen, so dass der Inhalt in einer Pipeline wie folgt aussieht:

```
$( Get-Process ; Get-Service )
```

Dies ist aber nur sinnvoll, wenn die nachfolgenden Befehle in der Pipeline korrekt mit heterogenen Pipeline-Inhalten umgehen können. Die Standardausgabe der PowerShell kann dies. In anderen Fällen bedingt der Typ des ersten Objekts in der Pipeline die Art der Weiterverarbeitung (z. B. bei `Export-Csv`).



```

Windows PowerShell
PS T:\> $( Get-Process i* ; Get-Service i* ) | Get-PipelineInfo

Count TypeName                               String
-----
1 System.Diagnostics.Process                 System.Diagnostics.Process (Idle)
2 System.Diagnostics.Process                 System.Diagnostics.Process (iisexpress)
3 System.Diagnostics.Process                 System.Diagnostics.Process (iisexpressray)
4 System.ServiceProcess.ServiceController   icssvc
5 System.ServiceProcess.ServiceController   IKEEXT
6 System.ServiceProcess.ServiceController   InstallService
7 System.ServiceProcess.ServiceController   iphlpvc
8 System.ServiceProcess.ServiceController   IpXlatCfgSvc
  
```

Bild 5.32 Anwendung von `Get-PipelineInfo` auf eine heterogene Pipeline

■ 5.15 „Kastrierung“ von Objekten in der Pipeline

Die Analyse des Pipeline-Inhalts zeigt, dass es oftmals sehr viele Mitglieder in den Objekten in der Pipeline gibt. In der Regel braucht man aber nur wenige. Nicht nur aus Gründen der Leistung und Speicherschonung, sondern auch in Bezug auf die Übersichtlichkeit lohnt es sich, die Objekte in der Pipeline hinsichtlich ihrer Datenmenge zu beschränken.

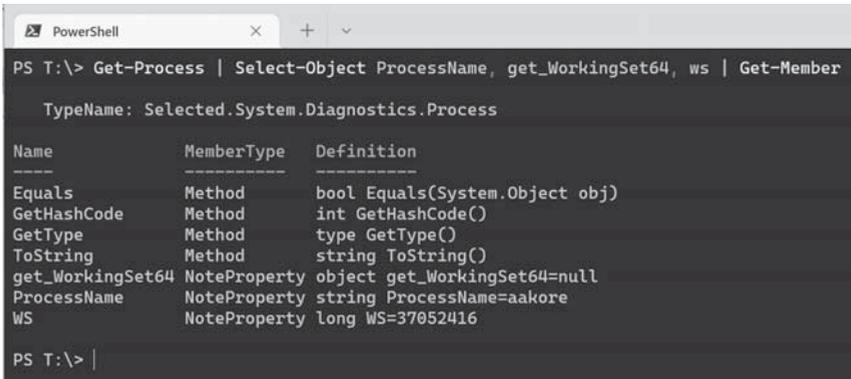
Mit dem Befehl `Select-Object` (Alias: `Select`) kann ein Objekt in der Pipeline „kastriert“ werden, d. h., (fast) alle Mitglieder des Objekts werden aus der Pipeline entfernt, mit Ausnahme der hinter `Select-Object` genannten Mitglieder.

Beispiel:

```
Get-Process | Select-Object processname, get_minworkingset, ws | Get-Member
```

lässt von den `Process`-Objekten in der Pipeline nur die Mitglieder `processname` (Eigenschaft), `get_minworkingset` (Methode) und `workingset` (Alias) übrig (siehe folgende Abbildung). Wie das Bild zeigt, ist das „Kastrieren“ mit zwei Wermutstropfen verbunden:

- Get-Member zeigt nicht mehr den tatsächlichen Klassennamen an, sondern PSCustomObject, eine universelle Klasse der PowerShell.
- Alle Mitglieder sind zu Notizeigenschaften degradiert.



```

PS T:\> Get-Process | Select-Object ProcessName, get_WorkingSet64, ws | Get-Member

TypeName: Selected.System.Diagnostics.Process

Name           MemberType      Definition
-----
Equals         Method          bool Equals(System.Object obj)
GetHashCode    Method          int GetHashCode()
GetType        Method          type GetType()
ToString       Method          string ToString()
get_WorkingSet64 NoteProperty    object get_WorkingSet64=null
ProcessName    NoteProperty    string ProcessName=aakore
WS             NoteProperty    long WS=37052416

PS T:\> |

```

Bild 5.33 Wirkung der Anwendung von Select-Object



TIPP: Mit dem Parameter `-exclude` kann man in `Select-Object` auch Mitglieder einzeln ausschließen.

Dass es neben den drei gewünschten Mitgliedern noch vier weitere in der Liste gibt, ist auch einfach erklärbar: Jedes, wirklich jedes .NET-Objekt hat diese vier Methoden, weil diese von der Basisklasse `System.Object` an jede .NET-Klasse vererbt und damit an jedes .NET-Objekt weitergegeben werden.

■ 5.16 Sortieren

Mit `Sort-Object` (Alias `Sort`) sortiert man die Objekte in der Pipeline nach den anzugebenden Eigenschaften. Die Standardsortierrichtung ist aufsteigend. Mit dem Parameter `-descending` (kurz: `-desc`) legt man die absteigende Sortierung fest.

Der folgende Befehl sortiert die Prozesse absteigend nach ihrem Speicherverbrauch:

```
Get-Process | Sort-Object workingset64 -desc
```

Mit Komma getrennt kann man mehrere Eigenschaften aufführen, nach denen sortiert werden soll. In folgendem Beispiel werden die Systemdienste erst nach Status und innerhalb eines Status dann nach Displayname sortiert.

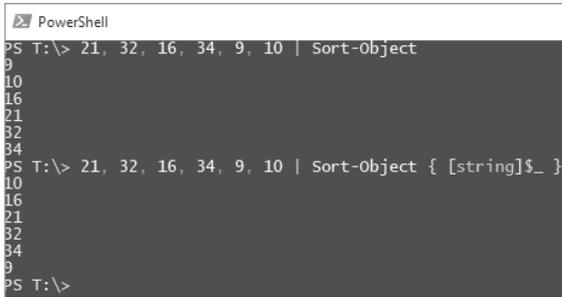
```
Get-Service | Sort-Object Status, Displayname
```

Auch Listen elementarer Datentypen lassen sich sortieren. Hier muss man keine Eigenschaft angeben, nach der man sortieren will:

```
21, 32, 16, 34, 9, 10 | Sort-Object
```

Möchte man diese Zahlen nicht numerisch, sondern alphabetisch sortieren, dann gibt man als Parameter einen Ausdruck an, der eine Typkonvertierung mit einem Typbezeichner (Details zu Typkonvertierungen erfahren Sie im Kapitel 7 „PowerShell-Skriptsprache“) enthält:

```
21, 32, 16, 34, 9, 10 | Sort-Object { [string]$_ }
```



```
PowerShell
PS T:\> 21, 32, 16, 34, 9, 10 | Sort-Object
9
10
16
21
32
34
PS T:\> 21, 32, 16, 34, 9, 10 | Sort-Object { [string]$_ }
10
16
21
32
34
9
PS T:\>
```

Bild 5.34
Numerische versus alphabetische
Sortierung von sechs Zahlen

■ 5.17 Duplikate entfernen

Sowohl `Select-Object -Unique` als auch `Get-Unique` entfernen Duplikate aus einer Liste.
Beispiel

```
1,5,7,8,5,7 | Select-Object -Unique
```

liefert als Ergebnis eine Pipeline mit vier Zahlen: 1,5,7 und 8.



ACHTUNG: Bei `Get-Unique` muss die Liste vorher sortiert sein!

Richtig ist daher:

```
1,5,7,8,5,7 | Sort-Object | Get-Unique
```

Falsch wäre:

```
1,5,7,8,5,7 | Get-Unique
```

Beide Commandlets arbeiten nicht nur auf elementaren Datentypen wie Zahlen und Zeichenketten, sondern auch auf komplexen Objekten, z. B.

```
(Get-process | Select-Object -Unique).Count
(Get-process | sort-object | get-unique).Count
```

```

pwsh
PS X:\> 1,5,7,8,5,7 | Select-Object -Unique
1
5
7
8
PS X:\> (Get-process ).Count
267
PS X:\> (Get-process | sort-object | get-unique).Count
101
PS X:\> 1,5,7,8,5,7 | Get-Unique
1
5
7
8
5
7
PS X:\> 1,5,7,8,5,7 | Sort-Object | Get-Unique
1
5
7
8
PS X:\> (Get-process | get-unique).Count
101
PS X:\> (Get-process | sort-object | get-unique).Count
101
PS X:\> _

```

Bild 5.35

Einsatz von Get-Unique und Select-Object -unique

Praxislösung: Microsoft-Office-Wörterbücher zusammenfassen

Wer auf mehreren Rechnern arbeitet und kein Roaming-Profil nutzen kann oder will, kennt das Problem: Auf jedem PC gibt es ein eigenes benutzerdefiniertes Wörterbuch für Microsoft Word, Outlook etc. (.dic-Datei mit Namen *benutzer.dic* bzw. *custom.dic*). .dic-Dateien sind einfache ASCII-Dateien und man kann natürlich mit jedem beliebigen Texteditor oder einem Merge-Werkzeug die Wörterbücher zusammenführen. Ganz elegant ist die Zusammenführung aber mit einem PowerShell-Einzeiler möglich. Der Befehl geht davon aus, dass sich im Ordner d:\Woerterbuecher mehrere .dic-Dateien befinden. Die Ausgabe ist ein konsolidiertes Wörterbuch *MeinWoerterbuch.dic*. Doppelte Einträge werden natürlich mit Get-Unique eliminiert.

```
Dir "X:\Woerterbuecher" -Filter *.dic | Get-Content | Sort-Object | Get-Unique | Set-Content "X:\Woerterbuecher\MeinWoerterbuch.dic"
```

■ 5.18 Gruppierung

Mit Group-Object (Alias: Group) kann man Objekte in der Pipeline nach Eigenschaften gruppieren.

Mit dem folgenden Befehl ermittelt man, wie viele Systemdienste laufen und wie viele gestoppt sind:

```
Get-Service | Group-Object status
```

Dabei liefert das Commandlet drei Spalten (siehe nächste Abbildung): Count, Name und Group (mit den Elementen in der Gruppe). Über die Eigenschaft Group kann man dann die Gruppenmitglieder abrufen, z.B. die Mitglieder der ersten Gruppe (Zählung beginnt bei 0, runde Klammern nicht vergessen):

```
(Get-Service | Group-Object status)[0].Group
```

Braucht man die Gruppenmitglieder nicht, verwendet man als Zusatz `-NoElement` (das spart etwas Speicherplatz, was aber nur bei großen Ergebnismengen relevant ist):

```
Get-Service | Group-Object status -NoElement
```

Ein weiteres Beispiel gruppiert die Dateien im `System32`-Verzeichnis nach Dateierweiterung und sortiert die Gruppierung dann absteigend nach Anzahl der Dateien in jeder Gruppe.

```
Get-ChildItem c:\windows\system32 | Group-Object extension |
Sort-Object count -desc
```

```
PS T:\> Get-Service | Group-Object status
-----
Count Name                               Group
-----
124 Running                               {AdobeARMSvc, AMD External Events Utility, AntiVirusKit Client, AppHostSvc...}
143 Stopped                               {AJRouter, ALG, AppIDSvc, AppMgmt...}

PS T:\> Get-Service | Group-Object status -NoElement
-----
Count Name
-----
124 Running
143 Stopped

PS T:\> Get-ChildItem c:\windows\system32 | Group-Object extension | Sort-Object count -desc
-----
Count Name                               Group
-----
3420 .dll                                {aadauthelper.dll, aadcloudap.dll, aadjcsp.dll, aadtb.dll...}
671 .exe                                {acu.exe, AgentService.exe, aitstatic.exe, alg.exe...}
138                                     {0409, 1029, 1033, 1036...}
120 .NLS                                {C_037.NLS, C_10000.NLS, C_10001.NLS, C_10002.NLS...}
42 .msc                                 {adsiedit.msc, azman.msc, certlm.msc, certmgr.msc...}
30 .dat                                 {amde3la.dat, amdcdxx.dat, aticdxx.dat, ativce02.dat...}
18 .cpl                                 {appwiz.cpl, bthprops.cpl, desk.cpl, Firewall.cpl...}
17 .png                                 {@AudioToastIcon.png, @BackgroundAccessToastIcon.png, @bitlockertoastimage.png, @edp...}
15 .tlb                                 {activeds.tlb, amcompat.tlb, mqa.tlb, mqa10.tlb...}
15 .ax                                  {bdaplgln.ax, g7llcodc.ax, ksproxy.ax, kstvtune.ax...}
14 .mof                                 {hypervisor.mof, msmpub.mof, msmtcr.mof, msmtcrRemove.mof...}
13 .xml                                 {AppDatabase.xml, AppxProvisioning.xml, DefaultParameters.xml, LServer_PKConfig.xml...}
13 .rs                                  {cero.rs, cob-au.rs, csrr.rs, djctq.rs...}
8 .uce                                 {bopomofo.uce, gb2312.uce, ideograf.uce, kanji_1.uce...}
7 .bin                                 {AverageRoom.bin, DefaultHrtfs.bin, edgehtmlpluginpolicy.bin, LargeRoom.bin...}
6 .scr                                 {Bubbles.scr, Mystify.scr, PhotoSaver.scr, Ribbons.scr...}
6 .ocx                                 {dmview.ocx, hhctrl.ocx, msdxm.ocx, sysmon.ocx...}
6 .acm                                 {imaadp32.acm, l3codeca.acm, l3codecp.acm, msadp32.acm...}
5 .xsl                                 {dfsHealthReport.xsl, dfsrPropagationReport.xsl, EventViewer_EventDetails.xsl, WsmP...}
5 .com                                 {chcp.com, format.com, mode.com, more.com...}
5 .config                             {AppVStreamingUX.exe.config, ClusterUpdateUI.exe.config, DfsMgmt.dll.config, dsac.ex...}
4 .tsp                                 {hidphone.tsp, kmddsp.tsp, remotesp.tsp, unimdm.tsp}
```

Bild 5.36 Einsatz von Group-Object



TIPP: Wenn es nur darum geht, die Gruppen zu ermitteln und nicht die Häufigkeit der Gruppenelemente, dann kann man auch `Select-Object` mit dem Parameter `-unique` zum Gruppieren einsetzen:

```
Get-ChildItem | Select-Object extension -Unique
```



TIPP: Man kann bei `Group-Object` auch einen Ausdruck angeben, der wahr oder falsch liefert, und dadurch zwei Gruppen bilden.

**BEISPIEL:**

```
Get-Childitem c:\Windows | Where { !$_.PsIsContainer } |
Group-Object { $_.Length -gt 1MB}
```

teilt alle Dateien im aktuellen Verzeichnis in zwei Gruppen ein: solche, die größer als 1 MByte sind, und solche, die es nicht sind (Verzeichnisse werden bereits vorher ausgeschlossen, auch wenn dies nicht erforderlich wäre, da sie die Größe 0 besitzen).

```
PS C:\Users\hs.ITU> Get-Childitem c:\Windows | Where { !$_.PsIsContainer } | Group-Object { $_.Length -gt 1MB}
Count Name Group
-----
46 False <Ascd.tmp.ini, hfsvc.exe, bootstat.dat, DtcInstall.log...>
2 True <explorer.exe, WindowsUpdate.log>
```

Bild 5.37 Ergebnis des obigen Befehls (Zahlen können in Abhängigkeit vom Betriebssystem abweichen)

Praxislösung 1

Es sollen in einer Menge von Zeichenketten (hier: Feldnamen für Work Items in Azure DevOps) Duplikate ermittelt werden. Der eingebettete Here-String wird zunächst mit dem Split-Operator zeilenweise in eine Menge von Zeichenketten aufgespalten. Danach wird diese Menge mit Group-Objekt gruppiert. Im Ergebnis findet man die doppelten Zeichenketten, indem man prüft, bei welchen Elementen die Eigenschaft count größer als eins ist.

Listing 5.4 [Finde doppelte Zeichenketten.ps1]

```
# Finde doppelte Zeichenketten
# Eingabemenge: Zeichenketten (eingebettet als "Here-String" oder aus einer Datei)
# Ausgabe: Liste der doppelt vorkommenden Zeichenketten

$eingabe = @"
Microsoft.VSTS.Build.FoundIn
Microsoft.VSTS.Build.IntegrationBuild
Microsoft.VSTS.CMMI.ActualAttendee1
Microsoft.VSTS.CMMI.ActualAttendee2
Microsoft.VSTS.CMMI.ActualAttendee3
Microsoft.VSTS.CMMI.ActualAttendee4
Microsoft.VSTS.CMMI.ActualAttendee5
Microsoft.VSTS.CMMI.ActualAttendee6
Microsoft.VSTS.CMMI.ActualAttendee7
Microsoft.VSTS.CMMI.ActualAttendee8
Microsoft.VSTS.CMMI.Analysis
Microsoft.VSTS.CMMI.Blocked
Microsoft.VSTS.CMMI.CalledBy
Microsoft.VSTS.CMMI.CalledDate
Microsoft.VSTS.CMMI.Comments
Microsoft.VSTS.CMMI.Committed
Microsoft.VSTS.CMMI.ContingencyPlan
Microsoft.VSTS.CMMI.CorrectiveActionActualResolution
Microsoft.VSTS.CMMI.CorrectiveActionPlan
Microsoft.VSTS.CMMI.Escalate
Microsoft.VSTS.CMMI.FoundInEnvironment
Microsoft.VSTS.CMMI.HowFound
Microsoft.VSTS.CMMI.ImpactAssessmentHtml
```

Microsoft.VSTS.CMMI.ImpactOnArchitecture
Microsoft.VSTS.CMMI.ImpactOnDevelopment
Microsoft.VSTS.CMMI.ImpactOnTechnicalPublications
Microsoft.VSTS.CMMI.ImpactOnTest
Microsoft.VSTS.CMMI.ImpactOnUserExperience
Microsoft.VSTS.CMMI.Justification
Microsoft.VSTS.CMMI.MeetingType
Microsoft.VSTS.CMMI.Minutes
Microsoft.VSTS.CMMI.MitigationPlan
Microsoft.VSTS.CMMI.MitigationTriggers
Microsoft.VSTS.CMMI.OptionalAttendee1
Microsoft.VSTS.CMMI.OptionalAttendee2
Microsoft.VSTS.CMMI.OptionalAttendee3
Microsoft.VSTS.CMMI.OptionalAttendee4
Microsoft.VSTS.CMMI.OptionalAttendee5
Microsoft.VSTS.CMMI.OptionalAttendee6
Microsoft.VSTS.CMMI.OptionalAttendee7
Microsoft.VSTS.CMMI.OptionalAttendee8
Microsoft.VSTS.CMMI.Probability
Microsoft.VSTS.CMMI.ProposedFix
Microsoft.VSTS.CMMI.Purpose
Microsoft.VSTS.CMMI.RequiredAttendee1
Microsoft.VSTS.CMMI.RequiredAttendee2
Microsoft.VSTS.CMMI.RequiredAttendee3
Microsoft.VSTS.CMMI.RequiredAttendee4
Microsoft.VSTS.CMMI.RequiredAttendee5
Microsoft.VSTS.CMMI.RequiredAttendee6
Microsoft.VSTS.CMMI.RequiredAttendee7
Microsoft.VSTS.CMMI.RequiredAttendee8
Microsoft.VSTS.CMMI.RequirementType
Microsoft.VSTS.CMMI.RequiresReview
Microsoft.VSTS.CMMI.RequiresTest
Microsoft.VSTS.CMMI.RootCause
Microsoft.VSTS.CMMI.SubjectMatterExpert1
Microsoft.VSTS.CMMI.SubjectMatterExpert2
Microsoft.VSTS.CMMI.SubjectMatterExpert3
Microsoft.VSTS.CMMI.Symptom
Microsoft.VSTS.CMMI.TargetResolveDate
Microsoft.VSTS.CMMI.TaskType
Microsoft.VSTS.CMMI.UserAcceptanceTest
Microsoft.VSTS.CodeReview.AcceptedBy
Microsoft.VSTS.CodeReview.AcceptedDate
Microsoft.VSTS.CodeReview.ClosedStatus
Microsoft.VSTS.CodeReview.ClosedStatusCode
Microsoft.VSTS.CodeReview.ClosedStatusCode
Microsoft.VSTS.CodeReview.ClosingComment
Microsoft.VSTS.CodeReview.Context
Microsoft.VSTS.CodeReview.ContextCode
Microsoft.VSTS.CodeReview.ContextOwner
Microsoft.VSTS.CodeReview.ContextType
Microsoft.VSTS.Common.AcceptanceCriteria
Microsoft.VSTS.Common.ActivatedBy
Microsoft.VSTS.Common.ActivatedDate
Microsoft.VSTS.Common.Activity
Microsoft.VSTS.Common.BusinessValue
Microsoft.VSTS.Common.ClosedBy
Microsoft.VSTS.Common.ClosedDate
Microsoft.VSTS.Common.Discipline
Microsoft.VSTS.Common.Issue

Microsoft.VSTS.Common.Priority
Microsoft.VSTS.Common.Rating
Microsoft.VSTS.Common.Resolution
Microsoft.VSTS.Common.ResolvedBy
Microsoft.VSTS.Common.ResolvedDate
Microsoft.VSTS.Common.ResolvedReason
Microsoft.VSTS.Common.ReviewedBy
Microsoft.VSTS.Common.Risk
Microsoft.VSTS.Common.Severity
Microsoft.VSTS.Common.StackRank
Microsoft.VSTS.Common.StateChangeDate
Microsoft.VSTS.Common.StateCode
Microsoft.VSTS.Common.TimeCriticality
Microsoft.VSTS.Common.Triage
Microsoft.VSTS.Common.ValueArea
Microsoft.VSTS.Feedback.ApplicationLaunchInstructions
Microsoft.VSTS.Feedback.ApplicationStartInformation
Microsoft.VSTS.Feedback.ApplicationType
Microsoft.VSTS.Scheduling.CompletedWork
Microsoft.VSTS.Scheduling.DueDate
Microsoft.VSTS.Scheduling.Effort
Microsoft.VSTS.Scheduling.FinishDate
Microsoft.VSTS.Scheduling.OriginalEstimate
Microsoft.VSTS.Scheduling.RemainingWork
Microsoft.VSTS.Scheduling.Size
Microsoft.VSTS.Scheduling.StartDate
Microsoft.VSTS.Scheduling.StoryPoints
Microsoft.VSTS.Scheduling.TargetDate
Microsoft.VSTS.TCM.AutomatedTestId
Microsoft.VSTS.TCM.AutomatedTestName
Microsoft.VSTS.TCM.AutomatedTestStorage
Microsoft.VSTS.TCM.AutomatedTestType
Microsoft.VSTS.TCM.AutomationStatus
Microsoft.VSTS.TCM.LocalDataSource
Microsoft.VSTS.TCM.Parameters
Microsoft.VSTS.TCM.QueryText
Microsoft.VSTS.TCM.ReproSteps
Microsoft.VSTS.TCM.Steps
Microsoft.VSTS.TCM.SystemInfo
Microsoft.VSTS.TCM.TestSuiteAudit
Microsoft.VSTS.TCM.TestSuiteType
Microsoft.VSTS.TCM.TestSuiteTypeId
System.AreaId
System.AreaPath
System.AssignedTo
System.AttachedFileCount
System.AuthorizedAs
System.AuthorizedDate
System.BoardColumn
System.BoardColumnDone
System.BoardLane
System.ChangedBy
System.ChangedDate
System.CommentCount
System.CreatedBy
System.CreatedDate
System.Description
System.ExternalLinkCount
System.History

```

System.HyperLinkCount
System.Id
System.IterationId
System.IterationPath
System.NodeName
System.Reason
System.RelatedLinkCount
System.RemoteLinkCount
System.Rev
System.RevisedDate
System.State
System.Tags
System.Tags
System.TeamProject
System.Title
System.Watermark
System.WorkItemType
"@

# Alternativ: Einlesen einer Datei
# $eingabe = get-content "eingabedatei.txt"

# Der eingebettete Here-String wird zunächst mit dem Split-Operator zeilenweise in
eine Menge von Zeichenketten aufgespalten.
$gespaltet = $eingabe -split "`n" |Sort-Object
# Danach wird diese Menge mit Group-Objekt gruppiert.
$gruppiert = $gespaltet | Group-Object

$anz = ($gespaltet).Count
$anzGruppiert = ($gruppiert).Count
$Duplikate = $gruppiert | where count -gt 1

if ($Duplikate.Count -eq 0)
{
    Write-Host "$Anz Elemente. Keine Duplikate!" -ForegroundColor Green
}
else
{
    Write-Host "$($Duplikate.Count) Zeichenketten kommen mehrfach vor /
$anzGruppiert verschiedenen Zeichenketten in $anz Zeilen:" -ForegroundColor red
    $Duplikate | Ft Name, Count
}

    $Duplikate | Ft Name, Count
}

```

Praxislösung 2

Wenn man sich die Elemente der einzelnen Gruppen liefern lässt, so kann man diese weiterverwenden, indem man über die Eigenschaft `group` mit `Foreach-Object` iteriert.

Beispiel: Ermittle aus dem Verzeichnis `System32` alle Dateien, die mit dem Buchstaben „b“ beginnen. Beschränke die Menge auf diejenigen Dateien, die größer als 40 000 Byte sind, und gruppier die Ergebnismenge nach Dateierweiterungen. Sortiere die Gruppen nach der Anzahl der Einträge absteigend und beschränke die Menge auf das oberste Element. Gib für alle Mitglieder dieser Gruppe die Attribute `Name` und `Length` aus und passe die Spaltenbreite automatisch an.

■ 5.20 Berechnungen

Measure-Object (Alias: measure) führt verschiedene Berechnungen (Anzahl, Durchschnitt, Summe, Minimum, Maximum) für Objekte in der Pipeline aus. Dabei sollte man die Eigenschaft nennen, über welche die Berechnung ausgeführt werden soll. Sonst wird die erste Eigenschaft verwendet, die aber häufig ein Text ist, den man nicht mathematisch verarbeiten kann.

Measure-Object liefert im Standard nur die Anzahl. Mit den Parametern -sum, -min, -max und -average muss man weitere Berechnungen explizit anstoßen.

Beispiel: Informationen über die Dateien in `c:\Windows`

```
Get-ChildItem c:\windows | Measure-Object -Property length -min -max -average -sum
```

```
PS C:\Windows\System32\WindowsPowerShell\v1.0> Get-Childitem c:\Windows\system32 | measure-object -Property length -min
-max -average -sum
Count       : 2590
Average     : 465515.188030888
Sum         : 1205604337
Maximum     : 26575296
Minimum     : 35
Property    : length
PS C:\Windows\System32\WindowsPowerShell\v1.0>
```

Bild 5.39 Beispiel für den Einsatz von Measure-Object

■ 5.21 Zwischenschritte in der Pipeline mit Variablen

Ein Befehl mit Pipeline kann beliebig lang und damit auch beliebig komplex werden. Wenn der Befehl unübersichtlich wird oder man Zwischenschritte genauer betrachten möchte, bietet es sich an, den Inhalt der Pipeline zwischenspeichern. Die PowerShell ermöglicht es, den Inhalt der Pipeline in Variablen abzulegen. Variablen werden durch ein vorangestelltes Dollarzeichen [\$] gekennzeichnet. Anstelle von

```
Get-Process | Where-Object {$_.name -eq "iexplore"} | Foreach-Object { $_.ws }
```

kann man die folgenden Befehle nacheinander in getrennte Zeilen eingeben:

```
$x = Get-Process
$y = $x | Where-Object {$_.name -eq "iexplore"}
$y | Foreach-Object { $_.ws }
```

Das Ergebnis ist in beiden Fällen gleich.

Der Zugriff auf Variablen, die keinen Inhalt haben, führt so lange nicht zum Fehler, wie man später in der Pipeline keine Commandlets verwendet, die unbedingt Objekte in der Pipeline erwarten.

```

Administrator: Windows PowerShell
PS T:\> $x
PS T:\> $x | get-member
get-member : Sie müssen ein Objekt für das Cmdlet "Get-Member" angeben.
In Zeile:1 Zeichen:6
+ $x | get-member
+ ~~~~~
+ CategoryInfo          : CloseError: (:) [Get-Member], InvalidOperationException
+ FullyQualifiedErrorId : NoObjectInGetMember,Microsoft.PowerShell.Commands.GetMemberCommand
PS T:\> _

```

Bild 5.40 Zugriff auf Variablen ohne Inhalt



ACHTUNG: Wenn ein Pipeline-Befehl keinen Inhalt liefert, dann erhält die Variable den Wert `$null`, der für „kein Wert“ steht.

Beispiel:

```
$x = Get-Service x*
```

Die Ausgabe für `$null` ist nichts.

■ 5.22 Verzweigungen in der Pipeline

Manchmal möchte man innerhalb einer Pipeline das Ergebnis nicht nur in der Pipeline weiterreichen, sondern auch in einer Variablen oder im Dateisystem zwischenspeichern. PowerShell bietet dafür verschiedene Möglichkeiten.



TIPP: Verzweigungen in der Pipeline lassen sich ganz einfach abbilden, indem man die Zwischenschritte in verschiedenen Variablen ablegt, auf die man später wieder zugreifen kann. Die in diesem Unterkapitel gezeigten Techniken sind für Leute gedacht, die unbedingt möglichst viel in einem einzigen Pipeline-Befehl unterbringen wollen.

Tee-Object

Der Verzweigung innerhalb der Pipeline dient das Commandlet `Tee-Object`, wobei hier das „Tee“ für „verzweigen“ steht. `Tee-Object` reicht den Inhalt der Pipeline unverändert zum nächsten Commandlet weiter, bietet aber an, den Inhalt der Pipeline wahlweise zusätzlich in einer Variablen oder im Dateisystem abzulegen.

Der folgende Pipeline-Befehl verwendet `Tee-Object` gleich zweimal für beide Anwendungsfälle:

```
Get-Service | Tee-Object -var a | Where-Object { $_.Status -eq "Running" } | select
name | Tee-Object -filepath x:\dienste.txt | ft name
```

Die erste Verwendung von Tee-Object speichert die Liste der Dienste-Objekte in der Variablen \$a und gibt die Objekte aber gleichzeitig weiter in die Pipeline.

Die zweite Verwendung speichert die Liste der laufenden Dienste in der Textdatei g:\dienste.txt und gibt sie zusätzlich an die Standardausgabe aus.

Nach der Ausführung des Befehls steht in der Variablen \$a eine Liste aller Dienste und in der Textdatei *dienste.txt* eine Liste der laufenden Dienste.



ACHTUNG: Bitte beachten Sie, dass man bei Tee-Object beim Parameter `-variable` den Namen der Variablen ohne den üblichen Variablenkennzeichner „\$“ angeben muss.

Parameter -OutVariable

Alternativ zum Commandlet Tee-Object kann man den allgemeinen Parameter `-OutVariable` (kurz: `-ov`) einsetzen, der das Ergebnis eines Commandlets in einer Variable ablegt und dennoch das Ergebnis in der Pipeline weiterreicht. Das Beispiel aus dem vorherigen Unterkapitel kann man so umformulieren:

```
Get-Service -OutVariable a | Where-Object { $_.Status -eq "Running" } | select name |
Set-Content x:\dienste.txt -PassThru | ft name
```

Anders als Tee-Object kann `-OutVariable` nichts direkt in einer Datei speichern. Zum Speichern kommt daher hier `Set-Content` zum Einsatz mit `-PassThru`, was ein zusätzliches Durchleiten der Ergebnisse bewirkt.



ACHTUNG: Nach `-OutVariable` ist von der Variablen nur der Name anzugeben. Das Dollarzeichen muss weggelassen werden.

Parameter -PipelineVariable

Der mit PowerShell-Version 4.0 eingeführte allgemeine Parameter `-PipelineVariable` (kurz: `-pv`) sorgt dafür, dass das jeweils aktuelle Objekt nicht nur in der Pipeline weitergereicht wird, sondern zusätzlich auch in einer Variablen abgelegt wird. Dies ist immer dann sinnvoll, wenn die Pipeline ein Objekt in seiner Struktur verändert (z. B. `Select-Object`), man aber später noch auf den früheren Zustand zugreifen will. Nach `-PipelineVariable` ist von der Variablen nur der Name anzugeben. Das Dollarzeichen muss weggelassen werden.

Beispiel 1

Das folgende Beispiel setzt dies ein, um am Ende eine Liste von Ausgaben aus zwei verschiedenen Objekten zu liefern: den Namen und das Workingset eines Prozesses von `Get-Process` und den Namen und den zugehörigen Security Identifier des Benutzers, unter dem der Prozess läuft. Die Pipeline beginnt mit dem Holen der laufenden Prozesse unter Einbeziehung der Benutzeridentität, die in der Form „Domäne\Benutzername“ geliefert wird. Dabei wird

das aktuelle Process-Objekt mit `-pv` auch in der Variablen `$p` abgelegt. Im zweiten Schritt wird für den Benutzernamen das zugehörige WMI-Objekt `Win32_User` geholt. Im dritten Pipeline-Schritt werden dann zuerst die zwei Informationen aus dem Process-Objekt ausgegeben (das sich in `$p` befindet) sowie die Informationen aus dem `Win32_UserAccount`-Objekt, die sich nun in der Pipeline befinden (`$_`).

```
Get-Process -IncludeUserName -pv p | % { Get-WmiObject Win32_UserAccount -filter
"name='${($_.username -split "\\")[1]}'" } | % { $p.name + ";" + $p.ws + ":" +
$_ .Name + ";" + $_.SID }
```



ACHTUNG: Der Parameter `-PipelineVariable` funktioniert nicht wie gewünscht, wenn Commandlets in der Pipeline sind, die die Ergebnisse puffern (z. B. `Sort-Object`, `Group-Object`), da der Parameter `-PipelineVariable` sich ja immer nur auf das aktuelle Objekt bezieht, was in diesen Fällen also immer das letzte Objekt ist.

Beispiel 2

Der folgende Einzeiler listet alle 64516-IP-Adressen zwischen 192.168.0.0 und 192.168.254.254 auf.

```
1..254 | Foreach-Object -PipelineVariable x { $_ } | Foreach-Object { 1..254 } |
foreach-Object { "192.168.$x.$_" }
```

5.23 Vergleiche zwischen Objekten

Mit `Compare-Object` kann man den Inhalt von zwei Pipelines vergleichen. Mit der folgenden Befehlsfolge werden alle zwischenzeitlich neu gestarteten Prozesse ausgegeben:

```
$ProzesseVorher = Get-Process
# Hier einen Prozess starten
$ProzesseNachher = Get-Process
Compare-Object $ProzesseVorher $ProzesseNachher
```

```
pwsh
PS X:\> $vorher = Get-Process
PS X:\> notepad
PS X:\> notepad
PS X:\> mmc
PS X:\> $nachher = Get-Process
PS X:\> Compare-Object $vorher $nachher

InputObject                               SideIndicator
-----
System.Diagnostics.Process (mmc)           =>
System.Diagnostics.Process (notepad)      =>
System.Diagnostics.Process (notepad)      =>
PS X:\> _
```

Bild 5.41

Vergleich von zwei Pipelines

■ 5.24 Weitere Praxislösungen

Dieses Kapitel enthält einige Beispiele für die Anwendung von Pipelining und Ausgabebefehlen:

- Beende durch Aufruf der Methode `Kill()` alle Prozesse, die „chrome“ heißen, wobei die Groß-/Kleinschreibung des Prozessnamens irrelevant ist.

```
Get-Process | Where { $_.processname -ieq "chrome" } | foreach { $_.Kill() }
```

oder synonym und kürzer:

```
(Get-Process "chrome").Kill()
```

- Sortiere die Prozesse, die das Wort „chrome“ im Namen tragen, gemäß ihrer CPU-Nutzung und beende den Prozess, der in der aufsteigenden Liste der CPU-Nutzung am weitesten unten steht (also am meisten Rechenleistung verbraucht).

```
Get-Process | Where { $_.processname -ilike "*chrome*" } | Sort-Object -property cpu | Select-Object -last 1 | foreach { $_.Kill() }
```

- Gib die Summe der Speichernutzung aller Prozesse aus.

```
Get-Process | Measure-Object workingset-sum
```

- Gruppier die Einträge im System-Ereignisprotokoll nach Benutzernamen.

```
Get-EventLog -logname system | Group-Object username
```

- Zeige die neuesten zehn Einträge im System-Ereignisprotokoll.

```
Get-EventLog -logname system | Sort-Object timegenerated -desc | Select-Object -first 10
```

- Oder kürzer und schneller (dieses Commandlet besitzt eine eingebaute Filterfunktion):

```
Get-EventLog System -newest 10
```

- Importiere die Textdatei `test.txt`, wobei die Textdatei als eine CSV-Datei mit dem Semikolon als Trennzeichen zu interpretieren ist und die erste Zeile die Spaltennamen enthalten muss. Zeige daraus die Spalten `ID` und `Url`.

```
Import-CSV d:\_work\test.txt -delimiter ";" | Select-Object ID,Url
```

- Ermittle aus dem Verzeichnis `System32` alle Dateien, die mit dem Buchstaben „a“ beginnen. Beschränke die Menge auf diejenigen Dateien, die größer als 40 000 Byte sind, und gruppier die Ergebnismenge nach Dateinamenerweiterungen. Sortiere die gruppierte Menge nach dem Namen der Dateierweiterung.

```
Get-ChildItem c:\windows\system32 -filter a*. * | Where-Object {$_.Length -gt 40000} | Group-Object Extension | Sort-Object name | Format-Table
```

- Ermittle aus dem Verzeichnis System32 alle Dateien, die mit dem Buchstaben „b“ beginnen. Beschränke die Menge auf diejenigen Dateien, die größer als 40 000 Byte sind, und gruppier die Ergebnismenge nach Dateierweiterungen. Sortiere die Gruppen nach der Anzahl der Einträge absteigend und beschränke die Menge auf das oberste Element. Gib für alle Mitglieder dieser Gruppe die Attribute Name und Length aus und passe die Spaltenbreite automatisch an.

```
Get-ChildItem c:\windows\system32 -filter b*. * | Where-Object {$_.Length -gt 40000}  
| Group-Object Extension | Select-Object -first 1 | Select-Object group | foreach  
{$_.group} | Select-Object name,length | Format-Table -autosize
```