


# PowerShell

Skript-Programmierung


← → ↻ <https://docs.microsoft.com/en-us/powershell/>


 | **Docs** | Documentation | Learn | Q&A | Code Samples | Shows | Events


**PowerShell** | [Module Browser](#) | [API Reference](#) | [Utility modules](#) | [DSC](#) | [VS Code Extension](#) | [PowerShell Gallery](#)


## PowerShell Documentation


Official product documentation for PowerShell


 **GET STARTED**  
[Overview](#)

 **DOWNLOAD**  
[Setup and installation](#)

 **HOW-TO GUIDE**  
[Sample scripts](#)

 **DEPLOY**  
[PowerShell Gallery](#)

 **REFERENCE**  
[PowerShell Module Browser](#)

 **ARCHITECTURE**  
[PowerShell on GitHub](#)

- Die in diesem Seminar verwendete Werkzeuge und Frameworks sind frei erhältlich
- Dies ist ein Programmier-Seminar
  - Damit werden die Inhalte durch Übungen vertieft und verinnerlicht
  - Musterbeispiele werden zur Verfügung gestellt
    - GitHub-Repository
- Dokumentation und Ressourcen stehen auch im Internet zur Verfügung

© Integrata AG

Integrata AG  
Zettachring 4  
70567 Stuttgart

**Alle Rechte, einschließlich derjenigen des auszugsweisen Abdrucks,  
der fotomechanischen und elektronischen Wiedergabe vorbehalten.**

Die PowerShell kennenlernen	6
PowerShell-Cmdlets	13
Cmdlets im Detail	23
Programmiergrundlagen	55
Skript-Programmierung	85
PowerShell-Module	105

1

# DIE POWERSHELL KENNENLERNEN

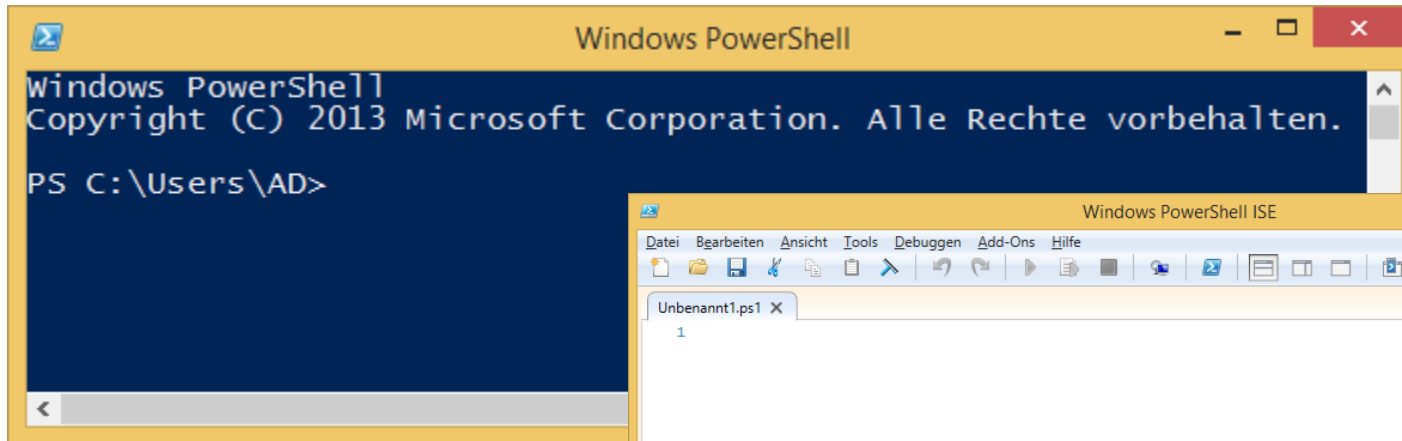
1.1

# EINFÜHRUNG

PowerShell Version	Release Date	Standard Windows Version	Verfügbare Windows Version
PowerShell 1.0	November 2006	Windows Server 2008	Windows XP SP2 / SP3 Windows Server 2003 SP1 / SP2 Windows Server 2003 R2 Windows Vista Windows Vista SP2
PowerShell 2.0	Oktober 2009	Windows 7 Windows Server 2008 R2	Windows XP SP3 Windows Server 2003 SP2 Windows Vista SP1 / SP2 Windows Server 2008 SP1 / SP2
PowerShell 3.0	September 2012	Windows 8 Windows Server 2012	Windows 7 SP1 Windows Server 2008 SP2 Windows Server 2008 R2 SP1
PowerShell 4.0	Oktober 2013	Windows 8.1 Windows Server 2012 R2	Windows 7 SP1 Windows Server 2008 R2 SP1 Windows Server 2012
PowerShell 5.0	April 2014	Windows 10	Windows 8.1 Windows Server 2012 R2

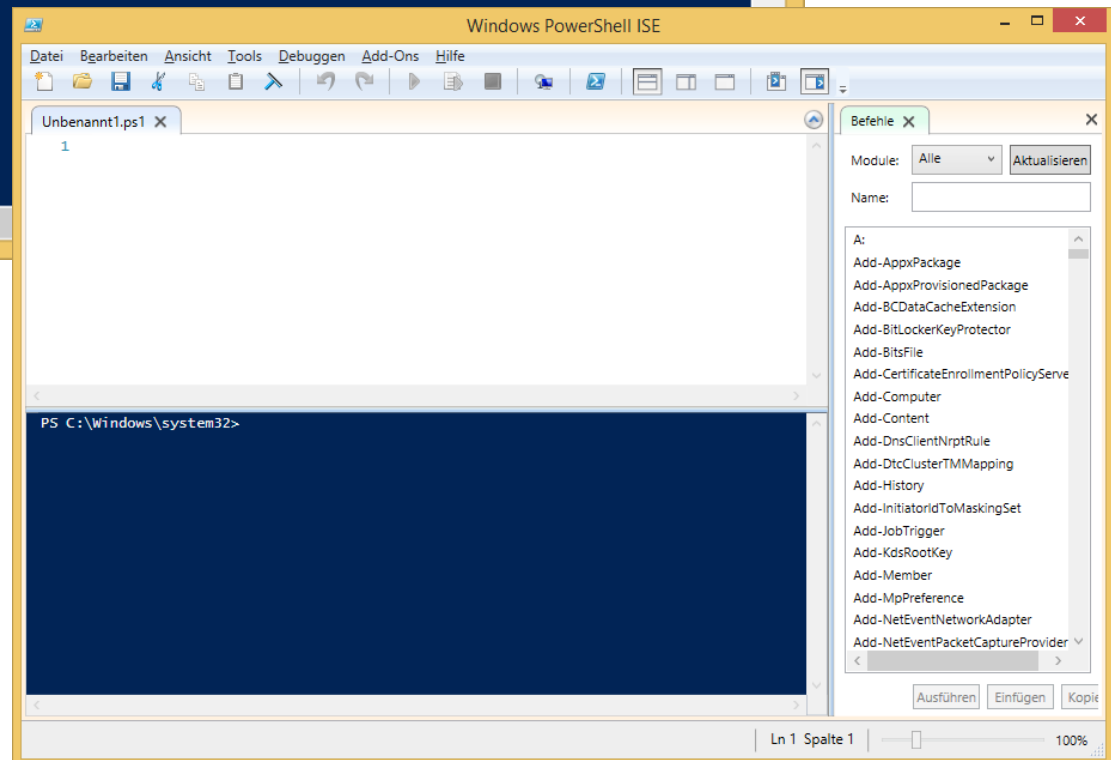


- PowerShell-Konsole





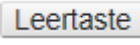
The screenshot shows the Windows PowerShell console window. The title bar reads "Windows PowerShell". The window has a dark blue background with white text. The text inside the console reads: "Windows PowerShell", "Copyright (C) 2013 Microsoft Corporation. Alle Rechte vorbehalten.", and "PS C:\Users\AD>".

- PowerShell ISE



1.2

## **ERSTES ARBEITEN**

- Sowohl die PowerShell-Konsole als auch die PowerShell ISE haben Mechanismen zur Erleichterung Ihrer Eingaben:
  - Die PowerShell-Konsole hilft Ihnen mit der sogenannten Tabulator-Vervollständigung. Wenn Sie einen Teil eines Befehls oder eines Pfads eingeben, können Sie mit der Tabulatortaste  eine Vervollständigung des Befehls oder Pfads auf Basis der bisher erfolgten Eingabe erreichen.
  - Die PowerShell ISE stellt eine weitergehende Hilfe namens IntelliSense zur Verfügung. Grafisch orientierte Menüs helfen Ihnen automatisch bei der Vervollständigung einer Eingabe. Wird ein solches Hilfsmenü gerade nicht angezeigt, können Sie es durch die Tastenkombination   einblenden.

- Setzen Sie die PowerShell-Konsole ein, wenn Sie ...
  - eine schnell startende Konsole für interaktive Eingaben benötigen
  - den Komfort einer Windows-Anwendung nicht benötigen
  - Befehle verwenden, die eine interaktive Eingabe des Benutzers erwarten (Diese Art von Befehlen funktioniert in der PowerShell ISE nicht. Die betreffenden Kommandos können Sie anzeigen, wenn Sie in der PowerShell ISE den Befehl `$psUnsupportedConsoleApplications` eingeben.)
- Setzen Sie die PowerShell ISE ein, wenn Sie ...
  - IntelliSense-Menüs verwenden wollen
  - sich automatisch Syntaxfehler anzeigen lassen wollen
  - den Komfort einer Windows-Anwendung einem Konsolenfenster vorziehen
  - in erster Linie mit der PowerShell programmieren und längeren, fortgeschrittenen Code produzieren
  - bemerken, dass die Eingabe zu langer Zeilen im Konsolenfenster zu unübersichtlich wird
  - die Unterstützung von Unicode-Zeichensätzen (z. B. für Sprachen mit anderen Schriftzeichen) benötigen, was von der Konsole nicht unterstützt wird

2

## **POWERSHELL-CMDLETS**

2.1

## **GRUNDLAGEN**

- Syntax der Cmdlets  
Verb-Substantiv [–Parameter [Wert]]
- Beispiele  
Get-Command  
Get-EventLog

Typ	Beschreibung
N	<b>Benannter Parameter:</b> Name des Parameters <b>muss</b> stets angegeben werden.  <Cmdlet> –Parameter Wert
P(x)	<b>Positionsparameter:</b> Name des Parameters <b>kann</b> angegeben werden. Wird der Parametername nicht angegeben, bezeichnet die Zahl in Klammern die Position, an der der Wert des Parameters stehen muss.  <Cmdlet> [–Parameter] Wert
S	<b>Switch-Parameter:</b> Parameter, der als Ein-/Aus-Schalter fungiert und ohne Wertangabe auskommt  <Cmdlet> –Parameter



Parameter	Typ	Beschreibung
-Verbose	S	Zeigt sehr ausführliche Informationen zu dem Vorgang an, der mit dem Befehl ausgeführt wird
-Debug	S	Zeigt zusätzlich Informationen für Programmierer, wenn Warnungen und Fehler bei dem ausgeführten Vorgang auftreten
-WarningAction	N	Bestimmt das Verhalten des Cmdlets, falls eine Warnung auftritt. Gültige Werte sind: <ul style="list-style-type: none"> <li>✓ SilentlyContinue: Warnung unterdrücken, Ausführung fortsetzen</li> <li>✓ Continue: Warnung ausgeben, Ausführung fortsetzen</li> <li>✓ Inquire: Warnung ausgeben, weitere Ausführung nachfragen</li> <li>✓ Stop: Warnung ausgeben, Ausführung beenden</li> </ul>
-WarningVariable	N	Speichert auftretende Warnmeldungen in der angegebenen Variablen
-ErrorAction	N	Bestimmt das Verhalten des Cmdlets, falls eine Warnung auftritt. Gültige Werte sind: <ul style="list-style-type: none"> <li>✓ SilentlyContinue: Warnung unterdrücken, Ausführung fortsetzen</li> <li>✓ Continue: Warnung ausgeben, Ausführung fortsetzen</li> <li>✓ Inquire: Warnung ausgeben, weitere Ausführung nachfragen</li> <li>✓ Stop: Warnung ausgeben, Ausführung beenden</li> </ul>
-ErrorVariable	N	Speichert auftretende Fehlermeldungen in der angegebenen Variablen
-OutVariable	N	Zeigt die Ausgabeobjekte des Befehls an und speichert sie in der angegebenen Variablen

## ■ Risikominderungsparameter

Parameter	Typ	Beschreibung
-WhatIf	S	<p>Die Anweisung wird nicht ausgeführt. Eine Meldung beschreibt die Auswirkungen, wenn das Cmdlet wirklich ausgeführt wird.</p> <p>Beispiel:</p> <p>Stop-Process -Name PowerShell -WhatIf</p> <p>Mit dem Parameter -WhatIf wird das Cmdlet nicht ausgeführt, sondern folgende Meldung angezeigt:</p> <p>WhatIf: Ausführen des Vorgangs "Stop-Process" für das Ziel "PowerShell (&lt;Prozess-ID&gt;)".</p>
-Confirm	S	<p>Fordert Sie vor der Ausführung jeder Aktion zur Bestätigung auf.</p> <p>Beispiel:</p> <p>Stop-Process -Name PowerShell -Confirm</p> <p>Sie erhalten folgende Ausgabe:</p> <p>Bestätigung</p> <p>Möchten Sie diese Aktion wirklich ausführen?</p> <p>Ausführen des Vorgangs "Stop-Process" für das Ziel "PowerShell (&lt;Prozess-ID&gt;)".</p> <p>[J] Ja [A] Ja, alle [N] Nein [K] Nein, keine</p> <p>[H] Anhalten [?] Hilfe (Standard ist "J"):</p>

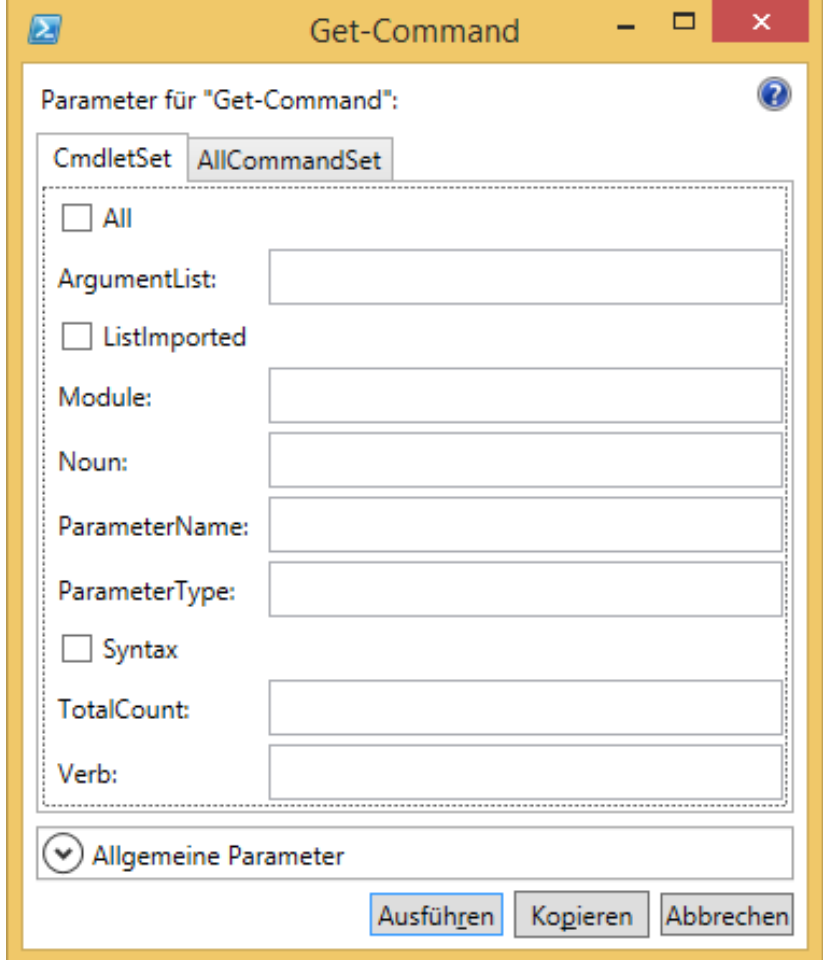
## 2.2

# ERSTE CMDLETS

## ■ Auswahl zur Informationsbeschaffung

Cmdlet	Kurzbeschreibung
Get-ChildItem	Listet den Verzeichnisinhalt eines angegebenen Verzeichnisses auf.
Get-Date	Ruft aktuelle Datums- und Uhrzeitangaben auf
Get-History	Zeigt eine Liste der zuletzt eingegebenen Befehle in der aktuellen PowerShell-Sitzung an
Get-Module	Zeigt die PowerShell-Module an, die importiert wurden bzw. für einen Import bereitstehen
Get-NetIPAddress	Ruft Informationen zur IPv4- und IPv6-Adressierung eines Rechners ab
Get-Process	Zeigt die aufgeführten Prozesse auf einem Rechner an
Get-Service	Ruft die Dienste auf einem Rechner ab
Get-PSDrive	Zeigt die PowerShell-Laufwerke der aktuellen Sitzung an
Get-PSProvider	Fordert Informationen über PowerShell-Provider an

- Befehle finden
  - Get-Command
- Weitere Informationen zu Befehlen
  - Get-Member
- Interaktiver Cmdlet-Generator
  - Show-Command
- Integrierte Hilfe
  - Get-Help



The screenshot shows a Windows-style dialog box titled "Get-Command". It has a yellow title bar with standard window controls. The main content area is titled "Parameter für 'Get-Command':" and contains two tabs: "CmdletSet" and "AllCommandSet". The "AllCommandSet" tab is active. Inside the tab, there is a dashed border containing several input fields and checkboxes. The fields are: "ArgumentList:", "Module:", "Noun:", "ParameterName:", "ParameterType:", "TotalCount:", and "Verb:". The checkboxes are: "All", "ListImported", and "Syntax". Below the dashed border, there is a section titled "Allgemeine Parameter" with a dropdown arrow. At the bottom right of the dialog, there are three buttons: "Ausführen" (highlighted in blue), "Kopieren", and "Abbrechen".

## ■ Beispiele für die Verwendung von Get-Help

Sie möchten ...	
Englische Hilfedateien installieren	Update-Help –UICulture "en-US"
alle PowerShell-Befehle auflisten	Get-Help
Tipps zur Verwendung des Cmdlets Get-Command durch die Anzeige von Beispielen erhalten	Get-Help –Name Get-Command -Examples
eine Liste aller PowerShell-Konzepte sehen, zu denen Hilfetexte zur Verfügung stehen	Get-Help –Name about_*
den Hilfetext zur Beschreibung des Parameters CommandType des Cmdlets Get-Command abrufen	Get-Help –Name Get-Command –Parameter CommandType
in Ihrem Standardbrowser die Onlinehilfe zum Cmdlet Get-Command anzeigen lassen	Get-Help –Name Get-Command -Online
die Hilfe zum PowerShell-Konzept der allgemeinen Parameter in einem eigenen Fenster lesen	Get-Help –Name about_CommonParameters –ShowWindows
alle Parameter des Cmdlets Get-Command inklusive Beschreibung und Definition anzeigen	Get-Help –Name Get-Command –Parameter *

3

## **CMDLETS IM DETAIL**

3.1

## PIPELINES



## ■ Prinzip der Pipeline

Bearbeitungsschritt	Erläuterung	Häufige Cmdlets
1 Daten bereitstellen	Im ersten Schritt benötigen Sie Daten, mit denen Sie arbeiten wollen. Dafür setzen Sie in der Regel Cmdlets mit dem Verb Get ein.	<b>Get-...</b>
2 Ergebnis bearbeiten, bis es den Erwartungen entspricht	Hier werden Cmdlets eingesetzt, die mit Daten arbeiten, die sie über die Pipeline erhalten. Häufig finden Sie in diesem zweiten Schritt, der auch aus etlichen Teilschritten bestehen kann, Cmdlets mit dem Substantiv Object.	<b>...-Object</b>
3 Endergebnis ausgeben	Lassen Sie diesen Schritt weg, erfolgt eine Ausgabe im Konsolenfenster. Ansonsten haben Sie die Möglichkeit, <ul style="list-style-type: none"><li>✓ die Formatierung Ihren Wünschen anzupassen,</li><li>✓ die Ausgabe umzuleiten, z. B. in eine Datei zu exportieren.</li></ul>	<b>Export-...</b> <b>Format-...</b> <b>Out-...</b>

# Verarbeitung vorliegender Daten: Object-Cmdlets

Cmdlet	Kurzbeschreibung
ForEach-Object	Erlaubt die Anwendung eines Skriptblocks auf jedes übergebene Objekt, funktioniert wie eine Schleife, die alle einzelnen Elemente durchläuft <b>Beispiel: 3,4,5,6,7   ForEach-Object -Process {\$_*2}</b> Sie wollen alle vorliegenden Zahlenwerte verdoppeln.
Group-Object	Gruppiert übergebene Objekte nach den Werten ihrer Eigenschaften <b>Beispiel: Get-Service   Group-Object -Property Status</b> Die Liste der Dienste auf Ihrem Rechner sollen nach ihrem Status (gestartet oder beendet) gruppiert werden.
Measure-Object	Berechnet numerische Eigenschaften von Objekten wie Anzahl, Mittelwert, Summe etc. <b>Beispiel: Get-Command -CommandType cmdlet   Measure-Object</b> Sie wollen ermitteln, wie viele Cmdlets aktuell in der PowerShell verfügbar sind.

# Verarbeitung vorliegender Daten:

## Object-Cmdlets

Cmdlet	Kurzbeschreibung
Select-Object	<p>Wählt Eigenschaften eines Objekts gemäß Ihren Wünschen aus</p> <p><b>Beispiel: Get-Process   Select-Object -Property Id, ProcessName</b></p> <p>Sie möchten von der Liste der laufenden Prozesse nur die Eigenschaften Id und ProcessName sehen.</p>
Sort-Object	<p>Sortiert Objekte nach den Werten der Eigenschaften und entfernt bei Bedarf Mehrfachwerte</p> <p><b>Beispiel: Get-ChildItem   Sort-Object -Property length</b></p> <p>Sie wollen die Dateien bei einer Auflistung des aktuellen Verzeichnisses nach Größe sortieren.</p>
Tee-Object	<p>Die Ausgabe des Befehls wird in einer Datei oder Variablen gespeichert und zusätzlich in der Konsole angezeigt.</p> <p><b>Beispiel: Get-Service   Tee-Object -FilePath .\dienste.txt</b></p> <p>Eine Liste der Dienste auf Ihrem Rechner wird in der Konsole angezeigt und gleichzeitig in die Datei dienste.txt im aktuellen Verzeichnis gespeichert.</p>
Where-Object	<p>Filtert Objekte nach beliebigen Kriterien</p> <p><b>Beispiel: Get-ChildItem C:\Windows\System32   Where-Object {\$_.length -gt 10mb}</b></p> <p>Sie suchen im Verzeichnis C:\Windows\System32 nach Dateien mit einer Dateigröße von mehr als 10 MB.</p>

- **Select-Object**

Get-Service -Name s\* | Select-Object -Property Name, Status

Get-Process -Name PowerShell | Select-Object -Property \*

"Sam", "Hugo", "Sam", "Sam", "Karla", "Hugo" | Select-Object -Unique

- **Sort-Object**

Get-EventLog -LogName System -Newest 20 | Sort-Object -Property InstanceID

Get-ChildItem C:\Windows -File | Sort-Object -Property Extension, Length

- **Where-Object**

Vereinfachte Syntax: Get-Service | Where-Object Status -EQ Running

Klassische Syntax: Get-Service | Where-Object { \$\_.Status -EQ "Running" }

Vereinfachte Syntax: Get-ChildItem -Path C:\Windows -File | Where-Object Length -LT 100KB

Klassische Syntax: Get-ChildItem -Path C:\Windows -File | Where-Object { \$\_.Length -LT 100KB }

- Where-Object

Nur mit klassischer Syntax möglich:

```
Get-ChildItem -Path C:\Windows -File | Where-Object { $_.Length -LT 100KB -OR  
                                                    $_.Length -GT 1MB }
```

- ForEach-Object

```
1..3 | ForEach-Object -Process { Test-Connection 127.0.0.$_ -Count 1 }
```

```
1..3 | ForEach-Object -Begin { Clear-Host; Write-Host "Ich beginne zu pingen : " }  
                        -Process { Test-Connection 127.0.0.$_ -Count 1 }  
                        -End { Write-Host "Aktion beendet." }
```

```
"Client1", "Server2", "Server3", "Client7" | ForEach-Object -Process { Stop-Computer  
                                                                -ComputerName $_ -WhatIf }
```

- Group-Object

Get-ChildItem C:\Windows –File | Group-Object –Property Extension

Get-Command | Group-Object –Property Verb –NoElement | Sort-Object –Property Count

- Measure-Object

7, 3, 5, 22, -5, 4, 11, 17, 2 | Measure-Object –Average –Sum –Maximum –Minimum

Get-Content –Path .\beispiel.txt | Measure-Object –Line –Word –Character

Get-ChildItem –Path C:\Windows | Measure-Object –Property Length –Sum

- Tee-Object

Get-Process | Tee-Object –FilePath .\beispiel.txt

Get-Date | Tee-Object –Variable beispiel | Select-Object –Property Month, DayOfWeek

## ■ Out-Cmdlets

Cmdlet	Kurzbeschreibung
Out-Default	Ein PowerShell-internes Standardformatierungsprogramm (ETS; extended type system) sorgt für eine automatische, typabhängige Formatierung vorliegender Daten. In den meisten Fällen erfolgt dies als Tabelle mit automatisch formatierten Spalten. Danach wird die Ausgabe an das Standard-Ausgabe-Cmdlet gesendet.
Out-File	Die Ausgabe wird an eine Datei gesendet.
Out-GridView	Die Ausgabe erfolgt in einer interaktiven Tabelle in einem eigenen Fenster.
Out-Host	Standard-Ausgabe-Cmdlet, das die Ausgabe zur Anzeige an die Befehlszeile des Power-Shell-Hosts sendet
Out-Null	Die Ausgabe wird gelöscht. Es wird nichts angezeigt.
Out-Printer	Sendet die Ausgabe an einen Drucker
Out-String	Wandelt die Objekte einer Ausgabe in Text um

- Out-File

Get-Process | Out-File -FilePath C:\Daten\prozesse.txt -NoClobber

- Out-GridView

Get-Process | Out-GridView

Get-Process | Out-GridView -PassThru | Out-File -FilePath .\prozesse.txt

- Out-Host

Get-ChildItem -Path C:\Windows\System32 | Out-Host -Paging

- Out-Printer

Welche Drucker sind installiert?

Get-WmiObject -Class Win32\_Printer | Select-Object -Property Name

Get-Process | Out-Printer -Name "Microsoft XPS Document Writer"



## ■ Export-Cmdlets

Cmdlet	Kurzbeschreibung
Export-Clixml	Export von Objekten in XML-Format
Export-Csv	Umwandlung der Daten in kommaseparierte Textdateien

## ■ Export-Clixml

Get-Process -Name PowerShell | Export-Clixml -Path .\ps-prozess.xml

## ■ Export-Csv

Get-Process -Name PowerShell | Select-Object Company, Description, Id, StartTime |  
Export-Csv -Path ps-prozess.csv

Get-Process | Export-Csv -Path .\prozesse.csv -Delimiter ";" -NoTypeInfoation

Get-Process | Export-Csv -Path .\prozesse.csv -UseCulture

## ■ ConvertTo-Cmdlets

Cmdlet	Kurzbeschreibung
ConvertTo-Csv	Konvertiert Objekte in kommaseparierte Werte um. Anders als Export-Csv speichert das Cmdlet die Werte nicht direkt in einer Datei.
ConvertTo-Html	Umwandlung von Objekten in HTML-Inhalt.
ConvertTo-SecureString	Konvertierung von Objekten in verschlüsselte Zeichenfolgen
ConvertTo-Xml	Wandelt Objekte in XML um

## ■ ConvertTo-Csv

Get-Process -Name PowerShell | ConvertTo-Csv  
ConvertTo-Csv -InputObject (Get-Date) -NoTypeInfo

## ■ ConvertTo-Html

Get-Process | ConvertTo-Html -Property Name, Path, FileVersion -Body "<h1>Übersicht  
über die laufenden Prozesse</h1>" | Set-Content -Path .\prozesse.html  
Get-Service -Name d\* | ConvertTo-Html -As List

## ■ ConvertTo-SecureString

ConvertTo-SecureString -String "Hallo!" -AsPlainText -Force  
ConvertTo-SecureString -String (Read-Host [-AsSecureString]) -AsPlainText -Force

## ■ ConvertTo-Xml

Get-Process -Name PowerShell | ConvertTo-Xml -As String | Set-Content -Path .\prozesse.xml

- Endergebnis

```
Get-ChildItem -Path C:\Windows\*.exe |  
  Where-Object { $_.Length -gt 20kb } |  
  Select-Object -Property Length, Name |  
  Sort-Object -Property Length -Descending |  
  ConvertTo-Html -Body "<h1>Beispielüberschrift</h1>" |  
  Set-Content -Path .\daten_kollegen.html
```

- So einfach wie möglich
  - Reicht ein Cmdlet für die Aufgabe aus, die Sie erledigen wollen, dann beschränken Sie sich auf das Cmdlet und dessen Parameter.
  - Für eine komplexere Aufgabe setzen Sie mehrere Cmdlets ein, die über die Pipeline miteinander verbunden sind. Auch hier gilt: Verwenden Sie einzelne Cmdlets, wenn sich über sie – ohne Pipeline – Einzelaufgaben erledigen lassen.
  - Wollen Sie Aufgaben automatisieren, speichern Sie die Befehle bzw. Befehlsfolgen in einem Skript zur späteren Verwendung (siehe spätere Kapitel).
  
- Pipeline zur Absicherung verwenden

Stop-Process –Name PowerShell

oder

Get-Process –Name PowerShell

Get-Process –Name PowerShell | Stop-Process

## 3.2

# DATENSPEICHER IN DER POWERSHELL

PowerShell-Provider	Funktion
Alias	Bietet einen Datenspeicher für alternative Kurzbefehle (vgl. Kapitel 6)
Environment	Datenspeicher für Umgebungsvariablen (vgl. Kapitel 7)
FileSystem	Datenspeicher für das Dateisystem (Laufwerke, Freigaben) (vgl. Abschnitt 5.2)
Function	Datenspeicher für Funktionen (vgl. Kapitel 9)
Registry	Datenspeicher für zwei wichtige Hauptschlüssel der Windows-Registry (vgl. Abschnitt 5.2)
Variable	Datenspeicher für Variablen (vgl. Kapitel 8)

- Informationen über Provider einholen  
Get-PSProvider [-PSProvider]

- Anzeigen

Get-PSDrive –Name Daten

Get-PSDrive –Name Daten | Select-Object –Property \*

Get-PSDrive –PSProvider FileSystem

- Anlegen

New-PSDrive –Name Daten –PSProvider FileSystem –Root \\Fileserver1\Daten

New-PSDrive –Name Z –PSProvider FileSystem –Root \\Fileserver1\Daten –Persist

New-PSDrive –Name Microsoft –PSProvider Registry –Root

HKLM:\SOFTWARE\Microsoft

- Entfernen

Remove-PSDrive –Name Daten

Remove-PSDrive –Name Microsoft –Force

Cmdlet	Erläuterung
Get-Location	Ruft Informationen über das aktuelle Verzeichnis ab
Set-Location	Wechselt in das angegebene Verzeichnis
Push-Location	Fügt das aktuelle Verzeichnis in eine Liste von Verzeichnissen (Stapel, stack) ein, die Sie sich merken wollen
Pop-Location	Ändert das aktuelle Verzeichnis in ein Verzeichnis, das Sie mithilfe von Push-Location einer "Merkliste" hinzugefügt haben

- **Get-Location**  
Get-Location –PSProvider Registry  
Get-Location –PSDrive HKLM
- **Push-Location**  
Push-Location C:\ –StackName wichtig  
Push-Location –StackName wichtig
- **Pop-Location**  
Pop-Location –StackName wichtig



# Cmdlets für die Arbeit mit Elementen und ihren Eigenschaften

Cmdlet	Erläuterung
Clear-Item	Löscht den Inhalt eines Elements, das Element selbst bleibt erhalten. Verwendet wird das Cmdlet z. B. für das Löschen von Werten einer Variablen oder Zweige in der Registry.
Clear-ItemProperty	Löscht den Wert einer Eigenschaft eines Elements, nicht aber die Eigenschaft selbst
Copy-Item	Kopiert ein Element an einen anderen Ort
Copy-ItemProperty	Kopiert die Eigenschaft eines Elements an einen anderen Ort
Get-Item	Ruft Informationen eines Elements ab
Get-ItemProperty	Ruft die Eigenschaften eines Elements ab
Invoke-Item	Führt die im Betriebssystem festgelegte Standardaktion für das angegebene Element aus. Auf diesem Weg können Sie ausführbare Dateien starten oder Dateien mit registrierten Dateiendungen in ihrem zugewiesenen Standardprogramm öffnen.
Move-Item	Verschiebt ein Element an einen anderen Ort
Move-ItemProperty	Verschiebt die Eigenschaft eines Elements an einen anderen Ort

# Cmdlets für die Arbeit mit Elementen und ihren Eigenschaften

Cmdlet	Erläuterung
New-Item	Erstellt ein neues Element
New-ItemProperty	Erstellt für ein vorhandenes Element eine neue Eigenschaft und legt dessen Wert fest
Remove-Item	Löscht ein Element
Remove-ItemProperty	Löscht eine Eigenschaft eines Elements
Rename-Item	Benennt ein Element um
Rename-ItemProperty	Benennt eine Eigenschaft eines Elements um
Set-Item	Ändert den Wert eines Elements
Set-ItemProperty	Ändert bzw. erstellt den Wert einer Eigenschaft eines Elements

## ■ Beispiele

```
Invoke-Item -Path .\update.exe
```

```
Invoke-Item -Path "C:\Meine Dateien\Bearbeiten\rechnung.xlsx"
```

```
Copy-Item -Path C:\Windows\Logs\DISM\dism.log -Destination C:\MeineLogs
```

```
Copy-Item -Path C:\Windows\Logs -Destination C:\MeineLogs -Recurse
```

```
Get-ChildItem -Path C:\Test -Include *.tmp | Remove-Item
```

```
Get-ItemProperty -Path C:\Users\Administrator\test.txt | Select-Object -Property *
```

```
Set-ItemProperty -Path .\fertig.pptx -Name IsReadOnly -Value $true
```

Cmdlet	Erläuterung
Add-Content	Fügt einem oder mehreren Elementen den angegebenen Inhalt hinzu
Clear-Content	Löscht den Inhalt eines Elements, z. B. den Text einer Datei
Get-Content	Ruft den Inhalt eines Elements ab
Set-Content	Schreibt Inhalt in ein Element neu oder ersetzt vorhandenen Inhalt

- Get-Content

Get-Content –Path .\notiz.txt

Get-ChildItem –Path C:\ \*.txt -Recurse | Get-Content –TotalCount 2

- Set-Content

Get-Process | Select-Object –Property Id, Name | Set-Content –Path .\prozesse.txt

Set-Content –Path .\notiz.txt -Value "Wichtiger Termin 14 Uhr!" –PassThru

- Add-Content

Add-Content –Path .\notiz.txt –Value (Get-Date)

Add-Content –Path .\neu.txt –Value (Get-Content –Path .\notiz.txt)

- Clear-Content

Clear-Content –Path C:\Protokolldateien –Filter \*.log –Force

Clear-Content –Path \* –Include abc\* –Exclude \*234\*

- Der Provider Registry bietet die beiden Laufwerke HKCU: (für HKEY\_CURRENT\_USER) und HKLM: (für HKEY\_LOCAL\_MACHINE). Weitere Cmdlets speziell für die Registry bietet die PowerShell nicht. Alle vorgestellten Cmdlets eignen sich für einen Einsatz in diesem Bereich.
- Beispiele
  - Set-Location –Path HKLM:\SOFTWARE
  - New-Item –Path "Meine Software" –ItemType Container
  - New-ItemProperty –Path "HKLM:\SOFTWARE\Meine Software " –Name Version –Value "5.02 Professional " –PropertyType String
  - New-ItemProperty –Path "HKLM:\SOFTWARE\Meine Software" –Name Hexwert –Value 0xABC –PropertyType DWORD
  - Set-ItemProperty –Path "HKLM:\SOFTWARE\Meine Software" –Name Version –Value "5.23 Professional"
  - Remove-ItemProperty –Path "HKLM:\SOFTWARE\Meine Software" –Name Hexwert

- Der Provider Environment stellt das Laufwerk Env: zur Verfügung. In ihm befinden sich als Einzelobjekte die Windows-Umgebungsvariablen. Die Objekte können Sie mit den vorgestellten Mittel auslesen, wie z. B.:

Get-ChildItem –Path Env:\

- Wenn Sie Umgebungsvariablen verwenden wollen, sprechen Sie die gewünschte Variable mit folgender Syntax an:

\$Env:<Variablenname>

### 3.3

## **ALIASE – ALTERNATIVE KURZBEFEHLE**



- Alias (Spitzname)
- Vordefinierte und eigene Aliase
- Ex- und Import von Aliasen
- Aliase löschen
- Übung

- Aliase werden vornehmlich aus zwei Gründen eingesetzt:
  - Erleichterung des Einstiegs in die PowerShell durch die Verwendung bekannter Befehle, die z. B. noch nicht bekannte Cmdlets aufrufen
  - Arbeitserleichterung durch Definition kurzer Befehlsnamen für sperrige oder nicht leicht zu merkende Befehle

- Mit vordefinierten Aliasen arbeiten
  - Get-Alias  
Sie erhalten eine Liste mit etwa 150 vordefinierten Aliasen.
  - Get-Alias [-Name] <string>
    - Get-Alias -Name gal führt zur (verkürzten) Ausgabe: gal -> Get-Alias.
    - Get-Alias gcm ergibt die Ausgabe: gcm -> Get-Command.
  - Get-Alias -Definition <string>
    - Get-Alias -Definition Get-Service führt zur Ausgabe: gsv -> Get-Service.
    - Get-Alias -Definition Get-ChildItem ergibt die Ausgabe: dir, gci, ls -> Get-ChildItem.
  - dir Alias:
    - Der Befehl listet alle Aliase auf, indem es alle Objekte des Informationsspeichers Alias wie in einem Laufwerk durchläuft.
  - start notepad.exe
    - Der Alias start ersetzt hier das Cmdlet Start-Process.

- Eigene Aliase definieren

`Set-Alias [-Name] <String> [-Value] <String>`

`New-Alias [-Name] <String> [-Value] <String>`

- Beispiele

`Set-Alias -Name hilfmir -Value Get-Help`

`New-Alias paint C:\Windows\System32\mspaint.exe`

- Eigens definierte Aliase stehen nur in der aktuellen Instanz der PowerShell zur Verfügung. Wenn Sie die PowerShell schließen und erneut öffnen, stehen die von Ihnen definierten Aliase nicht mehr zur Verfügung.

- Beispiele

Export-Alias –Path alias-export.csv

Export-Alias –Path alias-export.ps1 –As Script –NoClobber

Import-Alias –Path alias-export.csv –Force

- Suchen Sie das geeignete Cmdlet, werden Sie feststellen, dass für diese Tätigkeit kein spezielles Cmdlet existiert.  
Zum Löschen müssen Sie einen anderen Weg gehen. Sie verwenden das allgemeine Cmdlet Remove-Item zum Löschen von Elementen. Sie wenden den Löschvorgang auf das gewünschte Objekt im virtuellen Laufwerk Alias: an.  
**Remove-Item –Path Alias:\<Name>**
- Manche vordefinierte Aliase sind schreibgeschützt. Der Versuch, sie zu löschen, mündet in einer Fehlermeldung.  
Welche Aliase sind schreibgeschützt?  
**Get-Alias | Where-Object Options –Match "ReadOnly" | Select-Object –Property Name, Options**

4

## PROGRAMMIERGRUNDLAGEN

4.1

## **VARIABLEN**



- Varablentypen
  - Einstellungsvariablen
    - \$ErrorActionPreference
    - \$MaximumAliasCount
    - \$MaximumHistoryCount
    - \$WhatIfPreference
  - Automatische Variablen
    - \$?
    - \$Error
    - \$HOME
    - \$PROFILE
    - \$PSHome
    - \$PSVersionTable

- Benutzerdefinierte Variablen
  - Anzeigen:
    - Get-Variable [-Name]
    - Get-ChildItem -Path Variable:
  - Erstellen:
    - New-Variable -Name <name>                      => Ohne Parameter -Value => \$null
    - New-Variable -Name <name> -Value <wert>
    - Zuweisung Wert
  - Ändern:
    - Set-Variable -Name <name> -Value <wert>
    - Zuweisung (neuer) Wert
  - Löschen:
    - Clear-Variable -Name <name> (Wert löschen)
    - Remove-Variable -Name <name> (Variable löschen)
    - Zuweisung \$null

- Variablentypen bei der Typisierung

Typ	Beschreibung
[int]	Ganzzahlwert (32 bit)
[long]	Ganzzahlwert (64 bit)
[string]	Zeichenfolge (Unicode)
[char]	16-bit-Zeichen (Unicode)
[byte]	Zeichen (8 bit)
[bool]	Boolscher Wert (True/False-Wert)
[decimal]	128-bit-Dezimalwert
[single]	Gleitkommazahl (32 bit)
[double]	Gleitkommazahl (64 bit)
[xml]	XML-Objekt
[array]	Array
[hashtable]	Hashtable

- Erstellen
  - `[Typ[]] <name> = Wert(e)`
  - `$<name> = Wert(e)`
  - `New-Variable -Name <name> -Value <wert(e)>`
  - `$<name> = @(Wert(e))`
- Lesen
  - `$<name>`  $\Rightarrow$  alle Werte
  - `$<name>[Index]`  $\Rightarrow$  einzelne Werte
- Ändern
  - `$<name>[Index] = Wert`
  - `$<name>.SetValue(Wert, Index)`
- Erweitern
  - `$<name> += Wert`
- Löschen
  - Siehe Variablen

- Eine Konstante ist eine Sonderform einer Variablen. Die Besonderheit besteht darin, dass eine Konstante nicht gelöscht und ihr Wert nicht mehr verändert werden kann.

**New-Variable –Name <Name> –Value <Wert> –Option Constant**

4.2

## **OPERATOREN**

## ■ Verhalten bei der Wertzuweisung

Beispielwert –Value	Wert der Variablen	Erläuterung
"Herdt-Verlag"	Herdt-Verlag	Der Wert ist als Zeichenkette gekennzeichnet und wird entsprechend übernommen.
37 * 11	Fehler	Die PowerShell sucht vergeblich nach Positionsparametern, die das Argument "*" unterstützen. Durch die Leerzeichen vermutet die PowerShell an dieser Stelle die Übergabe von drei Argumenten.
37*11	37*11	PowerShell erkennt den Wert als Zeichenkette.
(37 * 11)	407	Durch die Klammersetzung ist PowerShell angehalten, erst das Ergebnis der Multiplikation zu bilden. Das Ergebnis 407 wird dann als Zahlwert der Variablen zugewiesen.
Get-Date	Get-Date	Eine Auswertung des Cmdlets findet nicht statt, es wird als Zeichenfolge interpretiert.
(Get-Date)	z. B.: Montag, 19. Mai 2014 14:24:31	Durch die Klammersetzung wird das Cmdlet Get-Date ausgeführt. Es ergibt sich damit als Wert der Variablen das aktuelle Datum.

Operator	Name	Bedeutung	Beispiel	Ergebnis
+	Addition	$\$a + \$b$ ergibt die Summe von $\$a$ und $\$b$ .	$\$a = 10$ $\$b = 2$ $\$c = \$a + \$b$	$\$c = 12$
-	Subtraktion	$\$a - \$b$ ergibt die Differenz von $\$a$ und $\$b$ .	$\$a = 10$ $\$b = 2$ $\$c = \$a - \$b$	$\$c = 8$
*	Multiplikation	$\$a * \$b$ ist das Produkt aus $\$a$ und $\$b$ .	$\$a = 10$ $\$b = 2$ $\$c = \$a * \$b$	$\$c = 20$
/	Division	$\$a / \$b$ ist der Quotient von $\$a$ und $\$b$ .	$\$a = 10$ $\$b = 2$ $\$c = \$a / \$b$	$\$c = 5$
%	Modulo	$\$a \% \$b$ ist der Rest der ganzzahligen Division von $\$a$ und $\$b$ .	$\$a = 10$ $\$b = 3$ $\$c = \$a \% \$b$	$\$c = 1$
++	Präinkrement	$++\$a$ erhöht die Variable $\$a$ um 1 <b>vor</b> der weiteren Verwendung.	$\$a = 10$ $\$b = 2$ $\$c = ++\$a + \$b$	$\$a = 11$ $\$c = 13$



Operator	Name	Bedeutung	Beispiel	Ergebnis
--	Prädekrement	--\$a verringert die Variable \$a um <b>1 vor</b> der weiteren Verwendung.	\$a = 10 \$b = 2 \$c = --\$a + \$b	\$a = 9 \$c = 11
++	Postinkrement	\$a++ erhöht die Variable \$a um <b>1 nach</b> der Verwendung.	\$a = 10 \$b = 2 \$c = \$a++ + \$b	\$a = 11 \$c = 12
--	Postdekrement	\$a-- verringert die Variable \$a um <b>1 nach</b> der Verwendung.	\$a = 10 \$b = 2 \$c = \$a-- + \$b	\$a = 9 \$c = 12
+=	Zuweisungsoperator	\$a += \$b weist der Variablen \$a den Wert \$a + \$b zu (Kurzschreibweise für \$a = \$a + \$b).	\$a = 10 \$a += 5	\$a = 15
-=	Zuweisungsoperator	\$a -= \$b ist die Kurzschreibweise für \$a = \$a - \$b.	\$a = 10 \$a -= 5	\$a = 5
*=	Zuweisungsoperator	\$a *= \$b ist die Kurzschreibweise für \$a = \$a * \$b.	\$a = 10 \$a *= 5	\$a = 50
/=	Zuweisungsoperator	\$a /= \$b ist die Kurzschreibweise für \$a = \$a / \$b.	\$a = 10 \$a /= 5	\$a = 2

Operator	Bedeutung	Beispiele
-eq	Ist gleich (equal to)	2 -eq 3 (Ergebnis: FALSE) "Harry" -eq "harry" (Ergebnis: TRUE)
-ne	Ist ungleich (not equal to)	2 -ne 3 (Ergebnis: TRUE) "Harry" -ne "harry" (Ergebnis: FALSE)
-gt	Größer als (greater than)	2 -gt 3 (Ergebnis: FALSE) "Harry" -gt "harry" (Ergebnis: FALSE)
-ge	Größer oder gleich (greater than or equal to)	2 -ge 3 (Ergebnis: FALSE) "Harry" -ge "harry" (Ergebnis: TRUE)
-lt	Kleiner als (less than)	2 -lt 3 (Ergebnis: TRUE) "Harry" -lt "harry" (Ergebnis: FALSE)
-le	Kleiner oder gleich (less than or equal to)	2 -le 3 (Ergebnis: TRUE) "Harry" -le "harry" (Ergebnis: TRUE)
-Contains	Prüft eine gegebene Werteliste, ob ein Wert enthalten ist	"abc ", "def " -Contains "abc" (Ergebnis: TRUE) "abc ", "def " -Contains "bc" (Ergebnis: FALSE)
-NotContains	Umkehrung des Operators -Contains	"abc ", "def " -NotContains "abc" (Ergebnis: FALSE) "abc ", "def " -NotContains "bc" (Ergebnis: TRUE)

Operator	Bedeutung	Beispiele
–In	Wie –Contains, nur in umgekehrter Reihenfolge	"abc " –In "abc", "def" (Ergebnis: TRUE) "bc " –In "abc","def " (Ergebnis: FALSE)
–NotIn	Umkehrung des Operators –In	"abc " –NotIn "abc", "def" (Ergebnis: FALSE) "bc " –NotIn "abc","def " (Ergebnis: TRUE)
–Is	Prüfung auf Typgleichheit	[string]\$string = "Herdt-Verlag" \$string –Is [string] (Ergebnis: TRUE)
–IsNot	Umkehrung des Operators –Is	[string]\$string = "Herdt-Verlag" \$string –IsNot [string] (Ergebnis: FALSE)
–Like	Einfacher Textabgleich; prüft, ob sich eine Zeichenkette in einer anderen befindet	"abcdef " –Like "abc" (Ergebnis: FALSE) "abcdef " –Like "abc*" (Ergebnis: TRUE) "abcdef " –Like "*c*" (Ergebnis: TRUE)
–NotLike	Umkehrung des Operators –Like	"abcdef " –NotLike "abc" (Ergebnis: TRUE) "abcdef " –NotLike "abc*" (Ergebnis: FALSE) "abcdef " –NotLike "*c*" (Ergebnis: FALSE)
–Match	Suche nach Substring ohne Wildcards (auch Mustervergleich mit regulären Ausdrücken)	"abcdef " –Match "c" (Ergebnis: TRUE)
–NotMatch	Umkehrung des Operators –Match	"abcdef " –NotMatch "c" (Ergebnis: FALSE)

- Komplexere Vergleiche mit mehreren Einzelvergleichen

Operator	Bedeutung	Beispiele
–and	Alle Einzelbedingungen müssen erfüllt sein.	( (2 –gt 3) –and (2 –gt 1) –and (0 –gt -1) ) (Ergebnis: FALSE)
–or	Mindestens eine der Einzelbedingungen muss erfüllt sein.	( (2 –gt 3) –or (2 –gt 1) –or (0 –gt -1) ) (Ergebnis: TRUE)
–xor	Genau eine der Einzelbedingungen muss erfüllt sein, nicht aber mehrere oder alle.	( (2 –gt 3) –xor (2 –gt 1) –xor (0 –gt 1) ) (Ergebnis: TRUE)

## 4.3

# KONTROLLSTRUKTUREN

- Herzstück einer Programmiersprache ist die Möglichkeit, eine Programmierung individuell nach eigenen Vorstellungen zu gestalten. Kontrollstrukturen steuern dabei den Ablauf einer Programmierung durch Verzweigungen und Schleifen.
- Verzweigungen führen – abhängig vom Ergebnis der Prüfung von Bedingungen – Anweisungsblöcke aus. Schleifen steuern die wiederholte Ausführung von Anweisungsblöcken, bis eine Bedingung erfüllt ist.

```
If (Bedingung)
{
    Anweisungsblock
}
```

- Beispiel

```
If ($x -eq 27)
{
    Write-Host "Der Wert der Variablen x ist 27."
}
```

```
If (Bedingung)
{
    Anweisungsblock 1
}
Else
{
    Anweisungsblock 2
}
```



## ■ Beispiel

```
$x = 27          # Wert zum Testen anpassen
If ($x -eq 27)
{
    Write-Host "Der Wert der Variablen x ist 27."
}
Else
{
    Write-Host "Der Wert von x – sofern vorhanden – ist nicht 27."
}
```

```
If (Bedingung 1)
{
    Anweisungsblock 1
}
Elself (Bedingung 2)
{
    Anweisungsblock 2
}
[Elself (Bedingung n)
{
    Anweisungsblock n
}]
Else
{
    Anweisungsblock Else
}
```

## ■ Beispiel

```
$x = 19          # Wert zum Testen anpassen
If ($x -gt 27)
{
    Write-Host "Der Wert der Variablen x ist größer als 27."
}
Elseif ($x -eq 27)
{
    Write-Host "Der Wert der Variablen x ist genau 27."
}
Else
{
    Write-Host "Der Wert von x – sofern vorhanden – ist kleiner als 27."
}
```

```
switch ($variable)
{
    Wert-1 {
        Anweisungsblock 1
        [break]
    }
    Wert-2 {
        Anweisungsblock 2
        [break]
    }
    Default {
        Anweisungsblock 3
    }
}
```

## ■ Beispiel

```
$a = 3           # Wert zum Testen anpassen
Switch ($a)
{
    1 { Write-Host "A = 1" }
    2 { Write-Host "A = 2" }
    3 { Write-Host "A = 3" }
    Default { Write-Host "Nicht 1, nicht 2, nicht 3." }
}
```

- **While- bzw. Do-While-Schleife:**  
Wenn Ihnen als Programmierer nicht bekannt ist, wie oft eine Anweisung wiederholt werden soll, bzw. Sie eine Schleife so lange ausführen möchten, bis eine bestimmte Bedingung eingetroffen ist, verwenden Sie die **While-** bzw. **Do-While-Schleife**.
- **For-Schleife:**  
Wenn Sie die genaue Anzahl kennen oder diese vorher in der PowerShell ermitteln können, wie oft eine Anweisung wiederholt werden soll, verwenden Sie die **For-Schleife**.
- **ForEach-Schleife:**  
Wenn Sie eine Anzahl von Werten, z. B. in einer Array-Variablen, der Reihe nach lesen und weiterbearbeiten möchten, ist dieser Schleifentyp die richtige Wahl.

```
While (Bedingung)  
{  
    Anweisungsblock  
}
```

## ■ Beispiel

```
$a = 0          # Wert zum Testen anpassen  
While ($a -le 3)  
{  
    Write-Host $a  
    $a = $a + 1  # alternativ: $a++ oder $a += 1  
}
```

```
Do
{
    Anweisungsblock
}
While (Bedingung)
```

## ■ Beispiel

```
$a = 0          # Wert zum Testen anpassen
Do
{
    Write-Host $a
    $a = $a + 1  # alternativ: $a++ oder $a += 1
}
While ($a -le 3)
```



```
Do
{
    Anweisungsblock
}
Until (Bedingung)
```

## ■ Beispiel

```
$a = 0          # Wert zum Testen anpassen
Do
{
    Write-Host $a
    $a = $a + 1  # alternativ: $a++ oder $a += 1
}
Until ($a -gt 3)
```

**For (Initialisierung; Bedingung; Reinitialisierung)**

```
{  
    Anweisungsblock  
}
```

## ■ Beispiel

```
for ($a = 1; $a -le 5; $a++)  
{  
    Test-Connection -ComputerName 127.0.0.$a -Count 1  
}
```

```
ForEach ($<name> in <Quelle>)  
{  
    Anweisungsblock  
}
```

## ■ Beispiel

```
$array = 2, 4, 6, 8, 10  
ForEach ($wert in $array)  
{  
    $wert * 5  
}
```

- Schleifenabbruch mit break
  - Nicht immer ist es notwendig, eine Schleife bis zum definierten Ende zu durchlaufen. Auch innerhalb des Schleifendurchlaufs können andere Kriterien geprüft werden, die das weitere Durchlaufen der Schleife überflüssig machen. Wenn Sie z. B. 100 Schleifendurchläufe planen, das gewünschte Ergebnis bereits nach 10 Durchläufen finden, machen die restlichen 90 Schleifendurchläufe keinen Sinn mehr. Sie können in diesem Fall die Operation mit dem Schlüsselwort break abbrechen.
- Vorzeitiger Sprung zum nächsten Schleifendurchlauf mit continue
  - Wenn Sie erreichen möchten, dass für einen bestimmten Wert die weiteren Anweisungen innerhalb der Schleife nicht mehr ausgeführt werden, dann verwenden Sie das Schlüsselwort continue. Die Schleife selbst ist davon nicht betroffen und wird so oft durchlaufen, bis die von Ihnen definierte Bedingung nicht mehr erfüllt ist.
  - Mit continue springt das Programm zurück zum Anfang der Schleife und beginnt ihren nächsten Durchlauf. Bei For-Schleifen wird zudem die Reinitialisierungs-Anweisung ausgeführt.

5

## **SKRIPT-PROGRAMMIERUNG**

5.1

## FUNKTIONEN

- Eine Funktion besteht aus einer Liste von Anweisungen, die Sie mit einer Bezeichnung versehen haben. Wenn Sie eine Funktion ausführen wollen, geben Sie die Bezeichnung (den sogenannten Funktionsnamen) ein. Die Liste der Anweisungen wird dann wie bei einer manuellen Eingabe an der Eingabeaufforderung ausgeführt.
- Funktionen können über Parameter verfügen. Sie können Werte zurückgeben, die angezeigt, Variablen zugewiesen oder an andere Funktionen oder Cmdlets übergeben werden können.
- Funktionen sind ein Teil des Skriptings. Sie erstellen ein Skript mit Funktionsdefinitionen und verwenden sie dann interaktiv oder in anderen Skripten.

```
Function <Name>  
{  
    <Anweisungsblock>  
}
```

Die komplette Funktionsdefinition muss einmal in der aktuellen PowerShell-Sitzung ausgeführt werden. Ab diesem Zeitpunkt steht Ihnen die Funktion zur Verfügung.

Um Funktionen zu verwenden, rufen Sie sie mit ihrem Namen auf. Eventuelle Parameter geben Sie jeweils durch Leerzeichen getrennt im Anschluss ein. Die Syntax eines Funktionsaufrufs gleicht der Syntax der Cmdlets:

```
<Funktionsname> [-Parameter1 [Wert1]] [-Parameter2 [Wert2]] [-ParameterN  
[WertN]]
```



```
Function <Name>
{
    Param ($Parameter1[, $Parameter2])
    <Anweisungsblock>
}
```

Oder

```
Function <Name> [($Parameter1[, $Parameter2])]
{
    <Anweisungsblock>
}
```

Bestandteile einer Funktionsdefinition sind:

- Das Schlüsselwort **Function**
- Der von Ihnen vergebene Funktionsname
- Optional: beliebig viele Parameter der Funktion
- Beliebig viele PowerShell-Befehle in geschweiften Klammern („{}“)

- Beispiel

```
Function Get-LargeFiles ($length)
{
    Get-ChildItem C:\Windows | Where-Object {$_.length -ge $length}
}
```

- In der Funktionsdefinition wird ein Parameter \$length definiert. Die Variable \$length wird in der Funktionsdefinition bei der Formulierung der Bedingung \$\_.length -ge \$length verwendet.
- Ein mit dem Funktionsaufruf angegebener Wert wird in der Funktionsdefinition mit diesem Variablennamen angesprochen.
- Aufruf der Funktion:  
Get-LargeFiles 500  
Get-LargeFiles 1MB  
Get-LargeFiles

- Beispiel

```
Function Get-LargeFiles ($location = "C:\Windows", $length = 1MB)
{
    Get-ChildItem C:\Windows | Where-Object {$_.length -ge $length}
}
```

- Aufruf der Funktion:

```
Get-LargeFiles
Get-LargeFiles -location C:\Users\Administrator
Get-LargeFiles -location F:\Daten -length 100kb
```

Ein Switch-Parameter erfordert keinen Wert. Stattdessen geben Sie den Funktionsnamen und danach den Namen des Parameters ein. Wird der Parameter angegeben, erhält er den Wert \$True, wird er nicht angegeben, besitzt er den Wert \$False.

Wenn Sie einen Switch-Parameter definieren möchten, geben Sie den Typ [switch] vor dem Parameternamen an, wie Sie im folgenden allgemeinen Beispiel sehen:

```
Function Test-Switch ([switch]$AnAus)
{
    If ($AnAus)
    {
        Write-Host "Eingeschaltet..."
    }
    Else
    {
        Write-Host "Ausgeschaltet..."
    }
}
```

- Als Shell liefert die PowerShell keine Rückgabewerte, sondern schreibt Ausgaben an die Ausgabe-Pipeline.
- Beispiel

```
Function rueckgabe
{
    1 + 1
    "Hallo"
    Get-Process *calc*
}
$returnvar = rueckgabe
```

Jeder Befehl innerhalb einer Funktion, der eine Ausgabe in der Ausgabe-Pipeline erzeugt, generiert somit automatisch eine Ausgabe der Funktion. Wird diese Ausgabe in einer Variablen gespeichert, so enthält die Variable ein gemischtes Array mit allen Ausgabeelementen.

- Auch PowerShell kennt die Anweisung **return**, aber sie folgt einer anderen Logik. Grundsätzlich dient sie dazu, die Ausführung einer Funktion oder eines Code-Abschnitts zu beenden und die Kontrolle an den übergeordneten Block zu übertragen.

Ergänzt man das **return**-Statement um einen Parameter, dann gelangt dessen Wert tatsächlich zurück zum Aufrufer der Funktion. Das gilt jedoch auch für alle anderen Anweisungen, die eine Ausgabe erzeugen, so dass sich diese zusammen mit dem Parameter von **return** in einer Variablen finden.

```
Function rTest
{
    $a = 5
    $b = "Das ist ein Test"
    $a
    return $b
}
$r = rTest
```

# Objekte über die Pipeline an eine Funktion übergeben

```
Function <Name>
{
    Begin
    {
        <Anweisungsblock>
    }
    Process
    {
        <Anweisungsblock>
    }
    End
    {
        <Anweisungsblock>
    }
}
```

- Zu Beginn der Funktionsausführung wird der Begin-Anweisungsblock einmalig ausgeführt.
- Der Process-Anweisungsblock wird für jedes Objekt ausgeführt, das der Funktion über die Pipeline übergeben wird. Während der Ausführung des Anweisungsblockes repräsentiert die Variable `$_` das aktuelle Objekt.
- Nach der Verarbeitung aller Objekte wird der Anweisungsblock einmalig ausgeführt, der durch das Schlüsselwort `End` eingeleitet wird.
- Falls die Schlüsselwörter `Begin`, `Process` und `End` nicht verwendet werden, werden alle Anweisungen wie ein End-Anweisungsblock behandelt.



# Objekte über die Pipeline an eine Funktion übergeben

## ■ Beispiel

```
Function Get-Pipe1
{
    Begin
    {
        Write-Host "Pipeline-Funktion (Werte: $input)"
    }
    Process
    {
        Write-Host "Wert des aktuellen Pipeline-Objekts: $_."
    }
    End
    {
        Write-Host "Verarbeitete Werte: $input."
    }
}
```

1, 2, 3, 4, 5, 6, 7, 8, 9 | Get-Pipe1

- Wenn Sie eine Funktion in einer Pipeline verwenden, werden die über die Pipeline an die Funktion übergebenen Objekte automatisch der Variablen `$input` zugewiesen. Diese beinhaltet am Ende der Funktionsausführung die Werte aller übergebenen Objekte.
- Das aktuelle Objekt können Sie im Process-Anweisungsblock über die Variablen `$_` oder `$input` an sprechen.
- Die Variable `$input` kann in einem Begin-Anweisungsblock nicht ausgegeben werden. Vor der Verarbeitung der einzelnen Objekte besitzt die Variable noch keinen Wert.
- Falls Sie den Process-Anweisungsblock verwenden, ist die Variable `$input` im End-Anweisungsblock leer.
- Nur wenn Sie den Process-Anweisungsblock weglassen, können Sie im End-Anweisungsblock mit der Variablen `$input` auf die Werte der übergebenen Objekte zugreifen.

# Objekte über die Pipeline an ein Skript übergeben

- Beispiel: pipescript.ps1

```
Begin
{
    Write-Host "Große Dateien vor rotem Hintergrund:"
}
Process
{
    If ($_.length -lt 1MB) { $_ }
    Else
    {
        $Standardfarbe = $host.UI.RawUI.BackgroundColor
        $host.UI.RawUI.BackgroundColor = "Red"
        $_
        $host.UI.RawUI.BackgroundColor = $Standardfarbe
    }
}
End
{
    Write-Host "Ende der Verarbeitung."
}
```

- Aufruf

```
Get-ChildItem C:\Windows | .\pipescript.ps1
```

5.2

## **FILTER**

Als Filter wird ein spezieller Funktionstyp bezeichnet, der für jedes Objekt in der Pipeline ausgeführt wird. Filter ähneln Funktionen, bei denen sich alle Anweisungen in einem Process-Block befinden.

```
Filter <Name>  
{  
    <Anweisungsblock>  
}
```

## ■ Beispiel

```
Filter NewFiles ([int]$tage)  
{  
    $jetzt = Get-Date  
    $_ | Where-Object { ($jetzt - $_.LastWriteTime).Days -lt $tage}  
}
```

5.3

## **EIN PAAR DETAILS**

Alle Funktionen und Filter in der PowerShell werden automatisch auf dem Laufwerk Function: gespeichert. Dieses Laufwerk wird vom PowerShell-Provider Function zur Verfügung gestellt.

- Funktionen und Filter anzeigen

## **Get-ChildItem Function:**

- Definitionen von Funktionen und Filtern anzeigen

**(Get-ChildItem Function:\<Funktionsname>).Definition**

- Eigene Funktionen und Filter löschen

**Remove-Item -Path <genaue Bezeichnung>**

```
<#  
.SYNOPSIS  
Kurzbeschreibung  
.DESCRIPTION  
Ausführliche Beschreibung  
.PARAMETER <ParameterName-1>  
Beschreibung des ersten Parameters  
.PARAMETER <ParameterName-N>  
Beschreibung des n. Parameters  
.EXAMPLE  
Beispielanwendung und -erläuterung  
.EXAMPLE  
Weitere Beispielanwendung und -erläuterung  
.NOTES  
Weitere Hinweise  
.LINK  
Angabe von URLs oder ähnlichen Cmdlets  
#>
```



6

## **POWERSHELL-MODULE**

6.1

## ÜBERSICHT

- Module können als Erweiterungen der PowerShell verstanden werden. Es handelt sich um themenspezifische Sammlungen von Befehlen. Teilweise wird mit dem Laden eines Moduls auch ein PowerShell-Provider definiert, der ein neues PowerShell-Laufwerk definiert. Alle PowerShell-Cmdlets entstammen grundsätzlich Modulen.
- Das Modul Microsoft.PowerShell.Core beinhaltet einen Kernsatz an Cmdlets, Funktionen und Providern, die direkt mit der PowerShell installiert werden und damit immer zur Verfügung stehen. Weitere Module können zu jedem Zeitpunkt einer PowerShell-Sitzung nachgeladen werden.
- Auslesen der möglichen Speicherorte:  
**\$Env:PSModulePath**

- Aktuell geladene Module anzeigen  
**Get-Module**
  
- Verfügbare Module, geladen oder nachladbar, anzeigen  
**Get-Module –ListAvailable**
  
- Welche Module zur Verfügung stehen, hängt von verschiedenen Faktoren ab:
  - Betriebssystem
  - Installation von Betriebssystemkomponenten
  - Installation von Rollen und Features (Windows Server)
  
- Welchem Modul sind welche Cmdlets zugeordnet  
**Get-Command –CommandType <Cmdlet>**

- Module werden bei Bedarf automatisch nachgeladen, wenn Sie ein Cmdlet eines noch nicht geladenen Moduls aufrufen.  
Allerdings empfiehlt es sich nach wie vor, häufiger verwendete Module im Voraus zu laden, z. B. über ein Profilskript. Die mit dem Laden verbundene Wartezeit wird so auf den Start der PowerShell verschoben und verlangsamt nicht die Abarbeitung von Eingaben oder Skripten im laufenden Betrieb.
- Laden von Modulen in der aktuellen Sitzung  
**Import-Module –Name <Modulname>[, <Modulname2>]**
- Modul aus der aktuellen Sitzung entfernen  
**Remove-Module –Name <Modulname>[, <Modulname2>]**
- Welche Befehle enthält ein spezielles Modul  
**Get-Command –Module <Modulname>[, <Modulname2>]**

6.2

## BEISPIELE

- Das Modul ServerManager bietet zusätzliche Befehle zum Automatisieren und Bereitstellen von Rollen und Features auf Windows Server 2012 R2. Die Befehle sind als Alternative zur Aktion an der grafischen Oberfläche im Server-Manager zu verstehen.
- Beispiele

## **Get-WindowsFeature**

**Get-WindowsFeature | Where-Object {\$\_.Installed -eq \$true}**

**a) Get-WindowsFeature –Name Web-\* | Install-WindowsFeature  
–ComputerName Server3**

**b) Install-WindowsFeature –Name Web-Server –IncludeAllSubFeature  
–ComputerName Server3**

- Die Verwaltung von DNS-Servern durch die PowerShell ist erst ab PowerShell Version 3.0 mithilfe des neuen Moduls DnsServer möglich. Bislang konnten Sie DNS-Server über die grafische Oberfläche oder zum Teil mit dem Befehlszeilentool dnscmd.exe verwalten.

- Beispiele

## **Get-DnsServer**

**Get-DnsServerResourceRecord -ZoneName <Zone> | Where-Object  
{\$\_RecordType -eq "A" }**

**Get-DnsServerZone -ComputerName Server3 | Format-List**

**Export-DnsServerZone -Name goettingen.ad -FileName export-  
goettingen.ad.dns**



- Das Modul ServerCore liefert zwei zusätzliche Cmdlets für eine Situation, die Sie bislang nur durch einen Eingriff in die Registry lösen konnten. Das Modul wurde entwickelt, um die Bildschirmauflösung eines Rechners im Server Core-Modus auszulesen und zu ändern.
- Beispiele

**Get-DisplayResolution**

**Set-DisplayResolution –Width 1024 –Height 768**

- Ein Modul zum Ansprechen einer Datenbank
  - MSSql
  - MySql
  - Oracle
  - ...