

# PowerShell 7

## Praxis für Fortgeschrittene

(MS 112)

Peter Monadjemi  
pm@activetraining.de

v1.1.0 – 14/11/25

# Die Themen für Tag 1

2

Lektion	Tag	Thema
1	1	AI Shell im Überblick
2	1	Visual Studio Code als Alternative zur PowerShell ISE
3	1	PowerShell 7.x im Überblick
4	1	Moderne PowerShell
5	1	Die Objektpipeline in Theorie und Praxis
	1	Übungen für den ersten Tag

# Die Themen für Tag 2

3

Lektion	Tag	Thema
6	2	Functions und Advanced Functions
7	2	Umgang mit Modulen
8	2	Arrays und Hashtables
	2	Übungen für den 2. Tag

# Die Themen für Tag 3

4

Lektion	Tag	Thema
9	3	Textdaten verarbeiten
10	3	PowerShell-Skripte debuggen
11	3	Regeln für gute Skripte
12	3	Tipps für die Praxis
		Übungen für den 3. Tag (optional)

# Themen für Tag 4

5

Lektion	Tag	Thema
X1	4	Umgang mit Klassen
X2	4	PowerShell Remoting mit SSH
X3	4	Secrets verwalten mit dem SecretManagement-Modul
X4	4	Module und Skripte mit Pester testen

# Formalitäten

6

- Kurszeiten
- Pausen, Mittagessen usw.
- Login am Computer
- Internet-Zugang per WLAN
- Unterlagen als Pdf (bitte auf die Versionsnummer achten)
- Adresse für die Beispiele...

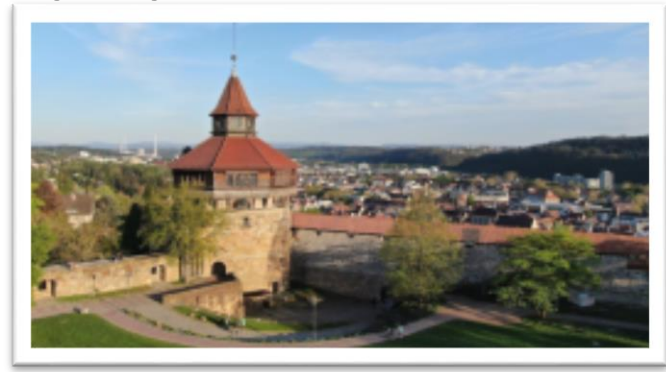
<https://github.com/pemo11/MS112>

- **Tipp:** Herunterladen über `git clone` in der Befehlszeile (setzt Git für Windows voraus)

# Über mich...

7

- Seit vielen Jahren Trainer mit dem Schwerpunkten Windows-Automatisierung, PowerShell und Software-Entwicklung
- Lebe seit vielen Jahren in Esslingen am Neckar
- Seit 2022 offizieller Bachelor Medieninformatik (HS Emden Leer – Seniorenstudium😊)
- Aktuell Student im Master Studiengang Medieninformatik



(Burganlage mit Blick auf die Stadt Richtung Südosten)

# Das Ziel der Schulung

8

- ❑ PowerShell-Kenntnisse auffrischen und vertiefen
- ❑ Ein tieferes Verständnis für die Arbeitsweise der PowerShell und den Umgang mit der internen Syntax, der Pipeline und den Commands und Modulen
- ❑ Tipps für die Praxis (z.B. Performance-Tipps) mitnehmen
- ❑ Erfahrungsaustausch
- ❑ Gelegenheit sich an vier Tagen (ohne Ablenkungen) mit der PowerShell 7 zu beschäftigen



# Vorbereitungen

# Die Kursumgebung

10

- ❑ PowerShell 7.x und Visual Studio Code müssen eventuell nachträglich installiert werden
- ❑ Das Y-Laufwerk als gemeinsame Dateiablage
- ❑ Es können nicht alle Apps installiert werden

# Kein Admin?

11

- Alle Beispiele gehen von einer „Admin-Shell“ aus
- Die meisten Beispiele funktionieren aber auch **ohne** eine Administratorberechtigung
- **Ausnahmen:** PS-Remoting, Zugriff auf den HKLM-Zweig der Registry, Systemdienste starten/stoppen, Schreibzugriff auf bestimmte Verzeichnisse usw.
- In einigen Fällen gibt es eine Lösung, z.B. der **Scope**-Parameter bei **Set-ExecutionPolicy** oder **Install-Module**
- **Wichtig:** Module können daher in der Regel ohne "Adminberechtigung" installiert werden

# Aktualisieren der Hilfe

12

- Seit > 20 Jahren ein Thema, bei dem nicht nur Freude aufkommt
- **Problem:** Bei der Windows PowerShell geht [Update-Help](#) nicht ohne Adminberechtigung
- Keine Hilfe zu haben ist auch keine Lösung (es gibt alle Inhalte aber auch online)
- Bei PowerShell 7.x gibt es ebenfalls einen [Scope](#)-Parameter
- Fehlermeldungen am Schluss sind „normal“

Windows PowerShell – Adminberechtigung ist Voraussetzung

```
Update-Help -UICulture en-US -ErrorAction SilentlyContinue -Verbose
```

PowerShell 7 – Adminberechtigung ist nicht erforderlich

```
Update-Help -Scope CurrentUser -UICulture en-US -ErrorAction SilentlyContinue -Verbose
```

# Wenn ein Proxy im Spiel ist

13

- ❑ Ausgangspunkt: Proxy setzt Authentifizierung voraus
- ❑ Es gibt bei Update-Help keinen Proxy-Parameter
- ❑ Die Parameter Credential/UseDefaultCredentials bringen nichts
- ❑ Übliche Einstellungen über Internet Explorer
- ❑ Ansonsten ist ein kleiner Workaround erforderlich

Sollte nicht erforderlich sein

```
$wc = New-Object System.Net.WebClient  
$wc.Proxy.Credentials = [System.Net.CredentialCache]::DefaultNetworkCredentials
```

Ausführliche Beschreibung der Problematik mit Lösungen (2018) - geht aber nicht auf die Schnelle

<http://woshub.com/using-powershell-behind-a-proxy/>

# Die Beispiele für die Schulung

14

- Sind Teil des Zip-Downloads
- Alternativ über das GitHub-Repo
- <https://github.com/pemo11/MS112>
- Download als Zip-Datei oder per git clone (setzt Git for Windows voraus) **1**
- Zip-Datei auspacken z.B. in Documents-Verzeichnis

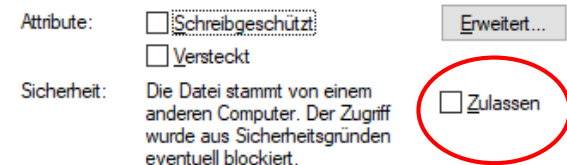
**1**

```
PS>git clone https://github.com/pemo11/MS112
```

# Die Rolle der Zone-Info (1)

15

- In manchen Umgebungen werden Dateien nach dem Download mit einer Zone-Info „markiert“
- Zu erkennen an dem Zulassen-Button im Eigenschaftendialogfeld
- Für PowerShell-Skripte und Module muss dieser „Zone.Identifier“ (ADS) entfernt werden:
  - Über die Zulassen-Checkbox
  - Über das Unblock-File-Command
  - Über die lokale Sicherheitsrichtlinie -> Benutzerkonfiguration -> Windows-Komponenten->Anlagen-Manager->Zoneninformationen in Dateianlagen nicht beibehalten



# Die Rolle der Zone-Info (2)

16

- Keine Zoneninformationen schreiben per Registry aktivieren
- Wie wird das per PowerShell gemacht?

## OPTION TWO

### To Enable or Disable Downloaded Files from being Blocked using a REG file

✍ The downloadable .reg files below will add and modify the DWORD value in the registry keys below.

HKEY\_CURRENT\_USER\Software\Microsoft\Windows\CurrentVersion\Policies\Attachments

HKEY\_LOCAL\_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Policies\Attachments

**SaveZoneInformation** DWORD

(delete) = Enable

1 = Disable



# Installation des PoshKurs-Moduls

17

- Das (optionale) PoshKurs-Modul enthält Functions (PowerShell-Befehle), die für die Schulung eine Rolle spielen
- Es steht in meinem Repository zur Verfügung:  
<https://www.myget.org/F/poshrepo/api/v2>
- Download per Install-Module 2
- Zuvor muss das Repository in der PowerShell über Register-PSRepository hinzugefügt werden 1

1

```
Register-PSRepository -Name PoshRepo -SourceLocation  
https://www.myget.org/F/poshrepo/api/v2
```

2

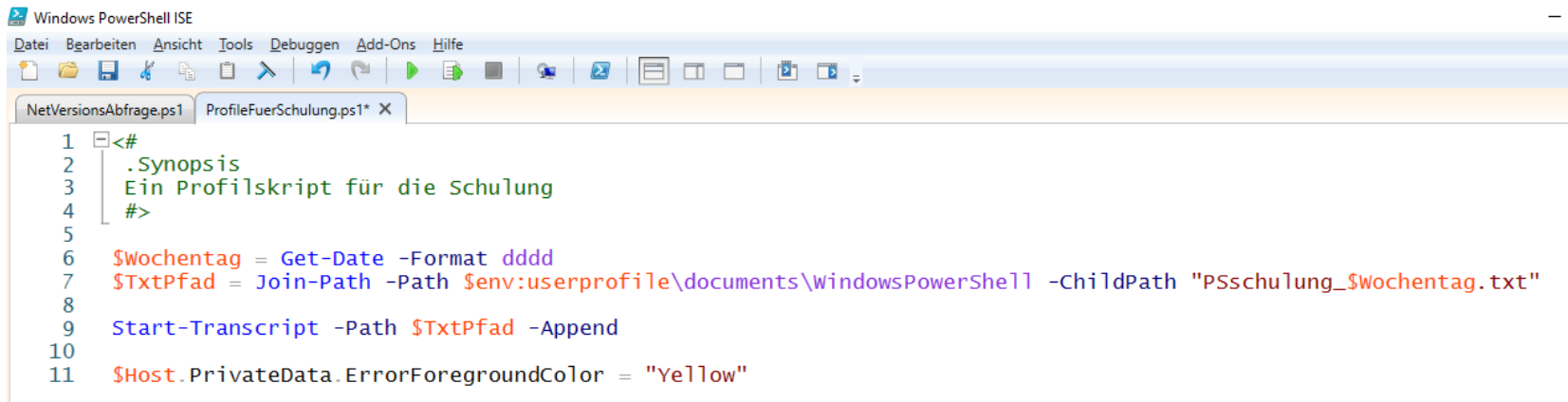
```
Install-Module -Name Poshkurs-Repository PoshRepo
```

```
Get-Command -Module Poshkurs
```

# Anlegen eines Profilskripts

18

- Das Profilskript wird automatisch bei jedem Start einer Host-Anwendung ausgeführt
- Muss einen festgelegten Namen besitzen und in einem festgelegten Verzeichnis abgelegt werden
- **Beispiel:** Profile.ps1 in %userprofile\documents\windowspowershell



```
Windows PowerShell ISE
Datei Bearbeiten Ansicht Tools Debuggen Add-Ons Hilfe
NetVersionsAbfrage.ps1 ProfileFuerSchulung.ps1* X
1 <#
2 .Synopsis
3 Ein Profilskript für die Schulung
4 #>
5
6 $Wochentag = Get-Date -Format dddd
7 $TxtPfad = Join-Path -Path $env:userprofile\documents\WindowsPowerShell -ChildPath "PSSchulung_$Wochentag.txt"
8
9 Start-Transcript -Path $TxtPfad -Append
10
11 $Host.PrivateData.ErrorForegroundColor = "Yellow"
```

# AI Shell, CoPilot&Co

## Themenblock 1

# CoPilot– so viele Möglichkeiten

20

- Ändert fast alles
- Erstellt funktionsfähige PowerShell-Skripte durch simple Prompts
- Das Ergebnis kann sich sehen lassen...
- Vorteil: Jetzt kann wirklich jeder „skripten“
- Es gibt eine „Erklärfunktion“ zu jedem Detail!
- Muss man es noch verstehen können?



Prompt

```
Erstelle ein PowerShell-Skript, das alle Dateien im Documents-Verzeichnis in ein  
neues Verzeichnis verschiebt, die älter als 365 Tage und 5 Minuten sind
```

# CoPilot – eine Musterlösung

21

```
# Pfade festlegen
$sourceDirectory = "$env:USERPROFILE\Documents"
$destinationDirectory = "$env:USERPROFILE\Documents\OldFiles"
# Datum berechnen
$cutOffDate = (Get-Date).AddDays(-365).AddMinutes(-5)
# Überprüfen, ob das Zielverzeichnis existiert. Falls nicht, erstellen
if (-not (Test-Path $destinationDirectory))
{
    New-Item -ItemType Directory -Force -Path $destinationDirectory | Out-Null
}
# Alle Dateien im Quellverzeichnis abrufen
$files = Get-ChildItem -Path $sourceDirectory -File
# Dateien überprüfen und verschieben
foreach ($file in $files)
{
    if ($file.LastWriteTime -lt $cutOffDate)
    {
        Move-Item -Path $file.FullName -Destination $destinationDirectory -Force
        Write-Host "Die Datei '$($file.Name)' wurde verschoben."
    }
}
```

# CoPilot oder ChatGPT?

22

- Beides sind generative KI-Tools
- ChatGPT stammt von OpenAI, Codepilot wurde von Microsoft zusammen mit OpenAI entwickelt
- Copilot wurde mit „riesigen Mengen von Code“ von GitHub trainiert
- In aktuellen Versionen von Visual Studio Code ist CoPilot integriert
- Nicht kostenlos, aber es gibt eine kostenlose Testphase

<https://code.visualstudio.com/docs/copilot/overview>

# Nahtlose Integration

23

- Nahtlose Integration in Visual Studio Code über eine Extension (setzt aber ein Abo voraus)
- Vorteile:
  - ▣ Intelligente Vervollständigung/Vorschläge
  - ▣ Korrekturvorschläge (in einem eigenen Fenster – Pair Programming)
  - ▣ Code-Generierung aus Kommentaren und dem Dateinamen der Ps1-Datei
  - ▣ Kommentare schreiben wird enorm erleichtert
  - ▣ Insgesamt ein großer Produktivitätsverstärker
- Nachteile:
  - ▣ Es gibt keine Garantie für 100% Funktionsfähigkeit
  - ▣ Verlernen wir wichtige Fähigkeiten?
  - ▣ Machen wir uns von der KI abhängig?
  - ▣ Gigantischer Ressourcenverbrauch durch KI-Rechenzentren

# CodePilot in der Praxis

24

- ❑ Das Chat-Fenster muss nicht geöffnet werden, einfach loslegen...
- ❑ Ob CoPilot aktiv ist, erkennt man dem Icon in der Statusleiste
- ❑ Über das Icon kann CoPilot abgeschaltet werden
- ❑ Mit der Eingabe von Code wird „Geistertext“ produziert
- ❑ Übernahme per Tab, Ablehnen per Esc
- ❑ Per Enter-Taste geht es weiter
- ❑ Meine Empfehlung Anforderung als Kommentar schreiben
- ❑ **Tipp:** Vorschläge über Strg+Enter anzeigen lassen



# CoPilot– für diese Schulung

25

- ❑ Kann gerne verwendet werden (es gibt aber leider keine Schulungs-Accounts – ansonsten ChatGPT)
- ❑ Per CoPilot generierte Skripte sollten aber erklärt werden können
- ❑ Beispiele für den Einsatz von CoPilot:
  - ▣ Erklär mir diesen Befehl/dieses Skript
  - ▣ Welche Fehler enthält dieses Skript?
  - ▣ Kann dieses Skript optimaler umgesetzt werden?

# CoPilot – mein Fazit

26

- ❑ CoPilot spielt bereits jetzt eine wichtige Rolle
- ❑ Echter „Produktivitätsverstärker“
- ❑ Theoretisch muss man nicht mehr „skripten“ können
- ❑ Für kleine Aufgaben/Anforderungen ideal
- ❑ Für größere Anforderungen kann es einen Rahmen anlegen oder mehr
- ❑ CoPilot und andere KI-Tools werden unseren Arbeitsalltag verändern
- ❑ Zu wissen, was geht und was nicht ist daher wichtig

# AI-Shell (1)

27

- ❑ Konsole mit integrierter KI
- ❑ Eigenständig oder als PowerShell-Modul
- ❑ Vorteil: Skriptgenerierung erfolgt im selben Fenster
- ❑ Aktuell entweder Azure oder OpenAI – daher nicht kostenlos nutzbar und Datenschutzproblematik
- ❑ Immer noch Preview (Stand: November 2025)
- ❑ Weitere Infos unter <https://aka.ms/AIShell-Docs>

# AI-Shell (2)

28

- Es gibt zwei Varianten:
  - ▣ Eigenständiger Prozess `ai.sh.exe` mit eigenem Befehlsinterpreter
  - ▣ PowerShell-Module
- Installation von `ai.sh.exe` nach 1  
`$env:LOCAL\APPDATA\Programs\AIShell`

1

```
Invoke-Expression "& { $(Invoke-RestMethod 'https://aka.ms/install-aihell.ps1') }
```

# AI-Shell (3)

29

- Konfiguration des ChatGPT-Agenten für die eigenständige AI-Shell
- **Schritt 1:** API-Key bei OpenAI besorgen
- **Schritt 2:** AI-Shell per `aish.exe` starten
- **Schritt 3:** Json-Datei per `/agent config` aufrufen
- **Schritt 4:** Key eintragen, Agenten-Konfiguration auskommentieren und den Rest löschen
- **Schritt 5:** `/agent refresh`
- Dann irgendetwas eingeben;)

# Visual Studio Code als Alternative zu ISE

## Themenblock 2

# Die Themen

31

- PowerShell ISE – der aktuelle Stand
- Vorteile von Visual Studio Code
- Wann sollte man Visual Code nicht verwenden?
- Kurze Einführung in Visual Studio Code

# PowerShell ISE – der aktuelle Stand

32

- ❑ Windows PowerShell 5.1 und PowerShell ISE werden nicht weiterentwickelt
- ❑ ISE geht nicht mit PowerShell 7
- ❑ Grundsätzlich spricht nichts dagegen, die ISE für die Windows PowerShell weiter zu verwenden
- ❑ Guter Allround-Editor, mit dem man immer ans Ziel kommt
- ❑ Visual Studio + PowerShell Extension ist die Zukunft



# Visual Studio Code im Überblick

33

- Vielseitiger Allround-Editor für alle Plattformen
- Klassisches Open Source-Projekt
- Vorbilder sind VIM, Atom etc.
- Modular, performant, sehr nah an den Wünschen der Anwender
- Wird von Microsoft in Zürich entwickelt unter der Leitung von Erich Gamma
- Nicht zu verwechseln mit Visual Studio, das es nur für Windows gibt

# Visual Studio Code und PowerShell

34

- ❑ Die PowerShell Extension macht aus Visual Studio Code eine hervorragende Scripting-Entwicklungsumgebung
- ❑ Klare Optik, reaktionsschneller Editor
- ❑ Komfortabler Debugger mit viel Komfort (u.a Variablenanzeige)
- ❑ Zahlreiche Eingabehilfen dank integriertem PSScriptAnalyzer
- ❑ Das Look&Feel inkl. Tastaturshortcuts der ISE gibt es auch für VS Code
- ❑ Die PowerShell Extension wird laufend weiterentwickelt

# Visual Studio Code einrichten

35

**1**

**Visual Studio Code installieren**

**2**

**PowerShell Extension installieren**

**3**

**Git for Windows installieren**

**4**

**PoshKurs Repo öffnen**

**5**

**Skripte ausführen/debuggen**

# Kurze Einführung in Visual Studio Code (1)

36

- Vieles ist selbsterklärend, aber nicht alles
- Zuerst muss die PowerShell Extension installiert werden
- **Wichtig:** Damit etwas „passiert“, muss eine Ps1-Datei gespeichert werden
- Empfehlung: Zuerst das Verzeichnis öffnen, in dem die Ps1-Datei gespeichert werden soll bzw. in dem sich bereits Ps1-Dateien befinden

# Visual Studio Code kennenlernen

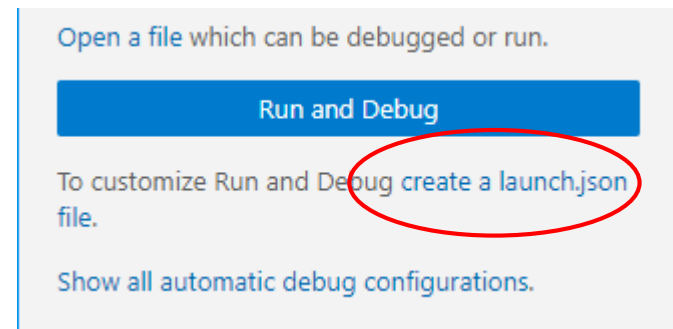
37

- ❑ Meine Empfehlung – etwas Zeit nehmen, um sich mit den wichtigsten Einstellungen vertraut zu machen
- ❑ Layout und Schriftart auswählen
- ❑ Aktionsleiste am linken Rand kennenlernen
- ❑ Verzeichnis mit den Übungsbeispielen öffnen und einzelne Skripte ausführen
- ❑ Debugger ausprobieren
- ❑ PowerShell-Session beenden und neu starten
- ❑ Repository (z.B. GitHub) direkt öffnen

# Skripte ausführen

38

- Wie üblich über F5
- Die aktuelle Zeile bzw. ein markierter Bereich werden über F8 ausgeführt
- Haltepunkte umschalten per F9 usw. (alles wie in der ISE)
- Auf den Debugger umschalten per F6
- Anlegen einer launch.json-Datei (u.a. für die Befehlszeilenargumente)



# Die Rolle der Einstellungen

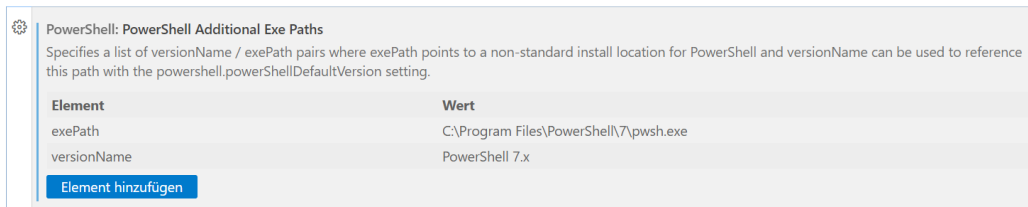
39

- **Settings.json** für die Einstellungen zu VS Code und seinen Erweiterungen
  - ▣ Einstellungen pro Benutzer/Arbeitsbereich
  - ▣ Aufruf über Datei | Einstellungen
  - ▣ Einstellungen per Default über die GUI oder direkt in Settings.json
- **Launch.json** ist optional
  - ▣ Legt fest, was nach dem Drücken von F5 passiert
  - ▣ Wird u.a. für Befehlszeilenargumente benötigt

# Skripte per PowerShell 7 ausführen

40

- Eventuell ist die Windows PowerShell als Default-Shell eingestellt
- Es spielt keine Rolle, welche PowerShell im Terminalfenster ausgewählt wurde
- Änderung in Einstellungen (bzw. Settings.json)



z.B. am Ende von Settings.json

```
"powershell.powerShellAdditionalExePaths": {  
  "exePath": "C:\\Program Files\\PowerShell\\7\\pwsh.exe",  
  "versionName": "PowerShell 7.x"  
},
```



```
"powershell.powerShellDefaultVersion": "PowerShell 7.x",
```



# Zusammenfassung

41

- Visual Studio Code mit PowerShell Extension bietet sowohl für „Anfänger“ als auch für erfahrene Anwender viel Komfort
- Die große Stärken von VS Code sind der hervorragende Editor und die unzähligen Erweiterungen
- Zu den vielen Extras gehören u.a. ein direkter Zugang in das Azure-Portal, per SSH in das WSL oder andere Server-Umgebungen, die nahtlose Copilot-Integration und vieles mehr

# PowerShell 7.x im Überblick

## Themenblock 3

# Die Themen

43

- ❑ PowerShell 7 versus Windows PowerShell
- ❑ PowerShell unter Linux&Co
- ❑ Breaking Changes
- ❑ Echte Parallelverarbeitung
- ❑ Umgang mit Null-Werten
- ❑ Automatische Background-Ausführung
- ❑ Weitere Neuerungen
- ❑ Wie bleibt man auf dem Laufenden?

# PowerShell 7 versus Windows PowerShell (1)

44

- ❑ PowerShell basiert auf .Net (Core)/Windows PowerShell basiert auf dem .NET Framework
- ❑ Cmdlets/Module wurden entfernt, z.B. **Get-Eventlog** oder **PSScheduledJob**
- ❑ Keine Workflow-Funktionalität, keine Transaktionen
- ❑ DSC ist nicht mehr Teil der PowerShell (separates Projekt)
- ❑ Viele neue Cmdlets, z.B. **Remove-Service**, **Get-UpTime**, **Test-Json** oder **Remove-Alias**

# PowerShell 7 versus Windows PowerShell (2)

45

- ❑ PowerShell 7.x ist nur eine Anwendung
- ❑ Parallelbetrieb mit Windows PowerShell ist kein Problem
  - ▣ Es gibt unterschiedliche (Modul-) Verzeichnisse
- ❑ Installation z.B. im Programme-Verzeichnis, die Programmdatei ist *Pwsh.exe*
- ❑ Optionales Update über Windows Update

# PowerShell unter Linux&Co

46

- ❑ 100% identisch zur PowerShell unter Windows
- ❑ Es gibt zwangsläufig sehr viel weniger Cmdlets und Module
- ❑ > 6.000 Commands bei PowerShell 7.26 unter Windows, 270 bei PowerShell 7.26 unter Linux&Co
- ❑ 251 Module unter Windows 10, 9 unter Ubuntu
- ❑ Gute Übersicht in der Microsoft-Dokumentation

<https://docs.microsoft.com/de-de/powershell/scripting/whats-new/unix-support>

# "Breaking-Changes"

47

- Änderung erforderlich - der **Encoding**-Parameter von **Get-Content** kennt bei PowerShell den Wert "Byte" nicht mehr
- Keine Änderung erforderlich - bei **Export-CSV** ist der **NoTypeInfoInformation**-Parameter optional

Windows PowerShell

```
Get-Content -Path .\Test.dat -Encoding Byte
```

PowerShell

```
Get-Content -Path .\Test.dat -ReadCount 0 -AsByteStream
```

# Ein Blick in die Doku lohnt sich...

48

- Welches Cmdlet/Modul unter welcher Version verfügbar ist, ist übersichtlich dokumentiert

<https://docs.microsoft.com/de-de/powershell/scripting/whats-new/cmdlet-versions>

## Modulreleaseverlauf

Modulname/PS-Version	5,1	7.0	7.2	7.3	Hinweis
CimCmdlets	✓	✓	✓	✓	Nur Windows
ISE (eingeführt in 2.0)	✓				Nur Windows
Microsoft.PowerShell.Archive	✓	✓	✓	✓	
Microsoft.PowerShell.Core	✓	✓	✓	✓	
Microsoft.PowerShell.Diagnostics	✓	✓	✓	✓	Nur Windows



# Die wichtigsten Neuerungen bei PowerShell 7

49

1. Parallelverarbeitung bei **ForEach-Object** durch den Parameter **-Parallel**
2. Ternärer Operator **?** und **:** (ersetzt in vielen Situationen einen if/else-Block)
3. Null-Member-Operatoren (z.B. **??** oder **\${<varName>}?.**, vergleichbar mit C#)
4. Backgroundjobs per **&** am Ende einer Befehlszeile
5. Kürzere Fehlermeldungen

# Parallelverarbeitung bei ForEach-Object

50

- ❑ **Neu:** Parallel-Parameter beim ForEach-Object-Cmdlet
- ❑ **Wichtig:** Nicht verwechseln mit `-AsParallel` beim ForEach-Befehl in einem Workflow
- ❑ Parallelverarbeitung auf der Basis der bereits mit Version 2.0 eingeführten Runspaces
- ❑ Daher die üblichen Einschränkungen was den Zugriff auf Variablen/Functions außerhalb des Scriptblocks betrifft
- ❑ Am besten ein paar Beispiele...

# Ternärer Operator

51

- Praktische Abkürzung für if/else, die überfällig war
- Allgemein: <Bedingung> ? True-Part : False-Part

```
$WSLimit = 200MB
$Treshold = 10
$BigProcess = Get-Process | Where-Object WS -gt $WSLimit
$status = $BigProcess.Count -gt $Treshold ? "Warnung" : "OK"
$status
```

# Null-Operatoren

52

- Umgang mit Null-Werten spielt in Skripten eine Rolle
- Null-Abfragen und Fehler, die mit Null-Werten zu tun haben, werden vermieden
- „to coalesce“ = Zusammenfügen

Bezeichnung	Syntax	Beispiel
Null Coalescing	??, ??=	<code>\$n = \$null</code> <code>\$n ?? "PowerShell,,</code> <code>PowerShell</code>
Null Conditional	?. und \${varname}?.	<code>\$d = Get-Item GibtNicht.txt</code> <code>-EA Ignore</code> <code>\${d}?.Compress()</code>

# Umgang mit Null-Werten

53

- Es treten weniger unerwartete Fehler auf

```
class Test
{
    $P1 = 0
    $P2
    [void]DoIt(){}
}
$t = [Test]::new()
$t1 = $null
${t}?.DoIt()
${t1}?.DoIt()
```



Dank ?. anstelle von . gibt es keine Fehlermeldung

# Automatische Background-Ausführung

54

- Wird an eine Befehlszeile ein & angehängt, wird die Befehlszeile als Backgroundjob ausgeführt
- Das Ergebnis ist ein Job-Objekt, das über Get-Job und Receive-Job abgefragt wird
- Spielt vor allem unter Linux eine Rolle

```
PS C:\Users\pemo20> Get-Process | Where-Object WS -gt 100MB &
```

Id	Name	PSJobTypeName	State	HasMoreData	Location	Command
3	Job3	BackgroundJob	Running	True	localhost	Microsoft.PowerShell.Man...

```
PS C:\Users\pemo20> Get-Job
```

Id	Name	PSJobTypeName	State	HasMoreData	Location	Command
1	Job1	BackgroundJob	Completed	False	localhost	Microsoft.PowerShell.Man...
3	Job3	BackgroundJob	Completed	True	localhost	Microsoft.PowerShell.Man...

```
PS C:\Users\pemo20> |
```

# Weitere Neuerungen

55

- ❑ Der .NET Supportzyklus
- ❑ Automatische Updates
- ❑ PowerShell über den Windows Store
- ❑ DSC für PowerShell 7
- ❑ „Native Experience“ dank Crescendo
- ❑ Secret Management-Module
- ❑ Aktuelle Versionen u.a. PlatyPS, PowerShellGet, PSReadline usw.

<https://devblogs.microsoft.com/powershell-community/my-crescendo-journey/>

# Moderne PowerShell

## Themenblock 4



# Die Themen

57

- ❑ Terminal statt Eingabeaufforderung
- ❑ OhMyPosh
- ❑ Tipps für die Eingabeaufforderung
- ❑ Fehler reduzieren durch `#requires` und `Set-StrictMode`
- ❑ Erweiterungsmethoden `ForEach{}` und `Where{}`
- ❑ Generische Listen statt Arrays
- ❑ `Using namespace` statt lange Typennamen
- ❑ Modularer Ansatz statt einem „Megaskript“
- ❑ Auslagern von variablen Daten in einer Config-Datei

# Terminal statt Eingabeaufforderung

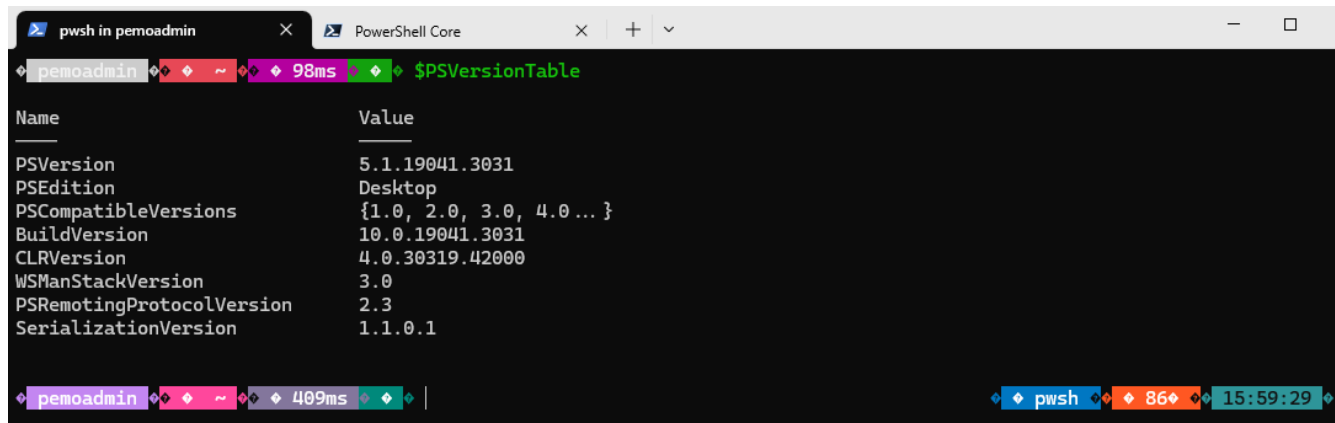
58

- ❑ Bei Windows 11 standardmäßig aktiviert
- ❑ Ein Fenster, beliebig viele Shells
- ❑ Jeder Shell kann eine eigene Konfiguration zugeordnet werden
- ❑ Sehr praktisch, wenn PowerShell 7, PowerShell 7 unter WSL (Linux), Windows PowerShell parallel betrieben werden müssen
- ❑ Für Session-Neustart einfach neues Register anlegen

# OhMyPosh

59

- ❑ Ausgefallene Erweiterung für Windows Terminal
- ❑ Bunte Prompts (teilweise richtig genial)
- ❑ Funktioniert mit jeder (!) Shell, die in Windows Termin eingebunden wird (daher auch mit Windows PowerShell)
- ❑ <https://ohmyposh.dev/>



The screenshot shows a Windows Terminal window with two tabs: "pwsh in pemoadmin" and "PowerShell Core". The active tab is "pwsh in pemoadmin", which displays a colorful prompt: `pemoadmin ~ 98ms $PSVersionTable`. Below the prompt, the output of the `$PSVersionTable` command is shown as a table with two columns: "Name" and "Value".

Name	Value
PSVersion	5.1.19041.3031
PSEdition	Desktop
PSCompatibleVersions	{1.0, 2.0, 3.0, 4.0 ... }
BuildVersion	10.0.19041.3031
CLRVersion	4.0.30319.42000
WSManStackVersion	3.0
PSRemotingProtocolVersion	2.3
SerializationVersion	1.1.0.1

At the bottom of the terminal, another prompt is visible: `pemoadmin ~ 409ms |`. In the bottom right corner, there is a status bar showing `pwsh 86 15:59:29`.

# Tipps für die Eingabeaufforderung

60

- ❑ Path-Umgebungsvariable über Profilskript erweitern
- ❑ Komfortable Befehlshistorie dank PSReadline
- ❑ Suche in der Befehlshistorie per F8
- ❑ PSReadline bietet viel Komfort:
  - ▣ Mehrzeiliges Editieren (sehr praktisch)
  - ▣ Viele Shortcuts, z.B. Strg+A (Get-PSReadLineKeyHandler)
  - ▣ Farbige Tokens (Get-PSReadLineOption)
  - ▣ Eingabevervollständigung mit Machine Learning-Techniken

# Farbige Ausgaben

61

- Bei PowerShell 7 werden einige Ausgaben automatisch farbig
- Ausgabe wird über VT100-Escapesequenzen gesteuert
  - ▣ Wurde mit einem Update von Windows 10 möglich
  - ▣ Gilt allgemein für Konsolenprogramme
- Das Esc-Zeichen 0x1b ist bei PowerShell 7 vordefiniert (``e`)
- Die Variable **\$PSStyle.OutputRendering** steuert die Ausgabe

Gelb auf blauem Hintergrund

```
$logMsg = "`e[93m*** Starte Protokollierung *** `e[0m"
```

<https://duffney.io/usingansiescapesequencespowershell/>

# #requires und Set-StrictMode

62

- Reduzieren Fehler
- #requires verhindert Skriptstart unter falschen Voraussetzungen
- Set-StrictMode erzeugt Fehler, wenn eine nicht initialisierte Variable verwendet wird


```
#requires -modules @{ModuleName="Pester";ModuleVersion="5.0.0"}
```

# Erweiterungsmethoden ForEach{} und Where{}

63

- Gibt es für alle Arrays (und Listen)
- Legen keine Pipeline an (Performance)
- Kompaktere Syntax
- Können verkettet werden

DateTime-Wert



```
$dListe = @($d1, $d2, $d3, $d4, $d4)
$dListe.Where{$_ .DayOfWeek -eq 'Saturday' -or $_ .DayOfWeek -eq 'Sunday'}.ForEach{
    "Der $($_.ToString("d")) ist am Wochenende"
}
```

# Generische Listen statt Arrays

64

- Die .Net-Runtime bietet zahlreiche generischen Listenklassen, vor allem List[T]
- Kleinere Vorteile gegenüber Arrays

```
# Beispiel für eine generische Liste mit DateTime-Objekten
using namespace System.Collections.Generic

# Eine generische Liste erstellen

$d1 = Get-Date -Date "4.5.2024"
$d2 = Get-Date -Date "1.9.2024"
$d3 = Get-Date -Date "31.09.2024"

$dListe = [List[DateTime]]::new()
$dListe.Add($d1)
$dListe.Add($d2)
$dListe.Add($d3)

$dListe[0]
```



# Using namespace statt lange Typennamen

65

- Praktische Abkürzung
- Macht Skripte etwas besser lesbar
- Vor allem, wenn WinForms oder Datenbanken im Spiel sind

## Ohne

```
[System.Windows.Forms.MessageBox]::Show("Noch einmal?", "Wichtiger Hinweis",  
"YESNO","Exclamation")  
  
[System.Windows.Forms.MessageBox]::Show("Alles klar?")
```

## Mit

```
using namespace System.Windows.Forms  
  
[MessageBox]::Show("Noch einmal?", "Wichtiger Hinweis", "YESNO","Exclamation")  
  
[MessageBox]::Show("Alles klar")
```

# Modularer Ansatz dank psm1-Dateien

66

- ❑ Auslagern von Functions und Klassendefinitionen in psm1-Dateien
- ❑ Hat Vorteile, aber auch Nachteile
- ❑ Vorteile: Skripte werden kleiner, Psm1-Datei können in mehreren Skripten verwendet werden
- ❑ Nachteile: Höherer Pflegeaufwand

# Psd1-Dateien für externe Daten

67

- ❑ Importieren per Import-PowerShellDataFile
- ❑ Psd1-Datei enthält Hashtable-Schreibweise

## ScriptConf.psd1

```
@{  
    AnzahlDurchlaufe=10  
    Username = "pemo"  
    ConString = "Data Source=.\SQLEXPRESS;Initial Catalog=TestDb;Integrated Security=SSPI"  
    # LogPfad = "C:\Users\pemo24\Documents\Posh1.log"  
}
```

```
$ConfigPath = Join-Path -Path $PSScriptRoot -ChildPath "ScriptConf.psd1"  
$ConfigData = Import-PowerShellDataFile -Path $ConfigPath  
$ConfigData.AnzahlDurchlaufe
```

# Zusammenfassung

68

- ❑ Moderne PowerShell wird auf zwei Ebenen umgesetzt:
  - ❑ Moderne Tools (u.a. VS Code, Terminal)
  - ❑ Nutzen der Möglichkeiten, die PowerShell bietet (z.B. `#requires`, Konfigurationsdaten in `psd1`-Datei, SecretManagement-Modul)
- ❑ Auch Kleinigkeiten gehören dazu (z.B. Farbige

# Die Objektpipeline in Theorie und Praxis

## Themenblock 5

# Die Themen

70

- Die Rolle der Pipeline
- Was waren noch einmal Objekte?
- Der [Pipeline]-Parameter
- Pipeline-Objekte zählen
- Das Prinzip der Parameterbindung
- Parameterbindung per Name einer Eigenschaft
- Parameterbindung per Wert
- Die Parameterbindung sichtbar machen

# Die Rolle der Pipeline

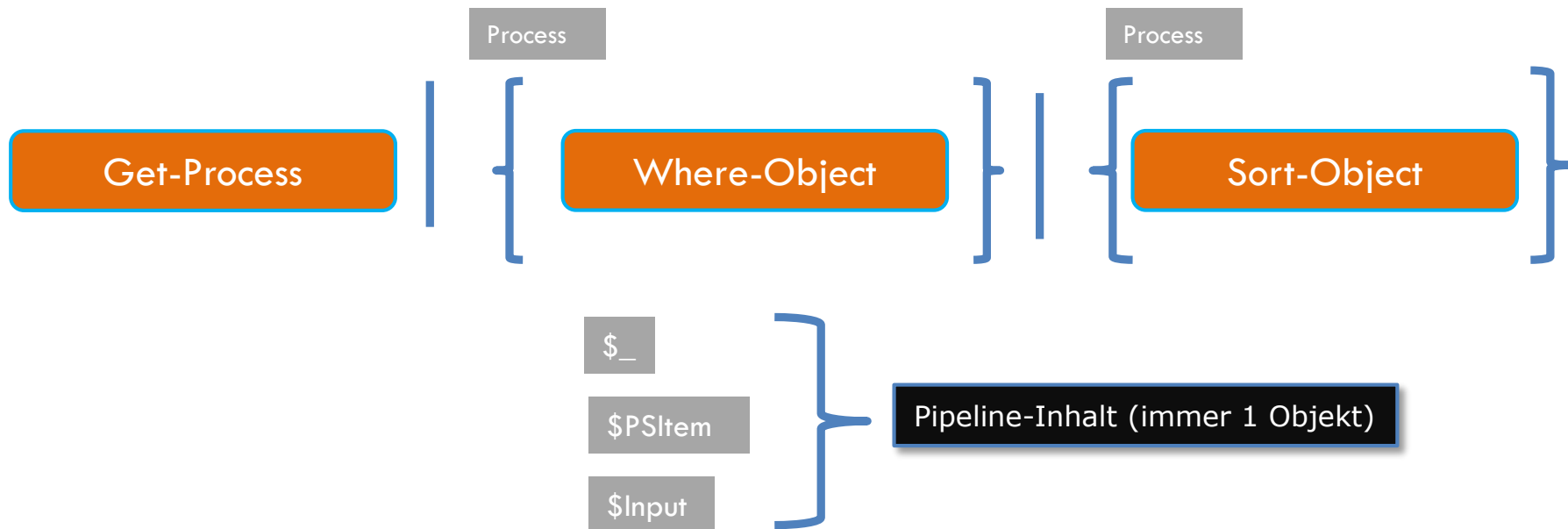
71

- Die Pipeline verbindet die Ausgabe eines Cmdlets mit einem oder mehreren Parametern eines zweiten Cmdlets
- Die Pipeline wird pro Befehlsausführung angelegt
- Die Abarbeitung der Pipeline verläuft immer in drei Schritten:
  - Begin – wird am Anfang der Pipeline-Verarbeitung einmal ausgeführt
  - Process – wird pro Objekt, das in die Pipeline gelegt wird, einmal ausgeführt
  - End – wird am Ende der Pipeline-Verarbeitung ausgeführt

# Die Pipeline-Verarbeitung (1)

72

- Am wichtigsten ist der Process-Block
- Über `$_` wird der aktuelle Inhalt der Pipeline angesprochen





# Die Pipeline-Verarbeitung (2)

73

## Am Beispiel des ForEach-Object-Cmdlets

```
Get-ChildItem -Path C:\Windows\*.ini | ForEach-Object -Begin { "Pipeline-  
Verarbeitung beginnt..." } -Process { "Pipeline-Inhalt: $_" } -End { "Pipeline-  
Verarbeitung fertig..." }
```



Pipeline-Inhalt

# Einzelne Objekte in der Pipeline auswählen

78

- **Select-Object** besitzt mehrere Parameter
  - ▣ **Index** n – auf ein bestimmtes Objekt anhand seiner Reihenfolge zugreifen
  - ▣ **Skip** n – die ersten n Elemente überspringen
  - ▣ **First** n – die letzten Elemente zurückgeben
  - ▣ **Last** n – die letzten Elemente zurückgeben
- Die Gesamtzahl aller Objekte erhält man z.B. über die Count-Eigenschaft (runde Klammern)

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

```
Get-Process | Where-Object WS -gt 100MB -OutVariable Proz100MB  
@($Proz100MB).Count
```

# Was waren noch einmal die Objekte?

79

- Ein Objekt fasst mehrere Informationen und Befehle für einen „Gegenstand“ (z.B. ein Prozess) zusammen und stellt diese über Members (Eigenschaften, Methoden usw.) zur Verfügung
- Über Objekte wird die Weiterverarbeitung von Abfragen vereinfacht, da die Detaildaten über Eigenschaften angesprochen werden
- **Wichtig:** Alle Get-Cmdlets geben Objekte zurück und keinen Text
- **Tipp:** Möchte man trotzdem Text, muss ein **Out-String** angehängt werden

# Kurz und knapp

80

- Ein Objekt fasst alle Merkmale eines „Gegenstandes“ (Prozess, Verzeichnis, Benutzerkonto usw.) zusammen
- Jedes Merkmal besitzt einen eigenen Namen
- Zwischen dem Namen des Gegenstandes und dem Namen des Merkmals steht immer ein Punkt (oder Doppelpunkt)

# Objekte versus Text

81

- Objekte sind immer dann besser, wenn die Rückgabe eines Cmdlets weiterverarbeitet werden soll

## Rückgabe als Text

Tasklist



Abbildname	PID	Sitzungsname	Sitz.-Nr.	Speichernutzung
=====	=====	=====	=====	=====
System Idle Process	0	Services	0	4 K
System	4	Services	0	300 K
smss.exe	368	Services	0	500 K



Nur Text, muss zerlegt werden

1

## Rückgabe als Objekte

Get-Process | Sort-Object  
-Property Id



Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
-----	-----	-----	-----	-----	-----	-----	-----
0	0	0	4	0		0	Idle
3868	0	1020	300	5		4	System
347	27	13184	12028	121		288	svchost
44	2	248	500	4		368	smss

1

Eigenschaft

# Objekte und ihre Members

82

- Objekte besitzen Members
- Members = Eigenschaften (Properties), Methoden (Methods), NoteProperty, ScriptProperty, PropertySet usw.
- Die Members eines Objekts erhält man per **Get-Member-**Cmdlet

# Das Prinzip der Parameterbindung

83

- Parameterbindung bedeutet, dass ein Parameter seinen Wert von dem Objekt in der Pipeline enthält
- Es gibt zwei Sorten der Parameterbindung:
  - ▣ Über den Namen der Eigenschaft des Objekts in der Pipeline (**ByPropertyName**)
  - ▣ Über den Wert in der Pipeline (**ByValue**)
- Welche Sorten der Parameterbindung ein Parameter unterstützt, erfährt man aus der PowerShell-Hilfe

```
-Name <String[]>  
  Specifies the service names for the service to be started.  
  
Erforderlich?           true  
Position?               1  
Standardwert  
Pipelineeingaben akzeptieren? true (ByPropertyName, ByValue)  
Platzhalterzeichen akzeptieren? false
```

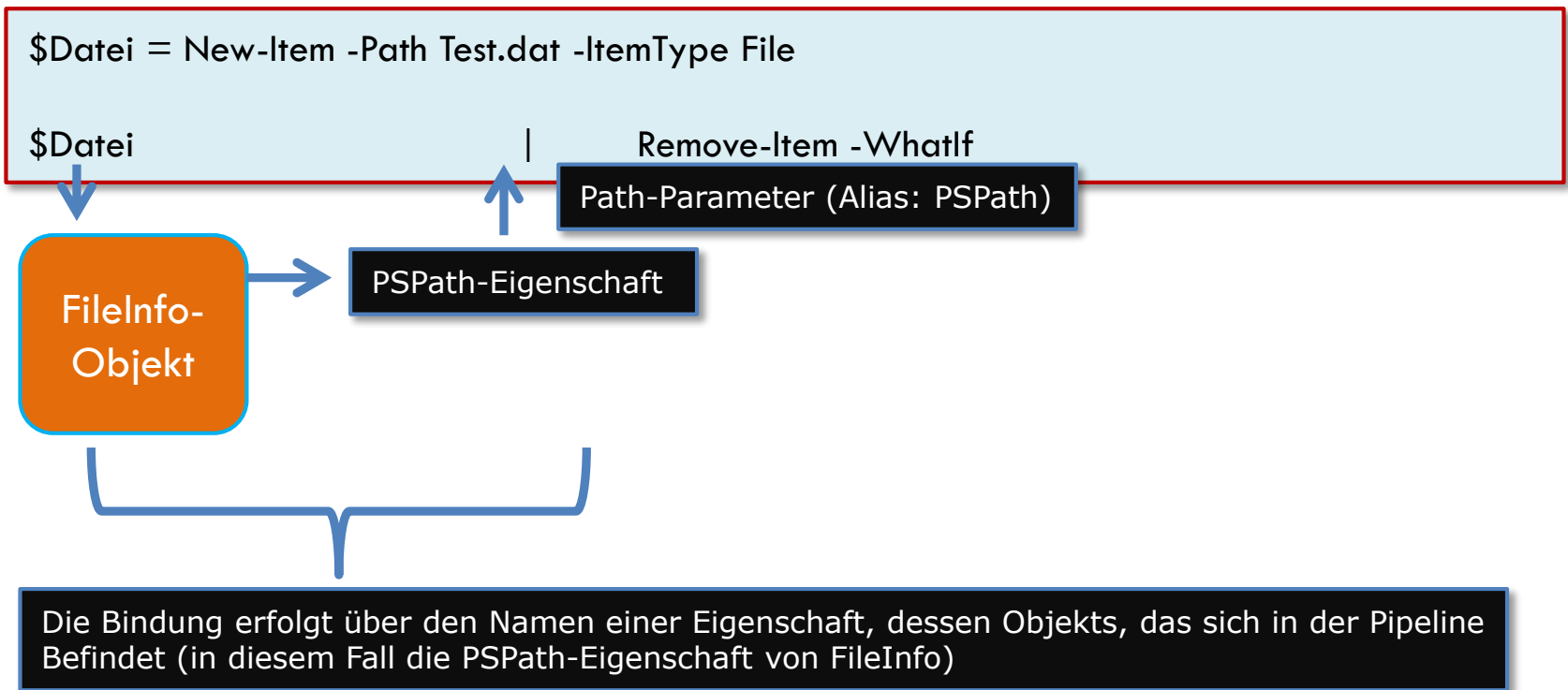
Parametername

Art der Parameterbindung, die möglich ist

# Parameterbindung per Name einer Eigenschaft

84

- Der Name der Eigenschaft des Objekts in der Pipeline liefert den Wert für den Parameter





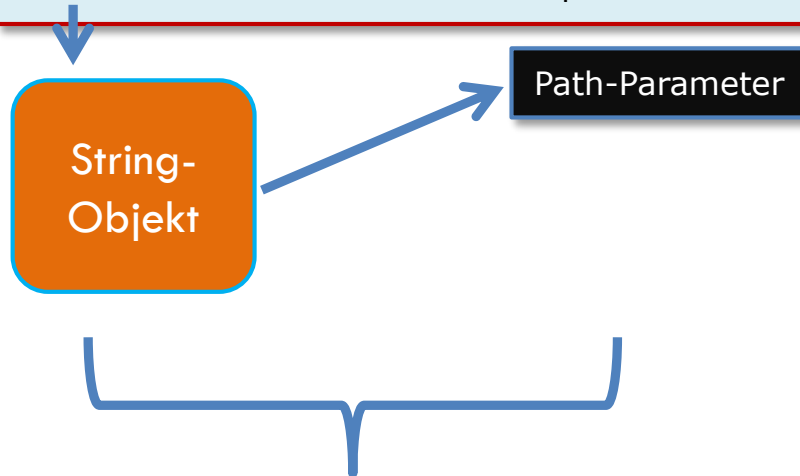
# Parameterbindung per Wert

85

- Der Wert in der Pipeline wird an den Parameter gebunden
- Es kann daher immer nur einen Parameter bei einem Cmdlet geben, der diese Form der Bindung anbietet

```
$Datei = New-Item -Path Test.dat -ItemType File
```

```
"Test.dat" | Remove-Item -WhatIf
```



Die Bindung erfolgt über den Wert, der sich in der Pipeline befindet

# Die Parameterbindung sichtbar machen

86

- Über das **Trace-Command-Cmdlet**
- Lehrreich, um das Prinzip der Parameterbindung besser nachvollziehen zu können

```
Trace-Command -Name ParameterBinding -PSHost -Expression { "Test.dat" | Remove-Item }
```

```
DEBUG: ParameterBinding Information: 0 : BIND NAMED cmd line args [Remove-Item]
DEBUG: ParameterBinding Information: 0 : MANDATORY PARAMETER CHECK on cmdlet [Remove-Item]
DEBUG: ParameterBinding Information: 0 : CALLING BeginProcessing
DEBUG: ParameterBinding Information: 0 : BIND PIPELINE object to parameters: [Remove-Item]
DEBUG: ParameterBinding Information: 0 : PIPELINE object TYPE = [System.String]
DEBUG: ParameterBinding Information: 0 : BIND arg [Test.dat] to parameter [Path]
DEBUG: ParameterBinding Information: 0 : Adding scalar element of type String to array position 0
DEBUG: ParameterBinding Information: 0 : BIND arg [System.String[]] to param [Path] SUCCESSFUL
DEBUG: ParameterBinding Information: 0 : CALLING EndProcessing
```

gekürzte Fassung

# Übung zum Thema Pipeline-Verarbeitung

87

- Ausgangspunkt ist eine Textdatei „Prozesse.txt“, die die Namen von Prozessen enthält (pro Zeile einen Namen)
- Der Aufruf funktioniert nicht ①
- **Aufgabe:** Welches Cmdlet muss „zwischengeschaltet“ werden, damit der Aufruf funktioniert und die Prozesse beendet werden?
- **Tipp:** Die einfachste Lösung hat etwas mit „CSV“ zu tun

①

```
Get-Content -Path .\Prozesse.txt | Stop-Process
```

# Zusammenfassung

88

- Alle Get-Commands geben Objekte zurück (die sich immer in ihrem Typ unterscheiden)
- Die PowerShell-Pipeline ist eine Objekt-Pipeline
- Parameterbindung = Ein Parameter erhält den aktuellen Inhalt der Pipeline als Eingabewert
- Es gibt **zwei** Bindungsarten:
  - ▣ Über eine Eigenschaft des Objekts (ByPropertyName)
  - ▣ Über den gesamten Wert (ByValue)

# Quiz (1)

89

- Welche Begriffe bezeichnen die vorhandenen Bindungsarten bei der PowerShell-Pipeline?

- a) Parameterpipelinebindung
- b) Objektparameterbindung
- c) Bindung über den Inhalt
- d) Bindung über den Namen einer Eigenschaft
- e) Bindung über den Wert

# Quiz (1)

90

☐ Antwort: c,d und e

# Functions und Advanced Functions

## Themenblock 6

# Die Themen

92

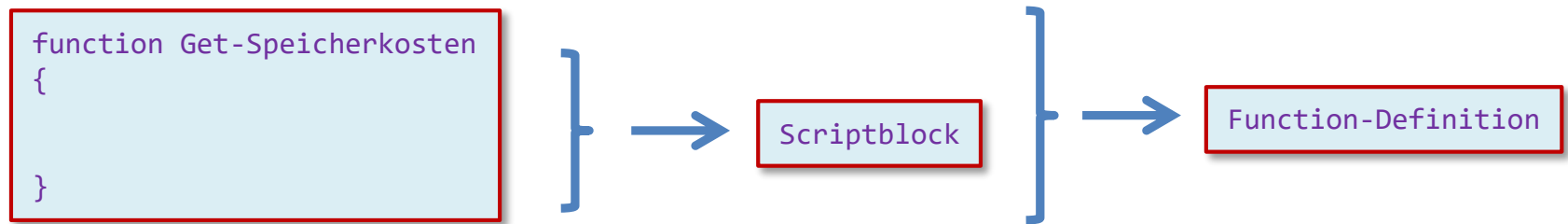
- Functions – eine kurze Wiederholung
- Function-Parameter
- Was macht eine Function „advanced“?
- Die Rolle der Attribute
- Das [Parameter]-Attribut
- Die Parameterbindung festlegen
- Attribute für die Parametervalidierung
- Das [CmdletBinding]-Attribut
- Die SupportsShouldProcess-Eigenschaft



# Functions – eine Wiederholung

93

- Eine Function ist ein Name, der für einen Scriptblock steht

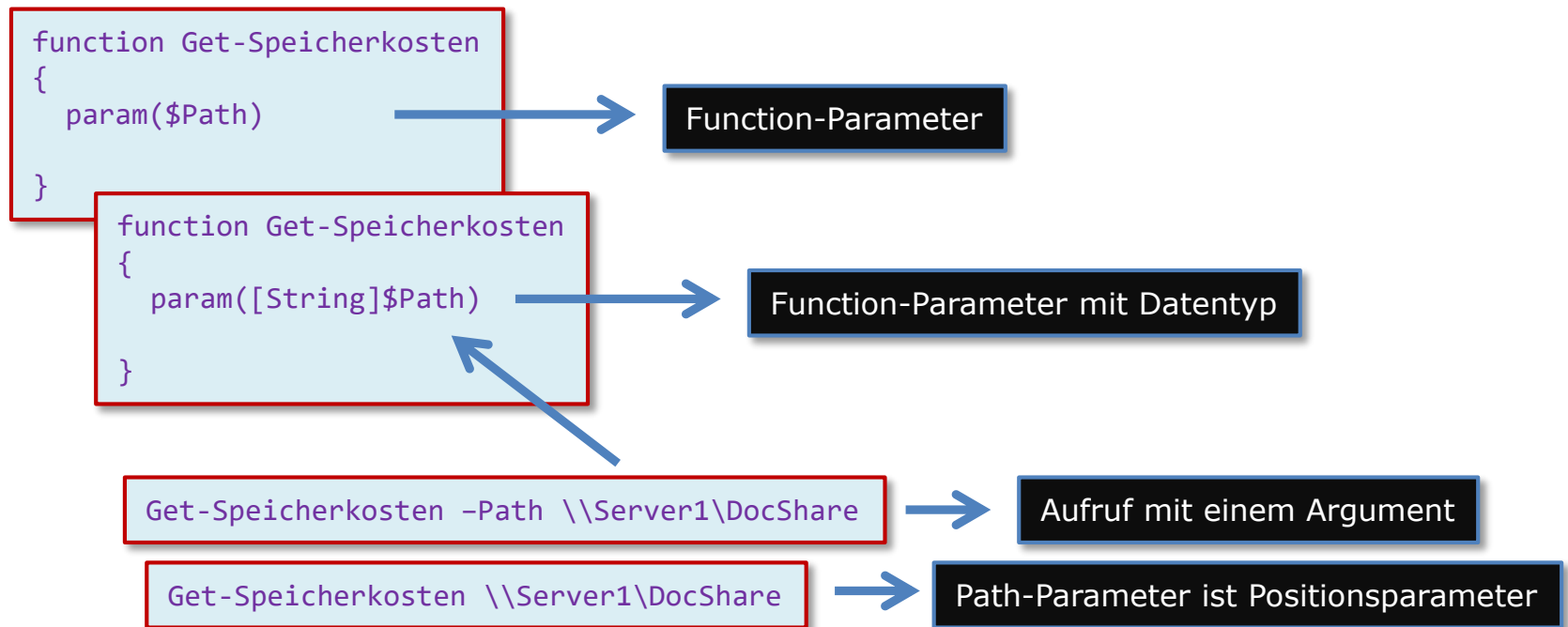


- Eine Function wird durch Eingabe des Namens aufgerufen
- Eine Function muss in einem Skript vor (!) ihrem Aufruf definiert werden

# Functions und ihre Parameter

94

- Functions besitzen im Allgemeinen Parameter
- Für jeden Parameter wird beim Aufruf der Function ein Wert (Argument) übergeben



# Was macht eine Function „advanced“?

95

- Der Begriff „advanced“ bezieht sich **ausschließlich** auf die Definition der Function, vor allem die ihrer Parameter, nicht auf ihren Inhalt
- Bei einer Advanced Function spielen die Formalitäten bezüglich der Parameter eine etwas größere Rolle
- Advanced Functions sind immer dann wichtig, wenn sich eine Function wie ein PowerShell-Cmdlet verhalten soll
- Wenn eine Function direkt aufgerufen wird, muss sie (natürlich) keine Advanced Function sein

# Advanced Functions in der Hilfe

96

- Das Thema „Advanced Functions“ ist in der Hilfe ausführlich und vollständig beschrieben

```
help about_Functions_Advanced
```

```
help about_Functions_Advanced_Parameters
```

```
help about_Functions_Advanced_Methods
```

# Ein Musterbeispiel für eine advanced Function

97

```
<#  
  .Synopsis  
    Ein leerer Rahmen für eine "advanced Function"  
#>  
function Muster-Beispiel  
{  
  [CmdletBinding(ConfirmImpact=<String>,  
                  DefaultParameterSetName=<String>,  
                  HelpURI=<URI>,  
                  SupportsPaging=<Boolean>,  
                  SupportsShouldProcess=<Boolean>,  
                  PositionalBinding=<Boolean>)]  
  param ([Parameter(...)] [Datentyp] $Parameter1)  
  
  Begin { }  
  
  Process { }  
  
  End { }  
}
```

# Die Rolle der Attribute

98

- Attribute sind allgemein ergänzende Informationen
- Bei der PowerShell ist ein Attribut ebenfalls ein Objekt mit Eigenschaften
- Bei der PowerShell werden die Attributnamen in eckige Klammern gesetzt, z.B. [Parameter]
- **Wichtig:** Ein Attribut bezieht sich immer auf ein anderes Element, z.B. einen oder alle Parameter einer Function

```
function Get-Speicherkosten
{
    param([Parameter(Mandatory=$true)][String]$Path)
}
```

Attributname

Attributeigenschaft

# Das [Parameter]-Attribut

99

- Erweitert die Definition eines Parameters um verschiedene Eigenschaften:
  - ▣ Mandatory (Pflichtparameter Ja/Nein)
  - ▣ ParameterSetName (Name des Parametersets, zu dem der Parameter gehört)
  - ▣ Position (Position des Parameters)
  - ▣ ValueFromPipeline (Art der Parameterbindung)
  - ▣ ValueFromPipelineByPropertyName (Art der Parameterbindung)
  - ▣ ValueFromRemainingArguments (der Parameter erhält die restlichen Argumente)
  - ▣ HelpMessage (Hilfetext)

# Die Parameterbindung festlegen

100

- Die Art der Parameterbindung wird über zwei Eigenschaften des [Parameter]-Attributs festgelegt:
  - ▣ ValueFromPipeline
  - ▣ ValueFromPipelineByPropertyName
- Beide geben kann, dass der Parameter seinen Wert auch (!) aus der Pipeline beziehen kann
- Bei **ValueFromPipelineByPropertyName** muss das Objekt in der Pipeline eine Eigenschaft besitzen, die dem Namen des Parameters entspricht
- **Tipp:** Über das [Alias]-Attribut erhält der Parameter einen Alias, der dem Namen der Eigenschaft entspricht, mit der eine Bindung möglich sein soll
- **Beispiel:** Ein Parameter mit dem Namen **Pfad** erhält **PSPath** als Alias



# Attribute für die Parametervalidierung

101

- ❑ Führen eine Validierung bei der Argumentzuordnung vor (!) dem Ausführen der Function durch
- ❑ **Vorteil:** Die Function wird nicht mit unpassenden Werten aufgerufen
- ❑ Werden in der Hilfe unter *about\_functions\_advanced\_parameters* beschrieben
- ❑ **Tipp:** In PowerShell 7.x gibt es weitere Validierungsattribute (u.a. *ValidateUserDrive* für einen Path-Parameter)

# Beispiele für Parameter-Validierung

102

- ❑ **[AllowNull]** - \$null-Werte sind explizit erlaubt
- ❑ **[ValidatePattern]** – Validierung des Wertes per Regex
- ❑ **[ValidateRange]** – erlaubter Bereich für Integer-Werte
- ❑ **[ValidateScript]** – Validierung des Wertes per Skript
- ❑ **[ValidateSet]** – es sind nur bestimmte Werte zugelassen
- ❑ **[ValidateUserDrive]** – ein Verzeichnispfad muss im Benutzerprofil liegen

```
function New-Password
{
    [CmdletBinding()]
    param([ValidateRange(8,16)[Int]$Length)
}
```

Minimum

Maximum

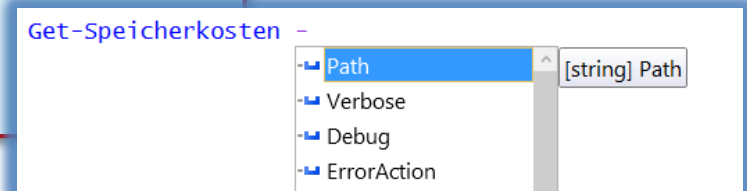
# Das [CmdletBinding]-Attribut

103

- Legt fest, dass die Parameterbindung einer Function wie bei einem Cmdlet durchgeführt wird:
  - Es sind keine Argumente erlaubt, die keinem Parameter zugeordnet werden können
  - Es stehen bei der Function die allgemeinen Parameter (ErrorAction, Verbose usw.) zur Verfügung
  - Geht dem param-Befehl voraus (dieser ist obligatorisch)
  - Wird unter **about\_Functions\_CmdletBindingAttribute** beschrieben

```
function Get-Speicherkosten
{
    [CmdletBinding()]
    param([Parameter(Mandatory=$true)][String]$Path)

}
```



# Eigenschaften von [CmdletBinding]

104

- Besitzt mehrere Eigenschaften:
  - ▣ ConfirmImpact=<String>
  - ▣ DefaultParameterSetName=<String>
  - ▣ HelpURI=<URI>
  - ▣ SupportsPaging=<Boolean>
  - ▣ SupportsShouldProcess=<Boolean>
  - ▣ PositionalBinding=<Boolean>

# Die SupportsShouldProcess-Eigenschaft

105

- Eine Eigenschaft von [**CmdletBinding**]
- Fügt die Parameter **Confirm** und **WhatIf** hinzu
- Setzt den **Confirm-Parameter** bei allen Cmdlets, die ihn anbieten, so dass jede Operation einzeln bestätigt werden muss
- Über **\$PSCmdlet.ShouldProcess()** wird eine explizite Bestätigung angefordert - für Befehle, ohne eingebautes Confirm

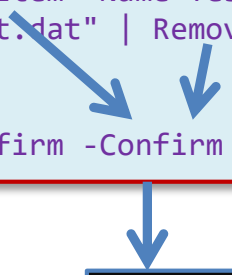
# Ein Beispiel für SupportsShouldProcess

106

## □ Ein Beispiel für **Confirm** bei Cmdlets

```
function Test-Confirm
{
    [CmdletBinding(SupportsShouldProcess=$true)]
    param()
    New-Item -Name Test.dat -ItemType File | Out-Null
    "Test.dat" | Remove-item
}

Test-Confirm -Confirm
```



Jedes Cmdlet mit Confirm-Parameter muss bestätigt werden

# SupportsShouldProcess bei eigenen Aktionen

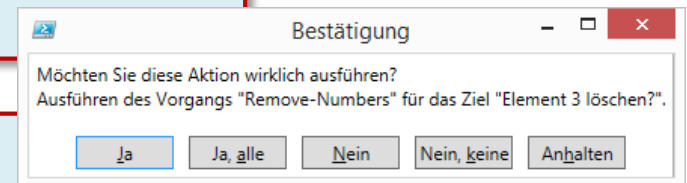
107

## □ Ein Beispiel für **Confirm** bei eigenen Aktionen

```
function Remove-Numbers
{
    [CmdletBinding(SupportsShouldProcess=$true)]
    param([Int[]]$BaseArray, [Int[]]$RemoveArray)
    foreach($z in $RemoveArray)
    {
        if ($PSCmdlet.ShouldProcess("Element $z löschen?"))
        {
            $BaseArray = $BaseArray -ne $z
        }
    }
    $BaseArray
}
```

```
$a = 1..10
$r = 3,5,7
```

```
Remove-Numbers -BaseArray $a -RemoveArray $r -Confirm
```



# Die Rolle von ConfirmImpact (1)

108

- Weitere Eigenschaft von **CmdletBinding()**
- Legt fest, ob eine Confirm-Bestätigung angefordert wird
- Mögliche Werte sind: High, Medium und Low
- Die Entscheidung wird immer im Vergleich zur **\$ConfirmPreference-Variablen** getroffen (Voreinstellung ist High)
- Spielt nur eine Rolle, wenn kein (!) **Confirm**-Parameter gesetzt wird
- In diesem Fall erfolgt eine Bestätigungsanforderung, wenn der bei **ConfirmImpact** angegebene Wert gleich oder höher als der Wert in **\$ConfirmPreference** ist



# Die Rolle von ConfirmImpact (2)

109

## □ Beispiel für die **ConfirmImpact**-Eigenschaft

```
function Remove-Number
{
    [CmdletBinding(SupportsShouldProcess=$true, ConfirmImpact="Medium")]
    param([Int[]]$BaseArray, [Int[]]$RemoveArray)
    foreach($z in $RemoveArray)
    {
        if ($PSCmdlet.ShouldProcess("Element $z löschen?"))
        {
            $BaseArray = $BaseArray -ne $z
        }
    }
    $BaseArray
}
```

`$ConfirmPreference="Medium"`

Entfernen muss bestätigt werden

`$ConfirmPreference="High"`

Keine Bestätigungsanforderung

# Abarbeiten der Pipeline (1)

110

- Jede Function besteht aus drei Scriptblöcken
  - ▣ **Begin** - wird vor der Pipeline-Abarbeitung 1 x ausgeführt
  - ▣ **Process** – wird für jedes Objekt in der Pipeline ausgeführt
  - ▣ **End** – wird am Ende der Pipeline-Abarbeitung 1 x ausgeführt
- Damit kann in die Function „gepiped“ werden
- **Begin** und **End** sind optional
- **Wichtig:** Wird der **process**-Block verwendet, kann die Function keine Befehle außerhalb eines **begin**-, **process**- oder **end**-Blocks enthalten

# Abarbeiten der Pipeline (2)

111

## □ Ein Beispiel für eine Pipeline-Function

```
<#  
  .Synopsis  
  Beispiel für eine Pipeline-Function  
#>
```

# Pipeline abarbeiten per Skript

112

## □ Auch Skripte können die Pipeline abarbeiten

```
<#  
.Synopsis  
Beispiel für ein Pipeline-Skript  
#>  
param([Parameter(ValueFromPipelineByPropertyName=$true)][Alias("PSPath")][String]$Path,  
      [Double]$SpeicherkostenMB=0.8)  
process  
{  
    Get-ChildItem -Path $Path -Directory -Recurse | ForEach {  
        $GroesseMB = (Get-ChildItem -Path $_.FullName -File |  
            Measure-Object -Property Length -Sum).Sum / 1MB  
        $Speicherkosten = $GroesseMB * $SpeicherkostenMB  
        New-Object -TypeName PSObject -Property @{Pfad=$_.FullName;  
                                                    Speicherkosten=$Speicherkosten}  
    }  
}
```

```
Get-ChildItem -Path C:\2017 | SpeicherKosten.ps1
```

# Übung zum Thema Advanced Functions (1)

113

- Ausgangspunkt ist die Function **Get-StorageCost**, welche die Speicherkosten für ein Verzeichnis berechnet (zu finden im Übungsordner)
- Wie kann die Function „pipelinefähig“ gemacht werden, so dass der Aufruf in möglich ist?
- **Tipp:** Es geht um den **Path**-Parameter
- **Hinweis 1:** Einem Parameter kann per `[Alias("Aliasname")]` ein Aliasname gegeben werden
- **Hinweis 2:** Damit eine Function die Pipeline vollständig abarbeitet, benötigt sie einen Process-Block

# Übung zum Thema Advanced Functions (2)

114

- **Aufgabe:** Erstellen einer Function mit dem Namen „Choose“
- Der Function wird ein Array übergeben, die Rückgabe ist ein per „Zufallsgenerator“ (Get-Random) ausgewähltes Element
- **Erweiterung:** Über einen weiteren Parameter (Anzahl) kann die Anzahl der Elemente ausgewählt werden, der Defaultwert soll 1 sein
- **Frage:** Muss überprüft werden, ob die Anzahl die Größe des Array übersteigt?

# Zusammenfassung

115

- Eine Advanced Function besitzt eine erweiterte Parameter-Deklaration
- Parameter-Attribute – Erweitern eine Parameter-Deklaration
- **[Parameter(Mandatory=\$true)]** – Parameter wird zum Pflichtparameter
- Über das **[Parameter]**-Attribut wird ein Parameter pipeline-bindungsfähig
- Auch eine Parametervalidierung ist über Attribute möglich

# Quiz (1)

116

- Welche Schreibweise für die Festlegung eines Pflichtparameters ist korrekt?

- a) `[Mandatory=$true]$Path`
- b) `[Parameter.Mandatory=$true]$Path`
- c) `[Parameter(Mandatory=$true)]$Path`
- d) `[Parameter(Mandatory=$true)][String]$Path`



# Quiz (2)

117

☐ Antwort: c und d

# Umgang mit Modulen

## Themenblock 7

# Was ist ein Modul?

119

- Ein Modul ist ein Verzeichnis, dessen Inhalt beim Importieren des Moduls in die PowerShell-Sitzung geladen wird
- Ein Modulverzeichnis enthält verschiedene Dateitypen (in der Regel Psm1-, Psd1-, Psxml1- und Dll-Dateien)
- Ein Modulverzeichnis kann sich in einem beliebigen Verzeichnis befinden
- Damit ein Modul implizit geladen werden kann, muss die Umgebungsvariable **\$PSModulePath** den Pfad des Elternverzeichnisses enthalten

# Module laden

120

- Ein Modul wird in der Regel implizit geladen, z.B. durch Ausführen einer Function, die in einem der Ps1-Dateien im Modulverzeichnis enthalten ist
- Voraussetzung ist, dass das Elternverzeichnis über **\$PSModulePath** gefunden werden kann
- Ansonsten wird ein Modul über **Import-Module** direkt geladen
- Per **RequiredVersion**-Parameter wird eine bestimmte Version eines Moduls geladen

# Auflisten der verfügbaren Module

121

- **Get-Module** listet nur die geladenen Module auf
- Der Parameter **ListAvailable** listet alle verfügbaren Module auf

```
Get-Module -Name ActiveDirectory -ListAvailable
```



Modul verfügbar?

# Modultypen

122

- Es gibt mehrere Modultypen:
  - ▣ Skriptmodule (Verzeichnis enthält .psm1-Datei)
  - ▣ Manifestmodule (Verzeichnis enthält .psd1-Datei)
  - ▣ Binäre Module (besteht aus einer einzelnen DLL-Datei, z.B. mit Cmdlets)
  - ▣ Dynamische Module (existieren nur temporär)
- In der Praxis spielen nur Skriptmodule und Manifestmodule eine Rolle
- Das Anlegen eines Moduls ist grundsätzlich einfach und setzt keine bis wenige Detailkenntnisse voraus

# Anlegen eines Skriptmoduls (1)

123

- **Vorteil:** Eine vorhandene Ps1-Datei mit Functions muss lediglich in eine Psm1-Datei umbenannt und in ein leeres Modulverzeichnis kopiert werden
- **Nachteil:** Es gibt keine Metadaten, z.B. Versionsnummer

# Anlegen eines Skriptmoduls (2)

124

- Im einfachsten Fall enthält das Modulverzeichnis eine Psm1 - Datei, die eine Reihe von Functions enthält
- Wichtig: Name der Psm1 -Datei muss dem Verzeichnisnamen entsprechen
- In zwei Schritten zum Skriptmodul:
  - ▣ Neues Verzeichnis anlegen (z.B. unter `$env:userprofile\documents\windowspowershell\modules`)
  - ▣ Psm1 -Datei in diesem Verzeichnis anlegen oder eine vorhandene Ps1 -Datei mit Functions als Psm1 -Datei in das Verzeichnis kopieren
- Eine Ps1 -Datei kann auch in der Psm1 -Datei dot-sourced ausgeführt werden, damit ihre Functions über das Modul zur Verfügung stehen



# Anlegen eines Manifestmoduls (1)

125

- Ein Manifestmodul wird über eine Manifestdatei (Erweiterung .Psd1) beschrieben
- Die Manifestdatei wird am einfachsten über das **New-ModuleManifest**-Cmdlet angelegt
- Die meisten Einträge sind optional
- Wichtige Einträge sind *ModuleVersion* und *RootModule* bzw. *NestedModule*
- Über *NestedModules* kann eine Psm1-Datei ausgewählt werden – damit wird aus einem Skriptmodul mit wenig Aufwand ein Manifestmodul, das eine Versionsnummer enthält

# Anlegen eines Manifestmoduls (2)

126

- Schritt 1: Anlegen eines Modulverzeichnis
- Schritt 2: Anlegen der Manifestdatei per **New-ModuleManifest**
- Schritt 3: Editieren der Psd1-Datei (Modulmanifestdatei)

# Automatische Modulverwaltung

127

- ❑ Mit den Functions im Modul PowerShellGet wird das Hinzufügen von Modulen aus einer Ablage (Repository) sehr einfach
- ❑ **Find-Module** findet Module, **Install-Module** fügt ein Modul lokal hinzu
- ❑ Das Standard-Repository ist die PowerShell Gallery
  - ▣ <https://powershellgallery.com>
- ❑ Es lassen sich mit wenig Aufwand eigene Repositories anlegen (z.B. im Intranet oder in der Cloud)

Beispiel: Laden eines Moduls von der PSGallery

```
Install-Module -Name Carbon
```

# Die TLS-Problematik

128

- ❑ Die PowerShell-Gallery war in der Vergangenheit zeitweise lahmgelegt
- ❑ Grund war eine Abhängigkeit der PowerShell von altem TLS-Standard
- ❑ Der Fehler sollte zwar nicht mehr auftreten...
- ❑ Workaround muss in Profilskript abgelegt werden
- ❑ Eventuell müssen auch abgelaufene Zertifikate ignoriert werden

Umstellen auf TLS12

```
[Net.ServicePointManager]::SecurityProtocol = [Net.SecurityProtocolType]::Tls12
```

Ungültige Zertifikate ignorieren

```
[Net.ServicePointManager]::ServerCertificateValidationCallback = {$true};
```

# Module veröffentlichen

129

- ❑ Veröffentlichung als Package in einem Repository per **Publish-Module-Cmdlet**
- ❑ Das Repository kann privat oder öffentlich sein
- ❑ Das gesamte Modulverzeichnis wird in eine Package-Datei (Erweiterung *.Package*, intern ZIP-Format) konvertiert und in das Repository übertragen
- ❑ Für die Veröffentlichung muss die Manifestdatei zusätzliche Metadaten (GUID, author und description) enthalten

# Zusammenfassung

130

- Ein Modul ist bei der PowerShell ein Verzeichnis
- Enthält entweder eine Psm1-Datei (Skriptmodul) oder eine Psd1-Datei (Manifestmodul)
- Es gibt vorgesehene Modulverzeichnisse (\$PSModulePath-Umgebungsvariable)
- Der wichtigste Vorteil von Manifestmodulen sind die Metadaten, in erster Linie die Versionierung

# Übung zum Thema Module

131

- Aufgabe: Anlegen eines Manifestmoduls mit dem Namen „PsKurs“
- Im Übungsordner befinden sich die Dateien
  - ▣ Pskurs.psm1 mit drei Functions Get-ComputerInfo, Get-OSInfo und Get-AppInfo
  - ▣ PsKursExtras.ps1 mit der Get-PCInfo
- **Ziel:** Das Ausführen von Get-PCInfo soll direkt möglich sein
- Wie muss die Psd1-Datei aufgebaut sein?

# Arrays und Hashtables

## Themenblock 8



# Die Themen

133

- Arrays sind Listen mit beliebigen Elementen
- Hashtables speichern Schlüssel-Wert-Paare
- Umgang mit Hashtables
- Tipp: GetEnumerator()

# Arrays fassen mehrere Werte zusammen

134

- Fast alle Get-Commands geben Arrays zurück
- Im einfachsten Fall ist ein Array mehrere per Komma getrennte Werte in runden Klammern
- Element wird über Index in eckigen Klammern angesprochen

Beispiel: Zusammenfassen von Werten zu einem Array

```
$a1 = (1234,[DateTime]::Now,(Get-Process -ID $PID))
```

```
$a1 = @(1234,[DateTime]::Now,(Get-Process -ID $PID))
```

```
$a1 = @()  
$a1 += 1234  
$a1 += [DateTime]::Now  
$a1 += (Get-Process -ID $PID)  
$a1[0]  
1234
```

# Arrays direkt anlegen

135

- Array ist eine abstrakte Klasse, d.h. New-Object oder die statische New-Methode gehen nicht
- Die Schreibweise für mehrdimensionale Arrays ist etwas gewöhnungsbedürftig

Beispiel: Zweidimensionales Array

```
$a2 = New-Object -Typename "Byte[,]" -ArgumentList 4,2
$a2[0,0] = 1
$a2[0,1] = 3
$a2.GetUpperBound(0)
1
$a2.GetUpperBound(1)
3
```

# Hashtable = Array mit Schlüssel-Wert-Paaren

136

- Eine Hashtable ist nur ein Array mit Schlüssel=Wert-Paaren statt Werten
- @{} statt @[]
- Schlüssel = beliebiger Wert, der eindeutig sein muss
- Wert = beliebiger Wert, der in einer Liste abgelegt werden soll
- Vorteil gegenüber einem Array
  - Jeder Wert wird über einen individuellen Schlüssel angesprochen
  - Sehr viel schneller Zugriff nach bestimmten Werten, da keine Suche erforderlich ist

# To hash = „zerhacken“

137

- ❑ Begriff stammt aus der Informatik
- ❑ Die deutsche Bezeichnung ist „Streuwert“ (Wikipedia)
- ❑ Jedem Wert der Hashtable wird intern ein Hashwert zugeordnet, der von einer Hashfunktion gebildet wird
- ❑ Am Ende ist der Hashwert nur eine Zahl
- ❑ Gute Beschreibung:

<https://docs.microsoft.com/en-us/powershell/scripting/learn/deep-dives/everything-about-hashtable?view=powershell-7.2>

# Hashtables statisch anlegen

138

- Die Schreibweise ist `@{key1=value1;key2=value2}`
- Leere Hashtable mit `@{}`

Beispiel: Hashtable mit drei Werten

```
$h = @{k1=100; k2=200; k3=300}
```

```
$h["k1"]  
100  
$h["k3"]  
300  
$h.k2  
200  
$h[k1]  
!!! Fehler !!!
```

```
$h.k3 = "400"
```

# Hashtables dynamisch anlegen

139

- Schlüssel-Wert-Paare können auch dynamisch hinzugefügt werden

Beispiel: Schlüssel = Monatsname Value= Anzahl Tage pro Monat

```
PS C:\Users\pemo20> $h = @{}
PS C:\Users\pemo20> for($m=1;$m-le12;$m++)
>> {
>>   $h[(Get-Date -Month $m -Format MMMM)] = [DateTime]::DaysInMonth(2012, $m)
>> }
PS C:\Users\pemo20> $h
```

Name	Value
----	-----
Mai	31
Juni	30
Juli	31
Dezember	31
Januar	31
April	30
September	30
Oktober	31
November	30
Februar	29
August	31
März	31

# Die Schlüssel einer Hashtable sind nicht sortiert

140

- ❑ **Lösung:** `[Ordered]` beim Anlegen verwenden
- ❑ Anstelle eines *Hashtable*- wird ein *OrderedDictionary*-Objekt angelegt

```
PS C:\Users\pemo20> $h = [Ordered]@{}
PS C:\Users\pemo20> for($m=1;$m-le12;$m++)
>> {
>>   $h[(Get-Date -Month $m -Format MMMM)] = [DateTime]::DaysInMonth(2012, $m)
>> }
PS C:\Users\pemo20> $h
```

Name	Value
----	-----
Januar	31
Februar	29
März	31
April	30
Mai	31
Juni	30
Juli	31
August	31
September	30
Oktober	31
November	30
Dezember	31



# Credential-Verwaltung über eine Hashtable

141

- Schlüssel = Servername/IP-Adresse
- Wert = *PSCredential*-Objekt

Beispiel: Credential-Zuordnung über eine Hashtable

```
$h = @{}  
$Cred1 = Get-Credential  
$Cred2 = Get-Credential  
$Cred3 = Get-Credential  
$h["Server1"] = $Cred1  
$h["Server2"] = $Cred2  
$h["Server3"] = $Cred3  
  
foreach($Server in $h.keys)  
{  
    Invoke-Command -Computername $Server -Scriptblock {ipconfig} -Credential $h.$Server  
}
```

# Sehr praktisch: GetEnumerator()

142

- Egal, wie viele Einträge eine Hashtable enthält, es ist immer ein einzelnes Objekt
- Sollen alle Key-Value-Paare als einzelne Objekte behandelt werden, muss ein *GetEnumerator()*-Aufruf angehängt werden

```
$h = @{}  
$h["Server1"] = 0  
$h["Server2"] = 1  
$h["Server3"] = 3
```

Beispiel: Geht nicht

```
$h | Where-Object Name -eq "Server1"
```

Beispiel: Geht

```
$h.GetEnumerator() | Where-Object Name -eq "Server1"
```

# Praxistipp: PSCustomObject in HashTable konvertieren

143

- Kann manchmal praktisch sein;)
- Viele gute Beispiele:  
<https://stackoverflow.com/questions/3740128/pscustomobject-to-hashtable>

```
$obj = [PSCustomObject]@{p1=100;p2=200;p3=300}  
  
$obj.psobject.properties | ForEach -Begin {$h=@{}} -Process  
{$h."$($_.Name)" = $_.Value} -End {$h}
```

# Hashtables in der Praxis

144

- Hashtables kommen in der PowerShell-Praxis an mehreren Stellen vor
  - ▣ Beim Bilden von Properties bei *Select-Object*
  - ▣ Beim Zusammenfassen mehrerer Parameterargumente (Stichwort: Splatting)
  - ▣ Als Parameterwert bei einigen Cmdlets (z.B. *New-Object*, *Invoke-WebRequest*, *Select-Xml*)
  - ▣ Beim Bilden von Objekten im Zusammenspiel mit dem Type Alias *[PSCustomObject]*

# Zusammenfassung

145

- Eine Hashtable ist eine Liste mit Schlüssel=Wert-Paaren
- Hashtables sind praktisch in vielen Situationen
- Der Begriff „hash“ stammt aus der Informatik
- Für die PowerShell-Praxis spielen Hashtables eine wichtige Rolle
- Das Prinzip ist einfach, man versteht es trotzdem selten beim ersten Mal, daher unbedingt dranbleiben 😊

# Übung zum Thema Hashtable

146

- Auf einem Zettel sind eine Reihe von Servernamen und Benutzerkonten aufgeschrieben
- Jeder Server besitzt eine Reihe von Benutzernamen
- Aufgabe: Die Daten sollen so mit Hilfe einer Hashtable umgesetzt werden, dass über den Servernamen die Namen aller Benutzerkonten, die dem Server zugeordnet sind, abgerufen werden

# Texte verarbeiten

## Themenblock 9

# Die Themen

148

- Objekte nach CSV, HTML, JSON und XML konvertieren
- Aus Text Objekte machen
- Kurze Einführung in reguläre Ausdrücke
- Textdaten aus dem Web verarbeiten



# Objekte nach CSV, HTML, JSON und XML konvertieren

149

## □ Eine der Stärken der PowerShell

Beispiel: Objekte als Text

```
$ProcData = Get-Process | Where-Object WS -gt 200MB | Select-Object -Property Name,StartTime,WS  
$ProcData | ConvertTo-CSV  
$ProcData | ConvertTo-HTML  
$ProcData | ConvertTo-JSON  
$ProcData | ConvertTo-XML -As String
```

# Aus Text Objekte machen

150

- ❑ Import-CSV macht aus Text Objekte
- ❑ Voraussetzung ist eine Unterteilung der Zeilen durch ein einheitliches Trennzeichen (Delimiter)
- ❑ Umlaute per Encoding-Parameter berücksichtigen

Beispiel: Textdatei Serverdaten.txt

```
Fujitsu_Primergy_RX300,EDV,2020-04-29  
Fujitsu_Primergy_RX350,EDV,2020-04-24  
HP_ProLiant_BL680,SUP,2020-03-20  
DELL_PowerEdge_R620,SUP,2020-03-19  
Lenovo_x3650,BUCH,2020-03-18  
DELL_PowerEdge_R640,SUP,2020-03-19  
DELL_PowerEdge_R640,OFF,2020-02-14  
HP_ProLiant_BL500,OFF,2020-03-07  
HP_ProLiant_BL440,SUP,2020-03-15  
Lenovo_x450,BUCH,2020-04-18  
Dell_PowerEdge_T110,OFF,2020-03-12  
Dell_PowerEdge_T320,OFF,2020-03-14
```

```
Import-CSV -Path .\Serverdaten.txt -Header "Servertyp", "Abteilung", "Datum"
```

# Reguläre Ausdrücke (1)

151

- Wirken kompliziert, sind es im Allgemeinen aber nicht
- Ein regulärer Ausdruck beschreibt ein allgemeines Muster, mit dem Texte durchsucht werden
- Jeder Treffer ist ein Match
- Beispiel NetStat-Ausgabe in Objekte konvertieren

```
"^\\s+TCP\\s+([0-9]{1,3}\\.[0-9]{1,3}\\.[0-9]{1,3}\\.[0-9]{1,3}):\\s+(\\d+).\\s+(\\d+)$"
```

- Einfach CoPilot fragen
- In der PowerShell-Hilfe gut erklärt

```
help about_regular_expressions
```

# Reguläre Ausdrücke (2)

152

- Select-String-Cmdlet
- Operatoren `-match` und `-notmatch` – das Ergebnis ist in der Variablen `$Matches` enthalten
- [Regex] Type Accelator mit `Match()` und `Replace()`

# Kleines Einmaleins der regulären Ausdrücke

153

Sonderzeichen	Steht für...
.	Beliebiges Zeichen
*	Kein mal, einmal oder mehrfach
+	Mindestens einmal
\w	Buchstabe, Ziffer, bestimmte Sonderzeichen
\d	Ziffer
\s	Whitespace, z.B. Leerzeichen
[]	Zusammenfassung mehrerer Ausdrücke (z.B. [\w+-0-9])
()	Gruppe (z.B. (\w+)_(\w+))
\	Escape-Zeichen (z.B. für runde Klammern, "\\(\\d\\+\\d\\)")
{n,m}	Mindestens n, maximal m Zeichen

# Beispiel: Logdateien auswerten

154

- Aus einer Webserverlogdatei sollen die IP-Adressen herausgezogen werden

```
$LogPfad = "WebserverLogs\*.log"

$Muster = "[0-9]{2,3}\.[0-9]{2,3}\.[0-9]{2,3}\.[0-9]{2,3}"
Select-String -Path $LogPfad -Pattern $Muster | Select-Object -Property @{n="IP-Adresse";e={$_.Matches[0].Value}},
                                                                    @{n="Datei";e={$_.Filename}},
                                                                    @{n="Zeile";e={$_.LineNumber}}
```



Erster Treffer

# Beispiel: Alle Übereinstimmungen finden

155

## □ Finden mehrerer Treffer per [Regex]::Matches()

```
$Text = @"
Deutscher Meister wird nur 1860 München, nur 1860 München. Deutscher Meister wird nur 1893 Bayern München, 1893
Bayern München. Deutscher Meister wird nur 1899 Hoffenheim
"@

$Text -match "\d{4}"

# Nur ein Treffer
$Matches

# Alle Treffer als Strings
$Text -split "[.,,]"
# Nur ein Treffer
$Matches

# Alle Treffer
[Regex]::Matches($Text, "\d{4}")
```

# Beispiel: E-Mail-Adressen

156

## □ Durchsuchen von Html-Dateien per Invoke-WebRequest

```
$Muster = "\b[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,}\b"

$Inhalt = (Invoke-WebRequest -Uri $Url -ErrorAction Ignore).Content
[Regex]::Matches($Inhalt, $Muster, "IgnoreCase") | Select-Object @{n="URL";e={$Url}}, @{n="E-Mail";e={$_.Value}}
```



# Zusammenfassung

157

- Umgang mit Textdaten ist eine der Stärken der PowerShell
- Objekte in Textdaten konvertieren
- Import-CSV macht aus Text Objekte
- Textdaten ohne feste Struktur werden durch Regexe zerlegt

# Übung zum Thema reguläre Ausdrücke

158

- Ausgangspunkt ist eine Textdatei mit mehreren Zeilen
- Jede Zeile besteht aus einem Text und einer Ziffernfolge
- Beide sollen per Regex getrennt werden
- **Tipp:** *Select-String* mit dem Parameter *-AllMatches*

```
$Text = @"  
Server123  
PC456  
Computer99  
"@
```

# Skripte debuggen

## Themenblock 10

# Die Themen

160

- Der PowerShell-Debugger im Überblick
- Debug-Cmdlets
- Der integrierte Debugger der PowerShell ISE
- Haltepunkte von Bedingungen abhängig machen
- Die `#requires`-Direktive
- Regeln für gute Skripte
- Der Script Analyzer von Microsoft

# Der PowerShell-Debugger im Überblick

161

- Ermöglicht das Setzen von Haltepunkten in einer PS1-Datei und das schrittweise Ausführen eines Skriptes
- Mit der Version 5.0 der Windows PowerShell wurden wichtige Verbesserungen eingeführt:
  - ▣ In den Debugger unterbrechen (z.B. über [Strg]+[Break] in der Konsole)
  - ▣ Debuggen von Background-Jobs
  - ▣ Debuggen von lokalen und Remote Runspaces
- Visual Studio Code bietet mehr Möglichkeiten als die Konsole

# Debugger-Cmdlets

162

- ❑ Set-PSBreakPoint/Get-PSBreakPoint
- ❑ Enable-PSBreakPoint/Disable-PSBreakPoint
- ❑ Remove-PSBreakPoint
- ❑ Im Debug-Modus ([DBG]) kann der Debugger in der Konsole über Kommandos gesteuert werden (ein ? zeigt alle Kommandos an)
- ❑ **Set-PSDebug** – Debugger global ein/ausschalten

# In den Debugger unterbrechen

163

- ❑ Ein ausführendes Skript kann jederzeit so unterbrochen werden, dass es in den Debugmodus übergeht
- ❑ Sehr praktisches Feature
- ❑ Per [Strg]+[Break] in der Konsole
- ❑ Per [Strg]+[B] in der PowerShell ISE
- ❑ Per [F6] in Visual Studio Code

# Haltepunkte von Bedingungen abhängig machen

164

- Ein Haltepunkt kann von einer Bedingung (z.B. der Wert einer Variablen ändert sich) abhängig gemacht werden
- Ausgangspunkt ist der Action-Parameter von des **Set-PSBreakPoint**-Cmdlets, mit dem ein Haltepunkt gesetzt wird

Beispiel: Haltepunkt soll aktiv werden, sobald \$z > 90 wird

```
Set-PSBreakpoint -Script .\HaltepunktBedingung.ps1 -Line 6 -  
Action { if ($z -gt 90) { break } }
```



# Das Set-PSDebug-Cmdlet

165

- Aktiviert generell den Debug-Modus für die Konsole und für Skripts
- Trace-Parameter (Werte 0,1 und 2)
- Step-Parameter
- Strict-Parameter
- Off-Parameter

# Der Debugger bei Visual Studio Code

166

- Vertraute Tastatur-Shortcuts
- Mehr Komfort als in der Konsole
  - ▣ Fenster für die Werte der Variablen
  - ▣ Bedingung für einen Haltepunkt kann flexibel gesetzt werden
  - ▣ Klare Optik
- Guter Überblick von Keith Hill (PowerShell MVP):
  - ▣ <https://devblogs.microsoft.com/scripting/debugging-powershell-script-in-visual-studio-code-part-1>
  - ▣ <https://devblogs.microsoft.com/scripting/debugging-powershell-script-in-visual-studio-code-part-2>

# Weitere Tipps zum Thema Debuggen

167

- Per PSEdit kann in einer Remote-Session eine Datei auf dem Remote-Computer editiert werden
  - ▣ Praktisch für das Editieren von Ps1-Dateien, die remote ausgeführt werden
- Etwas Fortgeschrittenere Themen
  - ▣ Runspace-Debugging
  - ▣ Debugger an einen beliebigen Prozess anhängen, in dem PowerShell-Befehle ausgeführt werden

# Zusammenfassung

168

- ❑ Debugger ermöglicht die schrittweise Ausführung eines Skripts
- ❑ Der Debugger muss in einem PowerShell-Host implementiert werden
- ❑ [F9]-Taste schaltet einen Haltepunkt um
- ❑ Bedingte Haltepunkte halten an, wenn eine Bedingung erfüllt ist
- ❑ In VS Code kann ein laufendes Skript per [F6] in den Debug-Modus versetzt werden

# Tipps für die Praxis

## Themenblock 11

# Themen

170

- ❑ Die „unsichtbare“ PsObject-Eigenschaft
- ❑ Listen statt Arrays
- ❑ Keine Strings in Schleifen zusammensetzen
- ❑ Große Dateien nicht per **Get-Content** einlesen
- ❑ Regex statt **Where-Object**
- ❑ Pipeline abbrechen mit **Select-String** und dem **First**-Parameter
- ❑ Parameter-Splatting
- ❑ Enumerationen mit dem **enum**-Befehl
- ❑ Umgang mit SymLinks
- ❑ Vergleiche mit \$null

# Die „unsichtbare“ psobject-Eigenschaft

171

- Jedes PowerShell-Objekt besitzt eine Eigenschaft **PSObject**
- Liefert ein Objekt, das die „Struktur“ des Objekts beschreibt
- Konrekt Members, Methods, Properties und TypeNames
- **Tipp:** Auflisten mit Get-Member -Force
- Bei einem Type-Objekt ist **PSObject** nicht erforderlich, da das RuntimeType-Objekt eigene Members anbietet

Beispiel: Auflisten der Konstruktoren eines Typs mit ihren Parametern

```
[PSCredential].GetConstructors() | % -Begin { $i=0} -Process {  
$i++; "Konstruktor $i :`n"; $_.GetParameters() | % { "Name: $_.Name"  
- Typ: $_.ParameterType" }}
```



Liefert das RuntimeType-Objekt, das den Typen System.Management.Automation.PSCredential beschreibt

# Listen statt Arrays

172

- Für großen Datenmengen sind Listen schneller als Arrays
- Listen müssen über die Typbezeichnung und die statische Methode `New()` angelegt werden
- Ein Beispiel ...



# Keine Strings in Schleifen zusammensetzen

173

- Performance-Killer Nr. 1
- Schuld ist der Umstand, dass Strings bei .NET „unzerstörbar“ (unveränderbar) sind (engl. „immutable“)
- Ein Beispiel ...

# Große Dateien nicht per Get-Content einlesen

174

- Bei sehr großen Textmengen kann das Einlesen über einen StreamReader aus der .NET Runtime schneller sein
- **Get-Content** ist aber nicht langsam, es gibt lediglich etwas mehr „Overhead“
- Ein Beispiel ...

# Regex statt Where-Object

175

- Wenn es um Performance geht, sollte die Pipeline bei großen Datenmengen vermieden werden
- Sollen z.B. große Textmengen durchsucht werden, kann die Verarbeitung per [**Regex**] deutlich performanter sein als ein **Where-Object** mit match-Operator
- **Nachteil:** Reguläre Ausdrücke sind etwas „speziell“
- Ein Beispiel ...

# Pipeline abbrechen mit **Select-Object** und dem **First-Parameter**

176

- Die Pipeline wird normalerweise komplett abgearbeitet
- Bei sehr großen Datenmengen wäre eine Begrenzung praktisch
- Der **First-Parameter** von **Select-Object** holt nur die angegebene Zahl an Objekten
- Ein Beispiel ...

# Pipeline extrem

177

- ❑ Einlesen einer sehr großen Textdatei (ca. 1.5 GB)
  - ▣ Im Material-Ordner die Datei Countries.txt
- ❑ Nach dem Einlesen gibt es ein Array mit ca. 11 Millionen Einträgen
- ❑ Ein Select-Object –First 10 geht sehr schnell, ein Select-Object –Last 10 dauert "ewig"

# Parameter-Splatting (1)

178

- Zusammenfassen mehrerer Parameterwerte in einer Hashtable
- Muster: `Parametername=Wert;Parametername=Wert` usw.
- **Wichtig:** Übergabe mit `@varname` und nicht `$varname`
- Sehr praktisch, wenn mehrere Parameter mehrfach mit denselben Werten übergeben werden sollen
- Parameter-Splatting kann mit expliziten Parametern kombiniert werden

# Parameter-Splatting (2)

179

## □ Vereinfachter Aufruf von **Invoke-Command**

Beispiel: Etwas umständlich bei Mehrfachaufrufen

```
$Cred = Get-Credential pemo23
Invoke-Command { ipconfig } -Computersname powerpc -Credential $Cred
Invoke-Command { netstat -a } -Computersname powerpc -Credential $Cred
Invoke-Command { date } -Computersname powerpc -Credential $Cred
```

Beispiel: Etwas kompakter (vor allem bei weiteren Parametern)

```
$Cred = Get-Credential pemo23
$paras = @{Computersname="powerpc";Credential=$Cred}

Invoke-Command { ipconfig } @paras
Invoke-Command { netstat -a } @paras
Invoke-Command { date } @paras -RunAsAdministrator
```

# Konstanten zu enums zusammenfassen

180

- enums = Zusammenstellung von Konstanten über den enum-Befehl
- Jeder Name steht für eine Zahl (in der Regel 0,1,2..)
- Praktisch, da mehrere Konstanten zu einer Gruppe (eigener Typ) zusammengefasst werden
- Ein Vorteil ist eine verbesserte Lesbarkeit
- Der **enum**-Befehl wurde in TB 10 vorgestellt



# enum-Konstanten beim switch-Befehl

181

- **Wichtig:** Der Name der enum-Konstanten wird bei einem Vergleich nicht in Anführungszeichen gesetzt

## Beispiel für enum-Konstanten

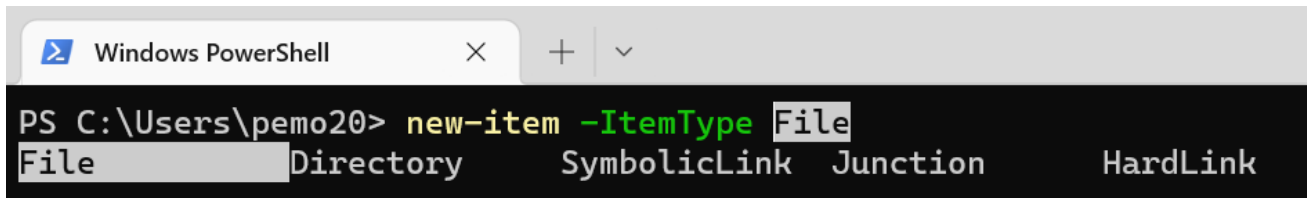
```
enum DataProvider
{
    SQLServer
    Oracle
    SQLite
}

$dbProvider = [DataProvider]::SQLServer
switch ($dbProvider)
{
    SQLServer { "Hier ist der SQL-Server" }
    Oracle { "Hier ist der Oracle-Server" }
    SQLite { "Und hier ist SQLite" }
    default { "Default-Aktion" }
}
```

# Umgang mit SymLinks (1)

182

- ❑ SymLink = Symbolische Verknüpfung
- ❑ Es gibt keine eigenen Commands
- ❑ Ein SymLink wird über den **ItemType**-Parameter des **New-Item-Cmdlet** angelegt



```
Windows PowerShell
PS C:\Users\pemo20> new-item -ItemType File
File Directory SymbolicLink Junction HardLink
```

Beispiel: SymLink im aktuellen Verzeichnis für das Windows PowerShell-Verzeichnis anlegen

```
New-Item -Path Posh -Target $PsHome -ItemType SymbolicLink
```

# Umgang mit SymLinks (2)

183

- ❑ Das Entfernen symbolischer Links ist bei der Windows PowerShell etwas „tricky“
- ❑ Remove-Item funktioniert nicht
- ❑ Ein Workaround ist die Delete()-Methode oder Cmd.exe
- ❑ Bei PowerShell 7 funktioniert alles wie beschrieben

Beispiel: Symbolisches Link bei Windows PowerShell entfernen

```
(Get-Item -Name Posh).Delete()
```

Beispiel: Symbolisches Link bei PowerShell 7.x entfernen

```
Remove-Item -Name Posh
```

# Vergleich mit \$null

184

- Bei Arrays/Listen kommt es auf die Reihenfolge an!
- \$null muss am Anfang stehen (der ScriptAnalyzer weist in VS Code deutlich darauf hin;)

```
$list1 = [System.Collections.Generic.List[String]]::new()  
$list1.add(1)  
$list1 -eq $null
```

Falsch  
(\*)

```
$list1 = [System.Collections.Generic.List[String]]::new()  
$null -eq $list1  
$false
```

Richtig  
(\*\*)

\* Der Vergleich wird mit jedem einzelnen Listenelement – gibt es keines, dass \$null ist, gibt es keine Ausgabe

\*\* Der Vergleich wird mit der Liste durchgeführt

# Wenn mehrere Rückgaben nur ein Objekt sind

185

- Eine Hashtable ist nur ein Objekt, auch wenn viele Zeilen ausgegeben werden (1)
- Eigenschaften vom Typ einer Collection müssen „expandiert“ werden (2)

1

```
Get-Command Get-Command).Parameters | Measure-Object
1
Get-Command Get-Command).Parameters.GetEnumerator() | Measure-Object
26
(Get-Command Get-Command).Parameters.GetEnumerator() | Where-Object Key -eq "All"

(Get-Command Get-Command).Parameters.GetEnumerator() | Where Key -eq "All" | Select
-ExpandProperty Value
```

2

```
Get-ACL -Path C:\ | Select-Object -ExpandProperty Access
```

# Dateien lesen und schreiben in einer Pipeline

186

- Wird Get-Content in runde Klammern setzen vermeidet eine „Prozess kann nicht auf Datei zugreifen“-Fehler
- **Tipp:** Zugriffsscheck mit Handle64 (SysInternals)

```
PS C:\temp> (get-content -Path .\Temp.csv) | Where-Object { $_ -ne ";" } |  
ForEach-Object { $cols = $_ -split ";";$cols[0].Substring(0,$Cols[0].Length-  
1), $Cols[1].Substring(0,$Cols[1].length-1) -join "," } | Set-Content  
.\Temp.csv  
PS C:\temp>
```

# Dateien über die Providerschreibweise ansprechen

187

- Allgemein `${drive:pfad}`
- Muss vom PSProvider unterstützt werden

```
${c:.\test2.txt} = ${c:.\test2.txt} | ForEach-Object { $_ -replace "/", "-" }
```

# Testen auf eine nicht leere Variable

188

- Was testet `if ($var) { }` ?
- **Antwort:** Ob `var` einen Wert ungleich `$null`/Leerstring besitzt, nicht, ob `var` existiert



# Zusammenfassung

189

- ❑ Die PowerShell ist ein klassischer Interpreter
- ❑ Beim Verarbeiten großer Datenmengen wird die Ausführung (sehr) langsam
- ❑ Niemals große Strings per += verknüpfen
- ❑ Große Dateien per StreamReader einlesen
- ❑ .NET-Laufzeit bietet ein reichhaltiges Repertoire an Listenklassen
- ❑ Regex für die schnelle Textverarbeitung

# Regeln für gute Skripte

## Themenblock 12

# Allgemeine Regeln

191

- ❑ Skripte und Functions immer mit Kommentarblöcken einleiten (<# ... #>)
- ❑ Kommentarbasierte Hilfe verwenden (z.B. .Synopsis)
- ❑ Auf Aliase verzichten
- ❑ Parameternamen ausschreiben
- ❑ Auf Einrückungen achten
- ❑ **Tipp:** PSScript Analyzer verwenden
  - ▣ Wird per **Import-Module** hinzugefügt
  - ▣ Aufruf über das **Invoke-ScriptAnalyzer**-Command

# Die #requires-Direktive

192

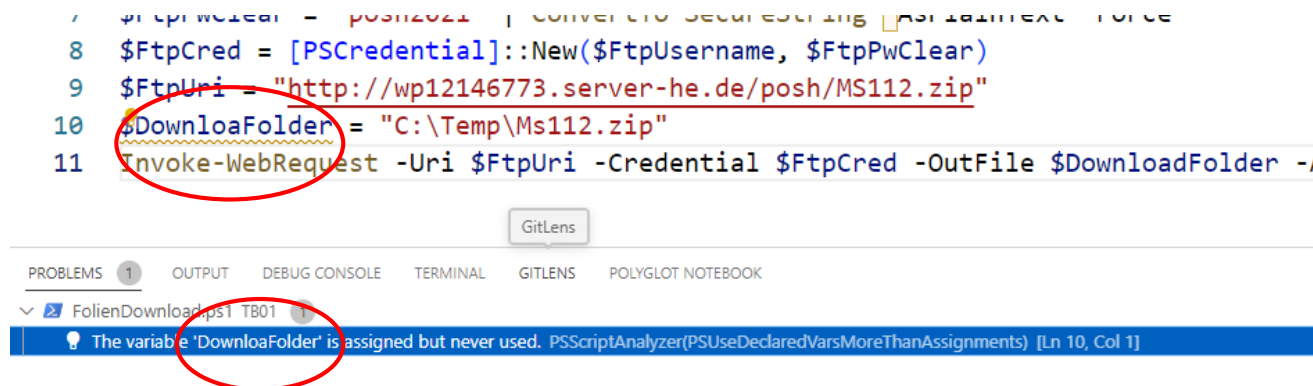
- Zu Beginn der Skriptausführungen werden bestimmte Voraussetzungen gecheckt
  - ▣ Wird das Skript mit der erwarteten Version der PowerShell ausgeführt?
  - ▣ Wurde das Skript als Administrator gestartet?
  - ▣ Sind die erforderlichen Module vorhanden?
- Trifft eine Bedingung nicht zu, bricht die Ausführung ab

```
#requires -runasadministrator  
#requires -modules activedirectory  
#requires -version 5.0
```

# Der Script Analyzer von Microsoft

193

- ❑ Analysiert ein Skript anhand eines Satzes an Regeln
- ❑ Soll die Qualität von Skripten verbessern und "Schwachpunkte" anzeigen
- ❑ Regeln lassen sich auf der Grundlage von Psm1-Dateien erweitern
- ❑ Von Anfang an Teil der PowerShell Extension von Visual Studio Code



```
7 $FtpPwClear = posh12021 | ConvertTo-SecureString -AsPlainText -Force
8 $FtpCred = [PSCredential]::New($FtpUsername, $FtpPwClear)
9 $FtpUri = "http://wp12146773.server-he.de/posh/MS112.zip"
10 $DownloadFolder = "C:\Temp\Ms112.zip"
11 Invoke-WebRequest -Uri $FtpUri -Credential $FtpCred -OutFile $DownloadFolder -,
```

The variable 'DownloadFolder' is assigned but never used. PSScriptAnalyzer(PsUseDeclaredVarsMoreThanAssignments) [Ln 10, Col 1]

# Verwenden einer Versionsverwaltung

194

- ❑ Versionsverwaltung – ermöglicht, mehrere Versionen einer Datei abzulegen
- ❑ Voraussetzung, wenn es mehrere Autoren für eine Datei gibt – Änderungen können im Detail nachvollzogen und rückgängig gemacht werden
- ❑ Für größere Skripte ist eine Versionsverwaltung praktisch Pflicht
- ❑ Für PowerShell empfiehlt sich Git
- ❑ Nahtlose Integration in Visual Studio Code

# Zusammenfassung

195

- Allgemeine Tipps
- #requires-Direktive
- PowerShell ScriptAnalyzer

# Umgang mit Klassen

Themenblock X1



# Die Themen

197

- Vorteile von Klassen
- Der class-Befehl
- Hinzufügen eines Konstruktors
- Hinzufügen von Eigenschaften
- Hinzufügen von Methoden
- Hinzufügen von Enumerationen
- Klassen ableiten
- Überschreiben von Methoden

# Vorteile von Klassen

198

- Eine Klasse definiert einen Typ, aus dem Objekte gemacht werden können
- Damit lassen sich Werte, die zueinander in einer Beziehung stehen (z.B. die Daten einer Bereitstellung) zusammenfassen
- Mit Klassen wird eine Programmiersprache flexibler was das Abbilden von Datenstrukturen angeht
- Bei PowerShell gibt es Klassen erst seit Version 5.0

# Neue Syntaxelemente für Klassen

200

- `class`
- `enum`
- `$this`
- `base()`
- `[hidden]`
- `[Typname]::new()`

# Der class-Befehl

201

- Definiert eine neue Klasse
- Es wird lediglich ein Name benötigt
- Innerhalb der Klassendefinition werden die Members der Klasse definiert

Beispiel für eine Klassendefinition

```
class Server
{
    [Int]$ServerId
}
```

# Aus Klassen werden Objekte

202

- Klasse = Definition
- Objekt = Struktur im Arbeitsspeicher, deren Aufbau durch die Klasse vorgegeben ist
- Zwei Varianten:
  - ▣ Statisches New-Member (seit Version 5.0)
  - ▣ **New-Object**-Cmdlet

Ein Objekt über new() anlegen

```
$S1 = [Server]::new()
```

Ein Objekt über New-Object anlegen

```
$S1 = New-Object -TypeName Server
```

# Objekte können auch ohne Klasse angelegt werden

203

- ❑ **New-Object** mit PObject/PSCustomObject als Typ
- ❑ **[PSCustomObject]** mit Hashtable
- ❑ Select-Object
- ❑ usw.
- ❑ Alle diesen Varianten verwenden einen vordefinierten Typ
- ❑ Ein selber definierter Typ bringt Vorteile:
  - ▣ Klare Struktur, mehr Flexibilität, eigene Formatierung bei der Ausgabe durch Format-Table

# Hinzufügen eines Konstruktors

204

- Konstruktor – Name für die Methode, die mit dem Instanzieren, z.B. per **new()**, automatisch ausgeführt wird
- Hier erhalten z.B. Properties ihre Werte
- **Wichtig:** Innerhalb der Klassendefinition werden alle Members über **\$this** angesprochen

Klasse mit Konstruktor

```
class Server
{
    [Int]$ServerId

    Server([String]$Id)
    {
        $this.ServerId = $Id
    }
}
```

# Hinzufügen von Eigenschaften

205

- Eine Eigenschaft ist lediglich eine Variable innerhalb der Klassendefinition
- Ein Typ ist „Pflicht“ (ansonsten [Object])
- Es gibt kein get/set und keinen expliziten Gültigkeitsbereich
- **Wichtig:** Innerhalb der Klassendefinition werden Variablen über **\$this** angesprochen



# Hinzufügen von Methoden

206

- Eine Methode ist ein Scriptblock mit einem Datentyp, einem Namen und optionalen Parametern
- Der Datentyp ist „Pflicht“ – gibt eine Methode nichts zurück, sollte `[void]` vor dem Namen angegeben werden
- Rückgaben immer per **return**-Befehl

## Klasse mit Methode

```
class Server
{
    [ServerStatus]$Status

    [void]Initialize()
    {
        $this.Status = [ServerStatus]::Initialized
    }
}
```

# Hinzufügen von Enumerationen

207

- Enumeration – fasst Konstanten mit einem Namen zusammen
- Eigener Typ – wird mit dem **enum**-Befehl definiert
- Grundsätzlich praktisch – ein *[ServerStatus]* ist besser als *[String]* oder *[Int]*

## Enumerationskonstante

```
enum ServerSize
{
    Small
    Medium
    Large
}
```

# Klassen ableiten

208

- Eine Klasse kann sich von einer anderen Klasse ableiten
- Sie übernimmt dadurch alle Members der Basisklasse
- **Vorteil:** Ein Satz von Members muss nur einmal definiert werden
- Die Basisklasse kann auch eine .Net-Klasse sein

Abgeleitete Klasse

```
class SpezialServer : Server
{
    [void]GetStatus()
    {
        return $this.Status
    }
}
```

# Überschreiben von Methoden

209

- In einer abgeleiteten Klasse können Methoden der Basisklasse ersetzt werden -> Überschreiben
- **Vorteil:** Mehr Flexibilität, da das „Verhalten“ in einer abgeleiteten Klasse anders implementiert werden kann

Methoden überschreiben

```
class SpezialServer : Server
{
    [void]Stop()
    {

    }
}
```

# Zusammenfassung

210

- ❑ Der **class**-Befehl definiert eine Klasse (Typ)
- ❑ Eine Klasse besitzt in der Regel Members und einen Konstruktor
- ❑ Eigenschaften und Methoden werden innerhalb der Klasse per `$this` angesprochen
- ❑ Enums sind praktisch für Konstantenlisten
- ❑ Klassen können auch abgeleitet und Methoden in abgeleiteten Klassen überschrieben werden

# Übung zum Thema Klassen

211

- Umsetzen einer (sehr) einfachen Rechenzentrum-Simulation
- Es gibt eine Klasse *PSRechenzentrum*
- Es gibt eine Klasse *PSServer* mit Properties und Methoden (z.B. Start und Stop)
- Wie werden die PSServer-Objekte mit dem PSRechenzentrum-Objekt zusammengebracht?

# PowerShell Remoting mit SSH

Themenblock X2

# Die Themen

213

- Warum SSH?
- OpenSSH unter Windows
- PowerShell 7.x für SSH konfigurieren
- Ein Beispiel



# Warum SSH?

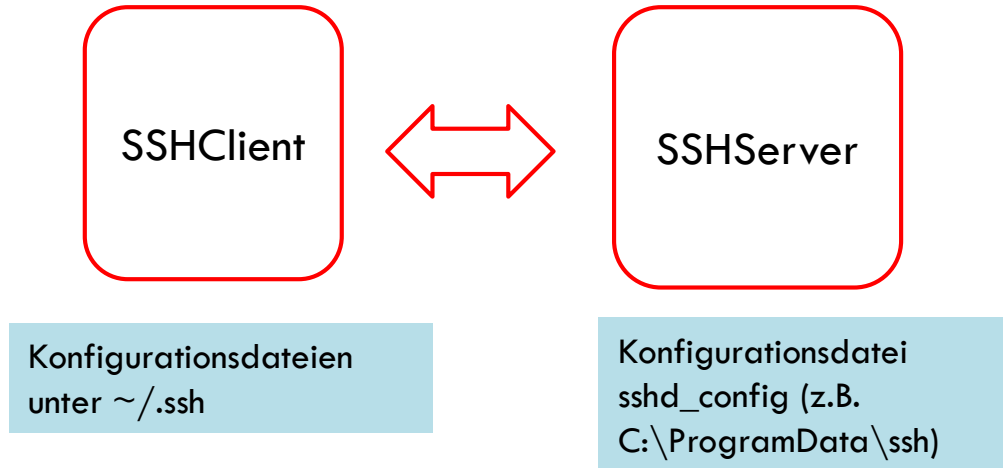
214

- ❑ SSH = Secure Shell
- ❑ Herstellen einer Remote-Verbindung zu einem anderen Computer in der Konsole
- ❑ TCP-Port, in der Regel 22
- ❑ Steht die Verbindung, werden alle eingegeben Kommandos auf dem Remote-Computer ausgeführt
- ❑ Auch Datei- und Bildschirmübertragung möglich
- ❑ Bei Unix/Linux seit > 20 Jahren ein Standard
- ❑ In Gestalt von OpenSSH bei Windows 10/Windows Server 2016 Teil des Betriebssystems (wird als Feature installiert)

# SSH-Client/SSH-Server

215

- Client: OpenSSH oder Putty
- Server: In der Regel OpenSSH Server



# OpenSSH unter Windows

216

- ❑ **Ziel:** Sichere Dateiübertragung zu anderen Computer per SSH
- ❑ Open Source-Bibliothek unter BSD-Lizenz
- ❑ Wird bei Windows Server/Windows 10 als Feature installiert
- ❑ Danach steht u.a. Ssh.exe (Client) und andere Tools zur Verfügung

```
Add-WindowsCapability -Online -Name OpenSSH.Client~~~~0.0.1.0  
Add-WindowsCapability -Online -Name OpenSSH.Server~~~~0.0.1.0
```

← Einstellungen

## Optionale Features



OpenSSH-Client

5,05 MB



OpenSSH-Server

4,71 MB

# SSH in der Praxis

217

- ❑ Der Umgang mit SSH ist grundsätzlich einfach und „mehr als ausreichend“ dokumentiert
- ❑ Im einfachsten Fall Aufruf von `ssh username@hostname` bzw. `ssh username@ip-adresse`
- ❑ Beim ersten Mal muss der Fingerprint des Public Key des Host bestätigt werden (wird in die Liste der „Known hosts“ aufgenommen)
- ❑ Anschließend werden alle Eingaben auf dem Host ausgeführt
- ❑ Mit PowerShell hat diese Variante nichts zu tun

# SSH und PowerShell Remoting

218

- Nur ab PowerShell 6.0 möglich
- Cmdlets wie **Enter-PSSession** besitzen einen **Hostname-**Parameter, der SSH „auswählt“
- Authentifizierung über Kennwort oder Public Key
- **Vorteile:**
  - ▣ SSH ist in der IT-Welt ein Standard
  - ▣ Einfachere Konfiguration (kein Enable-PSRemoting mehr, keine GPOs)
  - ▣ Keine Adminberechtigung erforderlich!
  - ▣ Keine Double Hop-Problematik
  - ▣ Eventuell bessere Performance

# SSH Server für PowerShell konfigurieren

219

- ❑ Unter Windows muss OpenSSH Server als Feature hinzufügen
- ❑ In `ssdh_config` muss ein subsystem-Eintrag für PowerShell hinzugefügt werden
- ❑ Gute Anleitung: <https://lazyadmin.nl/powershell/powershell-ssh>
- ❑ **Tipp:** Enable-SSHRemoting-Command aus dem Microsoft.PowerShell.RemotingTools-Modul

# SSH mit Public Key-Authentifizierung

220

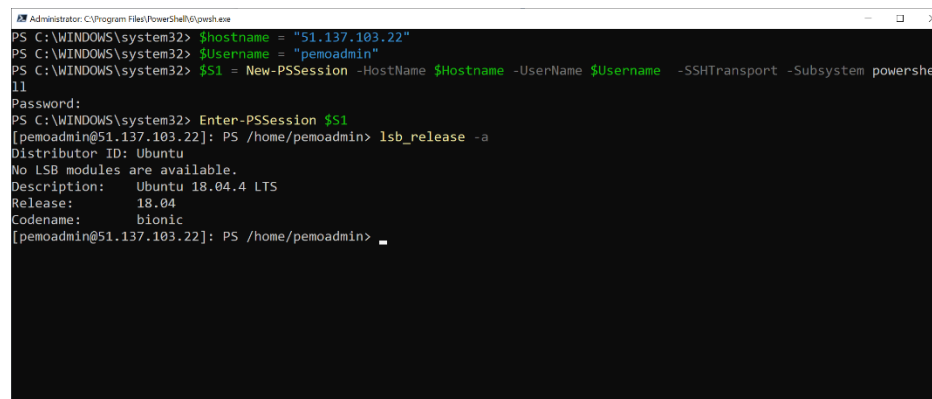
- ❑ Bei der Passwort-Authentifizierung muss das Kennwort jedes (!) Mal eingegeben werden
- ❑ PSCredentials gibt es bei SSH nicht
- ❑ Die Alternative ist die Authentifizierung über einen Public Key/Private Key
- ❑ Das Paar wird zuerst per `ssh-keygen.exe` generiert (sehr einfacher Aufruf)
- ❑ Über den *KeyFilePath*-Parameter wird der Pfad zur Datei mit dem Private Key angegeben
- ❑ Gute Anleitung unter <https://4sysops.com/archives/powershell-remoting-with-ssh-public-key-authentication/>

# Ein Beispiel für eine SSH-Remote-Session

221

- ❑ Anlegen einer Session über **New-PSSession**
- ❑ PasswordAuthentication Yes in *sshd.config* auf dem Host wird vorausgesetzt
- ❑ Es werden nur *Hostname* und *Username* angegeben
- ❑ Die Parameter *SSHTransport* und *Subsystem* sind optional

```
$S1 = New-PSSession -HostName $Hostname -UserName $Username  
Enter-PSSession -Session $S1
```



```
Administrator: C:\Program Files\PowerShell\powershell.exe  
PS C:\WINDOWS\system32> $hostname = "51.137.103.22"  
PS C:\WINDOWS\system32> $Username = "pemoadmin"  
PS C:\WINDOWS\system32> $S1 = New-PSSession -HostName $Hostname -UserName $Username -SSHTransport -Subsystem powershell  
Password:  
PS C:\WINDOWS\system32> Enter-PSSession $S1  
[pemoadmin@51.137.103.22]: PS /home/pemoadmin> lsb_release -a  
Distributor ID: Ubuntu  
Description:    Ubuntu 18.04.4 LTS  
Release:        18.04  
Codename:       bionic  
[pemoadmin@51.137.103.22]: PS /home/pemoadmin>
```



# SSH-Troubleshooting

222

- ❑ Bei der Passwort-Authentifizierung spielt die Public key/Private key-Thematik keine Rolle
- ❑ Für den Client spielen die SSH-Dienste keine Rolle
- ❑ SSH-Client-Daten komplett löschen in %userprofile%/.ssh
- ❑ Testen mit `ssh -v username@hostname`
- ❑ SSH-Server mit `ssh.exe` testen!
- ❑ Eventuell falscher Eintrag in `/etc/ssh/sshd_config`-Datei
- ❑ Es gibt viele Anleitungen im Internet

# Tipps zu PowerShell Remoting per SSH

223

- Viel Know-how als Teil der PowerShell Dokumentation und im Web

<https://docs.microsoft.com/en-us/powershell/scripting/learn/remoting/ssh-remoting-in-powershell-core?view=powershell-7>

<https://www.thomasmaurer.ch/2020/04/enable-powershell-ssh-remoting-in-powershell-7/>

# Zusammenfassung

224

- ❑ SSH ist der neue Standard für PowerShell-Remoting
- ❑ Setzt PowerShell 7.x voraus
- ❑ Wichtige Vorteile (u.a. keine PowerShell-Konfiguration, keine Admin-Berechtigung erforderlich)
- ❑ Public Key Authentication anstatt Password
- ❑ SSH-Konfiguration dank *Enable-SSHRemoting*-Command beim SSH-Host einfach
- ❑ Für die Windows PowerShell gibt es das Produkt *PowerShell Server* von *n/software*

225

# Secret Management

ThemenblockX3

# Der Umgang mit Kennwörtern

226

- Es gibt keine „Best Practices“ für den Umgang mit Kennwörtern
- Sie dürfen nicht Teil der Ps1-/Psm1-Datei sein
- Wenn sie verschlüsselt abgespeichert werden, ist der Schlüssel nicht übertragbar
- Einen „selbstgebauten“ Schlüssel zu verwenden ist auch nicht optimal
- Wird die Kennwortverwaltung geändert, müssen alle Skripte und Module angepasst werden
- Gesucht wird eine flexiblere Lösung, die die Kennwortabfrage von der verwendeten Speichermethode entkoppelt

# Das SecretManagement-Modul

227

- Das SecretManagement-Modul bringt mehr Flexibilität für das Abrufen von Kennwörtern (SecureStrings), PSCredentials usw.
- Stammt vom PowerShell-Team, modularer Aufbau
- Installation per **Install-Module** von der PowerShell Gallery
- Ausführliche Dokumentation unter <https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.secretmanagement>

# Installation

228

- Es müssen zwei Module installiert werden:
  - ▣ Microsoft.PowerShell.SecretManagement
  - ▣ Microsoft.PowerShell.SecretStore

```
Install-Module Microsoft.PowerShell.SecretManagement -Force -Verbose
```

```
Install-Module Microsoft.PowerShell.SecretStore -Force -Verbose
```

```
Get-Module -Name Microsoft.PowerShell.Secret* -ListAvailable
```

# Umgang mit Secrets und Vaults

229

- ❑ Secrets (z.B. SecureString, PSCredential, Text) werden in Vaults (Ablagen) abgelegt
- ❑ Es gibt von Anfang an eine Standard-Vault (Verzeichnis im Benutzerprofil-Verzeichnis)
- ❑ Weitere Vaults können registriert werden
- ❑ Ein Vault kann lokal oder remote angelegt werden
- ❑ Die Vault-Verwaltung ist erweiterbar (es gibt u.a. eine Extension für KeePass)



# Die Cmdlets im SecretManagement-Modul

230

Command	Was macht es?
Get-Secret	Ruft ein Secret aus einem Vault (Kammer) ab
Get-SecretInfo	Ruft die Metadaten eines Secret ab, z.B. den Vaultname
Register-SecretVault	Legt einen neuen Vault an
Remove-Secret	Entfernt ein Secret aus einem Vault
Set-Secret	Ändert ein Secret in einem Vault
Set-SecretInfo	Ändert die Metadaten über ein Secret
Set-SecretVaultDefault	Legt einen Vault als Default fest
Test-SecretVault	Testet die Integrität eines Vault
Unregister-SecretVault	Entfernt einen Vault

# Was kann in einem Vault abgelegt werden?

231

- Es werden alle wichtigen Datentypen unterstützt:
  - ▣ String
  - ▣ Byte[]
  - ▣ Hashtable
  - ▣ SecureString
  - ▣ PSCredential

# Beispiel (1)

232

- Ein Kennwort als SecureString speichern und nutzen

```
#requires -Modules Microsoft.PowerShell.SecretManagement  
  
Set-Secret -Name PosHPw -Secret "geheim+1234"  
Get-Secret -Name PosHPw
```

# Beispiel (2)

233

- Ein PSCredential als Secret speichern und nutzen

```
#requires -Modules Microsoft.PowerShell.SecretManagement

$PwSec = Read-Host -Prompt "Kennwort?" -AsSecureString
$PSCred = [PSCredential]::new("psadmin", $PwSec)

Set-Secret -Name AdminPw -Secret $PSCred -Vault LocalStore

Get-Secret -Name AdminPw
```

# Die Vorteile der Secrets

234

- ❑ Am Umgang mit Credentials ändert sich nichts
- ❑ Das SecretManagement-Modul bietet eine weitere Ebene, die den Umgang mit Kennwörtern flexibler macht
- ❑ Der Ort, an dem Kennwörter abgelegt werden, kann von außen konfiguriert werden
- ❑ Ein Skript verwendet den Vault, der auf jedem System, auf dem es ausgeführt wird, angelegt wurde
- ❑ Eine Authentifizierungsmethode kann geändert werden (z.B. Umstellung auf Azure Keys), ohne dass das Skript geändert werden muss

# Zusammenfassung

235

- ❑ Das *SecretManagement-Modul* bietet einen vereinheitlichten Umgang mit Credentials
- ❑ Stammt vom PowerShell Team
- ❑ Muss nachträglich installiert werden
- ❑ Nicht auf Kennwörter beschränkt
- ❑ Meine Empfehlung: Skripte auf SecretManagement umstellen

# Module und Skripte mit Pester testen

Themenblock X4

# Die Themen

237

- RTFM
- Was genau ist ein Test?
- Warum Tests?
- Warum Pester?
- Ein erstes Beispiel



# Bitte zuerst einen Blick in die Doku

238

- Ohne einen Blick in die Doku ist der „Frust“ vorprogrammiert
- <https://pester.dev/>
- Gute Einführung mit vielen Beispielen
- Ein Grund sind die großen Unterschiede zwischen Version 3.x und v5
- **Problem:** Bei Windows 10 ist Pester 3.4.0 vorinstalliert

# Was genau ist ein Test?

239

- ❑ Test = Funktionstest (Komponententest)
- ❑ Eine Methode/Function usw. wird mit definierten Parametern ausgeführt
- ❑ Der Rückgabewert (!) wird mit einem erwarteten Wert verglichen
- ❑ Stimmt der Wert überein, wurde der Test bestanden und die „grüne Lampe“ geht an
- ❑ Stimmt der Wert nicht überein, „rote Lampe“

# Warum Tests?

240

- ❑ Ein Funktionstest testet nicht, ob eine Anwendung/Skript funktioniert
- ❑ Ein Funktionstest testet lediglich, ob bei einem Aufruf einer Methode/Function mit bestimmten Argumenten der erwartete Rückgabewert entsteht
- ❑ Ein Funktionstest soll sicherstellen, dass eine Veränderung am Quelltext/Skript keine negativen Auswirkungen hat
- ❑ Software-Entwickler schreiben hunderte von Tests für Ihre Anwendung

# Warum Pester?

241

- ❑ Pester hat sich schnell zu dem Test-Modul für PowerShell-Skripte entwickelt
- ❑ Ist bei Windows 10/Windows Server 2016 von Anfang an dabei (aber in einer veralteten Version)
- ❑ Verfolgt den BDD-Ansatz (*Behavior Driven Development*)
- ❑ Mit Pester lässt sich alles testen, z.B. auch eine Verzeichnisstruktur oder eine Serverkonfiguration

# Ein erstes Beispiel (1)

242

- Ausgangspunkt ist eine simple Function

Function, die eine Operation ausführt und einen Wert zurückgibt

```
function New-Password
{
    param([Int]$Count)
    (1..$Count).ForEach{[Char](65..92 | Get-Random)} -join ""
}
```

# Ein erstes Beispiel (2)

243

- Die Function soll getestet werden

Pester-Test für einen Function-Aufruf

```
Describe "Anlegen eines Passwort" {  
  
    It "Erzeugt Passwort mit 8 Zeichen" {  
        $Pw = New-Password -Count 8  
        $Pw.Length | Should Be 8  
    }  
}
```



Ist-Wert



Soll-Wert

# Zusammenfassung

244

- ❑ Funktionstest sind wichtig
- ❑ In erster Linie bei größeren Skripten und Modulen
- ❑ Bei Team-Entwicklung sind sie Pflicht
- ❑ Funktionstest als Teil einer Release-Pipeline für PowerShell-Module
- ❑ Pester ist das Standardtestingtool für PowerShell und genial (auf Versionsnummer achten)
- ❑ „The Pester Book“ von Adam Bertram

# Übungen zum Thema Pester

245

- Der Autor der Passwort-Function erhält den Auftrag, dass die Passwortlänge 8 Zeichen sein muss und das erste Zeichen ein Sonderzeichen (z.B. !) sein muss
- **Übung Nr. 1:** Wie muss die Function angepasst werden?
- **Übung Nr. 2:** Wie muss der Test angepasst werden?
- **Übung Nr. 3:** Ein weiterer Test soll prüfen, ob das erste Zeichen des Passworts ein Sonderzeichen ist



# Weitere Informationen

Was noch zu sagen wäre...

# Know-how zur PowerShell

247

- ❑ Die offizielle Microsoft-Dokumentation unter [docs.microsoft.com/powershell](https://docs.microsoft.com/powershell)
- ❑ PowerShell Gallery
- ❑ Immer noch ein Klassiker: PowerShell Cookbook von Lee Holmes und PowerShell in Action von Bruce Payette
- ❑ CoPilot&Co – hier werden alle Fragen beantwortet 😊

# Zum Schluss...

248

- Ich hoffe, dass Ihnen/Dir die Schulung etwas gebracht hat
- Wenn noch Fragen sind, einfach eine Mail an [pm@activetraining.de](mailto:pm@activetraining.de)
- Alle Beispiele gibt es im GitHub-Repo (Adresse steht auf einer der ersten Folien)
- Nette Abwechslung: Der PowerShell Comic  
<https://learn.microsoft.com/de-de/powershell/scripting/community/digital-art>
- Vielen Dank für Ihre/Deine Teilnahme!