PowerShell für Einsteiger

Grundlagen und Lernsituationen

Inhaltsübersicht

- Begrüßung/Vorstellung
- Impulsvortrag PowerShell
- PowerShell Grundlagen (Cmdlets, Hilfe zur Selbsthilfe)
- Pipeline, Prozesse und Dienste verwalten
- Benutzer und Gruppen verwalten, Benutzerprofile
- Dateisystem und NTFS-Rechte, Freigaben, Netzlaufwerke
- Netzwerkkonfiguration
- Programmierung mit PowerShell (ps1-Skripte, Zugriff auf .NET-Objekte)
- Serveraufgaben, Loganalyse, Webzugriff, Jobs
- Klassenarbeiten und Lernsituationen mit PowerShell erstellen und vorstellen

PowerShell Serveraufgaben

Module, Überblick Active Directory Funktionen, Updates

PowerShell Module

- Module sind Sammlungen von Cmdlets
- Modulentwicklung und -integration laufend:
  - Windows 7: ca. 10 Module
  - Windows 10: ca. 70 Module
  - Windows Server 2016 (ohne AD): ca. 80 Module
- Exchange Server, SharePoint Server haben eine eigene PowerShell-basierte Verwaltungsshell

PowerShell Module

- Diverse Speicherorte auf dem Computer (OS-abhängig)
- Anzeige der Speicherorte für einen Computer:

```
PS C:\Users\anr> $Env:PSModulePath -split ";"
C:\Users\anr\Documents\PowerShell\Modules
C:\Program Files\PowerShell\Modules
c:\program files\powershell\7\Modules
C:\Program Files\WindowsPowerShell\Modules
C:\windows\system32\WindowsPowerShell\v1.0\Modules
```

- Modul-Management mit den Cmdlets der Module-Familie
- `Get-Module -ListAvailable` zeigt geladene und nachladbare Module
- `Get-Command -Module <MOD>` zeigt Befehle im Modul `<MOD>`

Auszug Modulliste Windows Server 2016
(als DC mit DNS Server)

- ActiveDirectory*
- ADDSDeployment*
- AppLocker
- BestPractices*
- BranchCache
- DnsClient
- DnsServer*
- GroupPolicy*
- NetTCPIP
- PKI
- PrintManagement
- RemoteDesktop*
- ScheduledTasks
- ServerCore*
- ServerManager*
- VpnClient

Module dynamisch nachladen

- Bei Verwendung eines Cmdlets wird das passende Modul dynamisch und automatisch nachgeladen.

- Hands-on:
  Nachladen des Moduls DnsClient durch Aufruf des Cmdlets Resolve-DnsName
  - 1. Aktive Module auflisten
  - 2. Domain Name www.cisco.com auflösen lassen
  - 3. Aktive Module erneut auflisten

Module manuell nachladen

- Explizites manuelles Laden mit `Import-Module`
- Grundsätzlich nicht notwendig, aber…
  - häufig in Anmeldeskripte etc. integriert
  - **spart Zeit bei Skriptausführung**
  - initiales Laden der PowerShell dauert dann länger

Das Modul ServerManager

- Ist das CLI Gegenstück zur Server Manager GUI
- Ermöglicht u.a. Installation und Verwaltung von Rollen und Features

Cmdlet	Aufgabe
`Get-WindowsFeature`	Rollen und Features auflisten
`Add-WindowsFeature` `Install-WindowsFeature`	Neue Rolle bzw. neues Feature installieren
`Remove-WindowsFeature` `Uninstall-WindowsFeature`	Installierte Rolle bzw. installiertes Feature deinstallieren

Installation von Rollen und Features

- Parameter zu den Cmdlets
  - entsprechen den Eingaben im Wizard der Server Manager GUI

Parameter	Bedeutung
`-Name`	Bezeichnung der Rolle bzw. des Features
`-ComputerName`	Zielgerät der Installation
`-IncludeAllSubFeatures`	alle abhängigen Dienste/Features mit installieren
`-IncludeManagementTools`	Verwaltungsshells mit installieren. Anders als mit GUI nicht automatisch!
`-Restart`	Computer neu starten, falls erforderlich

Installation von Rollen und Features

- Hands-on:
  Alle aktuell installierten Rollen und Features auflisten

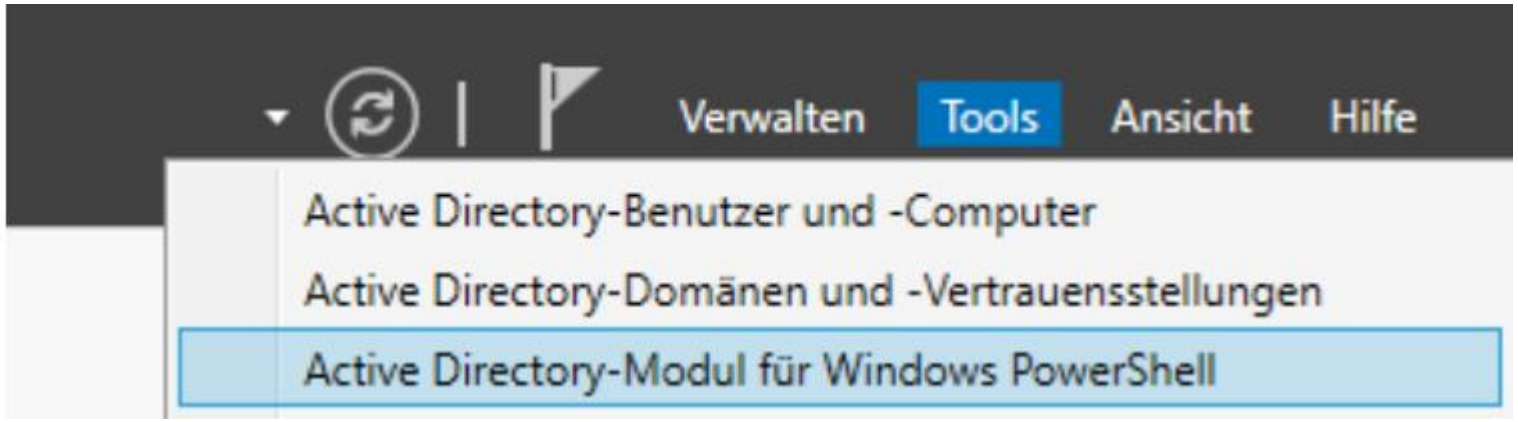
- Hands-on:
  Den Microsoft Webserver IIS mit allen Diensten auf dem Computer WS01 installieren.

Das Modul ActiveDirectory

- ActiveDirectory ermöglicht Verwaltung von Benutzern und Gruppen in einer Domäne.
- Eines der mächtigsten Module für Windows Server.
- Enthält ca. 150 eigene Cmdlets.

ActiveDirectory Shell

- Active Directory hat eine eigene Verwaltungsshell



- Hintergrund ist schwarz (statt blau wie ältere PowerShell Versionen)
- Alias für

```
%windir%\system32\WindowsPowerShell\v1.0\powershell.exe -NoExit
-Command Import-Module ActiveDirectory
```

ActiveDirectory Management

- ActiveDirectory etabliert PSDrive AD:
- Navigation im Laufwerk mit distinguished names (DN) als Pfade für Set-Location
- Benutzerverwaltung im AD
  - Familie `LocalUser`
  - Familie `LocalGroup`
  - Familie `LocalGroupMember`

ActiveDirectory Advanced Stuff

- Komplette Verwaltung der Domäne mit ActiveDirectory Modul möglich
- Verwaltung von OUs
- Schutz vor versehentlichem Löschen
- Anlegen, Konfigurieren und Vererben von Gruppenrichtlinien
- Einbinden von Domänen in Forests
- Vertrauensstellungen konfigurieren
- FSMO-Rollen anzeigen und übertragen
  - `Get-ADForest | Format-List -Property <FSMO-ROLE>`

Windows Updates

- Server(-Cluster) müssen regelmäßig mit Updates versorgt werden.
- PowerShell bietet (ab Windows Server 2016) hierfür das Modul *WindowsUpdateProvider*
- Provider ist Windows Update (Default) oder ein Server mit der Rolle für
  Windows Server Update Services (WSUS).
  - Windows Update managt automatisch
  - WSUS ermöglicht Update-Steuerung im (Firmen-)Netzwerk
- Funktionen im Modul funktionieren mit beiden Providern.

Suchen nach Updates

- Cmdlet `Start-WUScan`
- Eingrenzen der Updates über `-SearchCriteria <CONDITION_STRING>`
- Häufige Kombination: `"IsInstalled=0 AND IsHidden=0 AND IsAssigned=1“`
  - `IsInstalled=0` → noch nicht installierte Updates anzeigen
  - `IsHidden=0` → öffentlich verfügbare, einsehbare Updates anzeigen
  - `IsAssigned=1` → für die Platform vorgesehene Updates anzeigen

Installation von Updates

- Installation vom vordefinierten Provider.
- Cmdlet `Install-WUUpdates`

Übung PS70

- Anzeigen von Modulen und deren Inhalten
- Laden von Modulen
- Module ActiveDirectory

Analyse von Log Dateien

- Windows Logs können als evtx-Datei gespeichert/exportiert werden.
- Laden in PowerShell mit Cmdlet `Get-WinEvent`
- Verwendung einer Pipeline, um die Events zu filtern.

Übung PS71

- Machen Sie sich mit der Ereignisanzeige vertraut
- Analyse von Log Dateien
- Spurensuche à la IT-Forensik
- Dokumentieren von Ergebnissen

PowerShell goes Internet

HTTP-Interaktion, E-Mail-Versand

Interaktion mit Webseiten

- PowerShell ermöglicht Interaktionen per CLI
- Entspricht grob den UNIX Befehlen *wget* und *curl*
  - *wget* und *curl* ermöglichen das Herunterladen von Webseiten und Dateien
  - *curl* ermöglicht z.B. auch Senden von Login-Daten

- PowerShell bietet für alle diese Aktionen das Cmdlet `Invoke-WebRequest`
  - sendet HTTP(S) Requests an Webseiten
  - ist sehr flexibel einsetzbar (>20 Parameter)

E-Mails verschicken mit PowerShell

- Bei der Aufgabenplanung werden abhängig vom Auslösen von Triggern bestimmte Aktionen ausgeführt.
- Eine Aktion ist eine Info an den Administrator per E-Mail.
- PowerShell ermöglicht die Automatisierung mit dem Cmdlet `Send-MailMessage`

Übung PS72

- HTTP Anfragen stellen
- E-Mails via PowerShell versenden
- PowerShell wie wget und curl auf *Nix-Systemen nutzen

PowerShell CIM und WMI

Informationsbeschaffung zur Administration und Fernwartung

Common Information Model (CIM)

- Ist DMTF-Standard für Management von IT-Systemen
- CIM stellt lediglich ein Datenmodell zur Verfügung
- Unterschiedliche Implementierungen möglich
- Protokoll: WS-Man (Web Services-Management)

Windows Management Instrumentation (WMI)

- Das WMI ist ein CIM-Erweiterung von Microsoft
- Zugriff auf fast alle Einstellungen
  - Lesend und schreibend
  - Lokal oder über Netzwerk
- Bekanntes CLI auf der *cmd.exe* ist WMIC
- Protokoll: RPC (Remote Procedure Call)

Windows Management Instrumentation

- PowerShell implementierte WMI-Cmdlets vor Version 3.0
  - Diese sind nun deprecated
- Ab PowerShell 3.0 gibt es CIM-Cmdlets

WMI-Cmdlet (deprecated)	CIM-Cmdlet
Get-WmiObject	Get-CimInstance
Invoke-WmiObject	Invoke-CimMethod
Register-WmiEvent	Register-CimIndicationEvent
Remove-WmiObject	Remove-CimInstance
Set-WmiInstance	Set-CimInstance

Jobs und Sessions

Zugriff auf entfernte Rechner, Workflows, Parallelisierung

Steuern von Remotecomputern

- Hintergründe:
  - Administration ist häufig Fernwartung
  - Anzahl betreuter Maschinen kann (sehr) groß sein

- Idee:
  - Administration von einer zentralen Workstation
  - Aufgaben („Jobs“) werden von dort auf Remotecomputern ausgeführt
  - Automatisierung durch PowerShell Skripte

Verwendung von Jobs und Sessions

- Job meint in PowerShell einen Hintergrundprozess
  - Bekannt aus `Invoke-Command` (Netzwerkkapitel)
  - Manche Cmdlets kennen den Parameter `-AsJob`
  - Entspricht grob *NIX Option `--daemon` bzw. `&` hinter dem Befehl
- Ein Job blockiert die PowerShell nicht weiter
- Verbindung mit Remotecomputer über Session
  - lokal : Job Results nur in der Session verfügbar
  - remote : Job Results können abgerufen oder gespeichert werden

Typen von Sessions

- Lokale Session
  - Start mit `Enter-PSSession`
  - Ende mit `Exit-PSSession`
- Remote Session
  - Start mit `New-PSSession`
  - Session wird über eine Variable verwaltet

Lokale Session
Schematischer Ablauf

Schritt	Beschreibung	Cmdlets
1	Interaktive Session starten	`Enter-PSSession -ComputerName <Ziel>`
2	Gewünschte Jobs starten	`Start-Job -ScriptBlock { … }`
3	Ergebnisse der Jobs abfragen	`Receive-Job -Name <JobName>`
4	Jobs auf Zielsystem löschen	`Remove-Job -Name <JobName>`
5	Interaktive Session beenden	`Exit-PSSession`

Lokale Session
Hinweise

- Namen der gestarteten Jobs merken bzw. sichern
  - Generierter Name wird beim Start anzeigt (s. Beispiel)
  - Jobs können mittels `Get-Job` abgefragt werden
  - Job kann über Parameter `-Name` manuell benannt werden
- Erzeugte Jobs nach Abfrage der Ergebnisse löschen
  - Jobs leben sonst als „Zombies“ weiter
  - Ergebnisse mit Parameter `-Keep` mehrfach abrufbar
- Mehrere Jobs z. B. mit `ForEach-Object` verarbeiten

Lokale Session
Troubleshooting

- Der Dienst *Windows-Remoteverwaltung (WS-Verwaltung)* muss auf dem Zielsystem laufen.
  - Ggf. per `Start-Service -Name WinRM` manuell starten
  - Starttyp bei Bedarf auf automatisch setzen

- Die Netzwerkverbindung zum Zielsystem darf nicht öffentlich sein.
  - Außerhalb Domäne: Heimnetzwerk oder Arbeitsplatznetzwerk wählen
  - In Domäne: Domänennetzwerk wählen

- Bei Zugriffsverweigerung in Administrator PowerShell `winrm quickconfig` ausführen und Änderungen mit `y` bestätigen.

Lokale Session
Beispiel

```
PS C:\Windows\System32> Enter-PSSession -ComputerName localhost
[localhost]: PS C:\Users\anr\Documents> Start-Job -ScriptBlock { Get-Process | Me
asure-Object | Format-Table -Property Count }

Id     Name            PSJobTypeName   State         HasMoreData     Location
--     ----            -------------   -----         -----------     --------
1      Job1            BackgroundJob   Running       True            localhost


[localhost]: PS C:\Users\anr\Documents> Receive-Job -Name Job1

Count
-----
  189


[localhost]: PS C:\Users\anr\Documents> Remove-Job -Name Job1
[localhost]: PS C:\Users\anr\Documents> Exit-PSSession
```

Remote Session
Schematischer Ablauf

Schritt	Beschreibung	Cmdlets
1	Neue Session erzeugen und in Variable speichern	`$s=New-PSSession -ComputerName <Ziel>`
2	Remote Jobs starten	`Invoke-Command -Session $s -ScriptBlock { Start-Job { … } }`
3	Ergebnisse der Jobs abfragen und in Variable speichern	`$result=Invoke-Command -Session $s -ScriptBlock { Receive-Job … }`

Remote Session
Hinweise

- Alle Hinweise zu lokalen Session gelten weiterhin
- Eine Session kann mit `Invoke-Command -Disconnected` zwischenzeitlich verlassen werden.
- Ergebnis eines Jobs lokal verwenden (in Variable `$result`):
  - `$result=Invoke-Command -Session $s -ScriptBlock { Receive-Job … }`
- Ergebnis eines Jobs remote speichern (in Datei *C:\xyz*):
  - `Invoke-Command -Session $s -Command { Receive-Job … | Out-File C:\xyz }`

Remote Session
Beispiel

```
PS C:\Windows\System32> $s=New-PSSession -ComputerName localhost
PS C:\Windows\System32> Invoke-Command -Session $s -ScriptBlock { Start-Job -ScriptBlock { Get-Process | Measure-Object
| Format-Table -Property Count } }

Id     Name            PSJobTypeName   State         HasMoreData     Location             Command                  PSC
                                                                                                                  omp
                                                                                                                  ute
                                                                                                                  rNa
                                                                                                                  me
--     ----            -------------   -----         -----------     --------             -------                  ---
1      Job1            BackgroundJob   Running       True            localhost             Get-Process | Measure-O… lo…

PS C:\Windows\System32> $result=Invoke-Command -Session $s -ScriptBlock { Receive-Job -Name Job1 -Keep }
PS C:\Windows\System32> Invoke-Command -Session $s -Command { Receive-Job -Name Job1 | Out-File -FilePath "C:\Users\anr\
Downloads\ps.txt" }
```

Scheduled Jobs

- Automatisierung der klassischen Windows Aufgabenplanung (Task Scheduler)
- Jobs können mit Task Scheduler oder PowerShell verwaltet werden
- Scheduled Jobs heißen bei deutschem Windows *Aufgaben*

- Unterschied zu normalen Jobs:
  - Laufen häufig wiederkehrend (wie GNU/Linux cronjobs)
  - Ausführung hängt von Bedingungen (Trigger) ab
  - Job kann Optionen besitzen
  - Registrierte Jobs bleiben im System erhalten, z.B. zum Modifizieren

Scheduled Jobs
Schematischer Ablauf

Schritt	Beschreibung	Cmdlets
1	Trigger definieren	`New-JobTrigger`
2	Optionen definieren	`New-ScheduledJobOption`
3	Job anlegen (registrieren)	`Register-ScheduledJob`

- Trigger und Optionen werden am besten in Variablen verwaltet
- Diese Trennung erhöht die Übersichtlichkeit
- Ein Scheduled Job sollte einen *sprechenden* Namen erhalten

Scheduled Jobs
Hinweise

- Scheduled Jobs sind standardmäßig nicht verfügbar.
- Das Module *PSScheduledJob* muss importiert werden.
- Problem: Durch Kompatibilitätsrichtlinien wird der Import blockiert
- Details: https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_windows_powershell_compatibility?view=powershell-7.3
  - Deny List in *$PSHOME\powershell.config.json* bearbeiten
  - Administratorrechte erforderlich (am besten aus Admin PowerShell aufrufen)
  - Eintrag zu *PSScheduledJob* löschen, Datei speichern
  - PowerShell beenden

- Neue PowerShell als Administrator starten
- `Import-Module –Name PSScheduledJob -SkipEditionCheck`

Scheduled Jobs
Hands-on

- Es soll ein Scheduled Job (Aufgabe) mit dem Namen *Prozessuebersicht* angelegt werden. Dieser soll die Prozesse auf einem System auflisten und in einer Logdatei speichern. Die weiteren Randbedingungen sind:
  - Der Job soll nur dienstags und mittwochs laufen.
  - Der Job soll um 09:00 Uhr morgens ausgeführt werden.
  - Der Job soll mit erhöhten Rechten ausgeführt werden (ist hier theoretisch nicht nötig).
  - Wenn der Job sich im Hibernate- oder Sleep-Zustand befindet, soll er „aufgeweckt“ werden, damit der Job auf jeden Fall läuft.
  - Der Job soll auch dann ausgeführt werden, wenn das Netzteil nicht angeschlossen ist und das Zielsystem über den Akku betrieben wird.

Scheduled Jobs
Beispiel

- Unter Windows Server 2019
- `Import-Module PSScheduledJob` vorher ausführen

```
PS C:\Users\Administrator> $trigger=New-JobTrigger -Weekly -DaysOfWeek "Tuesday","Wednesday" -At "09:00"
PS C:\Users\Administrator> $jOptions=New-ScheduledJobOption -StartIfOnBattery -RunElevated -WakeToRun
PS C:\Users\Administrator> Register-ScheduledJob -Name "Prozessuebersicht" -ScriptBlock { Get-Process | Out-File "C:\use
rs\Administrator\Downloads\ps.txt" } -Trigger $trigger -ScheduledJobOption $jOptions

Id         Name            JobTriggers     Command                                 Enabled
--         ----            -----------     -------                                 -------
1          Prozessueber... 1               Get-Process | Out-File "C:\users\Adm... True
```

Übung PS73

- Installierte HotFixes auswerten
- Lokale und Remotesessions mit Jobs starten
- Einen Scheduled Job konfigurieren und testen
  - nur unter Windows Server

Workflows und Parallelisierung

Workflows

- Workflows sind nur bis einschließlich PowerShell 5 verfügbar!
- Häufig auftretende Arbeitsabläufe werden zu Workflows zusammengefasst
- Ausführung durch die Windows Workflow Foundation (WWF)
- PowerShell (<=5) Schlüsselwort ist `Workflow`
- Besonders geeignet für Parallelisierung von Aufgaben
  - z.B. Skripte, die auf mehreren Rechnern gleichzeitig laufen

Workflows
Beispiel

- Ausgabe von *Hello World!* auf *stdout*

- Wichtige Parameter für alle Workflows
  - `-AsJob` um den Workflow als Job auszuführen
  - `-PSComputerName` für das Zielsystem des Workflows
  - `-PSCredentials` für das Benutzerkonto, unter dem der Workflow läuft
  - `-JobName` für eine passende Benennung des Jobs

Workflows
Parallelisierung

- Die WWF ermöglicht weitere Steuerungsfunktionen
  - z.B. explizit angeforderte Parallelverarbeitung (Nebenläufigkeit)
  - Nur für Workflows (in der WWF), nicht für andere (PS-)Skripte

- Kenntnisse von WWF und Workflows nützen nur bei Legacy-PS
  - Aber was machen wir nun in PowerShell 7 ohne Workflows?

Parallelisierung ab PowerShell 7

- Das Cmdlet `ForEach-Object` hat nun den Parameter -Parallel
  - ermöglicht nebenläufige Ausführung von Skriptblöcken
  - erst ab Version 7 verfügbar
  - Version 6 hat weder Workflows noch ForEach -Parallel !
- Wichtiger Parameter -*ThrottleLimit*
  - legt maximale Anzahl der parallelen Blöcke fest
  - Standardwert ist 5
- Die Ausführungszeit von Befehlen kann mit `Measure-Command` gemessen werden.

Parallelisierung ab PowerShell 7
Beispiel

- Ausgabe von 5 Zeilen mit einer Sekunde Pause (sequentiell)

```
PS C:\Users\anr> 1..5 | ForEach-Object { Write-Host "Nummer $_"; Start-Sleep -Se
conds 1 }
```

- Demo

- Ausführungsdauer

```
PS C:\Users\anr> (Measure-Command -Expression { 1..5 | ForEach-Object { "Nummer
$_"; Start-Sleep -Seconds 1 } }).Seconds
5
```

Parallelisierung ab PowerShell 7
Beispiel

- Ausgabe von 5 Zeilen mit einer Sekunde Pause (parallel)

```
PS C:\Users\anr> 1..5 | ForEach-Object -Parallel { Write-Host "Nummer $_"; Start
-Sleep -Seconds 1 }
```

- Demo

- Ausführungsdauer

```
PS C:\Users\anr> (Measure-Command -Expression { 1..5 | ForEach-Object -Parallel
{ Write-Host "Nummer $_"; Start-Sleep -Seconds 1 } }).Seconds
1
```

Parallelisierung ab PowerShell 7
Beispiel

- Im Beispiel wird Nebenläufigkeit maximal ausgenutzt.

```
PS C:\Users\anr> 1..5 | ForEach-Object -Parallel { Write-Host "Nummer $_"; Start
-Sleep -Seconds 1 } -ThrottleLimit 3
```

- Wie lange dauert jetzt wohl die Ausführung ?

Parallelisierung
Verwendung bei Jobs

- Ursprüngliches Beispiel als Job

```
PS C:\Users\anr> $job = 1..5 | ForEach-Object -Parallel { Write-Host "Nummer $_"
; Start-Sleep -Seconds 1 } -AsJob
```

- Hole Ergebnisse wie gewohnt mit `Receive-Job` ab
  - Problem: wann sind alle nebenläufigen Stränge fertig ?
  - Lösung: verwende `Wait-Job`

- `Wait-Job` wartet auf das Beenden eines Jobs
  - Job-Ergebnisse in Pipeline weiterleiten

```
PS C:\Users\anr> $job | Wait-Job | Receive-Job
```

Parallelisierung
Hinweise und Best Practices

- Parallelisierung nur mit `ForEach-Object` (Cmdlet), nicht mit *ForEach* (als Schlüsselwort für Schleife)

- Parallelisierung verwenden…
  - bei rechenintensiven Skripten
  - bei Skripten, die auf ein Ereignis warten

- Parallelisierung nicht verwenden…
  - bei Befehlen mit kurzer Ausführungszeit (Overhead!)

Übung PS74

- Nebenläufige Skriptblöcke erstellen
- Laufzeitmessung von Befehlen
- Sequentielle und parallele Ausführung vergleichen