



PowerShell für Einsteiger

Grundlagen und Lernsituationen

© Julius Angres 2023

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
- ▶ Serveraufgaben, Loganalyse, Webzugriff, Jobs
- ▶ Programmierung mit PowerShell (ps1-Skripte, Zugriff auf .NET-Objekte)
- ▶ Klassenarbeiten und Lernsituationen mit PowerShell erstellen und vorstellen

PowerShell Skripte anlegen

Erstellen von ps1-Dateien, Ausführungsrichtlinie

PowerShell Skripte erstellen

- ▶ PowerShell Skripte sind Textdateien (wie Batch/bash-Skripte)
- ▶ Standard-Dateiendung ist .ps1
- ▶ PowerShell Integrated Scripting Environment (ISE)
 - Editor mit Syntaxhighlighting (oberer Bereich)
 - PowerShell Session (unterer Bereich)
 - Nur für Windows verfügbar
 - Als Administrator starten wegen der Ausführungsrichtlinie von Skripten

 **Demo**

PowerShell Skripte ausführen

- ▶ Ausführen per PowerShell durch Eingabe des Dateinamens mit „.\“ davor.
 - ▶ Ist die PS nicht im Verzeichnis des Skripts, relativen oder absoluten Pfad zur Skriptdatei verwenden.
 - ▶ Bsp.: PS> .\skript.ps1
- ▶ **Achtung:**
 - Ausführung von Batch-Dateien ist standardmäßig erlaubt
 - Ausführung von PS-Skripten muss per Execution Policy explizit erlaubt werden
- ▶ Ausführungsrichtlinien abfragen und ändern
 - Get-ExecutionPolicy
 - Set-ExecutionPolicy

Ausführungsrichtlinie

- ▶ PowerShell kennt 7 verschiedene Ausführungsrichtlinien (Execution Policies)
- ▶ Die Richtlinie kann für verschiedene User mit `-Scope` festgelegt werden.
- ▶ Restricted:
 - keine Ausführung irgendwelcher Skripte zugelassen
 - Standard für Windows Clients
- ▶ RemoteSigned:
 - Skripte aus dem Internet benötigen vertrauenswürdige Signatur
 - Standard für Windows Server

Ausführungsrichtlinie

- ▶ Unrestricted:
 - Alle Skripte dürfen ausgeführt werden
 - Standard für Nicht-Windows Computer
- ▶ Außerdem AllSigned, Bypass, Default, Undefined
- ▶ Setzen einer Ausführungsrichtlinie benötigt Administratorenrechte

👉 **Entscheidet selbst über eure Ausführungsrichtlinie.**

Übung PS81

Programmierung mit PowerShell

- ▶ Skriptdateien anlegen
- ▶ Verknüpfung mit geeignetem Editor herstellen
- ▶ Einfache Skripte erstellen
- ▶ Ausführungsrichtlinie konfigurieren

Subsessions in cmd.exe

Zugriff auf Legacy cmd-Umgebung innerhalb PowerShell

Subsessions in cmd

- ▶ Mit dem Befehl `cmd` wird in einer PS-Session eine cmd gestartet
- ▶ Dort sind nur cmd-Befehle verfügbar
- ▶ Verlassen der Subsession mit `exit`
- ▶ Nützlich für Verarbeitung von Legacy-Batch-Programmen o.ä.

Übung PS82

Programmierung mit PowerShell

- ▶ Eine cmd Subsession aufrufen
- ▶ Batchprogrammierung innerhalb der Subsession

Variablen und Kontrollstrukturen

Variablennotation, Datentypen, Verzweigung, Schleifen

Kontrollfluss: Sequenz, Struktur

- ▶ Skriptdatei wird von oben nach unten zeilenweise verarbeitet
- ▶ Sequenzierung von Befehlen in einer Zeile per Semikolon ;
- ▶ Anweisungsblöcke in geschweiften Klammern { }
- ▶ Syntax wie in C# (oder Java)
- ▶ Funktionen mit Schlüsselwort Function definierbar

Variablen

- ▶ Variablen mit \$ vor dem Namen definierbar, z.B. \$a=0, \$str="anr"
- ▶ Explizite Typisierung von Variablen *möglich*
- ▶ Explizite Typannotation mit [typ]::\$var

```
PS C:\Users\anr> [string]$str="anr"  
PS C:\Users\anr> [string]$a=10
```

- ▶ \$str ist explizit ein String
- ▶ \$a wird auch zum String

```
PS C:\Users\anr> $a*10  
101010101010101010
```

- ▶ \$a verhält sich auch wie ein String 😊

Variablen Cmdlets

- ▶ Cmdlets für die Verwaltung von Variablen in der Familie *Variable*.

```
PS C:\Users\anr\Downloads\App2> gcm -Noun Variable

CommandType      Name
-----
Cmdlet            Clear-Variable
Cmdlet            Get-Variable
Cmdlet            New-Variable
Cmdlet            Remove-Variable
Cmdlet            Set-Variable
```

- ▶ Konstanten anlegen mit -Option Constant
 - ▶ Änderungen erfordern Neustart der PowerShell
- ▶ Beim Entfernen der Variablen das Dollarzeichen \$ weglassen.

Listen, Arrays, Dictionaries

- ▶ Listen in runden Klammern anlegen
- ▶ Zugriff mit eckigen Klammern und Index (0-indiziert)
- ▶ Konkatination von Listen mit +

- ▶ Arrays, Dictionaries mit @-Symbol anlegen.
- ▶ Leeres Dictionary mit `$dict = @{}` anlegen.
- ▶ Dictionary wird (weitgehend) verwendet z.B. wie in Python.

Dictionary Beispiel

```
PS C:\Users\anr\Downloads\App2> $arr=@{}  
PS C:\Users\anr\Downloads\App2> $arr['anr']="Angres"  
PS C:\Users\anr\Downloads\App2> $arr['anr']  
Angres  
PS C:\Users\anr\Downloads\App2> $arr['anr'] += ",Julius"  
PS C:\Users\anr\Downloads\App2> $arr['anr']  
Angres,Julius
```

Datentypen: Zahlen

- ▶ Zahlendatentypen
 - Int, UInt, long, byte, short, ...
- ▶ Arithmetische Standardoperationen (auch modulo %) direkt als infix verfügbar
- ▶ Spezielle Funktionen in Bibliothek [math]
- ▶ Z.B. Potenz 2^{10}

```
PS C:\Users\anr> [math]::Pow(2,10)  
1024
```

- ▶ Gleitkommazahlen in einfacher und doppelter Genauigkeit verfügbar

Datentypen: Strings

- ▶ In doppelten Anführungszeichen:
 - Variablenausdrücke werden dynamisch ersetzt

```
PS C:\Users\anr> $str="anr"  
PS C:\Users\anr> Write-Host "Hallo $str"  
Hallo anr
```

- ▶ In einfachen Anführungszeichen:
 - Keine Ersetzung (Literal String)

```
PS C:\Users\anr> $str="anr"  
PS C:\Users\anr> Write-Host 'Hallo $str'  
Hallo $str
```

Datentypen: Boolesche Werte

- ▶ Wahrheitswerte sind vordefiniert in `System.Boolean`
- ▶ Man kann nicht mit teilweise rechnen (wie etwa in C/C++)
- ▶ `$True` ist das Verum
- ▶ `$False` ist das Falsum

Kontrollfluss: Verzweigung

- ▶ PowerShell kennt klassische If- und Switch-Anweisungen
- ▶ If benötigt nicht unbedingt ein Else

```
PS C:\Users\anr> If ($n % 2 -eq 0) { Write-Host "even" } Else { Write-Host "odd" }
```

Kontrollfluss: Wiederholungen

- ▶ PowerShell implementiert viele gängige Schleifentypen:
 - Schleife mit fester Durchlaufzahl und Indexvariable (For-Schleife)
 - Schleife mit fester Durchlaufzahl und Objektreferenz (ForEach-Schleife)
 - Schleifen ohne feste Durchlaufzahl (While-Schleife, Do-While-Schleife)
- ▶ Alle Schleifen sind semantisch äquivalent
- ▶ Running Example:
 - Ausgabe der natürlichen Zahlen von 1 bis 5

For-Schleife

- ▶ PowerShell bietet eine C-Style For-Schleife

```
PS C:\Users\anr> For ($i = 1; $i -le 5; $i++) { Write-Host "$i" }  
1  
2  
3  
4  
5
```

- ▶ Variablenreferenz ist `$i`
- ▶ Auf Vergleichsoperator achten (`-le` statt `<=` wie in C/C#/Java etc.)
- ▶ Die Anführungszeichen im Beispiel sind optional.

ForEach-Schleife

- ▶ PowerShell bietet zwei Möglichkeiten für Iteration über Listen:
- ▶ Eine ForEach-Schleife (wie in gängigen Sprachen)

```
PS C:\Users\anr> ForEach ($i in 1..5) { Write-Host $i }  
1  
2  
3  
4  
5
```

- ▶ Variablenreferenz ist `$i`
- ▶ Kein Vergleichsoperator notwendig

ForEach-Schleife

- ▶ PowerShell bietet zwei Möglichkeiten für Iteration über Listen:
- ▶ Das Cmdlet `ForEach-Object` (da PowerShell objektorientiert ist)

```
PS C:\Users\anr> 1..5 | ForEach-Object { Write-Host $_ }  
1  
2  
3  
4  
5
```

- ▶ Variablenreferenz ist `$_` (anonym, ähnlich Lambda in F# oder `this` in Java)
- ▶ `ForEach-Object` wird über die Pipeline gefüttert.
- ▶ Allgemein `$LO..$HI | ForEach-Object { ... }`

While-Schleife

- ▶ Klassische While-Schleife mit Schlüsselwort While
- ▶ Es gibt auch Do-While und Do-Until...

```
PS C:\Users\anr> $i=1
PS C:\Users\anr> While ($i -le 5) { Write-Host $i; $i++ }
1
2
3
4
5
```

- ▶ Variablenreferenz ist \$i
- ▶ Variable muss außerhalb der Schleife initialisiert werden

Übung PS83

Programmierung mit PowerShell

- ▶ Variablen verschiedener Datentypen anlegen und verwalten
- ▶ Verzweigung mit If-Strukturen
- ▶ Verschiedene Schleifen anwenden
- ▶ Vor- und Nachteile der jeweiligen Schleifentypen erkennen

Ranges und Pipeline

Range-Notation, PowerShell Pipeline und Fortschrittsbalken nutzen

Ranges

- ▶ Ranges können mit zwei Doppelpunkten definiert werden.
- ▶ Voraussetzung: Indexmenge mit Ordnungsrelation
 - Eine Ordnungsrelation ist reflexiv, transitiv und antisymmetrisch
- ▶ Beispiele:
 - $(\text{Integer}, \leq)$ natürliche Ordnung
 - $(\text{Char}, \leq_{\text{Lex}})$ lexikographische Ordnung
 - $(\text{Boolean}, \leq_{\text{B}})$ Falsum ist kleiner als Verum

Ranges

▶ Beispiele:

- `1..10` → Liste der ganzen Zahlen von 1 bis 10 (jeweils inklusive)
- `'a'..'z'` → Liste der Kleinbuchstaben des englischen Alphabets
- `$False..$True` → Liste der Zahlenwerte 0,1 zu den Wahrheitswerten

▶ Verwendung des Pipe-Operators | wie am Prompt möglich

Übung PS84

Programmierung mit PowerShell

- ▶ Mit Ranges arbeiten
- ▶ Die Pipeline in Skripten verwenden

Fortschrittsbalken

- ▶ Möglichkeit zur Aufwertung der Ausgabe
- ▶ Im Cmdlet `Write-Progress` implementiert
 - Informiert Benutzer über aktuellen Status einer Aktivität
 - Nützlich für lange laufende, rechenintensive Befehle
 - Aktivität kann beschreiben, was passiert
 - Status gibt Rückmeldung zum aktuellen Stand, z.B. Prozentanteile
 - Wichtige Parameter: *-Activity*, *-Status*

Fortschrittsbalken

Beispiel

- ▶ Simuliertes Suchen als zwanzigfaches Durchlaufen einer Schleife mit Fortschrittsbalken, der den prozentualen Fortschritt anzeigt.
- ▶ Setzen einer Variable auf den Wert 20 (damit der Wert nicht *hard coded* ist)

```
PS C:\Users\anr> $n = 20
PS C:\Users\anr> For ($i = 1; $i -le 20; $i++) { Write-Progress -Activity "Suche.."
-Status "$($i/$n*100)% abgeschlossen"; Start-Sleep -Milliseconds 100 }
```

- ▶  Demo

Fortschrittsbalken

Hinweise

- ▶ Eine künstliche Wartezeit von mind. 100 ms, damit der Balken erkennbar ist.
- ▶ Prozentualer Fortschritt als Status kann mit dem Ausdruck `"$ ($i/$n*100) %"` berechnet werden. Dabei ist i die Laufvariable der Schleife und n die Gesamtzahl.
- ▶ **Achtung:**
 - Anführungszeichen beachten (Status ist ein String)
 - Dollarzeichen um den berechneten Ausdruck (da innerhalb eines Strings)

Funktionen in PowerShell Skripten

User Defined Functions

Funktionen definieren

- ▶ Eigene PowerShell Funktionen im Skript definieren:
 - Schlüsselwort *Function*
 - Rumpf der Funktion in geschweiften Klammern (wie C/C++, C#, Java, ...)
 - Parameter **im Rumpf** der Funktion mit *param(...)*
 - Aufruf mit Namen und Parametern wie bei Cmdlets

Funktionen definieren

Nomenklatur und Aufruf

- ▶ Grundsätzlich sind Bezeichner (fast) frei wählbar
 - übliche Einschränkungen wie Schlüsselworte, etc.
- ▶ Konventionell sollte Verb-Noun-Syntax verwendet werden
 - zumindest für exportierte, d.h. von außen zugängliche Funktionen
 - nicht unbedingt bei kleinen Hilfsfunktionen
 - das ist kein Zwang, aber strongly recommended

Funktionen definieren

Nomenklatur und Aufruf

- ▶ Funktionen werden wie Cmdlets mit ihrem Namen aufgerufen
 - Parameter mit Leerzeichen dahinter
 - Parameter können wie bei Cmdlets als Named Parameter genutzt werden
- ▶ Um Funktionen im Scope einer PS Session zu haben, muss das Skript, das die Funktion enthält, per dot-Sourcing eingebunden werden:

```
PS C:\Users\anr\Downloads> . .\MyScript.ps1
```
- ▶ Nun sind alle Funktionen aus MyScript verfügbar
 - auch mit Hilfe und Tab-Completion!

Funktionen definieren

Aufruf von Parametern

- ▶ Funktion mit einem Parameter

```
Function MySingleFun { param($i) Write-Output $i }
```

- ▶ Aufruf:

```
PS C:\Users\anr> MySingleFun 10
```

- wie bei einem Cmdlet oder in funktionalen Sprachen (F#/Haskell)

- ▶ ...oder auch so:

```
PS C:\Users\anr> MySingleFun(10)
```

- wie bei C#/Java

Funktionen definieren

Aufruf von Parametern

- ▶ Funktion mit zwei Parametern

```
PS C:\Users\anr> Function MyTwinFun { param($i,$j) Write-Output "$i $j" }
```

- ▶ Aufruf:

```
PS C:\Users\anr> MyTwinFun 10 20
```

- ▶ ...oder auch so:

```
PS C:\Users\anr> MyTwinFun(10,20)
```

- ▶ Funktioniert auch, wenn ein Parameter eine Liste ist

Funktionen definieren

Eigene Named und Switch Parameter

- ▶ Alle definierten Parameter können automatisch wie Named Parameter bzw. Switch Parameter verwendet werden.
- ▶ Beispiel:
- ▶ Definition einer Funktion, die alle (lokalen) Benutzer und Gruppen auflistet.
- ▶ Parameter:
 - *\$User* ein aufzulistender einzelner Benutzer (als String)
 - *\$NoGroups* ein Switch Parameter, der die Gruppen nicht mit auflistet

Funktionen definieren

Eigene Named und Switch Parameter

► Demo:

- Kurze Quelltextbetrachtung zur Funktion
- Verschiedene Beispielaufrufe mit den Parametern

Funktionen definieren

Beispiel

- ▶ Eine Funktion, die die ersten 3 Prozesse (alphabetisch sortiert) anzeigt.

```
PS C:\Users\anr> Function Get-FirstThreeProcess { Get-Process | Select-Object -First 3 }  
PS C:\Users\anr> Get-FirstThreeProcess
```

Funktionen definieren

Beispiel

- ▶ Erweiterung bzw. Generalisierung:
 - Eine Funktion, die die ersten n Prozesse (alphabetisch sortiert) anzeigt.

```
PS C:\Users\anr> Function Get-FirstNProcess { Get-Process | Select-Object -  
First $n }
```

- ▶ Problem:
 - Variable n muss vor dem Funktionsaufruf deklariert und gesetzt werden.

```
PS C:\Users\anr> $n=3  
PS C:\Users\anr> Get-FirstNProcess
```

Funktionen definieren

Parameter

► Erweiterung bzw. Generalisierung:

- Die Funktion erhält einen Parameter *\$Number*
- Das Schlüsselwort *param* kann groß oder klein geschrieben werden

```
PS C:\Users\anr> Function Get-FirstNProcess { param($Number) Get-Process |  
Select-Object -First $Number }  
PS C:\Users\anr> Get-FirstNProcess 3
```

► Problem:

- Variable *\$n* muss mit dem korrekten Typ verwendet werden.
- Aufrufe führen sonst zu schwer vorhersagbaren ungewünschtem Verhalten

👉 Was liefert GetFirstNProcess mit 3.01, 3.5, “3“, “drei“ oder \$True ?

Funktionen definieren

Parameter

- ▶ Erweiterung bzw. Generalisierung:
 - Die Funktion erhält einen Parameter *\$Number* vom Typ *int*

```
PS C:\Users\anr> Function Get-FirstNProcess { param([int]$Number) Get-Processes | Select-Object -First $Number }  
PS C:\Users\anr> Get-FirstNProcess 3
```

- ▶ Problem:
 - Was passiert, wenn der Parameter beim Aufruf vergessen wird ?
 - Entscheidung: ErrorAction, Exception, Parameterhandling verändern ?

Funktionen definieren

Parameter

- ▶ Erweiterung bzw. Generalisierung:
- Die Funktion erhält einen Parameter *\$Number* vom Typ *int*, der zwingend angegeben werden muss

```
Function Get-FirstNProcess {  
    param(  
        [Parameter(Mandatory=$True)]  
        [int]$Number  
    )  
    Get-Process | Select-Object -First $Number  
}
```

- Problem:
 - Ein Zwang zum Nutzen des Parameters ist nicht immer gewünscht.

Funktionen definieren

Mandatory Parameter

► Bestimmungsgemäßer Aufruf

```
PS C:\Users\anr\Downloads> Get-FirstNProcess 3
```

NPM(K)	PM(M)	WS(M)	CPU(s)	Id	SI	ProcessName
-----	-----	-----	-----	--	--	-----
16	3,63	12,48	0,00	2924	0	AppHelperCap
27	20,20	14,13	0,36	9464	2	ApplicationFrameHost
39	26,50	13,60	0,00	12056	0	AWACMClient

Funktionen definieren

Mandatory Parameter

- ▶ Aufruf ohne Parameter („vergessener Parameter“)

```
PS C:\Users\anr\Downloads> Get-FirstNProcess  
  
cmdlet Get-FirstNProcess at command pipeline position 1  
Supply values for the following parameters:  
Number:
```

- ▶ Kein Abbruch, sondern Benutzerinteraktion
- ▶ Name des Parameters wird angezeigt
- ▶ Wert kann on-the-fly gegeben werden
- ▶ Funktion läuft danach normal weiter

Funktionen definieren

Parameter mit Standardwert

- ▶ Erweiterung bzw. Generalisierung:
- Die Funktion erhält einen Parameter *\$Number* vom Typ *int*, der den Standardwert 3 hat

```
Function Get-FirstNProcess {  
    param([int]$Number=3)  
    Get-Process | Select-Object -First $Number  
}
```

Funktionen definieren

Parameter mit Standardwert

- ▶ Aufruf ohne Parameter („vergessener Parameter“)

```
PS C:\Users\anr\Downloads> Get-FirstNProcess
```

NPM(K)	PM(M)	WS(M)	CPU(s)	Id	SI	ProcessName
-----	-----	-----	-----	--	--	-----
16	3,63	12,43	0,00	2924	0	AppHelperCap
27	20,20	14,13	0,34	9464	2	ApplicationFrameHost
39	26,50	13,58	0,00	12056	0	AWACMClient

- ▶ Parameter *\$Number* wird automatisch auf Wert 3 gesetzt
- ▶ No further action warranted 😊

Funktionen definieren

Dokumentation

- ▶ (Wichtige, große) Funktionen sollten dokumentiert werden
- ▶ Durch Kommentare im Quelltext
- ▶ Und durch spezielle Beschreibungssprache (wie z.B. Javadoc)
- ▶ Wichtige Tags sind
- ▶ `.SYNOPSIS`, `.DESCRIPTION`, `.PARAMETER`
- ▶ Diese Beschreibung wird bei Aufruf mit `Get-Help` angezeigt!

Funktionen definieren

Dokumentation

► Beispiel: Selbstgeschriebene Funktion *Set-NtfsPermissions*

```
<#  
.SYNOPSIS  
configure NTFS permissions for a folder  
.DESCRIPTION  
sets desired NTFS permissions for an existing object or an object that is to be created  
.PARAMETER Folder  
the folder the permissions of which are to be created resp. updated  
.PARAMETER BreakInheritance  
breaking up existing inherited permissions is disabled by default  
.PARAMETER FA  
list of comma separated principals to grant them FullAccess  
.PARAMETER MA  
list of comma separated principals to grant them ModifyAccess  
.PARAMETER RA  
list of comma separated principals to grant them ReadAccess  
#>
```

Funktionen definieren

Dokumentation

► Beispiel: Selbstgeschriebene Funktion *Set-NtfsPermissions*

```
PS C:\Users\anr> Get-Help Set-NtfsPermissions

NAME
    Set-NtfsPermissions

SYNOPSIS
    configure NTFS permissions for a folder

SYNTAX
    Set-NtfsPermissions [[-Folder] <String>] [-BreakInheritance] [[-FA]
    <Array>] [[-MA] <Array>] [[-RA] <Array>] [<CommonParameters>]

DESCRIPTION
    sets desired NTFS permissions for an existing object or an object
    that is to be created
```

Funktionen definieren

Hinweise

- ▶ Funktionen definieren ist grundsätzlich leicht
- ▶ Parameter im *param* Tag **innerhalb** des Funktionsrumpfes
- ▶ Aufruf mit Namen, Parameter ohne Klammern (empfohlen)
 - nicht wie in C#/Java
 - genauso wie in F#/Haskell
- ▶ Werte in runden Klammern werden als Listen interpretiert
 - normalerweise kein Unterschied

Funktionen definieren

Hinweise

- ▶ Parameter können verfeinert werden...
 - durch Typangabe (vor dem Namen in eckigen Klammern)
`[int]$Number`
 - durch Mandatory-Klausel zur Pflichtangabe gemacht werden
`[Parameter(Mandatory=$True)]`
 - durch Angeben eines Standardwertes
`[int]$Number=3`

Funktionen definieren

Ausblick

- ▶ Es gibt noch weitere Details zum Verwenden von Funktionen:
- ▶ Möglichkeit, Ausführung explizit zum Beginn (beim Betreten der Funktion) oder am Ende (kurz vor Verlassen der Funktion) auszuführen.
- ▶ ...und noch andere fortgeschrittene Konzepte

Übung PS85

Programmierung mit PowerShell

- ▶ Mit Fortschrittsbalken arbeiten
- ▶ Klassische mathematische Funktionen implementieren
- ▶ Parameter und weitere Strukturen in Funktionen verwenden

Arbeit mit COM- und .NET-Objekten

Objekte erstellen, Automatisierung von Office, System Tray Icons

Das Component Object Model (COM)

- ▶ Von Microsoft entwickelt
- ▶ Wurde 1992 mit Windows 3.1 eingeführt
- ▶ Dient der Kommunikation zwischen Prozessen
- ▶ Ermöglicht z. B. Zugriff auf Office-Applikationen

Das Component Object Model (COM)

- ▶ COM-Objekte sind unter Windows (immer noch) häufig anzutreffen
- ▶ Bieten vielfältige Funktionalitäten „frei Haus“
- ▶ Können direkt in PowerShell erzeugt und verwendet werden.
 - `Cmdlet New-Object` mit Parameter `-ComObject`
- ▶ Ziel häufig: Automatisierung von Office (Word, Excel, Access)

PowerShell und COM-Objekte

Beispiel Excel-Dokument

► Aufgabe:

- Ein Excel-Dokument via PowerShell Output befüllen

► Vorgehen:

- Passendes COM-Objekt erzeugen
- Excel-spezifische Objektstruktur (entsprechend COM) anlegen
 - Excel-Datei → Arbeitsmappe → Tabelle
- Daten schreiben
- Excel-Datei speichern und schließen

PowerShell und COM-Objekte

Beispiel Excel-Dokument

► Umsetzung:

```
PS C:\Users\anr> $dokument=New-Object -ComObject Excel.Application
PS C:\Users\anr> $mappe=$dokument.Workbooks.Add()
PS C:\Users\anr> $tabelle=$mappe.WorkSheets.Item(1)
PS C:\Users\anr> $tabelle.Cells.Item(1,1)="Windows PowerShell"
PS C:\Users\anr> $tabelle.Cells.Item(1,2)="Fortbildung"
PS C:\Users\anr> $tabelle.Cells.Item(2,1)="=2+2"
```

PowerShell und .NET-Objekte

- ▶ In PowerShell können jederzeit und überall .NET-Objekte erzeugt werden
- ▶ Benötigte Module müssen ggf. vorher geladen werden, z.B. die Windows Forms für das Erzeugen von GUI Elementen
- ▶ Graphische Objekte können `MessageBox`, etc. sein (grundsätzlich jedes Objekt, das ein Windows Form ist).

PowerShell und .NET-Objekte

Beispiel Message Box

- ▶ Aufgabe:
 - Erzeugen einer Message Box mit Begrüßungsnachricht.
- ▶ Vorgehen:
 - Laden der Windows Forms
 - Statischer Zugriff auf *MessageBox* Objekt
 - Nachricht über passende Methode anzeigen

PowerShell und .NET-Objekte

Beispiel Message Box

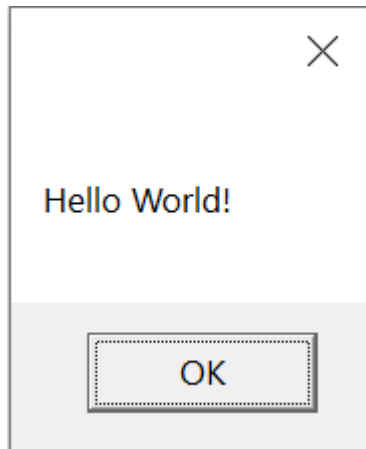
► Umsetzung:

```
PS C:\Windows\System32> [System.Reflection.Assembly]::LoadWithPartialName("PresentationFramework")

GAC      Version      Location
---      -
False    v4.0.30319    C:\Program Files\PowerShell\7\PresentationFramework.dll

PS C:\Windows\System32> [System.Windows.MessageBox]::Show("Hello World!")
OK
```

► Ergebnis:



PowerShell und .NET-Objekte

Beispiel RSS Feed

▶ Aufgabe:

- Einen RSS Feed auslesen

▶ Vorgehen:

- Objekt vom Typ *Net.Webclient* erzeugen
- Website mit der passenden URL herunterladen
- Ergebnis als XML-Dokument interpretieren
- Zugriff auf gewünschte Properties

PowerShell und .NET-Objekte

Beispiel RSS Feed

► Umsetzung:

```
PS C:\Users\anr> $webclient=New-Object Net.WebClient
PS C:\Users\anr> $xmlpage=[xml]$webclient.DownloadString("https://devblogs.microsoft.com/powershell/feed/")
PS C:\Users\anr> $xmlpage.rss.channel.Item | Format-List -Property title, link
```

► Ergebnis:

```
title : PowerShell/OpenSSH Team Investments for 2023
link  : https://devblogs.microsoft.com/powershell/powershell-openssh-team-invest

title : PowerShell Extension for Visual Studio Code January 2023 Update
link  : https://devblogs.microsoft.com/powershell/powershell-extension-for-visua

title : PowerShellGet 3.0 Preview 18
link  : https://devblogs.microsoft.com/powershell/powershellget-3-0-preview-18/
```

Übung PS86

Programmierung mit PowerShell

- ▶ COM-Objekte anlegen
- ▶ Steuerung einer Excel-Datei mit PowerShell
- ▶ Einen zweiten RSS Feed Reader erstellen
- ▶ Message Boxes und System Tray Icons verwenden

Komplexe Programme, HOF

Programmierung umfangreicherer Anwendungen

Komplexe Programme

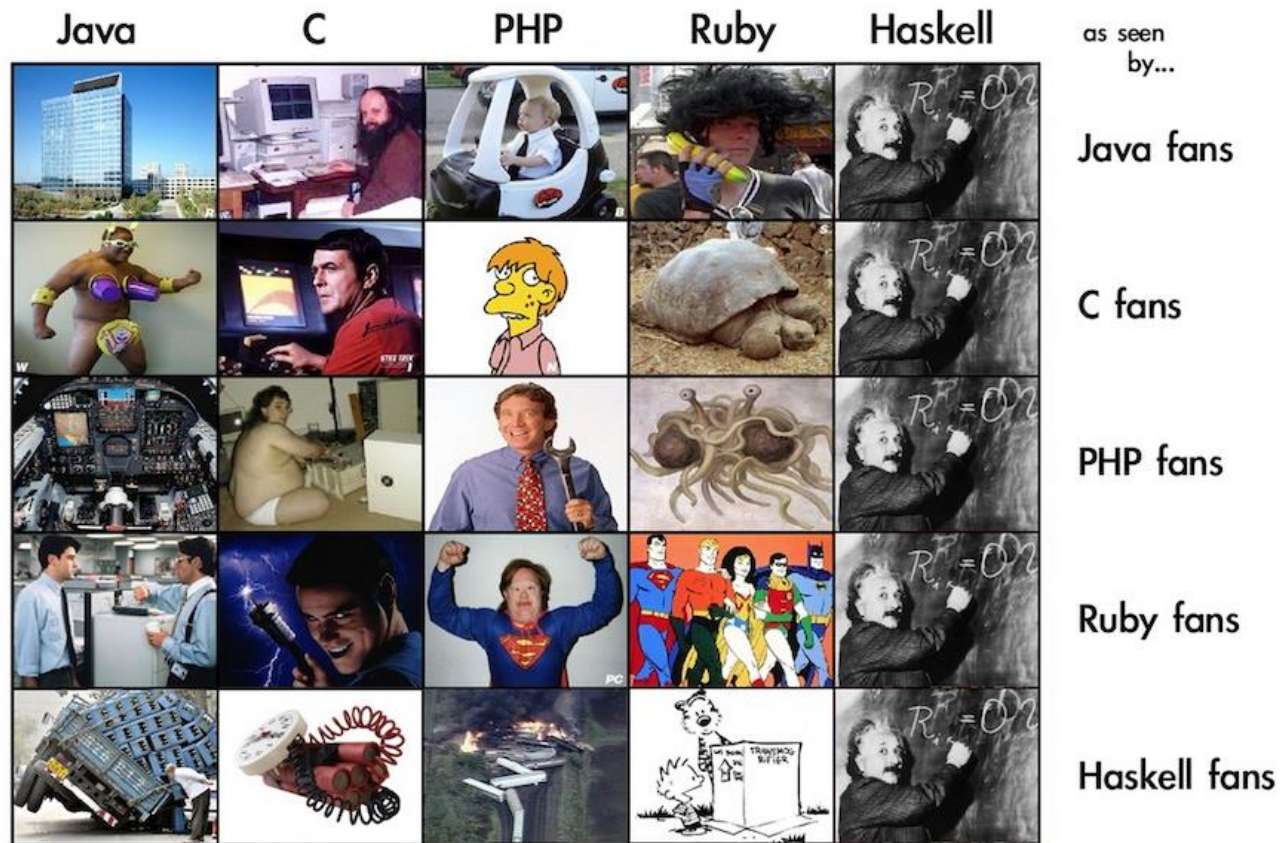
- ▶ Grundsätzlich sind beliebige Programme mit PowerShell erstellbar
- ▶ Beispiele (siehe Übung):
 - Cäsar-Verschlüsselung
 - 14-15-Puzzle

Higher Order Functions

- ▶ Funktionen höherer Ordnung (Higher Order Functions, kurz HOF) sind Funktionen, die eine Funktion als Parameter haben oder eine Funktion zurückliefern.
- ▶ HOF gehören klassisch zur funktionalen Programmierung.
- ▶ Mittlerweile funktionale Einflüsse in vielen Hochsprachen (Java, C#, Python)
- ▶ HOF ermöglichen ein wesentlich erhöhtes Abstraktionslevel.

Higher Order Functions

Code wird kürzer, übersichtlicher und cooler



Higher Order Functions

- ▶ Drei wichtige, fundamentale HOF:
 - Eine Funktion auf jedes Element einer Liste anwenden
 - Elemente einer Liste entsprechend einem Prädikat filtern
 - Eine Liste mit einer zweistelligen Funktion zu einem Wert zusammenfalten

Higher Order Functions

Scriptblocks

- ▶ In PowerShell können Ausdrücke, aber auch Funktionen als Variablen gespeichert werden.
- ▶ Der zugehörige Typ ist *Scriptblock*.
- ▶ Die in einer *Scriptblock*-Variable gespeicherte Funktion kann als Input für eine HOF genutzt werden.
- ▶ **Syntax:**
- ▶ `[scriptblock]$var = { <Expression> }`

Higher Order Functions

Scriptblocks

- ▶ Beispiel:
- ▶ Eine Funktion, die zur Eingabe die Konstante 5 addiert
- ▶ Klassische Funktionsdefinition:

```
PS C:\Users\anr> Function addiere5 { param($n) $n + 5 }
```

- ▶ Definition als Scriptblock:

```
PS C:\Users\anr> [scriptblock]$addiere5 = { $_ + 5 }
```

Higher Order Functions

Mapping

- ▶ Mapping heißt, eine Funktion auf alle Elemente einer Liste anzuwenden.
- ▶ Ergebnis ist (klassisch) eine Liste gleichen Typs.
- ▶ Entspricht in Haskell der Funktion *map* mit der Signatur
 $map :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]$
- ▶ In PowerShell wird Mapping durch das Cmdlet `ForEach-Object` implementiert
 - Ein Alias für `ForEach-Object` ist `%`

Higher Order Functions

Mapping Beispiel

- ▶ Eine Funktion, die zu jedem Element einer Liste 5 addiert.
- ▶ Klassische PowerShell-Lösung mit *Function addiere5*:

```
PS C:\Users\anr> 1..10 | ForEach-Object { addiere5 $_ }
```

- ▶ Lösung mit Scriptblock *\$addiere5*:

```
PS C:\Users\anr> 1..10 | % $addiere5
```

- ▶ Vorteil:
 - Scriptblock ist kürzer
 - Logik ist besser gekapselt (Wiederverwendbarkeit)

Higher Order Functions

Mapping Hinweise

- ▶ Generische Vorgehensweise:
- ▶ Definiere Funktion als Scriptblock *\$block*
- ▶ Definiere Input *\$input*
- ▶ Ausführung als Pipeline mit `ForEach-Object`:
- ▶ `PS> $input | % $block`
- ▶ Verkettung von Funktionen als Mapping-Pipeline umsetzbar

Higher Order Functions

Filter

- ▶ Filter heißt, ein Prädikat wird auf alle Elemente einer Liste angewendet.
- ▶ Ein Prädikat ist eine Funktion, die einen Wahrheitswert zurückliefert
- ▶ Ergebnis ist (klassisch) eine Liste gleichen Typs.
- ▶ Entspricht in Haskell der Funktion *filter* mit der Signatur
$$\text{filter} :: (a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a]$$
- ▶ In PowerShell wird Filtern durch das Cmdlet `Where-Object` implementiert
 - Ein Alias für `Where-Object` ist ?

Higher Order Functions

Filter Beispiel

- ▶ Beispiel:
- ▶ Eine Funktion, die prüft, ob ein Eingabewert eine gerade Zahl ist.
- ▶ Klassische Funktionsdefinition:

```
PS C:\Users\anr> Function gerade { param($n) If ($n % 2 -eq 0) { $True } Else { $False } }
```

- ▶ Definition als Scriptblock:

```
PS C:\Users\anr> [scriptblock]$gerade = { $_ % 2 -eq 0 }
```

Higher Order Functions


Filter Beispiel

- ▶ Eine Funktion, die prüft, ob ein Eingabewert eine gerade Zahl ist.
- ▶ Klassische PowerShell-Lösung mit *Function gerade*:

```
PS C:\Users\anr> 1..10 | Where-Object { gerade $_ }
```

- ▶ Lösung mit Scriptblock *\$gerade*:

```
PS C:\Users\anr> 1..10 | ? $gerade
```

- ▶  Welchen Output liefert `PS C:\Users\anr> 1..10 | % $gerade ?`

Higher Order Functions

Filter Hinweise

- ▶ Generische Vorgehensweise:
- ▶ Definiere Prädikat als Scriptblock *\$pred*
- ▶ Definiere Input *\$input*
- ▶ Ausführung als Pipeline mit `Where-Object`:
- ▶ `PS> $input | ? $pred`
- ▶ Es gibt auch ein Schlüsselwort *filter* → andere Geschichte

Higher Order Functions

Faltung

- ▶ Faltung heißt, eine List wird mithilfe einer zweistelligen Funktion zu einem Wert „zusammengefaltet“.
- ▶ Ergebnis ist (klassisch) ein Wert eines Typs (nicht notwendigerweise der Typ der Eingabeliste).
- ▶ Entspricht in Haskell der Funktion *foldr* mit der Signatur
$$\text{foldr} :: (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow [b]$$
- ▶ In PowerShell gibt es kein Cmdlet, das Faltung direkt implementiert
 - Das Modul *functional* aus der PowerShell Gallery bietet klassische HOF

Übung PS87

Programmierung mit PowerShell

- ▶ Größere strukturierte Programme erstellen
- ▶ Speech Synthesizer verwenden
- ▶ Cäsar-Verschlüsselung oder 15-Puzzle implementieren