Powershell Guide Documentation

Robert B.

27.04.2020

Inhaltsverzeichnis

KAPITEL 1

Was ist Powershell?

PowerShell (auch Windows PowerShell und PowerShell Core) ist ein plattformübergreifendes Framework von Microsoft zur Automatisierung, Konfiguration und Verwaltung von Systemen, bestehend aus einem Kommandozeileninterpreter sowie einer Skriptsprache.

PowerShell Core (auch PSCore) basiert auf der .NET Core Common Language Runtime (CoreCLR) und ist seit 2016 als plattformübergreifendes Open-Source-Projekt unter der MIT-Lizenz für Linux, macOS und Windows verfügbar.

Windows PowerShell basiert auf der Common Language Runtime (CLR) des .NET Frameworks und wird mit Windows als Teil des Windows Management Frameworks (WMF) unter einer proprietären Lizenz ausgeliefert. Seit 2016 gibt es auch die Windows PowerShell als Core Edition, welche wie PowerShell Core auf .NET Core basiert und als Teil von Windows Nano Server und Windows IoT ausgeliefert wird.

1.1 Powershell Basics

1.1.1 Hello World

Um in Powershell einen Text auszugeben gibt es die Funktion `Write-Host`.

Öffne eine Powershell Konsole und gib den folgenden Code ein:

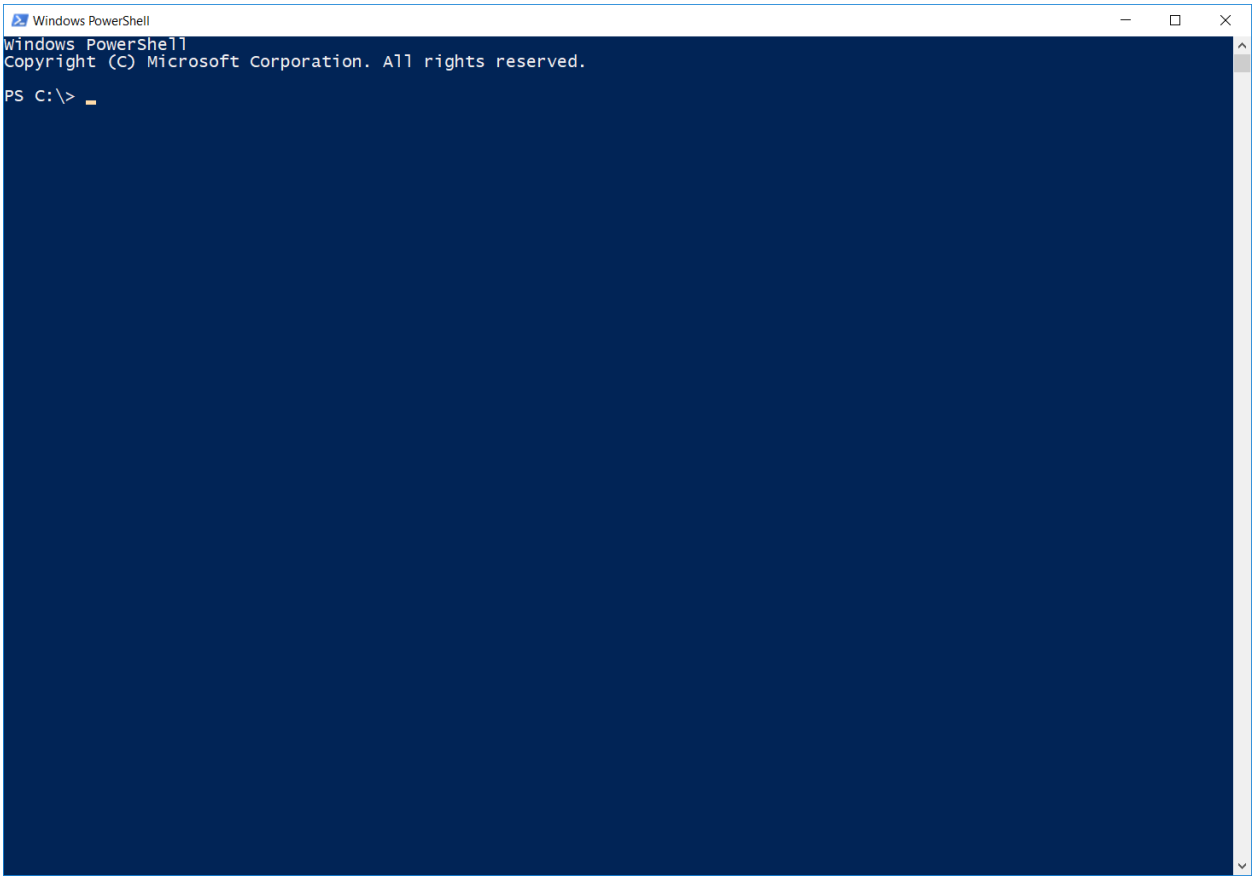
```
Write-Host "Hello World"
```

Sobald du dann diesen Ausführst sollte in der Konsole `Hello World` stehen.

In diesem Beispiel ist der Text `Hello World` ein Parameter für die Funktion `Write-Host`.

1.1.2 Kommentare

Kommentare sind kurze Texte, die im Quellcode an (fast) beliebigen Stellen eingefügt werden können. Das Ziel von Kommentaren ist schwierigere Codeteile zu erklären. Gerade wenn im Team gearbeitet wird, ist es sehr wichtig den Code ausreichend gut zu kommentieren


Windows PowerShell
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.
PS C:\>

In diesem Tutorial werden daher viele Kommentare benutzt, um Einzelschritte zu beschreiben.

Einzeilige Kommentare

Einzeilige Kommentare gehen, wie sich leicht raten lässt, nur über eine Zeile. Dabei wird einfach nur eine Raute (#) vor den Text gestellt. Zu beachten ist jedoch, dass Kommentare ab den Bindestrichen für die **ganze** Zeile gelten.

```
# Hello world! ausgeben
Write-Host "Hello world" # auch hier kann ein Kommentar stehen
```

In diesem Codebeispiel wird deutlich, dass die Ausgabe identisch zur Ausgabe aus dem Hello-World Skript ist.

Mehrzeilige Kommentare

Mehrzeilige Kommentare können mehrere Zeilen umfassen und genutzt werden, um einen Kommentar frühzeitig zu abzuschließen.

```
<#
    Hallo Welt,
    dies ist ein mehrzeiliger Kommentar.
    ...und noch eine Zeile
#>

Write-Host "Hallo, über mir ist ein mehrzeiliger Kommentar" <#der zweite␣
↪Anwendungszweck#>
```

An dieser Stelle ist zu sehen, wie sehr *Syntax Highlighting* (farbige Markierung des Quelltextes) zur Übersicht des Codes beitragen kann. Daher empfielt es sich in jedem Fall einen guten Editor wie Powershell ISE zu benutzen.

Übung

Gib irgendeinen Text aus und verwende beide Arten von Kommentaren

Lösung

```
<#
    Dieser Code ist kein Code
#>

# Text auf der Konsole ausgeben
Write-Host "Hallo Welt"
```

1.1.3 Variablen und Datentypen

Variablen sollten für die meisten aus der Mathematik bereits bekannt sein. In der Programmierung lässt sich eine Variable als Platzhalter für einen bestimmten Inhalt sehen.

So lässt sich sagen: Eine Variable ordnet einem **Wert** einen **Namen** zu.

Literale

Literale sind feste Werte, die direkt - so wie sie sind - im Code stehen.

```
Write-Host 4
Write-Host "Hi"
```

Sowohl `4` als auch `"Hi"` sind hier Literale.

Variablennamen

Damit der Powershell Interpreter (das Programm, das den Powershell Code ausführt) Variablen auch als solche identifizieren kann, muss man sich an bestimmte Regeln halten.

Ein Variable beginnt mit einem `$` Zeichen.

Der Name darf... * ...nur aus alphanumerischen Zeichen (= Buchstaben und Zahlen) bestehen (Sonderzeichen und Umlaute sind jedoch nicht erlaubt) * ...nicht mit einer Zahl anfangen (muss also mit einem Buchstaben anfangen)

Dabei ist zu beachten das die Groß-/Kleinschreibung von Powershell bei Variablennamen vernachlässigt wird.

Des Weiteren ist es sinnvoll sich an bestimmte Namenskonventionen zu halten, damit der Code auch von anderen Entwicklern möglichst schnell verstanden werden kann. Der wichtigste Punkt ist, dass du deinen Variablen eindeutige und selbsterklärende Namen geben solltest. Weiterhin solltest du überlegen, ob sich später vielleicht Personen aus dem internationalen Raum anschließen und dann typischerweise mit einer englischen Namensgebung und Kommentaren arbeiten.

Datentypen

Variablen in Powershell sind grundsätzlich *dynamisch typisiert*. Das bedeutet, dass eine Variable jeden beliebigen Typ annehmen und diesen wechseln kann, wenn diese nicht mit einem Typ initalisiert wurden. Das macht Powershell gerade auch für Anfänger besonders attraktiv, da es dadurch besonders einfach wird.

Beim Schreiben von Powershell Code muss lediglich die Syntax der Defintion von Literalen bekannt sein.

Beispiel

```
$zahl = 1
Write-Host $zahl

$zahl = "Test"
Write-Host $zahl
```

Das Beispielprogramm gibt erst die Zahl `1` aus und anschließend den Text `Test`

```
[int]$zahl = 1
Write-Host $zahl

$zahl = "Test"
Write-Host $zahl
```

Das Beispielprogramm gibt erst die Zahl `1` aus und wird beim zuweisen des Textes scheitern, da die Variable `$zahl` typisiert ist.‘

Zahlen

Je nach Art der Zahl gibt es zwei Datentypen für Zahlen in Powershell. Für Ganzzahlen (Zahlen ohne Dezimalstellen) sollte man *int* verwenden. Für Zahlen mit Dezimalen gibt es dann die Datentypen `single`, `double` und `decimal`, welche sich bei der maximalen Größe der Zahl unterscheiden.

Dezimalzahlen (also Zahlen mit Komma) müssen statt eines Dezimalkommas jedoch einen Dezimalpunkt verwenden.

int

- Wertebereich: 2147483647 bis -2147483648
- Größe: 4 Bytes

single

- Wertebereich: 3.402823E+38 bis -3.402823E+38
- Größe: 4 Bytes
- Anzahl an Dezimalen: 7

double

- Wertebereich: 1.79769313486232E+308 bis -1.79769313486232E+308
- Größe: 8 Bytes
- Anzahl an Dezimalen: 15-16

decimal

- Wertebereich: 79228162514264337593543950335 bis -79228162514264337593543950335
- Größe: 16 Bytes
- Anzahl an Dezimalen: 28-29

Beispiele

Zahlen können ganz einfach wie folgt definiert werden:

```
[int]$meine_zahl = 42 # ganze Zahl
Write-Host $meine_zahl
[decimal]$meine_zahl_decimal = 13.37 # Dezimalzahl mit Dezimalpunkt
```

An diesem Beispiel lässt sich sehen, wie eine Variable als Platzhalter eingesetzt werden kann.

Strings / Zeichenketten

Zeichenketten, also beliebige Aneinanderreihungen von Zeichen, in der Programmierung auch **Strings** genannt, müssen zwischen Anführungszeichen gestellt werden.

```
$meine_zeichenkette = "Hallo Welt!"
```

Array

Ein Array speichert mehrere Werte, ähnlich einer 2 spaltigen Tabelle.

```
$array = ("Wert 1", "Wert 2", "Wert 3") # Erstellen eines Arrays

Write-Host $array[2] # Gibt "Wert 3" aus
```

Index	Wert
0	Wert 1
1	Wert 2
2	Wert 3

Dabei ist zu beachten das die Nummerierung der Einträge **bei 0 beginnt!**

Man kann auch einen Wert zu einem bestehenden Array mittels der += Operation hinzufügen.

```
$array = ("Wert 1", "Wert 2", "Wert 3") # Array mit 3 Werten definieren

$array += "Wert 4" # Eintrage hinzufügen
```

Wenn man jetzt schnell die Zahlen 1 bis 10 als einen Array erstellen möchte gibt es `..` Operator.

```
$array = (1..10) # Array mit Zahlen von 1 bis 10 erstellen
```

Übung

Teil 1

Aufgabe dieser Übung ist das Definieren von 4 Variablen (die Namen der Variablen können frei gewählt werden, wenn nicht anders angegeben)

1. Eine Variable soll mit der **ganzen Zahl** `-5` initialisiert werden
2. Variable mit dem **Fließkommawert** `42.1337`
3. Variable mit dem **String** `Mein Name ist Fritz`
4. Variable 4 soll Variable 2 zugewiesen werden und den Namen `letzte_variable` tragen

Im Anschluss sollen alle 4 Variablen in der angeführten Reihenfolge ausgegeben werden.

Lösung

```
# Variablen definieren
$variable1 = -5
$variable2 = 42.1337
$variable3 = "Mein Name ist Fritz" # auf Anführungszeichen achten!
$letzte_variable = $variable2

# Variablen ausgeben
Write-Host $variable1 $variable2 $variable3 $letzte_variable
```

Teil 2

Welche der folgenden Variablen sind keine gültigen Variablennamen?

- `$variable1`
- `$1variable`
- `$fritz`
- `fritz`
- `$__`
- `$--`

Lösung

- `$1variable`, weil eine Zahl am Anfang ist
- `fritz`, weil die Variable mit `$` beginnen muss
- `$--` nicht erlaubtes Zeichen

Teil 3

Erstelle einen Array mit den Zahlen 1 bis 100.

Lösung

```
$array = (1..100)
```

1.1.4 Kontrollstrukturen

Bisher haben wir nur Variablen definiert und auf der Konsole ausgegeben. Damit lässt sich natürlich noch nicht viel machen; vor allem, weil wir nicht in der Lage sind Fallunterscheidungen zu machen oder bestimmte Codeteile zu wiederholen.

if-Bedingung

Jeder, der schonmal etwas mit Programmierung zu tun hatte kennt sie: Die if-Bedingung. Mit der if-Bedingung/-Verzweigung lässt ein bestimmter Codeteil ausführen, wenn eine Bedingung wahr (oder nicht wahr) ist.

So kann mit if-Bedingungen z.B. geprüft werden, ob eine vorher definierte Variable einen bestimmten Wert hat und wenn dies der Fall ist, wird etwas auf der Konsole ausgegeben:

```
# Irgendwas definieren
$irgendwas = 5

# Jetzt mit der if-Bedingung prüfen, ob irgendwas den Wert 5 hat
if ($irgendwas -eq 5) {
    Write-Host "Die Variable irgendwas hat den Wert 5"
}

# Eine andere Bedingung, die womöglich? nicht wahr ist
if ($irgendwas -eq 42) then
    Write-Host "Die Variable hat den Wert 42"
end

# Natürlich muss man die Variable nicht mit einem Literal vergleichen,
# sondern kann sie auch mit einer anderen Variablen vergleichen
$andere_variable = $irgendwas # Wert von '$irgendwas' der Variable '$andere_variable'␣
↪zuweisen

if ($andere_variable -eq $irgendwas) {
    Write-Host "irgendwas und andere_variable haben denselben Wert!"
}
```

Wie hoffentlich durch das Beispiel klar geworden ist, steht nach dem `if` in Klammern `( )` die Bedingung und zwischen dem `{` und dem `}` der Code, der ausgeführt wird, wenn die Bedingung wahr (`true`) ist.

Bei Vergleichen muss **immer** der `-eq`-Operator verwendet werden.

Vergleichsoperatoren

Vergleichsoperator	Beschreibung
a `-eq` b	Prüft a und b auf Gleichheit, **true** falls gleich, **false** andernfalls
a `-ne` b	Prüft a und b auf Ungleichheit
a `-gt` b	Wahr, wenn a (echt) kleiner als b ist
a `-ge` b	Wahr, wenn a kleiner oder gleich b ist
a `-lt` b	Wahr, wenn a (echt) größer als b ist
a `-le` b	Wahr, wenn a größer oder gleich b ist

Logische Junktoren

Es reicht meistens nicht nur eine einzelne Sachen abzufragen. Man möchte oft mehrere einzelne Abfragen miteinander verknüpfen.

Junktor	Beschreibung
a `-and` b	Prüft, ob wohl a **als auch** b wahr sind
a `-or` b	Prüft a oder b wahr ist
`-not` a	Negiert a, d.h. `true` wird zu `false` und `false` zu `true`

Folgendes Beispiel überprüft, ob sowohl a den Wert 2 als auch b den Wert 3 hat

```
# a und b definieren
$a = 2
$b = 4
# erster Ausdruck wird wahr sein, zweiter nicht
if ($a -eq 2 -and $b -eq 3) {
    Write-Host "a ist 2 und b ist 3"
} else {
    Write-Host "Gilt nicht"
}
```

if-else

Für if gibt es noch eine kleine Erweiterung: Den `else`-Block. Dieser Block wird ausgeführt, wenn die Bedingung zwischen den Klammern **nicht** nicht wahr ist.

```
$irgendwas = 1337

if ($irgendwas == 42) {
    Write-Host "irgendwas riecht nach dem Sinn des Lebens"
} else {
    Write-Host "Nein, Leetspeak ist besser"
}
```

elseif

Mit `elseif` ist es noch möglich mehrere Vergleichen hintereinander durchzuführen. Falls der erste Vergleich nicht wahr ergibt wird die nächste evaluiert, die `elseif` wird nicht erreicht falls die erste Prüfung wahr ergibt. Es können auch mehrere `elseif`'s folgen.

```
$irgendwas = 1337

if ($irgendwas == 42) {
    Write-Host "irgendwas riecht nach dem Sinn des Lebens"
} elseif ($irgendwas == 1337) {
    Write-Host "Nein, Leetspeak ist besser"
}
```

while-Schleife

Die wohl wichtigste, aber nicht am meisten verwendet Schleife ist die **while**-Schleife. Sie führt einen Codeteil (= Block) so lange aus wie eine Bedingung wahr ist.

```
# Zähler definieren
$mein_zaehler = 1

# Schleife so lange ausführen wie der Zähler kleiner als 5 ist
while ($mein_zaehler -lt 5) {
    # Zähler ausgeben
    Write-Host $mein_zaehler

    # Zähler erhöhen (= inkrementieren)
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
    $mein_zaehler = $mein_zaehler + 1
}
```

Achte immer darauf, dass eine Bedingung auch eintritt, ansonsten verharrt das Skript in einer sog. Endlosschleife und kommt (theoretisch) nie zum Ende.

Mit einer while-Schleife lassen sich alle anderen Schleifentypen nachbauen, jedoch erlauben andere Schleifentypen in vielen Fällen eine kürzere und elegantere Lösung.

do-while-Schleife

Die do-while Schleife unterscheidet sich von der `while` dadurch das sie immer das erste Mal ausgeführt wird und zum wiederholen die Bedingung geprüft wird.

```
# Zähler definieren
$mein_zaehler = 1

# Schleife so lange ausführen wie der Zähler kleiner als 5 ist
do {
    # Zähler ausgeben
    Write-Host $mein_zaehler

    # Zähler erhöhen (= inkrementieren)
    $mein_zaehler = $mein_zaehler + 1
} while ($mein_zaehler -lt 5)
```

for-Schleife

Die for-Schleife ist mit guten Grund die weitverwendetste Schleife.

```
# Einen Zähler von 1 bis 4 laufen lassen
for ($mein_zaehler = 1; $mein_zaehler -le 4; $mein_zaehler++) {
    Write-Host $mein_zaehler
}

# alternativ kann auch die Schrittgröße beim Hochzählen angegeben werden
# (negative Schritte sind ebenfalls möglich)
Write-Host "" # leere Zeile ausgeben
for ($mein_zaehler = 1; $mein_zaehler -le 3; $mein_zaehler += 0.5) {
    Write-Host $mein_zaehler
}
```

foreach-Schleife

Mit der foreach-Schleife kann man sehr einfach mit Arrays arbeiten.

```
$zahlen = (1..10)

# Das Programm geht durch alle Zahlen, multipliziert diese und gibt es aus.
# Es wird immer ein Wert aus dem Array genommen und in die Variable $zahl geschrieben
foreach ($zahl in $zahlen) {
    Write-Host $zahl * 2
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
}

$processes = Get-Process # Lädt alle laufende Prozesse in eine Variable
foreach ($process in $processes) {
    # Gibt die Namen der Prozesse aus
    Write-Host $process.Name
}

$processes = Get-Process

$processes | ForEach-Object { Write-Host $_.Name }
```

ForEach-Object

Mit dem `ForEach-Object` funktioniert gleich wie die `foreach` und wird über Piping verwendet.

```
$processes = Get-Process
$processes | ForEach-Object { Write-Host $_.Name }
```

break - Schleife abbrechen

Alle Schleifen können wie folgt mit dem Schlüssekwort `break` abgebrochen werden.

```
for ($i = 0; $i -le 10; $i++) {
    Write-Host $i
    if ($i -eq 5) {
        break
    }
}
```

Übung

Teil 1: Verständnisfragen

Im ersten Teil der Übung sollen Verständnisfragen beantwortet werden.

1. Was ist der Unterschied zwischen einer if-Bedingung und einer Schleife?
2. Was ist der Unterschied zwischen einer for- und while Schleife?
3. Kann man mit einem Schleifentyp allen anderen Typen darstellen? Falls ja, mit welcher zum Beispiel?

Lösung

1. Eine if-Bedingung prüft nur **einmalig** die Bedingung, wohingegen eine Schleife eine Bedingung überprüft und dann einen Vorgang z.B. bis zum Eintreten der Bedingung wiederholt
2. Eine for-Schleife verwendet immer einen Zähler und zählt bis zum Erreichen eines festgelegten Wertes. Eine while-Schleife führt Code so lange aus bis eine Bedingung nicht mehr eintritt
3. Ja, z.B. mit der while-Schleife lassen sich alle anderen Schleifen darstellen (siehe Teil 2)

Teil 2: Umformen zwischen Schleifentypen

Wie in *Teil 1* schon angekündigt wurde lassen sich alle Schleifentypen ineinander mehr oder weniger problemlos überführen (wobei die for-Schleife einen Sonderfall darstellt)

In dieser Aufgabe soll nun folgende while-Schleife jeweils in eine do-while Schleife und for-Schleife überführt werden:

```
$i = 50
while ($i -ne -10) {
    $i = $i - 2
    Write-Host $i
}
```

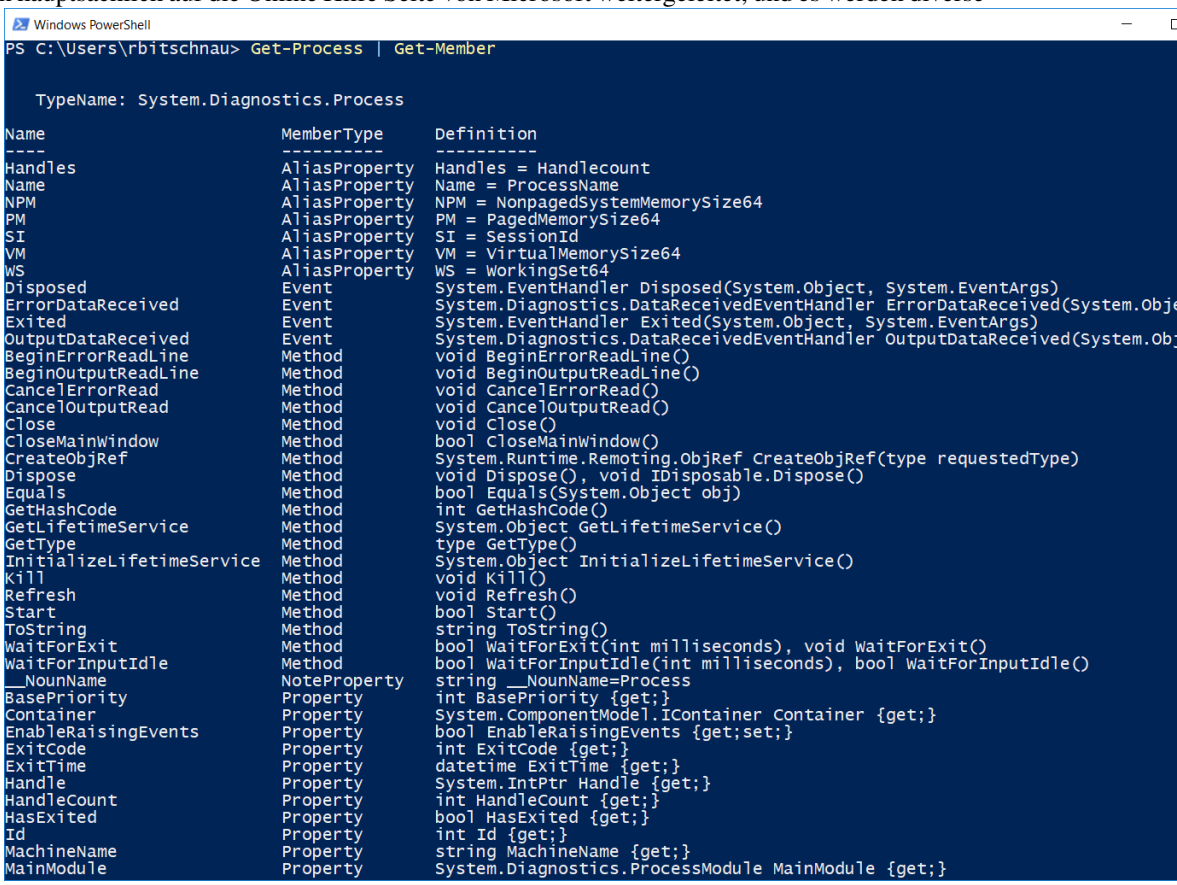
```
## do-while
$i = 50
do {
    $i = $i - 2
    Write-Host $i
} while ($i -ne -10)

# for
for ($i = 50-2; $i -ne -12; $i -= 2) {
    Write-Host $i
}
```

1.1.5 Hilfreiche Cmd-Lets

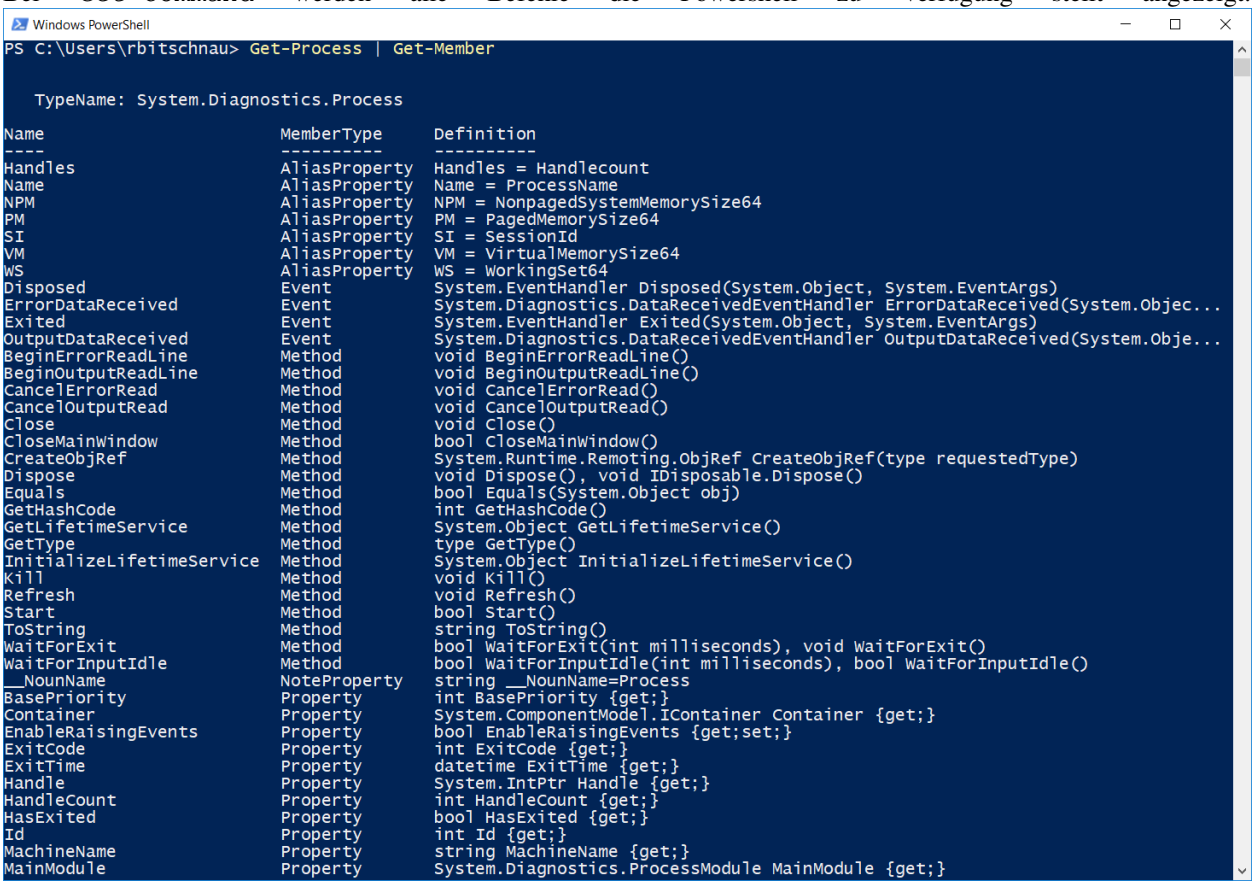
Get-Help

Bei `Get-Help` wird man hauptsächlich auf die Online Hilfe Seite von Microsoft weitergeleitet, und es werden diverse Befehlsvorlagen gezeigt.



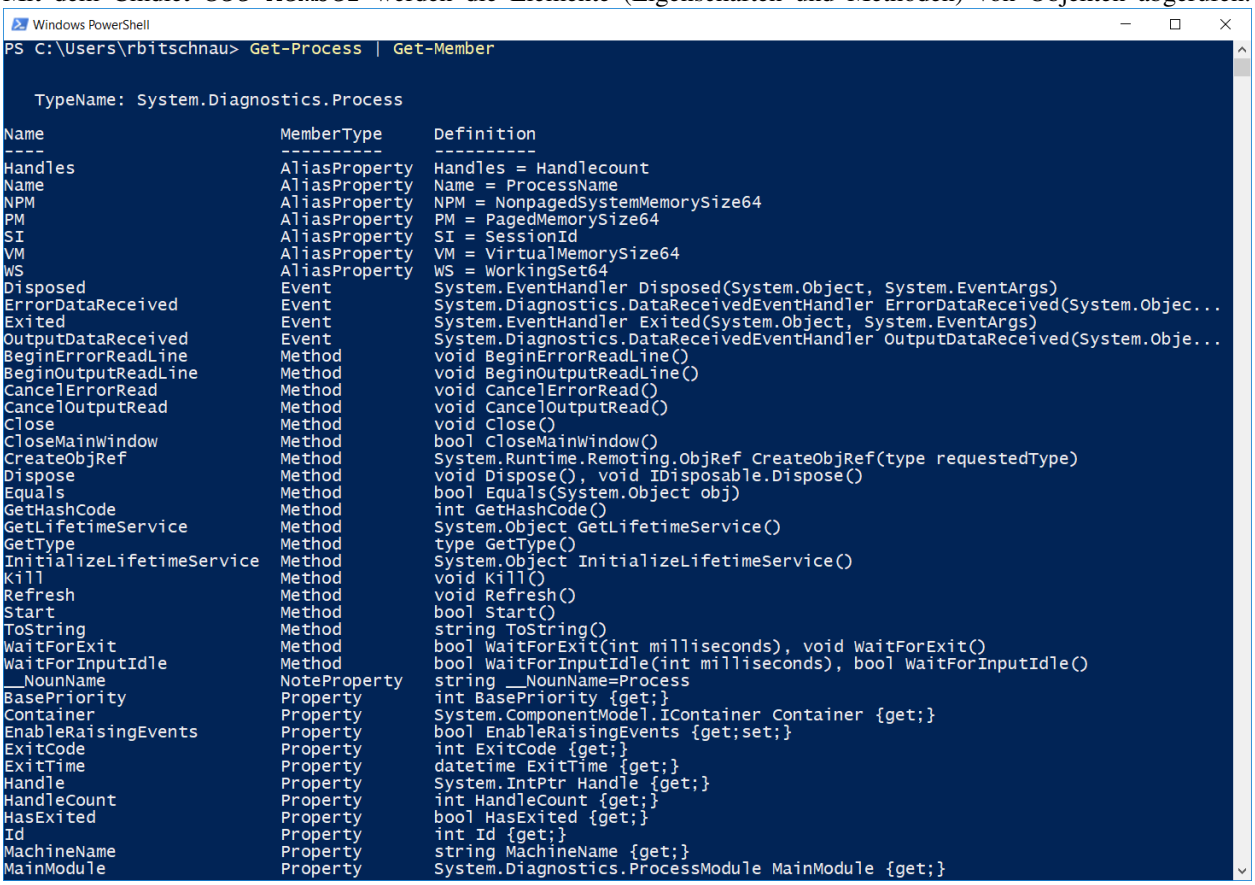
Get-Command

Bei `Get-Command` werden alle Befehle die Powershell zu Verfügung stellt angezeigt.



Get-Member

Mit dem Cmdlet `Get-Member` werden die Elemente (Eigenschaften und Methoden) von Objekten abgerufen.



Übung

Finde heraus in welchem Property in einem Objekt aus der Funktion `Get-PSDrive` die maximale Größe eines Datenträgers steht.

Lösung

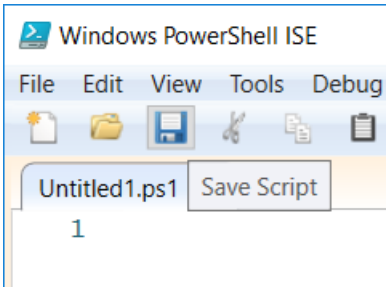
```
Get-PSDrive | Get-Member
```

In dem Property `MaximumSize` steht die maximale Größe eines Datenträgers

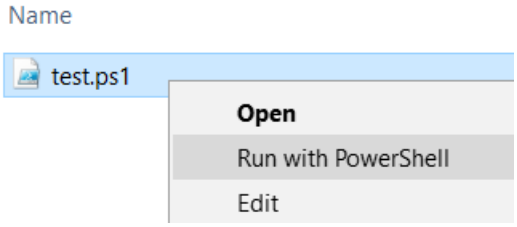
1.1.6 Speichern von Skripten

Eine .ps1 (PowerShell Script Datei) dient dazu um mehrfach benötigte Befehle und Funktionen zu speichern und einfach erneut auszuführen.

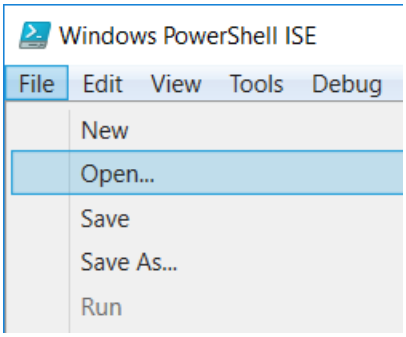
Ein PowerShell Script kann ganz einfach über die Powershell ISE geschrieben und abgespeichert werden.



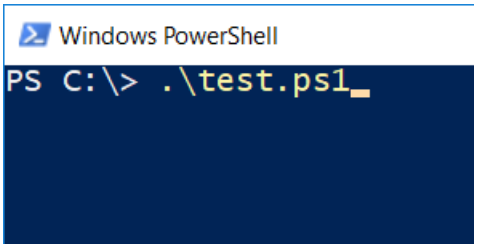
Eine Variante ein bereits gespeichertes PowerShell Script kann nun per Rechtsklick und anschließend mit “Mit PowerShell ausführen” ausführen.



Eine andere Variante wäre das PowerShell Script im Scripteditor “PowerShell ISE” über das Menü “Datei” und anschließend mit der Auswahl “Öffnen…” Öffnen und im Editor ausführen.



Noch eine Variante ein PowerShell Script auszuführen wäre einfach den Dateinamen mit dem dazugehörigen Pfad in PowerShell einzugeben und mit der Eingabetaste (Enter) auszuführen.



1.1.7 String Manipulation

Kombinieren von Strings

Natürlich lassen sich mehrere Strings auch zu einem gesamten String vereinigen. Dies wird auch Konkatenation genannt. Ob es sich dabei um einen String oder ein Stringliteral handelt, spielt keine Rolle. Die Konkatenation lässt sich mithilfe des +-Operators durchführen:

```
$text1 = "Hallo"
$text2 = "World"
$text_neu = $text1 + " " + $text2
```

.ToUpper()

Wandelt einen String in Großbuchstaben um.

```
$text = "Hallo"
$text.ToUpper() # aus Hallo wird HALLO
# oder
("Hallo").ToUpper() # aus Hallo wird HALLO
```

.ToLower()

Wandelt einen String in Kleinbuchstaben um.

```
$text = "Hallo"
$text.ToLower() # aus Hallo wird hallo
```

.Contains(Text)

Prüft ob ein String einen bestimmte Zeichenkette enthält.

```
$text = "Hallo"
$text.Contains("a") # Gibt Wahr zurück
$text.Contains("c") # Gibt Falsch zurück
```

.StartsWith(Text)

Prüft ob ein String mit bestimmte Zeichenkette beginnt.

```
$text = "Hallo"
$text.StartsWith("Ha") # Gibt Wahr zurück
$text.StartsWith("al") # Gibt Falsch zurück
```

.EndsWith(Text)

Prüft ob ein String mit bestimmte Zeichenkette endet.

```
$text = "Hallo"
$text.StartsWith("lo") # Gibt Wahr zurück
$text.StartsWith("ll") # Gibt Falsch zurück
```

.Replace(AlterText, NeuerText)

Ersetzt eine bestimmte Zeichenkette in einem String mit einer anderen String.

```
$text = "Hallo"
$text.Replace("Hallo", "World") # Hallo wird mit World ersetzt

####

$text = "Test"
$text.Replace("es", "se") # Der neue Wert ist `Tset`
```

.SubString(StartIndex, Länge)

Mit SubString kann man einen Teil eines Strings anhand der Position entfernen

```
$text = "Hallo"
$text.SubString(2) # Entfernt die ersten zwei Zeichen 'llo'
$text.SubString(0, 2) # Gibt zwei Zeichen ab dem ersten Zeichen aus 'Ha'
```

.TrimStart(ZeichenZumEntfernen)

Ersetzt eine bestimmte Zeichenkette am Anfang des Textes.

```
$text = "    Hallo"
$text.TrimStart(" ") # Entfernt alle Leerzeichen am Anfang des Stringes
```

.TrimEnd(ZeichenZumEntfernen)

Ersetzt eine bestimmte Zeichenkette am Ende des Textes.

```
$text = "Hallo------"
$text.TrimStart("-") # Entfernt alle - am Ende des Stringes
```

1.1.8 Try-Catch

Mit Try-Catch kann man Fehler in Powershell behandeln. Falls ein Befehl welcher im `try`-Block steht einen Programmabbruch verursacht springt das Programm zum `catch`-Block und dort kann der Entwickler den Fehler behandeln.

```
try {
    [int]$zahl = Read-Host "Zahl" # Powershell wirft einen Fehler falls man keine␣
↪Zahl eingibt
    Write-Host "Die Zahl" $zahl # Dieser Code wird nicht ausgeführt falls es einen␣
↪Fehler gibt
} catch {
    Write-Host "Keine Zahl" # Gibt einen Fehler aus.
}
```

Übung

Schreibe ein Programm welches eine eingegeben Zahl durch 2 dividiert und einen Fehler ausgibt falls der Benutzer keine Zahl eingibt.

```
try {
    [int]$zahl = Read-Host "Zahl"
    Write-Host $zahl / 2
} catch {
    Write-Host "Keine Zahl"
}
```

1.1.9 Piping

Mit Piping (Pipelining) können in PowerShell Befehlsketten gebaut werden. Wenn man eine Pipe (= |) in PowerShell verwendet wird das Ergebnis eines Befehls nicht direkt an den Benutzer weitergegeben sondern an den nächsten Befehl in der Befehlskette.

```
Get-Process | ForEach-Object { Write-Host $_.Name } # Gibt alle laufenden Prozesse aus
# oder
Get-Process | Get-Member # Gibt alle Methoden und Properties einenes Prozesses aus
# oder
(Get-Process | Get-Random).Name # Gibt den Namen eines zufälligen Prozesses aus
```

1.1.10 Funktionen

Oft ist man in der Situation, dass ein bestimmer Codeteil an bestimmten Stellen immer wieder benötigt wird (z.B. eine komplexere Berechnung). Diesen Code jedes Mal komplett zu kopieren würde nicht nur viele unnötige Codezeilen erzeugen, sondern auch die Wartbarkeit erheblich verringern, da im Falle eines Bugs in diesem Codeteil alle Stellen gefunden und repariert werden müssen, obwohl es sowieso überall der gleiche Code ist. Wie sich jetzt leicht vermuten lässt, lösen Funktionen dieses Problem.

Parameter

In vielen Fällen ist der Code in einer Funktion von zusätzlichen Eingaben abhängig. Damit nicht für jeden nur erdenklichen Fall dieser abhängigen Werte eine eigene Funktion geschrieben werden muss, wurden die sogenannten (Funktions-)parameter eingeführt.

Rückgabewert

Da in den meisten Fällen nicht nur eine bestimmte Prozedur ausgeführt wird, muss eine Funktion nicht nur Eingaben (über Parameter), sondern auch Ausgaben erzeugen können. Dazu gibt es die sogenannten **Rückgabewerte**.

Syntax

```
function meine_funktion([string]$parameter1, [string]$parameter2) {
    # beide Parameter (Eingaben) kombiniert und in Variable "ergebnis" speichern
    $ergebnis = $parameter1 + $parameter2

    # hier kann nahezu jeder beliebige Code stehen

    # Rückgaben werden mit dem "return" Schlüsselwort angegeben
    return $ergebnis
}
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
$rueckgabe = meine_funktion "Hello" "World"
Write-Host $rueckgabe

## oder

function meine_funktion_zahl([int]$parameter1, [int]$parameter2) {
    # beide Parameter (Eingaben) addiert und in Variable "ergebnis" speichern
    $ergebnis = $parameter1 + $parameter2

    # hier kann nahezu jeder beliebige Code stehen

    # Rückgaben werden mit dem "return" Schlüsselwort angegeben
    return $ergebnis
}

$rueckgabe = meine_funktion_zahl 2 3
Write-Host $rueckgabe
```

Funktionen werden mit dem Schlüsselwort function eingeleitet, direkt gefolgt vom Namen der Funktion. Danach folgen in runden Klammern die Namen der Parameter. Die Parameter verhalten sich innerhalb der Funktion genauso wie lokale Variablen. Schließlich wird mit dem Schlüsselwort return die lokale Variable ergebnis zurückgegeben.

Der Aufruf der Funktion sollte dir von der Funktion print bekannt vorkommen, die uns schon die ganze Zeit begleitet hat. Hier wird ganz intuitiv der Name der Funktion, gefolgt von runden Klammern, zwischen denen sich die Eingaben/Parameter der Funktion befinden, geschrieben. Dabei können natürlich sowohl Variablen als auch direkt Werte (Literale) übergeben werden.

1.2 Beispiele

1.2.1 Ein-/Ausgabe von Daten (Read-Host)

Mit dem Cmdlet `Read-Host` wird eine Eingabezeile aus der Konsole gelesen. Man kann mit ihm einen Benutzer zur Eingabe auffordern.

Erstelle ein Programm in welchem man sein Alter eingibt und diese dann wieder mittels `Write-Host` ausgegeben wird.

Lösung

```
$age = Read-Host "Please enter your age"
Write-Host "Your age is" $age
```

1.2.2 Passwort generieren

Generiere ein Passwort mit 5 Buchstaben, 2 Zahlen und einem Sonderzeichen.

Lösung

```
<#
    > (65..90) - ASCII Code für A-Z
    > (97..122) - ASCII Code für a-z
    mit ((65..90) + (97..122)) werden beide Arrays in einen neuen Array kombiniert.

    > Get-Random -Count 5
    Gibt 5 Einträge aus dem Array zurück

    > ForEach-Object {[char]$_}
    Mit [char]65 wird die Zahl in Text umgewandelt
#>
$letters = ((65..90) + (97..122) | Get-Random -Count 5 | ForEach-Object {[char]$_})
$numbers = ((48..57) | Get-Random -Count 2 | ForEach-Object {[char]$_})
$special = ((33, 35, 36, 37, 38, 64) | Get-Random -Count 1 | ForEach-Object {[char]$_}
↪)

<#
    > Sort-Object {Get-Random}
    Sortiert die Einträge der Arrays zufällig.

    > -join
    Kombiniert alle Einträge von den Arrays in eine Zeichenkette
#>
$passwort = -join ($letters + $numbers + $special | Sort-Object {Get-Random});
```

1.2.3 Zapfenrechnen

Berechne den Zapfen für eine Zahl welche der Benutzer eingeben kann.

Lösung

```
[float]$zahl = Read-Host -Prompt "Welche Zahl soll berechnet werden?"
$position = 0
$maximum = 9

do{
    $position += 1 # Zähler um 1 erhöhen
    $old = $zahl # Vorherige Zahl zwischenspeichern für die Ausgabe
    $zahl = $zahl * $position # Neue Zahl berechnen
    write-host "$old * $position = " $zahl # Ausgabe der Position
}
until ($position -eq $maximum)

# Zähler zurückstellen
$position = 0

do{
    $position += 1 # Zähler um 1 erhöhen
    $old = $zahl # Vorherige Zahl zwischenspeichern für die Ausgabe
    $zahl = $zahl / $position # Neue Zahl berechnen
    write-host "$old / $position = " $zahl # Ausgabe der Position
}
until ($position -eq $maximum)
```

1.2.4 Userverwaltung

In diesem Tutorial erstellen wir eine Userverwaltung.

Erstelle eine neue Datei mit dem folgenden Inhalt und speichere diese als **.csv** ab.

```
Abteilung;Nachname;Vorname
Einkauf;Schreiber;Florian
Einkauf;Bayer;Maik
Einkauf;Schweitzer;Michelle
Einkauf;Schweitzer;Marco
Verkauf;Seiler;Barbara
Verkauf;Amsel;Anne
Verkauf;Moeller;Stefanie
Verkauf;Moeller;Sven
Verkauf;Moeller;Stefan
```

Aufgabe 1

Lade die CSV mit dem Befehl `Import-Csv` und gib die Länge der Liste aus. Zusätzlich gebe jeden Nachname aus (keine duplikate).

Lösung

```
# -Delimiter ";" - um den trenner zwischen den Daten in einer Zeile festzulegen
# -Encoding Default - damit die Datei als Windows 1251 geöffnet wird␣
↪(Zeichenkodierung)
$users = Import-Csv -Delimiter ";" -Encoding Default "users.csv"

# $users.length gibt die Länge des Arrays zurück
Write-Host "Es gibt " $users.length " User"

# Mit $users.Nachname wird ein neuer Array erzeugt in welchem nur die Nachname sind
# Dabei exisiteren aber noch die Duplikate
$lastnames = $users.Nachname

# Mit diesem Befehl kann ich nun die eindeutigen Nachnamen ermitteln
$lastnames = $lastnames | Get-Unique

Write-Host $lastnames
```

Aufgabe 2

Generiere die Benutzernamen für alle Benutzer.

Dabei gibt es die folgenden Reglen: - keine Duplikate - erstes Zeichen von Vorname + Nachname - falls dies nicht möglich ist zwei Zeichen usw.. - falls dies nicht geht soll eine Zahl zwischen dem ganzen Vor- und Nachname eingefügt werden.

Lösung

```
# -Delimiter ";" - um den trenner zwischen den Daten in einer Zeile festzulegen
# -Encoding Default - damit die Datei als Windows 1251 geöffnet wird␣
↪(Zeichenkodierung)
$users = Import-Csv -Delimiter ";" -Encoding Default "users.csv"

# Fügt das Feld Username zu den Objekten im Array hinzu
$users | Add-Member Username ""


foreach ($user in $users) {
    $firstname_count = 0 # wird mit 0 initialisiert da es bei der Schleife direkt␣
↪erhöht wird
    $zahl = 0

    # Im ersten Schritt werden alle Zeichen in Kleinbuchstaben umgewandelt und dann␣
↪alle Zeichen welche im Benutzernamen nicht zulässig sind entfernt
    # hierfür wird .Replace() verwendet
    $firstname = $user.Vorname.ToLower().Replace("ä", "ae").Replace("ö", "oe").
↪Replace("ü", "ue").Replace("ß", "ss")
    $lastname = $user.Nachname.ToLower().Replace("ä", "ae").Replace("ö", "oe").
↪Replace("ü", "ue").Replace("ß", "ss")


    do {
        # Erhöht die Anzahl der Zeichen vom Vornamen
        $firstname_count++

        # Prüft ob die Anzahl der Zeichen vom Vornamen größer ist als die Länge des␣
↪Vornamens
        # falls ja dann wird probiert eine Zahl zwischen Vorname und Nachname␣
↪einzufügen
        if ($firstname_count -gt $firstname.length) {
            $zahl++
            $username = $firstname + $zahl + $lastname
        } else {
            # $firstname.Substring(0, X) gibt die Zeichen bis X
            # Wenn X gleich 1 ist gibt es das erste Zeichen zurück
            # Wenn X gleich 2 ist gibt es die ersten zwei Zeichen zurück
            $username = $firstname.Substring(0, $firstname_count) + $lastname
        }
        # $users.Username erstellt wieder einen Array mit allen Benutzernamen
        # mit -contains wird geprüft ob der Name in diesem Array existiert
        # dadurch wird die Schleife solange wiederholt bis ein Benutzername frei ist
    } while ($users.Username -contains $username)

    # Setzt den Benutzernamen beim Objekt
    $user.Username = $username
}

# Gibt alle User aus
Write-Host $users
```

Aufgabe 3

Generiere nun zusätzlich für jeden Benutzer ein Passwort und speichere es anschließend in eine CSV zurück.

Lösung

```
# -Delimiter ";" - um den trenner zwischen den Daten in einer Zeile festzulegen
# -Encoding Default - damit die Datei als Windows 1251 geöffnet wird␣
↪(Zeichenkodierung)
$users = Import-Csv -Delimiter ";" -Encoding Default "users.csv"

# Fügt das Feld Username zu den Objekten im Array hinzu
$users | Add-Member Username ""
$users | Add-Member Password ""


foreach ($user in $users) {
    $firstname_count = 0 # wird mit 0 initialisiert da es bei der Schleife direkt␣
↪erhöht wird
    $zahl = 0

    # Im ersten Schritt werden alle Zeichen in Kleinbuchstaben umgewandelt und dann␣
↪alle Zeichen welche im Benutzernamen nicht zulässig sind entfernt
    # hierfür wird .Replace() verwendet
    $firstname = $user.Vorname.ToLower().Replace("ä", "ae").Replace("ö", "oe").
↪Replace("ü", "ue").Replace("ß", "ss")
    $lastname = $user.Nachname.ToLower().Replace("ä", "ae").Replace("ö", "oe").
↪Replace("ü", "ue").Replace("ß", "ss")


    do {
        # Erhöht die Anzahl der Zeichen vom Vornamen
        $firstname_count++

        # Prüft ob die Anzahl der Zeichen vom Vornamen größer ist als die Länge des␣
↪Vornamens
        # falls ja dann wird probiert eine Zahl zwischen Vorname und Nachname␣
↪einzufügen
        if ($firstname_count -gt $firstname.length) {
            $zahl++
            $username = $firstname + $zahl + $lastname
        } else {
            # $firstname.Substring(0, X) gibt die Zeichen bis X
            # Wenn X gleich 1 ist gibt es das erste Zeichen zurück
            # Wenn X gleich 2 ist gibt es die ersten zwei Zeichen zurück
            $username = $firstname.Substring(0, $firstname_count) + $lastname
        }
        # $users.Username erstellt wieder einen Array mit allen Benutzernamen
        # mit -contains wird geprüft ob der Name in diesem Array existiert
        # dadurch wird die Schleife solange wiederholt bis ein Benutzername frei ist
    } while ($users.Username -contains $username)

    # Setzt den Benutzernamen beim Objekt
    $user.Username = $username

    # Dieser Teil ist vom Beispiel "Passwort generieren"
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
    <#
        > (65..90) - ASCII Code für A-Z
        > (97..122) - ASCII Code für a-z
        mit ((65..90) + (97..122)) werden beide Arrays in einen neuen Array␣
↪kombiniert.

        > Get-Random -Count 5
        Gibt 5 Einträge aus dem Array zurück

        > ForEach-Object {[char]$_}
        Mit [char]65 wird die Zahl in Text umgewandelt
    #>
    $letters = ((65..90) + (97..122) | Get-Random -Count 5 | ForEach-Object {[char]$_}
↪)
    $numbers = ((48..57) | Get-Random -Count 2 | ForEach-Object {[char]$_})
    $special = ((33, 35, 36, 37, 38, 64) | Get-Random -Count 1 | ForEach-Object
↪{[char]$_})

    <#
        > Sort-Object {Get-Random}
        Sortiert die Einträge der Arrays zufällig.

        > -join
        Kombiniert alle Einträge von den Arrays in eine Zeichenkette
    #>
    $user.password = -join ($letters + $numbers + $special | Sort-Object {Get-Random}
↪);
}

# -Delimiter ";" - um den trenner zwischen den Daten in einer Zeile festzulegen
# -Encoding Default - damit die Datei als Windows 1251 geöffnet wird␣
↪(Zeichenkodierung)
$users | Export-Csv -Path "users_with_password.csv" -NoTypeInformation -Delimiter ";"␣
↪-Encoding Default
```

Aufgabe 4

User im ActiveDirectory erstellen mit OUs für jede Abteilung. Für dieses Beispiel wird die CSV welche beim Aufgabe 3 erstellt wurde weiterverwendet.

```
"Abteilung";"Nachname";"Vorname";"Username";"Password"
"Einkauf";"Schreiber";"Florian";"fschreiber";"hB75&cfL"
"Einkauf";"Bayer";"Maik";"mbayer";"&w6ZUM5L"
"Einkauf";"Schweitzer";"Michelle";"mschweitzer";"&M7ep9PR"
"Einkauf";"Schweitzer";"Marco";"maschweitzer";"0M3%qpNb"
"Verkauf";"Seiler";"Barbara";"bseiler";"0BXeQ7!k"
"Verkauf";"Amsel";"Anne";"aamsel";"3WXhQ!M2"
"Verkauf";"Moeller";"Stefanie";"smoeller";"qF7P6!ED"
"Verkauf";"Moeller";"Sven";"svmoeller";"W13Cv!Pq"
"Verkauf";"Moeller";"Stefan";"stmoeller";"aN%36vwh"
```

Lösung

```
# -Delimiter ";" - um den trenner zwischen den Daten in einer Zeile festzulegen
# -Encoding Default - damit die Datei als Windows 1251 geöffnet wird␣
↪(Zeichenkodierung)
$allusers = Import-Csv -Delimiter ";" -Encoding Default "users_with_password.csv"

# Ermittelt alle eindeutigen Abteilungen
$abteilungen = $allusers.Abteilung | Get-Unique

foreach ($abteilung in $abteilungen) {
    # Domäne ist TEST.AT und die User sollen in TEST\Users abgelegt werden
    # der Pfad zur richtigen OU wird rückwärts angegeben
    $path = "OU=Users,OU=TEST,DC=test,DC=at"

    $filter = 'Name -eq "' + $abteilung + '"'

    # Versucht das OU Objekt für die Abteilung zu bekommen
    # falls dies fehlschlägt ist in $org kein Wert
    $org = Get-ADOrganizationalUnit -Filter $filter -SearchBase $path

    # Wenn die OU noch nicht vorhanen ist wird diese mit dem Befehl erstellt
    if (-Not $org) {
        New-ADOrganizationalUnit -Name $abteilung -Path $path -
↪ProtectedFromAccidentalDeletion $False
    }

    # Fügt die neue OU zum Pfad hinzu
    $path = "OU=" + $abteilung + "," + $path

    # Ermittelt alle User mit der gegeben Abteilung
    $users = $allusers | Where-Object {$_.Abteilung -eq $abteilung}

    foreach ($user in $users) {
        # Mit ConverTo-SecureString wird das Passwort gehashed, da bei New-AdUser␣
↪kein klartext Passwort zulässig ist.
        $password = $user.Password | ConvertTo-SecureString -AsPlainText -Force

        # Erstellt den AdUser
        # -Name - Username
        # -AccountPassword - das gehashte Passwort
        # -ChangePasswordAtLogon - das der User beim Login das Passwort ändern muss
        # -GivenName - Vorname
        # -Surname - Nachname
        # -Path - Pfad im AD
        # -Enable - Ob der Account aktiviert werden soll
        New-AdUser -Name $user.Username -AccountPassword $password -
↪ChangePasswordAtLogon $True -GivenName $user.Vorname -Surname $user.Nachname -Path
↪$path -Enabled $True
    }
}
```