Einführung in die Betriebssysteme

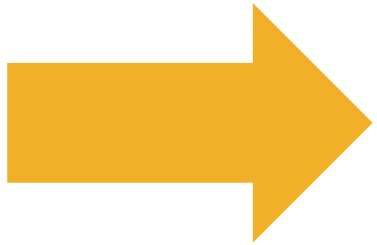
Martin Spörl

Shellprogrammierung II – Microsoft Powershell

Microsoft Powershell

Geschichte

- alle MS-DOS & Windows Versionen COMMAND.COM bzw. cmd.exe als Kommandozeile
- BAT als Scriptsprache
  - Bot einige Standardbefehle
  - Bot Zugriff auf andere Programme



Probleme

- nicht Flexibel
- kaum Erweiterbar
- nicht alle Funktionen der GUI verfügbar

Entwicklung von Windows Script Host (cscript.exe)

Microsoft Powershell

Windows Script Host

- kann mittels Script Engines Skripte starten
  - z.B. VBScript
- bot mehr Flexibilität als BAT

VBScript Beispiel

```
dim myVar

myVar = 1
if myVar = 0 then
    MsgBox "Hey – it is 0!"
else
    MsgBox "Oh it is actually 1…"
end if
```

Probleme

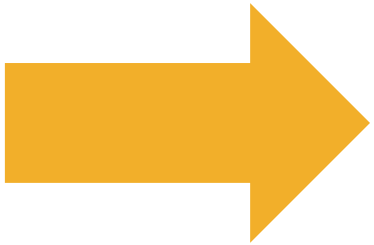
- Neben WSH noch mehr Interpreter (oft mit dediziertem Einsatz gebiet
  - z.B. WMI (Windows Management Instrumentation)
- sehr anfällig für Ausnutzung durch Viren und anderer Schadsoftware

Jeffrey Snover verfasst “Monad Manifest” (1999)

Microsoft Powershell

Monad Manifest

- Inhalt
  - Viele Befehle erfüllen bereits Standardfunktionen = Schnelles Skripten
  - Arbeiten mit Objekten
  - Remote Execution
  - Option für GUIs
- 2005 erschien Beta für „Monad Shell“



Powershell

- 2006 wurde aus „Monad Shell“ das heutige „Powershell“
- Exchange Server 2007 wurde als erstes vollständig über Powershell gesteuert
- 2008 wird Server 2008 als erstes OS mit Powershell veröffentlicht
- seit 2016 wird Powershell Open Source & Plattformunabhängig

Begriffe

Powershell

- übergeordneter Begriff
  - Framework (plattformunabhängig)
  - Skriptsprache

Powershell Core

- Plattformübergreifende Form von Powershell
- läuft auf
  - Windows
  - Linux
  - MacOS
- basiert auf .Net Core (plattformunabhängige Variante von .Net Framework) mit CoreCLR (.Net Core Execution Engine)

Windows Powershell

- Teil der Windows Management Framework (WMF)
- wird mit Windows ausgeliefert
- basiert auf .Net Framework mit CLR (.Net Execution Engine)

Komponenten

Powershell Engine

- Kommandozeileninterpreter
- Sammlung von .Net Klassen in *System.Management.Automation.dll*

Powershell Host

- Benutzerschnittstelle
- unter Windows *powershell.exe* oder *powershell_ise.exe* (Shell mit integriertem Skript-Editor)

PowerBash Scripting Language

- Sprache in der Powershell Skripte geschrieben werden

Cmdlets

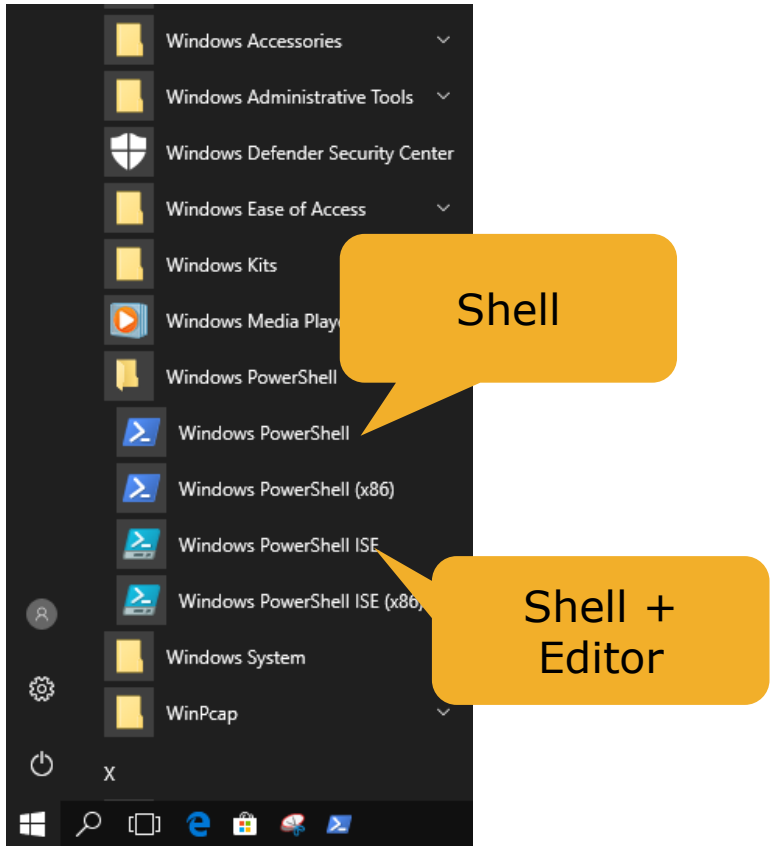
- Befehle der Powershell
- nicht lauffähig – nur über Powershell startbar (unter MS-DOS/Linux war/ist jeder Befehl ein eigenes Programm)

Powershell Provider

- bietet Zugriff auf schwer zugängliche Daten
  - Registry
  - Zertifikate
  - Active Directory Inhalt
- stellt sie als Laufwerk dar

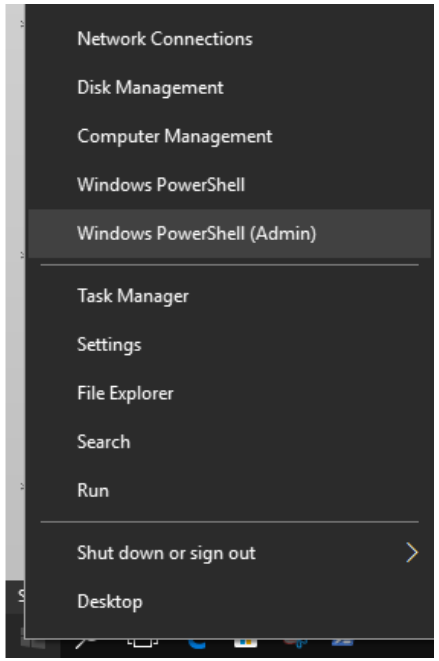
Powershell öffnen

Startmenü



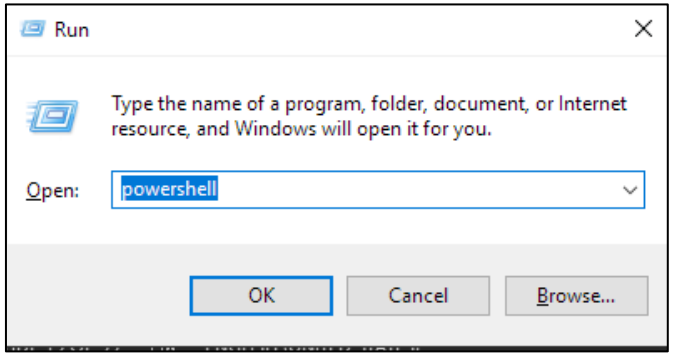
Start Kontextmenü

- Rechtsklick auf Windows-Logo



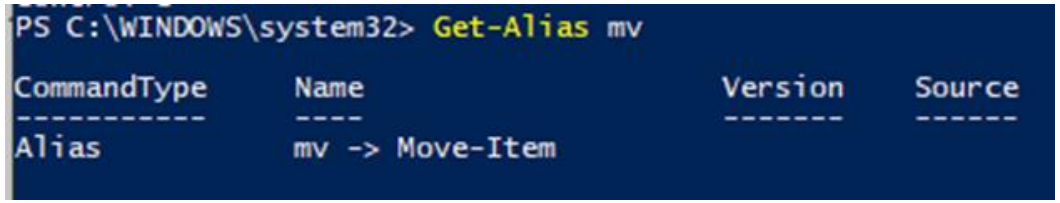
Tastenkombination

- Windows + R
- powershell



Allgemeine Hinweise

- Dateien enden auf „.ps1“
- fast alle DOS Befehle durch Alias vorhanden
  - Get-Alias <Befehl> zeigt Powershell-Pendant
  - vereinzelt haben Alias andere Logik
- nutzt als Sicherheitsmechanismus „Execution Policies“
  - definieren wann was ausgeführt / geladen werden darf
  - sichern das System gegen unbefugte Ausführung ab
  - Setzen und lesen mit „Set-ExecutionPolicy“ und „Get-ExecutionPolicy“



Policy	Erklärung
Restricted	Einzelne Befehle erlaubt, aber keine Skripte
AllSigned	Alle signierten und lokal erstellten Skripts dürfen laufen
RemoteSigned	Signierte aus dem Internet geladene Skripte dürfen laufen
Unrestricted	Alle Skripte dürfen laufen; Rückfrage bei Unsignierten Skripts
Bypass	Keine Einschränkung
Undefined	Entfernt alle zugewiesenen Policies

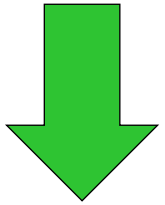
Grundaufbau

```
#einzeiliger Kommentar

<# mehr-
Zeiliger
Kommentar #>
```

Einzeiliger Kommentar

Oft für Metadaten (Autor, Datum, Version, Readme, …) genutzt



Speichern als basic_script.ps1 Datei

Befehlsseparator

- Powershell kennt 2 Separator für Befehle

- New Line (Zeilenumbruch)
  - Am Ende einer Zeile
  - Geeignet für „Single-Line-Commands“

- Semikolon („;“)
  - Trennt befehle voneinander
  - Kann mehrere Befehle pro Zeile ermöglichen

```
PS C:\Users\Marti> echo "Test"
Test
PS C:\Users\Marti> echo "Test"; echo "Test2"
Test
Test2
PS C:\Users\Marti> echo "Test" echo "Test2"
Test
echo
Test2
```

```
do ( var ++; var2++) while()

ist gleich mit

do {
  var++
  var2++
}while()
```

Variablen

Hinweise

- Keine Basistypisierung! (Objekte haben Typ)
- Vordefinierte Variablen (Auszug)

Befehl	Erklärung
HOST	Infos über Ausführungsumgebung
HOME	Gibt den Pfad des Benutzerordners zurück
true	Boolean „Wahr“
false	Boolean „Falsch“
null	NULL-Objekt („nichts“)
$_	Das aktuelle Objekte einer Pipeline, eines Filter und den verschiedenen Schleifen.
$env:<variable>	Abfragen von Umgebungsvariablen


```
$myvar=1234
echo $myvar
```

```
PS C:\INF - Betriebssysteme\Code\Powershell\Variablen> .\variablen.ps1
1234
```

```
echo "Hello $env:USERNAME"
echo "Your Windows is Running on $env:SystemRoot"
echo "Your home folder is at $HOME"
```

```
PS C:\INF - Betriebssysteme\Code\Powershell\Variablen> .\predefined_variables.ps1
Hello Martin
Your Windows is Running on C:\WINDOWS
Your home folder is at C:\Users\Martin
```

Arrays

Definieren

```
$myArray = @("Hello", "World!")
echo $myArray[0]
echo $myArray[1]
```

Arrays haben feste Größe!

Auslesen

```
$myArray = @("Hello", "World!",
"Whats", "Up?")
echo $myArray[0,1]
echo "-----"
echo $myArray[-1]
echo "-----"
echo $myArray[0 .. 2]
```

Anhängen

```
$myArray = @("Hello", "World!")
$myArray = $myArray + "New Item"
echo $myArray[0]
echo $myArray[1]
echo $myArray[2]
```

Erstellt im Hintergrund ein neues Array

Entfernen

```
$myArray = @("Hello", "World!", "Whats", "Up?")
$newArray = $myArray | Where-Object { $_ -ne
"Whats" }
echo $newArray
```

Benutzerinteraktion

Benutzereingabe lesen

```
$myvar = Read-Host -Prompt "Please enter your name"
echo "Hello $myvar"
```

„-Prompt“ ist Text der dem User angezeigt wird. Ein „:“ wir automatisch angehangen

Wird „-Prompt“ weggelassen, nimmt PS automatisch den anschließenden String. D.h. Read-Host „Enter“ ist gleich mit „Read-Host –Prompt „Enter“

Sensitive Benutzereingabe lesen

```
$myvar = Read-Host -Prompt "Please enter your pin" -AsSecureString
echo "Your pin is $myvar“ #does output just datatype
```

„-AsSecureString“ macht Eingabe unsichtbar

Textausgabe

echo

- Mapped intern auf Write-Output

Write-Output

- Schreibt in die aktuelle Pipeline

Write-Host

- Schreibt auf den aktuellen Host (z.B. Bildschirm)
- Bietet Formatierungsfunktion

Write-Error

- Schreibt auf den Fehler-Stream (STDERR)

Write-Verbose

- Schreibt in (per default) nicht sichtbaren Message stream
- -Verbose macht es sichtbar

Mit $(*Ausdruck*) kann ein Ausdruck (Variable, Funktion, etc.) ausgeben werden

Soll wert ohne Formatierung ausgeben werden, kann das direkt ohne cmdlet geschehen!

String-Operationen

Substring

```
$mystr = "Hello World!"
$hello = $mystr.Substring(0,5)
echo $hello
$world =
$mystr.Substring(6,$mystr.Length-6)
echo $world
$world2 = $mystr.Substring(6,5)
echo $world2
```

Startindex (0-Based)

Länge

Split

```
$mystr = "Hello;World!"
$tmp = $mystr.Split(";");
echo $tmp[0]
echo $tmp[1]
```

Erzeugt ein Array

Replace

```
$mystr = "Hello World!"
$tmp =
$mystr.Replace("World","Universe");
echo $tmp
```

Search

Replace

Concat

```
$mystr1 = "Hello"
$mystr2 = "World!"
$constr = $mystr1 + " " + $mystr2
echo $constr
```

Schleifen & Verzweigung

While

```
$myvar = 1
while($myvar -lt 10)
{
    $myvar+= 2
    echo $myvar
}
echo "Done"
```

If

```
$myvar = 1
if($myvar -eq 30)
{
    echo "Yes!"
}
else
{
    echo "No!"
}
```

For

```
for($i = 0; $i -le 10; $i++)
{
    echo $i
}
```

Do-While

```
$myvar = 1
do {
    $myvar += 2
    echo $myvar
}while($myvar -lt 10)
```

Läuft **solange** Bedingung wahr ist

Do-Until

```
$myvar = 1
do {
    $myvar += 2
    echo $myvar
}until($myvar -gt 10)
```

Läuft **bis** Bedingung wahr ist

Foreach

```
$myvar = @(1,2,3,4,5)
foreach($item in $myvar) {
    echo $item
}
```

Vergleichsoperatoren

Operator	Bedeutung	Beispiel
-eq	gleich („**eq**uals“)	$a -eq 1
-ne	ungleich („**n**ot **e**quals“)	$b -ne “World”
-lt	kleiner („**l**ower **t**hen“)	$a -lt 5
-le	kleiner oder gleich („**l**ess or **e**equals“)	$b -le 4
-gt	größer („**g**reater **t**hen“)	$a -gt $b
-ge	größer oder gleich („**g**reater or **e**quals“)	$b -ge 7

Mehrfachverzweigung

Switch als zentrales Element zur Fallunterscheidung!

Switch Case

```
$myvar = 1
switch($myvar){
                1 { echo "Num 1" }
                2 { echo "Num 2" }
                3 { echo "Num 3" }
                default { echo "AAAH" }
}
```

Variablenzuweisung

```
$myvar = 1
$result = switch($myvar){
                1 { "Num 1" }
                2 { "Num 2" }
                3 { "Num 3" }
                default { "AAAH" }
}
echo $result
```

Array

```
$myvar = @(1,3)s
switch($myvar){
                1 { echo "Num 1" }
                2 { echo "Num 2" }
                3 { echo "Num 3" }
                default { echo "AAAH" }
}
```

Arithmetische Operation I

Addition
```
$myvar1 = 1
$myvar2 = 2
$sum = $myvar1 + $myvar2
echo $sum
```

Multiplikation
```
$myvar1 = 1
$myvar2 = 2
$mul = $myvar1 * $myvar2
echo $mul
```

Modulo
```
$myvar1 = 1
$myvar2 = 2
$mul = $myvar1 % $myvar2
echo $mul
```

Subtraktion
```
$myvar1 = 1
$myvar2 = 2
$diff = $myvar1 - $myvar2
echo $diff
```

Division (! nicht ganzzahlig !)
```
$myvar1 = 1
$myvar2 = 2
$quot = $myvar1 / $myvar2
echo $quot
```

Shift
```
$shiftleft = 1 -shl 2
echo $shiftleft
$shiftright = 4 -shr 1
echo $shiftright
```

Arithmetische Operation II

Bitwise OR

```
$myvar1 = 1
$myvar2 = 2
$sum = $myvar1 -bor $myvar2
echo $sum
```

Bitwise AND

```
$myvar1 = 1
$myvar2 = 2
$sum = $myvar1 -band $myvar2
echo $sum
```

Bitwise XOR

```
$myvar1 = 1
$myvar2 = 2
$sum = $myvar1 -bxor $myvar2
echo $sum
```

unterschiedliche Zahlensysteme

```
#Hex 1
$myvar1 = 0x1
$sum = $myvar1 + $myvar1
echo $sum
```

Dateioperationen

Datei einlesen

```
$tmp = Get-Content FileRead.ps1
echo $tmp
```

Datei (über-)schreiben

```
$tmp = "This is not readable"
$tmp | Out-File Test.txt
$tmp2 = "This is readable"
$tmp2 | Out-File Test.txt
Get-Content Test.txt
```

Datei anhängen

```
$tmp = "This is readable"
$tmp | Out-File Test.txt
$tmp2 = "This is also readable"
$tmp2 | Out-File -append Test.txt
Get-Content Test.txt
```

-append weist Anfügen an

Funktionen

```
function MyFunction($p1, $p2)
{
   $param1= $p1
   $param2= $p2
   return $param1 * $param1 + $param2
}
```

Es gibt weitere Schreibformen

```
$erg=0
$erg = MyFunction -p1 2 -p2 3
echo $erg
```

„Benannte“ Parameter

Variablen typisieren

Hinweise

- Typisierung i.d.R nicht nötig, aber möglich
- Kann vor Falschzuweisung schützen
- Datentyp muss lediglich in [ ] vor Variablenname erscheinen

Richtig

```
[int32] $myvar=1234
echo $myvar
```

Fehler

```
[int32] $myvar=“Hello World!”
echo $myvar
```

```
Cannot convert value "Hello World!" to type "System.Int32". Error: "Input string was not in a correct format."
At C:\INF -Betriebssysteme\Code\Powershell\Variablen\typisierung_falsch.ps1:1 char:1
+ [int32] $myvar="Hello World!"
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : MetadataError: (:) [], ArgumentTransformationMetadataException
    + FullyQualifiedErrorId : RuntimeException
```

Parameter typisieren

Hinweise

- Typisierung i.d.R nicht nötig, aber möglich
- Kann vor Falschzuweisung schützen
- Datentyp muss leidglich in [ ] vor Variablenname erscheinen

```
function MyFunction([int32]$p1, [int32]$p2)
{
   $param1= $p1
   $param2= $p2
   return $param1 * $param1 + $param2
}


$erg=0
$erg = MyFunction -p1 2 -p2 3
echo $erg
```

```
MyFunction : Cannot process argument transformation on parameter 'p1'. Cannot convert value "Hello" to type "System.Int32". Error: "Input string was not in a correct
format."
At C:\INF - Betriebssysteme\Code\Powershell\Funktionen\funktion_typisiert.ps1:10 char:23
+ $erg = MyFunction -p1 "Hello" -p2 3
+                       ~~~~~~~
    + CategoryInfo          : InvalidData: (:) [MyFunction], ParameterBindingArgumentTransformationException
    + FullyQualifiedErrorId : ParameterArgumentTransformationError,MyFunction
```

Übliche Datentypen

Datentype	Erklärung	Beispiel
[Array]	Array	[Array]$myArray = @("A","B","C")
[Bool]	Boolscher Wert (TRUE oder FALSE)	[Bool]$myBool = $true
[DateTime]	Datum und Uhrzeit	[DateTime]$myDate = Get-Date
[Int32] (oder nur [Int])	32Bit Integer	[Int32]$myInt32 = 4 / [Int]$myInt = 3
[PSObject]	Powershell Object	[PSObject]$myObject = New-Object PSObject
[Float]	Fließkommazahl	[Float]$myFloat = 3.2
[String]	Zeichenkette	[String]$myString = "Hello World!";

Übergabeparameter

```
#run this script with parameter.ps1 <Your Name> <Your Age>
echo "Hello, my name is $($args[0])"
echo "I am $($args[1]) years old"
```

Argument sind Array von Werten

Achtung – Leerzeichen trennt Argument!

Verwechslungsgefahr:
Bei Powershell starten Parameter mit 0 ($args[0])
Bei Shell/Bash mit 1 ($0 ist das Script selbst)!

Benutzerdefinierte Objekte I

Vor Powershell V2

```
[PSObject]$myObject = New-Object
PSObject
$myObject | Add-Member
NoteProperty opt1 "Hello"
$myObject | Add-Member
NoteProperty opt2 "World"

echo $myObject.opt1
echo $myObject.opt2
```

Ab Powershell V2

```
[PSObject]$myObject = New-Object
PSObject -Property @{
   opt1 = "Hello"
   opt2 = "World"
}
echo $myObject.opt1
echo $myObject.opt2
```

Schreibweise einer Hashmap!

Benutzerdefinierte Objekte II

[Ab V2] Objekt Member per Variable

```
[hashtable]$hash = @{
   opt1 = "Hello"
   opt2 = "World"
}

[PSObject]$myObject = New-Object
PSObject -Property $hash
echo $myObject.opt1
echo $myObject.opt2
echo $myObject.opt3
```

Ab Powershell V2

```
[hashtable]$hash = @{
   opt1 = "Hello"
   opt2 = "World"
}

[PSObject]$myObject = New-Object
PSObject -Property $hash

Add-Member -InputObject $myObject
-MemberType ScriptMethod -Name
opt3 -Value { echo ($this.opt1 + " " +
$this.opt2) }

echo $myObject.opt1
echo $myObject.opt2
echo $myObject.opt3();
```

Methode kann auch in Variable stehen (um sie mehreren Objekten zuzufügen)

Objekte nach CSV exportieren

Export nach CSV

```
$userdata= @("Hans","Meyer",33),
                    ("Paul","Müller",30),
                    ("Johanna","Schmidt",20)
$header = @("Vorname","Nachname","Alter")

$users = @()

foreach($user in $userdata){
                    $obj = new-object PSObject
                    for($i = 0; $i -lt 3; $i++){
                                        $obj | add-member
                                        -membertype NoteProperty
                                        -name $header[$i]
                                        -value $user[$i]
                    }
                    $users += $obj
                    $obj = $null
}

$users | Export-CSV -NoTypeInformation users.csv


Get-Content users.csv
```

Bauen von Benutzerdefiniertem Objekt

„-NoTypeInformation“ unterdrückt Datentyp

Import aus CSV

```
$users = Import-CSV users.csv

foreach($user in $users){
                    echo "Hallo $($user.Vorname)
                    $($user.Nachname) - you are
                    $($user.Alter) years old"

}
```

Named vs. Positional Parameter

Positional Parameter

```
function MyFunction
{
   Param
   (
      [Parameter(Position=0)]
      [Int]$param1,

      [Parameter(Position=1)]
      [Int]$param2
   )
   return $param1 * $param1 +
$param2
}


$erg=0
$erg = MyFunction 3 2
echo $erg
```

Named Parameter

```
function MyFunction
{
   Param
   (
      [Int]$param1,
      [Int]$param2
   )
   return $param1 * $param1 +
$param2
}


$erg=0
$erg = MyFunction -param1 1 -
param2 2
echo $erg
```

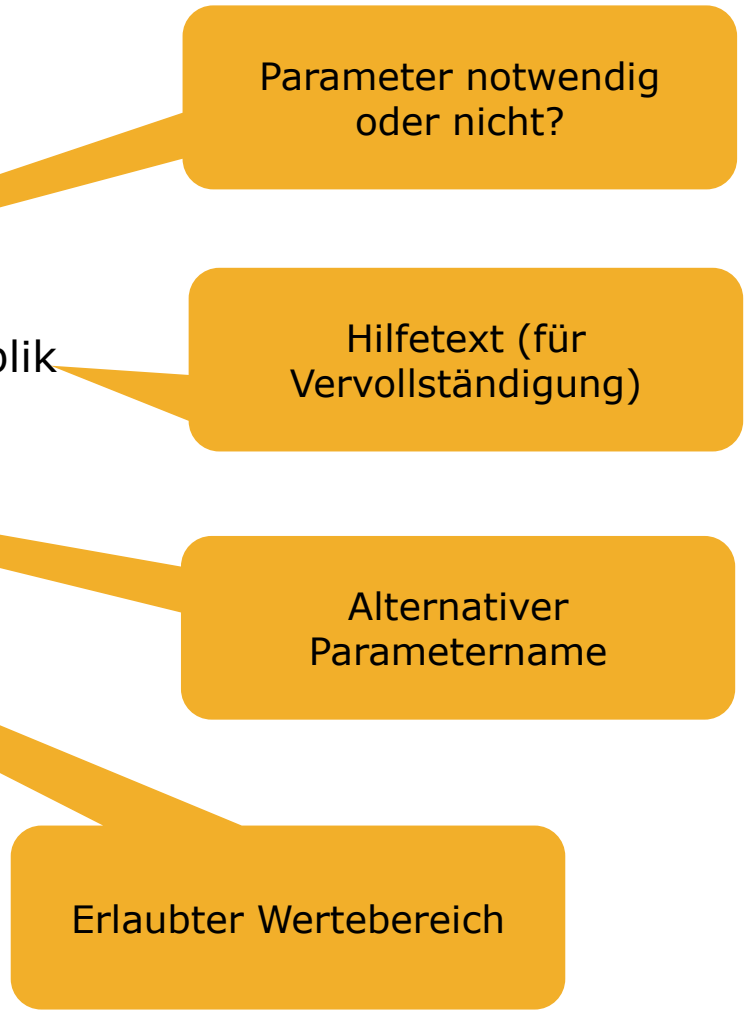
Parameter Advanced

Advanced Parameter Declaration

```
function MyFunction
{
    Param
    (

[Parameter(Mandatory=$true,HelpMessage="Multiplik
ation")]
        [alias("myfirstParam")]
        [ValidateRange(1,10)]
        [Int]$param1
    )
    return $param1 * $param1
}


$erg=0
$erg = MyFunction -myfirstParam 1
echo $erg
$erg = MyFunction -myfirstParam 11
echo $erg
```



Windows Command Prompt Befehle

Multiline String

Ausführen

```
$command = @'
echo Hello World!
'@

Invoke-Expression -Command:$command
```

Ergebnis Lesen

```
$output =  ""
$command = @'
echo Hello World!
'@

$output = Invoke-Expression -
Command:$command

echo "Das habe ich bekommen: $output"
```

Standard .Net Klassen nutzen

Statische Methoden

```
$myvar = 4;
echo $myvar
$myvar =
[System.Math]::Sqrt($myvar)
echo $myvar
```

Library

```
$assembly =
[Reflection.Assembly]::LoadFile("c:\path\file.dll")
$instance = New-Object Class.Of.Assembly
$instance.Property1 = $variable1
$instance.Property2 = $variable2
$instance.Property3 = $variable3
$result = $instance.function()
```

.Net Arrays

Definieren & Anhängen

```
$myArraylist =
[System.Collections.ArrayList]::new()
$maxIndex = $myArraylist.add(1);
$maxIndex = $myArraylist.add(2);

echo $myArraylist[0]
echo $myArraylist[1]
echo "Max Index: $maxIndex"
```

Arraylisten haben **keine** feste Größe!

Entfernen

```
$myArraylist =
[System.Collections.ArrayList]::new()
$maxIndex = $myArraylist.add(1);
$maxIndex = $myArraylist.add(2);

$myArraylist.Remove(1);
echo $myArraylist
```

Auslesen

```
$myArraylist =
[System.Collections.ArrayList]::new(@(1
,2,3))

echo $myArraylist[0,1]
echo "-----"
echo $myArraylist[-1]
echo "-----"
echo $myArraylist[0 .. 2]
```