PowerShell – Eigene Parameter definieren

Neben der Verwendung vorhandener Parameter kann man in PowerShell auch eigene Parameter für Skripte und Funktionen definieren.

Dadurch muss man Werte nicht fest im Skript eintragen, sondern kann sie beim Start des Skripts übergeben.

Warum eigene Parameter sinnvoll sind

Ohne Parameter wären Werte oft fest im Skript eingetragen.

Beispiel ohne Parameter:

```
$Name = "Felix"
Write-Output "Hallo $Name"
```

Das Problem:

Der Name ist fest im Skript eingetragen.
Wenn ein anderer Name verwendet werden soll, muss das Skript geändert werden.

Besser ist ein Parameter:

```
param(
    [string]$Name
)

Write-Output "Hallo $Name"
```

Aufruf:

```
.\gruss.ps1 -Name "Felix"
```

Ausgabe:

```
Hallo Felix
```

Der param()-Block

Eigene Parameter werden in PowerShell meistens mit einem `param()`-Block definiert.

Der `param()`-Block steht in Skripten normalerweise am Anfang der Datei.

Grundform:

```
param(
    [Datentyp]$Parametername
)
```

Beispiel:

```
param(
    [string]$Name
)

Write-Output "Hallo $Name"
```

Bedeutung:

Teil	Erklärung
param()	Bereich, in dem Parameter definiert werden
[string]	Datentyp des Parameters
$Name	Name des Parameters als Variable

Parameter beim Skriptstart übergeben

Ein Skript mit Parameter kann beim Start Werte annehmen.

Beispielskript:

```
param(
    [string]$Name,
    [int]$Alter
)

Write-Output "$Name ist $Alter Jahre alt."
```

Aufruf:

```
.\person.ps1 -Name "Felix" -Alter 30
```

Bedeutung:

Teil	Erklärung
.\person.ps1	Skript wird gestartet
-Name	Parametername
"Felix"	Wert für den Parameter Name
-Alter	Parametername
30	Wert für den Parameter Alter

Datentypen bei Parametern

Man kann festlegen, welchen Datentyp ein Parameter erwartet.

Beispiele:

Datentyp	Bedeutung	Beispiel
[string]	Text	"Felix"
[int]	Ganze Zahl	30
[bool]	Wahr/Falsch	$true
[string[]]	Mehrere Texte	"PC1","PC2"
[switch]	Schalterparameter	-VerboseMode

Beispiel:

```
param(
    [string]$ComputerName,
    [int]$Port
)

Write-Output "Computer: $ComputerName"
Write-Output "Port: $Port"
```

Aufruf:

```
.\test.ps1 -ComputerName "Server01" -Port 443
```

Parameter mit Standardwert

Ein Parameter kann einen Standardwert haben.

Wenn beim Aufruf kein Wert angegeben wird, verwendet PowerShell automatisch den Standardwert.

Beispiel:

```
param(
    [string]$Name = "Benutzer"
)

Write-Output "Hallo $Name"
```

Aufruf ohne Parameter:

```
.\gruss.ps1
```

Ausgabe:

```
Hallo Benutzer
```

Aufruf mit Parameter:

```
.\gruss.ps1 -Name "Felix"
```

Ausgabe:

```
Hallo Felix
```

Pflichtparameter

Ein Parameter kann verpflichtend gemacht werden.

Dafür verwendet man das Attribut:

```
[Parameter(Mandatory)]
```

Beispiel:

```
param(
    [Parameter(Mandatory)]
    [string]$Name
)

Write-Output "Hallo $Name"
```

Wenn der Parameter beim Aufruf fehlt, fragt PowerShell nach dem Wert.

Aufruf:

```
.\gruss.ps1
```

PowerShell fragt dann nach dem fehlenden Parameter.

Parameter-Attribut

Mit `[Parameter()]` kann man Eigenschaften eines Parameters festlegen.

Beispiel:

```
param(
    [Parameter(Mandatory)]
    [string]$ComputerName
)
```

Wichtige Eigenschaften:

Eigenschaft	Bedeutung
Mandatory	Parameter ist verpflichtend
Position	Parameter kann über seine Position erkannt werden
ValueFromPipeline	Wert kann aus der Pipeline kommen
ValueFromPipelineByPropertyName	Wert kann über passenden Eigenschaftsnamen aus der Pipeline kommen
HelpMessage	Hilfetext für den Parameter

Positionelle Parameter

Parameter können auch über ihre Position übergeben werden.

Beispiel:

```
param(
    [Parameter(Position=0)]
    [string]$Name,

    [Parameter(Position=1)]
    [int]$Alter
)

Write-Output "$Name ist $Alter Jahre alt."
```

Aufruf mit Parameternamen:

```
.\person.ps1 -Name "Felix" -Alter 30
```

Aufruf über Position:

```
.\person.ps1 "Felix" 30
```

PowerShell erkennt:

Position	Wert	Parameter
0	"Felix"	Name
1	30	Alter

Für saubere Dokumentation ist die Variante mit Parameternamen meist besser.

Parameter validieren

Parameterwerte können geprüft werden.

Dadurch kann verhindert werden, dass falsche Werte verarbeitet werden.

ValidateSet

Mit `ValidateSet` kann man erlaubte Werte festlegen.

Beispiel:

```
param(
    [ValidateSet("Start","Stop","Restart")]
    [string]$Action
)

Write-Output "Gewählte Aktion: $Action"
```

Erlaubte Aufrufe:

```
.\dienst.ps1 -Action Start

.\dienst.ps1 -Action Stop

.\dienst.ps1 -Action Restart
```

Nicht erlaubt:

```
.\dienst.ps1 -Action Delete
```

PowerShell würde den falschen Wert ablehnen.

ValidateNotNullOrEmpty

Mit `ValidateNotNullOrEmpty` wird verhindert, dass ein leerer Wert übergeben wird.

Beispiel:

```
param(
    [ValidateNotNullOrEmpty()]
    [string]$Name
)

Write-Output "Hallo $Name"
```

Das ist sinnvoll, wenn ein Parameter zwar angegeben wird, aber nicht leer sein darf.

Schalterparameter selbst definieren

Schalterparameter werden mit `[switch]` definiert.

Ein Schalterparameter braucht keinen eigenen Wert.

Beispiel:

```
param(
    [switch]$Ausfuehrlich
)

if ($Ausfuehrlich) {
    Write-Output "Ausführliche Ausgabe aktiviert"
} else {
    Write-Output "Normale Ausgabe"
}
```

Aufruf ohne Schalter:

```
.\test.ps1
```

Aufruf mit Schalter:

```
.\test.ps1 -Ausfuehrlich
```

Bedeutung:

Der Parameter ist aktiv, sobald er angegeben wird.

Parameter in Funktionen

Parameter können nicht nur in Skripten, sondern auch in Funktionen verwendet werden.

Beispiel:

```
function Begruessung {
    param(
        [string]$Name
```

```
    )

    Write-Output "Hallo $Name"
}
```

Aufruf:

```
Begruessung -Name "Felix"
```

Ausgabe:

```
Hallo Felix
```

Erweiterte Funktion mit Parametern

Eine etwas sauberere Funktion:

```
function Test-Verbindung {
    param(
        [Parameter(Mandatory)]
        [string]$ComputerName
    )

    Test-Connection -ComputerName $ComputerName -Count 2
}
```

Aufruf:

```
Test-Verbindung -ComputerName "server01"
```

Hier nimmt die eigene Funktion den Parameter `-ComputerName` entgegen und verwendet ihn intern für `Test-Connection`.

Unbenannte Parameter mit $args

PowerShell kann auch über `$args` Werte entgegennehmen.

Beispiel:

```
Write-Output "Erster Wert: $($args[0])"
Write-Output "Zweiter Wert: $($args[1])"
```

Aufruf:

```
.\test.ps1 Felix Berlin
```

Ausgabe:

```
Erster Wert: Felix
Zweiter Wert: Berlin
```

Das funktioniert, ist aber weniger sauber als ein richtiger `param()`-Block.

Besser:

```
param(
    [string]$Name,
    [string]$Ort
)

Write-Output "$Name aus $Ort"
```

Aufruf:

```
.\test.ps1 -Name "Felix" -Ort "Berlin"
```

Named Parameters vs. Positional Parameters

Es gibt zwei typische Arten, Parameter zu übergeben.

Art	Beispiel	Erklärung
Named Parameter	.\test.ps1 -Name "Felix"	Parametername wird angegeben
Positional Parameter	.\test.ps1 "Felix"	PowerShell erkennt den Parameter über die Position

Named Parameters sind besser lesbar und weniger fehleranfällig.

Mehrere Werte übergeben

Ein Parameter kann mehrere Werte annehmen, wenn er als Array definiert ist.

Beispiel:

```
param(
    [string[]]$ComputerName
)

foreach ($Computer in $ComputerName) {
    Write-Output "Prüfe $Computer"
}
```

Aufruf:

```
.\check.ps1 -ComputerName "PC1","PC2","PC3"
```

Ausgabe:

```
Prüfe PC1
Prüfe PC2
Prüfe PC3
```

Advanced Function und CmdletBinding

Mit `[CmdletBinding()]` kann eine Funktion sich stärker wie ein echtes Cmdlet verhalten.

Beispiel:

```
function Remove-TestDatei {
    [CmdletBinding(SupportsShouldProcess)]
    param(
        [Parameter(Mandatory)]
        [string]$Path
    )

    if ($PSCmdlet.ShouldProcess($Path, "Datei löschen")) {
        Remove-Item -Path $Path
```

```
    }
}
```

Vorteil:

Die Funktion unterstützt dadurch zum Beispiel `-WhatIf`.

Aufruf:

```
Remove-TestDatei -Path "C:\Temp\test.txt" -WhatIf
```

PowerShell zeigt dann nur an, was passieren würde.

Das ist besonders wichtig bei Funktionen, die etwas verändern oder löschen.

Warum Parameter besser sind als feste Werte

Schlechter Stil:

```
$ComputerName = "Server01"
Test-Connection -ComputerName $ComputerName
```

Besser:

```
param(
    [string]$ComputerName
)

Test-Connection -ComputerName $ComputerName
```

Aufruf:

```
.\ping.ps1 -ComputerName "Server01"
```

Vorteile:

Vorteil	Erklärung
Wiederverwendbar	Skript funktioniert mit verschiedenen Werten
Flexibler	Werte werden beim Start übergeben

Vorteil	Erklärung
Wartbarer	Skript muss nicht ständig geändert werden
Sicherer	Werte können validiert werden
Professioneller	Entspricht eher echter Administration

IHK-Einordnung

Für die IHK ist vor allem wichtig, das Prinzip zu verstehen:

Ein Parameter macht Skripte und Funktionen flexibel.

Man schreibt nicht alle Werte fest in den Code, sondern übergibt sie beim Aufruf.

Wichtige Punkte:

Punkt	Bedeutung
param()	Definiert eigene Parameter
[string], [int], [switch]	Legt Datentypen fest
Mandatory	Macht Parameter verpflichtend
Position	Erlaubt positionelle Übergabe
ValidateSet	Beschränkt erlaubte Werte
ValidateNotNullOrEmpty	Verhindert leere Werte
[switch]	Erstellt Schalterparameter
$args	Enthält unbenannte Argumente
CmdletBinding	Macht Funktionen cmdletähnlicher

Prüfungsnahes Beispiel

Aufgabe:

Ein Skript soll einen Computernamen als Parameter entgegennehmen und eine Verbindung prüfen.

Lösung:

```
param(
    [Parameter(Mandatory)]
    [string]$ComputerName
)
```

```
Test-Connection -ComputerName $ComputerName -Count 2
```

Aufruf:

```
.\check.ps1 -ComputerName "Server01"
```

Erklärung:

Teil	Bedeutung
param()	Definiert Eingabeparameter
[Parameter(Mandatory)]	Parameter ist verpflichtend
[string]	Erwartet Text
$ComputerName	Variable für den übergebenen Wert
Test-Connection	Prüft Netzwerkverbindung
-ComputerName $ComputerName	Nutzt den übergebenen Wert

Kurzer Merksatz

Mit `param()` definiert man eigene Parameter für PowerShell-Skripte oder Funktionen. Dadurch werden Skripte flexibel, wiederverwendbar und professioneller.
