Windows PowerShell

Windows hat jetzt auch eine Shell ;-)

Michael Elschner

hitforum

25. November 2009

Inhalt

Windows PowerShell

1 Einführung

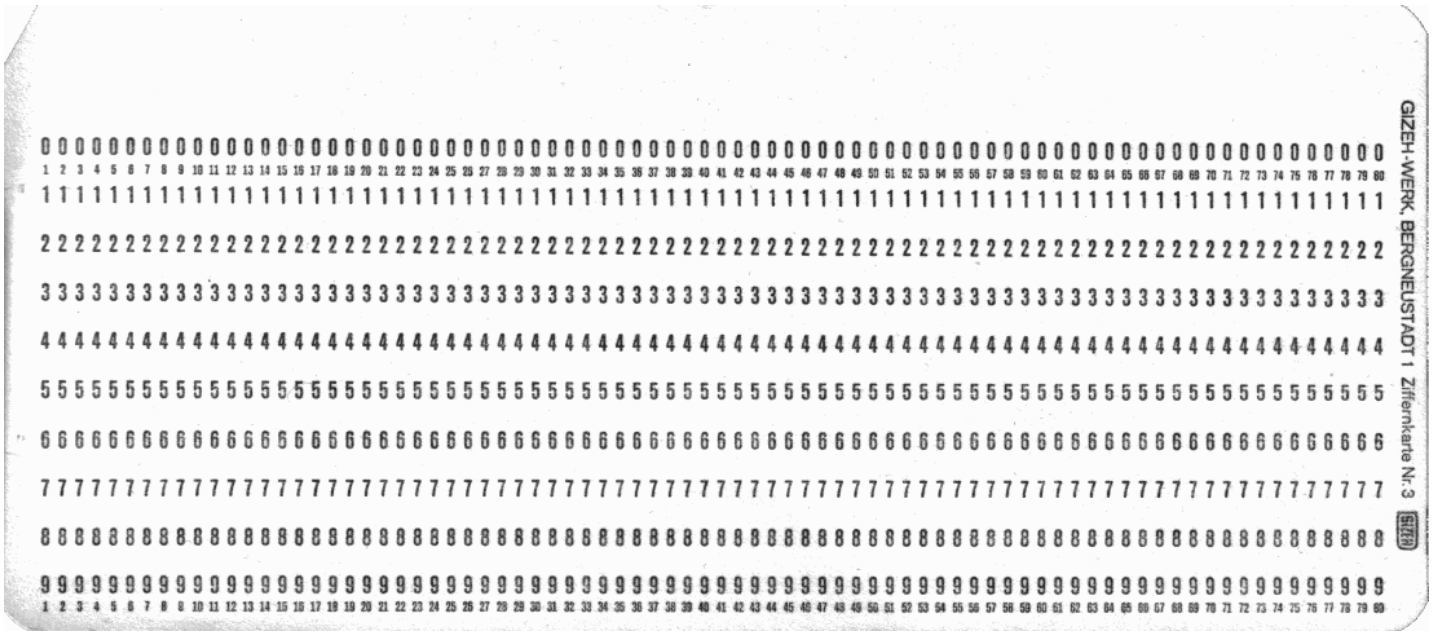
Windows PowerShell

1 Einführung

1 Entwicklung der Benutzerschnittstellen

Entwicklung der Benutzerschnittstellen (1)

Am Anfang war. . .



. . . die Lochkarte und dann ziemlich lange nichts. . .

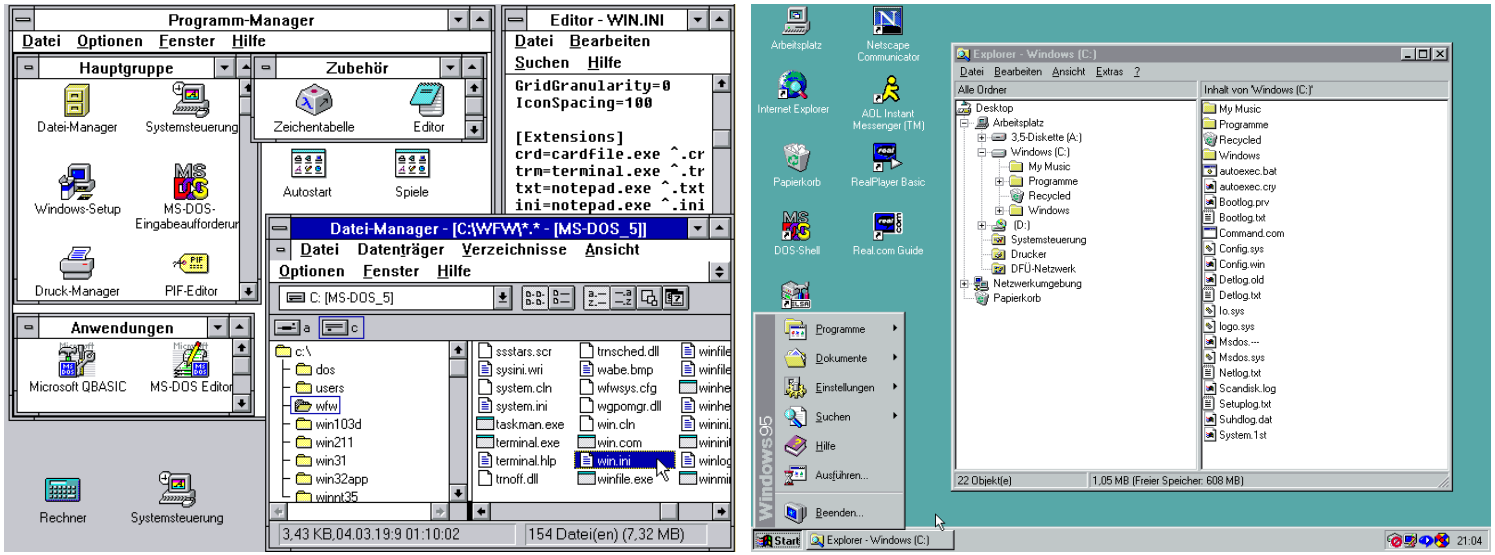
Entwicklung der Benutzerschnittstellen (2)

. . . dann kamen die Kommandozeilen (CLIs)
die einiges von den Lochkarten erbten.

- interaktive Bedienung über ein Terminal
- ursprünglich 80 Spalten und 24-25 Zeilen
- Kommandos werden zeilenweise eingegeben und interpretiert, sie wechseln sich mit Rückmeldungen des Systems ab
- alternativ:
  Zusammenfassen mehrerer Kommandos für eine Batch-Verarbeitung

Entwicklung der Benutzerschnittstellen (3)

Dann kamen die die Graphischen Benutzeroberflächen (GUIs).



Gerade die Betriebsysteme eines namhaften Redmonder Software-Herstellers vernachlässigten ihre CLI.

Das hat sich mit der PowerShell geändert!

Warum überhaupt (noch) eine Kommandozeile?

- Geschwindigkeit
  - Tippen ist vielleicht nicht so „intuitiv“ wie Klicken,
  - aber mit ein wenig Übung auf jeden Fall schneller
- Stabilität der Benutzeroberfläche
  - uralte DOS-Kommandos funktionieren auch noch in der `cmd.exe`
  - uralte Unix-Kommandos funktionieren auch in modernen Derivaten
  - die Windows-GUI ändert sich dagegen mit nahezu jeder Version!
- Skriptingfähigkeit
  - einfaches Automatisieren von Abläufen
- Manche Dinge können – auch in Windows – nur auf der Konsole getan werden!

Historie der Microsoft-Kommandozeilen

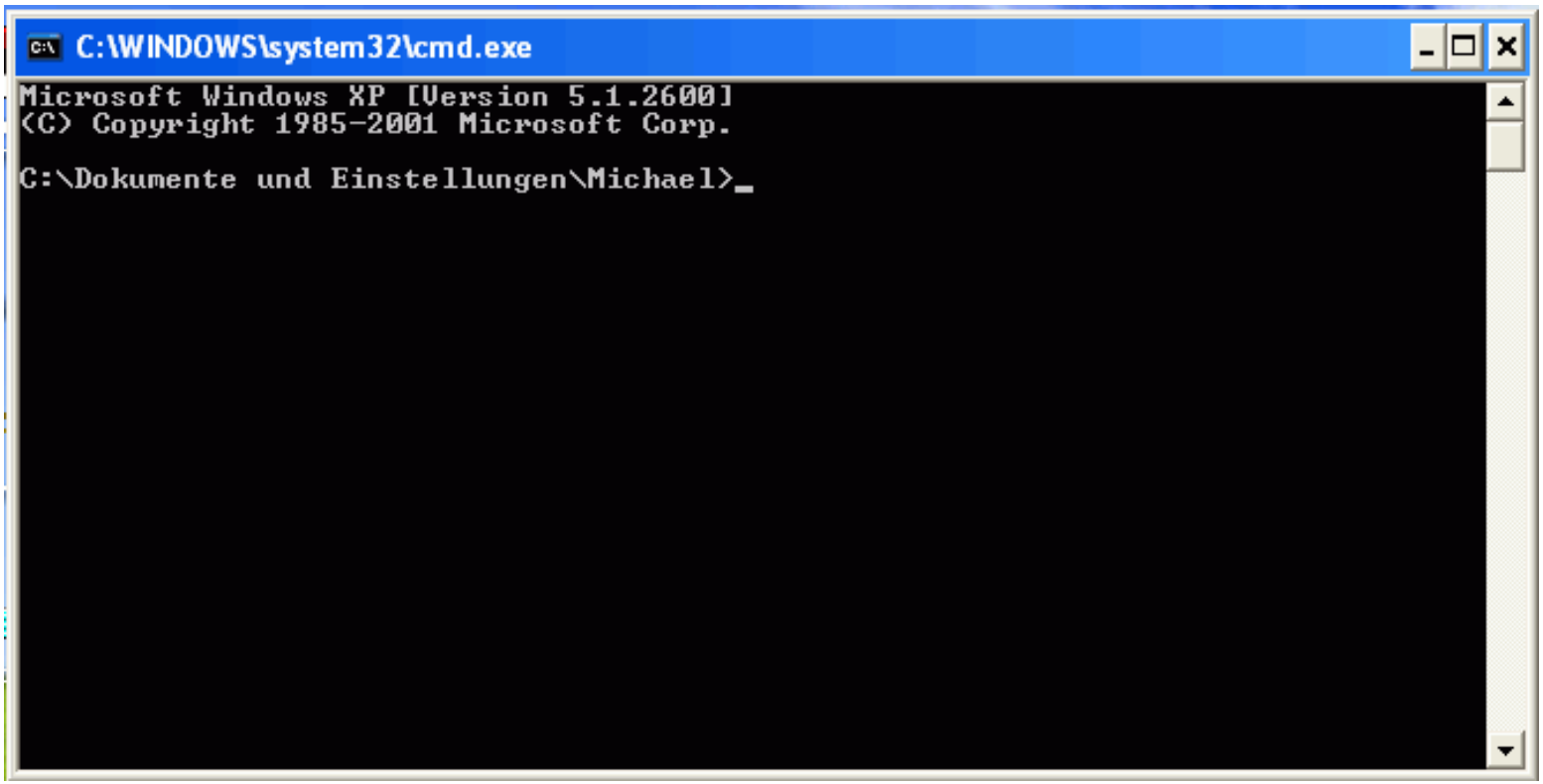
- DOS, das Vorgänger-„Betriebssystem“
  - keine graphische Benutzeroberfläche (GUI)
  - Kommandozeile `command.com` als Default-Bedienoberfläche
  - die eigentliche „DOS-Shell“
- Windows-Versionen bis ME
  - setzen auf DOS auf
  - `command.com` als „MS-DOS-Eingabeaufforderung“ aufrufbar
- Windows-Versionen ab NT
  - Windows setzt nicht mehr auf DOS auf
  - `cmd.exe` als „Eingabeaufforderung“ aufrufbar
  - `cmd.exe` wurde jedoch nur wenig weiterentwickelt

Windows PowerShell

1 Einführung

2 „Highlights“ der CMD-Shell

Wir erinnern uns: So „toll“ ist die CMD-Shell



„Obsfucated“ Programming: Eine Erfindung von DOS?

DOS-Batch als „esoterischen Programmiersprache“?

- undurchsichtige Mehrfachfunktione: z. B. `echo`, `date`
- geschwätzige Befehle: z. B. `copy`, `move`
- keine strukturierte Programmierung
- generell eher schlechte Dokumentation:
  man vergleiche nur `help` mit `man`

Mehrfach belegte Befehle: Der `echo`-Befehl

- `echo` gefolgt von einer Nachricht gibt diese Nachricht aus
- `echo` gefolgt von „on“ oder „off“ schaltet die Befehlsanzeige ein oder aus
- `echo` alleine gibt den Status der Befehlsanzeige aus
- Um eine leere Zeile bzw. ein „on“ oder „off“ auszugeben, wird der Befehl `echo.` gebraucht.

Erzwungen interaktive Befehle: Der `date`-Befehl

- `date` gibt nicht nur das aktuelle Datum aus,
  sondern erwartet auch eine Eingabe zum Setzen des Datums
- Ältere CMD-Versionen benötigten „interessante“ Konstrukte für den Zugriff aufs Datum:
  ```
  echo. | date | find "Datum:"
  ```
- Neuere Versionen kennen die „/T“-Option, um eine Eingabe auszuschließen.

Geschwätzige Befehle: Der `copy`-Befehl

- `copy` kopiert nicht nur Dateien,
  sondern quittiert dies immer auch mit einer Meldung:
  n `Datei(en) kopiert`
- wenn das stört, muss die Ausgabe explizit verworfen werden:
  `copy` *a b* `>NUL`

Strukturierte Programmierung? IF und GOTO!

- Es gibt keine einfache Schleifenanweisung!
- Schleifen müssen also durch IF und GOTO simuliert werden!

```
:BeginnDerSchleife
<Anweisung>
IF <Bedingung> GOTO :BeginnDerSchleife
:EndeDerSchleife
```

- Es gibt allerdings einen FOR-Befehl, der z. B. für das Iterieren über Dateien oder Zahlenwerte benutzt werden kann.

Obskures Verhalten: ENABLEDELAYEDEXPANSION

- Variablen werden direkt beim Auswerten des Befehls ersetzt, IF-Blöcke, FOR-Blöcke o.ä. gelten als ein Befehl!
- Erstellen einer Liste der Dateien im Verzeichnis (fehlerhaft):

```
set LISTE=
for %%i in (*) do set LISTE=%LISTE% %%i
echo %LISTE%
```

- Erstellen einer Liste der Dateien im Verzeichnis mit ENABLEDELAYEDEXPANSION (richtig):

```
set LISTE=
for %%i in (*) do set LISTE=!LISTE! %%i
echo %LISTE%
```

Lokale Variablen? Nur mit SETLOCAL

- Komplexere Skripte brauchen Variablen
- CMD kennt standardmäßig keine skriptlokalen Variablen (analog dem Unix-Befehl `export`)
- Abhilfe schafft die Erweiterung SETLOCAL

⇒ siehe den Unterschied zwischen den beiden `for`-Skripten

Vorspann für CMD-Skripte

CMD-Skripte sind also eigentlich nur mit folgendem Vorspann brauchbar:

```
VERIFY OTHER 2>nul
SETLOCAL ENABLEEXTENSIONS ENABLEDELAYEDEXPANSION
IF ERRORLEVEL 1 (
   echo Keine Befehlserweiterungen
   exit /b 8
)
```

Obsfucated Boole: Wozu boole'sche Operatoren?

- CMD-Skripte kennen keine boole'schen Operatoren
- Ein AND lässt sich aber „prima“ mit verketteten IF-Anweisungen erreichen:

```
IF <Bedingung> ^
IF <Bedingung> ^
   <Anweisung>
```

- . . . und ein OR fast genauso „gut“ mit GOTO

```
IF <Bedingung> GOTO BedingungErfuellt
IF NOT <Bedingung> ^
   GOTO BedingungNichtErfuellt
:BedingungErfuellt
 <Anweisung>
:BedingungNichtErfuellt
```

Fazit: Die CMD-Shell war und ist krank!

- Hauptgrund für die Benutzung ist die weite Verbreitung.
- Auf jedem Windows-System zu finden.
  Vorsicht: In älteren Versionen aber nicht alle Erweiterungen vorhanden.
- Für viele Sachen werden Hilfsprogramme benötigt, die dann auch verteilt werden müssen.
  - Setzen eines Dateidatums
  - Abwarten einer bestimmten Zeit
  - generelle Dateiverarbeitung
- Die Programmierung ist generell eher umständlich:
  - Nur eingeschränkte Strukturen; `goto`-lastige Programmierung
  - Stringverarbeitung leidlich möglich
  - Aktivieren von Erweiterungen muss geprüft werden

Windows PowerShell

2 Grundlagen der PowerShell

Windows PowerShell

2 Grundlagen der PowerShell

1 Grundlegendes

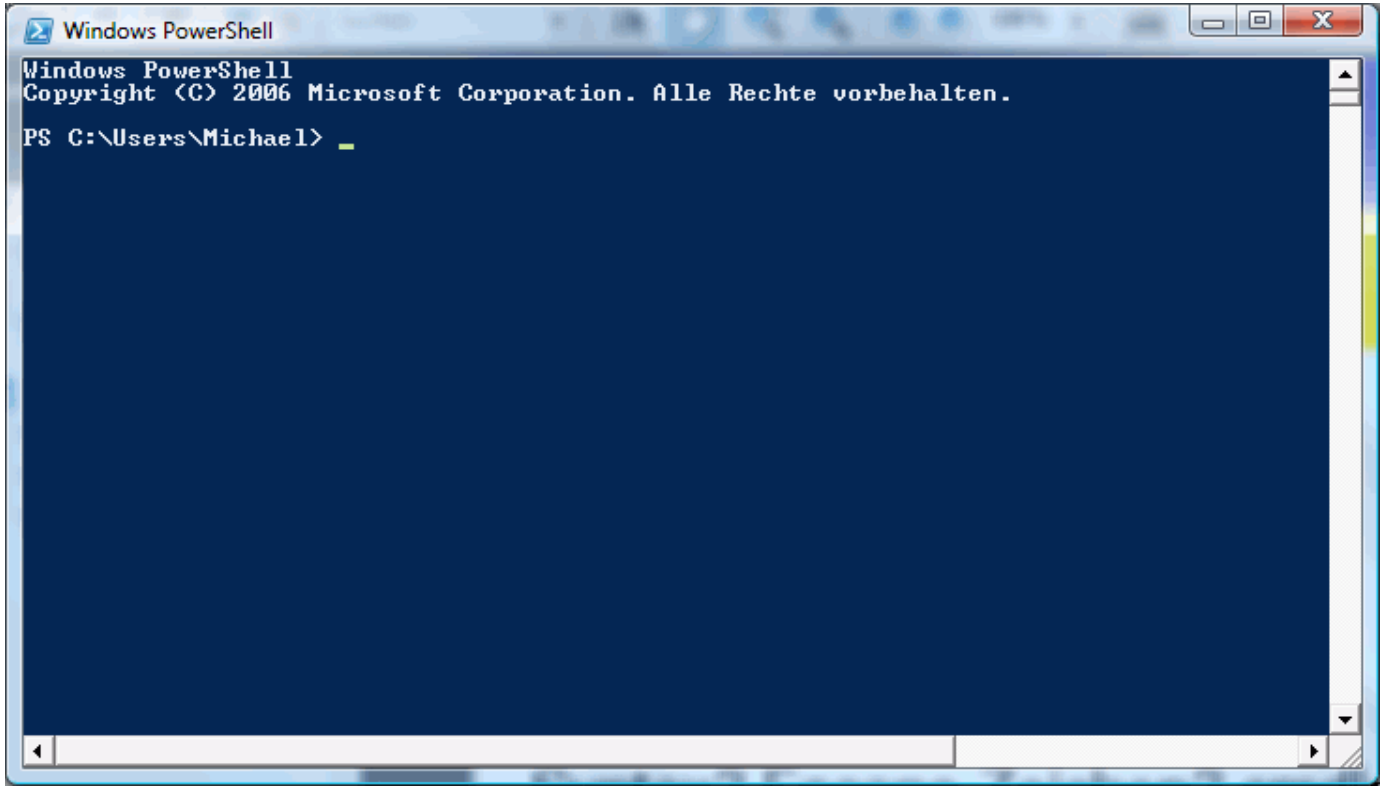
Historie

- Microsoft hat sich in der Entwicklung von Windows lange nur auf die GUI konzentriert.
- Die CMD-Shell fiel daher schnell hinter andere CLIs zurück, so sie es denn nicht schon war. . .
- Lösungen wie der Windows Scripting Host (WSH) verfolgen eine andere Zielsetzung, was man schon am Fehlen einer interaktiven Schnittstelle sieht.
- Schließlich: Entwicklung der PowerShell (PS)

Verfügbarkeit

- bereits vorhanden in
  - Windows Server 2008
  - Windows 7
- kann von Microsoft heruntergeladen für
  - Windows XP SP 2
  - Windows Server 2003 SP 1
  - Windows Vista
- weitere Voraussetzung: .NET-Framework 2.0
  In Server-Varianten ohne GUI ist sie daher nicht nutzbar!

Der erste Blick. . .



. . . unterscheidet sich nicht viel von der CMD

Grundlagen der PowerShell

- von Unix-Shells „inspiriert“
- Grundlage der Sprache sind die Cmdlets (sprich: „Commandlets“), bei denen es sich um spezielle .NET-Klassen handelt
- Cmdlets folgen allgemein einer *Verb-Substantiv*-Syntax
- für Umsteiger von anderen Shells sind *Aliase* definiert
- es können aber auch normale Programme aufgerufen werden
- naturgemäß das ganze .NET-Framework nutzbar
- aber auch Zugriff auf die WMI (Windows Management Instrumentations) und COM

Einfache Cmdlets: Verzeichnisse

Get-Location
: Ausgabe des aktuellen Verzeichnisses

Set-Location
: aktuelles Verzeichnis wechseln

Get-ChildItem
: Auflistung des Verzeichnis-Inhalts

Einfache Cmdlets: Dateien

Copy-Item
: Kopieren einer oder mehrerer Dateien / eines ganzen Verzeichnisbaums

Remove-Item
: Löschen einer Datei / eines Verzeichnisses

Rename-Item
: Umbenennen einer Datei / eines Verzeichnisses

Move-Item
: Verschieben einer Datei / eines Verzeichnisses

Get-Content
: Ausgabe einer Datei

Einfache Cmdlets: Hilfe

Get-Help

Hilfe zu einem Befehl anfordern

Get-Command –Noun *a*

alle Kommandos mit dem *Substantiv a* auflisten

Einfache Cmdlets: Prozessverwaltung

Get-Process

Liste aller momentan laufenden Prozesse

Stop-Process

Beenden eines laufenden Prozesses

Optionen

Optionen werden wie in Unix-Shells mit einem „–“-Zeichen angegeben:

- -Verbose
  endlich sind die Befehle im Normalfall weniger „geschwätzig“
- -Confirm
  immer Bestätigung abfragen
- -WhatIf
  „Was wohl passiert, wenn ich diesen Knopf hier drücke. . . “

Die hier aufgeführten Optionen sind in allen Standard-Cmdlets vorhanden.

Aliase

- Für Cmdlets können Aliase definiert werden,
  mit denen sie dann wahlweise aufgerufen werden können.
- Set-Alias
- Get-Alias
- Get-Alias *a*

z. B. Aliase für Get-ChildItem: `dir`, `ls` oder `gci`

Provider

- Mittels Set-Location und Get-ChildItem kann nicht nur das Dateisystem durchlaufen werden!
- Neben „Laufwerken“ wie „C:“, „D:“ existieren so auch „virtuelle Laufwerke“ für besondere Zwecke:

Alias:	Definierte Aliase
Env:	Umgebungsvariablen
HKLM:	Registry-Schlüssel „HKEY_LOCAL_MACHINE“
HKCU:	Registry-Schlüssel „HKEY_CURRENT_USER“
Variable:	alle definierten Variablen
Function:	alle definierten Funktionen

- Abfrage aller Provider mit Get-PSProvider bzw. Get-PSDrive

Windows PowerShell

2 Grundlagen der PowerShell

2 Pipes Reloaded

Pipes

- Ausgaben eines Kommandos werden an ein anderes Kommando weitergeleitet
- zur Kennzeichnung wird üblicherweise das „|“-Zeichen verwendet
- Geschicktes Verknüpfen von einfachen Befehlen erzielt komplexe Effekte
- bekannt von Systemen wie Unix, DOS, OS/2 oder Windows
- Möglichkeiten unter Windows bisher relativ beschränkt, da wenige Kommandos mit vollständiger Unterstützung

Pipes in der PowerShell

- auch in der PS kann die Ausgabe eines Cmdlets an ein anderes weitergereicht werden
- zum Einsatz kommt ebenso das „|“-Zeichen
- bei der PS arbeiten die Pipes jedoch auf Basis von Objekten
- Vorteil: mit Objekten kann direkt weitergearbeitet werden, während Text immer neu geparst werden muss
- jedes Kommando liest also Objekte ein und gibt Objekte aus

Formatierer

Wenn alles Objekte sind, warum sieht man Text?

- spätestens am Ende einer Befehlskette muss etwas menschenlesbares präsentiert werden
- die Objekte werden am Ende der Pipe daher aufbereitet
- jedes Objekt hat dafür eine Standard-Formatierer-Methode
- alternativ kann man auch andere Formatierer anwenden

Alternative Formatierer

Vergleiche die Ausgaben der folgenden Befehle!

- `Get-Process powershell | Format-List`
- `Get-Process powershell | Format-List Id, ProcessName, VirtualMemorySize, Handles`
- `Get-Process powershell | Format-List *`
- `Get-Process powershell | Format-Table Id, ProcessName, VirtualMemorySize, Handles`
- `Get-Process | Format-Table -GroupBy PriorityClass ProcessName, VirtualMemorySize, Handles`
- `Get-Date`
- `Get-Date | Format-List`

Anwendung für Pipes: Filter-Cmdlets

Filter sind in allen CLIs *die* Parade-Anwendung für Pipes, so auch in der PowerShell:

- `Sort-Object`
- `Where-Object`
- `Group-Object`

Beispiel-Pipes

- Prozess-Liste, sortiert nach Prozessnamen

  CMD-Shell

  ```
  tasklist | sort
  ```

  PowerShell

  ```
  Get-Process | Sort-Object ProcessName
  ```

- Prozess-Liste aller Prozesse mit dem Namen „firefox“

  CMD-Shell

  ```
  tasklist | find "firefox"
  ```

  PowerShell

  ```
  Get-Process | Where-Object
  { $_.ProcessName -eq "firefox" }
  ```

Weiteres Beispiel

Wie frage ich die Aliase ab, die für `Get-ChildItem` definiert sind?

```
Get-Alias * | Where-Object { $_.Definition -eq "Get-ChildItem"

CommandType     Name                                               Definition
-----------     ----                                               ----------
Alias           gci                                                Get-ChildItem
Alias           ls                                                 Get-ChildItem
Alias           dir                                                Get-ChildItem
```

Die implizite Variable $_ verweist auf das aktuelle Objekt der Pipe.

Windows PowerShell

3 Skripting

Windows PowerShell

3 Skripting

1 Sprachkonzepte

Objektorientierung

- alles in der PS ist ein .NET-Objekt
- `42 | Get-Member`
- `"Hitforum" | Get-Member`
- `"Hitforum".length`
- `(Get-ChildItem).Count`

String-Quoting

- doppelte und einfache Anführungszeichen
- bei doppelten Anführungszeichen werden Variablen ersetzt und Escape-Sequenzen ausgewertet

Escape-Sequenzen:

`'	einfaches Anführungszeichen	`b	Backspace
`"	doppeltes Anführungszeichen	`f	Wagenrücklauf
``	der Gravis	`n	Zeilenvorschub
`$	Dollar-Zeichen	`r	Wagenrücklauf
`0	0-Zeichen	`t	Tabulator
`a	Alert, Bell	`v	vertikaler Tabulator

Aus irgendeinem Grund benutzt Microsoft nicht den Backslash. . . ;-)

Operatoren

Arithmetisch

Addition	+	bei Strings Verkettung
Subtraktion	-	
Multiplikation	*	bei Strings Wiederholung
Division	/	
Modulus	%	

```
Get-Help about_Arithmetic_Operators
```

Vergleich

kleiner	-lt
kleiner oder gleich	-le
gleich	-eq
größer oder gleich	-ge
größer	-gt
ungleich	-ne

```
Get-Help
about_Comparison_Operators
```

Logisch

-and	und
-or	oder
-xor	exklusiv-oder
-not, !	nicht

Weitere Operatoren

Matching:

Wildcard-Matching	-like, -notlike
Regex-Matching	-match, -notmatch
Regex-Ersetzung	-replace

clike und cmatch beruecksichtigen groß und kleinschreibung

```
Get-Help about_Wildcard
```

Typen:

-as	Typumwandlung
-is, -isnot	ist (nicht) vom Typ

Spezialfälle:

-f (Formatierung)	spezielle .Net-String-Formatierung
-contains, -notcontains	in Liste (nicht) enthalten

Typabhängiges Verhalten

Das Verhalten der Operatoren hängt von den beteiligten Datentypen ab:

- `2 + 2`
- `2 + "2"`
- `2 * "hallo"` **vs.** `"hallo" * 2`
- `3 -is [int]` **vs.** `3/2 -is [int]`
- `[int](3/2)` **vs.** `[int](5/2)`

Variablen

- Variablennamen werden mit einem `$` eingeleitet und bestehen aus alphanumerischen Zeichen
- in `{ }` eingeschlossene Namen enthalten beliebige Zeichen
- Zuweisungen mit dem Operator `=` oder `+=`, `-=`, `*=`, `/=`
- Inkrement `++` und Dekrement `--`
- Gültige Variablennamen: `$Hitforum`, aber auch `$$` und `${|-| ! T /= () R \/ |V|}`
- Variablen können auch typisiert werden:

```
[int]$a =7
$a ="Twenty"
```

Systemvariablen

$null	das leere Objekt
$true	der boole'sche Wert „wahr“
$false	der boole'sche Wert „wahr“
$foreach	Enumerator einer foreach-Schleife
$LastExitCode	der Exit-Code des letzten Kommandos
$error	Informationen über den letzten Fehler
$stackTrace	Stacktrace über letzten Fehler
	und weitere. . .

Gültigkeitsbereiche

- Die PS kennt verschiedene Gültigkeitsebenen:

global	skriptweit
local	Im Ursprungsblock und allen aufgerufenen Blöcken.
private	Nur im Ursprungsblock.

- Der Default ist local.
- expliziter Zugriff mit `$global:a`, `$local:a` oder `$a`

Lokaler Gültigkeitsbereich

```
function showA { Write-Host $a }
$a = 10
showA
```

```
function showA { Write-Host $a }
showA
$a = 10
```

Arrays und Hashtabellen

- Arrays
  - $arr = ()
  - $arr = (1..6)
  - $arr = (1,2,3,4)
  - $arr[0]
  - $arr += 6
- Hashtabellen
  - $capital = {}
  - $capital["Deutschland"] = "Berlin"
  - $capital["Belgien"] = "Brüssel"

```
Get-Help about_array
```

Windows PowerShell

3 Skripting

2 Skripting

Skripte

- Skripte können in „.ps1“-Dateien abgelegt werden
- Kommentare werden mit „#“ eingeleitet
- für bessere Lesbarkeit empfiehlt es sich,
  die Langform der Befehle zu nutzen

Execution Policy

- Die „Execution Policy“ legt Regeln für das Ausführen von Skripten fest.
  Abfrage mit `Get-ExecutionPolicy`.
- In der Standardkonfiguration (Restricted) dürfen in der PS keine Skripte ausgeführt werden.
- Sie muss daher zunächst anders gesetzt werden.
  `Set-ExecutionPolicy {RemoteSigned}`
- Nun werden lokale Skripte ausgeführt, entfernte jedoch nur bei vorhandener Signatur.

Debuggen

Tabelle!

- Set-PSDebug -trace 1 — Echo der Zeilen
- Set-PSDebug -trace 2 — Echo der Variablenzuweisungen
- Set-PSDebug -step — schrittweises Ausführen

Ablaufsteuerung

- if-Anweisung

```
if () {} elseif () {} else {}
```

- switch-Anweisung

```
switch (<ausdruck>) {
    {<ausdruck mit $_>} {}
    <Literal> {}
    default {}
}
```

- for-Anweisung

```
for (<ausdruck>; <ausdruck>; <ausdruck>) { ... }
```

- while-Anweisung

```
while (<ausdruck>) { ... }
```

- foreach-Anweisung

```
foreach (a in b) { ... }
```

Funktionen und Filter

- `function <name> {}`
- Filter wenden immer wieder die gleiche Operation auf die Elemente der Eingabe-Pipe an
- Filter sind ein wichtiger Bestandteil der PS
- deswegen haben sie in der PS eine besondere Syntax
- Verwendung der $_-Variable für die Elemente der Eingabe-Pipe
- Beispiel: `filter double { $_ * 2}`
- Verwendung: `(1, 2, 3, 4) | double`

Windows PowerShell

4 Beispiele

Hello $name

```
1 $name = Read-Host "Wie heißt du?"
2 Write-Output "Hallo $name"
3 "Hallo $name"
4 "Hallo $name" | Format-List *
```

PS und COM: Fernsteuerung von Programmen

```
1  $wsh = New-Object -ComObject "WScript.Shell"
2  $wsh.Run("notepad")
3
4  Start-Sleep 1
5
6  if ($wsh.AppActivate("Editor") ) {
7      "Notepad aktiviert"
8
9      $keystrokes = "H", "I", "T", "-", "F", "o", "r", "u",
           "m"
10
11     foreach ($key in $keystrokes) {
12         Start-Sleep 1
13         "Schicke $key"
14         $wsh.SendKeys($key)
15     }
16 }
```

WMI: Systeminformationen abfragen

```
1 $(Get-WmiObject Win32_operatingSystem) | Format-List
      *
```

WWW und XML: Heise Newsticker abfragen

```
1 $wc = New-Object System.Net.WebClient
2 $xml = [Xml]$wc.DownloadString("http://www.heise.de/
      newsticker/heise-atom.xml")
3 $xml.GetElementsByTagName("entry") | Format-Table
      title
```