PowerShell Basics – Kapitel 1

# Unterschiedliche Typen

PowerShell ist unter anderem deshalb so großartig, weil es einem so manches Denken abnimmt, indem man die Variablen nicht deklinieren muss. Manchmal kann das aber echt zum Problem werden, nämlich wenn man gewisse Operationen machen möchte, diese aber gar nicht auf den Typ passen.

## Zeichen und Zahlen – Strings und Ints

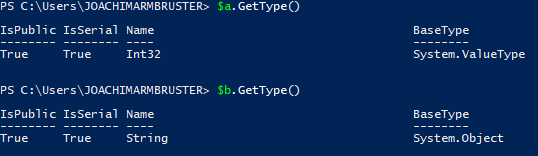
Nimmt man z.B. folgende Zuweisung:

$a = 1

$b = "1"



So bekommt man bei $a einen Int. Bei $b jedoch einen String. Den Typ einer Variable kann man auslesen, indem man „.GetType()“ dahinter schreibt. Also z.B. $a.GetTyp()



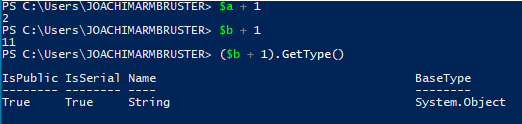
Beide Werte lassen sich in PowerShell trotzdem problemlos „addieren“, nur leider mit unterschiedlichem Ergebnis:

$a + 1

Ergebnis: 2

$b + 1

Ergebnis: 11



Warum? Weil die 1 in $a eine „Zahl“ aber die 1 in $b ein „Zeichen“ (wie ein Buchstabe) ist.

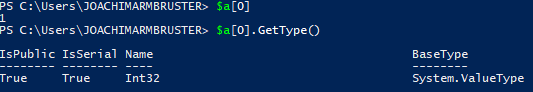
Das erkennt man recht gut, wenn man mittels der eckigen Klammer die einzelnen Elemente der Variablen begutachtet.

$a[0]

Ergebnis: 1

$a[0].GetType()

Ergebnis: Int32



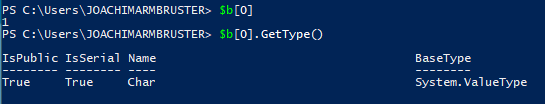
Dasselbe für den String.

$b[0]

Ergebnis: 1

$b[0].GetType()

Ergebnis: Char



Aber was ist das denn jetzt?

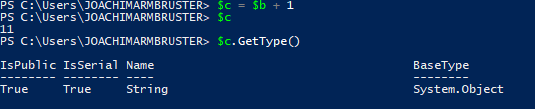
Der String $b beschreibt das Objekt als Ganzes. Wenn wir die eckige Klammer dahinter setzen, springen wir in das so-und-so-vielte Element des gesamten Strings. Ein kleines Beispiel:

Mehr dazu erfährst du in „Wie funktionieren Elemente?“

Addieren wir nun als $b + 1 und schreiben das in die Variable $c

$c = $b + 1

So bekommen wir einen String der aus 2 hintereinander folgenden „1“ Zeichen besteht.



Machen wir wieder +1, bekommen wir 3 x das Zeichen 1, etc.

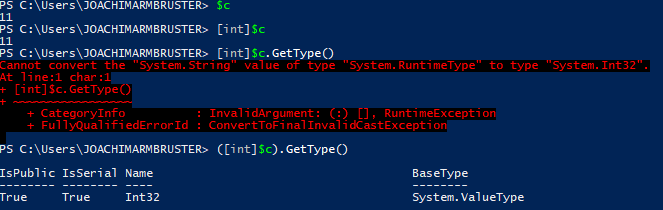
Wenn wir jetzt jedoch wirklich die Zeichenkette „11“ als Zahle „elf“ haben möchten, können wir die Variable entsprechend casten, indem wir den gewünschten Typ in eckige Klammern davor setzen:

[int]$c

Sieht gleich aus… ist aber eine Zahl. Das sehen wir mittels folgendem Befehl.

([int]$c).GetType()

Wir brauche die Klammer um [int]$c damit erst die Variable ale Int gecastet wird, bevor der Operator .GetType() loslegt. Würde man die Klammer weglassen, würde er erst den .GetType von $c machen und dann versuchen diese komplette Ergebnis als Int zu casten, was fehlschlagen wird.



Damit lässt sich rechnen!

[int]$c + 1

Ergebnis: 12



Na endlich.

PowerShell geht aber noch weiter… Denn manchmal interpretiert es sogar Strings unterschiedlich, je nach Operator.

$b

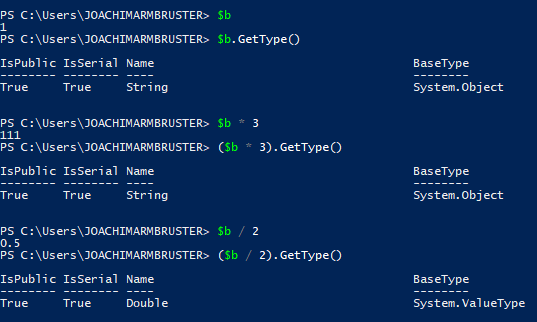
Ergebnis: 1 (String)

$b \* 3

Ergebnis: 111 (String)

$b / 2

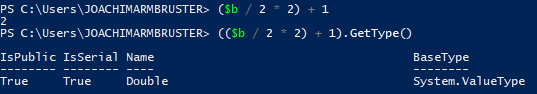
Ergebnis: 0.5 (Double)



Weil „geteilt durch 2“ definitiv eine mathematische Aktion ist, interpretiert er dafür $b als „Zahl“ und bekommt als Ergebnis ebenfalls eine Zahl, nämlich eine Double.

Jetzt geht es noch einen kleinen Schritt weiter… Rechnet man nun mit diesem Wert weiter, bleibt er bei Double.

($b /2) \*2 🡪 Wo wir wieder beim Ursprung wären… Ist noch immer ein Double. Somit können wir jetzt die als + 1 machen und bekommen nun nicht mehr „11“ sondern „2“:



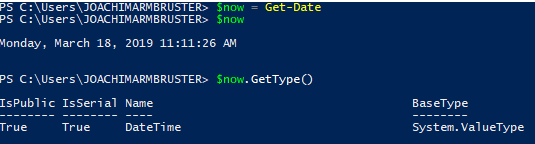
Nicht sonderlich alltagstauglich, aber es zeigt die Probleme, die auftreten können, wenn ausversehen Quotes bei Zahlen nutzt oder sonst irgendwie unvorsichtig ist.

Die Unterschiede zwischen, Int32, Int64, Double, Float, etc. entnehmt ihr bitte den Microsoft Artikeln. Google ist dein Freund.

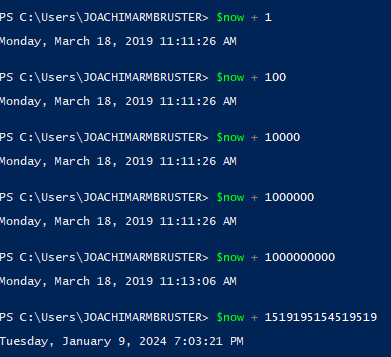
## Datum

PowerShell rechnet zudem super mit Datums-Variablen.

$now = Get-Date



Jetzt habe ich so groß getönt, dass man so toll mit den Zeiten rechnen kann, aber kaum macht man sowas, wird es komisch:



Normale Rechenoperatoren sind also eher unpraktisch. Dafür gibt es spezielle Operatoren die bei Variablen des Typs DateTime verfügbar sind:

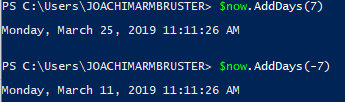
* AddDays
* AddHours
* AddMilliseconds
* AddMinutes
* AddMonths
* AddSeconds
* AddTicks (Keine Ahnung was das ist…)
* AddYears

Möchte man nun also ein paar Tage hinzufügen, kann man das wie folgt:

$now.AddDays(7)

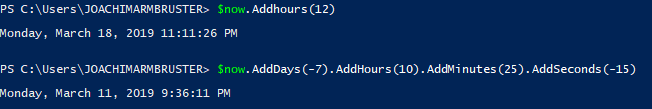
Und möchte man ein paar Tage abziehen, ruft man immer noch denselben Befehl auf, addiert jedoch „-x“:

$now.AddDays(-7)



Dasselbe gilt für alle anderen Operatoren genauso.

Man kann die einzelnen Operatoren auch in einer Zeile hintereinanderschreiben:



Um ein beliebiges Datum als Datums-Variable zu bekommen, ruft man den Befehl Get-Date auf und definiert die einzelnen Parameter:



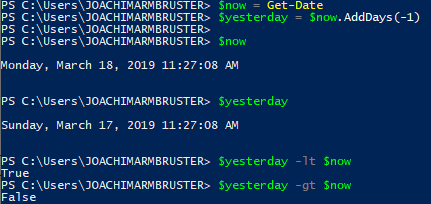
Mehr dazu unter folgendem Artikel:

<https://docs.microsoft.com/en-us/powershell/module/Microsoft.PowerShell.Utility/Get-Date?view=powershell-3.0>

Zwei Variablen vom Typ DateTime lassen sich wunderbar mittel Vergleichsoperatoren vergleichen:

-gt (größer als) / -ge (größergleich)

-lt (kleiner als) / -le (kleinergleich)



Oder kombiniert, um zu sehen ob „jetzt“ zwischen 2 definierten Daten liegt.



Das Ganze lässt sich dann natürlich entsprechend weit ausbauen. Aber es sollte reichen um eine Idee zu liefern.

## Attribute

Ein Schritt in die Objektorientierung ist das Zuweisen von Attributen. Man kann es sich schwer machen, oder man nimmt meine Anleitung…

Nehmen wir eine Variable die einen Menschen beschreiben soll und überlegen uns direkt welche Attribute dieser Mensch haben soll:

* FirstName
* LastName
* DateOfBirth

Wir erstellen die leere Variable und „selecten“ die gewünschten Attribute.

$person = "" | Select FirstName,LastName,DateOfBirth

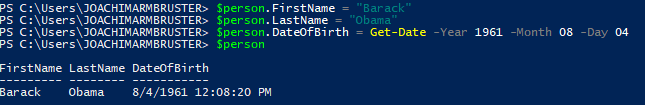
Diese weisen wir dann direkt zu:

$person.FirstName = "Barack"

$person.LastName = "Obama"

$person.DateOfBirth = Get-Date -Year 1961 -Month 08 -Day 04

$person



Den direkten Wert bekommen wir durch direkte Adressierung zurück:

$person.DateOfBirth

Mit einem einzigen Eintrag kommen wir in den meisten Fällen jedoch nicht weit. Darum brauchen wir etwas um mehrere dieser Einträge zu halten. Ein Array.

## Array

### Eindimensionales Arrays

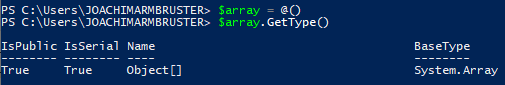
Das Array ist das von mir wohl am häufigsten benutzte Mittel um Daten zu verarbeiten. Mit einem (zweidimensionalen) Array kann man wunderbar Tabellen erstellen.

Ein Array ist ein Objekt, welches eine Sammlung von Werten oder weiteren Objekten darstellt.

Wollen wir zum Beispiel eine Sammlung von Werten erstellen (um z.B. Auswahlmöglichkeiten zu definieren), so kann man hier führ unterschiedlich vorgehen. Ich mache es wie folgt. Zu nächst wird ein leeres Array erstellt:

$array = @()

$array.GetType()



Nun können wir dieses Array befüllen. Das passiert indem man es selbst mit dem gewünschten Element addiert und es direkt abspeichert.

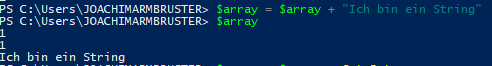
$array = $array + 1

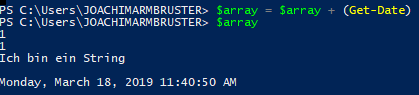


Machen wir dasselbe nochmal, erhalten wir nicht etwas „2“, sondern zweimal das Element „1“



So können wir nun also beliebig viele Elemente in das Array schieben. Wobei es egal ist, welchen Typ das Element hat. In unserem Fall ist jetzt zweimal die Zahl „1“ drin. Wir können jedoch auch Strings, Daten oder weitere Arrays hinzufügen.





Wie kommen wir nun aber an die einzelnen Elemente ran?

Das erfährst du hier:

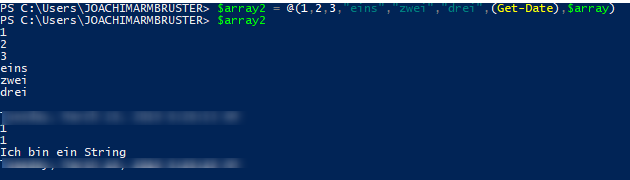
Wie funktionieren Elemente? 🡪 Elemente in Arrays

Um die Schreibweise ein wenig abzukürzen, kann auch einfach folgendes genutzt werden:

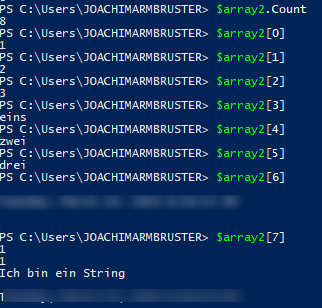
$array += 25

Eine weitere Option ein Array direkt zu füllen sieht wie folgt aus:

$array2 = @(1,2,3,“eins“,“zwei“,“drei“,(Get-Date),$array)

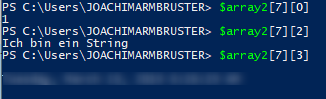


Dabei ist darauf zu achten, dass $array2[-1] (also das letzte Element) selbst ein Array ist.



Wenn man also an die Elemente innerhalb dieses Arrays? Wir gehen an den Index des Index.

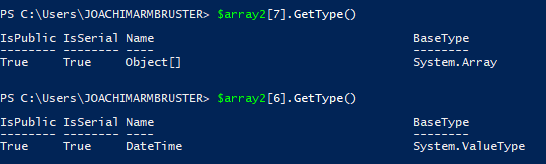
$array2[7][0]



Die 7 könnte in unserem Fall auch -1 sein, da es das letzte Element ist. Sie bezieht sich auf $array2

Die null bezieht sich auf das was sich hinter $array2[7] verbirgt. Also $array (nämlich das erste).

Das zeigt sich auch wenn wir die einzelnen Elemente mit GetType untersuchen:



### Zweidimensionale Arrays

Wenn wir jetzt also mehrere Personen aus dem vorherigen Beispiel in einer Tabelle zusammenfassen wollen, nehmen wir hierfür ein zweidimensionales Array. Es hört sich super schlimm an, ist aber nichts anders als das eindimensionale, nur dass die Elemente selbst weitere Attribute haben. Wir erstellen ein leeres Array names $allPersons

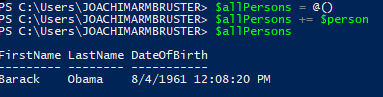
$allPersons = @()

Sofern nicht noch in der Shell, erstellen wir unsere Person wie im Eintrag „Attribute“ zu sehen.

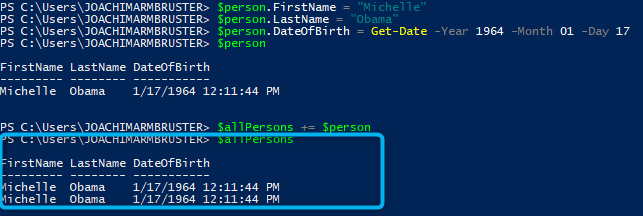
Jetzt können wir unserem Array die Person hinzufügen.

$allPersons += $person

$allPersons



Obama ist jetzt in unserer Liste. Überschreiben wir also die alte Person und füllen sie mit neuen Daten. Danach wird auch diese hinzugefügt. Aber Halt, was ist hier passiert:



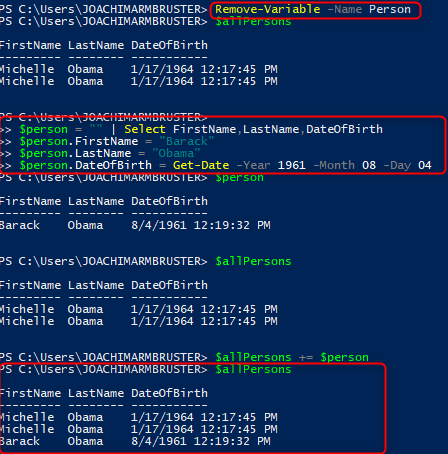
In dem Array wird auf die Variable verwiesen. Darum kann man die Variable nicht wiederverwenden. Nicht in diesem Kontext zumindest. Wenn wir später mit for, foreach und foreach-object arbeiten, wird das funktionieren, da die Variable nur pro Instanz gültig ist.

Erst wenn die Variable nicht mehr existiert, werden ihre Werte „hartgecoded“. Davor verweist der Wert im Array praktisch „live“ noch auf die Werte der Variable.

Remove-Variable -Name Person

$person = "" | Select FirstName,LastName,DateOfBirth

Etc…

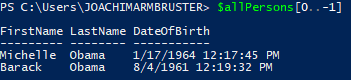


Jetzt habe wir die Michelle zweimal drin. Die soll einmal raus. Hier kenne ich keinen einfachen Trick, nur die Möglichkeit eine Spanne zu definieren. Dies geschieht über den Start-Index, gefolgt von zwei Punkten und dem Schluss-Index.

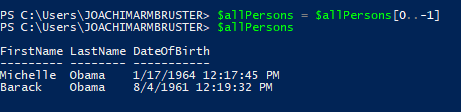
In unserem Fall wollen wir entweder das 0. oder das 1. Element weg haben. Also den Index 0, oder den Index 1. Also wollen wir den Index ab 1 bis Ende behalten

[1..“Ende“] was auch immer das Ende ist, denn es könnten ja auch zig tausende sein, oder mehr… und jedesmal zählen? Nein… Wir haben ja im Kapitel über Elemente und Index gelernt, dass man mittel „-1“ ans Ende kommt. Also

$allPersons[1..-1]



Jetzt gilt es noch das alte Array zu überschreiben und alles ist wie es sein sollte.

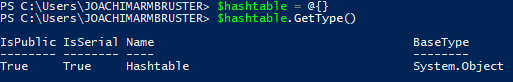


## Hashtables

Hashtables nutze ich in der Tat recht selten. Sie ermöglichen es 1 zu 1 Beziehungen in eine Variable zu schreiben. Im Endeffekt fühlt es sich dann an wie die Attribute die bereits oben beschrieben sind.

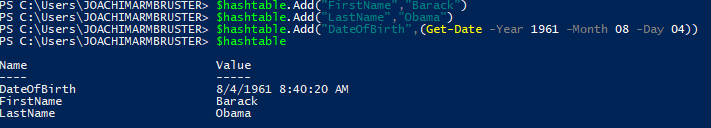
Ein Hashtable wird wie folgt erstellt:

$hashtable = @{}



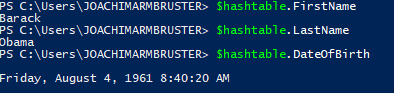
Um einen Hashtable zu füllen, braucht man den .Add Operation:

$hashtable.Add("FirstName","Barack")



Der Aufruf der einzelnen Werte erfolgt dadurch, dass man den Namen des Attributs mit Punkt auruft:

$hashtable.Firstname

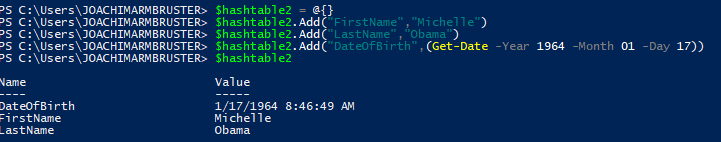


Wo ist aber dann der Unterschied zu den Attributen einer Variablen… Beides sind Objeke, aber der Hashtable ist praktisch ein Karton in dem Informationen drin sind. Die „normale Variable“ ist einfach nur eine Stück Papier auf dem was draufgeschrieben ist. Das Zeigt sich wie folgt, nämlich wenn wir ein Array bauen, die praktisch eine Tabelle werden soll. Z.B für einen CSV-, Datenbank-Export, etc.

Bauen wir uns noch einen zweiten Hashtable:

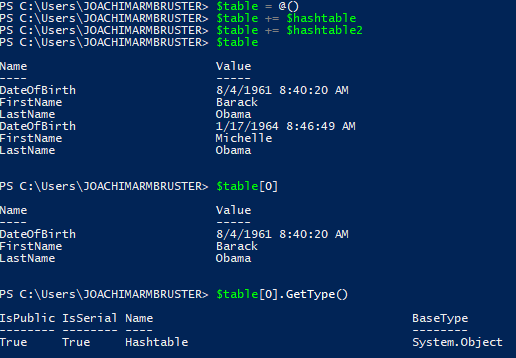
$hashtable2 = @{}

$hashtable2.Add("FirstName","Michelle")  
$hashtable2.Add("LastName","Obama")  
$hashtable2.Add("DateOfBirth",(Get-Date -Year 1964 -Month 01 -Day 17))  
$hashtable2



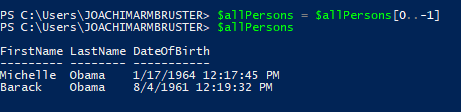
Jetzt ein neues Array als „Tabelle“ und fügen da mal die beiden Hashtables ein:

$table = @()  
$table += $hashtable  
$table += $hashtable2  
$table



Nicht so praktisch… für diesen Anwendungsfall zumindest.

Im Vergleich hierzu nochmal das Array mit den Variablen mit Attributen:



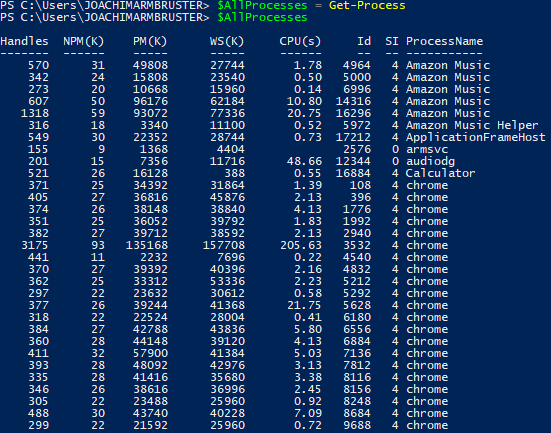
(Wundert euch nicht über die Uhrzeiten, ich habe die Screenshots an unterschiedlichen Tagen und Zeiten erstellt)

# Where-Object

Jetzt wo wir die Arrays als Tabellen nutzen können, müssen wir auch wissen wie wir uns darin zurechtfinden.

Nehmen wir mal ein Beispiel mit mehr Einträgen:

$AllProcesses = Get-Process  
$AllProcesses



Jetzt wollen wir vielleicht nur mal alle Prozesse bei denen „PowerShell“ im Namen ist… Dafür brauchen wir „Where-Object“, oder „Where“, oder „?“. Alles dasselbe, nur Aliasse:

Get-Help Where-Object

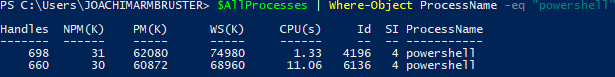
Wir nehmen nun unser Array und „pipen“ seinen Wert in den nächsten Befehl, nämlich Where-Object. Pipen geht mittels „|“:

$AllProcesses | Where-Object ProcessName

Zeigt uns alle Einträge aus dem Array $AllProcesses bei dem das Attribute ProcessName exisitiert… Ok, nicht ganz sinnig. Wir wollen alle PowerShell Prozess. Also brauchen wir Vergleichs-Operatoren. Mehr zu Vergleichs-Operatoren findest du hier:

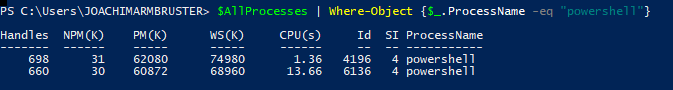
Vergleichs-Operatoren

$AllProcesses | Where-Object ProcessName -eq "powershell"



Aber Achtung! Das ist die Faullenzer-Variante. Korrekt wäre folgender Aufruf:

$AllProcesses | Where-Object {$\_.ProcessName -eq "powershell"}



Hinter das Where-Object gehört eine geschweifte Klammer, da darin Code ausgeführt wird. Da verwirrenste am Anfang wird für die meisten das „$\_.“ sein. Keine Angst, die Erklärung kommt.

Bei der Weitergabe von Objekten, weiß man prinzipiell erstmal nicht was darin vorgeht. Trotzdem muss man in der Lage sein ein jedes einzelne der Elemente in diesem Objekt adressieren und benutzen zu können. Das passiert mittels „$\_“. Mit „|“ leiten wir den gesamten Satz von $AllProcesses zum nächsten Befehl. Nämlich zu „Where-Object“. Dieses Where-Object geht von oben nach unten durch und adressiert in diesem Zuge jedes dieser Elemente einmal. Praktisch $AllProcesses[0] dann $AllProcesses[1] dann $AllProcesses[2], etc. Jetzt könnte man hier mittels for (kommt später dran) selbst aktiv werden, oder man lässt das System das machen indem er jedes dieser Elemente mittels „$\_“ einmal instanziiert. Je nachdem bei welchem Index er gerade ist, ist $\_ entsprechend $AllProcesses[Index]. Danach ist $\_ der nächste Index, usw.

Da wir und für das Attribute „ProcessName“ eines jeden Elements interessieren, fragen wir also nach

$\_.ProcessName

Also das instanziierte Element bei dem entsprechenden (automatisch fortgeführten) Index und dem entsprechenden Attribut.

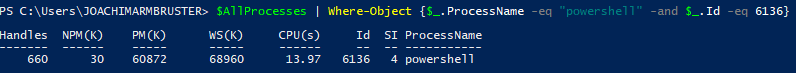
Um das noch weiter eingrenzen zu können, müssen wir logische Operatoren nutzen:

-AND  
-OR

PowerShell ist hier etwas unschön in dem was es akzeptiert:

$AllProcesses | Where-Object {$\_.ProcessName -eq "powershell" -and $\_.Id -eq 6136}

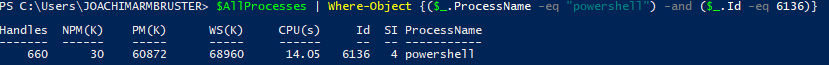
Das ist z.B. ein valider Ausdruck:



* Gib mir alle Prozesse (die vorher in die Variable gepusht wurden)
* Leite diese weiter
* Suche hierin nach
* Attribut ProcessName muss „powershell“ sein
* UND
* Attribut Id muss 6136 sein

Ich versuche diese Filter (denn nichts anderes ist es) optisch so gut wie möglich darzustellen, indem ich pro Vergleich eine runde Klammer setze.

$AllProcesses | Where-Object {($\_.ProcessName -eq "powershell") -and ($\_.Id -eq 6136)}

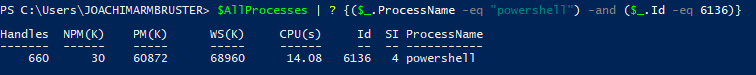


Dasselbe würde ich dann auch für Filter machen, bei denen auch mehrere logische Operatoren vorhanden sind.

Es ist wichtig zu beachten, dass Klammern immer von Innen nach Außen priorisiert werden. Die innerste Klammer wird zuerst aufgelöst, usw.

Der Einfachhalt halber nehme ich stets das „?“ statt dem Where-Object oder dem Where:

$AllProcesses | ? {($\_.ProcessName -eq "powershell") -and ($\_.Id -eq 6136)}



Aber es macht keinen Unterschied.

# Wie funktionieren ForEach und ForEach-Object

An dieser Nuss habe ich lange genagt… learning by doing… und immer mur Foo Bar, Foo Bar, Foo Bar… in jedem Codebeispiel… naja, ich mache es euch leichter.

Nehmen wir mal an, wir haben ein Array mit vielen Elementen darin. Z.B.:

$AllProcesses = Get-Process

Jetzt haben wir im Kapitel WHERE schon gelernt, dass man in solch einer „Liste“ auch mal von oben nach unten durchgehen muss und dabei jedes einzelne Element einmalig adressieren muss um irgendetwas damit zu tun. Genau diese sequentielle Adressierung passiert bei ForEach, bzw. ForEach-Object

Wir haben unsere Variable $AllProcesses. Um jetzt mit jedem dieser Elemente einmal etwas tun zu können, müssen wir dem Kind einen Namen geben. Einen x-beliebigen (außer den reservierten Namen natürlich). Dann geht PowerShell von oben nach unten durch und speichert dieses Element vorübergehend in einer Variablen die eben jeden x-beliebig ausgedachten Namen hat. Damit man diesen benutzen. Beispiel:

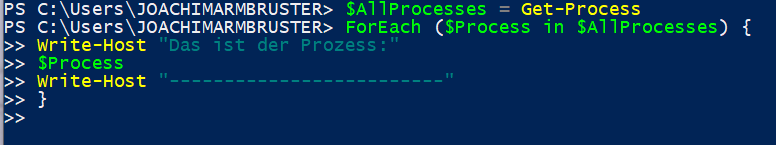
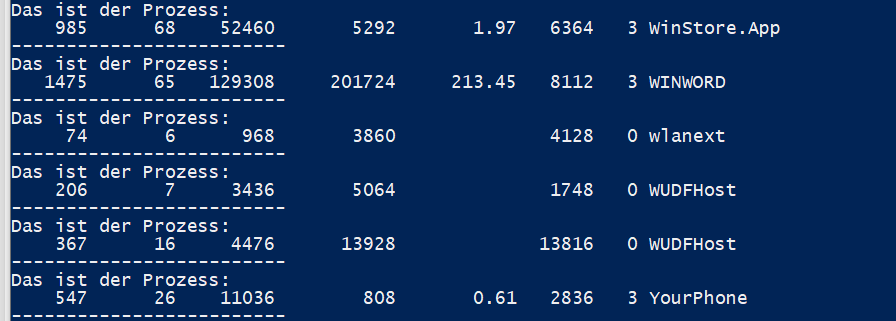
ForEach ($Process in $AllProcesses) {

Write-Host "Das ist der Prozess:"

$Process

Write-Host "-------------------------"

}

Nicht besonders einfallsreich, aber es hilft beim Verstehen. In der Regel würde man mit den einzelnen Werten was anderes machen wollen. Was auch immer dieses andere ist.

ForEach ($Process in $AllProcess) {

Write-Host "Pozess $($Process.Name) belegt $($([math]::round(($Process.PagedSystemMemorySize / (1024 \* 1024)),2))) MB im Pagefile."

}

Jetzt können wir also div. Operationen pro Zeile vornehmen, da wir das Kind beim Namen nennen können.

Hier noch ein ziemlich unsinniges Beispiel:

# Erstelle ein neues Array

$Table = @()

# Hole mir alles in $AllProcesses „wo“ der Name nicht svchost ist.

# Und gehe von darin von oben nach unten durch und adressiere jedes Element als $Process

ForEach ($Process in ($AllProcesses | ? {$\_.Name -ne "svchost"})) {

# Erstelle eine neue Variable mit Attributen

$Line = "" | Select Name,PageMB

# Fülle diese mit den gwünschten Werten

$Line.Name = $Process.Name

$Line.PageMB = [math]::round($Process.PagedSystemMemorySize / (1024 \* 1024),2)

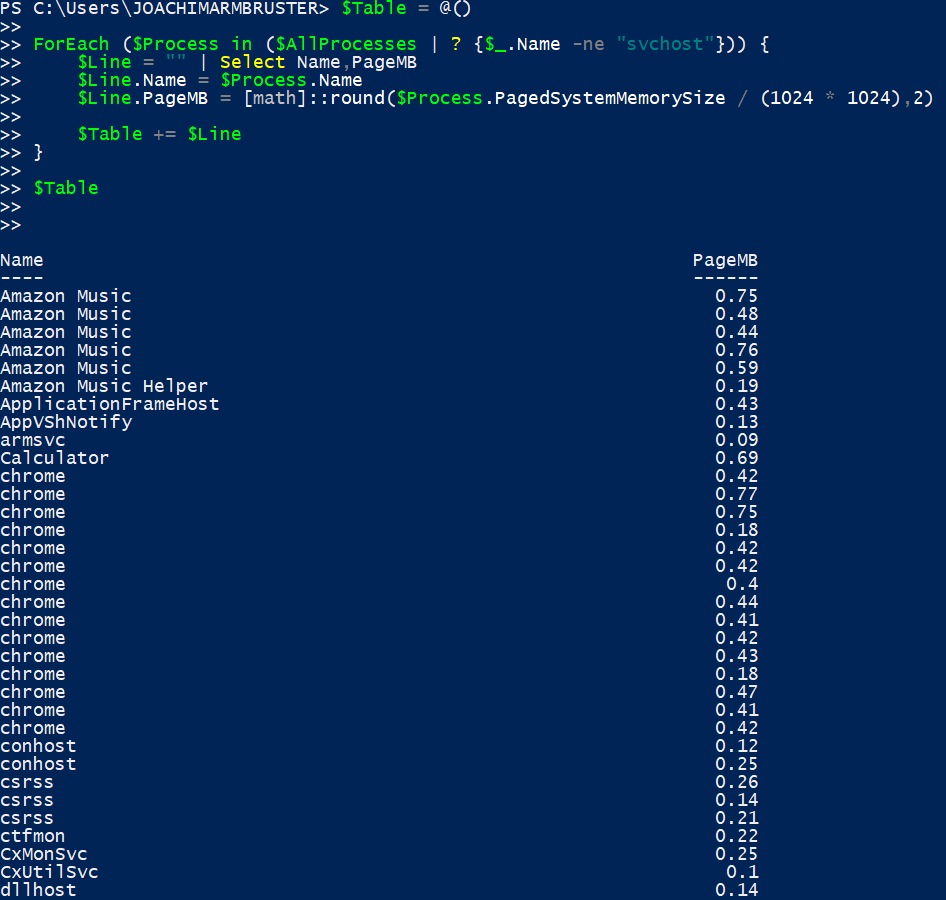
# Füge die Zeile dem Array hinzu

$Table += $Line

}

# Gebe das Array aus

$Table



Manchmal möchte man aber nicht den Umweg über die namentliche Adressierung gehen. Dann kommt ForEach-Object ins Spiel.

Wie bei Where-Object auch, wird ein Object zu ForEach-Object gepiped. Darauf folgt eine geschweifte Klammer. Und jedes Element in diesem Objekt, wir dann mittels $\_ adressiert.

So zum Beispiel folgender Code:

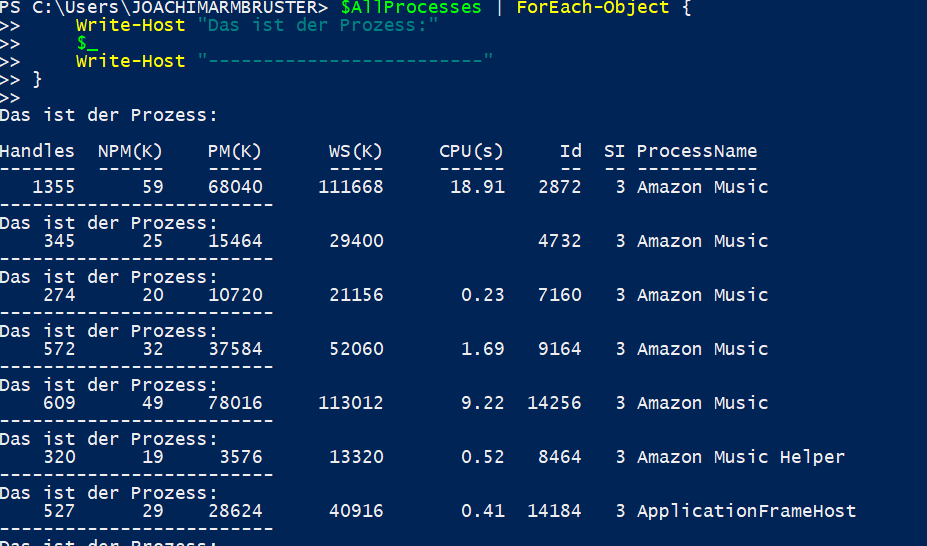
$AllProcesses | ForEach-Object {

Write-Host "Das ist der Prozess:"

$\_

Write-Host "-------------------------"

}



Oder das zweite Beispiel:

$Table = @()

($AllProcesses | ? {$\_.Name -ne "svchost"}) | ForEach-Object {

$Line = "" | Select Name,PageMB

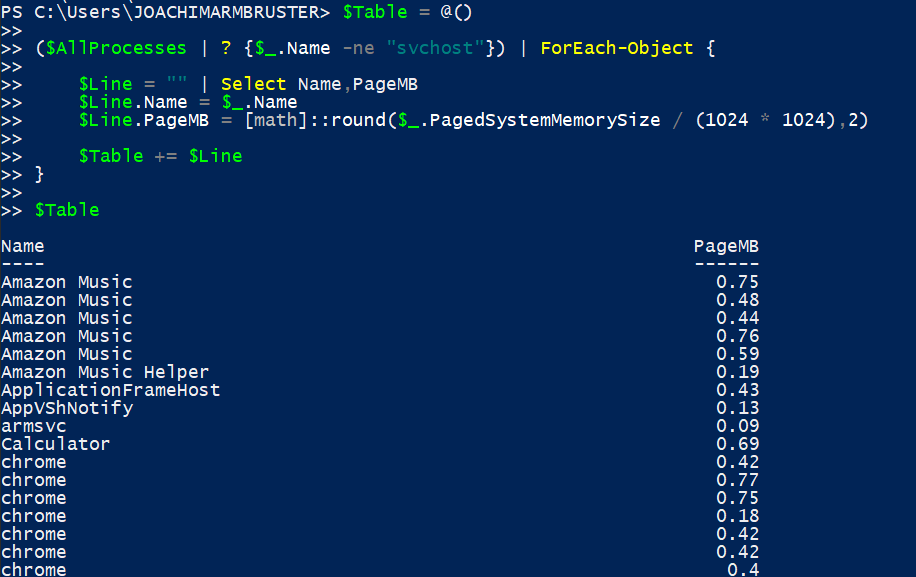
$Line.Name = $\_.Name

$Line.PageMB = [math]::round($\_.PagedSystemMemorySize / (1024 \* 1024),2)

$Table += $Line

}

$Table



Wobei einen keiner zwingen kann nicht doch einen anderen Weg einzuschlagen, je nach Bedarf… Hier habe ich z.B. $\_ der Variablen $Process zugewiesen, wodurch ich es dann doch wie vorher verarbeiten konnte.

$Table = @()

($AllProcesses | ? {$\_.Name -ne "svchost"}) | ForEach-Object {

$Process = $\_

$Line = "" | Select Name,PageMB

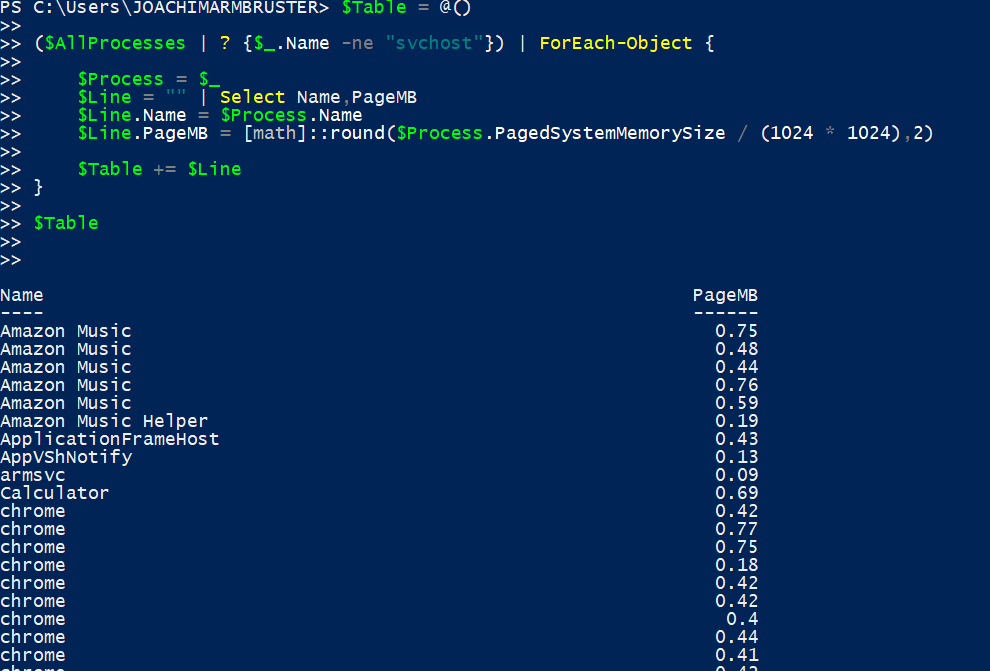
$Line.Name = $Process.Name

$Line.PageMB = [math]::round($Process.PagedSystemMemorySize / (1024 \* 1024),2)

$Table += $Line

}

$Table



# Wie funktionieren Elemente? Adressierung mittels Index.

## Elemente in Strings

$String = "Hallo Welt!"

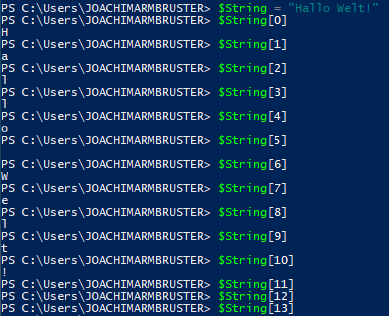
In der IT beginnt (fast) immer alles mit 0, nicht mit 1.

Zeichen H a l l o W e l t !

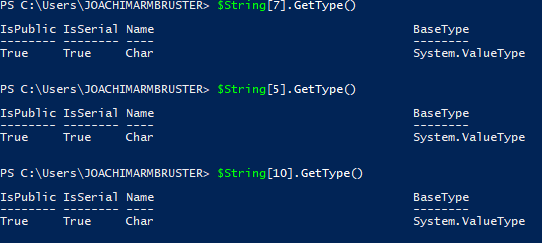
Element 0. 1. 2. 3. 4. 5. 6. 7. 8. 9. 10.

Das Zeichen „H“ ist also das „0-te“ Element des Strings „Hallo Welt!“. „a“ ist das 1., usw.

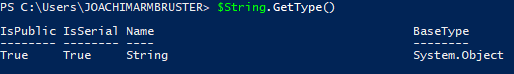
Diese Zahl nennt man den **Index**.



Und für jedes dieser Elemente können wir den Typen erfragen. In einem String ist das recht simpel… Es ist ein Zeichen.

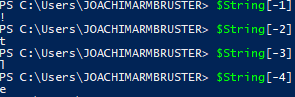


Jetzt wird vielleicht auch klar, warum der String selbst ein Objekt ist, nämlich weil es trotzdem aus (Unter-)Elementen besteht.



Kleine Zusatzinfo:

Wenn ihr das letzte Element eines Objekts haben wollt, kann man rückwärtsgehen, indem man das „-1“-te Element aufruft, etc.



Manchmal möchte man aber auch einfach wissen, wie viele Zeichen in solche einem String sind. Man könnte jetzt denken, dass man den String einfach mittel „count“ zählen kann:

$String.Count

Ergebnis: 1

Bei Strings darf man aber leider nicht zählen, sondern sich die Länge ausgeben lassen.

$String.Length

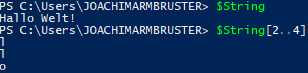
Ergebnis: 11



Count zähl einem also die Anzahl der Objekte in einer Variablen. Und da der String das Objekt selbst ist, ist das Ergebnis 1.

Der Index lässt es auch zu einer Spanne von Elementen anzuzeigen. Das passiert mittels Start-Index, gefolgt von zwei Punkten und dem Schluss-Index. Z.B:

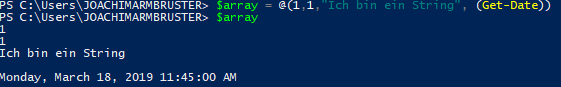
$String[2..4]



## Elemente in Arrays

So wie man in Strings mittels des Index an die einzelnen Positionen kommt, kann man auch an die entsprechenden Elemente in einem Array kommen. Erzeugen wir also ein Array:

$array = @(1,1,“Ich bin ein String“, (Get-Date))

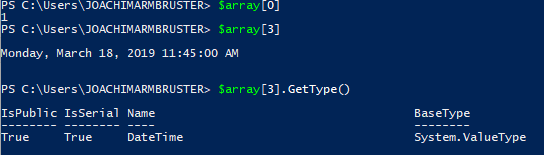


Setzen wir dann die eckige Klammer dahinter und adressieren das Element, bekommen wir dieses einzeln zurück. Diese können wir auch wieder mittels GetType analyisieren:

$array[0]

$array[3]

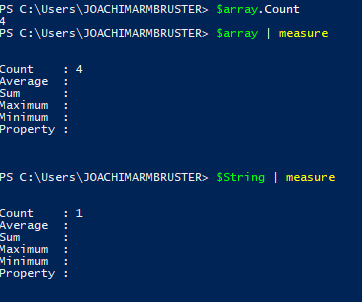
$array[3].GetType()



Auch -1, etc funktioniert hier wie gewohnt.



Bei Arrays funktioniert das Count wie gewünscht



Auch bei Arrays kann mit den Spannen gearbeitet werden.

Häufige Typen

Themen:

Out-GridView mit PassThru

Functions

While 🡪 Break

Datumsformat -> AsString

Variablen in Layern: Console -> Skripts -> Funktion

GUIs -> PoshGui

Operatoren

* Count
* Length
* Substring
* Split
* ToLower
* ToUpper
* ToString

Sort