PowerTips MONTHLY

Part of the PowerShell.com reference library, brought to you by **Dr. Tobias Weltner**

Volume 2 | July 2013

This Month's Topic:

Arrays and Hash Tables

Dr. Tobias Weltner

Table of Contents

1. Using Simple Arrays

Use the comma operator to create simple arrays. Arrays can store any data type. This will create an array with five elements, each with a different data type:

```
$myArray = 'Hello', 12, (Get-Date), $null, $true
$myArray.Count
```

To access array elements, use square brackets and the element index number. The index always starts at 0. To get the first element, type this:

```
$myArray[0]
```

Negative index numbers count backwards so to get the last element, use -1:

```
$myArray[-1]
```

You can also use arrays as index and select more than one element. The next line retrieves the first, the second and the last element:

```
$myArray[0,1,-1]
```

To discard the last two elements, select the remaining elements and reassign them:

```
$myArray = $myArray[0..2]
$myArray.Count
```

To add a new element to an existing array, simply use +=:

```
$myArray += 'new element'
$myArray.Count
```

Finally, to create a strongly-typed array, you should add the type in front of your array definition and append the type with an opening and a closing square bracket. The following array only accepts integer values:

```
[Int[]]$myArray = 1,2,3
$myArray += 12
$myArray += 14.678
$myArray[-1]
$myArray += 'this won't work because it's not convertible to an integer'
```

2. Strongly-Typed Arrays

When you assign a strongly-typed array to a variable, then the variable will contain a strongly-typed array. However, it is very fragile. As soon as you use the operator "+=" to add more values, it falls back to a generic type, and you can happily add whatever value types you want:

```
PS> $array = [Int[]](1,2,3,4,5)
PS> $array.GetType().FullName
System.Int32[]

PS> $array += "foo"
PS> $array.GetType().FullName
System.Object[]
```

This is because the complete array will be copied into a new one which is by default a generic array. You can work around this by assigning a type to the variable that stores the array:

```
PS> [Int[]]$array = 1,2,3,4,5
PS> $array.GetType().FullName
System.Int32[]

PS> $array += 6
PS> $array.GetType().FullName
System.Int32[]
```

Now, the array will remain an array of integers even though you used the += operator to add elements to the array.

3. Using Switch Statement with Arrays

Did you know that the Switch statement can accept arrays? Use this sample to translate numbers into words:

```
$myArray = 1,5,4,2,3,5,2,5

Switch ( $myArray ) {
  1 { 'one' }
  2 { 'two' }
  3 { 'three' }
  4 { 'four' }
  5 { 'five' }
}
```

Author Bio

Tobias Weltner is a long-term Microsoft PowerShell MVP, located in Germany. Weltner offers entry-level and advanced PowerShell classes throughout Europe, targeting mid- to large-sized enterprises. He just organized the first German PowerShell Community conference which was a great success and will be repeated next year (more on www.pscommunity.de).. His latest 950-page "PowerShell 3.0 Workshop" was recently released by Microsoft Press.

To find out more about public and in-house training, get in touch with him at tobias.weltner@email.de.

4. Check Arrays Using Wildcards

You may know the -contains operator. Try using it to check whether an array contains a specific element. The following example reads all files in your Windows folder and picks the file names in the property Name, so $names will contain a list of file names.

Next, -contains can check whether the list contains a specific name (like explorer.exe):

```
PS> $names = Get-ChildItem -Path $env:windir | Select-Object -ExpandProperty Name
PS> $names -contains ‘explorer.exe’
True
```

Unfortunately, -contains does not support wildcards, though:

```
PS> $names -contains ‘explorer*’
False
```

Simply use -like instead of –contains! It works on arrays, too:

```
PS> $names -like ‘explorer*’
explorer.exe
```

As you can see, the operator -contains returns $true or $false, but -like returns the actual matches. Note also that in PowerShell 3.0 there is a new operator called -in which works like -contains but with reverse operands.

Here are some simple examples:

```
PS> ‘Peter’, ‘Mary’, ‘Martin’ -contains ‘Mary’
True

PS> ‘Peter’, ‘Mary’, ‘Martin’ -contains ‘Ma*’
False

PS> ‘Mary’ -in ‘Peter’, ‘Mary’, ‘Martin’
True

PS> ‘Peter’, ‘Mary’, ‘Martin’ -like ‘Ma*’
Mary
Martin

PS> @(‘Peter’, ‘Mary’, ‘Martin’ -like ‘Ma*’).Count -gt 0
True
```

5. Creating Byte Arrays

You can try this to create a new empty byte array with 100 bytes:

```
$byteArray = New-Object Byte[] 100
```

Try this if you need to create a byte array with a default value other than 0:

```
$byteArray = [Byte[]] (,0xFF * 100)
```

The trick here is to use the comma, as a unary operator, to create a simple array with just one element, and then multiply this array as often as you want. When you multiply an array, it stays an array and just becomes larger.
The next line would create an array with the numbers 1 to 7 repeated 5 times:

```
$Array = (1..7) * 5
```

Here, no comma is needed because (1..7) is already an array.

6. Performance Optimization for Arrays

Arrays are slow when you need to add new elements or delete existing elements. That's because arrays have no way of inserting or removing elements. Instead, the entire array is copied into a new array. That's an expensive process.

You can convert array to ArrayLists, though. ArrayLists act like more sophisticated arrays that have methods like RemoveAt() and InsertAt(). This example will create a simple array with numbers from 1 to 10, then convert it into an ArrayList.

Now, the ArrayList can easily manipulate its elements—add new ones at the end or insert them at arbitrary positions. This is not just a lot more convenient. It is also a lot faster.

```
$array = 1..10
[System.Collections.ArrayList]$arraylist = $array
$arraylist.RemoveAt(4)
$null = $arraylist.Add(11)
$arraylist.Insert(0,'added at the beginning')
$arraylist
```

7. Reversing Array

To reverse the order of elements in an array, the most efficient way is to use the [Array] type and its static method Reverse():

```
$a = 1,2,3,4
[array]::Reverse($a)
$a
```

Technical Editor Bio

Aleksandar Nikolic, Microsoft MVP for Windows PowerShell, a frequent speaker at the conferences (Microsoft Sinergija, PowerShell Deep Dive, NYC Techstravaganza, KulenDayz, PowerShell Summit) and the cofounder and editor of the PowerShell Magazine (http://powershellmagazine.com). He is also available for one-on-one online PowerShell trainings. You can find him on Twitter: https://twitter.com/alexandair

8. Using Jagged Arrays

Have a look at how you can create "jagged arrays". Here's a jagged array which really is a nested array:

```
PS> $array = 1,2,3,(1,('a','b'),3),5
```

It has five elements, and the third element (start counting by 0) is an array itself (1,('a','b'),3) which in turn holds an another array ('a','b') in its own first element .

Check out its behavior:

```
PS> $array[2]
3

PS> $array[3]
1
a
b
3

PS> $array[3][0]
1

PS> $array[3][1]
a
b

PS> $array[3][1][0]
a

PS> $array[3][1][1]
b
```

9. Creating Multi-Dimensional Arrays

By default, PowerShell uses jagged arrays. To create conventional symmetric arrays, here's how:

```
$arrayMD = New-Object 'Int32[,]' 2,2
```

This creates a two-dimensional array of Integer numbers. Note that each dimension starts with 0, so this array goes from $array[0,0] to $array[1,1].

```
PS> $arrayMD
0
0
0
0

PS> $arrayMD[1,1]
0

PS> $arrayMD[1,1]=100

PS> $arrayMD[1,0]=200

PS> $arrayMD
0
0
200
100
```

If you want multi-dimensional arrays but no specific type, use this:

```
$array = New-Object 'object[,]' 10,20
```

This will create a 10x20 array with no specific type attached to it.

10. Returning Array in One Chunk (and Preserving Type)

When a function returns an array, the PowerShell pipeline automatically “unrolls” the array and processes the array elements individually. That’s default behavior. Have a look:

```
function Test {
  $someresult = 1..10

  return $someresult
 }


Test | ForEach-Object { “Receiving $_” }
```

As you’ll see, you get ten output texts, all starting with “Receiving”, indicating that ForEach-Object received ten individual numbers rather than one array.

If you want the array to be treated as one chunk instead, add a single comma:

```
  return $someresult
```

This comma actually creates a new array with just one element, which happens to be the other array. The pipeline again “unrolls” the array it receives, but since the array is now nested inside another array, the returned array is preserved and only processed once by the following ForEach-Object cmdlet.

You can use this trick if you want to preserve the type of an array. By default, the array type is never preserved because the pipeline unrolls the array and PowerShell then recreates a generic array. So although the function in the next example explicitly creates an ArrayList, this ArrayList is implicitly converted to a basic array:

```
function test {
  $al = [System.Collections.ArrayList](1..10)
  $al
}
```

```
PS> (Test).GetType().FullName
System.Object[]
```

If you use the comma, the original ArrayList is not unrolled by the pipeline and maintains its type:

```
function test {
  $al=[System.Collections.ArrayList](1..10)
  ,$al
}
```

```
PS> (Test).GetType().FullName
System.Collections.ArrayList
```

11. Creating Range of Letters

PowerShell can easily provide a range of numbers. You can also create ranges of letters. Simply use ASCII codes and convert them to characters. This will create lists of letters or disk drives:

```
65..90 | ForEach-Object { [char]$_ }
65..90 | ForEach-Object { “$([char]$_):” }
```

If you cast the numbers to a character array directly, then you do not even need a loop. This example will create lower-case letters and stores them in $letters:

```
$letters = [char[]](97..122)
$letters
```

12. Converting Cmdlet Results into Arrays

Whenever you call a function or cmdlet, PowerShell uses a built-in mechanism to handle results:

- If no results are returned, PowerShell returns nothing
- If exactly one result is returned, the result is handed to the caller
- If more than one result is returned, all results are wrapped into an array, and the array is returned.

So at design time, you never really know what you are getting at the run-time.
For example, Get-ChildItem (alias: Dir) returns nothing, a file object, a folder object or an array, depending on the folder you are listing:

```
PS> (Get-ChildItem C:\nonexistent -ea SilentlyContinue).GetType().FullName
You cannot call a method on a null-valued expression.
At line:1 char:1
+ (Get-ChildItem C:\nonexistent -ea SilentlyContinue).GetType().FullName
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : InvalidOperation: (:) [], RuntimeException
    + FullyQualifiedErrorId : InvokeMethodOnNull


PS> (Get-ChildItem $env:windir\explorer.exe -ea SilentlyContinue).GetType().FullName
System.IO.FileInfo

PS> (Get-ChildItem $env:windir -ea SilentlyContinue).GetType().FullName
System.Object[]
```

To always receive arrays, simply wrap the command into @():

```
PS> @( Get-ChildItem C:\nonexistent -ea SilentlyContinue).GetType().FullName
System.Object[]

PS> @( Get-ChildItem $env:windir\explorer.exe -ea SilentlyContinue).GetType().FullName
System.Object[]

PS> @( Get-ChildItem $env:windir -ea SilentlyContinue).GetType().FullName
System.Object[]

PS> @( Get-ChildItem C:\nonexistent -ea SilentlyContinue).Count
0

PS> @( Get-ChildItem $env:windir\explorer.exe -ea SilentlyContinue).Count
1

PS> @( Get-ChildItem $env:windir -ea SilentlyContinue).Count
105
```

The example thus illustrates how you can “normalize” command output and safely assume a resulting array.

Because this technique isn’t widely known, in PowerShell 3.0, each object (including NULL-values) supports the use of Count. This is just a workaround, though. It’s still more robust to make sure programmatically that commands return standard arrays by using @() wrapper.

You will still receive red error messages when you try and list a folder that does not exist. To get rid of these error messages, set the ErrorAction parameter to SilentlyContinue:

@(Dir C:\nonexistent -ea SilentlyContinue).Count

13. Arrays of Strings

In PowerShell, you can multiply strings: the string is repeated which can be useful for creating separators:

```
PS> '-' * 50
--------------------------------------------------
```

This works for words, too:

```
PS> 'localhost' * 10
localhostlocalhostlocalhostlocalhostlocalhostlocalhostlocalhostlocalhostlocalhostlocalhost
```

You can create a text array by converting the text to an array by first wrapping it into @() before multiplying it:

```
PS> @('localhost') * 10
localhost
localhost
localhost
localhost
localhost
localhost
localhost
localhost
localhost
localhost
```

Here, the array element is multiplied, and you can use this technique to create all kinds of predefined arrays. This will create an array with 20 elements, all initialized to an integer value of 0:

```
PS> @(0) * 20
0
0
0
0

(...)
```

As a shortcut, you can also use a comma to create the initial array:

```
PS> ,0 * 20
0
0
0
0

(...)
```

14. Using Hash Tables

Hash tables are a great way to organize data. A hash table stores key-value pairs.

To create a new hash table variable, try this:

```
$person = @{}
```

You can then add new key-value pairs by simply using the dot notation:

```
$person.Age = 23
$person.Name = 'Tobias'
$person.Status = 'Online'
$person.Name
$person
```

Optionally, you can use array syntax to access hash table elements:

```
$person['Age']
```

You can even use other variables to access its keys:

```
$info = 'Age'
$person.$info
```

15. Sorting Hash Tables

A hash table stores key-value pairs and you normally cannot sort its content. Let's define a hash table first to examine this:

```
$hash = @{Name='Tobias'; Age=66; Status='Online'}
```

When you output its content, you get two columns, key and value, and when you pipe the result into Sort-Object, the result is not sorted:

```
PS> $hash | Sort-Object Name

Name                           Value
----                           -----
Status                         Online
Name                           Tobias
Age                            66
```

To sort a hash table, you need to transform its content, by using GetEnumerator() method, into individual objects that can be sorted:

```
PS> $hash.GetEnumerator() | Sort-Object Name

Name                           Value
----                           -----
Age                            66
Name                           Tobias
Status                         Online
```

16. Using Ordered Hash Tables

In PowerShell 3.0, you can use a special type of a hash table that preserves key order. Keys in such hash table always keep the order in which they were defined.
This is how a regular hash table works:

```
PS> $hash = @{Name='Tobias'; Age=66; Status='Online'}

PS> $hash

Name                           Value
----                           -----
Status                         Online
Name                           Tobias
Age                            66
```

And this is how an ordered hash table works (requires PowerShell 3.0):

```
PS> $hash = [Ordered]@{Name='Tobias'; Age=66; Status='Online'}

PS> $hash

Name                           Value
----                           -----
Name                           Tobias
Age                            66
Status                         Online
```

As you see, the keys are not sorted, they are ordered. They appear exactly in the order they were initially defined.
This can be very useful when you use hash table to define objects as shown elsewhere in this document.

17. Converting Hash Tables to Objects

The key-value pairs stored in a hash table can be used to create new objects. Each key is then turned into a NoteProperty, and each value becomes the property value.

```
PS> $hash = @{Name='Tobias'; Age=66; Status='Online'}

PS> $hash

Name                           Value
----                           -----
Status                         Online
Name                           Tobias
Age                            66


PS> $object = New-Object PSObject -Property $hash

PS> $object

Status          Name           Age
------          ----           ---
Online          Tobias         66
```

Note that you have no control over the order of properties. This is because hash table by default does not preserve key order.

To control the order of object properties, either add a Select-Object statement like this:

```
PS> $object

Status                    Name                                              Age
------                    ----                                              ---
Online                    Tobias                                             66


PS> $object | Select-Object -Property Name, Status, Age

Name                      Status                                            Age
----                      ------                                            ---
Tobias                    Online                                             66
```

Or, in PowerShell 3.0, use ordered hash tables:

```
PS> $hash = [Ordered]@{Name='Tobias'; Status='Online'; Age=66}

PS> $object = New-Object PSObject -Property $hash

PS> $object

Name                      Status                                            Age
----                      ------                                            ---
Tobias                    Online                                             66
```

18. Using Hash Tables to Translate Numeric Values to Clear Text

There are situations in which you would like to convert a numeric return value to a meaningful text message. Hash tables can easily do that. First, define a hash table with the numeric values and their corresponding text messages:

```
$translate = @{
1 = “One”
2 = “Two”
3 = “Three”
}
```

Next, you can easily turn numbers into associated meaningful text:

```
PS> $translate[1]
One

PS> $translate[3]
Three

PS> $id = 2

PS> $translate[$id]
Two

PS>
```

19. Using Hash Tables for Calculated Properties

Hash tables can generate “calculated” object properties. All you need to do is store two pieces of information: a “Name,” which serves as new property name, and an “Expression,” which is the code used to calculate the column.

Here is a hash table that defines a property called “Age” which calculates the age of files and folders in days. Note the use of $_ inside of the expression script block: this variable represents the object that gets the added property:

```
$age = @{
 Name='Age'
 Expression={ (New-TimeSpan $_.LastWriteTime).Days }
}
```

Now check out what happens when you submit the hash table to Select-Object and apply it to files and folders:

```
PS> Get-ChildItem $env:windir | Select-Object Name, $age, Length -First 4


Name                                              Age Length
----                                              --- ------
addins                                            504
AppCompat                                         504
AppPatch                                           11
assembly                                            4
```

```
PS> Get-ChildItem $env:windir | Select-Object Name, $age, Length | Sort-Object -Property Age
-Descending


Name                                              Age                     Length
----                                              ---                     ------
twunk_16.exe                                     1447                      49680
system.ini                                       1447                        219
(…)
regedit.exe                                      1414                     427008
notepad.exe                                      1414                     193536


L2Schemas                                        1414
twain_32.dll                                      919                      51200
bfsvc.exe                                         919                      71168
explorer.exe                                      823                    2871808
(…)
winsxs                                             11
assembly                                            4
SysWOW64                                            4
Microsoft.NET                                       4
System32                                            2
inf                                                 2
setupact.log                                        2                      51585
Temp                                                0
WindowsUpdate.log                                   0                    1558682
bootstat.dat                                        0                      67584


PS> Get-ChildItem $env:windir | Select-Object Name, $age, Length | Where-Object { $_.Age -lt 5 }


Name                                              Age Length
----                                              --- ------
assembly                                            4
inf                                                 2
Microsoft.NET                                       4
System32                                            2
SysWOW64                                            4
Temp                                                0
bootstat.dat                                        0 67584
setupact.log                                        2 51585
WindowsUpdate.log                                   0 1558682
```

As you see, "Age" has become a true object property that you now can use for sorting or finding only the newest files within a time frame.

20. Remove Keys from Hash Tables

Once you create a hash table, it is easy to add new key-value pairs like this:

```
$myHash = @{}
$myHash.Name = 'Tobias'
$myHash.ID = 12
$myHash.Location = 'Hannover'
$myHash
```

Use its Remove() method to remove a key from a hash table:

```
$myHash.Remove('Location')
$myHash
```

21. Case-Sensitive Hash Tables

PowerShell hash tables are, by default, not case sensitive:

```
PS> $hash = @{}
PS> $hash.Key = 1
PS> $hash.keY = 2
PS> $hash.KEY

2
```

Should you need case-sensitive keys, and then create the hash table this way:

```
PS> $hash = New-Object System.Collections.Hashtable
PS> $hash.Key = 1
PS> $hash.keY = 2
PS> $hash[‘Key’]
1
PS> $hash[‘keY’]
2
```