**Work with Array**

By default, PowerShell assumes that the elements of an array are of the type *variant*. This means that you can mix different data types—that is, combine numerical values, dates, or strings in a single variable. However, if you want to restrict an array to a certain data type, you can declare the array in the following way:

## Arrays

For those that have never worked with arrays here’s a great way to understand them:  If a variable is a piece of paper then the stack of papers is an array.  It’s a list of variables or objects, and every programming/scripting language has ways to store these variables or objects linearly so you can access them later via a number of different methods.

So let’s look at how we can create an array of string objects in powershell:

**$array = @("test1", "test2", "test3")**

$myArray = 64,"Hello",3.5,"World", "Demo"

Will have the automatic index numbers:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
| 64 | Hello | 3.5 | World | Demo |

**$array**

$array.GetType()

|  |
| --- |
| $Fruits.Add("Kiwi") |

|  |  |
| --- | --- |
| 2 | $Fruits.Remove("Apple") |

|  |  |
| --- | --- |
| 3 | $Fruits.IsFixedSize |

You can also add an element to the end of an array:

**$array = @("test1", "test2", "test3")**

**$array += "test4"**

**$array**

You can also add arrays together:

**$array = @("test1", "test2", "test3")**

**$array2 = @("test4", "test5")**

**$array = $array + $array2**

**$array**

You can access an element of an array if you know the index number of the element you want.  Arrays are indexed by integers starting with 0.  This can be seen with the following code:

**$array = @("test1", "test2", "test3")**

**"First array value: " + $array[0]**

**"Second array value: " + $array[1]**

**"Third array value: " + $array[2]**

You can use that element as if it’s a regular variable at this point.  Since our array is an array of strings we can use string functions as if this was a regular variable set to that string value when we call the element of this array.  e.g.:

**$array = @("test1", "test2", "test3")**

**$array[1].ToUpper()**

## For Loops

Arrays can also be accessed linearly through the help of for loops.  A for loop generally has 3 bits of information in their declaration:

1. Initialization
2. Condition to continue the loop
3. A repeating occurrence – This is generally used to bring your loop closer to not meeting the condition for you loop so that the loop will eventually end.

|  |  |
| --- | --- |
| 1 | [int[]] $e = 1,2,3,4,5,6,7,8,9 |

Notice that a mere declaration is insufficient. You also have to assign values to the variable. If the data types don’t match, PowerShell will throw an error.

An elegant alternative for assigning consecutive numerical values is the use of the range operator, which allows you to simplify the above command:

|  |  |
| --- | --- |
| 1 | [int[]] $e = 1 .. 9 |

Assign values to an array

The first example demonstrates how to explicitly assign values to an array using the so-called simplified syntax. An alternative would be:

|  |  |
| --- | --- |
| 1 | $colors = @("black","white","yellow","blue") |

Under normal circumstances, you would want to avoid the additional effort of this notation. However, it helps you understand why you can create an empty array in PowerShell with this command:

|  |  |
| --- | --- |
| 1 | $colors = @() |

In many cases, you won’t assign values manually to an array. Instead, you will want to store the output of a cmdlet in a variable. If the output is an array, it can be tested like this:

|  |  |
| --- | --- |
| 1 | (Get-Process) -is [array] |

If you only want to store specific properties in the variables, and not the entire output of the command, you can filter the properties with the *Select-Object*:

|  |  |
| --- | --- |
| 1 | $mac = Get-NetAdapter | Select MacAddress |

This command would save all MAC addresses of the installed NICs in the variable $mac.

Add elements to an array

If you later want to add elements to an array, you don’t have to call a method because the operator “+” is used for this purpose.

|  |  |
| --- | --- |
| 1 | $colors = $colors + "orange" |

Or, if you want to use the combined assignment operator, you can rewrite the above example like this:

|  |  |
| --- | --- |
| 1 | $colors += "orange" |

You can also combine two arrays this way. For instance, if you have another array $pattern with the values “dotted,” “checked,” and “lined,” you could concatenate it to $colors with this command:

|  |  |
| --- | --- |
| 1 | $colors += $pattern |

Create a Multi-dimensional array:

$myMultiArray = @(

(1,2,3),

(40,50,60)

)

Add values to an Array.

This is done using the += operator

Append an extra element with a value of 'India' to the $countries array:  
$countries += 'India'

Adding items to a large array can be quite slow, a PowerShell array variable is immutable - meaning that in the background it creates a whole new array that includes the new value and then discards the old array.   
A faster alternative to use a .Net ArrayList:

$countries = New-Object System.Collections.ArrayList  
$countries.Add('India') > $null  
$countries.Add('Montenegro') > $null

Retrieve items from an Array

To retrieve an element, specify its number, PowerShell numbers the array elements starting at 0.

Return all the elements in an array:

$myArray

Return the first element in an array:

$myArray[0]

Return the seventh element in an array:

$myArray[6]

Return the 5th element through to the 10th element in an array:

$myArray[4..9]

Return the last element in an array:

$myArray[-1]

Return the first element from the first row in a multi-dimensional array:

$myMultiArray[0][0]

You can also combine named elements with a range, separating them with a +  
so $myArray[1,2+4..9] will return the elements at indices 1 and 2 and then 4 through 9 inclusive.

Return the length of an array (how many items are in the array):

$myArray.length

Loop through the elements in an array:

foreach ($element in $myArray) {$element}

In PowerShell 4.0+ arrays have the methods .Where() and .Foreach() a faster alternative to a traditional pipeline at the expense of a higher memory consumption:  
@(Get-Service).Where**(**{$\_.Status -eq 'stopped'}**)**  
or omitting the parenthesis:   
@(Get-Service).Where{$\_.Status -eq 'stopped'}

Comparisons -eq - gt -lt etc

With an array, comparison operators will work as a filter returning all the values which match.

PS C:\> $a = 1,2,3,2  
PS C:\> $a -eq 2  
2  
2

Care must be taken with comparing NULLs, because this is a filter you must place the $null on the left side of the comparison.

PS C:\> $a -eq $null # fails: returns $null  
PS C:\> $null -eq $a # works: returns $false

Data Types

When you create an array without specifying a datatype, PowerShell will create the array as an object array.  
To determine the data type of an array:

$myArray.gettype()

PowerShell is not case-sensitive so $myArray is treated the same as $Myarray

If you pipe an array to Get-Member, it will display information about the objects *in*the array. If you use Get-Member -InputObject, that will display information about the array itself:

get-member -inputobject $myArray

Set values

To change values in an array after it has been created, use the assignment operator (=) to specify a new value.

$myArray[4]=64

Alternatively use the SetValue method:

$myArray.SetValue(64,4)

$myMultiArray[2][4]=128

Add one array to another.  
This creates a new array containing all the values from the first two arrays:

$march=@(2, 3, 4, 5, 6)  
$april=@(7, 8, 9, 10, 11, 12)  
$all = $march + $april

Delete an Array (by deleting the array variable)

Remove-Item variable:monthly\_sales

Array with Loop

This is best seen by looking at a simple loop.  The following will initialize by setting $i to 1.  It will then test that $i is less than 6 and increase $i by one on each pass of the loop:

**for ($i=1;$i -lt 6; $i++) {**

**"This is line number " + $i**

**}**

In the above example we use the -lt comparison operator to test that the value of $i is less than 6.  There are many other comparison operators available like -gt (greater than), -le (less than or equal to), -ge (greater than or equal to), -eq (equal), and -ne (not equal).  To learn about the comparison operators available to you can use the following command:

**help about\_comparison\_operators**

## For Loops with Arrays

It’s easy to see how you can apply a loop to an array to iterate through each element of the array in order if only there was a way to test for how many elements are in the array.  We can do this by using a member of the array object called Length.  The thing to note here is that you want to initialize your index variable with 0 since that is the first element in your array, but you want to test that your index is less than the array length since the last element of the array will be one less than the number of elements in the array.   Here is the process in practice:

**$array = @("test1", "test2", "test3")**

**for ($i=0; $i -lt $array.length; $i++) {**

**$array[$i]**

**}**

An even faster way to do a for loop is by using a special foreach loop.  The foreach loop will set a variable to each item in a list with a much simpler construct:

**$array = @("test1", "test2", "test3")**

**foreach ($element in $array) {**

**$element**

**}**