# Windows PowerShell 4.0 For the IT Professional - Part 1

## Module 13: Arrays

Student Lab Manual

Version 2.0

## Conditions and Terms of Use

**Microsoft Confidential - For Internal Use Only**

# Contents

# Lab 13: Arrays

## Introduction

An array is a data structure designed to store a collection of objects. The objects can be the same type or different types.

## Objectives

After completing this lab, you will be able to:

- Create arrays
- Access arrays
- Sort arrays
- Modify arrays

## Prerequisites

Start all VMs provided for the workshop labs.

Logon to WIN8-WS as:

Username: **Contoso\Administrator**

Password: **PowerShell4**

## Estimated time to complete this lab

45 minutes

**NOTE:** These exercises use many Windows PowerShell commands. You can type these commands into the Windows PowerShell Integrated Scripting Environment (ISE) or the Windows PowerShell console. For some exercises, you can load pre-typed lab files into the Windows PowerShell ISE, allowing you to select and execute individual commands. Each lab has its own folder under **C:\PShell\Labs** on **WIN8-WS**.

Some exercises in this workshop may require running the Windows PowerShell console or ISE as an elevated user (Run as Administrator).

We recommend that you connect to the virtual machines (VMs) for these labs through Remote Desktop rather than connecting through the Hyper-V console. This allows you to use copy and paste between VMs and the host machine. If you are using the online hosted labs, then you are already using Remote Desktop.

# Exercise 13.1: Array Introduction

### Introduction

To create and initialize an array, assign multiple values to a variable. Delimit values stored in the array with a comma. Separate the array from the variable name by the assignment operator (=).

### Objectives

After completing this exercise, you will be able to:

- Create arrays using the comma operator
- Access array elements using an index

## Task 13.1.1: Creating Arrays

1. Windows PowerShell uses arrays extensively. When Cmdlets return more than one object, they are automatically stored in an array object.
   Right-click the PowerShell icon on the taskbar and select the "Run ISE as Administrator" option. This will launch the Windows PowerShell Integrated Scripting Environment. Type the examples below in the command window.

```
$processes = Get-Process
```

2. Since arrays themselves are objects, they also have members (properties and methods).

```
$processes.Count
$processes.GetUpperBound(0)
$processes.GetType()
```

3. Create arrays manually. Any collection of objects separated by a comma are stored in an automatically created array.

```
$users = "Johan","Chris","Gary"
$users.GetType()
$users
$users.Count
```

4. Create empty array objects by using the array sub-expression operator ( @() ).

```
$emptyArray = @()
```

5. Use the array object's Count() method to return the number of items it contains.

```
$emptyArray.Count
```

6. Add a new item to the empty array using the compound addition operator.

```
$emptyArray += "PC"
$emptyArray += "Phone"
```

```
$emptyArray += “Tablet”
$emptyArray += “Laptop”
```

7. Display the items and size of the array.

```
$emptyArray
$emptyArray.Count
```

## Task 13.1.2: Accessing Array Elements

**NOTE:** This lab depends on Active Directory users created in a previous exercise. To ensure they are present, you can run the following script in the lab folder. Ignore any errors.

C:\PShell\Labs\Lab_13\CreateUsers.ps1

1. In the script pane execute the following command to populate a new array with account names from Active Directory user objects.

```
$aryUsers = Get-ADUser -SearchBase "OU=Personnel,DC=Contoso,DC=com" -Filter * |
Select-Object -ExpandProperty SamAccountName
$aryUsers.Count
```

2. Use a zero-based index enclosed within square brackets to access each item in the array.

```
$aryUsers[0]
```

3. Access the eleventh item in the array

```
$aryUsers[10]
```

4. Negative numbers count from the end of the array. -1 represents the last element. Experiment using other negative values.

```
$aryUsers[-1]
```

5. The range operator can display a subset of values.

```
$aryUsers[10..15]
```

6. Multiple values can be accessed in a single statement.

```
$aryUsers[5,67,78]
```

7. Process arrays using the pipeline.

```
$aryUsers | Where-Object {$_ -like “*Smith*”}
```

8. Cmdlets can emit multiple objects, or a single object (singleton). Nevertheless, we can still treat a single object as an array.

```
$process = Get-Process lsass
$process.gettype()
$process[0]
```

# Exercise 13.2: Manipulating Arrays

### Introduction

You can change the elements in an array, add an element to an array, and combine the values from two arrays into a third array.

### Objectives

After completing this exercise, you will be able to:

- Sort, modify and extend an array.

## Task 13.2.1: Extending Arrays

1. Create an array of three strings. Use the compound assignment operator to add a new element to the array.

```
$users = "Johan","Chris","Luke"
$users += "Gary"
$users
$users.Count
```

2. Create a second array and combine the values from the two arrays into a third.

```
$users2 = "Russel","Sean","Amit","Pat","Matt","Darren"
$users3 = $users + $users2
$users3
$users3.Count
```

## Task 13.2.2: Sorting Arrays

1. Sort arrays easily using the Sort-Object Cmdlet.

```
$aryUsers = Get-ADUser -SearchBase "OU=Personnel,DC=Contoso,DC=com" -Filter * |
Select-Object -ExpandProperty SamAccountName

$aryUsers | Select-Object -First 10
$aryUsers | Sort-Object | Select -first 10
$aryUsers | Select-Object -First 10
```

**NOTE:** Notice that the order has not changed permanently in the array $aryUsers

2. To make the sort order permanent you must reassign the array.

```
$aryUsers = $aryUsers | Sort-Object
$aryUsers | Select-Object -First 10
```

3. The array data type implements static members that can (among other methods) permanently sort and reverse an array.

Note that this is permanent, unlike earlier use of sort-object.

```
[array]::Reverse($aryUsers)
$aryUsers | Select-Object -First 10

[array]::Sort($aryUsers)
$aryUsers | Select-Object -First 10
```

## Task 13.2.3: Modifying Arrays

1. To change the value of a particular element in an array, specify the array name and the index of the element that you want to change, and then use the assignment operator (=) to specify a new value for the element.

```
$aryUsers[7]
$aryUsers[7] = “K.Perry”
$aryUsers[7]
```

2. Extend arrays by using the “addition” compound assignment operator.

```
$aryUsers += "J.Jones"
$aryUsers[-1]
```

3. Arrays may also be searched using the IndexOf() method. This method searches the array for a value and returns the index element where the value was found.

```
$index = $aryUsers.IndexOf("J.Jones")
$index
$aryUsers.Count
$aryUsers[$index] = "D.Diver"
$aryUsers[$index]
```

**Note** : The value supplied to the indexof() method is case sensitive. The method returns -1 if the character or string is not found.

4. An array can hold simple or complex objects. Access and modify each object’s members by using dot-notation syntax.

```
$aryUsers2 = Get-ADUser -SearchBase "OU=Personnel,DC=Contoso,DC=com" -Filter *
$aryUsers2[10]
$aryUsers2[10].SID
$aryUsers2[10].Surname
$aryUsers2[10].Surname = “Smythe”
$aryUsers2[10].Surname
```

5. Array contents can be cleared with the Clear() method.

```
$aryUsers.Clear()
$aryUsers.GetType()
$aryUsers.Count
```

6. To delete the array object, assign a value of $null to the array.

```
$aryUsers2 = $null
$aryUsers2.GetType()
$aryusers2.Count
```

# Exercise 13.3: Array Data Type

## Introduction

Windows PowerShell creates a particular type of array object, called “System.Array”. This kind of array can hold any type of object.

## Objectives

After completing this exercise, you will be able to:

- Understand that arrays can store many different object types at once.

## Task 13.3.1: Array data type

1. Create a new array.

```
$genArray = 1,2,3
```

2. Piping the array to the Get-Member Cmdlet returns the data type information of the objects contained within the array, not of the array object itself.

```
$genArray | Get-Member
```

3. To view the members (properties and methods) of the array itself (not the array contents) use the Get-Member Cmdlet’s –InputObject parameter.

```
Get-Member -InputObject $genArray
```

4. Add new elements to the array, containing different object types.

```
$genArray += “John","Sue",(Get-Process)
$genArray
```

5. Use the Get-Member Cmdlet to confirm the array contains different object types.

```
$genArray | Get-Member
```

6. Arrays can contain other array objects. Access nested arrays, and the objects they contain, by using one or more indexes.

```
$genArray[5]
```

7. The counterpoint of the previous step is to define a type-constrained array. This means the array will only accept objects of the specified type.

```
[ int[] ]$intArray = "1","2",3,4,5,6,7,8,"9","10"
$intArray += "Fred"
```

If anybody asks, yes we CAN do multi-dimensional arrays in PowerShell, with the following syntax:

```
$arr2 = new-object 'object[,]' 10,20
$arr2[4,8]=300
$arr2[9,16]="test"
$arr2[4,8]

$arr3 = new-object 'object[,,]' 10,20,10
```

So we are iterating through the array, looking at each of the objects and will Get-Member on each type found.

Now we refer explicitly to the array object itself, not the objects within the array.

An array of objects – we have not restricted what kind of objects, so this array can hold any object we desire.

int[] is “an array of ints” and [ type ] is cast to type, so [ int[] ] can be read as “cast to a type whereby it contains an array of integers.”