Windows PowerShell 4.0 For the IT Professional - Part 1

Module 14: Hash Tables

Student Lab Manual

Version 2.0

Conditions and Terms of Use

Microsoft Confidential - For Internal Use Only

Contents

Lab 14: Hash Tables

Introduction

A hash table, also known as a dictionary or associative array, is a compact data structure that stores one or more key/value pairs.

Objectives

After completing this lab, you will be able to:

- Create and work with hash tables.

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 14.1: Introducing Hash Tables

Introduction

Hash tables are very efficient for finding and retrieving data. The larger the dataset, the greater the performance gain.

Objectives

After completing this exercise, you will be able to:

- Create and manipulate hash tables.

Task 14.1.1: Creating a hash table

1. A hash-table consists of a key and an associated value. The key is usually a string or integer data type and the value can be any object type.

```
$myHashTable = @{"UserName"="DanPark";"ComputerName"="WIN8-WS"}
$myHashTable
```

2. Open the following script within the Windows PowerShell ISE script pane and press <F5> to execute. C:\Pshell\Labs\Lab_14\hashtable1.ps1

```
$DEstates = @{"Saxony-Anhalt" = "Magdeburg"
              "Bavaria" = "Munich"
              "Brandenburg" = "Potsdam"
              "Harz" = "Wernigerode"
              "Saxony" = "Dresden"
              "Lower Saxony" = "Hanover"
              "Mecklenburg-Vorpommern" = "Schwerin"
              "Thuringia" = "Erfurt"
              "Baden-Württemberg" = "Stuttgart"
            }
$DEstates
```

Task 14.1.2: Manipulating Hash Tables

1. Hash tables have multiple ways to access, set, add and remove data.
   You can access the hash-table key in two different ways. Array syntax or dot-notation. Using the hash table created in the previous task, type the following commands.

   a. Array syntax

```
$DEstates[“Saxony”]
```

   b. Dot-notation syntax

```
$DEstates.Saxony
```

2. Assign a value to a key using the assignment operator.

```
$DEstates.Harz
$Destates.Harz = “Quedlinburg“
```

3. If the key does not exist, it will be created along with the associated value.

```
$DEstates["Saarland"] = "Merzig-Wadern"
$Destates.Saarland= "Saarbrucken"
```

4. The Add() method will raise an error condition if the key is already present.

```
$DEstates.Add("Hesse", "Wiesbaden")
$DEstates.Add("North Rhine-Westphalia", "Wuppertal")
```

5. Remove an unwanted key

```
$DEstates.Remove("Harz")
```

6. Search the hash table for a key or value.

```
$DEstates.ContainsKey("Thuringia")
$DEstates.ContainsValue("Schwerin")
```

7. The Get_Item() and Set_Item() methods are also available, however the simpler forms are normally used.

```
if ( $DEstates.ContainsKey("North Rhine-Westphalia") ) {
  $DEstates.Set_Item("North Rhine-Westphalia", "Düsseldorf")
}
```

8. Display the keys, values and total number of items.

```
$DEstates.Keys
$DEstates.Values
$DEstates.Count
```

Exercise 14.2: Sorting Hash Tables

Introduction

The items in a hash table are intrinsically unordered. The key/value pairs might appear in a different order each time that you display them. Although you cannot sort a hash table, you can use the GetEnumerator() method of hash tables to enumerate the keys and values, and then use the Sort-Object cmdlet to sort the enumerated values for display.

Objectives

After completing this exercise, you will be able to:

- Sort a hash table.

Task 14.2.1: Sorting a hash table

1. Sorting of hash tables requires we reference the GetEnumerator() method. Open the following script within the Windows PowerShell ISE script pane and press F5 to execute. C:\Pshell\Labs\Lab_14\hashtable2.ps1.
   Run the commands below.

```
$OZstates = @{"QLD" = "Brisbane"
              "NSW" = "Sydney"
              "ACT" = "Canberra"
              "VIC" = "Melbourne"
              "TAS" = "Hobart"
              "SA" = "Adelaide"
              "WA" = "Perth"
              "NT" = "Darwin"
             }
```

2. Note the order of the values displayed is not the same as the order of entry.

```
write-output "`$OZstates - normal"
$OZstates
```

3. Using the pipeline to the sort-object cmdlet does NOT work

```
$OZstates | Sort-Object
$OZstates
```

4. Use of the GetEnumerator() and then calling sort on the Name will work.

```
$OZstates.GetEnumerator() | Sort-Object -Property Name
```

Task 14.2.2: Ordered hash table

Ordered dictionaries differ from hash tables in that the keys always appear in the order in which you inserted them. The order of keys within a hash table is not determined, as they are based upon the hashes of the key string.

In the order that they were inserted. If anyone asks for deep information on the OrderedDictionary class, suggest they look at MSDN for the System.Collections.Specialized Namespace

1. Open the following script within the Windows PowerShell ISE script pane and press F5 to execute. C:\Pshell\Labs\Lab_14\hashtable3.ps1

```
$BZstates = [Ordered]@{"São Paulo" = "São Paulo"
             "Rio de Janeiro" = "Rio de Janeiro"
             "Pará" = "Belém"
             "Goiás" = "Goiânia"
             "Acre" = "Rio Branco"
             "Roraima" = "Boa Vista"
             "Sergipe" = "Aracaju"
             "Tocantins" = "Palmas"
             "Maranhão" = "São Luís"
             "Bahia" = "Salvador"
           }
```

2. Note the displayed order IS the same as the entry order

```
$BZstates
```

3. What happens if you try to cast an existing hash table to [ordered]?

Error. (See about_Hash_Tables) "You cannot use the [ordered] attribute to convert or cast a hash table. If you place the ordered attribute before the variable name, the command fails"

4. What happens if you copy and paste the above without the [ordered] type literal?

It is a normal hash table. (ie not Ordered)

Exercise 14.3: Hash table usage

Introduction

You can use hash tables to store lists and to create calculated properties in Windows PowerShell. In addition, the Cmdlet ConvertFrom-StringData converts strings to a hash table.

Objectives

After completing this exercise, you will be able to:

- Use hash tables in various real-world scenarios.

Task 14.3.1: Nested hash tables

The keys and values in a hash table can have any .NET object type, and a single hashtable can have keys and values of multiple types.

1. The hash table values can be anything, and we can access the value using the dot operator. That value in turn could be a hash table we created earlier. This is called a "nested hast table".
   Create an empty hash table.

```
$capitals = @{}
```

2. Assign the previous hash tables as values within the Capitals hash table.

```
$capitals.BZ=$BZstates
$capitals.OZ=$OZstates
$capitals.DE=$DEstates
```

3. Access the previous hash tables using either standard dot-notation or array syntax.

```
$capitals.OZ
$capitals.OZ.SA
$capitals.OZ["NSW"]
$capitals.DE.'Lower Saxony'
$capitals.BZ.Tocantins
```

Task 14.3.2: ConvertFrom-StringData

This Cmdlet converts a string containing one or more key/value pairs to a hash table. You can use the ConvertFrom-StringData cmdlet safely in the DATA section of a script, and you could use it with the Import-LocalizedData cmdlet to display user messages in the user-interface (UI) culture of the current user. (See: Get-Help about_Data_Sections)

1. Create a Here-String and use ConvertFrom-StringData.

```
$hereString = @'
   Msg1 = The string parameter is required.
   Msg2 = Credentials are required for this command.
   Msg3 = The specified variable does not exist.
'@

$hashtable = $hereString | ConvertFrom-StringData
$hashtable
```

Task 14.3.3: Calculated Properties

Let us explore the usage of hash tables in "Calculated Properties".

Make sure the student understands the usefulness of this! Trigger a discussion on where/when they might need it.

NOTE: The "Value" side of the key/value pair has a specific reference name of "Expression" when working with calculated properties generated within a script-block.

1. Open the script C:\Pshell\Part1\Labs\Lab_14\hashtable6.ps1 in the Windows PowerShell ISE script pane and press F5 to execute it.

```
Get-ChildItem C:\Windows |
Select-Object Name, CreationTime,  @{Name="Kbytes";Expression={$_.Length / 1Kb}} |
Format-Table -AutoSize

Get-ChildItem C:\Windows |
Select Name, @{Name="Age";Expression={((Get-Date)-$_.CreationTime).Days)}} |
Format-Table -AutoSize

Get-Process csrss |
FT ProcessName, @{Label="TotalRunTime";Expression={(get-date) - $_.StartTime}}
```

2. What other useful “calculated fields” might you create in your own environment? Try out your ideas, drawing upon the examples above for inspiration. Two examples are shown below.
3. Supply a hash table to Select-Object in order to allow parameter binding to complete successfully.

```
“2012R2-DC”,”2012R2-MS” |
Select-Object @{Name=”ComputerName”;Expression={$_}} |
Get-Service –Name BITS |
Format-Table –Property MachineName, Name, Status –AutoSize
```

4. Convert an enumeration into its Integer, Hexadecimal and Binary representations.

```
[Enum]::GetValues([System.IO.FileAttributes]) |
Select-Object @{n="String";e={$_}}, `
@{n="Integer";e={[int]$_}}, `
@{n="Hexadecimal";e={[Convert]::ToString([int]$_, 16)}}, `
@{n="Binary";e={[convert]::Tostring([int]$_, 2)}} |
Format-Table -Autosize
```

Task 14.3.4: Splatting

Some Cmdlets have *many* parameters. In order to collect much of the data, you can use a hash table. There is a way to pass all the key/value pairs directly to the cmdlet. This is referred to as "splatting". See: Get-Help About_Splatting

> **NOTE:** When splatting, you do not need to use a hash table or an array to pass all parameters. You may pass some parameters by using splatting and pass others by position or by parameter name. In addition, you can splat multiple objects in a single command just so you pass no more than one value for each parameter.

1. Create variables to contain the data we wish to pass into New-ADUser.

```
$path="OU=Finance,OU=Personnel,DC=Contoso,DC=com"
$lastname="Fife"
$firstName="Paul"
$name="P.Fife"
$title="Specialist"
```

> **NOTE:** This OU should exist from a previous lab.

2. Imagine we have received these values from files or database or user input. The list of parameters could be quite large. For example, New-ADuser accepts over fifty parameters. We can build up all the values into a single hash table, the key names matching the Cmdlet parameter names.

```
$splat =
@{"Name"=$name;"Surname"=$lastName;"GivenName"=$firstName;"Path"=$path;"Title"=$ti
tle}
```

3. Call the Cmdlet, with this single hash table reference, using the form @hashtable

```
4.  New-ADUser @splat
```

5. Try creating a simple function within a script and pass arguments using this technique.

Task 14.3.5: New-Object Properties

Beginning in Windows PowerShell 3.0, you can create an object from a hash table of properties and property values. See: Get-Help about_Object_Creation.

1. Use the PSCustomObject type available from Windows PowerShell v3.0 onward.

```
$obj = [PSCustomObject] @{
          Division = 'IT'
          Laptop   = 'Surface'
          Ferrari  = $false
        }

$obj | Format-List -Property *
```

2. Use the earlier v2.0 syntax.

```
$props = @{
          Division = 'IT'
          Laptop   = 'Surface'
          Ferrari  = $false
          }

$object = New-Object PSObject -Property $props
$object | Format-List -Property *
```

Task 14.3.6: Default Parameter Values

The $PSDefaultParameterValues preference variable lets you specify custom default values for any Cmdlet or advanced function. Cmdlets and functions use the custom default value unless you specify another value in the command.

1. This special preference variable is a hash table in the usual sense, so any keys or values can be added, removed, and altered in the same fashion as in earlier exercises. Define a default for the SMTP server parameter from the Send-MailMessage Cmdlet.

```
$PSDefaultParameterValues = @{"Send-MailMessage:SmtpServer"="2012R2-MS"}
```

2. Create a hash table of mail parameters.

```
$splat = @{"To"="SomeRecipient@contoso.com"
           "From"="PowerShell@Contoso.com"
           "Subject"="SMTPSERVER is NOT passed as a parameter"
           "Body"="the ease of using PSDefaultParameterValues"
         }
```

3. Send the e-mail message using the splatting syntax.

```
Send-MailMessage @splat
```

4. Check to see the mail was sent.

```
dir \\2012R2-MS\C$\Inetpub\mailroot\drop
```

5. Explore this further by adding further key/value pairs like this:

```
$PSDefaultParameterValues.Add("*:Confirm",$True)
$PSDefaultParameterValues.Add("Test-Connection:Quiet",$True)
```

6. Once you have tried a few things, you can clear the settings using the Clear() method.

```
$PSDefaultParameterValues.Clear()
```

Task 14.3.7: Search Performance

Let us explore the speed of access to an array or a hash table when searching for a value. As the sizes increase the disparity becomes huge. We will use Measure-Command to capture command execution timings.

For a hash table, you will usually see O(1) and a worst case of O(n). The bigger the data-set, the bigger the win, when searching for some random item.

1. Open the following script within the Windows PowerShell ISE script pane and press <F5> to execute. C:\Pshell\Labs\Lab_14\hashtable10.ps1

```
$hash = @{}
$array = @()
```

2. Measure the elapsed time when adding to a hash table versus an array.

```
Measure-Command -Expression {1..2555 | %{$hash.Add("192.168.0.$_",0)}}
Measure-Command -Expression {1..2555 | %{$array+= "192.168.0.$_"}}
```

3. Measure the elapsed time when looking up a value.

```
Measure-Command -Expression { 1..2555 | % {$hash.ContainsKey("192.168.0.$_")} }
Measure-Command -Expression { 1..2555 | % {$array -contains "192.168.0.$_"} }
```

4. Measure when accessing directly with a known index.

```
Measure-command -expression {$hash["192.168.0.2555"]}
Measure-command -expression {$array[2554]}
```

5. Create your own brief tests of hash tables versus arrays to explore this further.