**Here-string literals**

PS> $a = @"

One is "1"

Two is '2'

Three is $(2+1)

The date is "$(Get-Date)"

"@

PS> $a

One is "1"

Two is '2'

Three is 3

The date is "06/09/2017 14:54:10"

**Hashtable definitions**

$user = @{ FirstName = 'John'; LastName = 'Smith';

PhoneNumber = '555-1212' }

$user

Or

Simple:

$user = @{

FirstName = 'John'

LastName = 'Smith'

PhoneNumber = '555-1212'

}

Out put for both

Key Value

--- -----

LastName Smith

FirstName John

PhoneNumber 555-1212

Filter out put

PS> $user.firstname

John

This notation lets you treat a hashtable like an object and is intended to facilitate the use of

hashtables as a kind of lightweight data record. Now let’s look at using the array notation:

PS> $user['firstname']

John

PS> $user['firstname','lastname']

John

Smith

Another type add content

PS> $h = @{a=1; b=2; c=3}

PS> foreach ($pair in $h.GetEnumerator()) {

$pair.key + " is " + $pair.value

}

a is 1

b is 2

c is 3

**2.3.2. Ordered hashtables**

$oh = [ordered] @{ }

PS> $oh[[object] 5] = 'five'

Now the assignment succeeds but you have to do the same trick to retrieve the element by key:

PS> $oh[[object] 5]

five

If you pass a number without the explicit cast, it will look up using the element index. The

correct element index for the key 5 is 0:

PS> $oh[0]

Five

**2.3.3. Modifying and manipulating hashtables**

**(65)**

Next let’s look at adding, changing, and removing elements in the hashtable. First let’s add the

date and the city where the user lives to the $user table:

PS> $user.date = Get-Date

PS> $user['city'] = 'Seattle'

PS> $user

Name Value

---- -----

date 09/06/2017 15:18:12

city Seattle

PhoneNumber 555-1212

FirstName John

LastName Smith

remove() method:

PS> $user.remove("city")

If you want to create an empty hashtable, use @{ } with no member specifications between the

braces. This creates an empty table you can then add members to incrementally:

PS> $newHashTable = @{}

PS> $newHashTable

PS> $newHashTable.one = 1

PS> $newHashTable.two = 2

PS> $newHashTable

Key Value

--- -----

two 2

one 1

This technique can also be used for an ordered hashtable.

**Where() and ForEach() methods**

Consider a standard use of Where-Object:

PS> Get-Process | where Handles -gt 1000

The collection of processes is filtered and only those processes with more than 1,000 handles are

returned. You can use the Where() method to achieve the same result:

PS> (Get-Process).where({$\_.Handles -gt 1000})

PS> (Get-Process).where({$psitem.Handles -gt 1000})

You must use either $\_ or $psitem with the property on which you’re filtering. The () are optional

but we recommend you use them to make the syntax more obvious when you come to review it,

or when you’re writing for others to use. We’ll use the () in the rest of this section to make the

syntax more obvious but as an example of not using them you can write the previous two

examples as:

PS> (Get-Process).where{$\_.Handles -gt 1000}

PS> (Get-Process).where{$psitem.Handles -gt 1000}

Qualifiers can be applied to display the first or last member of the collection:

PS> (Get-Process).where({$\_.Handles -gt 1000}, 'First')

PS> (Get-Process).where({$\_.Handles -gt 1000}, 'Last')

This can be extended to the first or last *n* members:

PS> (Get-Process).where({$\_.Handles -gt 1000}, 'First', 3)

PS> (Get-Process).where({$\_.Handles -gt 1000}, 'Last', 3)

There’s an option to split the results:

PS> $proc = (Get-Process).where({$\_.Handles -gt 1000}, 'Split')

$proc is a collection—the first member contains the processes that match the filter and the second

member those that don’t.

You can further filter the results using Until and SkipUntil:

PS> (Get-Process | sort Handles).where({$\_.Handles -gt 1000}, 'Until')

Using Until will display all results until you reach results that match the filter defined in the

scriptblock. If you want to display only the results that match the filter then use SkipUntil:

PS> (Get-Process | sort Handles).where({$\_.Handles -gt 1000}, 'SkipUntil')

If you don’t sort the members of the collection, SkipUntil will display everything after the first

match irrespective of whether it matches the filter.

**3.6.2. ForEach() method**

The ForEach() method is a bit simpler than the Where() method you’ve just seen. Again,

demonstrating the use of this method is best achieved by some examples. First, create an array of

integers:

PS> $data = 1,2,3,4,5

You can execute a scriptblock within the ForEach() method:

PS> ($data).ForEach({$\_ \* 2})

PS> $data.ForEach({$\_ \* 2})

When the data is already an array you don’t need to wrap it in (). If you need to change the type

of the objects in the collection, use this approach:

PS> $data | Get-Member

PS> $data.ForEach([double]) | Get-Member

Values for a particular property can be displayed as follows:

PS> (Get-Process).foreach('Name')

If the objects within the collection have methods, they can be invoked:

PS> (Get-Process -Name notepad).foreach('Name')

PS> (Get-Process -Name notepad).foreach('Kill')

You can also pass arguments into the method if required.