Hands-On Lab LW05
Lab Manual

Monad Scripting

Please do not remove this manual from the lab
## Contents

### LAB 1: MONAD SCRIPTING

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lab Objective</td>
<td>1</td>
</tr>
<tr>
<td>Exercise 1 – Use Simple Shell Commands</td>
<td>1</td>
</tr>
<tr>
<td>Task 1 – Running Commands</td>
<td>1</td>
</tr>
<tr>
<td>Task 2 – Running Cmdlets</td>
<td>2</td>
</tr>
<tr>
<td>Task 3 – Running Cmdlets Safely</td>
<td>3</td>
</tr>
<tr>
<td>Task 4 – Getting Help</td>
<td>3</td>
</tr>
<tr>
<td>Task 5 – Create Aliases</td>
<td>4</td>
</tr>
<tr>
<td>Exercise 2 – Expressions</td>
<td>4</td>
</tr>
<tr>
<td>Task 1 – Create an Expression</td>
<td>4</td>
</tr>
<tr>
<td>Task 2 – Assignment</td>
<td>5</td>
</tr>
<tr>
<td>Exercise 3 – Using Pipelines</td>
<td>6</td>
</tr>
<tr>
<td>Task 1 – Understanding how the filter Cmdlets work</td>
<td>6</td>
</tr>
<tr>
<td>Task 2 – Using Cmdlets to discover information about what other Cmdlets produce</td>
<td>7</td>
</tr>
<tr>
<td>Task 3 – Using the filter Cmdlets to manipulate a dataset</td>
<td>7</td>
</tr>
<tr>
<td>Task 1 – Use the filter Cmdlets to create more complex operations</td>
<td>7</td>
</tr>
<tr>
<td>Exercise 4 – Branching Statements</td>
<td>8</td>
</tr>
<tr>
<td>Task 1 – If/ElseIf/Else</td>
<td>8</td>
</tr>
<tr>
<td>Task 2 – Switch</td>
<td>9</td>
</tr>
<tr>
<td>Exercise 5 – Looping</td>
<td>10</td>
</tr>
<tr>
<td>Task 1 – Understanding the WHILE statement</td>
<td>10</td>
</tr>
<tr>
<td>Task 2 – Understanding the &quot;DO WHILE&quot; statement</td>
<td>10</td>
</tr>
<tr>
<td>Task 3 – Understanding the &quot;DO UNTIL&quot; statement</td>
<td>10</td>
</tr>
<tr>
<td>Task 4 – Understanding the FOREACH statement</td>
<td>11</td>
</tr>
<tr>
<td>Task 5 – Understanding the FOR statement</td>
<td>11</td>
</tr>
<tr>
<td>Exercise 6 – Handling Errors</td>
<td>11</td>
</tr>
<tr>
<td>Task 1 – Retrieve Errors</td>
<td>12</td>
</tr>
<tr>
<td>Task 2 – Recognizing terminating and non-terminating errors</td>
<td>12</td>
</tr>
<tr>
<td>Task 3 – Using Trap</td>
<td>13</td>
</tr>
<tr>
<td>Exercise 7 – Formatting Output</td>
<td>13</td>
</tr>
<tr>
<td>Task 1 – Selecting your format look</td>
<td>14</td>
</tr>
<tr>
<td>Task 2 – Modifying the formatting</td>
<td>14</td>
</tr>
<tr>
<td>Exercise 8 – Creating Functions and Scripts</td>
<td>14</td>
</tr>
<tr>
<td>Task 1 – Declaring Function Names</td>
<td>14</td>
</tr>
<tr>
<td>Task 2 – Declaring Parameters and Dealing with Arguments</td>
<td>15</td>
</tr>
<tr>
<td>Task 3 – Functions and Input</td>
<td>15</td>
</tr>
<tr>
<td>Task 4 – Scripts</td>
<td>15</td>
</tr>
<tr>
<td>Exercise 9 – Solving Problems</td>
<td>16</td>
</tr>
<tr>
<td>Task 1 – Write a function</td>
<td>16</td>
</tr>
<tr>
<td>Task 2 – Write a function</td>
<td>16</td>
</tr>
<tr>
<td>Task 3 – Write a filter</td>
<td>16</td>
</tr>
</tbody>
</table>

### SOLUTIONS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task 1</td>
<td>18</td>
</tr>
<tr>
<td>Task 2</td>
<td>18</td>
</tr>
<tr>
<td>Task 3</td>
<td>18</td>
</tr>
</tbody>
</table>
Lab 1: Monad Scripting

Become familiar with Monad syntax and semantics. Use and understand the two modes of command parsing (command mode and expression mode).

Lab Objective

Estimated time to complete this lab: 90 minutes

The following tasks should help gain familiarity with the Monad shell.

- Use Simple Shell Commands
- Use Expressions
- Use pipelines to stream data from one command to another
- Use statements to create program flow
- Formatting your output
- Create functions
- Create scripts

Exercise 1 – Use Simple Shell Commands

Proctor Notes: This is all expository and should be fairly straightforward. It shows how the shell is able to run existing executables and new Cmdlets in the same way. Questions arising about "Command Mode" are likely to arise during this exercise; however, it should be clear by the end of exercise 2 what the differences between the two modes ("Command Mode" and "Expression Mode") are. The "Should Process" parameters are introduced. Questions may arise with regard to help and whether it is available from the GUI (not at present).

In this exercise, you will learn how to start the shell and run commands as well as how to get help about the various commands and topics about the shell. You will also discover the differences between stand-alone and built-in commands and how to create aliases to reduce the amount of typing that is needed. You will be using one of the two modes of the shell "Command Mode"; the other mode, "Expression Mode", will be discussed in Exercise 2.

Task 1 – Running Commands

Running commands in the Monad shell is the same as other shells. This task shows that the new shell is very similar to CMD.EXE - you type commands and execute programs. It is important to note that you are not running any built-in commands (such as DIR or COPY, etc) in this first task.

1. Start the shell
Start Menu -> Programs -> Windows Command Shell

2. Set the location to the shell install directory

   MSH> set-location $mshhome

3. Run some simple commands, note that you should see the default shell prompt "MSH> " in the new console window. Also note that we are running stand-alone applications.

   MSH> ipconfig
   MSH> notepad
   MSH> net statistics workstation

Note that the output for those commands is seen in the console window and that for graphical commands, the prompt returns right away. Also note that arguments are provided to the commands without special handling; for example, no quoting is needed to provide the string parameter values. When executing commands, the shell is in Command Mode, this means that generally quoting is not needed for arguments to commands.

More than one command (or Cmdlet) can be run on the same line. The ";" indicates that the statement is finished.

4. Run ipconfig followed by net statistics workstation on the same line

   MSH> ipconfig; net statistics workstation

Task 2 – Running Cmdlets

Cmdlets are the equivalent to the CMD.EXE built-in commands but have very specific naming requirements to help provide a consistent way to learn this new environment. Each Cmdlet is broken into two parts; a verb and a noun separated by hyphen.

1. Start the Shell.
2. Run some Cmdlets

   MSH> get-location
   MSH> get-childitem
   MSH> get-date

Note the format of these commands, all in the form of get-something. The shell has a large number of built-in commands (more than 120). Another cmdlet is provided to show you the list of Cmdlets.

3. Run the Get-Command cmdlet

   MSH> get-command

You will see that the list of Cmdlets is alphabetized by name, this will aid in locating the Cmdlet that you want. The Get-CommandCmdlet has parameters. A parameter is a way to provide more information to a command to get different or more specific information. By providing a parameter to the get-command Cmdlet, it is possible to get more information about a command.

4. Run the Get-Command Cmdlet with an argument

   MSH> get-command -name get-location -synopsis

You will see that the output is no longer a list of Cmdlets but the synopsis of how to use the Cmdlet. This can be useful when determining which parameters are available to the Cmdlet.
Task 3 – Running Cmdlets Safely

In this task you will see how Monad provides a level of safety by providing a mechanism to see what the Cmdlet will do without actually performing an operation that might be unsafe. The "Should Process" parameters include -whatif, -confirm and -verbose. The "-whatif" parameter will echo what this Cmdlet will do, without actually doing it.

1. Get the processes of the system that match MSH

   MSH> get-process msh*

See what would happen if you were to stop the processes of the system that match MSH

   MSH> get-process msh*
   MSH> stop-process -id 1124 -whatif

(using the Process ID number retrieved by get-process) You should see output similar to

   What if: Performing operation "stop-process" on Target "msh (1124)".

This tells you what would have happened if the Cmdlet had executed normally. It should be noted that not all Cmdlets support the -whatif parameter. This functionality must be implemented by the Cmdlet developer. Our guidance to developers is that if a Cmdlet changes the state of the system, they should support -whatif. However, individual Cmdlets may vary in implementation.

2. Get the processes of the system that match MSH and use -confirm to ensure that an operation does not take place unless the action is confirmed.

   MSH> get-process msh*

Use -confirm to have the shell inquire whether the process should be stopped.

   MSH> stop-process -id 1124 -confirm

You should see output similar to

   Confirm
   Are you sure you want to perform this action?
   Performing operation "stop-process" on Target "msh (1124)".

3. Get the processes of the system that match notepad and use -verbose to get detailed information about what the Cmdlet is doing. First, start notepad

   MSH> notepad

Use "-verbose" to have the shell report on what applications are stopped.

   MSH> get-process notepad
   MSH> stop-process -id 5772 -verbose

You should see output similar to

   Performing operation "stop-process" on Target "notepad (5772)".

Task 4 – Getting Help

Monad has online help available so you can find out about how to use the shell. A Cmdlet named Get-Help will provide help on the functionality of the shell including the Cmdlets, language elements and other conceptual topics.

1. Get the general help

   MSH> get-help

This will display the introduction to the shell help system. Review this material, note that some topics may not be available in this pre-release.

2. Get help about the Get-Help Cmdlet

   MSH> get-help get-help

3. Where there are examples in the help output, run those examples to further discover the shell environment.

4. Spend some time discovering the contents of the online help.
Task 5 – Create Aliases

Because the Monad shell environment is more verbose than other shells given our verb-noun format of Cmdlets we provide an aliasing capability that allows you to create a "nickname" for Cmdlets to reduce the amount of typing.

1. See what aliases are available by default by running the get-alias Cmdlet.

   MSH> get-alias

2. Use an alias

   MSH> gps

   Note that this is the same result as if you typed "Get-Process"

3. Create a new alias, called mydir which is an alias for get-childitem

   MSH> new-alias mydir get-childitem

4. Use the new alias

   MSH> mydir

It is important to note that aliases are simple "nicknames" for Cmdlets. It is not possible to create an alias for a Cmdlet and parameters. That will be done with functions in a later exercise.

Exercise 2 – Expressions

Proctor Note: Introduce expressions and clarify the difference between Expression and Command mode.

In this exercise, you will use expressions and discover the difference between Expression mode and Command mode

Task 1 – Create an Expression

With Monad an expression is created when it does not result in the execution of a Cmdlet or other executable. For example, If an unquoted string is typed at the prompt, the shell will attempt to find an executable using the string as a name, if the shell cannot find an executable, an error is returned. However, if a quoted string is typed at the prompt, the shell's expression mode is entered and the shell will echo back the quoted string as an expression.

1. Type an expression in the form of a quoted string

   MSH> "Hello World"

   Hello World

   Note that the string 'Hello World' is echoed back to the user. Strings, when found as the first element in a command place the shell in expression mode and any result from the expression will result in a string.

2. Create a string expression by combining two strings.

   MSH> "Hello" + " World"

   Hello World

   Note that in the Monad shell, it is possible to create strings by using the "+" operator. This allows strings to be created very easily. It is also possible to create strings via multiplication. The multiplication of strings requires that a string, followed by "*", followed by a number is present.

3. Create a string of "#"s that is 65 characters long.

   MSH> "#" * 65
4. Expression mode is also entered when a number is the first element of a command. This lets the shell act like a calculator.

```
MSH> 2 + 3 * 4 / 5
4.4
```

Note the precedence in this operation is first multiplication and division, reading left to right followed by addition and subtraction, reading left to right. The following is what the explicit precedence would be in the above example:

```
MSH> 2 + ((3 * 4)/5) - 6
-1.6
```

**Task 2 – Assignment**

An assignment expression has two parts, the left hand side (LHS) of the "=" and the right hand side (RHS). The LHS will be the storage of the results of the RHS, this storage is a variable, is generally a string and must begin with a "$". The RHS may be an expression (such as a string, or arithmetic expression), or it may be a command.

1. Create a Variable named `Var` with the value 2

```
MSH> $var = 2
```

Note that the RHS is an expression, not a command. In order to see the value of the variable, you only need to refer to it, no extra command is necessary to echo it to the screen.

2. See the contents of the variable

```
MSH> $var
2
```

3. It is also possible to store the results of a command execution in a variable. Try storing the results of the Get-Date Cmdlet to the variable `date`

```
MSH> $date = Get-Date
```

Note, in this case, the RHS is a command - this is an expression where the RHS is in Command Mode. Now, retrieve the contents of the variable

```
MSH> $date
Friday, August 19, 2005 11:06:54 AM
```

It is possible to store more than a single value in a variable. Variables can be arrays of values. The following shows how to create an array of 4 numbers and assign it to a variable and then retrieve the contents of the variable

```
MSH> $arr = 1,2,3,4
MSH> $arr
1
2
3
4
```

Note that the array is shown over 4 lines, since there are 4 elements in the array; each element gets its own line of output. Many Cmdlets return more than one result. These results can be stored in a variable in the same way as single results.

You can also force an array to be created. Using the array operator `@()` ensures that the contents between the `()` will be treated as an array. The following forces an array of 1 element to be created:

```
MSH> $arr = @("Hello There")
```
4. Associative arrays (also called hash tables) may be created in the Monad shell. The general syntax for creating a hash table is:

$hash = @( key1 = value; key2 = value )

where value is an expression that returns a value (so this could be a number, string, or pipeline). To refer to the values of a hash table, you can use the following syntax:

$hash['key1'] or $hash.key1.

Create a hash table with three keys, the value of the first is the number 1, the value of the second is the string "two" and the value of the third the result of the get-date Cmdlet. Then retrieve the value of the third key.

MSH> $hash = @( key1 = 1; key2 = "two"; key3 = get-date )
MSH> $hash.key3

5. Run a Cmdlet and save the results in a variable

MSH> $Processes = Get-Process

The variable "$Processes" now contains the results of the Get-Process Cmdlet. This is an array of Process Objects. These objects can now be retrieved.

It is very important to note that the variable contains the objects rather than a text representation of the objects. Later we will see how important this is.

6. Retrieve the first and second elements of the Processes variable

MSH> $Processes[0,1]

Since this variable is an array, it is possible to use the shell's array notation to retrieve elements from the variable. Run Get-Help about_array for more details.

Exercise 3 – Using Pipelines

In this section we will go through creating pipelines and using the filter Cmdlets. The filter Cmdlets (where, sort, foreach, select) are used extensively in pipelining. Although it is possible to create a pipeline without a filter Cmdlet, they're fairly limited in flexibility (get-process notepad | stop-process is useful, but much more so when combined with a filter or two).

In this exercise, you will understand what pipelines are in the Monad shell and how to create them. MSH allows you to combine two or more commands into a single custom command that creates an operation known as a pipeline. When commands are chained together in a pipeline, they pass data to each other as objects. The first command returns one or more objects that it passes down the pipeline to the second command. The second command processes those objects and then passes on new or revised objects to the third command. This process continues until each command in the pipeline has run.

In order to really get the most of pipelines, we will discover how the filter Cmdlets can be used to create very powerful pipelines

Task 1 – Understand how the filter Cmdlets work

A number of the *-Object commands may be used to filter or act on every object that it receives. In this task we will inspect the help for the filter Cmdlets. In each sub-task be sure to run the examples found in the help.

1. Inspect the help for the Sort-Object Cmdlet

MSH> get-help Sort-Object
2. Inspect the help for the Where-Object Cmdlet, specifically reviewing the about_script_block help.
It is important to remember that $_ is a variable that represents the current pipeline object. We
will use this a great deal when using where-object.

   MSH> get-help Where-Object
   MSH> get-help about_script_block

3. Inspect the help for the Foreach-Object Cmdlet. It is important to remember that $_ is a variable
that represents the current pipeline object. We will use this a great deal when using foreach-
object.

   MSH> get-help Foreach-Object

4. Inspect the help for the Select-Object Cmdlet

   MSH> get-help Select-Object

5. Inspect the help for the Group-Object Cmdlet

   MSH> get-help Group-Object

Task 2 – Using Cmdlets to discover information about what other Cmdlets produce

One of the most important Cmdlets is the Get-Member Cmdlet. This Cmdlet allows you to discover
things about the objects that are retrieved by the system. It is one of the most important tools of
discovery in the shell. You generally use this Cmdlet by piping the results of one Cmdlet to it.

1. Use the Get-Member Cmdlet to discover what types of objects the get-process Cmdlet returns

   MSH> get-process | get-member

2. You should be able to see that the previous pipeline retrieved information about the objects
returned by get-process. You can also get information about the elements of those objects.
Using the Get-Member Cmdlet in combination with Foreach-Object, we can discover more about
the process object.

   MSH> get-process | foreach-object { $_.MainModule } | get-member

3. In some cases, a Cmdlet may return more than one type of object. In these cases, the get-
member Cmdlet will return information for each of the types that it receives.

   MSH> get-childitem | get-member

Task 3 – Using the filter Cmdlets to manipulate a dataset

Now that we have a handle on how the utility Cmdlets work, we can use them to filter, group and
otherwise manipulate the data that is retrieved by other Cmdlets.

1. Use the Where-Object Cmdlet to retrieve only those process objects that have a handlecount
greater than 500

   MSH> get-process | where-object { $_.handlecount -gt 500 }

2. Use the Where-Object Cmdlet to retrieve only those files that have a size larger than 1 megabyte
in the Windows\System32 directory

   MSH> get-childitem C:\windows\system32 | Where-Object { $_.length -gt 1M }

3. For each object that is returned by the Get-Drive Cmdlet, only print the name of the drive

   MSH> get-drive | foreach-object { $_.name }

4. Select the top 10 processes based on VirtualMemorySize (This uses both sort-object and select-
object)

   MSH> get-process | sort-object VirtualMemorySize | select-object -last 10

Task 1 – Use the filter Cmdlets to create more complex operations

The following example will return all the parameters for all the Cmdlets that are available.

   MSH> get-command -type cmdlet
   foreach-object { $_.parametersets }
   foreach-object { $_.parameters }
   foreach-object { $_.name }
Let's go through each line of this script
1. Retrieve the CmdletInfo objects for all Cmdlets
2. For each one of the CmdletInfo objects, retrieve the ParameterSet Objects
3. For each one of the ParameterSet Objects, retrieve the individual Parameters
4. For each one of the parameters, retrieve the name
5. Send all the results to sort (this is a string so no property is needed)
6. Group the sorted results. Since this is a string, we will create the groups based on the parameter name.
7. Sort the grouped results based on the Count property of the Group-Object

It is important to know how to retrieve the information from objects; get-member can be used to "break up" the objects retrieved by other Cmdlets - that's how to discover that the Get-Command objects have the property ParameterSets.

Exercise 4 – Branching Statements

In this section we will investigate the different statements available in the Monad Language.

In this exercise, you will use the various branching functionality of the shell.

Task 1 – If/ElseIF/Else

You can use the IF statement to run a code blocks if a specified conditional test evaluates to true. You can also specify one or more additional conditional tests to run if all prior tests evaluate to false. Finally, you can specify an additional code block that is run if no other prior conditional test evaluates to true.
The following shows the IF statement syntax:

```powershell
if (<test1>)
    {<code_block1>}
[elseif (<test2>)
    {<code_block2>}
[else
    <code_block3>]
```
1. The comparison operators are used a great deal in branch logic, you should review the help for comparison operators by reviewing the help on about_comparison_operators

```
MSH> get-help about_comparison_operators
```
2. Create a variable "a" with the value of 1

```
MSH> $a = 1
```
3. Create an IF statement that if $a is greater than 0, it emits "YES!"

```
MSH> if ( $a -gt 0 ) { "YES!" }
```
4. Create an IF/ELSEIF/ELSE statement that first checks if $a is 0 and if so emit "0", then $a -eq 1 and if so, emit "1", otherwise emit "other"

```
MSH> if ( $a -eq 0 ) { "0" } elseif ( $a -eq 1 ) { "1" } else { "other" }
```
While you can use multiple elseif statements to chain a series of conditional tests, each of which is examined only if all the previous tests are false. However, if you need to create an if statement containing many elseif statements within it, consider using a switch statement instead.

**Task 2 – Switch**

The switch statement is a control statement that handles multiple selections by passing control to one of the conditional statements within its body. Each conditional statement must be matched with a statement block that will be executed if the conditional statement is met. The switch conditional has the following form:

```
switch [-regex|-wildcard|-exact][-casesensitive][-file filename | ( <pipeline> ) ]
{[default|string|number|variable|{ conditional expression } { statementblock }]}
```

By default, if no options are used, switch behaves as if a case insensitive exact match is in effect. If "pipeline" results in an array, each element of the array is evaluated in order. At least one conditional element must be present and only one default clause may be present. If more than one "default" clause is present, an error will result.

**Examples:**

```
MSH> $a = 3
MSH> switch ($a) {
1 { "got one"}
2 { "got two"}
3 { "got three"}
}
got three
```

In some cases, the special variable $_ is available in the script. $_ is an automatic variable that represents the current value being evaluated.

```
MSH> $var = "word2"
MSH> switch -regex ($var) {
"word2" { "Exact " + $_ ; break }
"word.*" { "Pattern match Exact1 " + $_ ; break }
"w.*"{ "Pattern match first letter " + $_ ; break }
default { "Default " + $_; break }
}
Exact word2
```

In the following example, the omission of the **break** keyword allows for multiple matches.

```
MSH> $var = "word1","word2","word3"
MSH> switch -regex ($var) {
"word1" { "Multi-match Exact " + $_ }
"word2" { "Multi-match Exact " + $_ }
"w.*2" { "Pattern match Exact " + $_ }
default { "Multi-match Default " + $_ }
}
Multi-match Exact word1
Multi-match Exact word2
Pattern match Exact word2
Multi-match Default word3
```

The following example uses **script** blocks to determine whether a match is made and the omission of the **break** keyword allows for multiple matches.
You should also review the help on about_break and about_continue which are generally used in switch statements.

**Exercise 5 – Looping**

MSH provides five looping statements; while, do while, do until, foreach and for.

**Task 1 – Understanding the WHILE statement**

The WHILE statement (also known as a while loop) is a language construct for creating a loop that runs commands in a script block as long as a conditional test evaluates to true. The "WHILE" statement is easier to construct than a "FOR" statement because it's syntax is less complicated. In addition, it's more flexible than the FOREACH statement because you specify a conditional test in the while statement to control how many times the loop runs.

The following shows the while statement syntax:

while (<condition>) {<script_block>}

1. Create a loop that counts from 1 to 10

   ```powershell
   MSH> $i = 1
   MSH> while($i -le 10) { $i; $i++ }
   ```

**Task 2 – Understanding the "DO WHILE" statement**

The "DO WHILE" statement is similar to the WHILE except that the condition is checked at the end of the command and repeats while the condition is true. The following shows the while statement syntax:

```
do {<script_block>} while (<condition>)
```

1. Create a loop that counts from 1 to 10

   ```powershell
   MSH> $i = 1; do { $i; $i++ } while($i -le 10)
   ```

2. What happens if the loop is run again? (without resetting the $i variable)

   ```powershell
   MSH> do { $i; $i++ } while($i -le 10)
   11
   ```

   This is because the evaluation for truth occurs at the end of the loop rather than at the beginning (as in the WHILE statement)

**Task 3 – Understanding the "DO UNTIL" statement**

The "DO UNTIL" statement is similar to the WHILE except that the condition is checked at the end of the command and it repeats until the condition is true.
The following shows the while statement syntax:
\[
do {<\text{script\_block}>} \text{ until } (<\text{condition}>)
\]
1. Create a loop that counts from 1 to 10
   MSH> $i = 1; \text{ do } \{ \text{$i$; $i++ } \text{ until ($i -eq 11$) }
2. What happens if the loop is run again? (without resetting the $i$ variable)
   MSH> \text{ do } \{ \text{$i$; $i++ } \text{ until ($i -eq 11$)
   11
   12
   ...
   This is because the evaluation for truth occurs at the end of the loop rather than at the beginning (as in the WHILE statement) and since $i$ is now not equal to 11 and never will be, it will continue until you press Ctl-C.

**Task 4 – Understanding the FOREACH statement**

The FOREACH statement (also known as a FOREACH loop) is a language construct for stepping through (iterating) a series of values in a collection of items.

The simplest and most typical type of collection to traverse is an array. Within a foreach loop it's common to run one or more commands against each item in an array.

The following shows the foreach syntax:

\[
\text{foreach ($<\text{item}> \text{ in } $<\text{collection}>\{}<\text{script\_block}>\}
\]
1. You should start this task by reviewing the help page for about_foreach
   MSH> \text{ get-help about_foreach}
2. After reviewing the help create a FOREACH loop that echoes the numbers 1 through 10
   MSH> \text{ foreach($number in 1,2,3,4,5,6,7,8,9,10 ) \{} \text{$number} \}
3. Create a FOREACH loop that echoes only the name of every file in the current directory
   MSH> \text{ foreach( $file in get-childitem ) \{} \text{$file.name } \}

**Task 5 – Understanding the FOR statement**

The FOR statement is very similar to the FOR statement in C or C#.

1. You should start this task by reviewing the help page for about_for
   MSH> \text{ get-help about_for}
2. After reviewing the help create a for loop that counts from 0 to 10
   MSH> \text{ for ( $i = 0; $i -le 10; $i++ ) \{} \text{$i} \}
3. Create a loop that counts from 0 to 10 by 2
   MSH> \text{ for ( $i = 0; $i -le 10; $i += 2 ) \{} \text{$i} \}

**Exercise 6 – Handling Errors**

*In this exercise we will describe the various mechanisms for discovering and handling errors that occur while using the MONAD shell.*
Task 1 – Retrieve Errors

In this section, we will cause errors to occur and retrieve those errors by using the $ERROR variable. The $ERROR variable is an array variable that contains all of the errors that occur while running Cmdlets or other language elements. $ERROR does not contain errors that are generated by native Win32 applications.

1. Create an Error by dividing 1 by $null
   MSH> 1 / $null
   Attempted to divide by zero.
   At line:1 char:4
   + 1 / <<<< $null

2. Retrieve the error information from the $ERROR array variable. This array variable is an inverted stack where the most recent errors are the lowest offset, so the last error that occurred is $ERROR[0].
   MSH> $error[0]
   Attempted to divide by zero.
   At line:1 char:4
   + 1 / <<<< $null

3. Discover the structure of the error by using the get-member Cmdlet
   MSH> $error[0] | get-member

Task 2 – Recognizing terminating and non-terminating errors

In general, the MONAD shell tries to continue to execute even in the face of failure. However, there are cases where you want the execution to halt when a problem arises. Because of this, we have created a variable whose value controls the behavior of what the shell does when an error occurs. The following table shows what the values and behaviors are of ErrorActionPreference

<table>
<thead>
<tr>
<th>Value</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>Continue</td>
<td>When ErrorActionPreference is set to &quot;Continue&quot;, the shell will report an error but keep executing if possible. This is the default behavior or the behavior when not set to one of the four valid values.</td>
</tr>
<tr>
<td>Inquire</td>
<td>When ErrorActionPreference is set to &quot;Inquire&quot;, the shell will interact with the user to determine whether it should continue to execute.</td>
</tr>
<tr>
<td>Stop</td>
<td>When ErrorActionPreference is set to &quot;Stop&quot;, the shell will report an error and stop executing.</td>
</tr>
<tr>
<td>SilentlyContinue</td>
<td>When ErrorActionPreference is set to &quot;SilentlyContinue&quot;, the shell will not report an error and keep executing if possible. Errors that occur with this setting still will populate the $ERROR variable.</td>
</tr>
</tbody>
</table>

1. Execute a foreach loop that causes a dividebyzero error
   MSH> foreach ( $n in 2,1,0,1,2 ) { 1 / $n }
   Note that even though there was an error (when 1/0 was executed), the loop continued.

2. Execute a foreach loop that causes a dividebyzero error but does not report it
   MSH> $ErrorActionPreference = "SilentlyContinue"
   MSH> foreach ( $n in 2,1,0,1,2 ) { 1 / $n }

3. Execute a foreach loop that causes a dividebyzero error and stops when the error occurs
   MSH> $ErrorActionPreference = "Stop"
   MSH> foreach ( $n in 2,1,0,1,2 ) { 1 / $n }

4. Execute a foreach loop that causes a dividebyzero error and inquires whether execution should continue
   MSH> $ErrorActionPreference = "Inquire"
Task 3 – Using Trap

Trap allows you to handle an error and compensate for it from within your script or function. Trap is similar to the try-catch behavior of C#. Trap requires a scripts block and optionally an Exception type. The following shows the syntax for trap

```
trap [ [ExceptionType] ] { <script block> }
```

If the ExceptionType is not provided, all exceptions will be trapped. It is also important to note that traps only apply to the current script block. From the command line, this means that a trap will only apply to the current line. In Exercise 8, we'll apply this to functions and scripts.

The script block can execute arbitrary code. If the `continue` keyword is used, control is passed back to the script in which the error took place and no error is reported. If the `break` keyword is used, the current script block is exited and an error is reported. If no keyword is used, an error is reported and the next instructions in the script block are executed.

1. Create a one line script that prints "hi", causes a divide by zero error and then prints "bye"

```
MSH> "hi"; 1/$null; "bye"
hi
Attempted to divide by zero.
At line:1 char:9
+ "hi"; 1/$ <<<< null; "bye"
bye
```

2. Add a trap statement to the previous script that prints "got an error" in the case of an exception.

```
MSH> trap { write-object "got an error" } "hi" ; 1/$null; "bye"
hi
got an error
Attempted to divide by zero.
At line:1 char:48
+ trap { write-object "got an error" } "hi" ; 1/$ <<<< null; "bye"
bye
```

3. Change the trap statement to silently continue when it handles an error

```
MSH> trap { continue } "hi" ; 1/$null; "bye"
```

4. Change the trap statement to halt the script when it handles an error

```
MSH> trap { break } "hi" ; 1/$null; "bye"
```

5. Change the trap statement to handle only DivideByZeroExceptions.

```
MSH> trap [Dividebyzeroexception] { write-object "divide by zero error"; continue } ; "hi"; 1/$null;"bye"
hi
divide by zero
bye
```

Exercise 7 – Formatting Output

In this exercise we briefly discuss the formatting capabilities of the Monad Shell. Specifically, we will investigate the format Cmdlets and the arguments that are possible. We will discuss the format.mshxml files, but not change them.

In this exercise, you will use the different formatting commands and create specialized formatting in the shape of tables and lists.
Task 1 – Selecting your format look

Monad provides 2 main formatting views; table and list.

1. Review the help for format-table and format-list
   
   MSH> get-help format-table
   MSH> get-help format-list

   As you can see from the help files, it is possible to create a fairly customized view of data.

2. Create a table of process objects selecting only the name, handlecount, processid and virtual memory size
   
   MSH> get-process | format-table name,handlecount,processID,virtualmemorysize

3. Create a list with the same properties as #2
   
   MSH> get-process | format-list name,handlecount,processID,virtualmemorysize

Task 2 – Modifying the formatting

It is possible to provide more than just the property name to the format-table command. The property can be expressed as a hash table. Certain keys have specific meaning to the format command - the following describes the keys and their meaning:

<table>
<thead>
<tr>
<th>Key</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>width</td>
<td>The width of the column in character</td>
</tr>
<tr>
<td>alignment</td>
<td>right, left or center column alignment</td>
</tr>
<tr>
<td>expression</td>
<td>A script block that returns a value to be used in the column. The $._ variable is available as the current object</td>
</tr>
<tr>
<td>label</td>
<td>The string to be used as the column header label</td>
</tr>
<tr>
<td>formatstring</td>
<td>The format string to be used to present the result</td>
</tr>
</tbody>
</table>

1. Create a table of process objects with two columns, processname and handlecount where the label is "handle" and the handlecount values are centered
   
   MSH> get-process | format-table processname,@{ label = "handle"
   expression = { $._.handlecount } |
   alignment = "center" }

Exercise 8 – Creating Functions and Scripts

_In this section we will start to put the previous 7 exercises together so we can start taking advantage of the full power of the shell._

In this exercise, you will learn how to create functions and scripts and also understand how Monad provides input and how to allow your functions and scripts to accept arguments and piped input.

Task 1 – Declaring Function Names

A function is a named block of code that you can reference within your MSH commands. When you call the function name, the code within the function runs as if you had typed in the function's code block directly. A function can accept import values that you specify as arguments when you call the function
or accept values that are passed to the function through the current pipeline. The function returns values that can then be assigned to variables or passed to other functions or Cmdlets.

1. Review the help for functions
   ```powershell
   MSH> get-help about_function
   ```

2. Create a function that returns the sum of 2 numbers
   ```powershell
   MSH> function add2 { $args[0] + $args[1] }
   ```

3. Use the function
   ```powershell
   MSH> add2 3 6
   9
   ```

**Task 2 – Declaring Parameters and Dealing with Arguments**

1. Create a filter that returns the property name (as an argument called prop) of the objects it receives
   ```powershell
   MSH> filter get-prop ($prop) { $_.$prop }
   ```

2. Use the filter
   ```powershell
   MSH> get-process | get-prop id
   2728
   2188
   268
   2612
   2756
   880
   4072
   1772
   ```

**Task 3 – Functions and Input**

1. Create a function that takes an array of integers as input and multiplies each object by 2
   ```powershell
   MSH> function mult2 { foreach( $number in $input ) { $number * 2 } }
   MSH> 1,2,3,4 | mult2
   2
   4
   6
   8
   ```

2. Create a function that takes an array of integers as input and multiplies each object by the supplied argument
   ```powershell
   MSH> function multx { foreach( $number in $input ) { $number * $args[0] } }
   MSH> 1,2,3,4 | multx 3
   3
   6
   9
   12
   ```

**Task 4 – Scripts**

A script allows you to consolidate commands, functions, filters and the other constructs in a file. There is one specific issue that scripts have with regard to accepting named parameters. If you wish a script to accept parameters by name, you must use the param statement as the first executable line of your script.

1. Create a script that has two parameters; “firstparam” and “secondparam”, each of these parameters should be echoed on its own line (the following is an example)
MSH> get-content script.msh
param ( $firstparam, $secondparam )
"Firstparam is $firstparam"
"Secondparam is $secondparam"

2. Use the script

MSH> ./script -first hi -second there
Firstparam is hi
Secondparam is there

MSH> ./script hi there
Firstparam is hi
Secondparam is there

Exercise 9 – Solving Problems

This is the opportunity for the student to apply all of the previous exercises

Task 1 – Write a function

that returns a sum of the arguments received:
1. Ensure that your function will ignore non-numbers
2. Be sure that your tests include positive, negative and decimal numbers
3. Check your function with the example at end of this document

Task 2 – Write a function

that returns the objects piped to it in reverse order. So if 1,2,3 were piped in, 3,2,1 would be returned.
1. Clue: Be sure that you always are working with an array, even when working with a single value
2. Clue: Remember that $input holds all the input piped to a function

Task 3 – Write a filter

that takes a regular expression as an argument and passes through those strings that are piped into
the filter that match the regular expression.
1. Clue: Review the help on about_comparison_operators
Lab Summary

In this lab you performed the following exercises.

- Learned how to use simple shell commands
- Learned how to create expressions in Monad
- Learned how pipelines transport objects from command to command
- Used the various statements available in the Monad shell
- Learned how Monad handles errors and how to handle errors in script
- Learned how to define your own formatting of the Monad shell output
- Learned how to create functions, filters and scripts

Congratulations! You now should have a good idea how expressive and powerful the Monad shell is. We encourage you to continue to use the Monad shell in this lab. The on-line help and the lab proctors should be able to answer any questions you might have about the shell.
Solutions

Task 1

function sum {
    $total = 0
    foreach ( $number in $args )
    {
        trap { write-host ignoring $number; continue }
        $total += $number
    }
    $total
}

Task 2

function rev {
    foreach ( $ in $input )
    {
        $r = @(n) + $r
    }
    $r
}

Task 3

Filter match {
    if ( $_ -match $args[0] )
    {
        $_
    }
}