

# MSH LANGUAGE QUICK START

## Arithmetic Operators (also see Unary and String operators)

+	addition, concatenation
-	subtraction
*	multiplication, string repetition
/	division
%	modulus

## Array Comparison

Return all elements equal to 3: 1,2,3,5,3,2 -eq 3

Return all elements less than 3: 1,2,3,5,3,2 -lt 3

Test if 2 exists: if (1, 3, 5 -eq 2) ...

## Other operators: -gt, -le, -ge

## Arrays

"a","b","c"	array of strings
1,2,3	array of integers
@()	empty array
@(2) or ,2	array of 1 element
1,(2,3),4	array within array
\$a[5]	sixth element of array*
\$a[2][3]	fourth element of the third element of an array

\* NB: Arrays are zero-based.

## Assignment Operators

=	Assigns a value to a variable
+=	Adds a value to a variable
-=	Subtracts a value from a variable
*=	Multiplies a variable by a value
/=	Divides a variable by a value
%=	Performs a modulus on a variable

## Associative Arrays (Hashtables)

\$hash = @{ }	Create empty hashtable
\$hash.key1 = 1	Assign 1 to key "key1"
\$hash.key1	Returns value of key1
\$hash["key1"]	Returns value of key1
\$hash.key1={cmd}	Assign code block to key1

\$hash.key1(1,2) Run code block in key1 with parameters 1,2

## Boolean Values

\$true = 1 -eq 1	Assigns True to \$true
1,2,3 -and \$true	true
\$() -and \$true	false
@() -and \$true	false
@(1) -and \$true	true
"" -and \$true	False
"word" -and \$true	True

## break (scripting)

The "break" commands exits a loop. Example:

```
while (1)
{
    $a = something
    if ($a -eq 1) break;
}
```

## Command Expansion Operators

\$()	Returns null
\$(1)	Returns 1
\$(1,2,3)	Returns an array : 1,2,3.
@(get-alias;get-process)	Executes the two commands and returns the results in an array

## Comments

```
# This is a comment.
$a = "#This is not a comment..."
$a = "something" # ...but this is.
```

## Comparison Operators

-band, -bor	bitwise and, bitwise or
-match, -notmatch	regex pattern matching
-like, -notlike	globbing pattern matching
-eq, -ne	Equal, Not equal
-gt, -ge	Greater than, greater or equal
-lt, -le	Less than, less or equal
-is	compare types (1 -is int)

Case Insensitive variants:

-imatch, -inotmatch, -ilike, -inotlike, -ieq, -ine, -igt, -ige, -ilt, -ile

## continue (scripting)

The continue statement continues the next iteration of a loop without breaking out of it. Example:

```
while (1)
{
    $a = something
    if ($a -eq 1) (continue)
# This line is not reached unless $a == 1
}
# This line is never reached.
```

## Dot Sourcing

Dot sourcing allows running functions, script blocks, and scripts in the current scope rather than a local one. Example:

```
. MyFunction
```

If **MyFunction** sets a variable, it is set in the current scope rather than the function's local scope.

## Escape Character and Escape Sequences

The MSH escape character is the backwards apostrophe, or ```. To make a character literal, precede it with ```. To specify a ``` use ````.

Special escape sequences

```
`0    (null)
`a    (alert)
`b    (backspace)
`f    (form feed)
`n    (new line)
`r    (carriage return)
`t    (tab)
`v    (vertical quote)
```

## Execution Order

MSH attempts to resolve commands in the following order: aliases, functions, cmdlets, scripts, executables, normal files

## for (scripting)

```
[[:label]] for (initializer; condition; iterator) {}
```

Example:

```
for ($i = 0; $i -lt 5; $i++) {write-object $i}
```

## **foreach (scripting)**

[[:label]] foreach (identifier in pipeline or collection) {}

Example:

```
    $i = 1,2,3
foreach ($z in $i) {write-object $z}
```

## **functions (scripting)**

```
function MyFunction {
    write-object $args[0]
}
```

## **Filters (scripting)**

```
filter MyFilter {
    $_.name
}
```

## **if/elseif/else (scripting)**

```
if (condition) {...}
elseif (condition) {...}
else {...}
```

On the command-line, the closing brace must be on the same line as elseif and else. This restriction does not exist for scripts

## **Invoke Operator**

The & operator can be used to invoke the contents of an object. Example:

```
    $a = "get-process"
        &$a
$a = { get-process | pick-head 2 }
        &$a
```

## **Logical Operators**

! and -not Not a single value

-and And two values

-or Or two values

## **Method Calls**

Methods can be called on objects. Examples:

```
    $a = "This is a string"
        $a.ToUpper()
        $a.SubString(0,3)
        $a.SubString(0,($a.length/2))
    $a.Substring(($a.length/2), ($a.length/3))
```

Static Methods may not be called

\*\*The above is old. You can call statics easily - aka.

## MSH Variables

Variables are case insensitive and case preserving.

\$\$	contains the last token of last line input into the shell
\$?	Contains that success/fail status of the last operation
\$^	contains the first token of the last line input into the shell
DebugPolicy	The action to take when data is written via write-debug in a script or <b>WriteDebug</b> in a cmdlet or provider.
ErrorPolicy	The action to take when data is written via write-error in a script or <b>WriteError</b> in a cmdlet or provider.
HistorySize	Number of entries saved in the command history.
MSHCommandPath	The paths where .cmdlet and .cmdletprovider files may be found. This is the msh equivalent of the CMD.EXE \$PATH.
ReportErrorShowExceptionClass	Set to true indicates that the class name of the exception(s) displayed will be shown. Default at internal startup is false.
ReportErrorShowInnerException	Set to true indicates that the chain of inner exceptions should be shown. Each exception message will be indented from the previous message. The display of each exception is governed by the same options as the root exception, meaning that the options dictated by \$ReportShowError* will be used to display each exception. Default is false.
ReportErrorShowSource	Set to true indicates that the assembly name from whence the exception originated will be displayed. Default at internal startup is true.
ReportErrorShowStackTrace	Set to true indicates that the stack trace of the exception will be emitted. Default at internal startup is false.
ShouldProcessPolicy	The action to take when <b>ShouldProcess</b> is used in a cmdlet.

<b>ShouldProcessReturnPolicy</b>	<b>ShouldProcess</b> will return this setting
<b>StackTrace</b>	The current execution stacktrace - currently, this may be overwritten.
<b>VerbosePolicy</b>	The action to take when data is written via write-verbose in a script or <b>WriteVerbose</b> in a cmdlet or provider.
<b>\$_</b>	The current pipeline object, used in script blocks and where
<b>\$Args</b>	Used in creating functions that require parameters
<b>\$Error</b>	Objects which had an error occur while processing that object in a cmdlet.
<b>\$ErrorPolicy</b>	The action to take when data is written via write-error in a script or <b>WriteError</b> in a cmdlet or provider.
<b>\$foreach</b>	Reference to the enumerator in a foreach loop
<b>\$Home</b>	The users home directory; set to %HOMEDRIVE%\%HOMEPATH%
<b>\$Input</b>	Can aid in code blocks that are in the middle of a pipeline, (see code block)
<b>\$MshHome</b>	The install location of MSH
<b>\$MshHost</b>	Information about the current executing host
<b>\$OFS</b>	Output Field Separator
<b>\$StackTrace</b>	contains detailed stack trace information about the last error

## Object Properties

An object's properties can be referenced directly with the "." operator.

```
$a = get-date
$a.Date
$a.TimeOfDay.Hours
```

## Operator Precedence

In MSH, operators are evaluated in the following precedence:

() {}, @

!, [, .., &, ++ --, Unary + -, \* / %, Binary + -, Comparison

Operators, -and -or, |, > >>, =

## Redirection

The > and >> operators redirect command output to files. The > operator creates a new file or truncates an existing one, while the >> operator appends to an existing file. Example:

```
1,2,3 >foo.txt
5,6 >>foo.txt
return (scripting)
```

The return command exits the current script or function and returns a value.

Example:

```
function foo {
    ...
}
```

## Script Blocks

Commands and expressions can be stored in a script block object and executed later. Example:

```
$block = {get-process; $a=1}
&$block
```

## Scripts

MSH commands can be stored in and executed from script files. The file extension for MSH scripts is ".msh". Parameters can be passed to a script and a script can return a value.

Example:

```
$sum = MyAdder.msh 1 2 3
```

## Strings and String Operators

String constants:

```
"this is a string, this $variable is expanded"
'this is a string, this $variable is not expanded'
```

String operators

+	Concatenate two strings
*	Repeat a string some number of times
-f	Format a string
-replace	replace elements in a string

Examples:

```

MSH> "test" + "this"
testthis
MSH> "{0:M}" -f $(get-date)
June 02
MSH> $a = 1,2,3,4
MSH> $a
1
2
3
4
MSH> $OFS = ":"
MSH> "$a"
1:2:3:4
MSH> "This is a test" -replace "is","IS"
THIS IS a test

```

## Switch

```

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

$var = "word2"
switch -regex ($var)
{
"word2" {"Multi-match Exact " + $_ }
"word.*" {"Multi-match Exact1 " + $_ }
default {"Multi-match Default " + $_; break}
"w.*" {"Previous Break terminated the matching"}
}

```

```

$var = "word1","word2","word3"
switch -regex ($var)
{
"word1" {"Multi-match Exact " + $_ ; continue}
"word2" {"Multi-match Exact " + $_ ; continue}
default {"Multi-match Default " + $_; continue}
}

```

## Trap

Execute a block of code in a terminating error condition. Example:

```

function handler1 { write-host "Hi, I'm a trap handler" }
function handler2 { write-host "Hi, I'm a trap handler2" }
trap [System.Management.Automation.ExecutionFailedException]
{ handler2 ; continue }
trap [System.Management.Automation.ExecutionBreakOnErrorException]
{ handler1 ; continue }
get-content thisisabadfilename -errorp notifystop
set-location thisisabadlocation

```

## Unary Operators

++ Increment a variable  
-- Decrement a variable  
+ Indicate that a number is positive  
- Indicate that a number is negative  
[type]object cast object to type

```
@$a = [int]"3"@  
@$a + 3@  
@6@
```

## Variables

Format: `[$scope:]name`

Examples:

```
$a = 1  
$global:a = 1  
$local:a = 1  
$env:path = "d:\windows"
```

Scope may be either global, local or script

```
while (scripting)  
[:label] while (condition)  
{  
...  
}
```