ELTE INFORMATIKA

Computer Systems

9. Powershell

Review

- Computers, information, number representation, code writing, architecture, file systems
- Base commands, foreground and background processes
- I/O redirection, filters, regular expressions
- Variables, command substitution, arithmetical, logical expressions
- Script control structures, sed, awk
- Batch, WSH
- PS overview

What comes today?

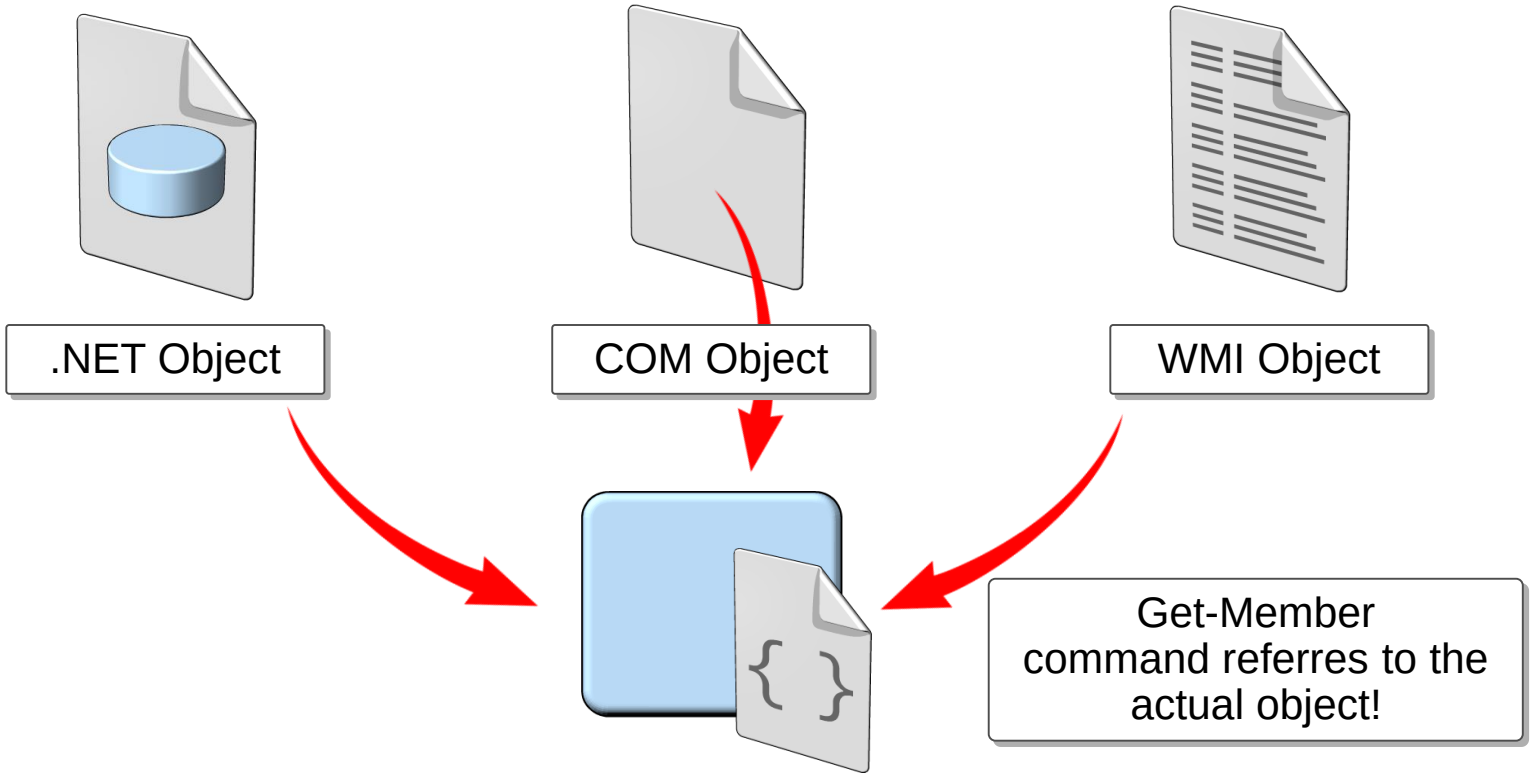
- PowerShell bases
- PowerShell language elements
- …

PowerShell is object oriented-everything is object

- get-date – result: y…november 20xx.
- „Pipe after pipe”: get-date|get-member
  - get-date object arrives to the input of get-member and as the output we get the properties and methods of the object
  - -inputobject is working at several cmdlet
  - „fradi” | get-member
- get-date|get-member –membertype method
  - We get the methods of get-date, similarly we can ask for the property-s, as well.
- (get-date).month, day, …ticks

NET, COM, WMI Objects as libraries

- PowerShell uses .NET natively. COM, WMI objects are handled directly.



Base PowerShell commands

- alias , writes out the defined abbreviations
  - Unix like commands
- gcm – Get-Command, writes out the commands
- echo – Write-Output, writes out to the screen, simple, pipe usage
  - Write-Host – [Console]::WriteLine, foreground, background settings
- Get-Help – short description of the commands
  - Update-Help – downloads and installs the help!
  - Get-Help –full Write-Host
  - set-alias gh get-help
- ps – Get-Process, writes out running processes
  - Sleep – Start-Sleep, waiting

Drive vs. Other datasource

- dir, ls – Get-ChildItem, directory content
  - Get-ChildItem c:\users\test*.* -include *.c,*.cpp # only c,cpp extensions
- Get-PsDrive – Powershell datasources
- Cd change directory – Set-Location command
  - Cd c:
  - Cd hklm:
  - Cd alias:
    - Dir- lists out the content of the actual drive, datasource
- Get-Location – pwd alias command

Important file handling commands

- New-Item – file or directory creation
  - New-Item –itemtype directory appledir #similar to mkdir in dos !
- Copy-Item source destination [–recurse] # copy (Alias: cp)
- Remove-Item – deletes file, directory (Alias: rm, rmdir, del,…)
- Move-Item – file, directory moving (Alias: mv)
- Rename-Item – renaming (Alias: ren)
- Get-Item – file, directory, reg.key result
  - Get-item $(c:\users).LastWriteTime, Get-item hkcu:\software |get-member
- Test-Path file or directory or reg.key #does it exist?

The structure of PowerShell parameters

- PowerShell command structure: Verb-noun
  - E.g: Get-Command
- Typical usage of parameters: -name value
  - A value can be: number, text or date
  - E.g.: Get-Command –Verb write
- History – F7 previous commands
  - Up-arrow, previous command
- Profile in directory :Documentums\WindowsPowerShell
  - Microsoft.PowerShell_profile.ps1
  - profile.ps1 - It is executed only by ISE!

PowerShell variables

- $name=value, $ symbol during definition as well
  - E.g.: $f=„RealMadrid”; echo $f
- Several commands can be written in the same line separating them by ;
- Important base types:

Data Type	Meaning	Example
[int]	Integer (32bit)	**-273, -1, 0, 10, 42**
[byte]	8-bit, byte	**0, 1, …, 254, 255**
[boolean]	Logical	**$false, $true**
[char]	Character(16 bit uni.)	**a, b, c, 1, 2, 3, !, #**
[string]	Text	**“FTC”**
[single]	32 bits real	**2.3e-1, 3.1415,…**
[datetime]	Date, Time	**April 1, 2008**

Usage of PowerShell variables

- We can decide the types (typecasting):
  - [int] $d=6.2e-4; echo $d # 0, $d will be an integer
  - $s= [string] 65; echo $s # 65 as text
  - $s1=[string] [char] 65; echo $s1 # A
  - $i=[int] ”65”; echo $i # 65 as an integer
- If we do not give the type the interpreter makes a decision about it
  - $d=6.2e-4; echo $d # 0,00062, will be real

The definition of PowerShell variables with a command

- Set-Variable –Name apple –value „jonatan” –option constant
  - constant definition
  - We can give a description with parameter –description
  - Get-Variable apple
- Clear-Variable apple # apple variable exists, but it has not got any content.
- Remove-Variable apple # apple variable does not exist further

Visibility of variables I.

- Variables defined in an environment we can use in the environment and in the functions or scripts deriving from it!
  - If there is a variable with the same name in the function or script the local one can be seen as the default
- We can change it by using get-variable –scope
  - Get-variable apple –scope 0 # actual environment
  - Get-variable apple –scope 1 # parent environment
  - Get-variable apple –scope 2 # grandparent environment
  - etc.

Visibility of variables II.

- General definition form:
  - $[scope:]name or ${name}
  - If the scope is not given the variable can be used in the actual script or function.
  - A scope may be: global, local, private, script.
    - $global:x=5 # x variable will be global it can be seen anywhere
    - $script:y=6 # it can be used in the whole script
    - $local:z=„MTK” # it can be seen localy and in child blocks
    - dir $env:ProgramFiles # element of the env drive

    You can handle environment-variables with it!
    - $env:Path += „;d:\tmp”

Arithmetical operations in PowerShell

- +,-,*,/,% (remaining)- base operations
  - You do not have to use a special command like expr in UNIX shell!
  - $a= 32*3; echo $a # 96
  - $a=„apple”; $f=„tree”; $c=$a + $f; echo $c #appletree
  - $a= „125” + „2”; echo $a # 1252!
  - $a= 12 + „4”; echo $a # 16
    - It converts automatically „4”
- Assigning: =, +=, -=, *=, /=, %=
- Post incrementation, decrementation: $a++, $b--
- Bit operations: -band,-bor,-bxor,-bnot,-shl,-shr

More operations

- Behind PowerShell is .NET FrameWork.
  - Each of the types can be used: double, decimal etc.
  - Not only the base types
  - Example: [System.IO.DirectoryInfo]$home=Get-Item D:\home
- The operator to reach a static property or method is ::
  - [DateTime]::Now # actual date
- The whole Math class is available
  - [math]::pi
  - [math]::sin(2), etc.
- Conversion: [system.convert]::toint32(„32”)
- etc. , ….Net FrameWork whole library usage

PowerShell variable summary

- $team=„RealMadrid”
- Automatic type deciding, but it can be overwritten
  - [int]$a=„apple” # it is an error!
- All of the base operations are available! + .NET
- Interesting: $b=$team*5 #ot is ok,”RealMadrid five times”
  - $c=5*$team #error!!, „RealMadrid” won’t be integer!!!
- Variable definition with command
  - Set-Variable –name a –value „pear” –option constant
- Text command execution operator:&
  - $dir=„dir”; &$dir

Variable substitution

- $a=„apple”
- „$atree” # result is empty
- „${a}tree” # result is appletree
- „red$a” #redapple
- $ character neutralization : `
  - „`$a variable value: $a”
  - In a regular expression use \ symbol for neutralization!
- There is no special form for command substitution!
  - $dirlist=dir # There is no need for using the `dir` form!

Texts, substitution

- Between „” the variables are replaced with their values!
- Between ,’ the variables are not replaced:
  ,It is not replaced: $a’
- Similar to Unix: ”echo ’$i ends’”
  - Here (too) $i is replaced!!!!
- In Powershell there is no input redirection (<,<<)
- Instead of it there is multiline text: @” …can be several lines… ”@
  - Between them the variabes are replaced!
  - @’ …. There can be several lines …. ’@ # no variable replacement

PowerShell arrays I.

- Often used naming of variables are: scalar, contains one data, e.g.: $data=„apple”
- Contains several scalar data: array
- Definition: $myarray="apple",„pear",„peach”
  - Entire formula: $myarray=@(”apple”,”pear”,”peach”)
  - Elements start with 0 index!
    - echo $myarray[1] # pear
    - echo $myarray[1..2] # pear peach
  - An element can be not only a simple scalar value but an array too: $myarray[2]=@(2,3,4); echo $myarray[2][1] # 3

PowerShell arrays II.

- An array is an object in fact. The length of an array can be reached by the Length property.
  - echo $myarray.Length
- Adding a new element: $myarray+=6;
- Writing out each elements: $myarray (the same as: echo $myarray
- We can concatenate arrays with: + symbol
  - $myarray2=2,3,4,5
  - $myarray2+=$myarray
  - echo $myarray2[3] # 2

PowerShell array operations

- -contains : containing (-notcontains)
  - 1,2,3,4 –contains 3 # True
- -eq, -ne Results is all of the elements which are equal, (not equal) with a given value
  - 1,2,3,4 –ne 3 # 1,2,4
- -lt, -gt Result is all of the elements which are smaller, (greater) to a given value
  - 1,2,3,4 –lt 3 # 12
- -le, -ge Smaller equal, greater equal
- -join, -split, -csplit (case split, small-capital letter)
- etc.

PowerShell associative arrays

- $aarray=@{„key”=„value”; …}
  - $at=@{a=4;b=5} # Among the elements ; !!!!!!
- Reaching elements: $at[a] or $at.a
- Element assignment: $at[a]=10
- Adding a new element: $at+=@{c=11}
- To write out the associative array: $at

```
PS C:\home\ps> $at
Name                    Value
----                    -----
a                       10
b                       6
c                       11
```

.NET FrameWork arrays

- System.Collections is the namespace of different data structures:
  - $a=new-object system.collections.arraylist
  - $a1= [system.collections.arraylist] (2,3,4)
  - $a1.add(10)
  - $a1.contains(3) # true
  - $a1.insert(2,20) # after 3 will be the place of 20!
  - $a1.sort()
  - etc.

Branches in PowerShell

- Comparing operators similar to ones at arrays.
  - -eq, -ne,-gt,-lt, -le, -ge
  - -not, -and, -or,-xor logical negotiation, and, or
  - At texts: -ceq, small, capital letters differ, -ieq are not different,
  - -like *,?,[ab.] characters, -match reg. exp. usage
- If instruction:
  - if (kif) {instructions} [elseif (instruction1)] else {instruction}

```
$a=3
if ($a -gt 2)
 { Write-Host " $a is greater than 2." }
else
 { Write-Host „is not greater than 2."}
```

Multibranches

- switch instruction – similar to .net languages
  - In subbranches there is no need to use break

```
# switch example
#
# Reading instruction: read-host
$a=Read-Host -prompt „Write your favourite fruit"
switch ($a)
{
  "apple"        { "a value: "+$a }  # write-output
  „peach"        { "a value: "+$a }
  „grape"        { "a value: "+$a }
  „plum„         { "a value: "+$a }
  „pear"         { "a value: "+$a }
  default        { "a is unknown: "+$a}
}
```

PowerShell Loops I.

- for – very similarly used as in C,… etc.
  - You always have to use {} around cycle seed
  - e.g.: for($i=0;$i –lt 5;$i++) {echo $i}
- foreach($i in array) {cycle seed}

```
#
$t=2,3,4,5
foreach($i in $t)
{
        write-host $i
}
```

PowerShell Cycles II.

- foreach – Foreach-Object, filter
  - Several commands in the same line: ;
  - One command in several lines: `
  - Similar to AWK: begin, end block

```
# continue a text in a new line: ` character, important!!!
#
get-process|foreach-object `
 -begin { Write-Host „I started to execute Get-Process!" } `
 -process { write-host $_.name -foreground green } `
 -end { Write-Host „I finished Get-Process!" }
```

PowerShell Cycles III.

- While cycle: like in C, do…while expression.– while true
- Until cycle: do …until expression.; while it is false!

```
$a=5
while ($a -gt 0)
{
 write-host $a
 $a--
}
```

```
do
{
 „It is executed at least once!"
 write-host $a
 $a--
} while ($a -gt 0)
```

```
$a=0
do {
 „It is executed at least once!"
 write-host $a
 $a++
} until ($a -gt 3)
```

Script structure – Foreach example

- Not only a foreach block can contain 3 blocks
- A script also can contain 3 blocks:
  - Begin { instructions} # It is executed once at the beginning of the script
  - Process { instructions } # It is executed for each of the „pipe” objects!
  - End { instructions} # It is executed once at the end of the script
- It is a general form of scripts processing on pipe data!
  - E.g.: pipeparam.ps1

Example: pipeparam.ps1

```
BEGIN {
  foreach( $i in $Args )
            { Write-Host "In begin block the parameters are $i" -F red }
   Write-Host "In begin block the pipeline: $_" -Fore red
}
PROCESS {
   foreach( $i in $Args )
            { Write-Host "In process block the parameters: $i" -F white }
   Write-Host "In process block the pipeline: $_" -F white
   $_.GetType().Name # The name of the object type
}
END  {
   foreach( $i in $Args )
            { Write-Host "In end block the parameters: $i" -F green }
   Write-Host "In end block the pipeline: $_" -F green
}
```

Windows PowerShell

```
PS C:\> 34, 35 | .\pipeparam.ps1 3 4 5
In begin block the parameters are 3
In begin block the parameters are 4
In begin block the parameters are 5
In begin block the pipeline:
In process block the parameters: 3
In process block the parameters: 4
In process block the parameters: 5
In process block the pipeline: 34
Int32
In process block the parameters: 3
In process block the parameters: 4
In process block the parameters: 5
In process block the pipeline: 35
Int32
In end block the parameters: 3
In end block the parameters: 4
In end block the parameters: 5
In end block the pipeline: 35
PS C:\>
```

ELTE INFORMATIKA

Thank you!