# Module 1 – Powershell begining (29.06)

--scripting language

--PShell ISE (Integrated Scripting Environment)

.net framework v4.5

Windows 7 SP1

dovoljno za pshell 5.0

-------------------------------------------------

Pshell shell (prozor powershell-a)

Command-line interface (CLI)

home - pocetak reda

end - kraj reda

CTRL + R - history search

CTRL + SPACE autocomplete

up/down arrows - komande koje si pustao

get-history - sve komande korišćene

invoke-history -id br. - pusti komandu iz istorije

-------------------------------------------------------

ISE demo

-blue pane - command pane

-white pane - scripting pane

npr get-process daje ti sve procese iz Task Managera, dopunis sa get-process -Name i mozes da biras ime procesa

write-host "Hello world" - napisi nesto na ekranu

-foregroundcolor - biras koja je boja teksta

-backgroundcolor - biras kojom ce bojom biti obojena pozadina teksta

-pored play dugmeta ima dugme kojim se run deo teksta

- show commands pane, desno ili skroz gore u toolbaru

omogucava da se pusti komanda kroz gui

# is a comment sign

<# #> comment block

#region #endregion - creates a block of code

tools -> options -> create a theme for both panes

# + control space -> history of commands

# Module 2 – Commands (30.06)

## External commands

-netsh.exe, reg.exe, sc.exe...

-get-command

- difficult to discover (hard to find them if you dont know them)

- output is text

## Commands in Pshell

- they are called cmdlet - commandlet

- structure

verb-noun + command parameters naming scheme

primer 1.

get-process

primer 2.

remove-item -path C:\temp -force

**verb-noun -parameter parameter value -switch parameter (its either on or off)**

## cmdlet syntax

-[-Filter <string>] -optional parameter []

- <string> - data to pass in

- [] in <> - list of data

parameter sets - different ways to use command

primer 1. get-command stop-process -syntax

positional parameter - place for parameter, you do not have to write its name, just value of it

get-process -name xyz = get-process xyz

## common parameters

-gives you information about the code, helps with errors etc.

-verbose - gives you detailed information about action

-erroraction (with parameters)

primer 1. stop-process chrome -verbose -erroraction silentycontiue

## risk mitigation parameters

-many cmdlets (not all) -whatif (wi) and -confirm (cf)

-deleting, changing data

primer 1. stop-process -name powershell\_ise -whatif or -confirm

## Command termination and line continuation

termination character - semicolon ;

line continuation >>

## Aliases

-can be created or changed by user

-do not change built in aliases

primer 1. get-childitem -Path C:\ ... samo gci\dir\ls

some of alias cmdlets

get-alias -name xyz - get definition of xyz alias

export\import-alias -path ... - export or import aliases

new\set-alias -name xyz -value xyz - To create a new alias, use Set-Alias or New-Alias. To change an alias, use Set-Alias.

# Module 3 – Objects (03.07)

Object is a structured piece of data. Combines bunch of information and abilities in one piece of data.

Set of information and set of action about some data.

properties (information) etc about TV (is it on?; volume?; channel?;)

methods (actions) etc about TV - buttons on remote control (power on/off, volume up/down, channel up/down)

code in the background we can tell object to do

every piece of data you get from cmdlet is an object, every object has properties and methods.

properties and methods are called members.

every individual object is called instance of a type.

---------------------------------------------------------------------------------------------------------------------

## Cmdlets to define properties and methods on object

get-member properties or methods

primer 1.

variable $date = get-date (date and time today)

$date - prints variable

Get-member -inputobject $date - properties and methods about variable

primer 2.

$date | get-member

Primer 1. = Primer 2.

primer 3. -daje samo properties

$date | get-member -membertype properties (|format-table - da se output prebaciu tabelu)

daje ti typename: (library).datetype (pr. System.Datetime)

primer 4. - get properties

$date.month - daje ti koji je mesec

$date.dayofyear - dan u godini

primer 5. -daje samo methods

$date | get-member -membertype methods

primer 6. - use methods

$date.addmonths(2)

$date.isdaylightsavingstime()

primer 7.

$date.gettype()

-get the type of date

$date.GetType() | GM -MemberType property |format-table

-get the properties of type of variable datettime

$date.gettype().FullName

-get the full name of type of variable

$date.gettype().FullName | gm -membertype methods|Format-Table

-get methods of fullname of type of variable

$date.gettype().FullName.toupper()

-make a fullname upper case

# Module 4 – Variables (05.07)

## Variables - definition

- unit of memory

- defined with $

- single or collection of objects

- naming can use spaces and special characters but its not recommended

- not case sensitive

two kinds of variables:

- automatic (built in) and user defined

User-defined variables

$variablename = <value>

-exist only in current session

## cmdlets

new-variable name -value value

-same as $variable = value

clear\remove -variable -name name

set-variable -name name -value newvalue

-same as $variable = newvalue

get-variable -name m\*

-find every variable that start with m

## Constant variables

-variables are set on creation (cant use =)

-cant be deleted or changed

primer1. new-variable -name pi -value 3.14 -option constant

## Read-only variables

-cant use =

-cant be easily deleted (must use remove-variable with -force)

-cant be easily changed (must use set-variable with -force)

## Parsing modes

-expression and argument mode

-argument mode pshell expects data from user, inside "" its argument

-$ telling pshell it is expression mode, & force into argument mode

2+2 - expression mode

write-output -inputobject 2+2 - argument mode

write-output -inputobject (2+2) - expression mode

invoke-expression to execute command stored inside of variable

## String-subexpressions

#$()

-$(command)

primer 1. write-host "the date is: $(get-date)" -foregroundcolor green

primer 2.

$service = get-service alg

write-host "service: $($service.displayname)" -foregroundcolor green

## Strings

-difference between " and ', they are the same but upon creation pshell decides if it is expandable or litteral

- "" - an expandable string - cita varijable kao promenljive

- '' - a litteral string - cita varijable bukvalno

## Here strings

-copy and paste from excel or something like that

@"

string

"@

## Type Operators

-another way to do type comparison

normal typecasting primer 1. ([int]"5").gettype()

primer 2. ("5" -as [int]).gettype()

data -is [datatype] - questions if data is asked datatype

## Types of data in pshell

-every object has a type, it is declared when data is created, it can be converted into other types

-it is not required to explicitly state datatype

general types: boolean, char, int, long, double, enum (set of named constants), array

gettype().fullname -gets type fullname

## typecasting

primer1. [int32]1.6 =>2

primer2. $number = [int]"000123"

$number bice 123

strongly and weakly type variables

weak $var1 =[int]1.3

$var1 bice 1

$var1 = 1.2

$var1 bice 1.2

strong [int]$var1 = 1.3

$var1 bice 1

$var1= 1.2

$var1 bice 1

## Escape character `

-main use remove special quality from special character

-can create special characters with it

-case sensitive

primer 1.

$a=123

"`$a is $a"

$a is 123

primer 2.

get-process -name `

powershell

## Stop parsing --%

-stop pshell from interpreting input (special symbols)

# Module 5 – Help System (06.07)

## Get-help

get-help cmdlet

-gives you extensive help for command you input

get-help cmdlet -full | more

-gives you info but sorted in pages

get-help cmdlet -showwindow

-information but in a separate window

get-help cmdlet -online

-online help

get-help cmdlet -parameter parameter name

-help on specific parameter

## Conceptual help

get-help about\_nesto

-help about something

## updatable help

-some data of help must be downloaded

-update-help (run it as admin)

it is possible to choose where to save help data, save-help -destitnationpath \\fsi for example

#.<help keyword>

#<help content>

primer keyword .synopsis, .description, .example

Primer1.

<#

.Synopsis

This function writes stuff in green

.Description

This fumction writes stuff in green

.Example

MyFunction -message "Powershell rocks"

.Parameter message

This is the message we write in green

#>

function MyFunction

{

param($message)

write-host $message -foregroundcolor green

}

get-help myfunction -ShowWindow

-help can be put above the code, bellow the code, inside curly braces

# Module 6 – Functions (02.08)

## Functions

-reusable block of pshell code

-can accept parameter and return outputs

function name

{

param() - your input parameters

what you want to be done

}

primer 1.

function myfunction

{

param($procname, $color="Green")

$proc = Get-Process $procname

write-host $proc.id -ForegroundColor $color

}

## Script block

-statement between {}

-package code to be reused

-bind keys to run a specific code

$sb={write-host "this is script block" -foregroundcolor cyan }

Set-PSReadLineKeyHandler -chord shift+ctrl+b -ScriptBlock $sb

-through pshell not pshell\_ise

## Remoting in a shell

## hostname

-gives you the name of machine youre on

enter-PSSession -computername

-gives you access remotely to another machine

exit-PSSession

-exit machine

invoke-command

new-pssession

# Module 7 – Providers (02.08)

## Providers in Pshell

data store

-active directory or sql server

powershell drive is entry point to data store from provider

## Drive cmdlets

get-psdrive

-all drives and details about them

-new-psdrive

create custom drive, goes away at the end of a session

primer 1.

New-PSDrive -name reggieclasses -PSProvider registry -root hkey\_classes\_root

cd reggieclasses

-to access drive

ctrl+c

-to stop executing command

Remove-psdrive

-removing drive, first you'll have to stop using it

## Item and Item property cmdlets

get-command -noun item or something else

-gives you every command which has an input word as a noun

get-item

get-itemproperty

## Content cmdlets

get-content

set-content

clear-content

primer 1.

--Adding content

get-date | set-content C:\Users\B100450\Desktop\test\date.csv

"this was today's date" |Add-Content C:\Users\B100450\Desktop\test\date.csv

--getting content

get-content C:\Users\B100450\Desktop\test\date.csv

02.08.2023. 15:40:15

this was today's date

--removing content

Clear-Content C:\Users\B100450\Desktop\test\date.csv

Get-Content C:\Users\B100450\Desktop\test\date.csv

## Path cmdlets

join-path

-creating path out of 2 variables

test-path

-testing if a path exists

split-path

-spliting path into parts

split-path "path" -parent\leaf

-gives you parent path or leaf part of path

convert\resolve-path

# Module 8 –Scripts (09.08)

## Basics

-reusable code in one file

-can be distributed to other machines

## Execution policy

Restricted

-cant run a script file

-default in 2008R2 or below

Allsigned

-run trusted scripts

Remotesigned

-Default in 2012R2 and more

-run all local script

-downloaded scripts must be from trusted source

Unrestricted

-runs all scripts

## Execution policy scope

-can come from various places

get-executionpolicy

-gives you your current policy

get-executionpolicy -list

-gives you list of all policies

set-executionpolicy -scope currentuser -executionpolicy xyz

-setting executionpolicy for you user and choosing which policy would be applied

Launching the script

## Script signing

-https://codesigningstore.com/how-to-sign-a-powershell-script

## Parameters and requier statements

-script param must be first in script

## Require statement

-special comment, #require -option (version, runasadministrator)

primer1. #requires -Version 3 (pshell 3 or up)

primer2. #requires -RunAsAdministrator (pshell 3 or up)

## command lookup precedence

full path (script.ps1) --> alias --> function --> cmdlet -->external commands

# Module 9 - Introduction to operators (16.08)

-used to compare values

- <,>,= are not used

- object on the left dictates the comparison, to be exact PShell convert the object on the right to the type of one on the left and then compares them

- no wildcards

-operators are case insensitive by default, could be case sensitive if you want( add c before command)

primer 1.

-eq => equat

-gt => greater than

-ge => greater or equal

-lt => less than

primer 2.

-Comparing values of status from service with string

$service = get-service alg

$service

#$service | Get-member -MemberType properties

$service.Status -eq "stopped"

## Wildcards in Operators

-like and -notlike

-by default it is CI

-case sensitive just add C (-clike)

\* - zero or any number of chars

? - exactly one char

[1az9] - any of given chars

[a-l] -given range of chars

primer 1.

"pear" -like "p\*"

"pear" -clike "P\*"

"abc" -like "???"

"abc" -like "[abc]"

"aCc" -like "[abc][abc][abc]"

## Regex

-match or -nomatch

-case insensitive by default

-case sensitive juste add c (-cmatch)

primer 1.

"Digit 5 is in this string" -match "\d"

-"\d" checks to see if a digit is in pattern

$Matches

-pre-defined variable which show all thing you found as a match in current session

## Arrays

-contains or -notcontains (if list contains something answer is true or false)

-case sensitive juste add c

-in or notin (if a value is in the list)

primer 1.

2,3,4,5,6 -contains 40

2 -in 3,4,5,6,71,2

(get-process).name -contains "notepad"

## Logical operators

-and, -or, -xor(one statement true and other is false)-exclusive or, -not or ! (to negate something)

Primer 1.

$true -and $true

$true -and $false

$service= get-service alg

$service.status -eq "running" -and $service.StartType -eq "manual"

$service.status -eq "stopped" -and $service.StartType -eq "manual"

$service.status -eq "running" -or $service.StartType -eq "manual"

$service.status -eq "stopped" -xor $service.StartType -eq "manual"

!($service.status -eq "stopped" -xor $service.StartType -eq "manual")

## Range and Multiplier Opeartor

4..10 moze i 4 .. -20

-generate inclusive list from 4 to 10

kb or mb

-numeric multipliers

-number of bytes in kb, mb, gb etc

# Module 10. Understanding the Windows PowerShell Pipeline (26.09)

## What is pipeline?

--this is a pipeline |

--pass object to command

primer 1.

$service = get-service alg

$service | get-member

## Object cmdlets

sort\select\group\measure\compare - object

primer 1. sorting

1. Get-Process | Sort-Object -Property handles

2. Get-Process | Sort -Property handles

3. Get-Process | Sort handles

4. Get-Process | Sort handles -Descending

5. Get-Process | Sort handles -Descending | Select-Object -First 5

6. Get-Process | Sort handles -Descending | Select -First 5

primer 2. selecting specific properties to show

1. get-service | select -Property name, starttype

primer 3. measuring, counting data

#2 navigate to sys32

#cd c:\windows\system32

count data

#dir -file | Measure-Object

# agg functions

#dir -file | Measure-Object -Property Length -sum -average

primer 4. grouping

Get-Service | Group-Object -Property starttype

get-eventlog -LogName Application | Group-Object -property entrytrype

## format cmdlets

format-list\table\wide

-last cmdlet on pipeline because its hard for engine to turn data into text to output

Primer 1. Format table

get-process | sort handles -Descending | select -first 5 | Format-Table -Property name, handles, id

get-process | sort handles -Descending | select -first 5 | Ft name, handles, id

Primer 2. Format list

get-process | sort handles -Descending | select -first 5 | Format-list name, handles, id

# -|- fl also

Primer 3. Format wide +autosize

get-alias | format-wide -AutoSize

get-process | sort handles -Descending | select -first 5 | Ft name, handles, id -AutoSize

## Export and import data cmdlets

1. Export-csv

2. Export-clixml

3. Import-csev

4. Import-clixml

Primer 1.

get-service | export-csv -path xyz.csv

get-service | export-clixml -path xyz.xml

import-csv xyz.csv

## Out-cmdlets

-out-default\out-host -paging

-out-printer sends data to printer

-out-string converts data to string

-out-null - deletes the output

-out-file - sends output to a file, creates a file or overwrites it, -append if you want to add it to already existing file, -noclobber dont delete file it it exists

-out-grid- interactive table with data

primer 1.

Get-Process | Out-GridView

-gives you interactive table of data you want

$proc= Get-Process | Out-GridView -PassThru

-it is possible to select certain amount of data, then click OK, and data is passed through in a $proc variable

and can be used

# Module 11 – Working with Pipelining (28.09)

## Pipeline variable

$\_ or $PSItem or custom name created with -pipelinevariable

Primer 1.

get-service | ForEach-Object -Process {$\_.DisplayName}

-gives us for each object certain property

get-service | where-object -filterscript {$true}

get-service | Where-Object -FilterScript {$\_.Status -eq "running"}

-gives us data depending of a condition we typed in, same as where in sql

Primer 2.

get-service | ForEach-Object -Process {$PSItem.DisplayName}

-same as 1st part of primer 1.

get-service -PipelineVariable Service | ForEach-Object -Process {$Service.DisplayName}

-naming the variable within pipeline variable

## More about Foreach

get-service | foreach-object -process {$\_.DisplayName} same as

get-service | foreach -process {$\_.DisplayName} same as

get-service | foreach {$\_.DisplayName} same as

get-service | % {$\_.DisplayName}

primer 1.

get-service | foreach-object -process {$\_.DisplayName} -begin {write-host "starting" -ForegroundColor green} -end {write-host "ending" -ForegroundColor red}

-shorter way to write code from above

$begin ={write-host "starting" -ForegroundColor green}

$end={write-host "ending" -ForegroundColor red}

$processblock={$\_.DisplayName}

get-service | ForEach-Object -process $processblock -begin $begin -end $end

primer 2.

function Pipetest

{

begin

{

write-host "starting" -ForegroundColor green

write-host "starting" -ForegroundColor green

}

process

{

$\_.DisplayName

}

end

{

write-host "ending" -ForegroundColor red

write-host "ending" -ForegroundColor red

write-host "ending" -ForegroundColor red

write-host "ending" -ForegroundColor red

}

}

-could be written as a function and pipe services into function

get-service | Pipetest

primer 3.

(get-service).displayname

-check if a array has this property, can be done if you are looking for 1 specific property

## More about Where-object

primer 1.

get-service | where-object -process {$\_.status -eq "running"} same as

get-service | where -process {$\_.status -eq "running"} same as

get-service | where {$\_.status -eq "running"} same as

get-service | ? {$\_.status -eq "running"} same as

get-service | where status -eq "running"

-last example can be used only if we are filtering for 1 specific column

primer 2.

get-service|Where {$\_.status -eq "running" -and $\_.canpauseandcontinue}

-filtering with 2 conditions and after that pipe it again like this

get-service|Where {$\_.status -eq "running" -and $\_.canpauseandcontinue} |foreach {$\_.displayname}

## Fitering parameters vs Where-object

in a nutshell where-object is slower than a regular -filter or -name parameter

## Pipeline input

-there is 2 ways for Powershell to accept it

-cmdlet accepts pipeline input in two ways: ByValue or ByPropertyName

primer 1. ByValue

'MS', 'DC' |Restart-Computer -WhatIf

primer 2. ByProperty

import-csv 'name.csv' | restart-computer -whatif

-csv has a column named computer name and takes it(it has to)

## parameter binding steps

1.bind all named parameters

2.bind all positional parameters

3.bind from the pipeline by value with exact match

4. -||- with conversion

5.bind from the pipeline by name with exact type match

6.-||- with type conversion

# Module 12 – Different Types of Operators (04.10)

## Arithmetic Operators

+,-,\*,/

% - moduo

Primer 1.

7%2

sollution: 1

--------------------------------------------------

6\*2

sollution: 12

-------------------------------------------------------

"ABC"\*2

sollution: ABCABC

-shl

-shr

shift-left\right bitwise

## Assignment Operators

by adding = to +,-,\*,/,% you can do an arithmetic operation on the input number faster

primer 1.

$a = 5

$a+=5 =>10

$a-=5 =>5

$a\*=100 =>500

$a++ => +1 ->501

$a-- => -1 ->500

Primer 2.

$a=5

$a

write-host (++$a) -ForegroundColor Green

--++ before variable adds value to variable and then changes its color

$a

write-host ($a++) -ForegroundColor Red

--++ after variable first changes color then add the number but shows it without the addition

$a

## Split, join, replace operators

--Split

Primer 1.

-split "Danilo Cimesa"

"Danilo,Cimesa,Pancevo" -split ","

"Danilo Cimesa,Pancevo" -split ","

"a.b.c.d" -split "."

"a.b.c.d".Split(".")

--Join

Primer 1.

-join ("a", "b","c")

"how", "are", "you", "doing?" -join " Zivojin "

--Replace

Primer 1.

"Windows Powershell 4.0" -replace "4.0", "5.0"

"a,b,c,d" -replace ",",";"

## Format

Primer 1.

"my name is {0} {1} my age is {2}" -f "danilo","cimesa", "52"

{} - value container

-f -format operator

Primer 2.

"{0}" -f 6533.0987

"{0:n}" -f 6533.0987 #number

"{0:n3}" -f 6533.0987 #number

"{0:n0}" -f 6533.0987 #number

---------------------------------------------------------

"{0:p}" -f 0.35 #percentage

"{0:p0}" -f 0.35 #percentage

----------------------------------------------------------

"{0:c}" -f 53256346346.644664 #currency

Primer 3.

"{1}{0,5}{2}" -f "-", "a", "b", "not", "used"

"{1}{0,-5}{2}" -f "-", "a", "b", "not", "used"

# Module 13 – Flow Control (14.11)

## While loop

$i=0

while ($i -lt 5)

{

$i

$i++

}

write-host "end of code" -foregroundcolor green

------------------------------------------------------

## Do while loop

<#

$i=0

while ($i -lt 5)

{

$i

$i++

}

write-host "end of loop1" -foregroundcolor green

#>

do

{

$i

$i++

}

while ($i -lt 5)

write-host "end of code" -foregroundcolor green

-----------------------------------------------------

## Do until

-same as do while, but it loops on a false value

-----------------------------------------------------

## For loop

-used for counting

Primer 1.

#$i=0

for ($i=0; $i -lt 5;$i++)

{

$i

}

write-host "end of code" -ForegroundColor Cyan

Primer 2.

--reference index on an array

$services = get-service

#$i=0

for ($i=0; $services.Count;$i++)

{

$services[$i].DisplayName

#$i

}

#write-host "end of code" -ForegroundColor Cyan

## Foreach loop

Primer 1.

$services = Get-Service

foreach ($service in $services)

{

$service.displayname

}

## IF statements

primer 1.

$i=0

if ($i -lt 5)

{

write-host "Less than 5" -ForegroundColor yellow

$i++

}

write-host $i -ForegroundColor Cyan

primer 2.

$nums = 0..20

foreach ($num in $nums)

{

if ($num -lt 5)

{

write-host $num -ForegroundColor green

}

elseif ($num -lt 10)

{

write-host $num -ForegroundColor yellow

}

elseif ($num -lt 15)

{

write-host $num -ForegroundColor magenta

}

else

{

$num

}

}

varijacija primer 2.

--run all condition separately

$nums = 0..20

foreach ($num in $nums)

{

if ($num -lt 5)

{

write-host $num -ForegroundColor green

}

if ($num -lt 10)

{

write-host $num -ForegroundColor yellow

}

if ($num -lt 15)

{

write-host $num -ForegroundColor magenta

}

else

{

$num

}

}

--------------------------------------------------------

## Switch statement

-runs any code that happens to be true

-bunch of ifs back to back

Primer 1.

$nums = 0..20

foreach ($num in $nums)

{

switch ($num)

{

{$num -lt 5} { write-host $num -ForegroundColor green}

{$num -lt 10} { write-host $num -ForegroundColor cyan}

{$num -lt 15} { write-host $num -ForegroundColor magenta}

default {$num}

}

}

-default write number in default colors

Primer 2.

without foreach, but switch builds a foreach loop for us

$nums = 0..20

switch ($nums)

{

{$\_ -lt 5} { write-host $\_ -ForegroundColor green}

{$\_ -lt 10} { write-host $\_ -ForegroundColor cyan}

{$\_ -lt 15} { write-host $\_ -ForegroundColor magenta}

default {$\_}

}

Primer 3.

a)

switch ("hello") - like -eq (its not case sensitive so it returns uppercase and lowercase)

{

"hello" {write-host "lowercase" -ForegroundColor green}

"HELLO" {write-host "uppercase" -ForegroundColor green}

"h\*" {write-host "lowercase WILDCARD" -ForegroundColor green}

"H\*" {write-host "uppercase WILDCARD" -ForegroundColor green}

}

b)

switch -casesensitive ("hello") - like -ceq (returns lowercase)

{

"hello" {write-host "lowercase" -ForegroundColor green}

"HELLO" {write-host "uppercase" -ForegroundColor green}

"h\*" {write-host "lowercase WILDCARD" -ForegroundColor green}

"H\*" {write-host "uppercase WILDCARD" -ForegroundColor green}

}

c)

switch -casesensitive -wildcard ("hello") (returns lowercase and lowercase wildcard)

{

"hello" {write-host "lowercase" -ForegroundColor green}

"HELLO" {write-host "uppercase" -ForegroundColor green}

"h\*" {write-host "lowercase WILDCARD" -ForegroundColor green}

"H\*" {write-host "uppercase WILDCARD" -ForegroundColor green}

}

d)

switch -casesensitive -regex ("hello") - like -ceq (returns lowercase + both wildcards)

{

"hello" {write-host "lowercase" -ForegroundColor green}

"HELLO" {write-host "uppercase" -ForegroundColor green}

"h\*" {write-host "lowercase WILDCARD" -ForegroundColor green}

"H\*" {write-host "uppercase WILDCARD" -ForegroundColor green}

}

Flow control keywords

-Break, Continue, return and exit

Primer 1.

a)

--Break, immeadiately exists the loop

$nums = 0..20

switch ($nums)

{

{$\_ -lt 5} { write-host $\_ -ForegroundColor green; break}

{$\_ -lt 10} { write-host $\_ -ForegroundColor cyan; break}

{$\_ -lt 15} { write-host $\_ -ForegroundColor magenta; break}

default {$\_}

}

b)

--continue goes back on the top and moves on

$nums = 0..20

switch ($nums)

{

{$\_ -lt 5} { write-host $\_ -ForegroundColor green; continue}

{$\_ -lt 10} { write-host $\_ -ForegroundColor cyan; continue}

{$\_ -lt 15} { write-host $\_ -ForegroundColor magenta; continue}

default {$\_}

}

c)

$i=0

while ($i -lt 5)

{

$i++

if($i -eq 3)

{

continue

}

write-host $i -ForegroundColor Green

}

Primer 2.

--return --finish my function or code

function returntest

{

param($val)

if ($val -gt 10)

{

return 0

}

$val+10

}

$test= returntest -val 10

$test = 20

$test= returntest -val 11

$test = 0

exit

-is used if you have errors, get out of code youre running

-terminate whatever youre running, provides error code if you need to

# Module 14 – Arrays (16.11)

-Array fixed size of memory that can hold data (list of data)

-simplified -> bunch of connected boxes together

-always starts from 0...

## Create

-simple way to create an array is just to write values followed with commas

$a=1,3,2,6,"hello", 2,3

-every get cmdlet is an array

$a2=get-service

-create empty array with @(), then fill it with value

$a3=@("hello")

-to check if it is an array, $a3.gettype() --> base type returns system array

## Accessing data

-access data from the front of an array

$a[0]

$a[1]

-access data from the back

$a[-1]

$a[-2]

-slicing the array, showing the array of an array

$a[1,4,0]

$a[0..3]

## Change

$a[0]=100

$a

$a[0]+=20

$a

$a2[4].DisplayName

## Adding to --inefficient

--array is fixed size, cannot just stick new piece of data

--arraylist to creat arrays more efficiently

--to add new value, it creates bigger array and copies values from previous and so on

$a3+="goodbye"

## Sorting array

$a

$a | Sort-Object

-sorting by default in ascending order

$a | Sort-Object -Descending

-sorting in descending order

$a= $a | Sort-Object -Descending

-sorting permanently

[array]::sort($a)

-array datatype, set static member called sort and hand him an array

## Members of an Array

$a2 | Get-Member --doesnt work very well

Get-member | inputobject $a2

--returns members of an array

--count property gives us length

($fake.count returns 0)

--set = $a[2]=value