ADMINISTRATION OF OPERATING SYSTEMS

PowerShell

Managing IP addresses

Cmdlet	Description
New-NetIPAddress	Creates a new IP address
Set-NetIPAddress	Sets properties of an IP address
Get-NetIPAddress	Displays properties of an IP address
Remove-NetIPAddress	Deletes an IP address

- New-NetIPAddress -IPAddress 192.168.1.10 -InterfaceAlias “Ethernet” -PrefixLength 24 -DefaultGateway 192.168.1.1

Managing routing

Cmdlet	Description
New-NetRoute	Creates an IP routing table entry
Set-NetRoute	Sets properties of an IP routing table entry
Get-NetRoute	Displays properties of an IP routing table entry
Remove-NetRoute	Deletes an IP routing table entry
Find-NetRoute	Identifies the best local IP address and route to reach a remote address

- New-NetRoute -DestinationPrefix 0.0.0.0/24 -InterfaceAlias “Ethernet” -DefaultGateway 192.168.1.1

Managing DNS clients

Cmdlet	Description
Get-DnsClient	Gets details about a network interface on a computer
Set-DnsClient	Set DNS client configuration settings for a network interface
Get-Dns ClientServerAddress	Gets the DNS server address settings for a network interface
Set-Dns ClientServerAddress	Sets the DNS server address for a network interface
Get-DnsClient	Gets details about a network interface on a computer

- Set-DnsClient -InterfaceAlias Ethernet -ConnectionSpecificSuffix “adatum.com”

Managing Windows Firewall

Cmdlet	Description
New-NetFirewallRule	Creates a new firewall rule
Set-NetFirewallRule	Sets properties for firewall rules
Get-NetFirewallRule	Gets properties for firewall rules
Remove-NetFirewallRule	Deletes firewall rules
Rename-NetFirewallRule	Renames firewall rules
Copy-NetFirewallRule	Makes a copy of firewall rules
Enable-NetFirewallRule	Enables firewall rules
Disable-NetFirewallRule	Disables firewall rules
Get-NetFirewallProfile	Gets properties for firewall profiles
Set-NetFirewallProfile	Sets properties for firewall profiles

Demonstration: Configuring network settings

- In this demonstration, you'll learn how to:

1. Test the network connection to **LON-DC1**.
2. Review the network configuration for **LON-CL1**.
3. Change the client IP address.
4. Change the DNS server for **LON-CL1**.
5. Change the default gateway for **LON-CL1**.
6. Confirm the network configuration changes.
7. Test the effect of the changes.

Server administration cmdlets

Group Policy management cmdlets

Cmdlet	Description
New-GPO	Creates a new GPO
Get-GPO	Retrieves a GPO
Set-GPO	Modifies properties of a GPO
Remove-GPO	Deletes a GPO
Rename-GPO	Renames a GPO
Backup-GPO	Creates a backup of a GPO
Copy-GPO	Copies a GPO from one domain to another
Restore-GPO	Restores a GPO from backup files
New-GPLink	Links a GPO to an AD DS container
Import-GPO	Imports GPO settings from a backed-up GPO
Set-GPRegistryValue	Configures one or more registry-based policy settings in a GPO

Server Manager cmdlets

Cmdlet	Description
Get-WindowsFeature	Obtains and displays information about Windows Server roles, services, and features on the local computer
Install-WindowsFeature	Installs roles, services, or features
Uninstall-WindowsFeature	Uninstalls roles, services, or features

- Install–WindowsFeature “nlb”

Hyper-V cmdlets

Cmdlet	Description
Get-VM	Gets properties of a VM
Set-VM	Sets properties of a VM
New-VM	Creates a new VM
Start-VM	Starts a VM
Stop-VM	Stops a VM
Restart-VM	Restarts a VM
Suspend-VM	Pauses a VM
Resume-VM	Resumes a paused VM
Import-VM	Imports a VM from a file
Export-VM	Exports a VM to a file
Checkpoint-VM	Creates a checkpoint of a VM

IIS administration cmdlets

Cmdlet	Description
New-IISSite	Creates a new IIS website
Get-IISSite	Gets properties and configuration information about an IIS website
Start-IISSite	Starts an existing IIS website on the IIS server
Stop-ISSSite	Stops an IIS website
New-WebApplication	Creates a new web application
Remove-WebApplication	Deletes a web application
New-WebAppPool	Creates a new web application pool
Restart-WebAppPool	Restarts a web application pool

Windows PowerShell in Windows 10

Managing Windows 10 using PowerShell

Cmdlet	Description
Get-ComputerInfo	Retrieves all system and operating system properties from the computer
Get-Service	Retrieves a list of all services on the computer
Get-EventLog	Retrieves events and event logs from local and remote computers (Only available in Windows PowerShell 5.1)
Get-Process	Retrieves a list of all active processes on a local or remote computer
Stop-Service	Stops one or more running services
Stop-Process	Stops one or more running processes
Stop-Computer	Shuts down local and remote computers
Clear-EventLog	Deletes all of the entries from the specified event logs on the local computer or on remote computers
Clear-RecycleBin	Deletes the content of a computer's recycle bin
Restart-Computer	Restarts the operating system on local and remote computers
Restart-Service	Stops and then starts one or more services

Managing permissions with PowerShell

Cmdlet	Description
Get-Acl	Gets objects that represent the security descriptor of a file or resource. The security descriptor includes the access control lists (ACLs) of the resource. The ACL lists permissions that users and groups have to access the resource.
Set-Acl	Changes the security descriptor of a specified item, such as a file, folder, or a registry key, to match the values in a security descriptor that you supply

- To update access permissions:

1. Use **Get-Acl** to retrieve the existing access control list rules for the object.
2. Create a new FileSystemAccessRule to be applied to the object.
3. Add the new rule to the existing ACL permission set.
4. Use **Set-Acl** to apply the new ACL to the existing file or folder.

Understand the pipeline

What is the pipeline?

- Consider the following regarding the PowerShell pipeline:
  - Windows PowerShell runs commands in a pipeline.
  - Each console command line is a pipeline.
  - Commands are separated by a pipe character (**|**).
  - Commands execute from left to right.
  - Output of each command is *piped* (passed) to the next.
  - The output of the last command in the pipeline is what you notice on your screen.
  - Piped commands typically follow the pattern **Get |Set**, **Get | Where**, or **Select | Set**.

What is the pipeline? (Slide 2)

- PowerShell objects can be compared to real-world items.
  - For example, consider a car as an object. The car's attributes can be described as engine, car color, car size, type, make and model. In PowerShell, these would be known as properties.
  - Properties of the object could be, in turn, objects themselves. For instan ce, the engine property is also an object with attributes, such as pistons, spark plugs, crankshaft, etc.
  - Objects have actions, corresponding to opening or closing doors, changing gears, accelerating, and applying brakes. In PowerShell, these actions are called methods.

Pipeline output

- Windows PowerShell commands produce objects as their output.
- An object is like a table of data in memory.
- Allows the **Get | Set** pattern to work.

Discovering object members

- Object members include:
  - Properties
  - Methods
  - Events
- Run a command that produces an object, and pipe that object to **Get-Member** to review a list of members.
- **Get-Member** is a discovery tool that's similar to Help. You can use it to learn how to use the shell.

Demonstration: Reviewing object members

- In this demonstration, you'll learn how to run commands in the pipeline and how to use **Get-Member**.

Formatting pipeline output

- Use the following cmdlets to format pipeline output:
  - **Format-List**
  - **Format-Table**
  - **Format-Wide**
- The *-Property* parameter:
  - Is common to all formatting cmdlets.
  - Filters output to specified property names.
  - Can only specify properties that were passed to the formatting command.

Demonstration: Formatting pipeline output

- In this demonstration, you'll learn how to format pipeline output.

Select, sort, and measure objects

Sorting objects by a property

- Each command determines its own default sort order.
- **Sort-Object** can re-sort objects in the pipeline.
  - **Get-Service | Sort-Object Name –Descending**
- Sorting enables grouping output by using:
  - The *-GroupBy* parameter.
  - The **Group-Object** command.

Demonstration: Sorting objects

In this demonstration, you'll learn how to sort objects by using the **Sort-Object** command.

Measuring objects

- **Measure-Object** accepts a collection of objects and counts them.
- Add *-Property* to specify a single numeric property, and then add:
  - *-Average* to calculate an average.
  - *-Minimum* to display the smallest value.
  - *-Maximum* to display the largest value.
  - *-Sum* to display the sum.
- The output is a measurement object.

```
Get-ChildItem -File | Measure -Property Length -Sum -Average -Minimum -Max
```

Demonstration: Measuring objects

- In this demonstration, you'll learn how to measure objects by using the **Measure-Object** command.

Selecting a subset of objects

- One of two uses for **Select-Object**.
- Use parameters to select the specified number of rows from the beginning or end of the piped-in collection:
  - *-First* for the beginning.
  - *-Last* for the end.
  - *-Skip* to skip a number of rows before selecting.
  - *-Unique* to ignore duplicated rows.
- You can’t specify any criteria for choosing specific rows.

Selecting object properties

- The second use of **Select-Object**.
- Use the *-Property* parameter to specify a comma-separated list of properties (wildcards are accepted) to include.
- You can combine the *-Property* parameter with *-First*, *-Last*, and *-Skip* to select a subset of rows.

Demonstration: Selecting objects

- In this demonstration, you'll learn several ways to use the **Select-Object** command.

Filter objects out of the pipeline

Comparison operators

Comparison type	Case-insensitive operator	Case-sensitive operator
Equality	-eq	-ceq
Inequality	-ne	-cne
Greater than	-gt	-cgt
Less than	-lt	-clt
Greater than or equal to	-ge	-cge
Less than or equal to	-le	-cle
Wildcard equality	-like	-clike

Basic filtering syntax

- The **Where-Object** command provides filtering.
- Examples of basic syntax include:

```
Get-Service |
Where Status -eq Running

Get-Process |
Where CPU -gt 20
```

Basic filtering syntax (Slide 2)

Limitations of the basic syntax:

- It supports only a single comparison—you can't compare two things.
- It doesn't support property dereferencing—you can refer to only direct properties of the c
```
Get-Service | Where Name.Length -gt 5
```
- This won't work:

Advanced filtering syntax

- Supports multiple conditions and has no restrictions on what kinds of expressions you can use.
- Requires a filter script that contains your filtering criteria and that evaluates to either True or False.
- Inside the filter script, use **$PSItem** or **$_** to refer to the object that was piped into the command.

Advanced filtering syntax (Slide 2)

Here are three examples of advanced filtering:

```
Get-Service | Where-Object -Filter {$PSItem.Status -eq 'Running' }
```

```
Get-Service | Where { $_.Status -eq 'Running' }
```

```
Get-Service | ? { $PSItem.Status -eq 'Running' }
```

Advanced filtering syntax (Slide 3)

- Use the Boolean operators **-and** and **-or** to combine multiple comparisons into a single expression:

```
Get-Volume | Where-Object -Filter {
  $PSItem.HealthStatus -ne 'Healthy'
    -or
  $PSItem.SizeRemaining -lt 100MB
}
```

Demonstration: Filtering

- In this demonstration, you'll learn various ways to filter objects out of the pipeline.

Optimizing filtering performance

- To improve performance, move filtering as close to the beginning of the command line as possible.
- Some commands have parameters that can filter for you, so whenever possible, use those parameters instead of **Where-Object**.

Enumerate objects in the pipeline

Purpose of enumeration

- To take a collection of objects and:
  - Run an action on each item.
  - Process them one at a time.
- Not necessary when PowerShell has a command that can perform the action you need.
- Useful when an object has a method that does what you want, but PowerShell doesn’t offer an equivalent command.

Basic enumeration syntax

```
Get-ChildItem –Path C:\Example –File |
ForEach-Object –MemberType Encrypt

Get-ChildItem –Path C:\Example –File |
ForEach Encrypt

Get-ChildItem –Path C:\Example –File |
% –MemberType Encrypt
```

Basic enumeration syntax (Slide 2)

Limitations:

- Can access only a single member (method or property) of the objects that were piped into the command.
- Can’t:
  - Run commands or code.
  - Evaluate expressions.
  - Make logical decisions.

Demonstration: Basic enumeration

- In this demonstration, you'll learn how to use the basic enumeration syntax to enumerate several objects in a collection.

Advanced enumeration syntax

- Allows you to perform any task by entering commands in a script block.
- Uses **$PSItem** or **$_** to reference the objects that were piped into the command:

```
Get-ChildItem C:\Test -File |  ForEach-Object { $PSItem.Encrypt() }
```

- Has additional parameters that allow you to specify actions to take before and after the collection of objects is processed.

Demonstration: Advanced enumeration

- In this demonstration, you'll learn two ways to use the advanced enumeration syntax to perform tasks on several objects.

Send pipeline data as output

Writing output to a file

- **Out-File** writes whatever is in the pipeline to a text file.
- The **>** and **>>** redirection operators are also supported.
- The text file is formatted exactly the same as the data would be on the screen—no conversion to another form occurs.
- Unless the data has been converted to another form, the resulting text file is usually suitable for reviewing only by a person.
- As you start to build more complex commands, you need to keep track of what the pipeline contains at each step.

Converting output to CSV

- The commands are:
  - **ConvertTo-CSV**
  - **Export-CSV**
- The commands send:
  - Properties as headers.
  - No type information.
- You can easily open large CSV files in Excel.

Converting output to XML

- **ConvertTo-CliXml**
- **Export-CliXml**
- Portable data format.
- Multiple value properties become individual entries.

Converting output to JSON

- The command is:
  - **ConvertTo-JSON**
- The advantages are:
  - Compactness.
  - Ease of use, especially with JavaScript.
  - A format like a hash table.

Converting output to HTML

- The command is:
  - **ConvertTo-HTML**
- The command creates a table or list in HTML.
- You must pipe the output to a file.
- The parameters include:
  - *-Head*
  - *-Title*
  - *-PreContent*
  - *-Postcontent*

Demonstration: Exporting data

- In this demonstration, you'll learn different ways to convert and export data.

Additional output options

- **Out-Host** allows more control of on-screen output.
- **Out-Printer** sends output to a printer.
- **Out-GridView** creates an interactive, spreadsheet-like view of the data.

Use variables

What are variables?

- A variable stores a value or object in memory.
- Some things you can do with a variable:
  - Store the name of a log file that you write data to multiple times.
  - Derive and store an email address based on the name of a user account.
  - Calculate and store the date representing the beginning of the most recent 30-day period, to identify whether computer accounts have authenticated during that time.
- You can access object properties stored in a variable.
- Variables and their values can be reviewed in the PSDrive named **Variable**.

Variable naming

- Variable names:
  - Should be easily understandable.
  - Can contain spaces if enclosed in braces.
  - Should contain only alphanumeric characters.
  - Are not case-sensitive.
- A common convention for variable names uses capital letters to separate words:
  - **$LogFile**
  - **$StartDate**
  - **$ipAddress**

Assigning a value to a variable

- Use standard mathematical operators when working with variables.
- To assign a value to a variable, use the **=** operator:
  - **$num1 = 5**
  - **$logFile = "C:\Logs\Log.txt"**
  - **$user = Get-ADUser Administrator**
  - **$service = Get-Service W32Time**
- To display the value of a variable, enter the variable name or use **Write-Host**:
  - **$num1**
  - **Write-Host "The log location is $logfile"**
- To clear a variable, use **$null**:
  - **$num1 = $null**

Variable types

- The variable type determines the data that can be stored in it:
  - String. Stores text, including special characters.
  - Int32. Stores integers without decimals.
  - Double. Stores numbers with decimals.
  - DateTime. Stores date and time.
  - Bool. Stores true or false.
- Windows PowerShell can automatically assign the type based on a value.
- Specify the type if data is going to be ambiguous.

Demonstration: Assigning a variable type

- In this demonstration, you will learn how to:

1. Set the value for variables.
2. Display the contents of a variable.
3. Review the properties of a variable.
4. Review variables in memory.
5. Use the **GetType** method to review variable types.
6. Force variable types when assigning values.
7. Add variables together.

Manipulate variables

Identifying methods and properties

- A variable's properties and methods are based on the variable type.
- To identify a variable's properties and methods, use:
  - **Get-Member**
  - Tab completion
- Documentation for properties and methods is available in the Microsoft .NET Framework Class Library.

Working with strings

- The only property available for strings is Length.
- Some commonly used methods for strings are:
  - **Contains(string value)**
  - **Insert(int startindex,string value)**
  - **Remove(int startindex,int count)**
  - **Replace(string value,string value)**
  - **Split(char separator)**
  - **ToLower()**
  - **ToUpper()**

Demonstration: Manipulating strings

- In this demonstration, you will learn how to:

1. Use the **Contains** method.
2. Use the **Insert** method.
3. Use the **Replace** method.
4. Use the **Split** method.
5. Use the **ToUpper** method.
6. Use the **ToLower** method.

Working with dates

- Commonly used DateTime properties:
  - **Hour**
  - **Date**
  - **Minute**
  - **DayOfWeek**
  - **Second**
  - **Month**

- Commonly used DateTime methods:
  - **AddDays(double value)**
  - **ToLongDateString()**
  - **AddHours(double value)**
  - **ToShortDateString()**
  - **AddMinutes(double value)**
  - **ToLongTimeString()**
  - **AddMonths(int value)**
  - **ToShortTimeString()**

Demonstration: Manipulating dates

- In this demonstration, you will learn how to:

1. Put output from **Get-Date** into a variable.
2. Review date properties.
3. Use date methods.

Manipulate arrays and hash tables

What is an array?

- An array contains multiple values or objects.
- Values can be assigned by:
  - Providing a list:
    $computers = "LON-DC1","LON-SRV1","LON-CL1"
  - Using command output:
    $users = Get-ADUser -Filter *
- You can create an empty array:
  $newUsers = @()
- You can force an array to be created when adding only a single item:
  [Array]$computers = "LON-DC1"

Working with arrays

- To display all items in an array:

  $users

- To display specific items in an array by using an index number:

  $users[0]

- To add items to an array:

  $users = $users + $user1

  $users += $user1

- To pipe the array to **Get-Member** to identify what you can do with the array contents:

  $files | Get-Member

Working with array lists

- Arrays are fixed-size, which limits performance and makes removing items difficult.
- Arraylists are variable-sized.
- To create an arraylist:
  - **$computers = New-Object System.Collections.ArrayList**
  - **[System.Collections.ArrayList]$computers = "LON-DC1","LON-SRV1"**
- To add or remove items from an arraylist:
  - **$computers.Add("LON-SVR2")**
  - **$computers.Remove("LON-CL1")**
  - **$computers.RemoveAt(1)**

Demonstration: Manipulating arrays and array lists

- In this demonstration, you will learn how to:

1. Create an array.
2. Review the contents of an array.
3. Add items to an array.
4. Create an array list.
5. Review the contents of an array list.
6. Remove an item from an array list.

What is a hash table?

- A *hash table* is a list of names and values.
- To refer to a value in the hash table, you provide the key:
  - **$servers.'LON-DC1'**
  - **$servers['LON-DC1']**

Key	IP address
LON-DC1	172.16.0.10
LON-SRV1	172.16.0.11
LON-SRV2	172.16.0.12

Working with hash tables

- To define a hash table:

  $servers = @{“LON-DC1”=“172.16.0.10”; “LON-SRV1”=“172.16.0.11”}

- To add an item to a hash table:

  $servers.Add(“LON-SRV2”,”172.16.0.12”)

- To remove an item from a hash table:

  $servers.Remove(“LON-DC1”)

- To update a value for an item in a hash table:

  $servers.’LON-SRV2’=“172.16.0.100”

Demonstration: Manipulating hash tables

- In this demonstration, you will learn how to:

1. Create a hash table.
2. Review the contents of a hash table.
3. Add an item to a hash table.
4. Remove an item from a hash table.
5. Create a hash table for a calculated property.

Introduction to scripting with Windows PowerShell

Overview of Windows PowerShell scripts

- Windows PowerShell scripts can be used for:
  - Repetitive tasks.
  - Complex tasks.
  - Reporting.

- Windows PowerShell scripts:
  - Use constructs such as **ForEach**, **If**, and **Switch**.
  - Have a .ps1 file extension.

Modifying scripts

- Modifying an existing script is easier and faster than creating your own.
- You should:
  - Understand how a downloaded script works.
  - Test a downloaded script in a non-production environment.
- You can get scripts from:
  - The PowerShell Gallery.
  - Blogs and websites.

Creating scripts

- Create a script if you can’t find one that meets your needs.
- Use code snippets from other sources when building your script.
- Develop scripts in an isolated environment that can’t affect production.
- Build scripts incrementally for easier testing during development.

What is the PowerShellGet module?

- Windows PowerShellGet:
  - Has cmdlets for accessing and publishing items in the PowerShell Gallery.
  - Is included in Windows Management Framework 5.0.
  - Can be downloaded for Windows PowerShell 4.0.
  - Uses the NuGet provider.
- You must enable TLS 1.2 to access the PowerShell Gallery
  - **[Net.ServicePointManager]::SecurityProtocol = [Net.SecurityProtocolType]::Tls12**
- You can implement a private PowerShell Gallery.
- Cmdlets for finding items are:
  - **Find-Module**
  - **Find-Script**

Running scripts

- To enhance security, the .ps1 file extension is associated with Notepad.
- Integration with File Explorer:
  - Open
  - Run with PowerShell
  - Edit
- To run scripts at the Windows PowerShell prompt:
  - Enter the full path - **C:\Scripts\MyScript.ps1**
  - Enter a relative path - **\Scripts\MyScript.ps1**
  - Reference the current directory - **.\MyScript.ps1**

The script execution policy

- The options for the execution policy are:
  - **Restricted**
  - **AllSigned**
  - **RemoteSigned**
  - **Unrestricted**
  - **ByPass**
- Modify the execution policy by using:
  - **Set-ExecutionPolicy**
  - The **Turn on Script Execution** Group Policy setting
- Override the execution policy for a single instance:
  - **Powershell.exe -ExecutionPolicy Bypass**
- Remove downloaded status from a script by using **Unblock-File**

Windows PowerShell and AppLocker

- You can use AppLocker to control the running of Windows PowerShell scripts.
- AppLocker can limit scripts based on:
  - Name.
  - Location.
  - Publisher (with digital signature).
- In Windows PowerShell 5.0 and newer, interactive prompts are limited to **ConstrainedLanguage** mode.

Digitally signing scripts

- Use digital signatures on scripts with the **AllSigned** execution policy.
- Use digitally signed scripts to:
  - Formalize the script approval process.
  - Prevent accidental changes.
- A trusted code-signing certificate is required to add a digital signature to a script.
- To set a digital signature:
  - **$cert = Get-ChildItem -Path "Cert:\CurrentUser\My" -CodeSigningCert**
  - **Set-AuthenticodeSignature -FilePath “C:\Scripts\MyScript.ps1” -Certificate $cert**

Demonstration: Digitally signing a script

- In this demonstration, you will learn how to:

1. Install a code-signing certificate.
2. Digitally sign a certificate.
3. Review the digital signature.

Script constructs

Understanding ForEach loops

- Example:

```
ForEach ($user in $users) {
    Set-ADUser $user -Department "Marketing"
}
```

- Commands between braces are processed once for each item in the array.
- You don’t need to know how many items are in the array.
- **$user** contains each array item during the loop.
- The indent is to make it easier to review.
- Variable names should be meaningful.
- The **ForEach-Object** cmdlet has the *-Parallel* parameter.

Demonstration: Using a ForEach loop

- In this demonstration, you will:

1. Review a script with a **ForEach** loop.
2. Run the script.

Understanding the If construct

- Use the **If** construct to make decisions.
- There can be zero or more **ElseIf** statements.
- **Else** is optional.
- Example:

```
If ($freeSpace -le 5GB) {
    Write-Host "Free disk space is less than 5 GB"
} ElseIf ($freeSpace -le 10GB) {
    Write-Host "Free disk space is less than 10 GB"
} Else {
    Write-Host "Free disk space is more than 10 GB"
}
```

Demonstration: Using the If construct

- In this demonstration, you will:

1. Review a script with the **If** construct.
2. Test the script functionality.

Understanding the Switch construct

- The **Switch** construct compares a variable to a list of values.
- Example:

```
Switch ($choice) {
    1 { Write-Host “You selected menu item 1” }
    2 { Write-Host “You selected menu item 2” }
    3 { Write-Host “You selected menu item 3” }
    Default { Write-Host “You did not select a valid option” }
}
```

- You can also use wildcards and regular expressions:
  - Multiple matches are possible.

Demonstration: Using the Switch construct

- In this demonstration, you will:

1. Review a script with the **Switch** construct.
2. Test the script functionality.

Understanding the For construct

- The **For** construct is used to run a script block a specific number of times based on the:
  - Initial state.
  - Condition.
  - Action.
- Example:

```
For($i=1; $i -le 10; $i++) {
    Write-Host "Creating User $i"
}
```

Understanding other loop constructs

- **Do..While**
  - Loops until the condition is false
  - Guaranteed to process the script block once.
  - Example:

```
Do {
    Write-Host "Code block to process"
} While ($answer -eq "go")
```

- **Do..Until**
  - Loops until the condition is true.
  - Guaranteed to process the script block once.
  - Example:

```
Do {
    Write-Host "Code block to process"
} Until ($answer -eq "stop")
```

Understanding other loop constructs (Slide 2)

- **While**
  - Processes the script block until the condition is false.
  - Execution on the script block is not guaranteed.
  - Example:

```
While ($answer -eq "go") {
    Write-Host "Script block to process"
}
```

Understanding Break and Continue

- **Continue** stops processing the current iteration of a loop:

```
ForEach ($user in $users) {
    If ($user.Name -eq “Administrator”) {Continue}
    Write-Host “Modify user object”
}
```

- **Break** completely stops loop processing:

```
ForEach ($user in $users) {
    $number++
    Write-Host “Modify User object $number”
    If ($number -ge $max) {Break}
}
```

Import data from files

Using Get-Content

- **Get-Content** retrieves content from a text file.
- Each line in the file becomes an item in an array:

```
$computers = Get-Content "C:\Scripts\computers.txt"
```

- Import multiple files by using wildcards:

```
Get-Content -Path "C:\Scripts\*" -Include "*.txt","*.log"
```

- Limit the data retrieved by using the *-TotalCount* and *-Tail* parameters.

Using Import-Csv

- The first row in the CSV file is a header row.
- Each line in the CSV file becomes an array item.
- The header row defines the property names for the items:

$users = Import-Csv C:\Scripts\Users.csv

- You can specify a custom delimiter by using the *-Delimiter* parameter.
- You can specify a missing header row by using the *-Header* parameter.

Using Import-Clixml

- XML can store more complex data than CSV files.
- **Import-Clixml** creates an array of objects:

```
$users = Import-Clixml C:\Scripts\Users.xml
```

- Use **Get-Member** to review object properties.
- Limit the data retrieved by using the *-First* and *-Skip* parameters.

Using ConvertFrom-Json

- JSON is:
  - A lightweight data format similar to XML.
  - Commonly used by web services.
- You can convert from JSON, but not import directly.

```
$users = Get-Content C:\Scripts\Users.json | ConvertFrom-Json
```

- Retrieve JSON data directly from web services by using **Invoke-RestMethod**.

```
$users = Invoke-RestMethod "https://hr.adatum.com/api/staff"
```

Demonstration: Importing data

- In this demonstration, you will learn how to:

1. Use **Get-Content** to retrieve text data.
2. Use **Import-Csv** to retrieve CSV data.
3. Use **Import-Clixml** to retrieve XML data.

Accept user input

Identifying values that might change

- Scripts might initially have static values:
  - Find users that haven't signed in for 30 days.
  - Find specific events on domain controllers.
- When scripts are reused, some of those values might need to change.
- To simplify changing values in scripts:
  - Use variables defined at the beginning of the script.
- To avoid changing scripts:
  - Accept user input.

Using Read-Host

- Use **Read-Host** to request user input while a script is running:

```
$answer = Read-Host "How many days"
```

- To avoid displaying a colon, combine **Write-Host** and **Read-Host**:

```
Write-Host "How many days?" –NoNewline
$answer = Read-Host
```

- The *-MaskInput* and *-AsSecureString* parameters hide content as it's entered.

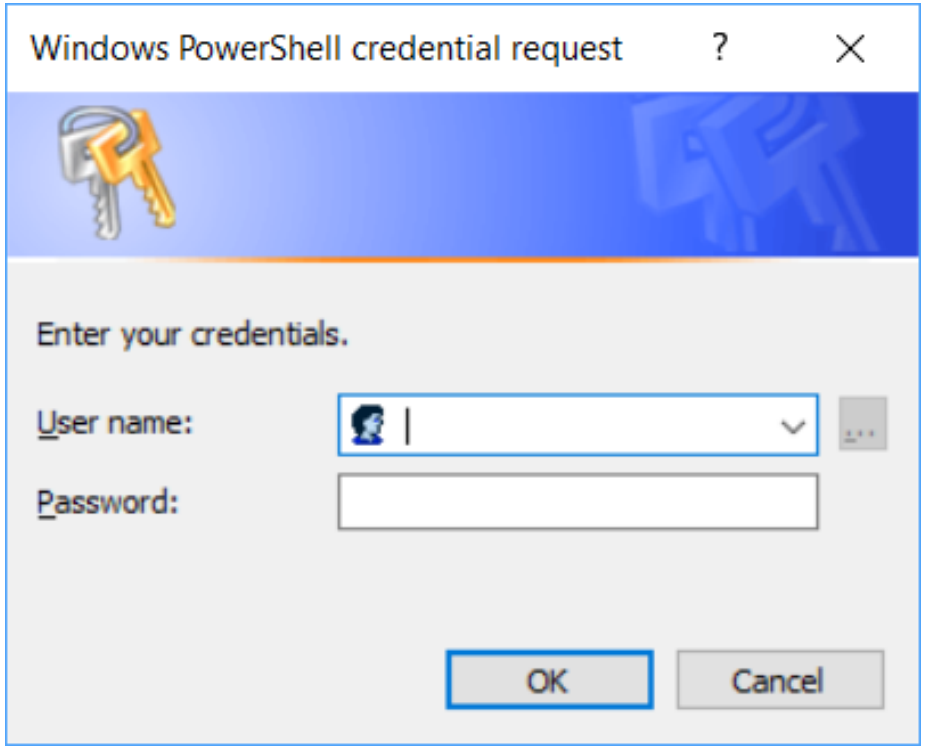
Using Get-Credential

- You can request credentials and use them to run commands in a script:

$cred = Get-Credential

Set-ADUser –Identity $user -Department "Marketing" -Credential $cred

- You can customize the credential prompt by using parameters:
  - *-Message*
  - *-UserName*
- You can store credentials securely:
  - **$cred | Export-Clixml C:\cred.xml**
  - The **SecretManagement** module



Using Out-Gridview

- **Out-GridView** can be used as a simple menu system

$selection = $users | Out-GridView -PassThru

- For more control over the items selected, you can use the -*OutputMode* parameter

Value	Description
None	Doesn’t allow any rows to be selected
Single	Allows zero rows or one row to be selected
Multiple	Allows zero rows, one row, or multiple rows to be selected

Demonstration: Obtaining user input

- In this demonstration, you will learn how to:

1. Use **Read-Host**.
2. Use **Get-Credential**.
3. Use **Out-GridView**.

Passing parameters to a script

- Use a **Param()** block to identify the variables that will store parameter values:

```
Param(
    [string]$ComputerName
    [int]$EventID
)
```

- The variable names in the **Param()** block define the parameter names:

```
.\GetEvent.ps1 -ComputerName LON-DC1 -EventID 5772
```

- It's a best practice to define variable types:
  - Errors are generated when data can't be converted.
  - **[switch]** defines parameters that are on or off.
- You can:
  - Define default values for parameters: **[string]$ComputerName = "LON-DC1"**
  - Request user input for parameters: **[int]$EventID = Read-Host "Enter event ID"**

Demonstration: Obtaining user input by using parameters

- In this demonstration, you will learn how to:

1. Review a script with a **param()** block.
2. Pass parameters to the script.
3. Request user input when a parameter isn’t supplied.
4. Set a default value for a parameter.

Troubleshooting and error handling

Understanding error messages

- Error messages are useful for troubleshooting.
- You might encounter error messages because:
  - You made a mistake while entering text.
  - You queried an object that doesn't exist.
  - You attempted to communicate with a computer that's offline.
- Errors are stored in the **$Error** array
  - The most recent error is stored in **$Error[0]**.

Adding script output

- Use **Write-Host** to display additional information when the script is running:
  - Variable values.
  - Location in the script.
- To slow down the data onscreen for easier reviewing, use:
  - **Start-Sleep**
  - **Read-Host**
- Comment out the **Write-Host** commands when not required.
- When you use **CmdletBinding()**, you can also use:
  - **Write-Verbose**
  - **Write-Debug**

Using breakpoints

- Use breakpoints to pause a script and query or modify variable values.
- **Set-PSBreakPoint** examples:
  - **Set-PSBreakPoint -Line 23 -Script "MyScript.ps1"**
  - **Set-PSBreakPoint -Command "Set-ADUser" -Script "MyScript.ps1"**
  - **Set-PSBreakPoint -Variable "computer" -Mode ReadWrite -Script "MyScript.ps1"**
- You use the *-Action* parameter to specify code to perform.
- The Windows PowerShell ISE has line-based breakpoints.
- Visual Studio Code:
  - Has conditional breakpoints.
  - Displays variable values in a dedicated area.

Demonstration: Troubleshooting a script

In this demonstration, you will learn how to:

1. Run a script the generates an error.
2. Review the **$Error[0]** variable.
3. Clear the **$Error** variable.
4. Configure and use and break point.
5. Remove all breakpoints.

Understanding error actions

- **$ErrorActionPreference** defines what happens for non-terminating errors:
  - **Continue**
  - **SilentlyContinue**
  - **Inquire**
  - **Stop**
- It's preferred to modify the error action for individual commands rather than globally:
  - **Get-WmiObject -Class Win32_BIOS -ComputerName LON-SVR1,LON-DC1 -ErrorAction Stop**
- Set the error action globally when it can't be done at the command:
  - For example, using methods on an object

Hvala na pažnji!