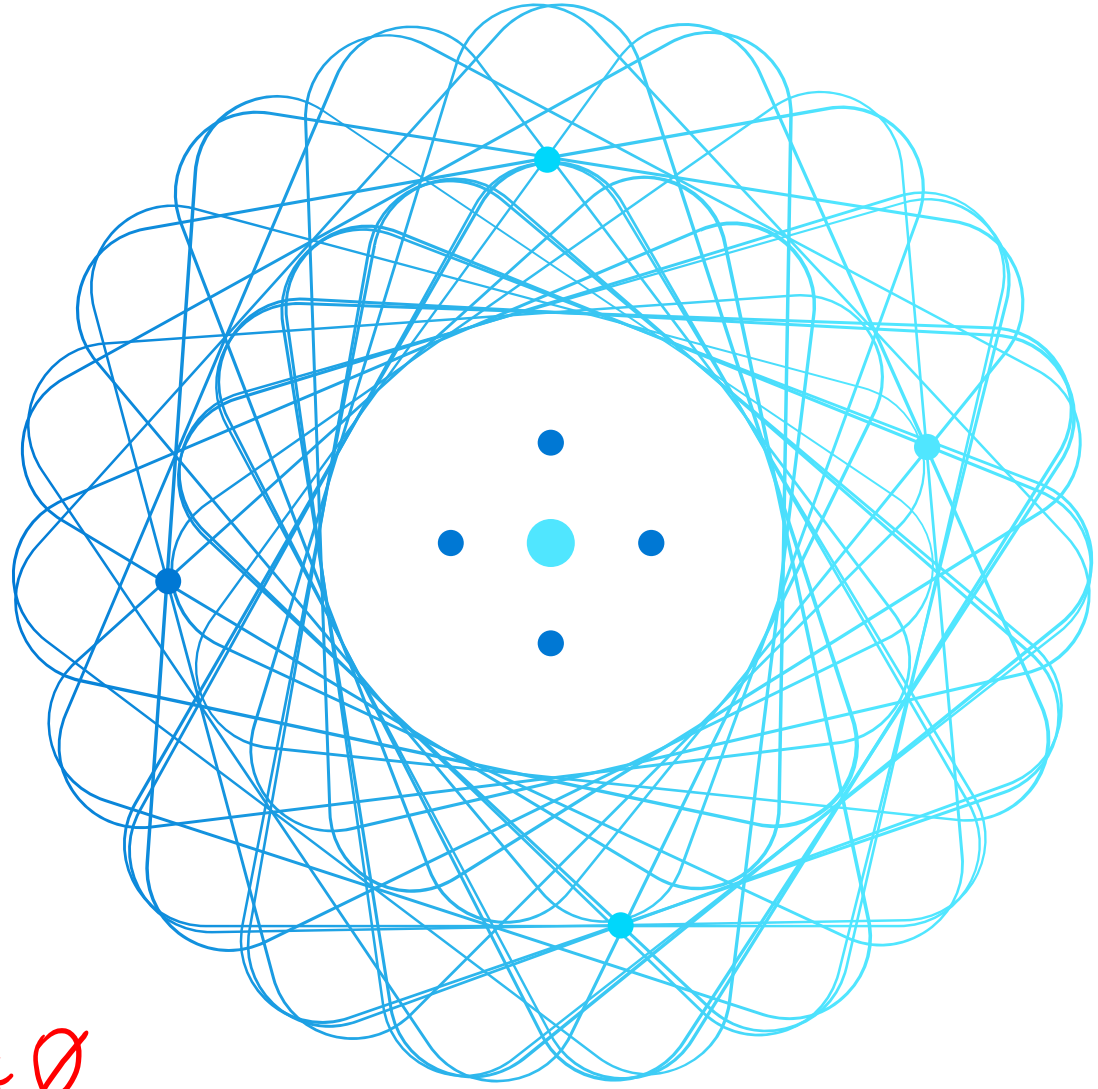
Microsoft Azure

Guten Morgen!

AZ-040

Automating Administration with PowerShell



Tipp:

github.com/www42/az-040

Course outline

Module 1: Getting started with Windows PowerShell

Module 2: Windows PowerShell for local systems administration

Module 3: Working with the Windows PowerShell pipeline

Module 4: Using PSProviders and PSDrives

Module 5: Querying management information by using CIM and WMI

Module 6: Working with variables, arrays, and hash tables

Module 7: Windows PowerShell scripting

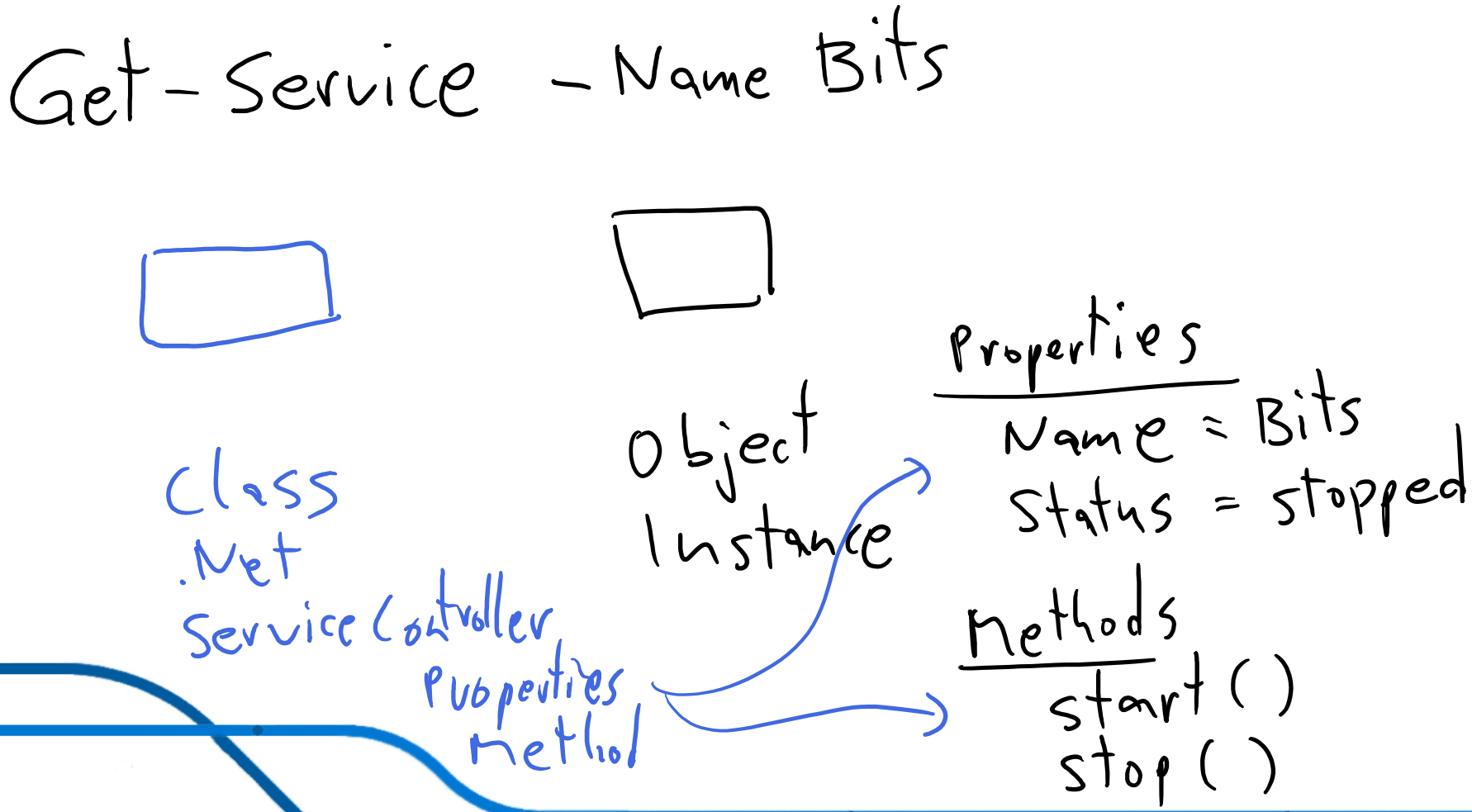
Module 8: Administering remote computers with Windows PowerShell

Module 9: Managing Azure resources with PowerShell

Module 10: Managing Microsoft 365 services with PowerShell

Module 11: Using background jobs and scheduled jobs

Module 3: Working with the Windows PowerShell pipeline



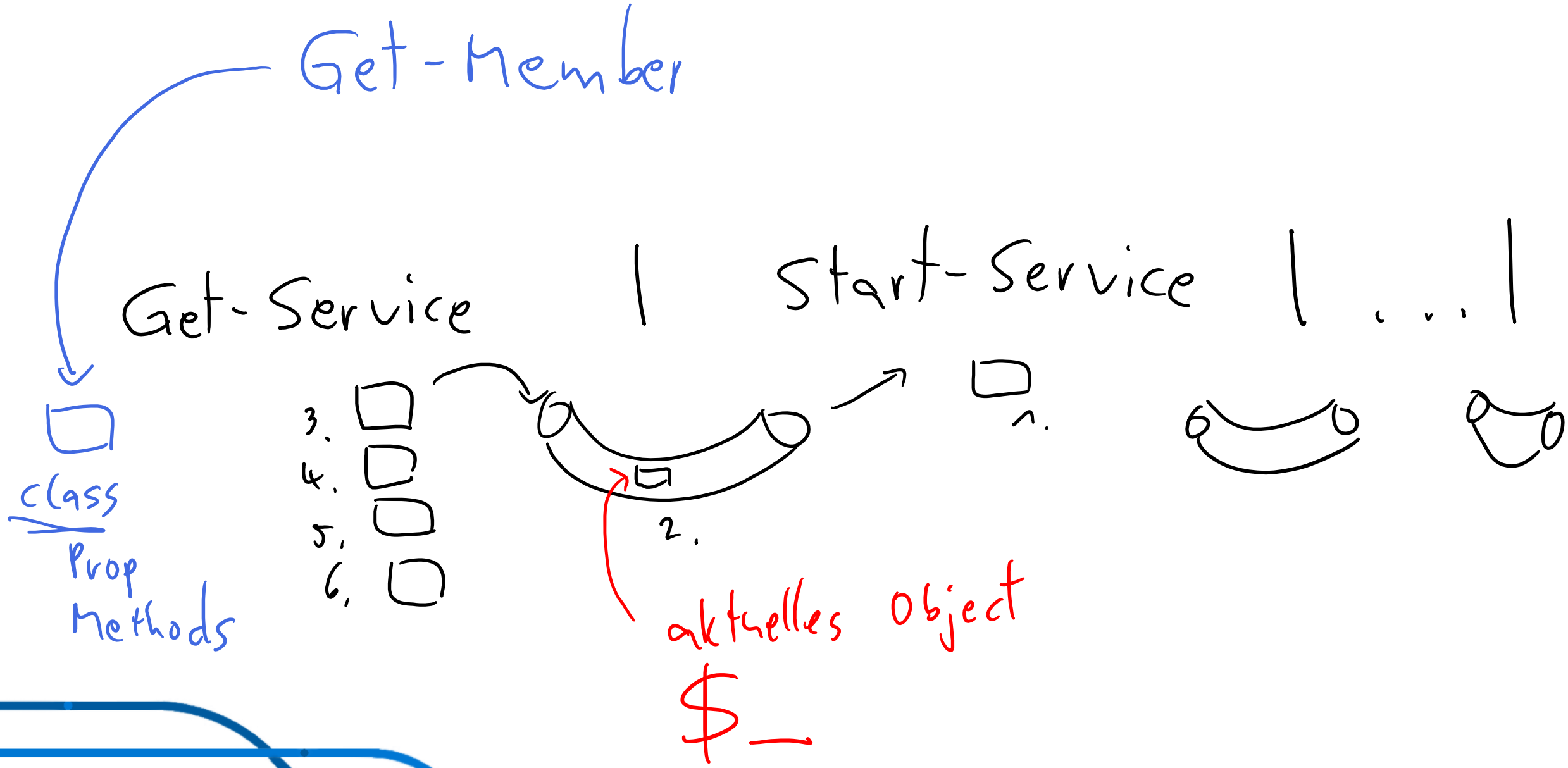
Module overview

In this module, you'll learn the purpose and use of the PowerShell pipeline. You'll use the pipeline to sort, filter, enumerate and display output data for PowerShell cmdlets.

Lessons:

- Lesson 1: Understand the pipeline
- Lesson 2: Select, sort, and measure objects
- Lesson 3: Filter objects out of the pipeline
- Lesson 4: Enumerate objects in the pipeline
- Lesson 5: Send pipeline data as output
- Lesson 6: Pass pipeline objects

Lesson 1: Understand the pipeline



Lesson 1 overview

In this lesson, you'll learn about the Windows PowerShell pipeline and some basic techniques for running multiple commands in it. Running commands individually is both cumbersome and inefficient. Understanding how the pipeline works is fundamental to your success in using Windows PowerShell for administration.

Topics:

- What is the pipeline?
- Pipeline output
- Discovering object members
- Demonstration: Reviewing object members
- Formatting pipeline output
- Demonstration: Formatting pipeline output

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**.

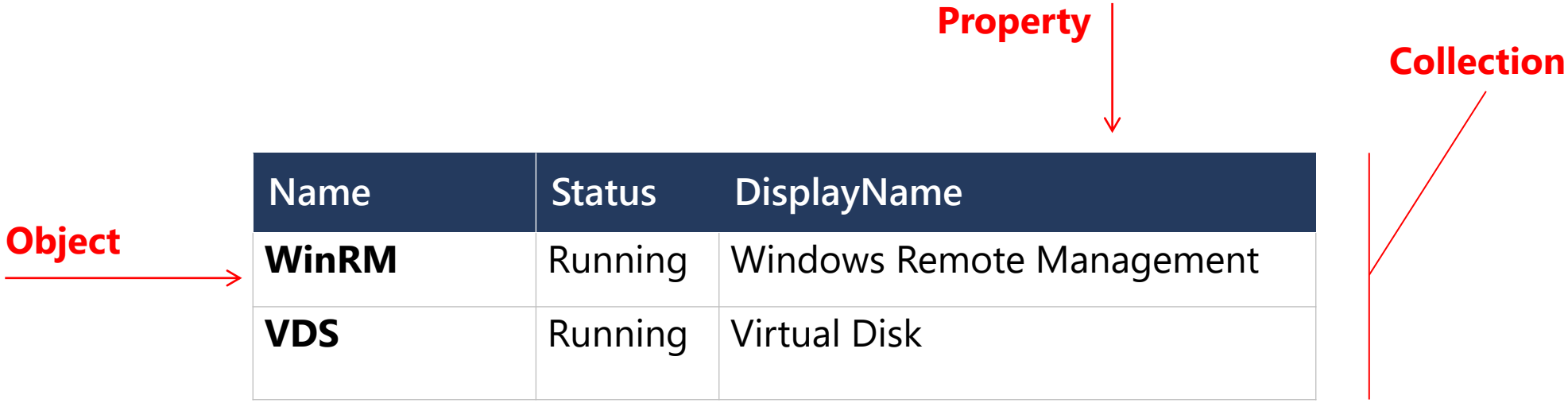
What is the pipeline? (Slide 2)

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 instance, 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.

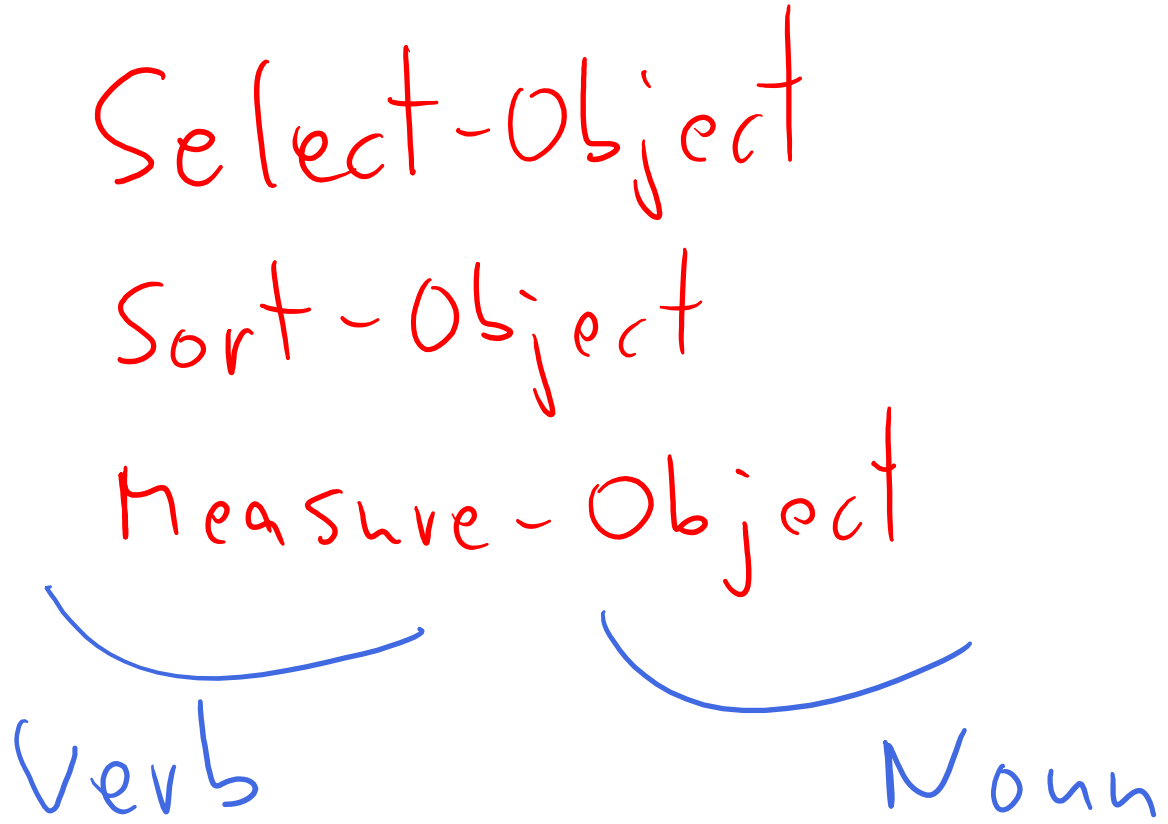
Demonstration: Reviewing object members

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.

Lesson 2: Select, sort, and measure objects



Lesson 2 overview

In this lesson, you'll learn to manipulate objects in the pipeline by using commands that sort, select, and measure objects. Selecting, sorting, and measuring objects is essential to successfully creating automation in Windows PowerShell.

Topics:

- Sorting objects by a property
- Demonstration: Sorting objects
- Measuring objects
- Demonstration: Measuring objects
- Selecting a subset of objects
- Selecting object properties
- Demonstration: Selecting objects
- Creating calculated properties
- Demonstration: Creating calculated properties

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

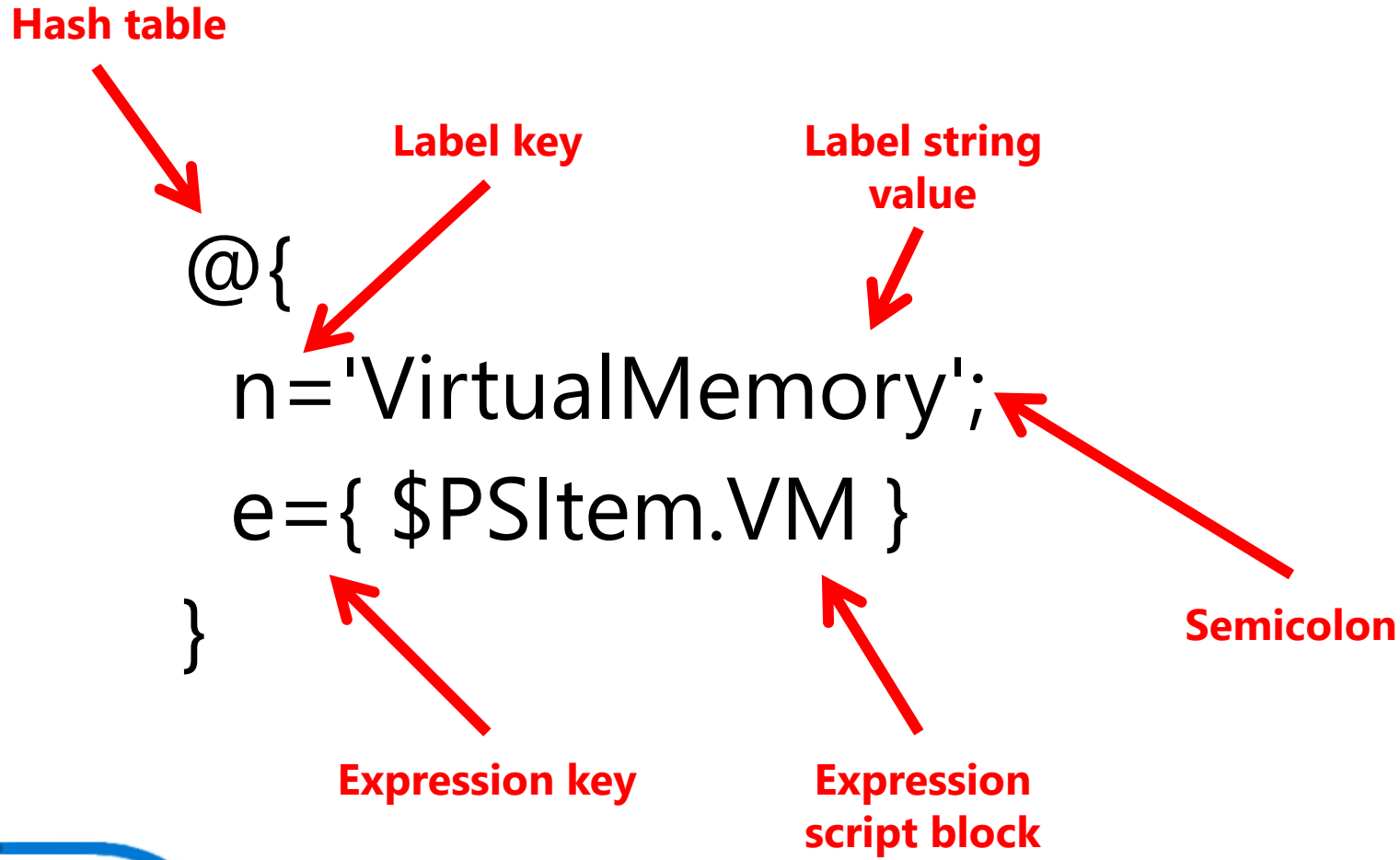
In this demonstration, you'll learn several ways to use the **Select-Object** command.

Creating calculated properties

- Calculated (custom) properties let you choose the output label and contents.
- Each calculated property works like a single regular property in the property list accepted by **Select-Object**.
- Create calculated properties by using a specific syntax:
  - **Label** defines the property name.
  - **Expression** defines the property contents.
  - Within the expression, **$PSItem (or $_)** refers to the piped-in object.

Creating calculated properties (Slide 2)

Calculated property hash table



Creating calculated properties (Slide 3)

To format calculated properties:

- Use shortcuts for specific memory amounts:
  - **KB** for kilobytes
  - **MB** for megabytes
  - **GB** for gigabytes
  - **TB** for terabytes
  - **PB** for petabytes
- Use the **-F** format operator to format numbers with a specified number of decimal places.

Creating calculated properties (Slide 4)

Example: Calculated properties

```
Get-Volume |
Select-Object –Property DriveLetter,
 @{
   n='Size(GB)';
   e={'{0:N2}' -f ($PSItem.Size / 1GB)}
 },
 @{
   n='FreeSpace(GB)';
   e={'{0:N2}' -f ($PSItem.SizeRemaining / 1GB)}
 }
```

Demonstration: Creating calculated properties

In this demonstration, you'll learn how to use **Select-Object** to create calculated properties. You'll then learn how those calculated properties behave like regular properties.

Lesson 3: Filter objects out of the pipeline

Lesson 3 overview

In this lesson, you'll learn how to filter objects out of the pipeline by using the **Where-Object** cmdlet to specify various criteria. This differs from the ability of **Select-Object** to select several objects from a collection because it provides more flexibility.

With this new technique, you'll be able to keep or remove objects based on criteria of almost any complexity.

-eq -gt -le -like

Topics:

- Comparison operators
- Basic filtering syntax
- Advanced filtering syntax
- Demonstration: Filtering
- Optimizing filtering performance

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 object piped into the command.
- This won't work: `Get-Service | Where Name.Length -gt 5`

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 -FilterScript {$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

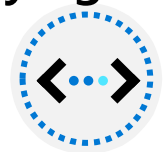
In this demonstration, you'll learn various ways to filter objects out of the pipeline.

Optimizing filtering performance

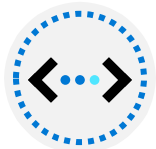
- To improve performance, move filtering as close to the beginning of the command line as possible.
- Some commands have parameters that can filter for you, so whenever possible, use those parameters instead of **Where-Object**.

Lab A: Using PowerShell pipeline

Exercise 1: Selecting, sorting, and displaying data



Exercise 2: Filtering objects



Sign-in information for the exercises:

Virtual machines:

- **AZ-040T00A-LON-DC1**
- **AZ-040T00A-LON-SVR1**
- **AZ-040T00A-LON-CL1**

Username: **Adatum\Administrator**

Password: **Pa55w.rd**

Lab scenario

Part of your administrative tasks at Adatum Corporation is to configure advanced PowerShell scripts. You need to ensure that you understand the foundation of working with the PowerShell pipeline by sorting, filtering, enumerating, and converting objects.

Lab-review questions

List the basic formatting commands and explain why you might use each one.

In the second exercise of this lab, were you able to achieve the goal without using the **Where-Object** command?

Lab-review answers

List the basic formatting commands and explain why you might use each one.

Format-List allows you to review many properties for an object at one time. The command truncates the property list when you also use **Format-Table**.

Format-Table creates output in a compact format that makes it easier to review more than one property for multiple objects. It also makes it possible to compare values among properties.

Format-Wide allows you to review many instances of a single property, such as names, in as compact a format as possible.

In the second exercise of this lab, were you able to achieve the goal without using the **Where-Object** command?

You should have been able to. The **Get-ADUser** command has *-Filter* and *-SearchBase* parameters that provide the filtering functionality that you needed. Using **Where-Object** would have been inefficient and incorrect in this scenario.

Lesson 4: Enumerate objects in the pipeline

Lesson 4 overview

In this lesson, you'll learn how to enumerate objects in the pipeline so that you can work with one object at a time during automation. Enumerating objects builds on the skills you've already learned and is a building block for creating automation scripts.

Topics:

- Purpose of enumeration
- Basic enumeration syntax
- Demonstration: Basic enumeration
- Advanced enumeration syntax
- Demonstration: Advanced enumeration

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.

Lesson 5: Send pipeline data as output

Lesson 5 overview

When you provide information about your network infrastructure, it’s often a requirement that you provide the information in specific formats. This might mean using a format for displaying on the screen, for printing a hard copy, or for storing in a file for later use. In this lesson, you'll learn how to send pipeline data to files and in various output formats.

Topics:

- Writing output to a file
- Converting output to CSV
- Converting output to XML
- Converting output to JSON
- Converting output to HTML
- Demonstration: Exporting data
- Additional output options

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.

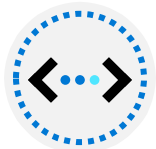
Additional output options

- **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.

Lab B: Using PowerShell pipeline

Exercise 1: Enumerating objects

Exercise 2: Converting objects



Sign-in information for the exercises:

Virtual machines:

- **AZ-040T00A-LON-DC1**
- **AZ-040T00A-LON-SVR1**
- **AZ-040T00A-LON-CL1**

Username: **Adatum\Administrator**

Password: **Pa55w.rd**

Lab scenario

Part of your administrative tasks at Adatum Corporation is to configure advanced PowerShell scripts. You need to ensure that you understand the foundation of working with the PowerShell pipeline by sorting, filtering, enumerating, and converting objects.

Lab-review question

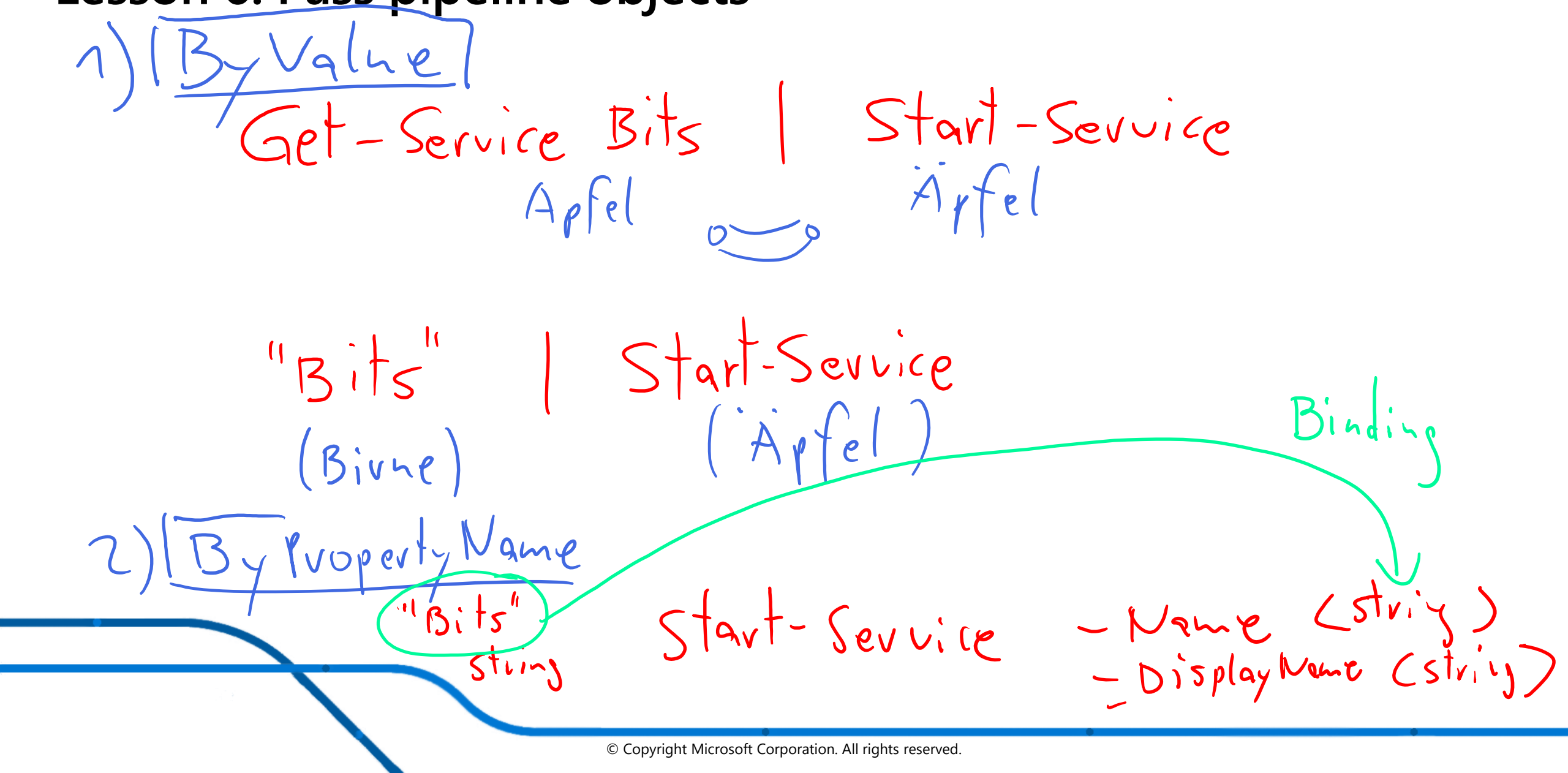
Can you use **ConvertTo-Csv** or **Export-Csv** to create a file delimited by a character other than a comma? For example, can you create a tab-delimited file?

Lab-review answer

Can you use **ConvertTo-Csv** or **Export-Csv** to create a file delimited by a character other than a comma? For example, can you create a tab-delimited file?

Yes. Both commands have a *-Delimiter* parameter that changes the delimiter used for the file. Use "`t" to specify a tab character and note that the quotation marks (" ") are required.

Lesson 6: Pass pipeline objects



Lesson 6 overview

In this lesson, you will learn how the Windows PowerShell command-line interface passes objects from one command to another in the pipeline. Windows PowerShell has two techniques it can use: passing data ByValue and ByPropertyName.

Topics:

- Pipeline parameter binding
- Identifying ByValue parameters
- Passing data by using ByValue
- Demonstration: Passing data by using ByValue
- Passing data by using ByPropertyName
- Identifying ByPropertyName parameters
- Demonstration: Passing data by using ByPropertyName

Lesson 6 overview (Slide 2)

Topics:

- Using manual parameters to override the pipeline
- Demonstration: Overriding the pipeline
- Using parenthetical commands
- Demonstration: Using parenthetical commands
- Expanding property values
- Demonstration: Expanding property values

Pipeline parameter binding

- Commands accept input only from one parameter:

```
Get–ADUser –Filter {Name –eq 'Perry Brill'} |Set–ADUser –City Seattle
```

- In the preceding example, two parameters are specified for **Set-ADUser**.
  - The one parameter you notice is *–City*.
  - Another parameter is used invisibly as part of *pipeline parameter binding*.
- Two techniques for pipeline parameter binding:
  - **ByValue** is always tried first.
  - **ByPropertyName** is tried if **ByValue** fails.

Identifying ByValue parameters

- The help file for a command indicates the parameters that can accept the pipeline input **ByValue**

```
Required?                    false
Position?                    Named
Default value                None
Accept pipeline input?       true (ByValue)
Accept wildcard characters?  false
```

Passing data by using ByValue

String objects in the pipeline

"BITS","WinRM" | Get-Service - Name

Attaches invisibly to the parameter that accepts String objects from the pipeline ByValue

Passing data by using ByValue (Slide 2)

- **Object** and **PSObject** are generic object types
- If a parameter accepts one of these object types, the parameter will accept any kind of object
- If the parameter accepts pipeline input **ByValue**, it can accept any kind of object from the pipeline
- This configuration is how **Sort-Object**, **Select-Object**, **Where-Object**, and many other commands work

Demonstration: Passing data by using ByValue

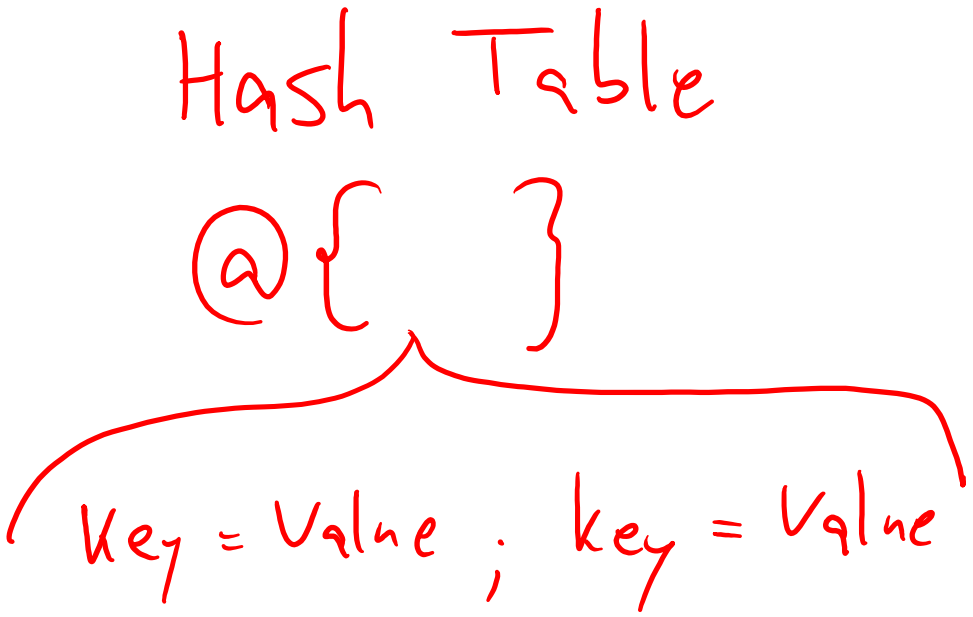
In this demonstration, you'll learn how Windows PowerShell performs pipeline parameter binding ByValue.

Passing data by using PropertyName

```
Get-ADComputer -Filter * | Select-Object
```

```
@{
n='ComputerName';
e={$PSItem.Name}
}
```

```
Get-Process
```



Object properties

- Name
- FullName
- Enabled
- Description

Calculated property

- ComputerName = $PSItem.Name

Command parameters

- Id
- Name

Identifying ByPropertyName parameters

- The full Help for a command lists the parameters that accept pipeline input by using **ByPropertyName**

```
Required?                    false
Position?                    named
Default value                Local computer
Accept pipeline input?       true (ByPropertyName)
Accept wildcard characters?  false
```

Demonstration: Passing data by using ByPropertyName

In this demonstration, you'll learn how to use ByPropertyName parameters.

Using manual parameters to override the pipeline

- The shell first looks to see what parameter it can use for pipeline parameter binding:
  - But if one of those parameters is specified manually, binding stops

```
Get-Process -Name Notepad | Stop-Process -Name Notepad
```

- The manual use of *–Name* overrides the pipeline and the pipeline input is not used; you will see an error

Demonstration: Overriding the pipeline

In this demonstration, you will see an example of an error when you manually specify a parameter that Windows PowerShell would usually have used in pipeline parameter binding.

Using parenthetical commands

- Any command or commands in parentheses will be run first
- The results will be inserted in place of the parenthetical command
- Works with any parameter if the command produces the kind of object that the parameter expects

```
Get-ADGroup "London Users" |
Add-ADGroupMember -Members (Get-ADUser -Filter {City -eq 'London'})
```

Demonstration: Using parenthetical commands

In this demonstration, you'll learn how to use parenthetical commands.

Expanding property values

- The **–ExpandProperty** parameter of **Select-Object** expands, or extracts, the contents of a single property
- Instead of returning an object with many properties, the command returns a simpler value:
  - Converts multiple value properties to a series of single items
  - Additionally, works when piping commands

Demonstration: Expanding property values

In this demonstration, you'll learn how to use parameter expansion to provide input from a parenthetical command.