Slide 1



Slide 2



Slide 3



**PowerShell Overview**

* **Scripting language for Windows**: PowerShell is a powerful scripting language designed for system administration and automation on Windows. It allows administrators to automate tasks using cmdlets, scripts, and even integrate with APIs.
* **Available since Windows Server 2008**: PowerShell was introduced with Windows Server 2008 as a robust alternative to batch scripting. Since then, it has become the default choice for system administrators and developers.
* **Object-based pipeline**: Unlike traditional scripting languages, PowerShell works with **objects** rather than text. This makes it easier to manipulate data because each element passed through the pipeline retains its object properties, enabling more complex and structured data handling.
* **Built on .NET**: PowerShell is built on the .NET Framework, which gives it access to a wide range of libraries and capabilities, including integrating with Windows APIs and .NET applications, making it highly extensible.
* **Currently at version 5**: As of this reference, PowerShell is currently at version 5. PowerShell has continued to evolve, with newer versions improving performance, security, and cross-platform capabilities (e.g., PowerShell Core).

Slide 4



Slide 5



**Key Concepts in PowerShell**

* **Commands**:
* PowerShell commands are called **cmdlets** (pronounced "command-lets").
* They follow a Verb-Noun naming convention, such as Get-Process, Set-Item, and Remove-Item.
* Cmdlets are case-insensitive, and they are used to perform various system administration tasks.
* **Syntax**:
* PowerShell syntax is straightforward and is based on the concept of cmdlets, piping, and using objects.
* Commands typically follow the pattern Verb-Noun, with parameters provided in a clear, structured format.
* Example: Get-Service -Name "W32Time"
* **Help System**:
* PowerShell includes an extensive built-in help system. You can use the Get-Help cmdlet to access detailed documentation for cmdlets, scripts, and modules.
* Example: Get-Help Get-Process -Full shows detailed help for the Get-Process cmdlet.
* **Pipeline**:
* PowerShell allows you to pass the output of one cmdlet as the input to another using the **pipeline** (|).
* Since PowerShell works with **objects** instead of text, each element passed through the pipeline retains its properties, making it easy to manipulate and filter data.
* Example: Get-Process | Where-Object { $\_.CPU -gt 100 }
* **Drive System**:
* PowerShell uses a **drive** system to represent data stores, like filesystems, registries, and certificates, as though they are disks.
* You can navigate and manipulate these data stores as if they were traditional filesystems.
* Example: Set-Location -Path "HKLM:\Software"
* **Modules**:
* PowerShell modules are packages of cmdlets, providers, functions, workflows, and scripts that can be imported to extend functionality.
* You can manage, install, and remove modules using cmdlets like Install-Module and Remove-Module.
* Example: Install-Module -Name Az to install the Azure PowerShell module.

Slide 6



**PowerShell Key Features**

* **Verb-Noun Syntax**:
* PowerShell uses a consistent Verb-Noun syntax for all its commands, making it easier to understand and use.
* Example: Get-Process, Set-Location, New-Item.
* This structure helps in categorizing and discovering commands for specific tasks (e.g., Get for retrieval, Set for modifications).
* **Discoverable**:
* PowerShell is highly discoverable, meaning it provides tools to help users find the cmdlets they need.
* You can use commands like Get-Command to discover available cmdlets and functions.
* Example: Get-Command \*Service\* helps you find all commands related to services.
* **Parameters**:
* Each cmdlet comes with a set of parameters that allow you to customize its execution.
* Parameters provide flexibility in how commands are run and can often be discovered using Get-Help.
* Example: Get-Process -Name "chrome" retrieves a specific process using the -Name parameter.

Slide 7



**Command Format in PowerShell**

* **Verb-Noun Structure**:
* PowerShell commands follow a consistent Verb-Noun structure.
* Example: Get-Process:
* **Verb**: Standard action you want to perform (Get to retrieve information).
* **Noun**: Singular object or resource you are interacting with (Process represents system processes).
* This clear format makes commands intuitive and easy to understand.
* **Frequently Prefixed Nouns**:
* Some cmdlets use a prefixed noun to represent specific tools or services.
* Example: Get-SCSMObject:
* Here, **SCSM** (System Center Service Manager) is the prefix, indicating a specific resource within the service manager toolset.
* PowerShell allows for such prefixes to distinguish between cmdlets from different services or modules.
* **98 Standard Verbs**:
* PowerShell includes a standardized set of **98 common verbs** that are recommended to ensure consistency and discoverability.
* You can use Get-Verb to list all available verbs, which are grouped into common categories like Add, Set, Remove, etc.
* These verbs standardize the way actions are performed across cmdlets, making it easier for users to learn and use new cmdlets.

Slide 8



Slide 9



**Common PowerShell Parameters**

PowerShell includes a set of common parameters that can be used across most cmdlets to control how information is processed, displayed, and handled. These common parameters provide flexibility and control when working with PowerShell scripts.

* **-Debug (db)**:
* Used to display debug messages during command execution. Helps in troubleshooting and understanding what the command is doing.
* Example: Get-Process -Debug
* **-ErrorAction (ea)**:
* Controls how PowerShell responds to errors. Common values include Continue, Stop, SilentlyContinue, and Inquire.
* Example: Get-Process -ErrorAction Stop
* **-ErrorVariable (ev)**:
* Stores error information in a specified variable for further processing or logging.
* Example: Get-Process -ErrorVariable errorVar
* **-InformationAction**:
* Determines how PowerShell handles information messages, similar to -ErrorAction. It can suppress or display information messages.
* Example: Get-Process -InformationAction SilentlyContinue
* **-InformationVariable**:
* Captures information messages in a variable for later use.
* Example: Get-Process -InformationVariable infoVar
* **-OutVariable (ov)**:
* Captures output into a variable instead of sending it to the console.
* Example: Get-Process -OutVariable outVar
* **-OutBuffer (ob)**:
* Specifies the number of objects to buffer before sending them to the pipeline.
* Example: Get-Process -OutBuffer 10
* **-PipelineVariable (pv)**:
* Creates a variable that holds the current pipeline object, allowing you to reference it within the pipeline.
* Example: Get-Process | ForEach-Object { $\_ - PipelineVariable proc }
* **-Verbose (vb)**:
* Displays additional details about the command execution process. Useful for understanding what a script is doing in greater detail.
* Example: Get-Process -Verbose
* **-WarningAction (wa)**:
* Similar to -ErrorAction, this controls how PowerShell handles warning messages (Continue, SilentlyContinue, etc.).
* Example: Get-Process -WarningAction SilentlyContinue
* **-WarningVariable (wv)**:
* Captures warning messages into a specified variable.
* Example: Get-Process -WarningVariable warnVar

Slide 10



* **Introduction to Parameter Sets:**
  + Start by explaining that PowerShell commands can have multiple parameter sets, each offering a different way to run the command depending on the parameters used.
  + Highlight that parameter sets are useful for organizing parameters into logical groups, making the command syntax easier to understand and use.
* **Parameter Set 1:**
  + Describe the first parameter set for the Start-Process command, which includes options like -FilePath, -LoadUserProfile, -NoNewWindow, and others.
  + Mention that this set is used when you want to run a process with a specified file path and may involve additional parameters like credentials or redirection of output.
  + Focus on specific options like -WindowStyle to control how the window is displayed (e.g., Normal, Minimized, Maximized).
* **Parameter Set 2:**
  + Discuss the second parameter set, which is simpler and focused on running a process with an argument list and window style.
  + This set doesn’t include some of the more advanced parameters like credential handling or output redirection, which are present in Set 1.
* **Importance of Parameter Sets:**
  + Emphasize that PowerShell automatically determines which parameter set to use based on the parameters provided by the user.
  + Parameter sets help avoid errors by ensuring that incompatible parameters are not used together.
* **Practical Usage:**
  + Share an example scenario where using different parameter sets is helpful. For example, if a user wants to run a process either with or without additional credentials or in different window styles.

Slide 11



**Command Discovery in PowerShell**

* **Using Get-Command**:
* PowerShell provides a built-in cmdlet called Get-Command that helps you discover available cmdlets, functions, workflows, and aliases in your environment.
* This cmdlet is especially useful when you are unsure of the exact command but know part of its name or its noun or verb.
* **Example: Searching with Noun**:
* In the example shown, the command Get-Command -Noun Process\* is used to find all cmdlets related to the noun **Process**.
* This search yields several cmdlets:
* Debug-Process: For debugging a process.
* Get-Process: For retrieving information about processes.
* Start-Process: For starting a new process.
* Stop-Process: For stopping a running process.
* Wait-Process: For waiting for a process to stop before continuing.
* **Filtering with Wildcards**:
* The use of Process\* with an asterisk (\*) wildcard shows how you can filter results for cmdlets where the noun starts with "Process". This is helpful when you are not sure of the full cmdlet name.
* Wildcards can be used for verbs as well (e.g., Get-Command -Verb Get\* to list all cmdlets with the Get verb).
* **Practical Use**:
* This command discovery technique makes PowerShell highly user-friendly and helps users quickly locate relevant cmdlets for specific tasks without having to memorize exact command names.

Slide 12



.\Start-Demo.ps1 –File .\Help.txt

Slide 13



.\Start-Demo –File .\Variables.txt

**PowerShell Variables**

* **Variable Names**:
* In PowerShell, variables are always prefixed with a $ symbol.
* Example: $Name, $Age, and $Birthday are variables in the example shown.
* Variable names are case-insensitive, but it is good practice to follow a consistent naming convention.
* **Values**:
* A variable holds a value, which can be a string, number, object, or other data types.
* Example:
* $Name holds the string 'Adam Driscoll'.
* $Age holds the integer 29.
* $Birthday holds a **DateTime** object representing a specific date.
* **Data Types**:
* PowerShell is dynamic in handling data types but allows explicit type declaration using square brackets ([]).
* Example: [DateTime] is used to cast the value '5/6/1986' as a DateTime object.
* Other data types like [int], [string], and [array] can also be specified for more control over variable types.
* **Accessing Help**:
* PowerShell provides extensive help for working with variables.
* You can use Get-Help about\_Variables to learn more about variable usage and types in PowerShell.

Slide 14



.\Start-Demo.ps1 -File .\Strings.txt

**PowerShell Strings**

* **Literal Strings**:
* Literal strings in PowerShell are enclosed in **single quotes (')**.
* Example: $Name = 'Literal String'
* In this case, the string is interpreted exactly as written. Variables or special characters inside the string will **not** be expanded or interpreted.
* This type of string is useful when you want to preserve the exact content of a string without any processing.
* **Expandable Strings**:
* Expandable strings are enclosed in **double quotes (")**.
* Example: "Expandable string $Name"
* In an expandable string, PowerShell will interpret the content and replace variables or special characters within the string. For instance, $Name will be replaced with the actual value of the $Name variable.
* This is useful when you want to dynamically include variable values within a string.
* **Here-Strings**:
* A **Here-String** is used to create multi-line strings that preserve new line characters and formatting. It starts with @' and ends with '@ (for literal Here-Strings) or @" and "@ (for expandable Here-Strings).
* Example:powershell
* Copy code
* @' Here string will preserve new line characters '@
* Here-Strings are especially useful when working with large blocks of text or multi-line content, such as configuration files or large output text.
* **Expandable Here-String**:
* If you want to include variable expansion within a Here-String, use **double quotes** (@" and "@).
* Example:powershell
* Copy code
* @" The value of Name is $Name "@
* This would expand the $Name variable, similar to a regular expandable string, but with multi-line support.
* **Help with Strings**:
* PowerShell provides detailed help documentation on working with strings.
* You can use Get-Help about\_Quoting\_Rules to learn more about how PowerShell handles strings and when to use single quotes versus double quotes.

Slide 15



**Overview:**

* This slide demonstrates how conditional operators like if, elseif, and else are used in scripting.
* The condition checks if certain criteria ($IsComputer or $IsUser) are true and executes corresponding code blocks based on those conditions.

**Key Points:**

* **Conditions** (Highlighted in Blue Arrows):
* Each if, elseif, and else statement evaluates a condition to determine which code block to execute.
* For example, if ($IsComputer) checks if the $IsComputer variable is true.
* **Expression Body** (Highlighted in Red Arrows):
* The expression body is the block of code that runs if the condition evaluates to true.
* In this example, when the condition $IsComputer is true, the script returns "Computer".
* Similarly, if $IsUser is true, it returns "User".
* If neither condition is met, the else block returns "Unknown".
* **Flow of Logic:**
* First, the script checks if the device is a computer.
* If that condition fails, it checks if it is a user.
* If both conditions fail, it defaults to an else block, returning "Unknown".
* **PowerShell Context:**
* This structure is commonly used in scripting languages like PowerShell.
* A good resource to understand more about if conditions in PowerShell is Get-Help about\_If, which provides additional details about conditional logic.

**Conclusion:**

* Conditional statements like if, elseif, and else help create decision-making logic in scripts.
* This simple example helps visualize how different conditions lead to different outcomes.

Slide 16



**Overview:**

* This slide illustrates how comparison operators work in scripting to evaluate conditions and return True or False.
* Comparison operators are essential in decision-making logic, especially when dealing with strings, numbers, or arrays.

**Key Points:**

* **Equality and Inequality Comparisons:**
* Line 1: 'Adam' -eq 'Bob' checks if the two strings are equal. In this case, it evaluates to False because "Adam" is not equal to "Bob".
* Line 2: 'Adam' -ne 'Bob' checks if the two strings are not equal. Since "Adam" and "Bob" are different, the result is True.
* **Less Than and Greater Than:**
* Line 3: 1 -lt 10 checks if 1 is less than 10, which returns True.
* Line 4: 1 -gt 11 checks if 1 is greater than 11, which evaluates to False.
* **Pattern Matching Operators:**
* Line 5: 'Adam' -match 'Ad' uses the -match operator to check if the string "Adam" matches the pattern 'Ad' (which it does), so it returns True.
* Line 6: 'Adam' -like 'Ad\*' uses the -like operator with a wildcard, checking if "Adam" starts with 'Ad'. This returns True because the pattern matches.
* **Array Containment Check:**
* Line 7: @('Adam', 'Bob') -Contains 'Bob' checks if the array @('Adam', 'Bob') contains the string 'Bob'. Since "Bob" is part of the array, this returns True.

**Additional Details:**

* **Equality (-eq)**: Compares values for equality.
* **Inequality (-ne)**: Checks if two values are not equal.
* **Less Than (-lt)**: Compares numerical values for less-than condition.
* **Greater Than (-gt)**: Compares numerical values for greater-than condition.
* **Pattern Matching**:
* -match: Uses regular expressions to match patterns.
* -like: Uses simple wildcard patterns (\* for any characters).
* **Containment (-Contains)**: Checks if a value exists within an array.

**PowerShell Reference:**

* For further details about comparison operators in PowerShell, refer to the Get-Help about\_Operators command, which will give a detailed explanation of different operators.

Slide 17



**Overview:**

* This slide demonstrates basic operations with arrays in PowerShell. Arrays are a collection of elements that allow for storage and manipulation of multiple values in a single variable.

**Key Points:**

* **Array Initialization:**
* Line 1: $Names = @('Adam', 'Bob', 'Bill')
* This initializes an array called $Names with three elements: 'Adam', 'Bob', and 'Bill'.
* The @() syntax is used to declare an array in PowerShell.
* Arrays can store strings, numbers, or other objects.
* **Adding Elements to an Array:**
* Line 2: $Names += 'John'
* The += operator is used to append an element ('John') to the existing $Names array.
* This operation dynamically increases the size of the array by adding 'John' as a new element at the end.
* Now the array becomes @('Adam', 'Bob', 'Bill', 'John').
* **Checking Array Length:**
* Line 3: $Names. Length
* The .Length property is used to find out how many elements are in the array.
* In this case, after appending 'John', the length of the array would be 4.
* This is a useful property to quickly assess how many items are stored in the array.

**PowerShell Reference:**

* For more details about arrays and their operations, the Get-Help about Arrays command provides additional documentation and examples.

Slide 18



**Overview:**

* This slide demonstrates three different types of loops commonly used in PowerShell scripting: foreach, while, and for.
* Loops are essential for executing repetitive tasks, iterating over collections, or repeating operations until a certain condition is met.

**Key Points:**

* **foreach Loop:**
* Syntax: foreach($Process in $Processes)
* The foreach loop iterates over each element in a collection, such as an array or object list.
* In this example, the loop iterates through the $Processes array, and for each element, it accesses the $Process.Id.
* This is useful when working with collections of data like processes, files, or other objects.
* **while Loop:**
* Syntax: while($i -lt 10)
* The while loop repeats its block of code as long as the specified condition is true.
* Here, it starts with $i = 0 and keeps incrementing $i ($i++) until $i is no longer less than 10.
* This is ideal when you need to repeat operations based on a condition that may change during execution.
* **for Loop:**
* Syntax: for($i = 0; $i -lt 10; $i++)
* The for loop provides an all-in-one syntax for initializing a counter, setting a condition, and incrementing the counter.
* This loop runs from $i = 0 to $i = 9, and the value of $i is printed during each iteration.
* The for loop is often preferred for situations where you need more control over the loop variables (initialization, condition, and iteration).

**Differences Between Loops:**

* **foreach** is ideal for iterating over collections or arrays.
* **while** loops are suited for situations where the number of iterations isn’t fixed, but a condition must be met.
* **for** loops are typically used when you have a set number of iterations with a defined loop counter.

**PowerShell Reference:**

* To learn more about these loops in PowerShell, you can use the following help commands:
* Get-Help about\_ForEach
* Get-Help about\_For
* Get-Help about\_While

Slide 19



**Overview:**

* This slide emphasizes the core concept of objects in PowerShell, which is an object-oriented scripting language. Unlike many traditional scripting languages that deal with text, PowerShell focuses on handling objects.

**Key Points:**

* **Objects Rather Than Text:**
* PowerShell commands return objects, not just plain text.
* Objects can represent complex data structures, like files, processes, services, and more.
* This provides greater flexibility when working with data, as objects contain information in a structured manner that can be easily manipulated.
* **Objects Have Properties and Methods:**
* Every object in PowerShell has properties and methods:
* **Properties** represent characteristics or attributes of an object. For example, a file object might have properties like Name, Size, and CreationTime.
* **Methods** are actions that an object can perform. For example, a process object may have methods to Start() or Stop() a process.
* This allows for more interactive and granular control over how data is processed in scripts.
* **Objects Can Be Formatted, Filtered, and Extended:**
* PowerShell provides tools to format objects for easier viewing using commands like Format-Table or Format-List.
* Objects can also be filtered with commands like Where-Object to retrieve specific data.
* You can even extend objects by adding new properties or methods to meet your scripting needs.

**PowerShell Reference:**

* To explore more about working with objects in PowerShell, use the command: Get-Help about\_Objects.

Slide 20



**Key Points:**

* **Using Get-Member:**
* The Get-Member cmdlet is a powerful tool in PowerShell that allows you to discover the properties and methods of an object.
* The command [DateTime]::Now | Get-Member is used to examine the members of the DateTime object, which represents the current date and time.
* **Understanding Output:**
* The output lists the **Name**, **MemberType**, and **Definition** of each member:
* **Name**: The name of the method or property (e.g., Add, AddDays, CompareTo).
* **MemberType**: Specifies whether the member is a method, property, etc.
* **Definition**: Describes the method, including its parameters (e.g., AddDays(double value) allows you to add a number of days to a date).
* This output helps identify what operations can be performed with the object and how to use its methods effectively.
* **Practical Use of Get-Member:**
* Developers and administrators can use Get-Member to explore unfamiliar objects or find available methods for scripting.
* In this case, the DateTime object provides methods to manipulate dates, such as AddDays, AddHours, and AddMonths, which can be used in automation tasks.

**PowerShell Reference:**

* You can use Get-Help Get-Member to get more information about the cmdlet and its usage.

Slide 21



.\Start-Demo.ps1 –File .\Pipeline.txt

**Key Points:**

* **Chain Commands Together:**
* The pipeline operator (|) is used to link multiple commands together.
* The output of one command is automatically fed into the next command.
* This allows for the composition of complex tasks in a simple, readable format.
* Example: Get-Process | Sort-Object CPU | Select-Object -First 5
* This command gets all processes, sorts them by CPU usage, and selects the top 5 processes.
* **Simplify Looping:**
* Pipelines eliminate the need for explicit loops in many cases, allowing you to process multiple objects without writing repetitive loop structures.
* Commands that support the pipeline automatically handle multiple objects.
* For example, Get-Process | Where-Object { $\_.CPU -gt 100 } will filter only the processes where CPU usage is greater than 100.
* **Deal with Sets of Objects Easier:**
* Pipelines make it easier to work with large sets of data or objects, by processing them in chunks or step by step.
* This avoids the need to manually handle individual items in a collection, improving script readability and efficiency.

**PowerShell Reference:**

* For more detailed information, you can refer to the Get-Help about\_Pipelines command in PowerShell to learn about the syntax and more advanced usage.

Slide 22



Slide 23



.\Start-Demo -File .\Modules.txt

Slide 24



Slide 25



Slide 26

