Microsoft

WorkshopPLUS - Windows PowerShell: Foundation Skills

Microsoft Services

Microsoft

Objects, Variables and Data Types

Microsoft Services

Learning Units covered in this Module

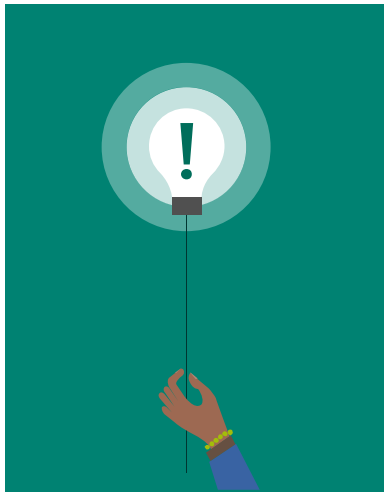
- Object Models
- Variables and Data Types

Object Models

Objectives

After completing Object Models, you will be able to:

- Describe objects and types



What is an object?

What is an Object?

- Structured Data
- Combines similar information and capabilities into one entity
- A collection of **parts** and **how to use** them

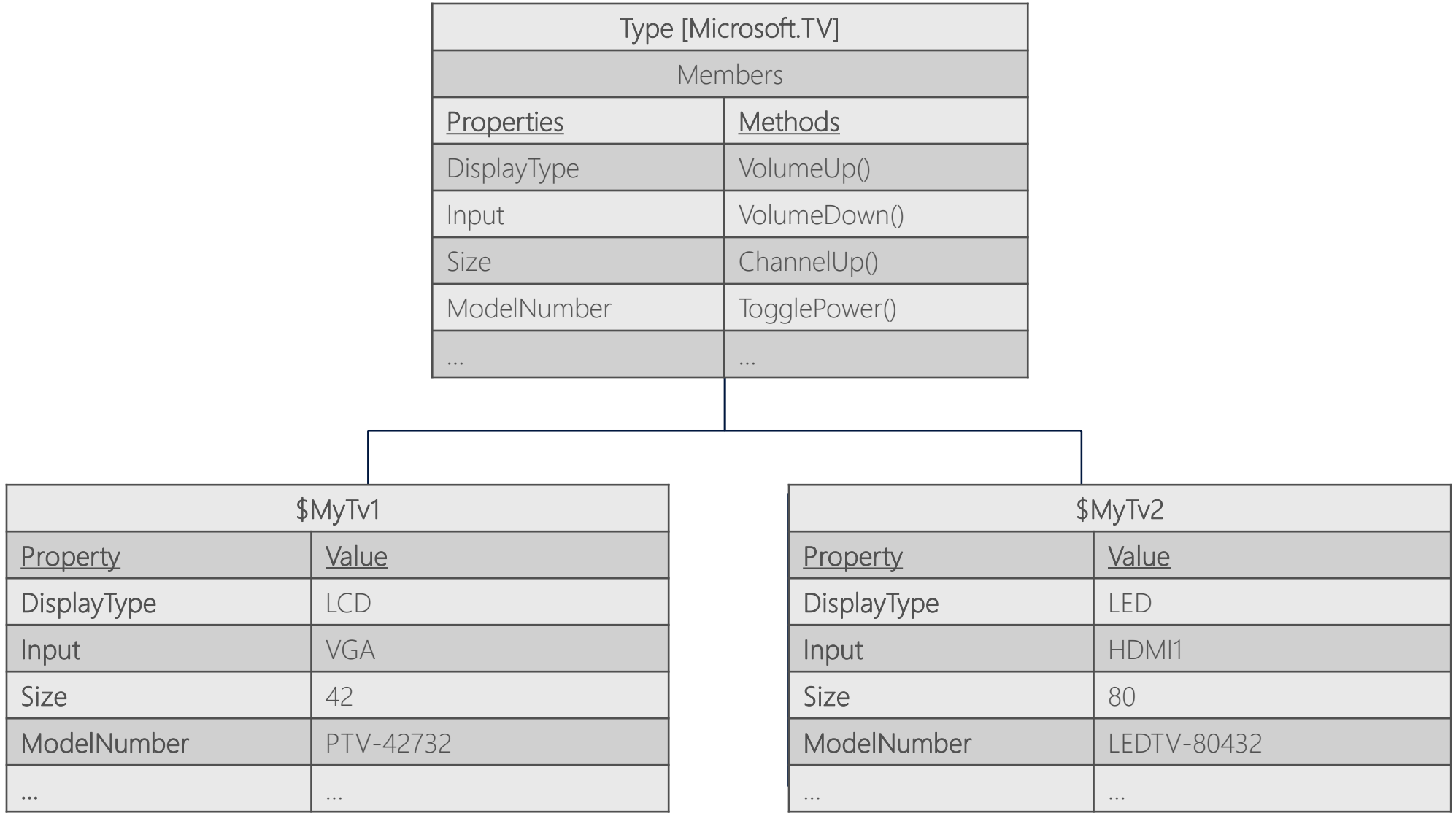
How Would You Model a TV?

Properties (Information)
Is it on?
Current Channel
Current Volume
Screen Size
Brand
Input
Screen Type

Methods (Actions)
Toggle Power
Channel Up
Channel Down
Volume Up
Volume Down
Change Input
Set Channel(<int>)

To change the channel to a particular one we have to pass in data (the channel number).

Understanding Instances



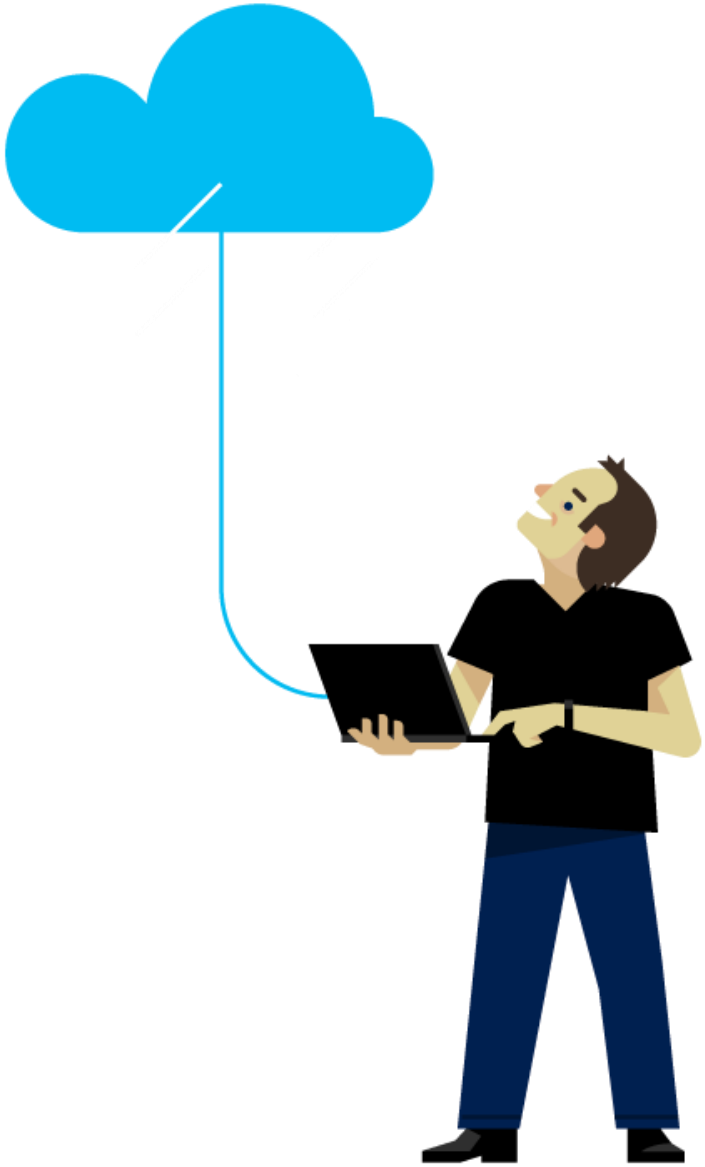
Object-Based Shell

- Everything is represented as an OBJECT
- OBJECTS have data fields (PROPERTIES) and procedures (METHODS)
- PROPERTIES and METHODS are collectively known as MEMBERS
- An OBJECT is an INSTANCE of a TYPE
- A TYPE represents a construct that defines a template of MEMBERS

Demonstration

PowerShell Objects

Questions?



Identify PROPERTIES and METHODS for an object

Get-Member

- Get-Member displays PROPERTIES and METHODS
- PROPERTIES are columns of information
- METHODS are actions that can be taken
- Typically used in pipeline

What Object Type Am I Using?

- Get-Member
  - Only shows type name
  - Any object can be passed or piped into Get-Member to retrieve type information in addition to Members list.
  - In the pipe the member type of the object thrown will be used as input. Might differ from the root object. (Example Array)

Get-Member

```
PS C:\> "" | Get-Member

 TypeName: System.String

Name         MemberType            Definition
----         ----------            ----------
…
```

What Object Type am I Using?

- Get-Type
  - All objects will have a “Get-Type” method which returns the type
  - “Get-Type” also returns detailed type information
  - The Return value is itself an object representing the type, it has a FullName property

GetType

```
PS C:\> ("").GetType().FullName
System.String

PS C:\> ("").GetType().Assembly
mscorlib, Version=4.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089

PS C:\> ("").GetType().Basetype
System.Object
```

Understanding Get-Member Definitions

Property Definition

```
PS C:\> Get-Item C:\Windows\System32\drivers\etc\hosts| Get-Member -Name
LastWriteTime

 TypeName: System.IO.FileInfo

Name           MemberType     Definition
----           ----------     ----------
LastWriteTime  Property       datetime LastWriteTime {get;set;}
```

This Property is a [datetime] type.

This Property can be get OR set.

```
PS C:\> $file = Get-Item C:\Windows\System32\drivers\etc\hosts
PS C:\> $file.LastWriteTime = (Get-Date)
PS C:\> Get-Item C:\Windows\System32\drivers\etc\hosts
Directory: C:\Windows\System32\Drivers\etc

Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----        4/21/2017  11:23 AM            894 hosts
```

Understanding Get-Member Definitions

Method Definition

```
PS C:\> Get-Item C:\Windows\notepad.exe | Get-Member -Name CopyTo

 TypeName: System.IO.FileInfo

Name         MemberType       Definition
----         ----------       ----------
CopyTo       Method           System.IO.FileInfo CopyTo(string destFileName),
                              System.IO.FileInfo CopyTo(string destFileName, bool overwrite)
```

Two Parameter Sets

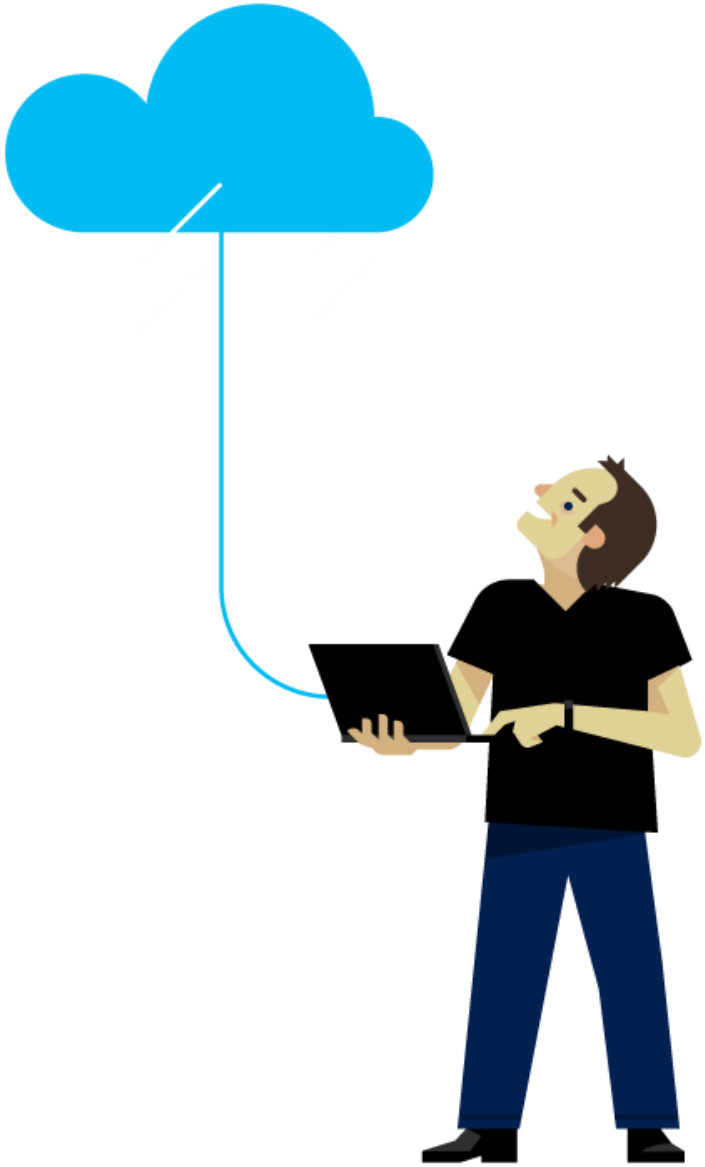
This Method RETURNS a System.IO.File info, which is the newly copied file.

```
PS C:\> $file = Get-Item C:\Windows\notepad.exe
PS C:\> $file.CopyTo("C:\Temp\notepad.exe", $True)

 Mode                 LastWriteTime         Length Name
 ----                 -------------         ------ ----
 -a----         7/16/2016   7:43 AM         243200 notepad.exe
```

Demonstration

Object Members and Object Types

Questions?



Identify the TYPE of an object

Object TYPE Returned by the Get-Date Cmdlet

```
PS C:\> Get-Date | Get-Member

TypeName: System.DateTime

Name                 MemberType     Definition
----                 ----------     ----------
Add                  Method         datetime Add(timespan
value)
AddDays              Method         datetime AddDays(double
value)
AddHours             Method         datetime AddHours(double
value)
…
```

- Get-Member displays the TYPE name

Object TYPE Returned By The Get-Date Cmdlet

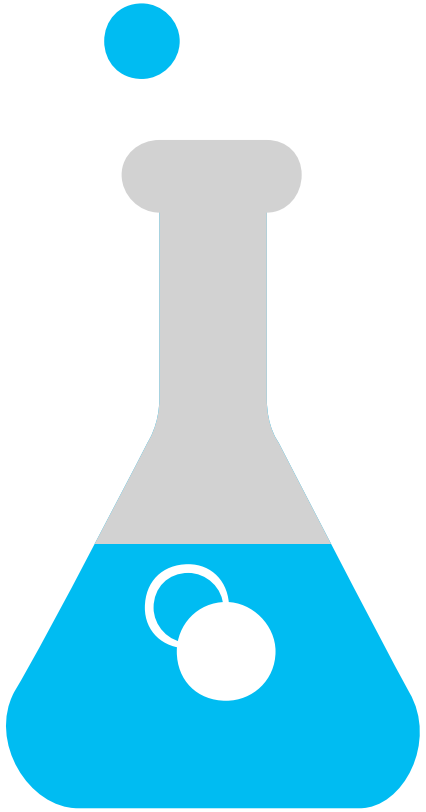
```
PS C:\> (Get-Date).GetType()

IsPublic IsSerial Name
-------- -------- ----
True     True     DateTime
```

- GetType() METHOD retrieves the TYPE name
- Available on ALL objects

Demonstration

Gettype() And Get-Member

Object Models



LAB

Variables and Data Types

Learnings covered in this Unit

- What are Variables?
- User-Defined Variables
- Strings
- Types
- Type Operators
- Parsing Modes
- Escape Character
- Stop Parsing

What are Variables?

What Are Variables?

- Unit of memory
- Defined and accessed using a dollar sign prefix ($)
- Holds an object which can also be a collection of objects
- Variable names can include spaces and special characters
- Not case-sensitive

- Kinds of variables:
  - Automatic (built-in)
  - User-defined

Automatic Variables

- Built-in
- Created and maintained by PowerShell
- Store PowerShell state

Automatic Variables – Examples

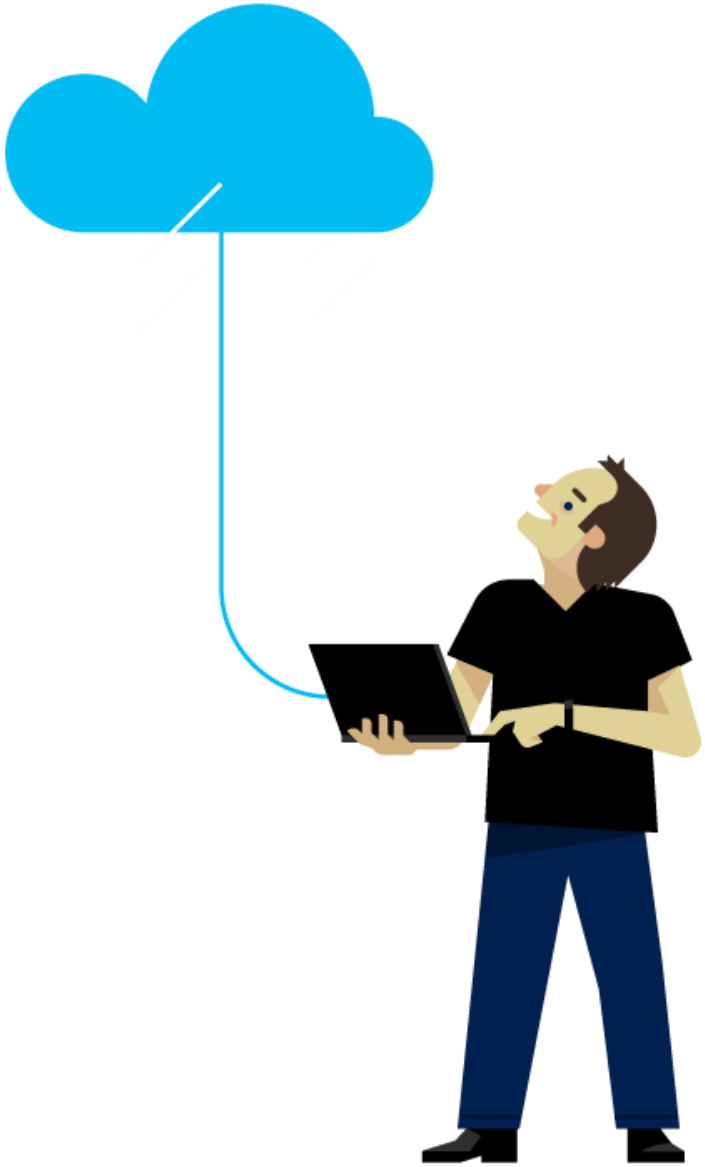
Get-Help about_Automatic_Variables

`PS C:\> $Error`	List of all errors
`PS C:\> $?`	Execution status of last operation
`PS C:\> $HOME`	User's home directory
`PS C:\> $Host`	Current host application for PowerShell
`PS C:\> $null`	NULL or empty value
`PS C:\> $PSHOME`	Full path of installation directory for PowerShell
`PS C:\> $true`	Represents TRUE in commands
`PS C:\> $false`	Represent FALSE in commands

Demonstration

Automatic Variables

Questions?



User-Defined Variables

User-Defined Variables

- Created and maintained by user
- Exist only in current session
- Lost when session is closed

Variable Cmdlets

Name	Example
New-Variable	`PS C:\> New-Variable zipcode -Value 98033`
Clear-Variable	`PS C:\> Clear-Variable -Name Processes`
Remove-Variable	`PS C:\> Remove-Variable -Name Smp`
Set-Variable	`PS C:\> Set-Variable -Name desc -Value "Description"`
Get-Variable	`PS C:\> Get-Variable -Name m*`

Constant Variables

- Variables can only be made constant at creation (cannot use "=")
- Cannot be deleted
- Cannot be changed

```
PS C:\> New-Variable -Name pi -Value 3.14159 -Option Constant
```

ReadOnly Variables

- Cannot mark a variable ReadOnly with "="
- Cannot be easily deleted (must use Remove-Variable with -Force)
- Cannot be changed with "=" (must use Set-Variable with -Force)

```
PS C:\> New-Variable -Name max -Value 256 -Option ReadOnly
```

User-Defined Variable

```
PS C:\> $svcs = Get-Service
  #or
PS C:\> Get-Service -OutVariable svcs
  #or
PS C:\> New-Variable -Name svcs -Value (Get-Service)

PS C:\> $svcs

Status   Name               DisplayName
------   ----               -----------
Stopped  AeLookupSvc        Application Experience
Stopped  ALG                Application Layer Gateway Service
Running  AppIDSvc           Application Identity
Running  Appinfo            Application Information
...
```

Variables and Data Types

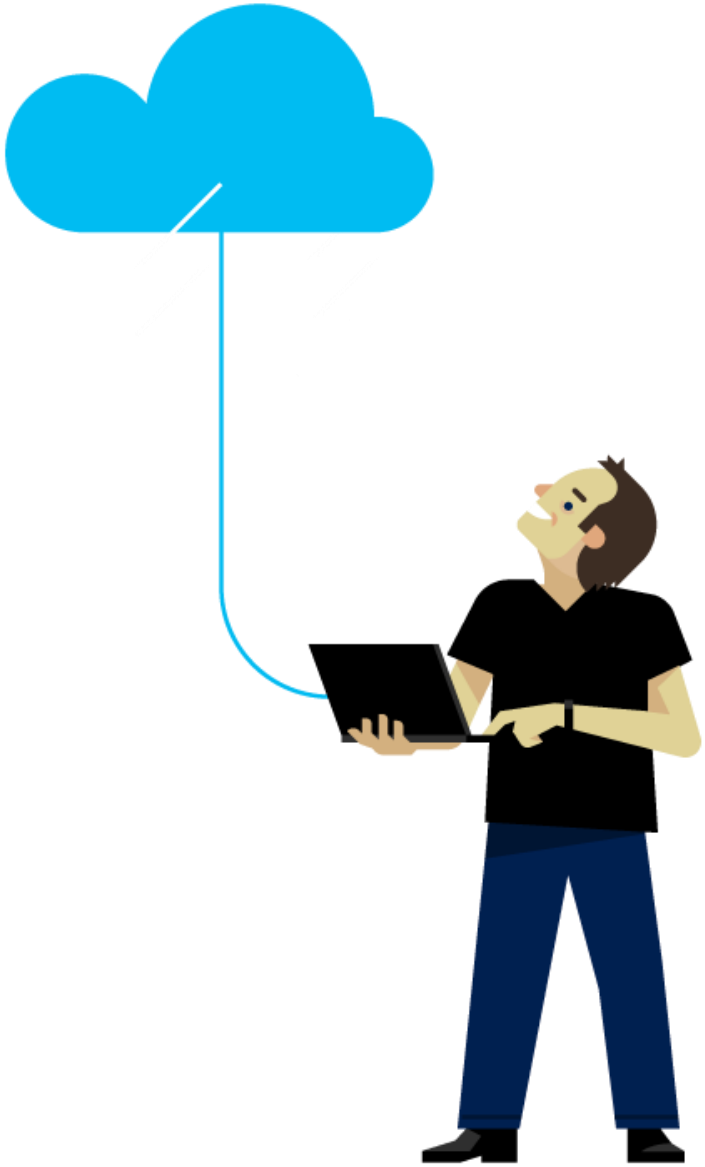
Remember, in PowerShell:

- Everything is an OBJECT
- Each OBJECT has a TYPE
- Variables reference OBJECTS

Demonstration

User Defined Variables

Questions?



Strings

Literal Strings

Create a variable

```
PS C:\> $a = 123
```

Include the variable in

```
PS C:\> $b = 'As easy as $a'
```

Notice that $a is not expanded

```
PS C:\> $b
As easy as $a
```

Expandable Strings

Create a variable

```
PS C:\> $a = 123
```

Include the variable in an expandable string (double-quotes)

```
PS C:\> $b = "As easy as $a"
```

Notice that $a is expanded

```
PS C:\> $b
As easy as 123
```

Literal or Expandable String Spanning Multiple Lines

Literal String

```
PS C:\> $lString = '
As
easy
as
$a
'

PS C:\> $lString
As
easy
as
$a
```

Expandable String

```
PS C:\> $eString = "
As
easy
as
$a
"

PS C:\> $eString
As
easy
as
123
```

Here Strings

- Simplify use of longer, more complex string assignments
- Here String can contain quotes, @ sign, etc.

Literal Here String

```
PS C:\> $lhere = @'
As
'easy'
as
$a
'@

PS C:\> $lhere
As
'easy'
as
$a
```

Expandable Here String

```
PS C:\> $ehere = @"
As
"easy"
as
$a
"@

PS C:\> $ehere
As
"easy"
as
123
```

Sub-Expression

- Within an expandable string, it might be necessary to display the results of an operation or a property of an object.
- Utilizing the Dollar Sign ($) followed by enclosing parenthesis, we can surround certain statements to so they are processed. This is called Expression Mode

```
# Properties Not expanded
PS C:\> $a = Get-Service -Name BITS
PS C:\> $b = "$a.Name is $a.Status"
System.ServiceProcess.ServiceController.name is
System.ServiceProcess.ServiceController.status
# RIGHT WAY using SubExpression
PS C:\> $a = Get-Service -Name BITS
PS C:\> $b = "$($a.Name) is $($a.Status)"
BITS is Running
# This can also be used on any operation that you want to run in a string
PS C:\> $a = "Your Lucky Number is $(Get-Random)" # Get-Random gives you a
random number
PS C:\> $a
Your Lucky Number is 1023023027
```

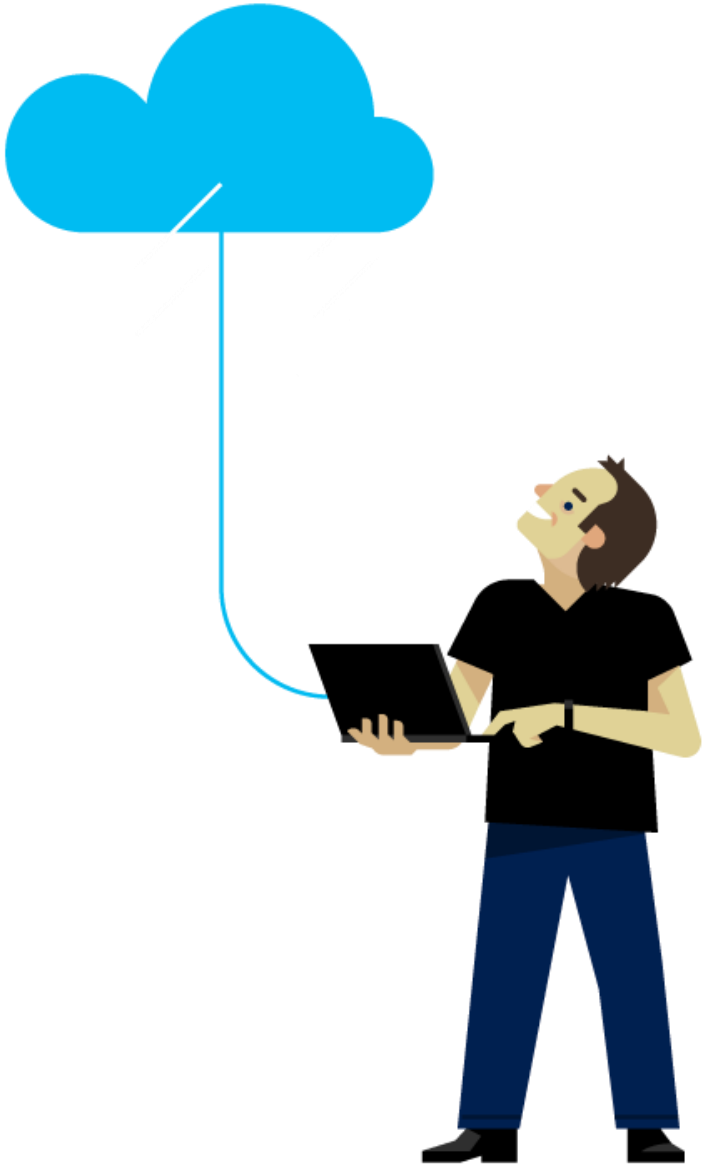
Note the colorization. PowerShell is not processing the properties as part of the Expansion.

When a variable is expanded, the ToString method is called. Most objects default for ToString is to display their Type Name.

Demonstration

Strings, Here Strings and Subexpression

Questions?



Types

Types

- Every object exists of a type
- Object types are declared when created
- Object types can be converted into other types using type casting
- PowerShell is a non declared language meaning, PowerShell will search for a best match type for you when not casted

General Types

Alias	Full Name	Description
Object	System.Object	Every type in PowerShell is derived from object
Boolean	System.Boolean	$true and $false
Char	System.Char	Stores UTF-16-encoded 16-bit Unicode code point
Int	System.Int32	-2147483648 to 2147483647
Long	System.Int64	-9223372036854775808 to 9223372036854775807
Double	System.Double	Double-precision floating-point number
Enum	System.Enum	Defines a set of named constants
Array	System.Array	One or more dimensions with 0 or more elements
DateTime	System.DateTime	Stores date and time values

What Object Type am I Using?

PowerShell typically picks object type

Examples of PowerShell choosing appropriate Type

```
PS C:\> (1024).GetType().FullName
System.Int32

PS C:\> (1.6).GetType().FullName
System.Double

PS C:\> (1tb).GetType().FullName
System.Int64
```

Type Casting

- You can control object types
- Using a type or class in [Square Brackets] in front of an object will force that type
- Some common types have simpler type alias'

Examples of Type Casting

```
PS C:\> [system.int32]1.6
2

PS C:\> $MyNumber = [int]"000123"
PS C:\> $MyNumber
123

PS C:\> $MyNumber.GetType().FullName
System.Int32
```

Variables Can Be Strongly Typed

Variables are weakly typed by default

Type cast the variable name during creation to strongly type

Variable will only hold that type of object

Weakly Typed Variable

```
PS C:\> $var1 = [int]1.3
PS C:\> $var1
1

PS C:\> $var1 = 1.2
PS C:\> $var1
1.2
```

Strongly Typed Variable

```
PS C:\> [int]$var1 = 1.3
PS C:\> $var1
1

PS C:\> $var1 = 1.2
PS C:\> $var1
1
```

Strong Typing a variable

```
PS C:\> [int]$var1 = 123.5
PS C:\> $var1
124

PS C:\> [string]$var2 = 987.6
PS C:\> $var2
987.6

PS C:\> $var1.GetType().FullName ; $var2.GetType().FullName
System.Int32
System.String

PS C:\> $var1 =  "Fred"
 Cannot convert value "Fred" to type "System.Int32".
Error: "Input string was not in a correct format."
```

Demonstration

Type Casting

Static Members

- Static Member is callable without having to create an instance of a type
- Static Member is accessed by type name (not instance name)
- Static Members are accessed using the Static Operator 

Discover Static Members

A type can be used in PowerShell using square brackets

```
PS C:\> [char]

IsPublic IsSerial Name   BaseType
-------- -------- ----   --------
True     True     Char   System.ValueType
```

Use Get-Member -Static to discover 'useful' members

```
PS C:\> [char] | Get-Member -Static
```

Get-Member without -Static discovers members that provide information about the type

```
PS C:\> [char] | Get-Member
```

Calling a STATIC member

```
PS C:\> [char]::IsWhiteSpace(" ")
True
```

Print ASCII table using PowerShell

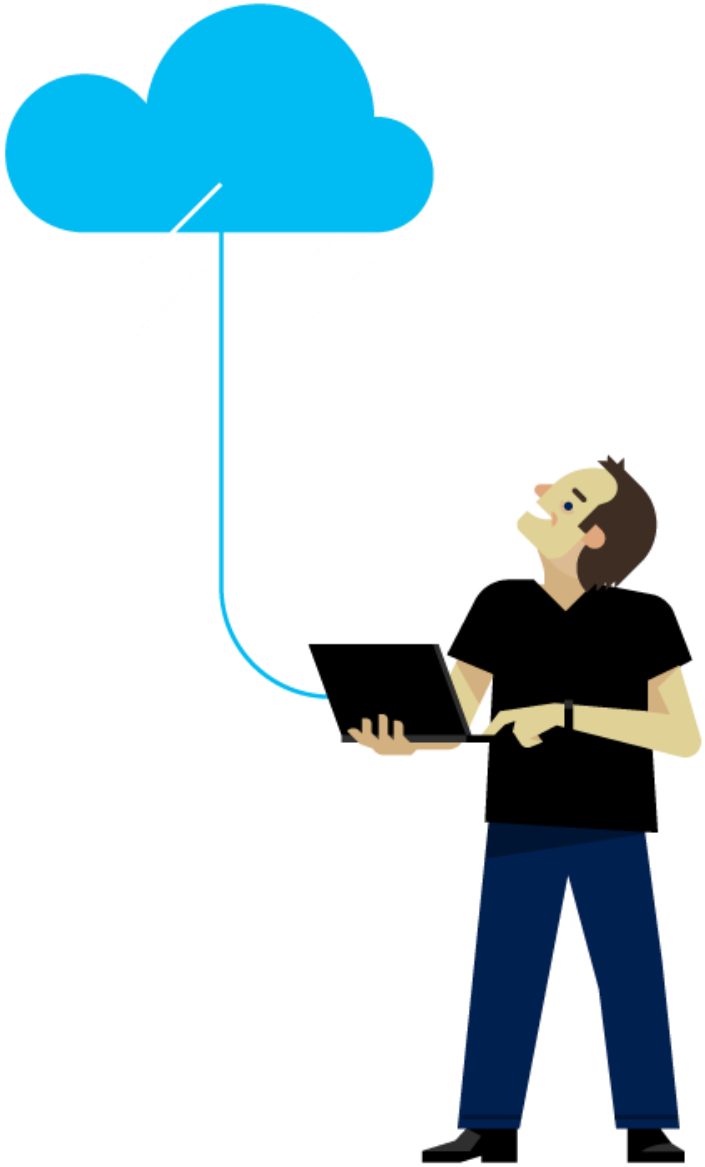
Use range operator and Char type to convert numbers to ASCII Characters

```
PS C:\> 33..255 | ForEach-Object {
Write-Host "Decimal: $_ = Character: $([Char]$_)"
}

Decimal: 33 = Character: !
Decimal: 34 = Character: "
Decimal: 35 = Character: #
Decimal: 36 = Character: $
Decimal: 37 = Character: %
Decimal: 38 = Character: &
Decimal: 39 = Character: '
...
```

Demonstration

Static Methods

Questions?



Type Operators

Type Operators – Test Object Types

Operator	Example

-is	`PS C:\> (get-date) -is [DateTime]`
	
	`True`

-isNot	`PS C:\> (get-date) -isNot [DateTime]`
	
	`False`

Type Operators – Type Cast via Operator

Operator	Example

-as	`PS C:\> "27/12/2017" -as [datetime]`
	
	`Wednesday, 27 December 2017 12:00:00 AM`

Same Result

```
PS C:\> [datetime]"27/12/2017"

Wednesday, 27 December 2017 12:00:00 AM
```

Demonstration

Type Operator

Parsing Modes

Parsing Modes

- PowerShell parser divides commands into “tokens“
- Parser is in either EXPRESSION or ARGUMENT mode depending on the “token”
- In **expression** mode, the parsing is conventional: strings must be quoted, numbers are always numbers, and so on.
- In **argument** mode, numbers are treated as numbers but all other arguments are treated as strings unless they start with $, @, ', ", or (.

Parsing Mode Examples

Example	Mode	Result
`2+2`	Expression	4
`Write-Output -InputObject 2+2`	Argument	"2+2"
`Write-Output -InputObject (2+2)`	Expression	4
`$a = 2+2 ; $a`	Expression	4
`Write-Output $a`	Expression	4
`Write-Output $a/H`	Argument	"4/H"
`Get-ChildItem "C:\Program Files"`	Expression	

Turn Argument Mode Into Expression Mode With $()

Get-Date is a token interpreted as an ARGUMENT by the parser
Get-Date is not treated as an expression

```
PS C:\> Write-Host "The date is: Get-Date"
The date is: Get-Date
```

$ changes the parsing mode to EXPRESSION

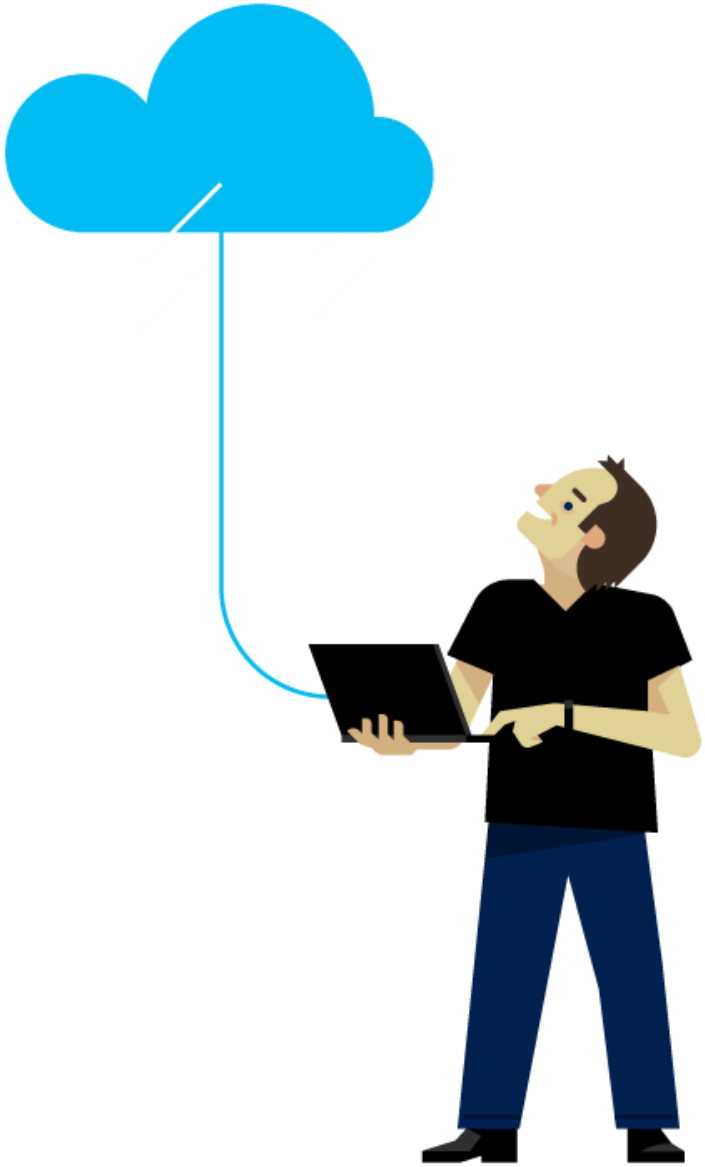
```
PS C:\> Write-Host "The date is: $(Get-Date)"
The date is: 1/3/2018 11:03:53
```

Turn Expression Mode into Argument Mode With &

Store a string representing a Cmdlet name in a variable
Call the command by using the ampersand (&) character

```
PS C:\> $cmd =  "Get-Process"
PS C:\> $cmd
Get-Process

PS C:\> & $cmd

 Handles  NPM(K)    PM(K)      WS(K) Id      Name
-------  ------    -----      ----- -----   ------
     333      18     8032       8.80   8244  AcroRd32
     413      39   155844     131.89  10140  AcroRd32
      84       8     1128       0.13   2228  armsvc
…
```

Questions?



Escape Character

Escape Character

- Assigns a special interpretation to characters that follow
- Backtick (grave accent)
- ASCII 96 

Backtick Uses

Force a special character to be literal	`PS C:\> $a = 123` `PS C:\> Write-Host "`$a is $a"` `$a is 123`

Force a literal character to be special
Limited List (next slide)
	`PS C:\> Write-Host "There are two line breaks`n`nhere. "`
	`There are two line breaks`
	
	`here.`

Line continuation *Must be last char*	`PS C:\scripts> Get-Service	`` `Where-Object name -EQ `` `alg`

Special Characters

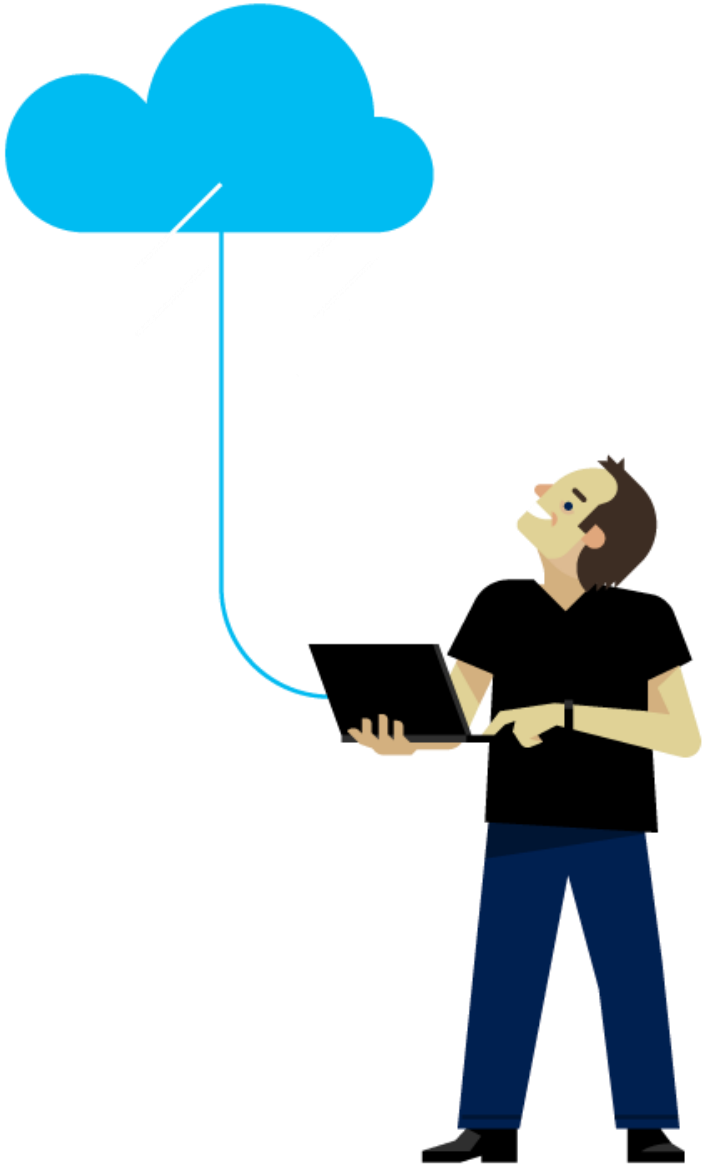
Character	Description
`0	Null
`a	Alert
`b	Backspace
`f	Form feed
`n	New line
`r	Carriage return
`t	Horizontal tab
`v	Vertical tab

Note:
These characters are case-sensitive and only have effect within double quotes

Demonstration

Special Escape Characters

Questions?



Stop Parsing

Stop Parsing

- Stops PowerShell from interpreting input
- Use when entering external command arguments (rather than escape characters)
- Only takes effect until next newline or pipe character

```
--%
```

Stop Parsing External Command Arguments

Parsing and special characters can make external commands challenging

```
PS C:\> icacls X:\VMS /grant Dom\HVAdmin:(CI)F

CI : The term 'CI' is not recognized as the name
of a cmdlet, function, script file, or operable
program.
```

Parenthesis cause issue

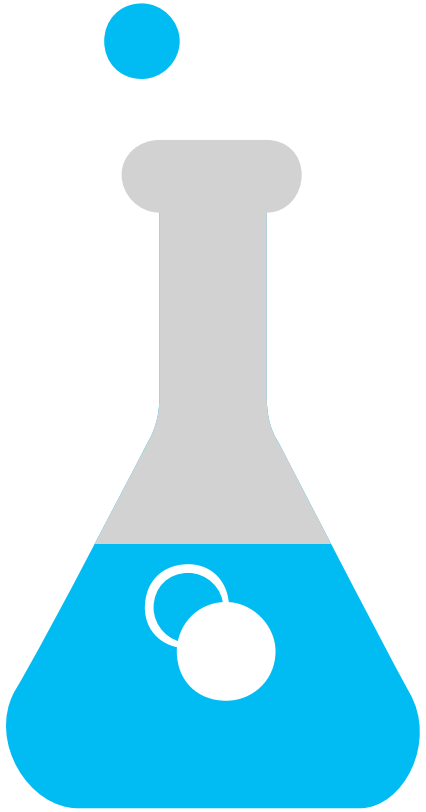
Use --% to stop PowerShell parsing

```
PS C:\> icacls X:\VMS --% /grant Dom\HVAdmin:(CI)F
```

Sends the following command to the icacls program

```
X:\VMS /grant Dom\HVAdmin:(CI)F
```

Variables and Datatypes



LAB