June Blender
Technology Evangelist
SAPIEN Technologies, Inc.
juneb@sapien.com
@juneb_get_help

A Class of  Wine

CREATING CLASSES IN THE WINDOWS POWERSHELL 5.0 PREVIEW

What is a *class*?

In Windows PowerShell, a class defines a **type** of object.

The class is **not** an object.

It's like a **specification** of an object.

You can create objects based on the specification.

To discover classes, use Get-Member


```
PS C:\ps-test> Get-Process | Get-Member


   TypeName: System.Diagnostics.Process

Name                      MemberType     Definition
----                      ----------     ----------
Handles                   AliasProperty  Handles = Handlecount
Name                      AliasProperty  Name = ProcessName
NPM                       AliasProperty  NPM = NonpagedSystemMemorySize
PM                        AliasProperty  PM = PagedMemorySize64
VM                        AliasProperty  VM = VirtualMemorySize64
WS                        AliasProperty  WS = WorkingSet64
Disposed                  Event          System.EventHandler Disposed(S
ErrorDataReceived         Event          System.Diagnostics.DataReceive
Exited                    Event          System.EventHandler Exited(Sys
OutputDataReceived        Event          System.Diagnostics.DataReceive
BeginErrorReadLine        Method
BeginOutputReadLine       Method
```

Wine

```
PS C:\ps-test> $PSWine | Get-Member


   TypeName: Wine

Name        MemberType Definition
----        ---------- ----------
Equals      Method     bool Equals(System.Object obj)
GetHashCode Method     int GetHashCode()
GetType     Method     type GetType()
toString    Method     string toString()
Color       Property   string Color {get;set;}
Description Property   string Description {get;set;}
isSparkling Property   bool isSparkling {get;set;}
Name        Property   string Name {get;set;}
Rating      Property   uint32[] Rating {get;set;}
Sweetness   Property   WineSweetness Sweetness {get;set;}
Winery      Property   string Winery {get;set;}
Year        Property   uint32 Year {get;set;}
```

```
PS C:\ps-test> $myGlass | Get-Member


   TypeName: WineGlass

Name        MemberType Definition
----        ---------- ----------
Equals      Method     bool Equals(System.Object obj)
GetGlass    Method     psobject GetGlass()
GetHashCode Method     int GetHashCode()
GetType     Method     type GetType()
Refill      Method     void Refill(uint32 Ounces)
Sip         Method     void Sip(uint32 Ounces)
ToString    Method     string ToString()
Amount      Property   uint32 Amount {get;set;}
Pour        Property   uint32 Pour {get;set;}
Wine        Property   Wine Wine {get;set;}
```

WineGlass

What is an *instance*?

It's an object that's based on a class.

Like a house that's built from a model (but more reliable!)

It has the properties and methods that the class defines, like Roof and Door.

But it's property values might differ from other instances, like:

Roof = Composite
Door = 1

To discover instances, use Format-List -Property * (fl *)


```
PS C:\ps-test> Get-Process PowerShell | fl *


__NounName                  : Process
Name                        : powershell
Handles                     : 242
VM                          : 2199551356928
WS                          : 11685888
PM                          : 8859648
NPM                         : 15144
Path                        : C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
Company                     : Microsoft Corporation
CPU                         : 0.6875
FileVersion                 : 10.0.10018.0 (fbl_srv2_ci_mgmt_rel.150216-2200)
ProductVersion              : 10.0.10018.0
Description                 : Windows PowerShell
Product                     : Microsoft® Windows® Operating System
Id                          : 1944
PriorityClass               : Normal
HandleCount                 : 242
WorkingSet                  : 11685888
PagedMemorySize             : 8859648
PrivateMemorySize           : 8859648
VirtualMemorySize           : 528101376
TotalProcessorTime          : 00:00:00.7187500
```

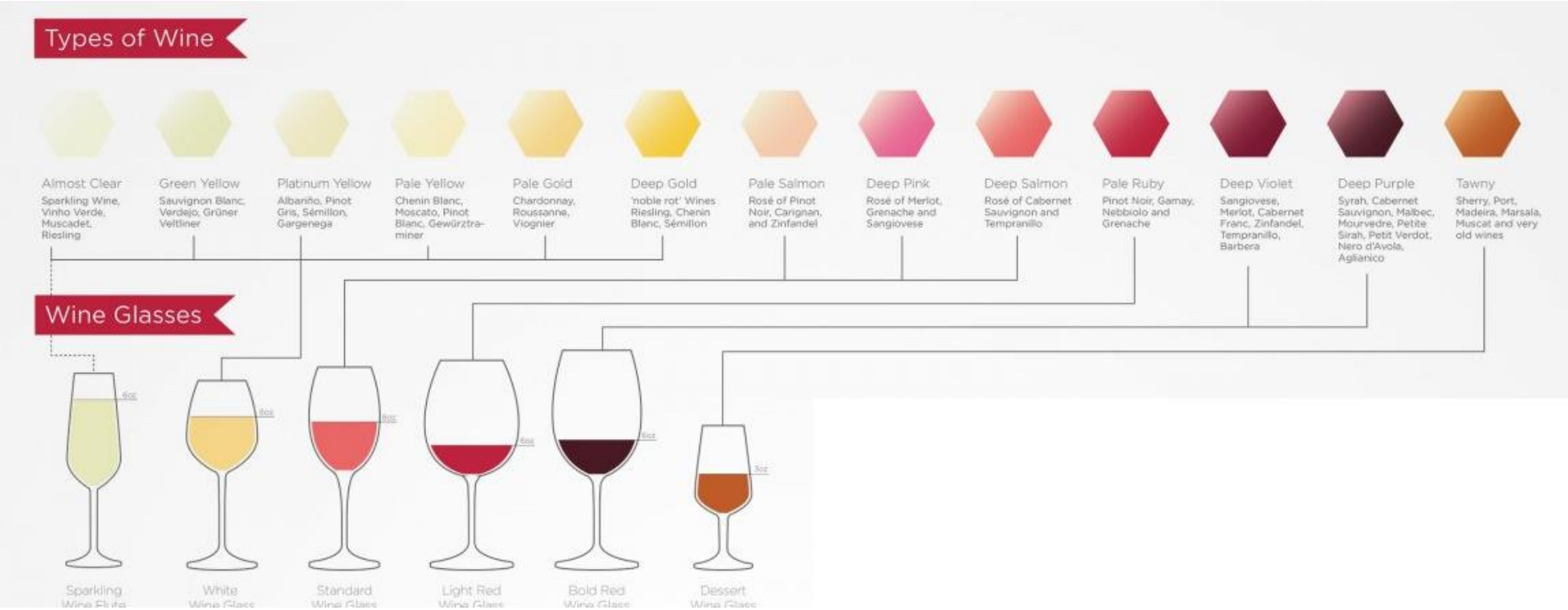
Wine

```
PS C:\ps-test> $PSWine | fl *


Name        : PSWine
Winery      : Chateau Snover
Year        : 2012
isSparkling : False
Color       : Red
Sweetness   : Dry
Description : Pithy
Rating      : {96, 94}
```

WineGlass

```
PS C:\ps-test> $myGlass | fl *


Wine   : Wine
Pour   : 8
Amount : 6
```

A Wine class and a WineGlass class



Why?

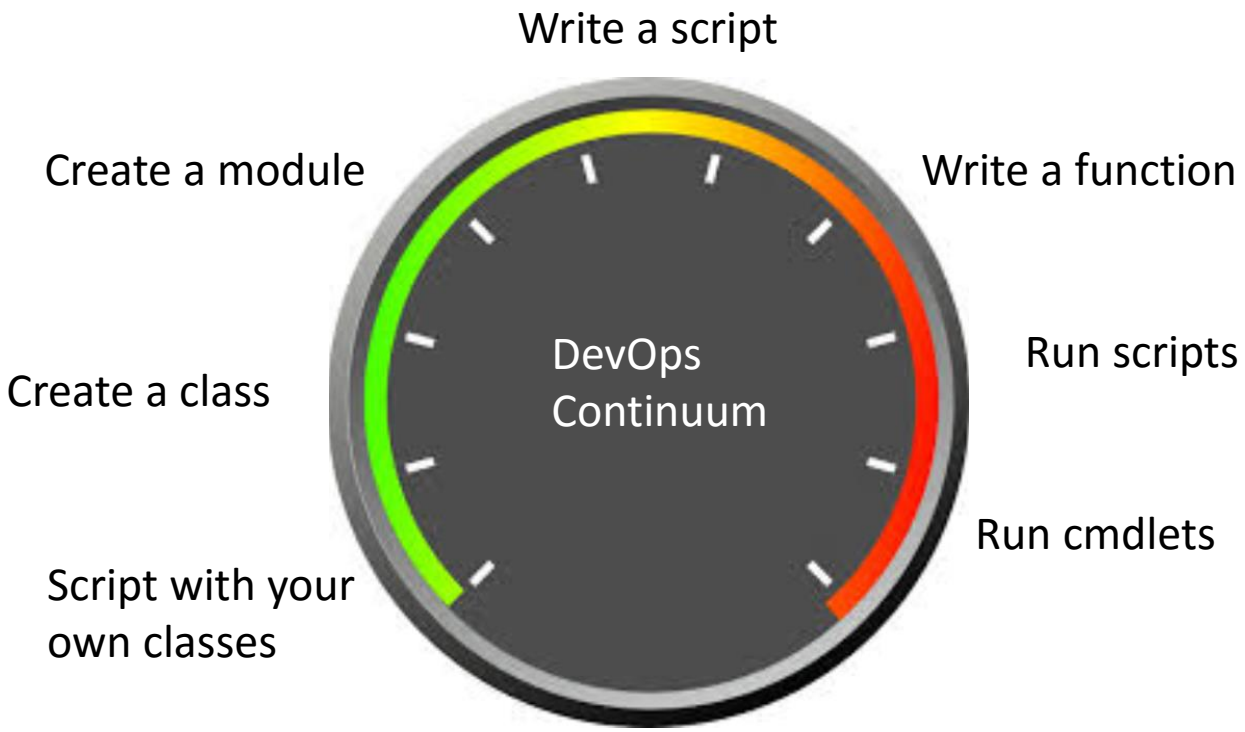
You create custom objects … from hash tables, CSV files, Select-Object

```
PS C:\ps-test> $Keyword = [PSCustomObject]@{Name= $Table.Name; Description = $Template.Reference}
```

What if you could create classes -- reusable object models -- with methods?

```
PS C:\> $if = $keywords | where Name -eq "If"
PS C:\> $if.GetHelp()
TOPIC
    about_If

SHORT DESCRIPTION
    Describes a language command you can use to run statement lists based
    on the results of one or more conditional tests.


LONG DESCRIPTION
    You can use the If statement to run code blocks if a specified
    conditional test evaluates to true. You can also specify one or more
...
```

Classes: The Final Frontier



Demo: Wine Class

Define the Wine class

Create instances of the Wine class

Use their properties and methods

You can't create types at the command-line, because "you can't use a type literal ([MyClass]) outside the script/module file in which the class is defined."

New-Object doesn't work, because "you can't refer to the type name as a string."

Create a class : **Class** keyword

Syntax:

```
class <className> { }
```

Example:

```
class Wine { }
```

Add properties

Syntax:

```
class <className>
{
        [<Attribute>] [<Type>] <PropertyName> [ = <defaultValue>]
}
```

Example:

```
class Wine
{
        [String] $Name
}
```

Add properties

Syntax:

```
class <className>
{
        [<Attribute>] [[<Type>]] <PropertyName>
}


class Wine
{
        $Name
        [ValidateSet("Red", "White", "Rose")]$Color
        [int32]$Year = (Get-Date).Year
}
```

Wine class properties

```
class Wine
{
  [String] $Name
  [String] $Winery
  [Int32] $Year
  [Boolean] $isSparkling = $False
  [ValidateSet("Red", "White", "Rose")][String] $Color
  [WineSweetness] $Sweetness
  [String] $Description
  [Int] $Price
}
```

Add methods : syntax

```
[returnType] <methodName> ([<parameters>])
{
      <process parameter values>
      [return] <returnType expression>
}
```

Add methods : syntax

```
[returnType] <methodName> ([<parameters>])
{
        <process parameter values>
        [return] <returnType expression>
}
```

```
-- Parentheses around parameters are required  ()
-- All parameters are mandatory and positional
-- Multiple methods with same name and different parameter types ("overload")
-- Default return type is void (nothing)
-- All paths of method must return the return type
-- Return keyword is required to return anything (no standard output)
-- Parameter "(" must be on first line
```

Return Types: Contract

Optional. The default is "[void]" : Cannot return anything.

Every logical path must return the same type.

Return keyword is required.

```
[String] GetHelp()
{
    [String]$firstRef = $this.Reference.Split(",").Trim() | Select-Object -First 1
    try
    {
        return Get-Help $firstRef
    }
    catch
    {
        return ""
    }
}
```

$this

Refers to the current instance of the class ($_ for classes)
Distinguishes properties from parameters and variables.

```
class Tree
{
  #Properties
  [String] $Species
  [Int32]  $Height

  Tree ($Species, $Height)
  {
      $this.Species = $Species
      $this.Height = $Height
  }
}
```

Wine class methods

```
class Wine
{
    ...
     # toString()
     # Override: Returns a human-readable string.

    [String] toString()
    {
       $color = $this.Color
       if ($this.isSparkling) {$color = "Sparkling $color"}

       return "$color Wine: $($this.Name) `
               $($this.Year) by $($this.Winery). `
               Priced at: $("{0:C}" -f $this.Price). `
               Described as: $($this.Description). "
     }
}
```

Add methods : Example

```
#Properties
[int] $Amount

#Methods
[String] Sip ([Int]$Ounces, [String]$Response)
{
   if ($this.Amount -ge $Ounces)
  {
        $this.Amount -= $Ounces
        return "Oh! $Response"
   }

}
```

Calling class methods

No parameter names; All parameters are mandatory and positional; Parentheses are required

```
[String] Sip ([Int]$Ounces, [String]$Response)
{
    if ($this.Amount ge- $Ounces)
        {
          $this.Amount -= $Ounces
           return "Oh! $Response"
    }
}
```

```
$myWine.Sip(2, "What a delightful finish!")
```

~~$myWine.Sip("what a delightful finish", 2)~~

~~$myWine.Sip(2)~~

~~$myWine.Sip(-Amount 2 -Response "what a delightful finish")~~

Create a wine
(instance of the Wine class)

(Feb. v5.0.10018.0)

- To create an instance of a class, use the **New** static method.
- The parameters of the New method match a constructor.
- New-Object doesn't work, because "you can't refer to the type name as a string."
- You can't create types at the command-line, because "you can't use a type literal ([MyClass]) outside the script/module file in which the class is defined."

Constructors : about constructors

Special methods for creating an instance of the object.

Constructor name is always the class name.

Multiple constructors for a class.

Each constructor must take a different number or type of parameter values.

Windows PowerShell adds a null constructor (no parameters) to every class.

Constructors

```
# Constructors
Wine () {}   # Null constructor; Automatic in Windows PowerShell

Wine ([string]$Name) {$this.Name =  $Name}

Wine ([string]$Name, [int][ValidateRange(10, 10000)]$Price)
{
    $this.Name =  $Name
    $this.Price = $Price
}
```

```
$myWine = [Wine]::new()  #Using null constructor
$myWine = [Wine]::new("Great Duck")
$myWine = [Wine]::new("PSWine", 40)  #No parameter names!
```

Calling a constructor: create new instance

```
[<className>]::New(<constructor>)
```

```
Constructor:
      Wine () {}
Call:
      $myWine = [Wine]::New()

Constructor:
     Wine ([string]$Name, [int]$Price)
      {
         $this.Name =  $Name
         $this.Price = [ValidateRange(10, 10000)$Price
      }
Call:
      $myWine = [Wine]::new("PSWine", 40)  #No parameter names!
      $myWine = [Wine]@{Name="PSWine"; Price=40}  #Requires null/"parameter-less" constructor
```

Create a wine
(hash table)

(Feb. v5.0.10018.0)

```
$myWine = [Wine]::New()      # Must be explicit

$myWine = [Wine]@{Name="Great Duck";
          Winery="Escalante Winery";
          Year="2003";
          isSparkling = $True;
          Color = "White";
          Sweetness = 4;
          Description= "Rich and magnificent";
          Price = 22}
```

Interacting Classes

Create a WineGlass class

```
class WineGlass
{
    [Wine]$Wine
    [Int] $Size
    [Int] Amount

    Sip ($Amt) { ... }
    Sip() {...}
    Pour ($Amt) { ... }
    Refill()
    ...
}
```

Enumerated types ("enums")

```
enum WineSweetness
{
    VeryDry
    Dry
    Moderate
    Sweet
    VerySweet
}
```

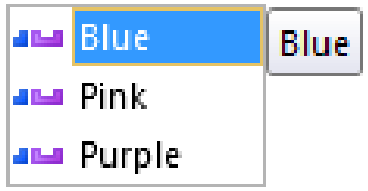
```
class Wine
{
    #Properties
    ...
    [WineSweetness] $Sweetness
    ...
}
```

```
enum WineSweetness {VeryDry; Dry; Moderate; Sweet; VerySweet}
```

Enum value rules

- Default values are zero-based integers.
- You can change to a different int, but you c change the base type.
- Values can be expressions that return an in
- Can't be the result of an invoked command ("Must be a parse-time constant.")
- You can't define it or refer to it at the command-line. ("You can't use a type literal outside the script/module file in which the class is defir

```
1   enum basicColor
2   {
3       Red=1
4       Blue = 2
5       Green = 3
6   }
7
8   enum FavoriteColor
9   {
10      Blue = [basicColor]::Blue
11      Pink = 7PB + 3TB
12      Purple = (Get-Date).Year
13  }
14
15  function Get-Color ([FavoriteColor]$myFave)
16  {
17      "My favorite color is $myFave."
18  }
19
20  Get-Color -myFave
```



Using enums

```
enum WineSweetness {VeryDry; Dry; Moderate; Sweet; VerySweet}
```

```
#Properties
[WineSweetness] $Sweetness
```

```
$myWine.Sweetness = 4
$hisWine.Sweetness = VerySweet
```

Enums: set alternate int values

```
enum WineSweetness
{
    VeryDry = 10
    Dry = 20
    Moderate = 50
    Sweet = 90
    VerySweet = 100
}
```

```
class Wine
{
   #Properties
   ...
   [WineSweetness] $Sweetness
   ...
}
```

Get/Set Property Values
(in other languages)

```
class Wine
{
   [String] $Name
   ...
}

$myWine = [Wine]::new()

$myWine.Name = "Great Duck"
Error: Name is a read-only property.   #Not PowerShell

$myTree.Name
Error: Name is a private value. #Not PowerShell
```

Automatic Getters/Setters

```
class Wine
{
   [String] $Name
}
```

```
$myTree = [Wine]::new()

$myTree.set_Name = "Sycamore"  # Set the property value

$myTree.get_Name               # Get the property value
Sycamore
```

Automatic Getters/Setters

When you create a property in a scripted class, Windows PowerShell adds get and set methods for you.

All properties are public (you can get the value) and read-write (you can change the value).

All variables in a class are properties of the class.

```
class Wine { $Name }
$myWine = [Wine]::new()

$myWine | Get-Member -Force
...
get_Name Method       string get_Name()
set_Name Method       void set_Name(string )
Name     Property     string Name{get;set;}
```

Hidden: sort of…

Hidden keyword

Hidden hides them from Get-Member and other cmdlets (Format-List *), but you can see them in Get-Member -Force.

```
class WineGlass : Glass
{
 hidden [int] Consumed
 hidden [int] TotalPoured
}
```

Inheritance

Hierarchy of classes

Child classes inherit properties and methods
(and events and other members) from parent classes

If they don't like them, they can override them.

Terminology:

Parent class: Superclass, base class
Child class: Subclass

Inherit from System.Object

All .NET classes are derived from System.Object ("superclass")

Superclass
|
Subclass

System.Object
|
Wine, Tree

System.Object
|
Glass
|
WineGlass

Inherit from other classes

Base a scripted class on a .NET class (inheritable, not sealed)

Base a scripted class on another scripted class

```
System.Object
      |
Wine, Tree

System.Object
      |
   Glass
      |
 WineGlass

System.Object
      |
KeyedCollection (.NET class)
      |
 FileCollection
```

Create a subclass : syntax

```
class <subClassName> : <SuperClass>
class WineGlass : Glass {}
```

Scripted classes inherit from System.Object

```
PS C:\ps-test> $myObj = New-Object -TypeName System.Object
PS C:\ps-test> $myObj | Get-Member

   TypeName: System.Object

Name        MemberType Definition
----        ---------- ----------
Equals      Method     bool Equals(System.Object obj)
GetHashCode Method     int GetHashCode()
GetType     Method     type GetType()
ToString    Method     string ToString()
```

```
class Wine {  }
$myWine = [Wine]::new()
```

```
PS C:\ps-test> $myWine | Get-Member

   TypeName: Wine

Name        MemberType Definition
----        ---------- ----------
Equals      Method     bool Equals(System.Object obj)
GetHashCode Method     int GetHashCode()
GetType     Method     type GetType()
ToString    Method     string ToString()
```

System.Object is a "superclass" (parent)
Scripted class is a "subclass" (child)

Subclasses

Superclass

Overload: Same name, different signature

In same class or parent class

```
Drink ([int]Amount)
{ ...          }


Drink ()
{ ...          }
```

Override: Same name, same signature

In superclass class and subclass : Local takes precedence

```
class Glass {
   Drink ([int]Amount)
    { ...        }
}
class WineGlass : Glass {
    Drink ([int]Amount)
    { ...        }
}
```

Interfaces

An *interface* is a like a contract.

It guarantees that classes have the members (properties/methods/events) that it specifies.

But, the interface does **not** specify how the members work.

Classes that "sign the contract" *implement* the interface.

They have the members that the interface specifies ("signature").

Each class can **implement** multiple interfaces.

Classes can inherit multiple interfaces and override them.

Interfaces in .NET

IEnumerable: enumerating, foreach

IComparable: sorting

IClonable: duplicate instances of the class

IFormattable: change toString() behavior

IDisposable: garbage collection, releasing unused resources

To find the interfaces a class implements:

```
[<className>].ImplementedInterfaces   # PowerShell 4.0
[<className>].GetInterfaces()         # PowerShell 5.0
```

Interfaces : syntax

Scripted classes can implement interfaces

You cannot (yet?) create/define interface

```
class <className> : <interfaceName>
class <className> : <superclass>, <interfaceName> [,<interfaceName>, ...]
```

Get-Help

Script help, About topics

Best practices:

-- Examples:

-- Show how to create an instance (constructors).

-- Show how to use the properties and methods.

-- Explain the purpose of the class and its instances.

-- Describe the properties and their values.

-- Explain the methods, their parameters/values, and return values.

-- Note inheritance, interfaces, overrides.

Classes in 5.0.10018.0

You can create scripted classes in Windows PowerShell 5.0 preview.

```
class <className> [: <Superclass>, <Interface]
```

Classes have constructors to make instances of the class

Classes have properties and methods.

Windows PowerShell adds getter/setter methods for properties.

Classes can inherit from other classes (and System.Object)

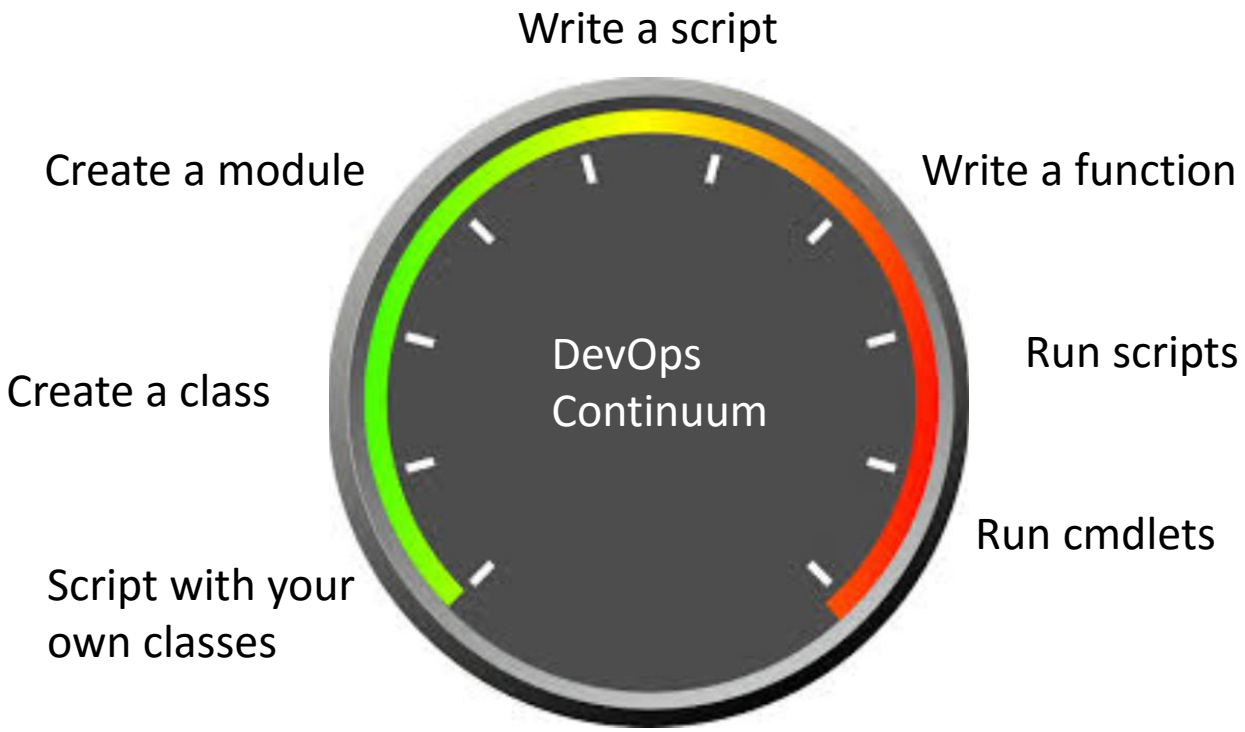
You can override inherited methods.

Classes can implement .NET interfaces.

You can include classes in scripts and modules.

You can write help for classes.

Classes: The Final Frontier



Understanding MSDN

Process Class

.NET Framework 4.5 | Other Versions

Provides access to local and remote processes and enables you to start and stop local system processes.
To browse the .NET Framework source code for this type, see the Reference Source.

Inheritance Hierarchy

System.Object
System.MarshalByRefObject
System.ComponentModel.Component
System.Diagnostics.Process

Constructors

	Name	Description
	Process	Initializes a new instance of the Process class.

	CreateObjRef	Creates an object that contains all the relevant information required to generate a proxy used to communicate with a remote object. (Inherited from MarshalByRefObject.)
	Dispose	Releases all resources used by the Component. (Inherited from Component.)
	Dispose(Boolean)	Infrastructure. Release all resources used by this process. (Overrides Component.Dispose(Boolean).)
S	EnterDebugMode	Puts a Process component in state to interact with operating system processes that run in a special mode by enabling the native property **SeDebugPrivilege** on the current th
	Equals(Object)	Determines whether the specified object is equal to the current object. (Inherited from Object.)

Resources

about_Classes

Release Notes

Implementing a .NET Class in PowerShell v5 by Trevor Sullivan

Playing with Classes in PowerShell v5 Preview by Lee Holmes

Writing Classes With PowerShell V5-Part 1 & 2 by Thomas Lee

Beyond custom objects: Create a .NET class by June Blender

Enumerated Types in Windows PowerShell 5.0 by June Blender

Thanks!

Trevor Sullivan

Sergei Vorobev

Joel Bennett

Doug Finke

June Blender
Technology Evangelist
SAPIEN Technologies, Inc.
juneb@sapien.com
@juneb_get_help

A Class of  Wine

CREATING CLASSES IN THE WINDOWS POWERSHELL 5.0 PREVIEW