Iron Scripter 2018: Prequel 1

You were presented with the following code as your first challenge on the path to Iron Scripter 2018.

```
$Monitor = Get-WmiObject wmiMonitorID -namespace root\wmi
$Computer = Get-WmiObject -Class Win32_ComputerSystem

$Monitor ¦ %{
    $psObject = New-Object PSObject

      $psObject | Add-Member NoteProperty ComputerName ""
    $psObject | Add-Member NateProperty ComputerType ""
    $psObject | Add-Member NoteProperty ComputerSerial ""

    $psObject | Add-Member NoteProperty MonitorSerial ""
    $psObject | Add-Member NoteProperty MonitorType ""

    $psObject.ComputerName = $env:computernome
    $psObject.ComputerType = $Computer.model
    $psObject.ComputerSerial =  $Computer.Name

    $psObject.MonitorSerial = ($_.SerialNumberID -ne 0 | %{[char]$_}) -join ""
    $psObject.MonitorType = ($_.UserFriendlyName -ne 0 | %{[char]$_}) -join ""

    $psObject

}
```

Your goal for this challenge was to make the code work and to create a re-usable artefact from the code.

The following reminders were also part of the challenge:

- Following your faction's aims is the most important aspect of this challenge
- Use best practice if it doesn't conflict with your faction's aims
- The solution must work for single or multiple monitors on the local machine
- How would this work with remote machines?
- PowerShell v5.1 is the assumed standard for your code. If you can also make the solution work with PowerShell v6 that is a bonus

Let's work through the changes you need to make to the code to satisfy the requirements.

The first task is to get the code working. There are four typographical errors in the code as highlighted below

```
$Monitor = Get-WmiObject wmiMonitorID -namespace root\wmi
$Computer = Get-WmiObject -Class Win32_ComputerSystem

$Monitor ¦ %{
    $psObject = New-Object PSObject
```

```
       $psObject | Add-Member NoteProperty ComputerName ""
    $psObject | Add-Member NateProperty ComputerType ""
    $psObject | Add-Member NoteProperty ComputerSerial ""

    $psObject | Add-Member NoteProperty MonitorSerial ""
    $psObject | Add-Member NoteProperty MonitorType ""

    $psObject.ComputerName = $env:computernome
    $psObject.ComputerType = $Computer.model
    $psObject.ComputerSerial =  $Computer.Name

    $psObject.MonitorSerial = ($_.SerialNumberID -ne 0 | %{[char]$_}) -join ""
    $psObject.MonitorType = ($_.UserFriendlyName -ne 0 | %{[char]$_}) -join ""

    $ps0bject

}
```

The first error has a ¦ instead of a pipeline symbol | in this line:

```
$Monitor ¦ %{
```

The second use of Add-Member uses NateProperty instead of NoteProperty:

```
$psObject | Add-Member NateProperty ComputerType ""
```

Thirdly, the environment variable should be computername not computernome:

```
$psObject.ComputerName = $env:computernome
```

The final error is the in the last line with the use of a zero 0 instead of capital O

```
$ps0bject
```

The corrected code looks like this:

```
$Monitor = Get-WmiObject wmiMonitorID -namespace root\wmi
$Computer = Get-WmiObject -Class Win32_ComputerSystem

$Monitor | %{
    $psObject = New-Object PSObject

       $psObject | Add-Member NoteProperty ComputerName ""
    $psObject | Add-Member NoteProperty ComputerType ""
    $psObject | Add-Member NoteProperty ComputerSerial ""

    $psObject | Add-Member NoteProperty MonitorSerial ""
    $psObject | Add-Member NoteProperty MonitorType ""

    $psObject.ComputerName = $env:computernome
    $psObject.ComputerType = $Computer.model
    $psObject.ComputerSerial =  $Computer.Name

    $psObject.MonitorSerial = ($_.SerialNumberID -ne 0 | %{[char]$_}) -join ""
    $psObject.MonitorType = ($_.UserFriendlyName -ne 0 | %{[char]$_}) -join ""

    $psObject

}
```

If you run the code it may or may not run without error because some monitors don’t define a monitor type.

```
PS> $monitor | select UserFriendlyName


UserFriendlyName
----------------
{50, 50, 69, 65...}
```

```
PS>
```

So, an error is generated:

```
ComputerName   :
ComputerType   : 4318CTO
ComputerSerial : W510W10
MonitorSerial  : 304NDJX51788
MonitorType    : 22EA63

Cannot convert value "True" to type "System.Char". Error: "Invalid cast from
'Boolean' to 'Char'."
At line:19 char:60
+ ... sObject.MonitorType = ($_.UserFriendlyName -ne 0 | %{[char]$_}) -join ...
+                                                        ~~~~~~~~~
    + CategoryInfo          : InvalidArgument: (:) [], RuntimeException
    + FullyQualifiedErrorId : InvalidCastIConvertible

ComputerName   :
ComputerType   : 4318CTO
ComputerSerial : W510W10
MonitorSerial  : 0
MonitorType    :
```

You can deal with this by testing for a $null value:

```
PS> $monitor[0].UserFriendlyName -eq $null

PS> $monitor[1].UserFriendlyName -eq $null
True
```

You now have working code. The next step is to create a re-usable artefact. In PowerShell terms this means creating a function that you could eventually add to a module.

```
function Get-MonitorInfo {
  $Monitor = Get-WmiObject wmiMonitorID -namespace root\wmi
  $Computer = Get-WmiObject -Class Win32_ComputerSystem

  $Monitor | %{
    $psObject = New-Object PSObject

       $psObject | Add-Member NoteProperty ComputerName ""
    $psObject | Add-Member NoteProperty ComputerType ""
    $psObject | Add-Member NoteProperty ComputerSerial ""

    $psObject | Add-Member NoteProperty MonitorSerial ""
    $psObject | Add-Member NoteProperty MonitorType ""

    $psObject.ComputerName = $env:computername
    $psObject.ComputerType = $Computer.model
    $psObject.ComputerSerial =  $Computer.Name

    $psObject.MonitorSerial = ($_.SerialNumberID -ne 0 | %{[char]$_}) -join ""

    if ($_.UserFriendlyName -ne $null){
       $psObject.MonitorType = ($_.UserFriendlyName -ne 0 | %{[char]$_}) -join ""
    }

    $psObject

  }
}
```

The basic tasks are now complete. Its time to look at the secondary aims. The easiest one to deal with first is the requirement to follow best practice. There are a number of places where current best practice isn't followed in this code including:

- Use of positional parameters on Get-WmiObject
- Fetching the WMI monitor data into a variable and then piping the variable contents into Foreach-Object is an unnecessary step_
- Use of aliases - % for Foreach-Object

- Creating an object, using Add-Member to create properties and the setting the values of those properties create unnecessary actions within the code that slow it down and make it harder to maintain
- Standardising the variable usage (by case) improves the readability of the code

Fixing these issues gives something like this:

```
function Get-MonitorInfo {
  $computer = Get-WmiObject -Class Win32_ComputerSystem

  Get-WmiObject -Class wmiMonitorID -namespace root\wmi | foreach {
    $minfo= New-Object -TypeName PSObject -Property @{

      ComputerName = $env:computername
      ComputerType = $Computer.model
      ComputerSerial =  $Computer.Name
      MonitorSerial = ($_.SerialNumberID -ne 0 | foreach {[char]$_}) -join ""
      MonitorType = ''

    }
    if ($_.UserFriendlyName -ne $null){
      $minfo.MonitorType = ($_.UserFriendlyName -ne 0 | foreach {[char]$_}) -join ""
    }

    $minfo
  }
}
```

The code is much more compact and understandable.

There are still two generic requirements to meet – ensuring the code works on PowerShell v6 and dealing with remote machines. Remote machines are managed by adding the ComputerName parameter to Get-WmiObject. You need to get the computer name to the cmdlets so you need to add a parameter to the function

The code still won't run on PowerShell v6.0 because the WMI cmdlets aren't available. The CIM cmdlets are available so a simple change of cmdlet name solves this issue. Combining these changes gives:

```
function Get-MonitorInfo {
  param (
    [string]$ComputerName
  )

  $computer = Get-CimInstance -ClassName Win32_ComputerSystem -ComputerName $ComputerName

  Get-CimInstance -ClassName wmiMonitorID -namespace root\wmi -ComputerName $ComputerName |
foreach {
    $minfo= New-Object -TypeName PSObject -Property @{

      ComputerName = $computer.Name
      ComputerType = $Computer.model
      ComputerSerial =  $Computer.Name
      MonitorSerial = ($_.SerialNumberID -ne 0 | foreach {[char]$_}) -join ""
      MonitorType = ''

    }
    if ($_.UserFriendlyName -ne $null){
      $minfo.MonitorType = ($_.UserFriendlyName -ne 0 | foreach {[char]$_}) -join ""
    }

    $minfo
  }
}
```

At this point you've met the technical requirements of the challenge but what about the faction orientated requirements? Remember the faction aims:

- Daybreak Faction - beautiful code
- Flawless Faction - flawless code
- Battle Faction - good enough to get the job done

For Battle faction the current solution is probably enough. The code works and gets the job done. You can now start to solve another problem and create more code to meet your faction's ultimate goal.

Daybreak faction is renowned for its beautiful code. How can you meet this goal? A little bit of reformatting will take you a long way to meeting Daybreak faction's expectations:

```
function Get-MonitorInfo
{
  param
  (
    [string]$ComputerName
  )

  $computer = Get-CimInstance -ClassName Win32_ComputerSystem -ComputerName $ComputerName

  Get-CimInstance -ClassName WmiMonitorID -Namespace ROOT/wmi -ComputerName $ComputerName |
  ForEach-Object  -Process {

     $minfo= New-Object -TypeName PSObject -Property @{
       ComputerName    = $computer.Name
       ComputerType    = $computer.model
       ComputerSerial  = $computer.Name
       MonitorSerial   = (
                           $psitem.SerialNumberID -ne 0 |
                           ForEach-Object {[char]$psitem}
                         ) -join ''
       MonitorType     = ''
     }

    if ($psitem.UserFriendlyName -ne $null)
    {
      $minfo.MonitorType = (
                              $psitem.UserFriendlyName -ne 0 |
                              ForEach-Object {[char]$psitem}
                           ) -join ''
    }

    $minfo
  }
}
```

A dedicated follower of the Daybreak faction will undoubtedly have other things they want to change but that's a task best left to them.

Flawless faction believes in flawless code. This means no mistakes, surprises and everything is controlled and understood. A member of the flawless faction may decide the code should look like this:

```
function Get-MonitorInfo {
  [CmdletBinding()]
  param (
    [Parameter(Mandatory=$true)]
    [ValidateScript({Test-Connection -ComputerName $_ -Quiet -Count 1})]
    [string]$ComputerName
  )

  try {
    $computer = Get-CimInstance -ClassName Win32_ComputerSystem -ComputerName $ComputerName -
ErrorAction Stop
  }
  catch {
    Throw "Could not retreive Win32_ComputerSystem data for computer $ComputerName"
  }

  Write-Verbose -Message "Retreived Win32_ComputerSystem data for computer $ComputerName"
  Write-Debug -Message $Computer

  try {
    Get-CimInstance -ClassName WmiMonitorID -namespace root\wmi -ComputerName $ComputerName -
ErrorAction Stop |
    foreach {
      $minfo= New-Object -TypeName PSObject -Property @{

        ComputerName = $computer.Name
        ComputerType = $computer.model
        MonitorSerial = ($_.SerialNumberID -ne 0 | foreach {[char]$_}) -join ""
        MonitorType = ''

      }
      if ($_.UserFriendlyName -ne $null){
        $minfo.MonitorType = ($_.UserFriendlyName -ne 0 | foreach {[char]$_}) -join ""
      }

      $minfo

        Write-Verbose -Message "Retreived Monitor data for computer $ComputerName"
        Write-Debug -Message $minfo
    }
  }
  catch {
    Throw "Could not create monitor information object for computer $ComputerName"
  }
}
```

The ComputerSerial property was duplicating the ComputerName property and was deleted as being unnecessary. The ComputerName parameter is made mandatory to ensure it’s not missed and the connection to the remote machine is tested to ensure its available. Try-Catch blocks are used to manage exceptions. Verbose and Debug statements are added to aid tests. Flawless faction’s goal is that the code will run – if there are issues they are managed so that nothing interrupts the flawless execution of the code.

You may decide, as a member of your faction, that the aims of the faction haven’t been fully met in which case feel free to post your solutions to the Iron Scripter prequel forum on PowerShell.org.