Advanced Scripting   
OO Models

Last Updated: 3/13/2024 6:25 AM Version 2  
Document Prepared for: CYBER360 Student

# Name Click here to enter name ID Click here to enter id

# Instructions

Save a copy of this document. Use Microsoft Word to edit and answer all questions directly in this document. You will save and upload this completed document as your homework submission.

# Overview

The folks at Microsoft made object-oriented models of phenomena that most Cybersecurity professionals work with every day: timestamps, file metadata, and IP addresses. Let’s explore these.

# Setup

Launch PowerShell Core (pwsh).

# Task 1—Objects for dates and times

## Steps

1. At your pwsh prompt, enter   
   **$earlier = Get-Date**
2. Wait a few seconds or longer, then enter   
   **$later = Get-Date**   
   Now you have two variables, one named **$earlier** and the other named **$later**.
3. Show the data assigned into the first variable:   
   **$earlier**   
   Your output: Click or tap here to enter text. This is an example of using PowerShell as an *expression evaluator*. It didn’t actually execute a command. Instead, it interrogated a variable and retrieved its value.
4. Show the data assigned to the other variable:   
   **$later**   
   Your output: Click or tap here to enter text. As you have probably already figured out, all variables in PowerShell start with a dollar symbol, and that symbol must be present in both get (read, retrieval) and set (write, assignment) operations. *(Note: if you have bash experience, you’ll recognize this as slightly different than bash variable syntax. In bash, read operations use the dollar symbol prefix, but write assignments don’t.)*
5. Let’s see what abstract data type (or class) is stored in each variable:
   1. Enter:   
      **$earlier.GetType()**
   2. Your output should consist of four properties, named **IsPublic**, **IsSerial**, **Name**, and **BaseType**, respectively. What is the value of the **Name** property? Click or tap here to enter text.
   3. **$later.GetType()**   
      The output for both variables is identical, because even though the object *instances* are different, their abstract data types are the same.
      1. Each of these commands is an example of passing a ***message*** to an object. As with most object-oriented coding languages, this happens using the dot (**.**, period) operator. The method on the right side of the dot, **GetType()**, is the message that is passed. The variable on the left contains the object that receives the message.
      2. *Every* PowerShell object has a **GetType()** method. In other words, *every* object will respond to a **.GetType()** message.
   4. Let’s send a different message, using the Second property:
      1. **$earlier.Second**   
         Your output: Click or tap here to enter text.
      2. **$later.Second**   
         Your output: Click or tap here to enter text.
      3. In each of these commands, the *property* (to the right of the dot) is the message that is sent to the *variable* (to the left of the dot).
      4. Properties are a lot like variables. The difference is that a property contains just one component of an object’s structured data, while a variable usually contains the entirety of the object’s structured data.
   5. Let’s try using assignments to change the **Second** property of each object:
      1. **$earlier.Second = 1**
      2. **$later.Second = 59**
      3. Did either of those successfully change that property? (If not, what happened?) Click or tap here to enter text.
   6. Pipe one of the objects into the Get-Member command, such as:   
      **$later | Get-Member**
      1. Look at the beginning lines of your output. How is the **TypeName** different from the **Name** property that was reported by the **GetType()** method? Click or tap here to enter text.
      2. The rest of the output is a table. Look at the column headings: **Name**, **MemberType**, and **Definition**. Do you see the **GetType** method name in one of the rows of the table?
      3. Look at the other methods. One of them is named **AddYears** and its description looks like this: **datetime AddYears(int value)**
      4. Let’s use the **AddYears()** method to get the date and time exactly one year after **$later**. Enter this command:   
          **$nextYear = $later.AddYears(1)**
      5. Go back to the table and look at the list of Properties. Do you see the **Second** property?
      6. Notice that there’s another property called **DayOfWeek**. Let’s use it to see what next year’s day of the week will be:   
         **$nextYear.DayOfWeek**   
         Your output: Click or tap here to enter text.
   7. PowerShell wasn’t developed completely from scratch. It leverages Microsoft’s .NET technologies.
      1. **$later.GetType().FullName**   
         Your output: Click or tap here to enter text.
      2. The **DateTime** class is part of .NET’s System assembly. Visit <https://learn.microsoft.com/en-us/dotnet/api/> to search Microsoft’s .NET API browser. In the search field there, enter **System.DateTime**. Select the first entry among the search results, which is [**System.DateTime**](https://learn.microsoft.com/en-us/dotnet/api/system.datetime) **Struct**, as shown here: A screenshot of a browser

         Description automatically generated
      3. That will bring you to this web page: <https://learn.microsoft.com/en-us/dotnet/api/system.datetime> . You will see a lot of C# code on that page. Most of the NET technologies were developed using the object-oriented *compiled* language C# (pronounced “see-sharp”). Don’t worry; you do not need to know C#, nor have any experience with it, to successfully use PowerShell.
      4. In your web browser, quickly scroll down about three-quarters of the way through that web page, until you find these headings: **Constructors**, **Fields**, **Properties**, **Methods**, and so forth. Remember the output of Get-Member from the previous step? Among other things, it showed a bunch of methods and properties. This web page is where you will find additional documentation for each of those methods and properties.
      5. Under the Methods heading, select **ToUniversalTime()**. According to the **Returns** header on that web page, an object of what class is returned by this method? Click or tap here to enter text.
      6. Go back to your PowerShell prompt and try it:   
         **$later.ToUniversalTime()**   
         Your output: Click or tap here to enter text.
      7. Enter:  
         **$later.ToUniversalTime().Kind**   
         Is your output consistent with the documentation that you found on the API webpage for the **ToUniversalTime()** method? Click or tap here to enter text.

# Task 2—Objects for file metadata

## Steps

1. Change to your filesystem’s temporary folder, with **Set-Location C:\Temp** if you’re using Windows, or **Set-Location /tmp** if you’re using Linux or macOS.
   1. *If you used a short alias like* **cd** *instead of the full command name* **Set-Location**, *congratulate yourself on wisely choosing efficiency instead of blindly following instructions!*
2. Create a new file named **file1.tmp**:  
   **New-Item file1.tmp**
3. Put some text into that file:  
   **Set-Content file1.tmp important\_data**
4. List the new file and put the result in the variable **$item**. Then get its type:  
   **$item = Get-ChildItem file1.tmp**  
   **$item.GetType()**  
   Record the type’s name here: Click or tap here to enter text.. This object doesn’t represent the contents of the file. Rather, it represents the file’s *metadata*.
5. Pass the object contained in **$item** through a pipe to the **Get-Member** command:  
   **$item | Get-Member**
   1. What is the object’s full TypeName? Click or tap here to enter text.
   2. List all four methods that start with the letter O: Click or tap here to enter text.
6. Let’s look at some of the properties we saw in the output of the **Get-Member** command:
   1. **$item.DirectoryName**  
      Resulting output: Click or tap here to enter text.
   2. **$item.BaseName**  
      Resulting output: Click or tap here to enter text.
   3. **$item.Extension**  
      Resulting output: Click or tap here to enter text.
   4. **$item.CreationTime**  
      Resulting output: Click or tap here to enter text.
   5. **$item.Length**  
      Resulting output: Click or tap here to enter text.
7. Visit <https://learn.microsoft.com/en-us/dotnet/api/> again, and search for **System.IO.Fileinfo**. Select the first entry among the search results, which is [**System.IO.FileInfo**](https://learn.microsoft.com/en-us/dotnet/api/system.io.fileinfo) **Class**.
   1. Scroll about halfway down that web page to find Constructors, Fields, Properties, Methods, and so forth. What result would we get from the Extension property for a file that does not have an extension suffix? Click or tap here to enter text.
8. Clean up:  
    **Remove-Item file1.tmp**

# Task 3—Objects for IP network addresses

## Steps

1. Visit <https://learn.microsoft.com/en-us/dotnet/api/> again, and search for **IPAddress**. Select the first entry among the search results, which is [**System.Net.IPAddress**](https://learn.microsoft.com/en-us/dotnet/api/system.net.ipaddress) **Class**. Scroll down and browse the properties and methods of that class.
2. Create variables that contain this private IPv4 (Internet protocol version 4) address: 192.168.3.4:  
   **$addr\_string = "192.168.3.4"**   
   **$addr\_string.GetType()** # Your output shows that it’s a **String** object.  
   **$addr\_ip = [IPAddress] "192.168.3.4"**   
   **$addr\_ip.GetType()** # Your output shows that it’s an **IPAddress** object.  
   **$addr\_ip.GetType().FullName**   
   Your output: Click or tap here to enter text.
3. Dig into the **IPAddress** object to find its properties and methods:
   1. **$addr\_ip | Get-Member**   
      You should see many, but not all, of the properties and methods you browsed at the [**System.Net.IPAddress**](https://learn.microsoft.com/en-us/dotnet/api/system.net.ipaddress) **Class** documentation web page. Some of them are missing.
   2. **$addr\_ip | Get-Member -Static**   
      You should now see the properties and methods that were missing. (Later we’ll study more about the difference between *instance* and *static* members.)
   3. Enter   
      **$addr\_ip | Get-Member -MemberType Property**   
      to just see the instance properties.
      1. Let’s examine the value of one of them:  
         **$addr\_ip.AddressFamily**   
         Your output: Click or tap here to enter text.
   4. Enter   
      **$addr\_ip | Get-Member -MemberType Method**   
      to just see the instance methods.
      1. Let’s invoke one of them:  
         **$addr\_ip.MapToIPv6()**   
         What is your output’s value for **IPAddressToString**? Click or tap here to enter text. (This is how an IPv4 address is represented in IPv6.)
   5. **$addr\_ip | Get-Member -MemberType Properties**   
      Notice that **IPAddressToString** isn’t a regular property. What is its **MemberType**? Click or tap here to enter text.

# Deliverable

Upload this document with completed answers to I-Learn Canvas.