Advanced Scripting   
OO Princples

Last Updated: 3/13/2024 7:17 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

Let’s see some examples of the principles Abstraction, Encapsulation, Inheritance, and Polymorphism in the .NET assemblies upon which PowerShell was built. We’ll introduce a few useful features of PowerShell along the way.

# Setup

Launch PowerShell Core (pwsh).

# Task 1—Abstraction

Let’s use two of the built-in numeric primitive data types to illustrate the concept of abstraction.

## Steps

1. At your pwsh prompt, enter:   
   **0.GetType()**   
   You should see a “ParserError” exception message. Copy all five lines of the error message here: Click or tap here to enter text.   
   PowerShell started to misinterpret that as a command that it should execute, as if there was an executable file named *0* with filename extension *GetType*, or perhaps an executable file named *0.GetType.exe*. But then the parentheses made it try to interpret what you typed as an expression instead of a command, which confused its parser.
2. In a PowerShell expression, use parentheses for evaluation precedence. In other words, parentheses (round brackets) signify: “evaluate this first.” Try starting the expression with a parenthesis, like this:   
    **(0).GetType()**   
   What’s the name of this data type? Click or tap here to enter text.
3. **(1).GetType()**   
   What’s the name of this data type? Click or tap here to enter text.
4. **(0.0).GetType()**   
   What’s the name of this data type? Click or tap here to enter text.
5. **(1.1).GetType()**   
   What’s the name of this data type? Click or tap here to enter text.
6. Each of 0, 1, 0.0, and 1.1 is a distinct *literal* number, and PowerShell interpreted each as an object *instance*. However, each instance was interpreted as one of two different *classes*, or object *types*: Int32 (thirty-two-bit integer) or Double (double-precision floating point). In your own words, what do you think is the difference between these two abstract data types? Click or tap here to enter text.
7. For primitive data types like numbers, we can use the name of the class to convert (or *cast*) from one type to another.
   1. **[double]1**   
      Your output: Click or tap here to enter text.
   2. **([double]1).GetType().Name**   
      Your output: Click or tap here to enter text.
   3. **[int]1.1**   
      Your output: Click or tap here to enter text.   
      *(You can use* **int** *as a common alias for* **Int32***.)*   
      Notice that the number changed. Integer data types can never have fractional parts.
   4. **[int]1.9**   
      Examine your output. Did PowerShell *truncate* the fractional part, or did it *round* the double precision floating point value to its nearest integer? Click or tap here to enter text.
   5. **([int]1.9).GetType().Name**   
      Your output: Click or tap here to enter text.

# Task 2—Encapsulation

PowerShell is an interpreted scripting language, *not* a compiled language. A script’s source code *is* its program, so you won’t ever see *private* implementation details specified in scripts. But the **Get-Member** cmdlet is a useful tool for observing the *public* properties and methods encapsulated inside an object or a class.

## Steps

1. At your pwsh prompt, enter   
   **0 | Get-Member**   
   You should see a table showing the methods encapsulated inside the **Int32** abstract class.
   1. **(0 | Get-Member).Count**   
      How many methods are in the **Int32** class? Click or tap here to enter text.
   2. Enter   
      **0 | Get-Member | Measure-Object**   
      This is another convenient way to count results. (*You can also use the alias* **measure**. *Try it!*)
   3. **(1.1 | Get-Member).Count**   
      How many methods are in the **Double** class? Click or tap here to enter text.
   4. **"zero" | Get-Member**   
      You should see a table showing the methods and properties of the String abstract class.
   5. **("zero" | Get-Member).Count**   
      How many members are in the **String** class? Click or tap here to enter text.
   6. **("zero" | Get-Member -MemberType Method).Count**   
      How many methods are encapsulated in the **String** class? Click or tap here to enter text.
   7. **("zero" | Get-Member -MemberType Property).Count**   
      How many properties are available in the **String** class? Click or tap here to enter text.
   8. The string class has yet another member, but its **MemberType** is neither Method nor Property. What’s the **Name** of that member? Click or tap here to enter text. What’s the **MemberType** of that member? Click or tap here to enter text.
2. Assign a String value to a variable: **$s = "1.1"**
   1. What’s the value of the **Length** property of that string? Click or tap here to enter text. (*Hint: pass the Length message to the variable, like this:* **$s.Length**. *You can also get the same result by passing the message to the string literal, like this:* **("1.1").Length**. *Try it!*)
   2. The characters of this string can be interpreted as a double-precision floating point value, so let’s try using the **ToDouble()** method to convert the string to a double: $s.ToDouble(). You’ll notice it threw an exception. Copy the error message here: Click or tap here to enter text.
   3. In your output of step 1.4, closely examine the ToDouble row of the table. Glancing at the description, it looks like we mustn’t leave the parentheses empty when we use the method. We need to find a “provider” object to put between the parentheses. The .NET architects at Microsoft made a class **CultureInfo**. We can use those objects as format providers. Let’s start with the “English-United States” culture provider:   
        
       **$value = s.ToDouble([CultureInfo]"en-us")**  
       **$value.GetType().Name**   
      Your output: Click or tap here to enter text.
   4. Many cultures use a comma instead of a period to separate the integer and fraction parts of a floating point number. Change the string variable accordingly, and we’ll try the “Portuguese-Brazil” provider:   
        
      **$s = "1,1"**   
      **$value = s.ToDouble([CultureInfo]"pt-br")**   
      **$value**   
      Your output: Click or tap here to enter text.
   5. **$value.GetType().Name**   
      Your output: Click or tap here to enter text.

# Task 3—Inheritance

## Steps

1. Create a generic object. Enter:   
   **$a = New-Object -Type Object**.   
   A generic object has no properties but has a few methods.
   1. Enter: **$a | Get-Member**   
      You see four instance methods. Let’s take advantage of PowerShell to make it easier to copy the method names to this exercise document.
   2. **($a | Get-Member).Name**.   
      Now you should see a list of just the method names, instead of the entire member table.
   3. **($a | Get-Member).Name -join ", "**   
      Now you should see the method names conveniently arranged on just one line of output, separated by commas. Copy your list here: Click or tap here to enter text.   
      (*In almost all your exercises, whenever your result is a multiple-value array or a list, you should use -join to combine all of the elements on one line, rather than clutter up your submitted document with every element on its own line.*)
2. Every class in .NET and PowerShell is *derived* from the **Object** class. This means that every object inherits the methods of the **Object** class. Using a “family tree” metaphor, we also say that every class is a child (or grandchild, or great-grandchild, etc.) of the **Object** class. We’ve already been using the **GetType()** method. Let’s use our variables from Task 2 to try out the other three instance methods inherited from **Object**:
   1. Enter: **$value.ToString()**. Your output: Click or tap here to enter text.
   2. Enter: **$s.ToString()**. Your output: Click or tap here to enter text.
   3. Enter: **$value.GetHashCode()**. Your output: Click or tap here to enter text.
   4. Enter: **$value.Equals($s)**. Your output: Click or tap here to enter text.

# Task 4—Polymorphism

## Steps

1. By default, the ToString() method just returns the full name of the object’s class:
   1. **$a.ToString()**   
      Your output: Click or tap here to enter text.
   2. **$a.GetType().FullName**   
      Your output: Click or tap here to enter text.
   3. **$a.ToString().Equals($a.GetType().FullName)**   
      Your output: Click or tap here to enter text.
   4. In your own words, try to explain why you get the result you recorded in step 1.3: Click or tap here to enter text.
2. **$x = New-Object -Type Exception**   
    **$x.ToString()**   
   Your output: Click or tap here to enter text.   
   *(Note: You did not produce any actual errors, even though your output looks like an error message! This is merely the full name of one of the classes that is used in .NET’s error handling system. But the output was slightly altered: its default error message was appended after the full class name.)*
3. Almost all of the object classes (data types) in .NET and PowerShell alter the behavior of the inherited **ToString()** method, to make it produce a useful string to represent an object’s properties, rather than just its abstract class name.
   1. **$value.ToString()**   
      Your output: Click or tap here to enter text.
   2. **(New-Object -Type DateTime).ToString()**   
      Your output: Click or tap here to enter text.
   3. In your own words, try to explain why you get the result you recorded in step 3.2: Click or tap here to enter text.
   4. **$ip = New-Object -Type IPAddress -ArgumentList 0**   
       **$ip.ToString()**   
      Your output: Click or tap here to enter text.
   5. In your own words, try to explain why you get the result you recorded in step 3.4: Click or tap here to enter text.
4. Altering the code that gets executed when a method is invoked is just one way that a class may implement polymorphism. There’s another way, known as “run-time” polymorphism. Let’s see how the **+** “plus” operator may behave differently at run-time when it is used in expressions.
   1. **4 + "2"**   
      Your output Click or tap here to enter text.
   2. **"4" + 2**   
      Your output Click or tap here to enter text.
   3. **(4 + "2").GetType().Name**   
      Your output Click or tap here to enter text.
   4. **("4" + 2).GetType().Name**   
      Your output Click or tap here to enter text.
   5. Based on these four outputs, explain in your own words how the **+** operator behaved differently depending on the data types of its operands: Click or tap here to enter text.

# Deliverable

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