Advanced Scripting   
Static Methods and Instance Methods

Last Updated: 4/15/2024 12:20 PM Version 1  
Document Prepared for: CYBER360 Student

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

# Instructions

Save a copy of this document. Answer all questions directly in this document. You will save and upload this completed document as your homework submission.

# Overview

Now let’s learn how to make classes with not just properties, but methods too. By the end of this exercise, you will have coded a PowerShell class with a *static property*, two *instance properties*, three *constructor* methods, two *static methods*, and two *instance methods*.

# Requirements

* PowerShell
* VS Code

# Setup

# Task 1—Create a Circle class

Unlike PowerShell functions, methods *must* declare a returned data type.

## Steps

1. In VSCode (or the PowerShell ISE if you prefer), create a file named Methods.ps1. Enter the following code to start your script:  
   A screenshot of a computer

   Description automatically generated
2. Run (dot-source) your code.
3. In the VSCode Terminal pane, create an instace of Circle to verify that it works.
4. Set the Diameter of the circle to some value, then interrogate the object to make sure it works.
5. Set the Color to some value, then interrogate the object to make sure it works.

# Task 2—Adding Methods to a Class

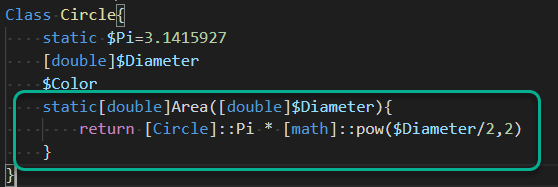
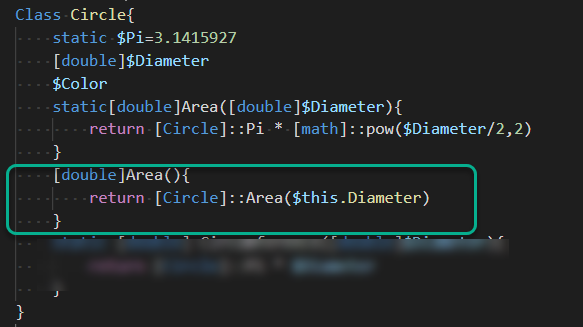
In PowerShell, the properties and methods in classes and objects are called *members*. Properties are essentially “variables” contained inside a class, and methods are essentially “functions” contained inside a class.

There are two types of members: *instance* and *static*. Instance members can only be used after an object of the class has been created (*instantiated*). Static members belong to the class itself; you don’t need an instance of the class to use a static member.

Notice that you created a static property in the class named **$Pi**. It can be accessed without an instance of the class. Here’s how:   
**[Circle]::Pi**

Let’s add some more code to our script. Remember, each time you make changes you will need to run your code again to see the changes.

## Steps

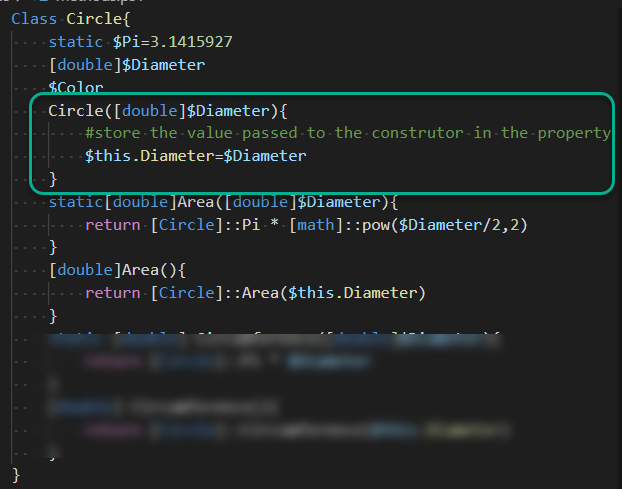
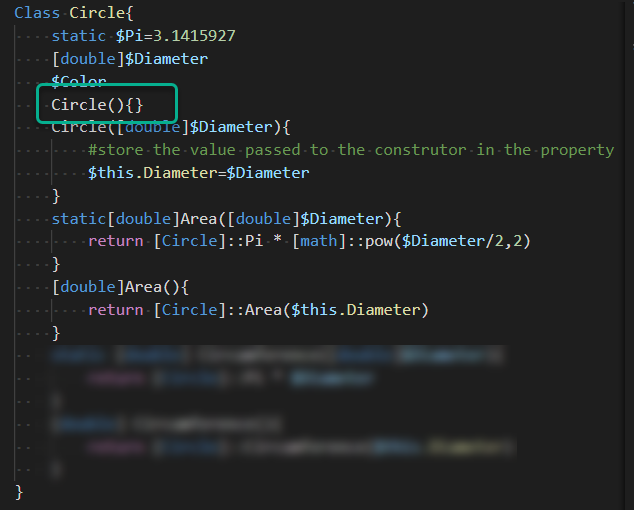
1. Create a static method that calculates the area of a circle. Code written for static methods can only access static properties, so our static method cannot access the $Diameter or $Color properties, because those are instance properties. This means that the static method needs a parameter, to pass the diameter value of the circle to the method. Add the following code to your class:   
      
   Let’s study this method’s code for just a moment.
   1. This method’s name is **Area**.
   2. The **static** keyword means this method belongs to the Circle class, not to any instance of the Circle class.
   3. The **[double]** declaration at the left of the method name means this method will return an object of type **Double**.
   4. Between the parentheses to the right of the method name is a *parameter* named Diameter. The argument for this parameter is *constrained* to only be of **Double** type.
   5. Now let’s look at the multiplication expression:
      1. **[Circle]::Pi** is PowerShell syntax; that’s how we access the value of the static **$Pi** parameter in our **Circle** class.
      2. **[math]::Pow()** is similar PowerShell syntax; that’s how we access the “raise to a power” method named **Pow** which is provided by the .NET library’s **Math** class.
      3. You should now recognize that the returned result is computed using the familiar “area of a circle” formula that you studied and used in your childhood math classes.
2. Save and execute your updated code. Use the new method to calculate the area of a circle that is 2.5 in diameter:   
   [Circle]::Area(2.5)
   1. What is its area? Click or tap here to enter text.
3. To calculate the Circumference of a circle, the formula is *Pi \* Diameter*. Create a static method to calculate a Circle’s circumference, patterned after our static **Area** method.
   1. Test it. What is the circumference of a Circle that has a diameter of 3.75? Click or tap here to enter text.
4. Now let’s create an instance method to calculate the area. This method won’t need any parameters; instead, it will just access the Diameter property. Here’s an obvious way to code it, ***but don’t do it this way!***   
     
   **[double]Area() {**   
    **$d = $this.Diameter**   
    **return [Circle]::Pi \* [math]::pow($d/2,2)**   
   **}**   
     
   Since we already have a static method that calculates the area, let’s just use that rather than rewrite the calculation expression. (By the way, we use the object placeholder **$this** as a *proxy* variable name when we access an instance member. Here’s why: there’s no way to know in advance what variable name the object might be stored in, or even whether it’s stored in a variable at all! So we use **$this** in our code, and PowerShell properly handles it when our code finally gets executed.)   
   Enter the following new code:   
   
5. Save and execute your script. To test the new instance method, first create an instance, then set the Diameter property:   
   $c1=[circle]::new()   
   $c1.Diameter=8   
   Then call the method:   
   $c1.Area()
   1. What is this circle object’s area? Click or tap here to enter text.
6. Now add to your script an instance method that calculates the circumference. Save your changes and test your code.

# Task 3—Create Constructors

A *constructor* is a method that is called the moment an object is created (*instantiated*). In PowerShell (as with almost every other object-oriented language), a constructor always has the same name as its class.

If you don’t provide a constructor, uninitialized properties inside every new instance of the class will be **$null**. Programmers and scripters often code constructor methods so that crucial properties are set to something more useful.

## Steps

1. If you don’t provide a constructor, uninitialized properties inside every new instance of the class will be **$null**. Try it!   
   **$c2 = [circle]::new()**   
   **$c2.Color.GetType()** # *this should cause a* “can’t call a method on a null value” *exception.*
2. Let’s create a constructor that allows you to provide a value to initialize the Diameter when a new Circle object is created. Add the following code.  
      
   (*Note*: it’s customary to write constructors before any other methods, as illustrated here.)
3. Run the script again and test it:   
   $c3=[circle]::new(12)   
   $c3   
    *Here’s how to do the same thing with the* **New-Object** *cmdlet*: **$c3=New-Object Circle(2)**
   1. *Important*: don’t let yourself be confused by the code in this constructor! On the left side of the assignment expression, **$this.Diameter** refers to the **$Diameter** *property* in the instance. On the right side of the assignment expression, **$Diameter** refers to the passed *parameter*. PowerShell always distinguishes properties from parameters, so it will never be confused if they happen to share a name. Neither should you!
4. Okay, now try to create an instance without passing the diameter value:   
   $c4=[circle]::new()
   1. What was the result? Click or tap here to enter text.
   2. Here’s what happened: Before we wrote a constructor, PowerShell automatically provided a default constructor that accepted no arguments and didn’t do any initialization. But after we wrote the constructor, PowerShell assumes we don’t want a default constructor anymore, and now it insists we use our new constructor that requires a Double value as an argument.
5. If we still want the default constructor, we have to actually code it, so let’s insert the following line:   
      
   *That code,* **Circle(){}** *, is the default* **Circle** *constructor that has no parameter and does nothing*.
6. Run your script and test it.   
   $c5=[circle]::new()
   1. Did it work? Click or tap here to enter text.
   2. If it’s not initialized, what’s the default value of a Circle’s **$Diameter** property? Click or tap here to enter text.
7. Add another constructor that accepts a diameter and a color.
   1. Run your script again and test your code.

# Wrap-up

Paste your completed code here:

Click or tap here to enter text.

# Deliverable

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