2023

Programming281

Lecturer: Raymond Hood

ASSIGNMENT 1

WERNER JANSE VAN RENSBURG (577930)

17/07/2023

Contents

Question 1:

a) Inheritance

According to (Taher, 2019) inheritance in general would be when something, this meaning an class in this case, gets something from someone else, meaning another class. (Taher, 2019) used the example when a child inherits his parents house one day. When can then use this knowledge to understand how inheritance in C#, using Object-Orientated-Programming (OOP), works. So when a class is created in C#, and we create another different class, but want to have the variables, methods or properties from the first class, we can use inheritance to allow the second class to access them. The first class would then be the parent class and the second class would then be the child class the receives from the parent. A concern that could be made is when you want a class only to inherit certain methods for example from another class. According to (Taher, 2019), the programmer can then make use of encapsulation to allow this. Encapsulation will be discussed later in this research assignment.

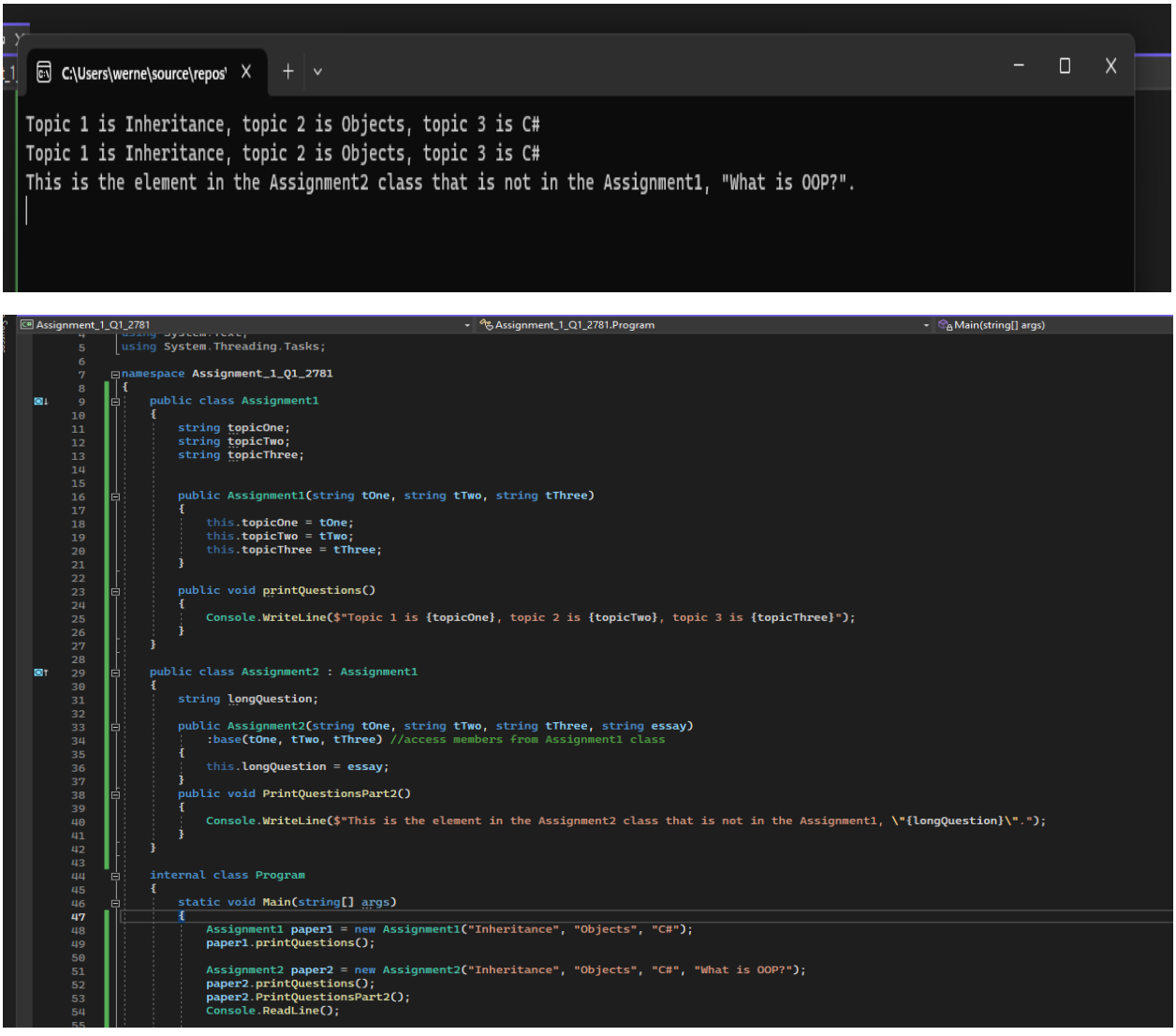

Figure 1 Inheritance Example

In the example created is a class called Assignment1 and gave it fields namely different topics for the an assignment. After it, a class called Assignment2 is created which inherits the topics from Assignment1 and then adds it own additional field, namely “longQuestion”. We inherit from the class by first stating the class you are creating, then a “:” followed by the class it inherits from. To allow the Assignment 2 to access Assignment1’s fields we use the “base:” keyword a Assignment2’s constructor which fetches these fields above.

b) Abstraction

According to (Taher, 2019) abstraction could be looked at as an abstract class in OOP. This means that the properties of this class for example methods are abstract as well and are declared with the abstract keyword. This class is only used by another class so it inherits from that certain class but overrides the abstract property or method. So the abstract method in the abstract method in the abstract class is declared but has no value. The programmer just uses the method in another class and is forced to if the parent class has a method marked as abstract. That is why the abstract class needs to be overwritten in the class the inherits the abstract class method. The reason to use abstract classes is to allow the use of one method in many classes that inherits from it.

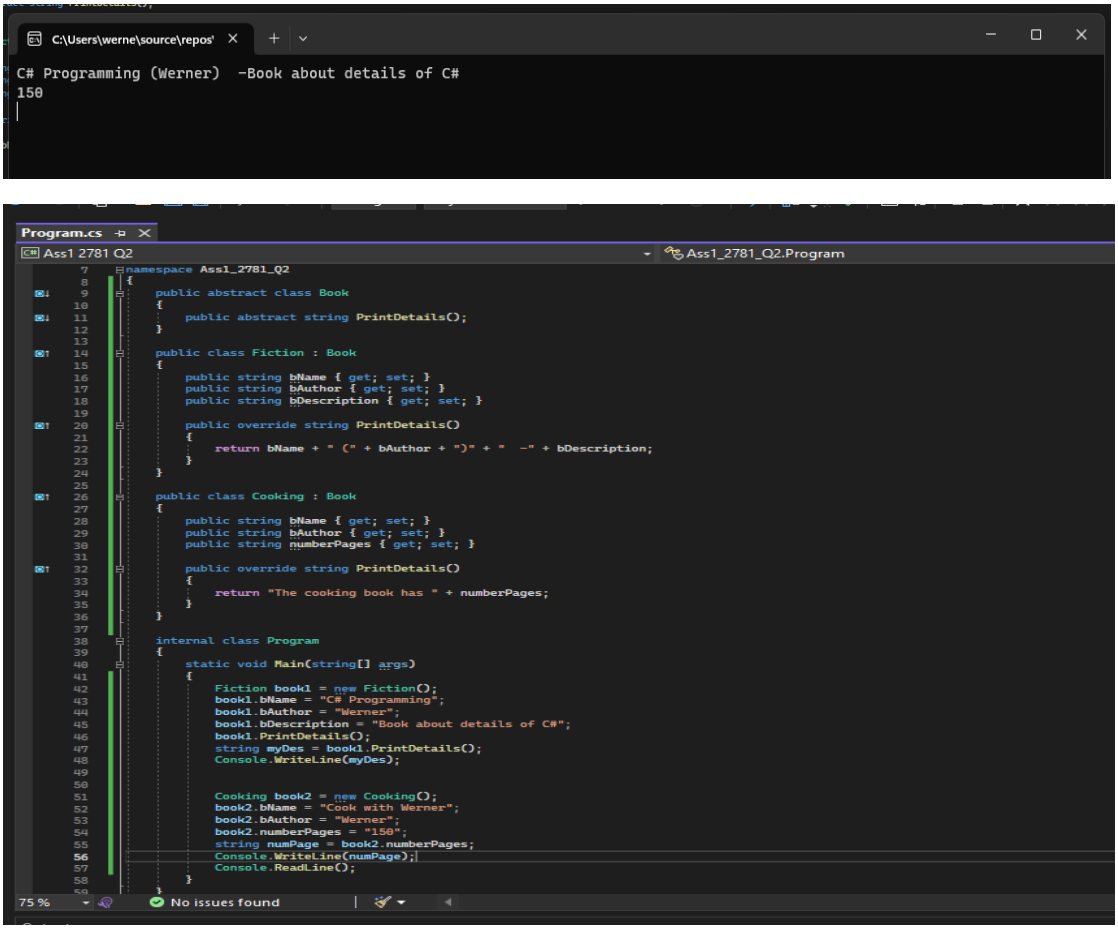

Figure 2 Abstraction Example

In the example above an abstract class *Book* is created with an abstract method called *PrintDetails()*. After this two classes of types of books are created namely, *Fiction and Cooking.* Both of these classes

inherit the abstract method from the book class, they are forced to. In each class the method is then overwritten to do something else. The method is then called in Main showing different outputs.

c) Encapsulation

Encapsulation can be seen as when the visibility of something is chosen. In C# this can be done using access modifiers: Public, Private, Protected, Internal, Internal Protected.

We can use encapsulation to specify which classes can be seen by each other. This means which classes have access to another class. This a form of control and security in your code.

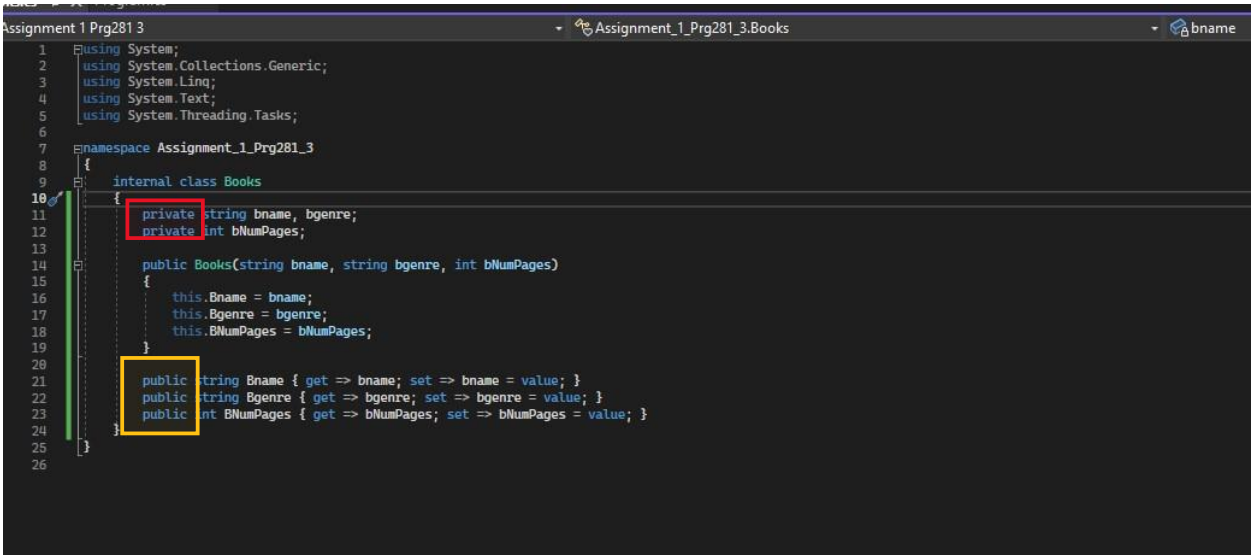

Figure 3 Creation of class Books using different encapsulation

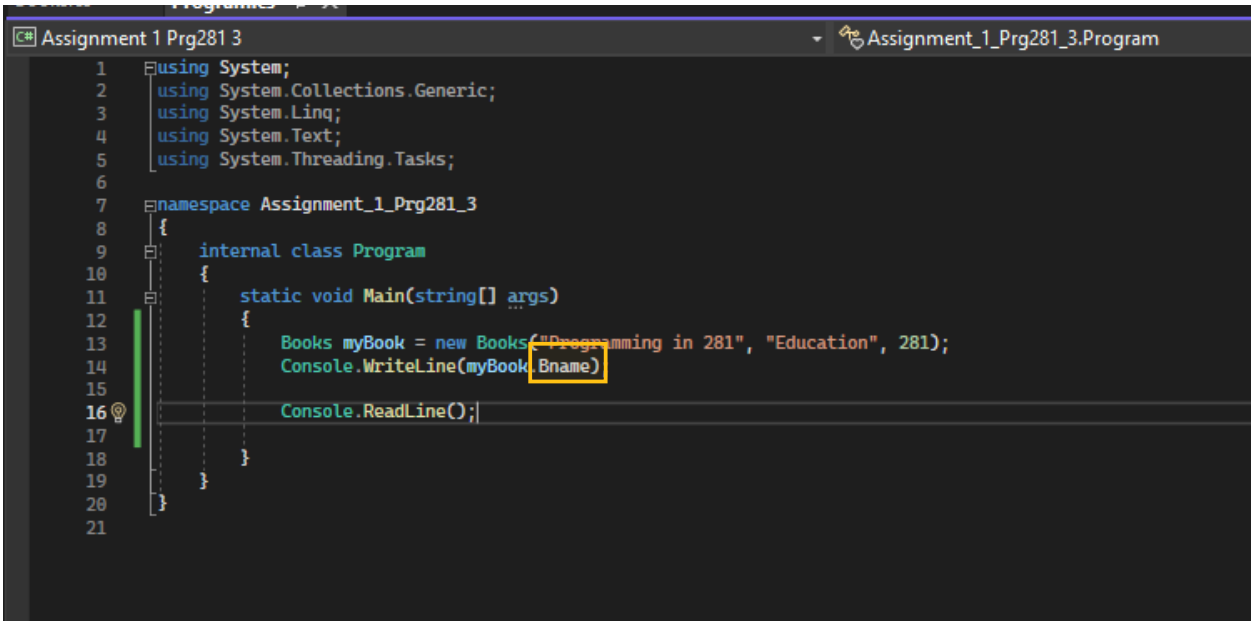

Figure 4 Creating new object from the Book class and printing the name given

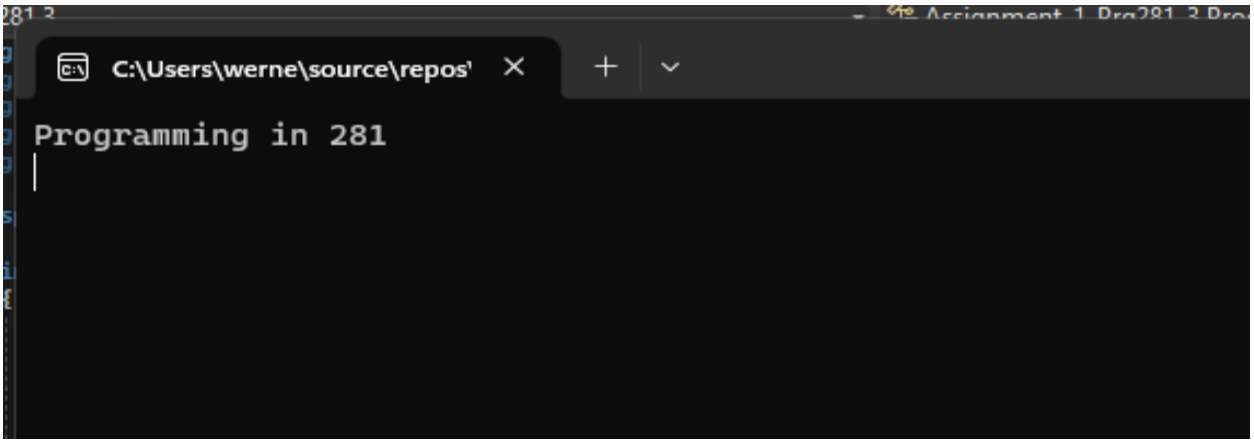

Figure 5 Output from class Program

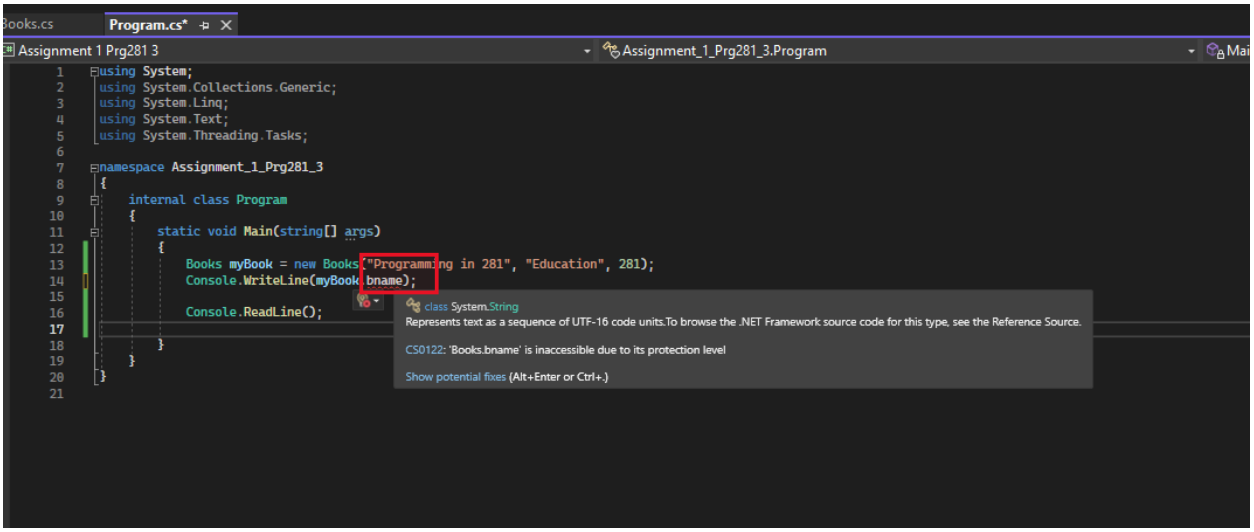

Figure 6 Showing error due message

In these examples above, encapsulation is explained. In figure 3 the red indicator shows that those fields are set to private. This means they are not accessible to the book object created in figure 6. That is why there is an error message displayed when trying to run the program. To give access to the inputs we create properties for each field in the class, as showed in figure 3. We make them public, given access to the Books constructor. The Books constructor is encapsulated to public allowing access to the Main to use this method to create new objects.

d) Polymorphism

The definition of polymorphism, according to (Turner, 2019) is anything that may take many different forms. An individual who plays several positions in a company is an example of polymorphism. As a result, one individual might take on several different responsibilities in this situation. Static and dynamic polymorphism are two separate forms of polymorphism that may be used in C# (Taher, 2019). In dynamic polymorphism, the role of a method would be defined at runtime as opposed to static polymorphism, where the role of a method (the form) is established by taking into account the compile time. An example of polymorphism would be in *Figure 2* when the *PrintDetails()* function is called exactly the same way in main but on different objects and display different outputs. So the function does multiple things, therefore we can say it has different roles.

Question 2:

As (Taher, 2019) polymorphism can be derived into static-and dynamic polymorphism. It can then be divided into further sections to explain it even better (Trivedi, 2023). Static polymorphism can be divided into Metod Overloading and Operator Overloading, while Dynamic polymorphism can be divided into Virtual/ Overriding Methods. We can then use these to implement it in OOP.

Static Polymorphism:

Method Overloading:

This is when a method has the same name in classes but the parameters of the method is different.

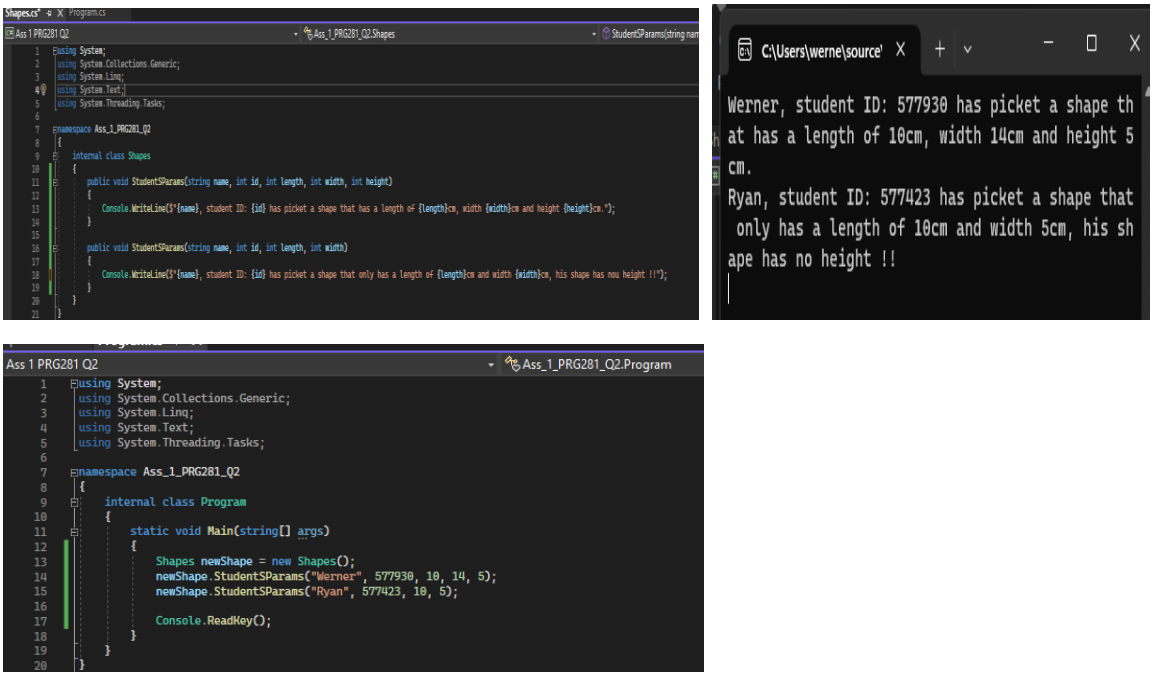

Figure 7 Method Overloading Example

Operator overloading:

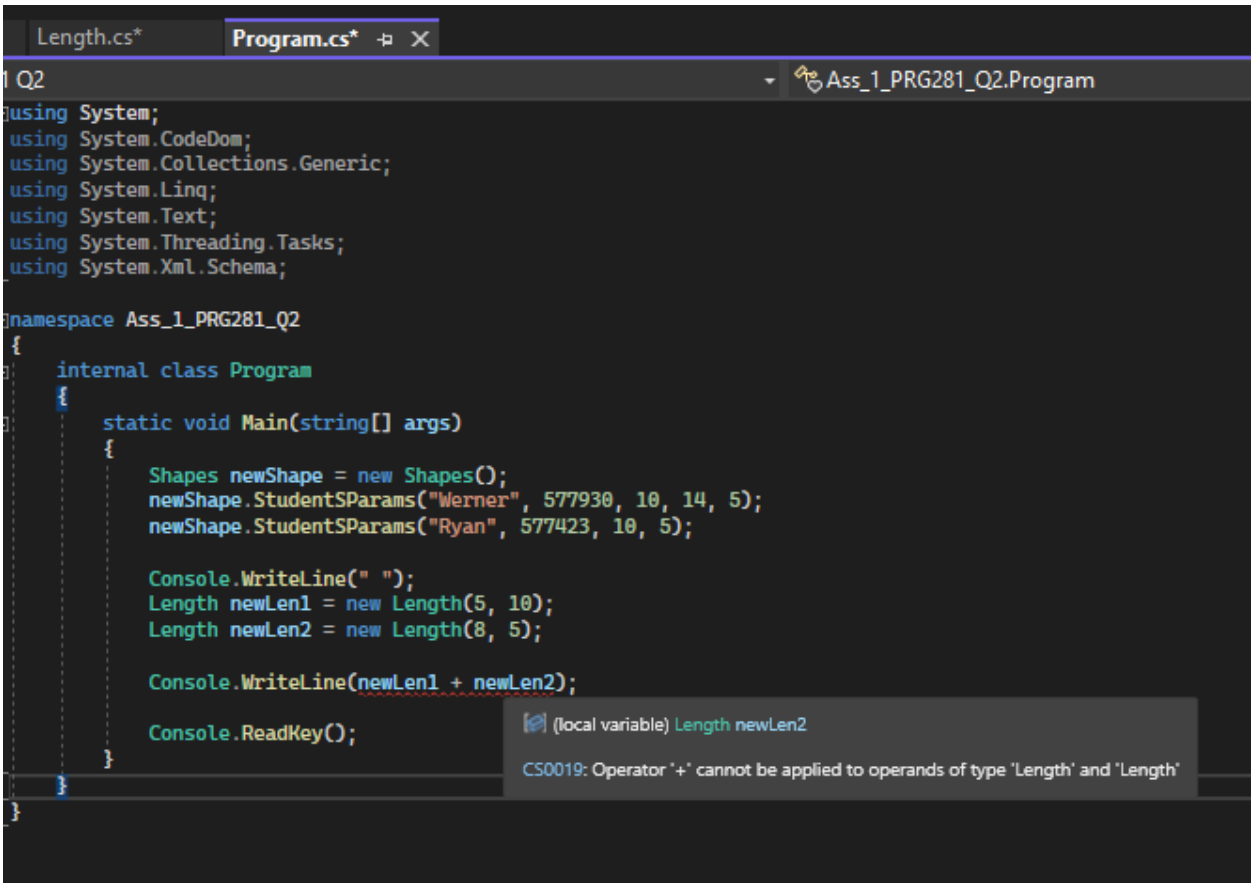

Figure 8 Trying to get the total perimeter of obj 1 and obj 2 combined

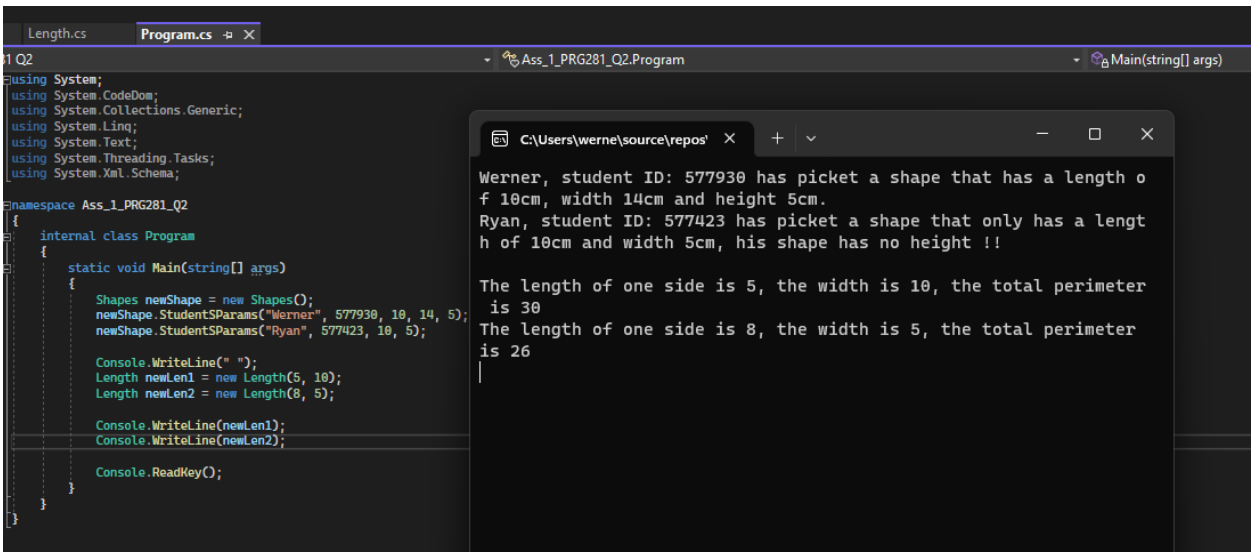

Figure 9 Can get each one individually

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Ass_1_PRG281_Q2
{
    internal class Length
    {
        private int myLen, myWidth;
        private int perimeter;


        public Length(int myLen, int myWidth)
        {
            this.MyLen = myLen;
            this.MyWidth = myWidth;
            this.Perimeter = (myLen * 2) + (myWidth * 2);
        }

        public int MyLen { get => myLen; set => myLen = value; }
        public int MyWidth { get => myWidth; set => myWidth = value; }
        public int Perimeter { get => perimeter; set => perimeter = value; }

        public override string ToString()
        {
            return $"The length of one side is {myLen}, the width is {myWidth}, the total perimeter is {Perimeter}";
        }

        public static Length operator +(Length obj1, Length obj2)
        {
            int totLen = obj1.myLen + obj2.myLen;
            int totWidth = obj1.myWidth + obj2.myWidth;
            return new Length(totLen, totWidth) ;
        }


    }
```

Figure 10 Used operator overloading

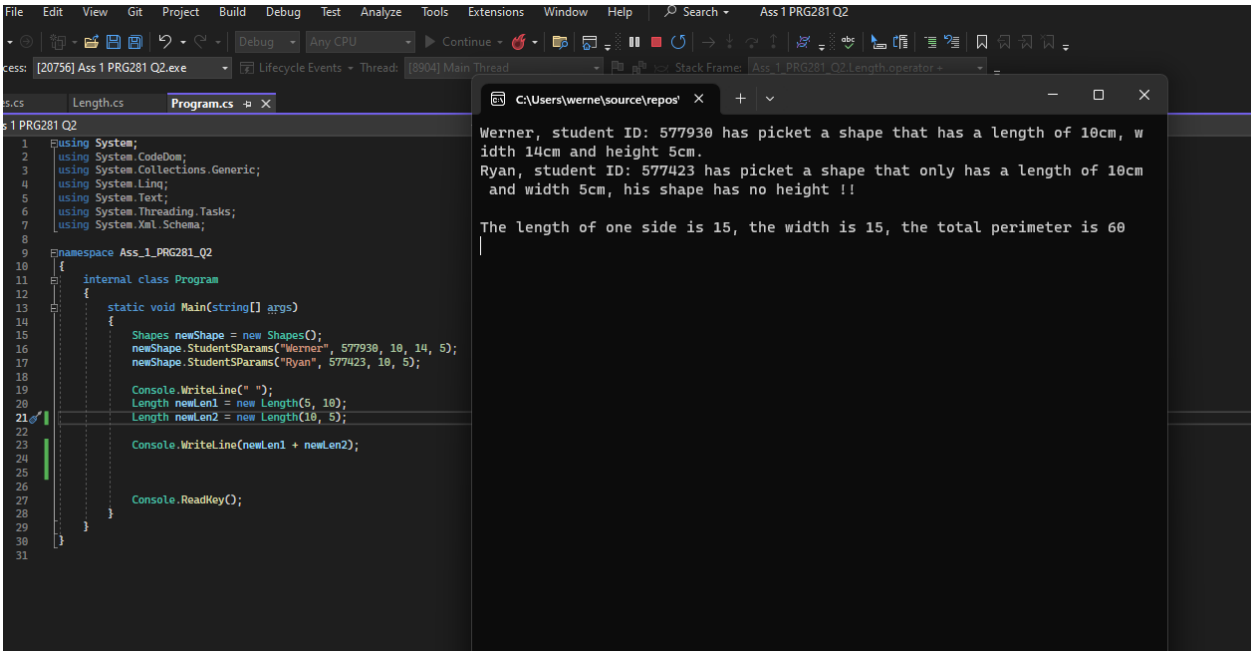

Figure 11 Display total perimeter

In the above example we want to get the total circumference of to shape objects created. Logically you want to add the circumference of object 1 and add it to object 2. As seen in figure 8, it is not possible to add objects together in main by just simply adding them together as if they are normal integers. That is why a Length operator is created. The parameters of the operator is obj 1 and object 2. There is a "+" sign in front of the brackets as well, as seen in figure 10. This whole operator, that performs like a function, basically replaces the whole expression in main when two Length objects need to be added together. So now when objects *newLen1* and *newLen2* needs to be added together, the Length operator is called which overloads the normal "+" expression and returns a new object, which would be the total circumference of the two shapes created. The return value then replaces the Perimeter variable in the *ToString()* method.

Dynamic Polymorphism:

Virtual/ Overriding Method:

According to (Trivedi, 2023) method overriding can be done using inheritance. This means the class which has a method and the class inheriting the method would use the same name for the method.

```
using System;
using System.CodeDom;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Xml.Schema;

namespace Ass_1_PRG281_Q2
{
    public class MyClass
    {
        public virtual string PrintName()
        {
            return " ";
        }

    }
    public class TestClass : MyClass
    {
        private string myName;

        public TestClass(string myName)
        {
            this.MyName = myName;
        }

        public string MyName { get => myName; set => myName = value; }

        public override string PrintName()
        {
            return $"{MyName}";
        }
    }
    internal class Program
    {
        static void Main(string[] args)
        {

            TestClass newTest = new TestClass("Werner");
            Console.WriteLine(newTest.PrintName());


            Console.ReadKey();
```

```
C:\Users\werne\source'
Werner
```

Figure 12 Overriding method

Above the *MyClass* that contains a virtual method called *PrintName().* The *TestClass* inherits from this class meaning that it inherits the method as well. The method then gets overridden and returns something new as stated by the *TestClass*. The method is therefore a form of polymorphism due to the fact that they look the same but do different things.

References

Taher, R., 2019. *Hands-On Object-Oriented Programming with C#.* Birmingham: Packt Publishing.

Trivedi, J., 2023. *C# Corner.* [Online]
Available at: https://www.c-sharpcorner.com/UploadFile/ff2f08/understanding-polymorphism-in-C-Sharp/
[Accessed 17 August 2023].

Turner, R., 2019. *C#: The Ultimate Beginner's Guide to Learn C# Programming Step by Step.* s.l.:Publishing Factory.