ASSESSMENT - 3

CS 202 Software Tools and Techniques for CSE

Pulakurthi Manohar (23110259)
Sowpati Raj Kamal (23110319)

LAB-9

Github link: https://github.com/Rajkamal065/stt_lab_9_10

Introduction:

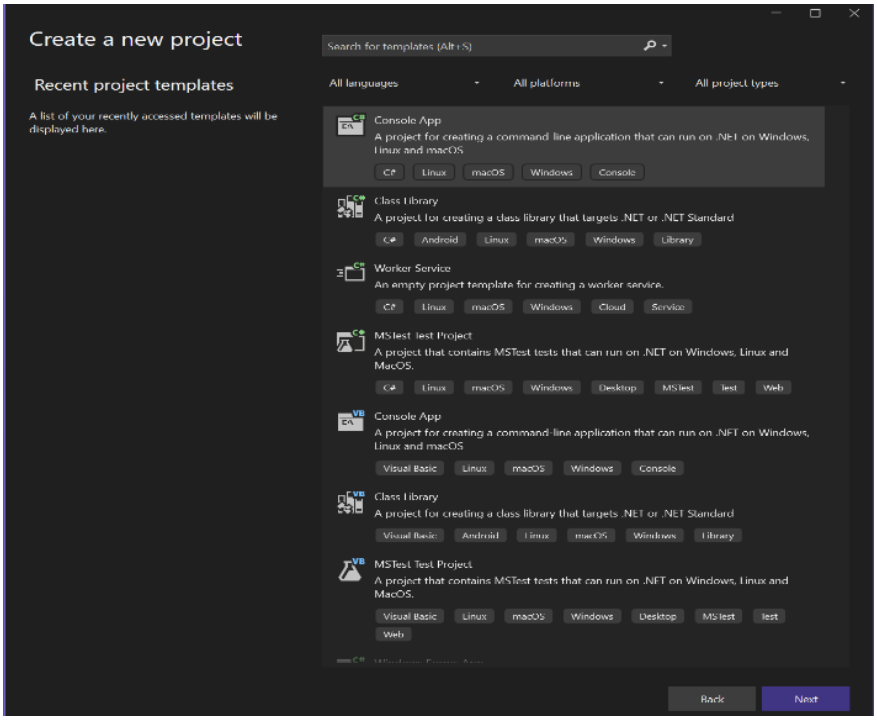
The lab session 9 focused on introducing students to the fundamentals of .NET development using the C# programming language in Visual Studio. In this session, we learned how to create simple C# console applications to understand the basic syntax, control structures, and object-oriented programming principles. We also explored the use of Visual Studio IDE for the first time and understood how .NET provides the runtime and framework support for C# programs. The exercises in this lab helped in building a foundation for developing structured and object-oriented applications using C#.

Tools:

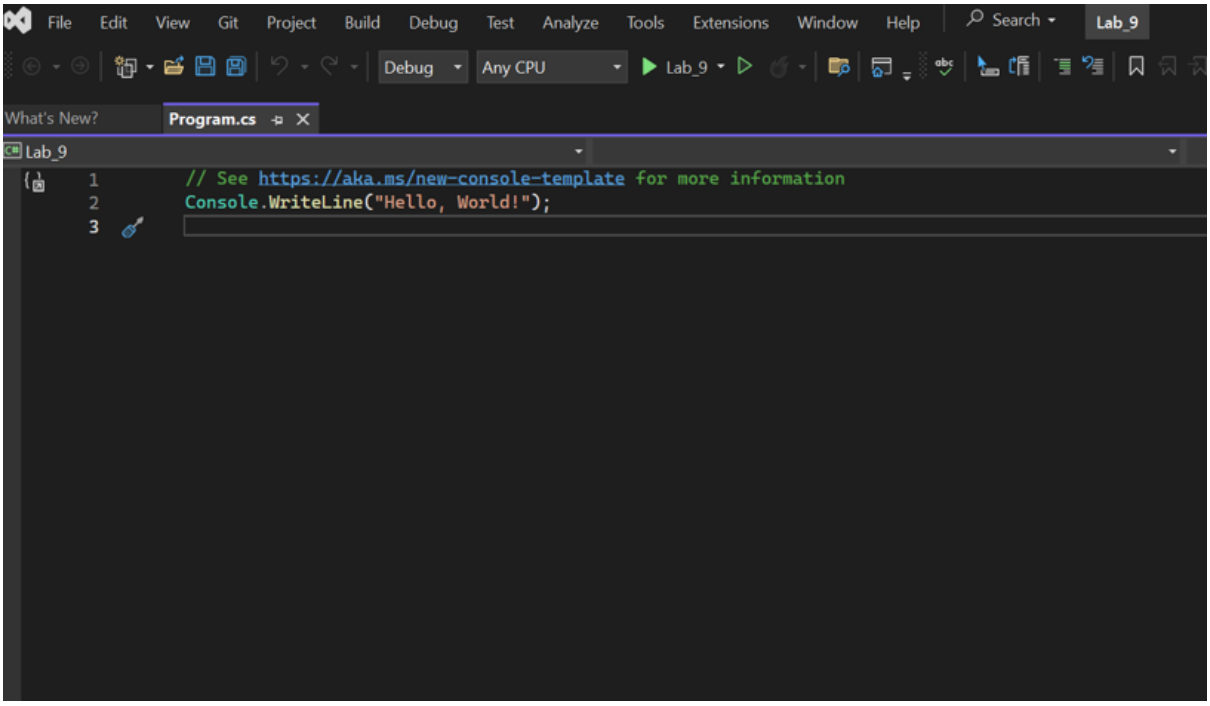
For this lab, I worked on the Windows operating system on my personal laptop. I installed Visual Studio Community Edition 2022, which comes with the .NET SDK required for C# development. All programs were implemented and executed within Visual Studio using the C# console application template targeting the .NET Framework.

Setup:

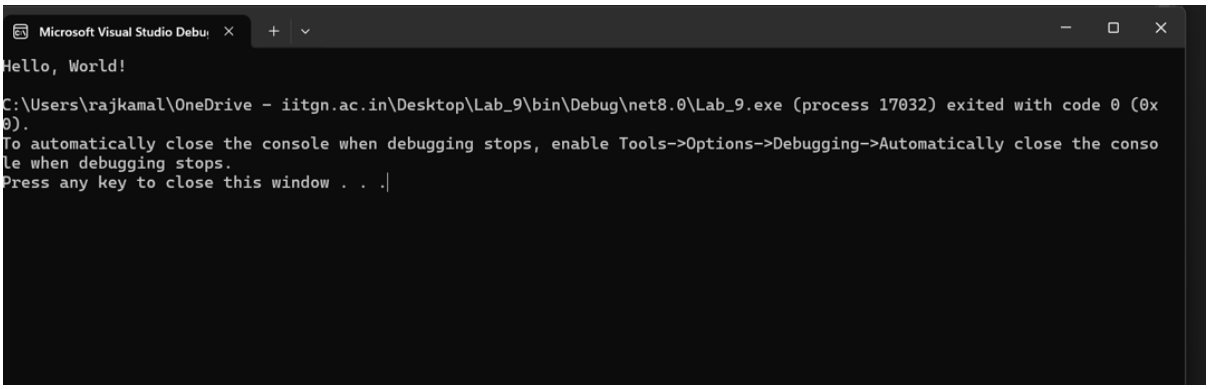
For this lab, I first installed Visual Studio Community Edition 2022 on my Windows laptop. During the installation, I selected the .NET desktop development workload to ensure compatibility with C# and .NET frameworks. After the installation was complete, I launched Visual Studio and created a new project by selecting the Console App template under C# development.



While creating the project, I selected .NET 8.0 as the target framework to make use of the latest features and performance improvements in the .NET ecosystem. Visual Studio automatically generated the initial Program.cs file containing the default Main() method.
Once the environment was ready, I successfully built and executed a simple “Hello World” program to verify that my setup was working correctly.



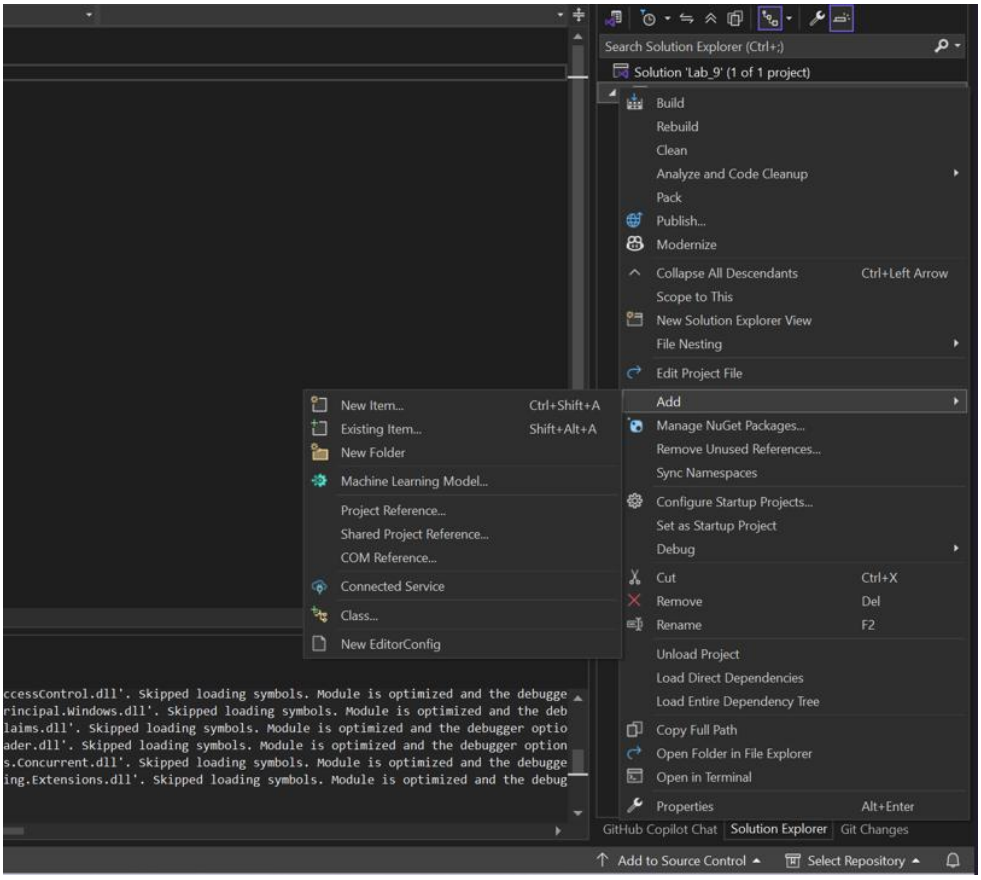
The output I got from running the above is:

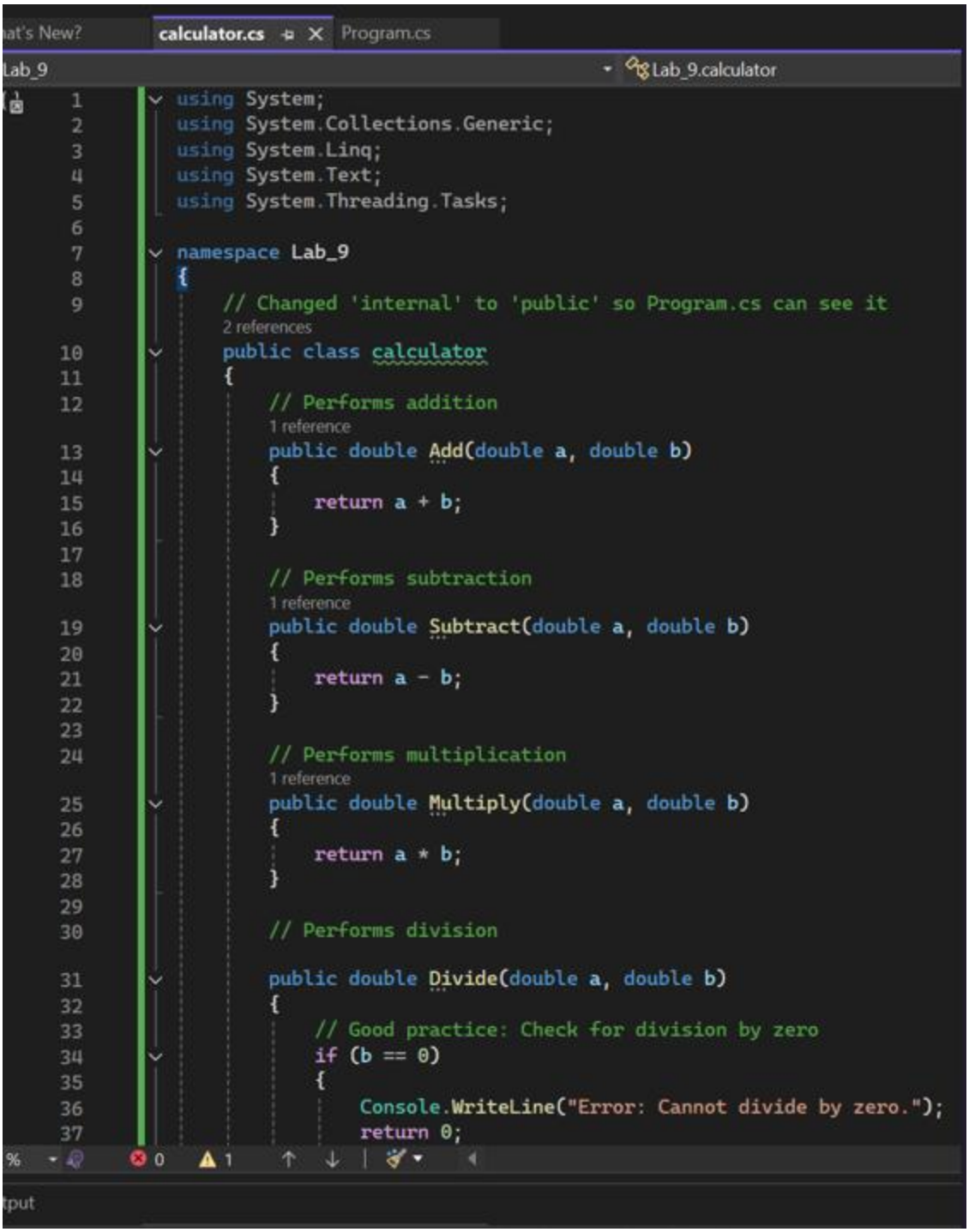


Methodology and Execution of the Lab:

A) Understanding Basic Syntax and Control Structures:

In this section of the lab, we were told to create a simple object-oriented C# program that takes two numbers as user inputs and performs addition, subtraction, multiplication, and division. For this, what I did was first I created a class Calculator by clicking on Solution Explorer and creating a file called Calculator.cs, and then I wrote the code needed for the operations they asked for.





Then, in the Program.cs file, I checked whether the sum of the numbers is even or odd by using an if-else condition as mentioned in the lab assignment.

```
using System;

namespace Lab_9
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            calculator myCalculator = new calculator();

            Console.WriteLine("Enter the first number:");
            double num1 = double.Parse(Console.ReadLine());

            Console.WriteLine("Enter the second number:");
            double num2 = double.Parse(Console.ReadLine());

            Console.WriteLine("--------------------");

            double sum = myCalculator.Add(num1, num2);
            double difference = myCalculator.Subtract(num1, num2);
            double product = myCalculator.Multiply(num1, num2);
            double quotient = myCalculator.Divide(num1, num2);

            Console.WriteLine($"Addition:       {num1} + {num2} = {sum}");
            Console.WriteLine($"Subtraction:    {num1} - {num2} = {difference}");
            Console.WriteLine($"Multiplication: {num1} * {num2} = {product}");
            Console.WriteLine($"Division:       {num1} / {num2} = {quotient}");

            if (sum % 2 == 0)
            {
                Console.WriteLine($"The sum ({sum}) is an EVEN number.");
            }
            else
            {
                Console.WriteLine($"The sum ({sum}) is an ODD number.");
            }
        }
    }
}
```

At last, for printing the output, I used Console.WriteLine(). The output I got is shown in the Result and Analysis section.

B) Using Loops and Functions:

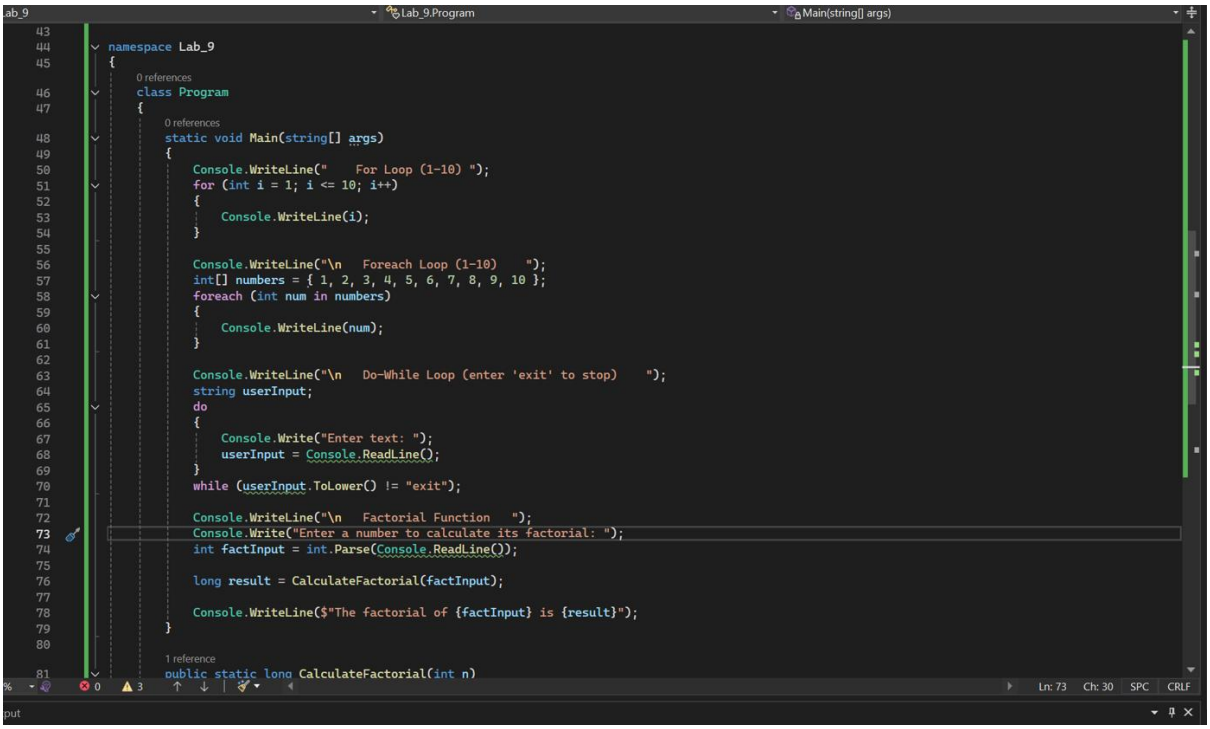
In this part of the lab, we were told to create an object-oriented program that uses a for loop to print numbers from 1 to 10, a foreach loop to print numbers from 1 to 10, and a do-while loop that keeps taking user input until the user types exit. Since these are small operations or loops, I directly wrote the code for these three loops in the Program.cs file.

```
Lab_9.Program

namespace Lab_9
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            Console.WriteLine("    For Loop (1-10) ");
            for (int i = 1; i <= 10; i++)
            {
                Console.WriteLine(i);
            }

            Console.WriteLine("\n   Foreach Loop (1-10)    ");
            int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
            foreach (int num in numbers)
            {
                Console.WriteLine(num);
            }

            Console.WriteLine("\n   Do-While Loop (enter 'exit' to stop)    ");
            string userInput;
            do
            {
                Console.Write("Enter text: ");
                userInput = Console.ReadLine();
            }
            while (userInput.ToLower() != "exit");
```

We were also asked to write a function that calculates the factorial of a number provided by the user. First, I wrote a function called CalculateFactorial that takes an input n and gives the factorial output.

```
        1 reference
        public static long CalculateFactorial(int n)
        {
            if (n == 0)
            {
                return 1;
            }

            long result = 1;
            for (int i = n; i >= 1; i--)
            {
                result = result * i;
            }
            return result;
        }
    }
}
```

Using this function, I then wrote the rest of the code. This entire section was written in the Program.cs file of the lab_9 project.



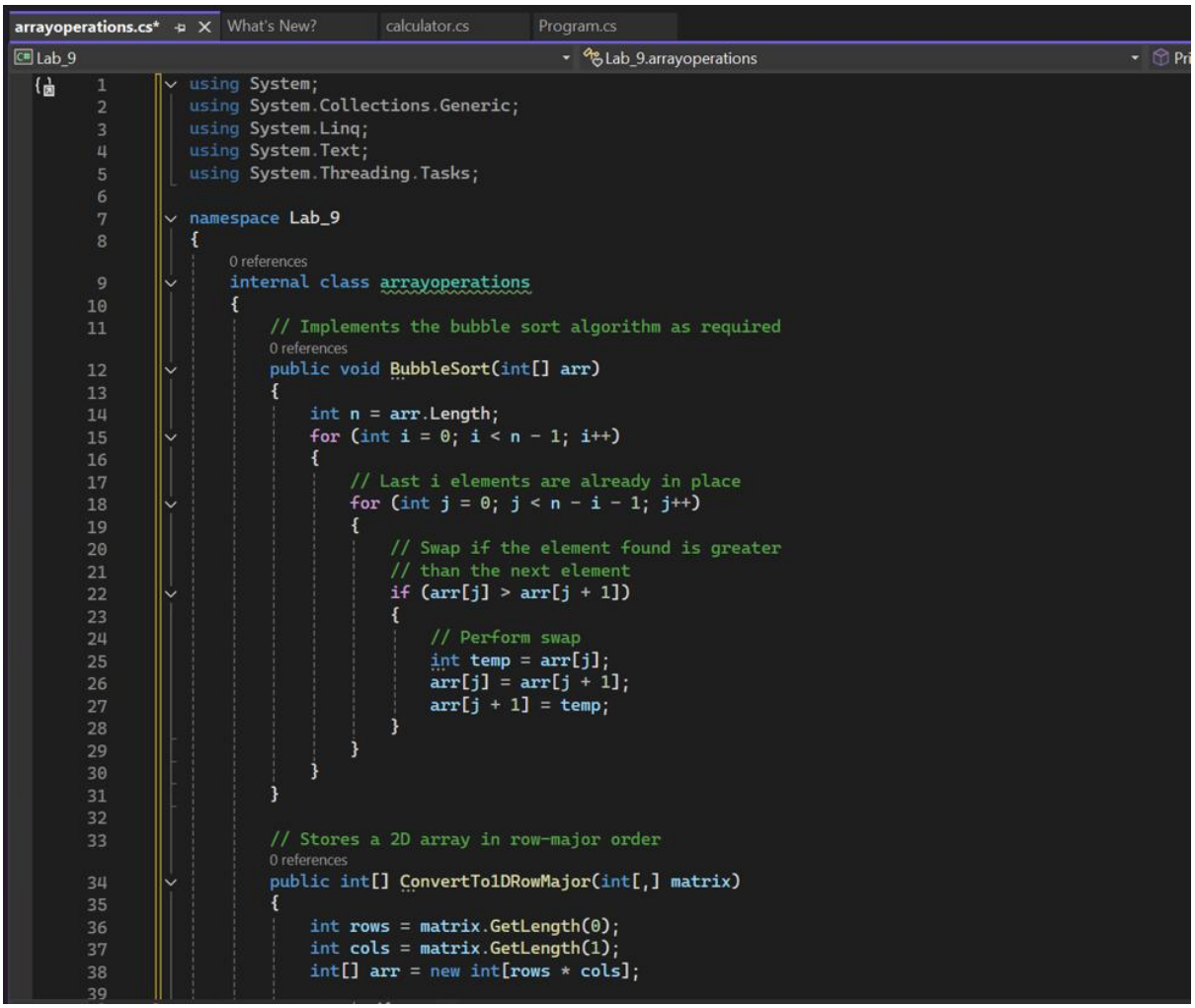
```
lab_9                                   Lab_9.Program                          Main(string[] args)
43
44      namespace Lab_9
45      {
            0 references
46          class Program
47          {
                0 references
48              static void Main(string[] args)
49              {
50                  Console.WriteLine("    For Loop (1-10) ");
51                  for (int i = 1; i <= 10; i++)
52                  {
53                      Console.WriteLine(i);
54                  }
55
56                  Console.WriteLine("\n   Foreach Loop (1-10)    ");
57                  int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
58                  foreach (int num in numbers)
59                  {
60                      Console.WriteLine(num);
61                  }
62
63                  Console.WriteLine("\n   Do-While Loop (enter 'exit' to stop)    ");
64                  string userInput;
65                  do
66                  {
67                      Console.Write("Enter text: ");
68                      userInput = Console.ReadLine();
69                  }
70                  while (userInput.ToLower() != "exit");
71
72                  Console.WriteLine("\n   Factorial Function   ");
73                  Console.Write("Enter a number to calculate its factorial: ");
74                  int factInput = int.Parse(Console.ReadLine());
75
76                  long result = CalculateFactorial(factInput);
77
78                  Console.WriteLine($"The factorial of {factInput} is {result}");
79              }
80
                1 reference
81              public static long CalculateFactorial(int n)
% 0 3                                                                  Ln: 73  Ch: 30  SPC  CRLF
put
```

The output of the code is shown in the Result and Analysis section.

C) Using Single and Multi-dimensional Arrays:

In this section of the lab, we were told to create an object-oriented program that implements the bubble sort algorithm on an array of integers and also performs array operations like matrix multiplication and storing a 2D array into a 1D array.

Just like in the first section, I added a class called ArrayOperations, where I defined every operation they asked for—from bubble sort to matrix multiplication. Then, in the Program.cs file, I used this class and called its methods to demonstrate each operation.


arrayoperations.cs*
What's New?
calculator.cs
Program.cs
Lab_9
Lab_9.arrayoperations
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
namespace Lab_9
{
0 references
internal class arrayoperations
{
// Implements the bubble sort algorithm as required
0 references
public void BubbleSort(int[] arr)
{
int n = arr.Length;
for (int i = 0; i < n - 1; i++)
{
// Last i elements are already in place
for (int j = 0; j < n - i - 1; j++)
{
// Swap if the element found is greater
// than the next element
if (arr[j] > arr[j + 1])
{
// Perform swap
int temp = arr[j];
arr[j] = arr[j + 1];
arr[j + 1] = temp;
}
}
}
}
// Stores a 2D array in row-major order
0 references
public int[] ConvertTo1DRowMajor(int[,] matrix)
{
int rows = matrix.GetLength(0);
int cols = matrix.GetLength(1);
int[] arr = new int[rows * cols];

```
using System;

namespace Lab_9
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            // Create an instance of our arrayoperations class
            // Note the lowercase 'a' matches your class file
            arrayoperations operations = new arrayoperations();

            // --- 1. Bubble Sort ---
            Console.WriteLine("--- Bubble Sort ---");
            int[] unsortedArray = { 64, 34, 25, 12, 22, 11, 90 };
            Console.WriteLine("Original array:");
            // Call the static helper method using the lowercase class name
            arrayoperations.PrintArray(unsortedArray);

            operations.BubbleSort(unsortedArray); // Sorts the array in place

            Console.WriteLine("Sorted array:");
            arrayoperations.PrintArray(unsortedArray);

            Console.WriteLine("\n--- 2D to 1D Conversion ---");
            int[,] matrix = {
                { 1, 2, 3 },
                { 4, 5, 6 }
            };
            Console.WriteLine("Original 2D Matrix:");
```

No issues found

Finally, using Console.WriteLine(), I printed all the results. The outputs are shown in the Result and Analysis section.

Result and Analysis:

A) Understanding Basic Syntax and Control Structures:

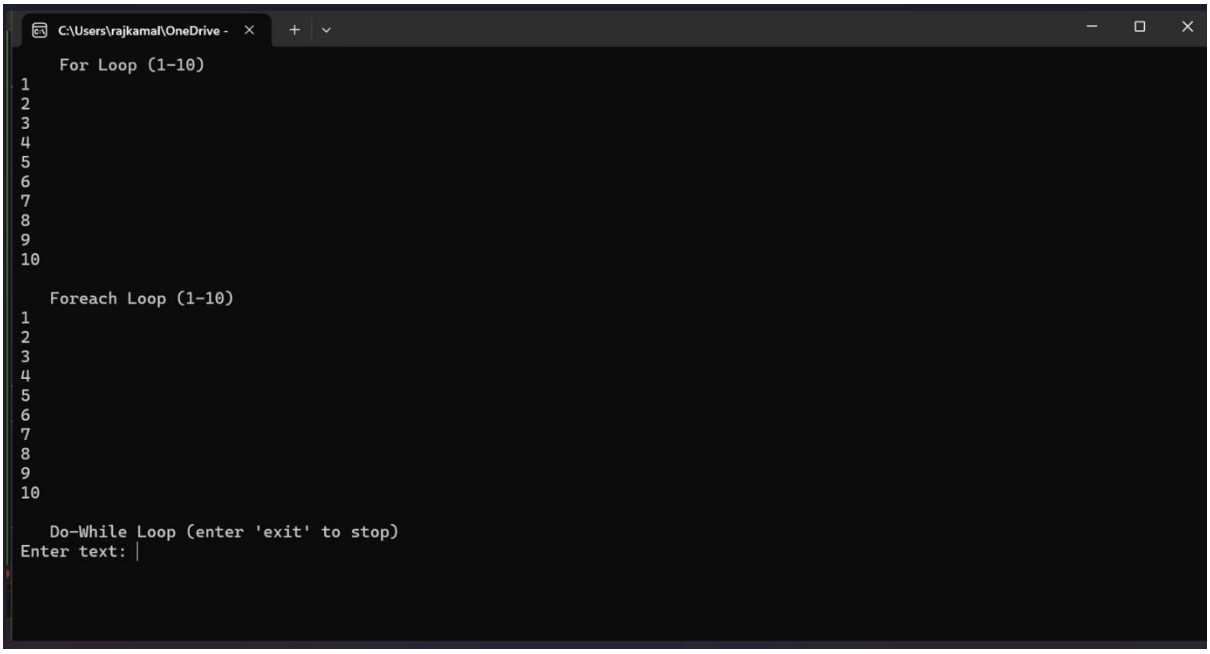
```
Program.cs                                                    Lab_9
Lab_9.Program                                  Main(string[] args)
{
    calculator myCalculator = new calculator();

Microsoft Visual Studio Debu
Enter the first number:
8
Enter the second number:
3
--------------------
Addition:       8 + 3 = 11
Subtraction:    8 - 3 = 5
Multiplication: 8 * 3 = 24
Division:       8 / 3 = 2.6666666666666665
The sum (11) is an ODD number.

C:\Users\rajkamal\OneDrive - iitgn.ac.in\Desktop\Lab_9\bin\Debug\net8.0\Lab_9.exe (process 9968) exited with code 0 (0x0
).
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the conso
le when debugging stops.
Press any key to close this window . . .
```

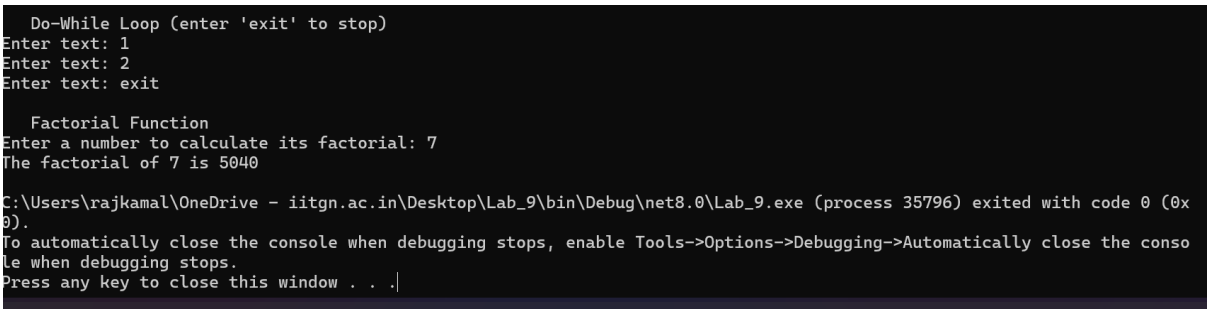
The code worked perfectly, and the output (using inputs 8 and 3) shows all the lab requirements were met:

- OOP: My Program.cs successfully created a calculator object and called its methods (Add, Subtract, Multiply, Divide), proving the object-oriented part worked.

- Math Operations: All calculations were correct. Using the double data type was a good choice, as it handled division correctly (8 / 3 = 2.666...).
- If-Else Condition: The program correctly checked the sum (11) and used the if-else block to print “The sum (11) is an ODD number.”
- Displaying Results: All outputs were printed clearly using Console.WriteLine(), as required.

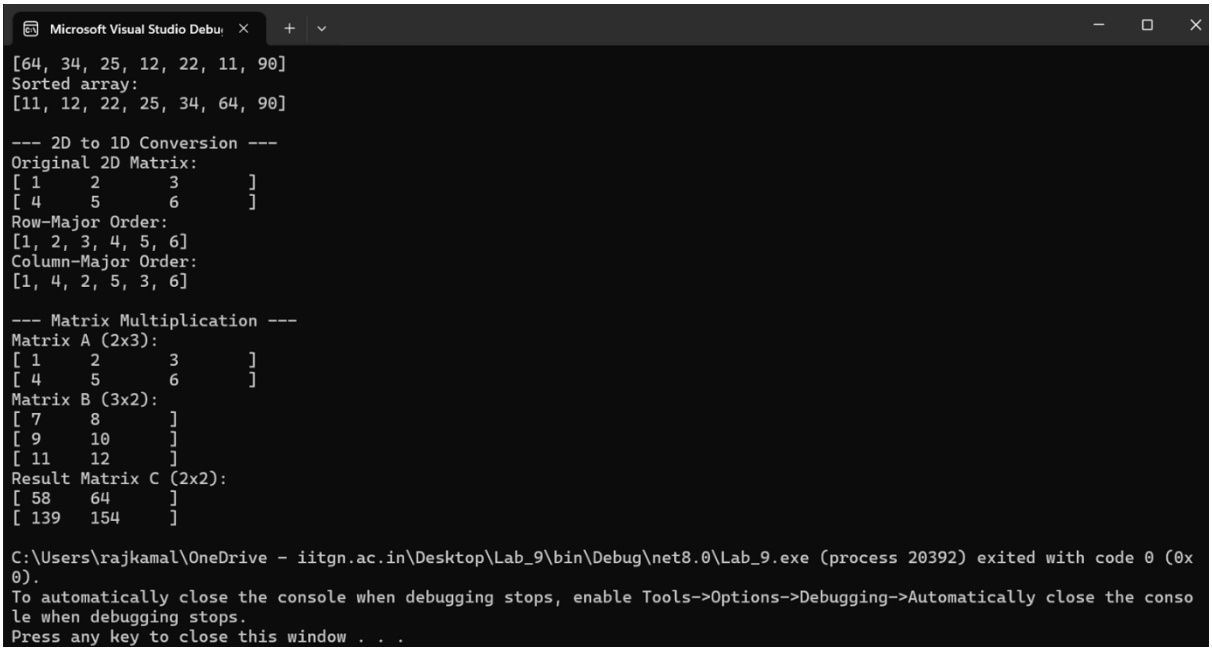
B) Using Loops and Functions:





- Both the for and foreach loops worked, printing 1 to 10 as required.
- The do-while loop correctly kept asking for input until “exit” was typed.
- The CalculateFactorial function also worked perfectly. Making it static allowed Main() to call it, and using long prevented overflow errors for large factorials.

C) Using Arrays and Matrix Operations:



This part also worked perfectly. The output from the program confirmed that all three array tasks were completed successfully.

- OOP: Using a separate ArrayOperations class was a good decision. My Program.cs just had to create an object and call the methods, keeping it clean.
- Bubble Sort: The BubbleSort function worked correctly. I implemented the sorting logic manually (using two nested loops) without any built-in library functions.
- 2D-to-1D Conversion: The conversion from the 2D matrix into a 1D array worked for both row-major and column-major order.
- Matrix Multiplication: The MatrixMultiplication function, though the most complex, also worked as expected. It used the standard three-loop logic to multiply matrices A and B and store the result in C.

D) Output Reasoning (Level 0):

Q1)

```
166     using System; //namespace
        0 references
167     class Program //default visibility4 is 'internal' if not specified
168     {
            0 references
169         public static void Main(string[] args)
170         {
171             int a = 0; //default visibility is 'private' (in a class)
172             Console.WriteLine(a++);
173         }
174     }
```

Output:

```
Microsoft Visual Studio Debu
0

C:\Users\rajkamal\OneDrive - iitgn.ac.in\Desktop\Lab_9\bin\Debug\net8.0\Lab_9.exe (process 21824) exited with code 0 (0x
0).
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the conso
le when debugging stops.
Press any key to close this window . . .
```

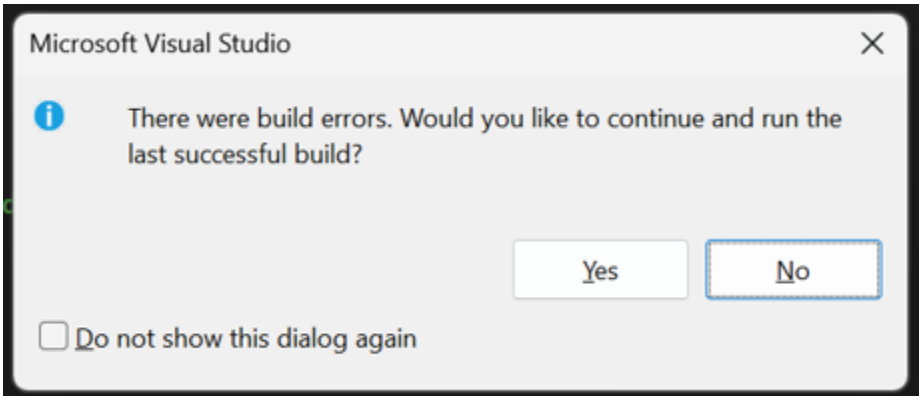
0

Reasoning: This code uses the postfix increment operator (a++). It returns the current value of a (which is 0) to the Console.WriteLine() method and then increments a to 1 afterward. Hence, it prints 0.

Q2)

```
using System;
0 references
class Program
{
    0 references
    public void Main(string[] args)
    {
        int a = 0;
        Console.WriteLine(a++);
    }
}
```

Output:



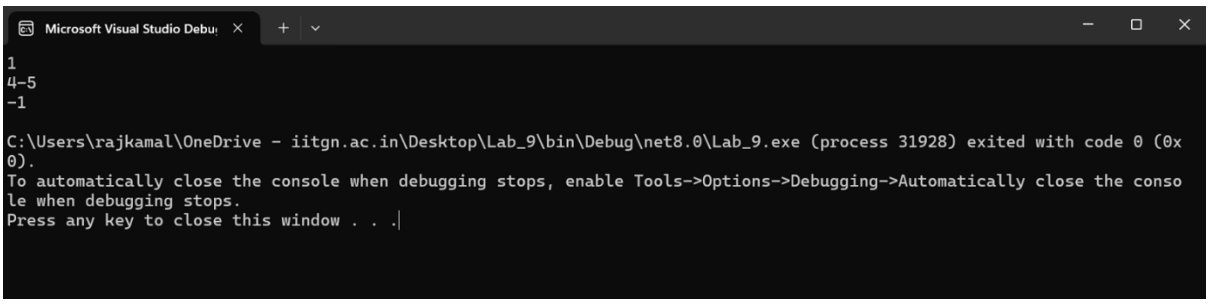
Compile-time error.
Reasoning: C# console applications require a static Main() method as the entry point. Since this code defines Main() without static, the compiler cannot find a valid entry point and throws an error.

E) Output Reasoning (Level 1):

Q1)

```
0 references
class Program
{
    0 references
    public static void Main(string[] args)
    {
        int a = 0;
        int b = a++;
        Console.WriteLine(a++.ToString(), ++a, -a++);
        Console.WriteLine((a++).ToString() + (-a++).ToString());
        Console.WriteLine(~b);
    }
}
```

Output:



1

4-5
-1

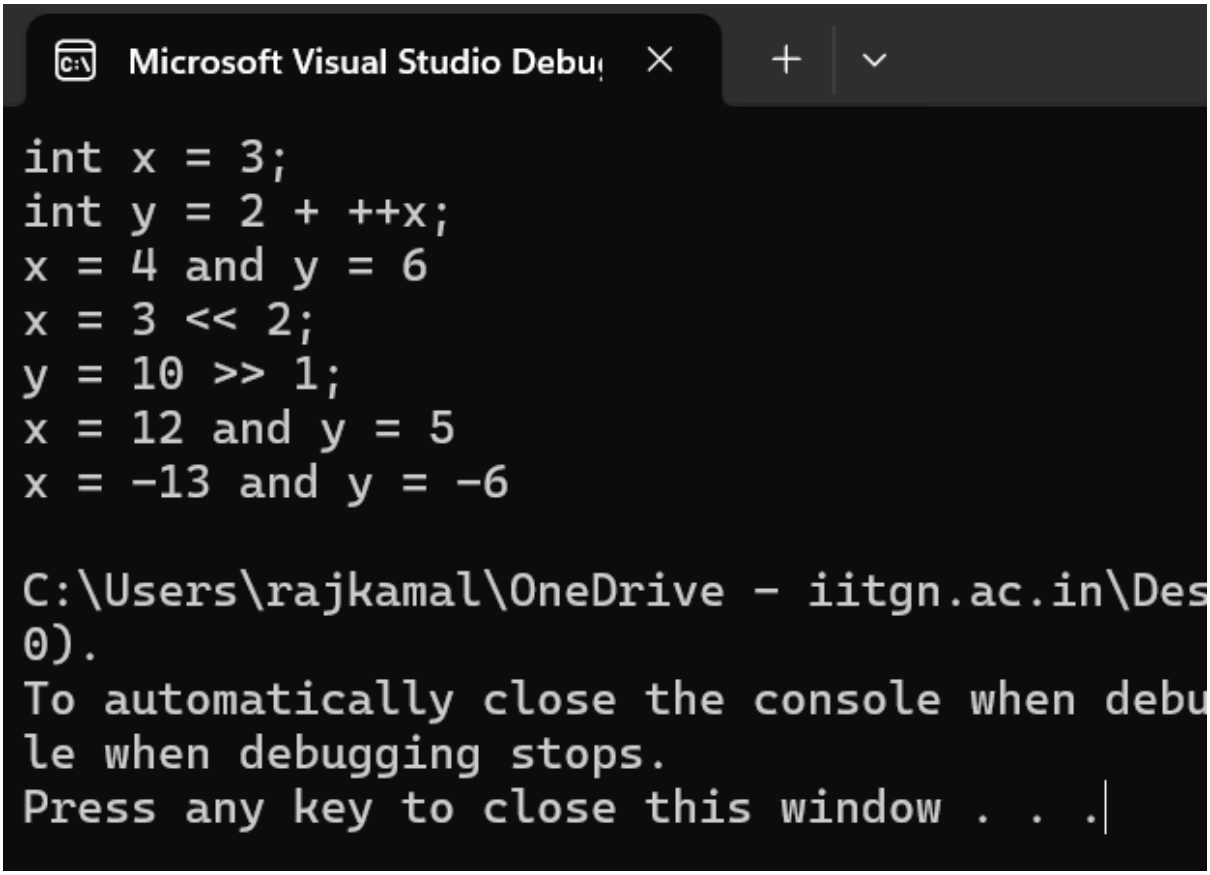
Reasoning:

- int b = a++; sets b = 0 and increments a to 1.
- The first Console.WriteLine(a++.ToString(), ++a, -a++); prints 1 since the other arguments are ignored (no placeholders in format string).
- The second Console.WriteLine((a++).ToString() + (-a++).ToString()); prints 4-5.
- The third Console.WriteLine(~b); prints -1 since ~0 = -1.

Q2)

```
using System;
/*you can also write top level code outside of a class. C# takes
care of this by providing internal entry point Main*/
Console.WriteLine("int x = 3;");
Console.WriteLine("int y = 2 + ++x;");
int x = 3; //default visibility is 'internal' (outside a class)
int y = 2 + ++x;
Console.WriteLine($"x = {x} and y = {y}");
Console.WriteLine("x = 3 << 2;");
Console.WriteLine("y = 10 >> 1;");
x = 3 << 2;
y = 10 >> 1;
Console.WriteLine($"x = {x} and y = {y}");
x = ~x;
y = ~y;
Console.WriteLine($"x = {x} and y = {y}");
```

Output:



int x = 3;

int y = 2 + ++x;

x = 4 and y = 6

x = 3 << 2;

y = 10 >> 1;

x = 12 and y = 5

x = -13 and y = -6

Reasoning:

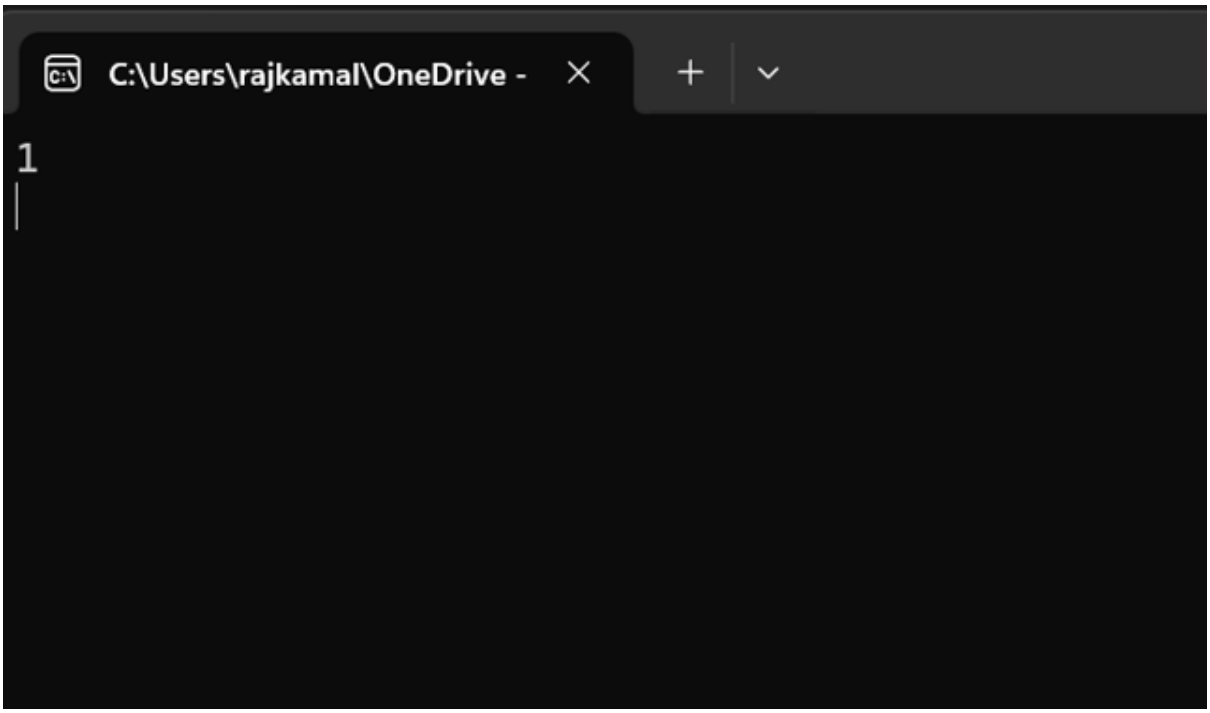
- ++x increments x before use, so y = 2 + 4 = 6.
- 3 << 2 shifts bits left → 12; 10 >> 1 shifts bits right → 5.
- Bitwise NOT (~) inverts bits: ~12 = -13, ~5 = -6.

F) Output Reasoning (Level 2):

Q1)

```
Output Reasoning(Level 2)

qn-1*/
using System;
0 references
public class Program
{
    0 references
    static void Main()
    {
        try
        {
            int i = int.MaxValue;
            Console.WriteLine(-(i + 1) - i);
            for (i = 0; i <= int.MaxValue; i++) ; //note semicolon here
            Console.WriteLine("Program ended!");
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.ToString());
        }
    }
}
```

Output:



Prints 1, then enters an infinite loop.

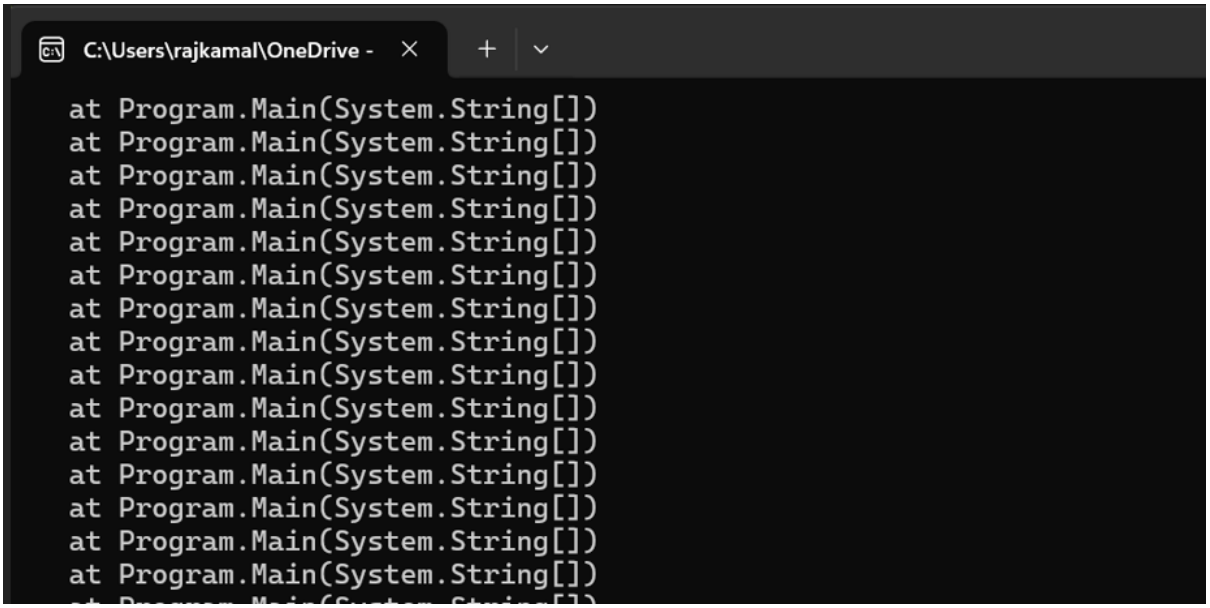
Reasoning:

- The WriteLine: The expression -(i+1)-i is evaluated with i = int.MaxValue. In C#, integer arithmetic overflows and "wraps around" by default. The (i+1) operation wraps to int.MinValue. The subsequent negations and subtractions also overflow, with the final computed result being 1.
- The for Loop: The loop for(i=0; i<=int.MaxValue;i++) is an infinite loop. i is reset to 0. When i eventually increments to int.MaxValue, the condition i <= int.MaxValue is true. The i++ then executes, causing i to overflow and wrap around to int.MinValue. The condition is checked again, and int.MinValue is still less than int.MaxValue. The loop's condition will never be false.
- Final Result: The program prints 1, then enters the infinite loop. The Console.WriteLine("Program ended!"); is never reached. Since default integer overflow doesn't throw an error, the catch block is also never executed.

Q2)

```
using System;
0 references
public class Program
{
    1 reference
    static void Main(string[] args)
    {
        Main(["CS202"]);
    }
}
```

Output:



Program crashes with StackOverflowException.

Reasoning:

- The Main method contains a call to itself: Main(["CS202"]);. This is known as recursion.
- The problem is that there is no "base case" or if statement to tell the recursion when to stop.
- This creates an infinite loop: Main calls Main, which calls Main, which calls Main, and so on.
- Each time a method is called, it's added to a memory area called the "call stack." Because the Main method is called infinitely, the call stack eventually runs out of memory, causing the program to crash with a StackOverflowException

Discussion and Conclusion:

From this lab, what I've learned is how structured, logical, and well-organized C# programming is within the .NET environment. This lab was not just about writing simple code but about understanding how to design a program properly from creating classes to applying object-oriented principles effectively.

The most important takeaway for me was definitely the object-oriented part. At first, creating separate classes like Calculator or ArrayOperations seemed like unnecessary extra steps, but once I implemented them, I realized how much cleaner and more modular my code became.

The Output Reasoning tasks were quite challenging but extremely insightful. They forced me to deeply understand how operators work in C#, such as the difference between a++ and ++a, how integer overflow behaves (especially with int.MaxValue), and how the static keyword affects

program behavior. These exercises made me more careful and analytical while reading and writing code.

Overall, this lab provided a very strong foundation in C# and .NET development. From setting up Visual Studio and writing simple console programs to implementing loops, arrays, and object-oriented design, I gained both practical experience and conceptual clarity. I now feel confident not just in writing C# programs but also in understanding the reasoning behind their structure and execution

References:

- Lecture 9 slides
- https://learn.microsoft.com/en-us/dotnet/csharp/

Lab Assignment 10

INTRODUCTION

This lab was an introduction to some of the most important concepts in C# and object-oriented programming (OOP). The focus was on the "lifecycle" of an object—how it's born (constructors) and how it's cleaned up (destructors)—and on inheritance, which is how we create "is-a" relationships between classes.

The goal was to build two small console applications. The first one required me to manage objects and manually control their cleanup to observe this lifecycle. The second application involved building a hierarchy of Vehicle, Car, and Truck classes to see how a "child" class can inherit and change the behavior of its "parent" class, a concept known as polymorphism.

OBJECTIVES

1. Create and manage classes with encapsulation, constructors, and destructors.
2. Use inheritance effectively to create hierarchical relationships and method overriding.
3. Understand the difference between virtual and override methods.
4. Observing runtime polymorphism by using a base-class array to hold derived-class objects.
5. Learned about the C# Garbage Collector (GC) and how to interact with it.

ENVIRONMENT, VERSIONS, AND TOOLS USED

For this lab, I used the following environment and tools:

- Operating System: Windows 11
- Code Editor: Visual Studio Code
- SDK: .NET 8.0 SDK
- Language: C#
- Extension: C# Dev Kit (for VS Code)

SETUP

I started by setting up my environment. The first step was to install the .NET 8.0 SDK and Visual Studio Code, along with the C# Dev Kit extension, which provides all the necessary C# language support in VS Code.

A C# console project can only have one Main() method (entry point). Since the lab had two separate runnable activities, I couldn't put them in the same project. I solved this by creating two separate projects in my Desktop folder using the .NET CLI:

1. I opened my terminal, navigated to the Desktop, and ran: dotnet new console -o CS202_Lab10
2. Then, for the second activity, I did the same: dotnet new console -o CS202_Lab10_Activity2

This gave me two clean, separate project folders, which I could then open in VS Code to write my code.

METHODOLOGY AND EXECUTION

Activity 1: Constructors and Data Control

The first task was to create a Program class to demonstrate the object lifecycle. I implemented this class with a constructor, a destructor (~Program), a private data field, set_data/show_data methods, and a static counter to track the number of active objects. The constructor incremented this counter, and the destructor decremented it.

The main challenge was ensuring the "Object Destroyed" messages would appear, as C#'s Garbage Collector (GC) runs automatically. To solve this, I structured my code to control when the objects were no longer in use.

1. I created a separate helper method, static void CreateObjects().
2. Inside this CreateObjects() method, I placed all the logic for creating the three Program objects, setting their data, and showing their data.
3. In the Main method, I first called CreateObjects(). As soon as this method finished executing, the three objects it created (obj1, obj2, obj3) went out of scope, as they were local to that method.
4. This meant the objects were now "garbage" and eligible for collection.
5. Immediately after the CreateObjects() call returned, I placed the lines GC.Collect() and GC.WaitForPendingFinalizers() in the Main method. This explicitly told the GC to run a collection and to wait for all destructor messages to be printed before the program continued.

This approach cleanly separated object creation from garbage collection, allowing me to observe the full object lifecycle reliably.

Analysis: The output clearly shows the expected object lifecycle.

1. First, the CreateObjects() method runs, calling the constructor three times and incrementing the static counter. It then prints the data.
2. Once that method finishes, the Main method resumes. The GC.Collect() call finds the three objects (which are now out of scope) and triggers their destructors.
3. The "Object Destroyed" messages are printed, and the static counter correctly decrements back to zero, confirming all objects were cleaned up.

```
PS C:\Users\manoh\OneDrive\Desktop\CS202_Lab10> dotnet run
Constructor Called.
Number of active objects: 1
Constructor Called.
Number of active objects: 2
Constructor Called.
Number of active objects: 3
Data value is: 10
Data value is: 20
Data value is: 30
Object Destroyed.
Number of active objects: 2
Object Destroyed.
Number of active objects: 1
Object Destroyed.
Number of active objects: 0
```

Activity 2: Inheritance and Method Overriding

This activity was more straightforward and demonstrated polymorphism. I created a base class Vehicle and two derived classes, Car and Truck.

1. **Base Class:** I first defined a Vehicle class. Inside this class, I included protected fields for speed and fuel. I then created Drive() and ShowInfo() methods, marking them with the virtual keyword. This keyword indicates that derived classes are permitted to provide their own implementations.
2. **Derived Classes:** Next, I created two new classes, Car and Truck, that inherited from Vehicle using the : Vehicle syntax.
    - Inside Car, I added a new passengers field and used the override keyword to provide a new version of Drive() and ShowInfo().
    - I did the same for the Truck class, adding a cargoWeight field and overriding the Drive() and ShowInfo() methods with truck-specific logic.

3. **Execution:** To test the polymorphism, the Main method in this project was key. I created an array of Vehicle references: Vehicle[] vehicles = new Vehicle[3];. Because Car and Truck *are* Vehicles, I was able to store one Vehicle, one Car, and one Truck object in this array. I then looped through the array and called the v.Drive() and v.ShowInfo() methods on each item.

This setup allowed me to observe runtime polymorphism, as the program would call the correct (most specific) version of the Drive() method at runtime, depending on the *actual* object type.

```
PS C:\Users\manoh\OneDrive\Desktop\CS202_Lab10_Activity2> dotnet run
Vehicle is moving...
Speed: 60 km/h, Fuel: 95%

Car is moving with 4 passengers.
Speed: 60 km/h, Fuel: 90%, Passengers: 4

Truck is moving with 500kg of cargo.
Speed: 60 km/h, Fuel: 85%, Cargo Weight: 500kg
```

Analysis: This output is a perfect demonstration of runtime polymorphism. Even though the foreach loop was iterating over a Vehicle array (e.g., Vehicle v in vehicles), the .NET runtime correctly identified the *actual* object's type at runtime. It executed the Vehicle version for the Vehicle object, the Car's overridden version for the Car object, and the Truck's overridden version for the Truck object, all from the same v.Drive() method call.

DISCUSSION AND CONCLUSION

This lab was a very effective and practical exploration of two fundamental pillars of C#.

The second activity on inheritance was a clear demonstration of how polymorphism makes code cleaner and more extensible. I learned how virtual and override work together to allow a base class to define a "contract" that child classes can then specialize.

The first activity on the object lifecycle was the most challenging and insightful. I learned that C# memory management is very different from a language like C++. The non-deterministic nature of the Garbage Collector was a new concept. My biggest takeaway from the debugging process was the **critical difference between Debug and Release builds**. I spent a significant amount of time troubleshooting why my objects weren't being collected, only to learn that the Debug build was intentionally keeping them alive. This taught me that the compilation environment is not a neutral factor but an active part of the program's behavior.

In the end, I now have a much stronger understanding of how C# objects are created, managed, and destroyed, and how to build flexible class hierarchies using inheritance.

Output Reasoning (Level 0)

1.

```
using System;
int a = 3;
int b = a++;
Console.WriteLine($"a is {+a++}, b is {-++b}");
int c = 3;
int d = ++c;
Console.WriteLine($"c is {-c--}, d is {~d}");
```

Reasoning:

1. Initially a = 3, b = a++ → b = 3, then a = 4.
2. +a++ → current a = 4, print +4, then a = 5.
3. -++b → pre-increment b → b = 4, then -b → -4.
   Prints: a is 4, b is -4.

Now second part:

1. c = 3, d = ++c → c = 4, d = 4.
2. -c-- → print -4, then c = 3.
3. ~d → bitwise NOT of 4 → ~4 = -5 (since ~x = -(x+1)).
   Prints: c is -4, d is -5.

Output:

```
a is 4, b is -4
c is -4, d is -5
```

2.

```
using System;
2 references
class Program
{
    int age;
    1 reference
    Program() => age = age == 0 ? age + 1 : age - 1;
    0 references
    static void Main()
    {
        int k = "010%".Replace('0', '%').Length;
        Console.Write("[" + (k << ++new Program().age).ToString() + "]");
        Console.Write("[" + "010%".Split('1')[1][0] + "]");
        Console.Write("[" + "010%".Split('0')[1][0] + "]");
        Console.Write("[" + int.Parse(Convert.ToString("123".ToCharArray()[~-1])) + "]");
    }
}
```

Reasoning:

"010%".Replace('0','%') → "%%1%" → .Length = 4 → k = 4.

new Program() calls constructor:

- age default 0. Constructor: age = (age == 0) ? age + 1 : age - 1 → age = 1.

++new Program().age increments that age before use → age becomes 2. So expression is k << 2 → 4 << 2 = 16. Prints [16].

"010%".Split('1') → ["0","0%"]. [1][0] is '0'. Prints [0].

"010%".Split('0') → ["", "1", "%"]. [1][0] is '1'. Prints [1].

"123".ToCharArray()[~-1] evaluate ~-1:

- ~ is **bitwise NOT**. In C#, ~(-1) = 0. (Because ~x = -(x+1): ~(-1) = -(-1+1) = 0.)
- So index = 0. ToCharArray()[0] → '1'. Convert.ToString('1') → "1". int.Parse("1") → 1. Prints [1].

Output:

```
[16][0][1][1]
```

3.

```
using System;
0 references
class Program
{
    0 references
    static void Main()
    {
        int[] nums = { 0, 1, 0, 3, 12 };
        int pos = 0;
        for (int i = 0; i < nums.Length; i++)
        {
            if (nums[i] != 0)
            {
                int temp = nums[pos];
                nums[pos] = nums[i];
                nums[i] = temp;
                pos++;
            }
        }
        Console.WriteLine(string.Join(", ", nums));
    }
}
```

Reasoning:

- i=0: nums[0]=0 → skip.
- i=1: nums[1]=1 ≠0 → swap nums[1] with nums[pos=0]:
    - after swap: nums = [1,0,0,3,12], pos → 1
- i=2: nums[2]=0 → skip
- i=3: nums[3]=3 ≠0 → swap with nums[pos=1]:
    - after swap: nums = [1,3,0,0,12], pos → 2
- i=4: nums[4]=12 ≠0 → swap with nums[pos=2]:
    - after swap: nums = [1,3,12,0,0], pos → 3

Output:

```
1, 3, 12, 0, 0
```

Output Reasoning (Level 1):

1.

```
using System;
2 references
class Program
{
    int age;
    1 reference
    Program() => age=age==0?age+1:age-1;
0 references
static void Main()
    {
        int k = "010%".Replace('0', '%').Length;
        Console.Write("[" + (k << ++new Program().age).ToString() + "]");
        Console.Write("[" + "010%".Split('1')[1][0] + "]");
        Console.Write("[" + "010%".Split('0')[1][0] + "]");
        Console.Write("[" + int.Parse(Convert.ToString("123".ToCharArray()[~-1]))
    }
}
```

Output:

```
[16][0][1][1]
```

Reasoning:

"010%".Replace('0','%') → "%%1%" → .Length = 4 → k = 4.

new Program() calls constructor:

- age default 0. Constructor: age = (age == 0) ? age + 1 : age - 1 → age = 1.

++new Program().age increments that age before use → age becomes 2. So expression is k << 2 → 4 << 2 = 16. Prints [16].

"010%".Split('1') → ["0","0%"]. [1][0] is '0'. Prints [0].

"010%".Split('0') → ["", "1", "%"]. [1][0] is '1'. Prints [1].

"123".ToCharArray()[~-1] evaluate ~-1:

- ~ is **bitwise NOT**. In C#, ~(-1) = 0. (Because ~x = -(x+1): ~(-1) = -(-1+1) = 0.)
- So index = 0. ToCharArray()[0] → '1'. Convert.ToString('1') → "1". int.Parse("1") → 1. Prints [1].

2.

```
using System;
6 references
class Program
{
    int f;
    0 references
    public static void Main(string[] args)
    {
        Console.WriteLine("run 1");
        Program p = new Program(new int() + "0".Length);
        for (int i = 0, _ = i; i < 5 && ++p.f >= 0; i++, Console.WriteLine(p.f++))
        {
            for (; p.f == 0;);
            {
                Console.WriteLine(p.f);
            }
        }
        Console.WriteLine("\nrun 2");
        p = new Program(p.f);
        Console.WriteLine(p.f);
        Console.WriteLine("\nrun 3");
        p = new Program();
        Console.WriteLine(p.f);
    }
    1 reference
    Program() => f = 0;
    2 references
    Program(int x) => f = x;
}
```

Reasoning:

new int() → 0. "0".Length → 1. So constructor arg = 0 + 1 = 1. So Program(1) sets f = 1.

for (int i = 0, _ = i; i < 5 && ++p.f >= 0; i++, Console.WriteLine(p.f++));

- Loop condition uses ++p.f (pre-increment). Iterator prints p.f++ (post-increment).
- Start: i=0, f=1
    - Check cond: i<5 true AND ++p.f → f becomes 2 → 2 >= 0 true.
    - Body is empty; now iterator runs: i++ → i=1, then Console.WriteLine(p.f++) prints current f (2) then f→3.
- Next iteration:
    - i=1 < 5 AND ++p.f → f becomes 4 → true.
    - iterator prints 4 then f→5, i→2.

- Next:
    - i=2 AND ++p.f → f→6; iterator prints 6 then f→7, i→3.
- Next:
    - i=3 AND ++p.f → f→8; iterator prints 8 then f→9, i→4.
- Next:
    - i=4 AND ++p.f → f→10; iterator prints 10 then f→11, i→5.
- Next check: i=5 → i < 5 is false → loop ends.
- Printed sequence inside loop: 2 4 6 8 10. After loop f = 11.

The following block:

- for (; p.f == 0;); → condition false (f=11), so it does nothing.
- Next block Console.WriteLine(p.f); prints 11.

Run 2: p = new Program(p.f) → Program(11) sets f=11. Console.WriteLine(p.f) prints 11.

Run 3: p = new Program() → default constructor sets f=0. Prints 0.

Output:

```
run 1
2
4
6
8
10
11

run 2
11

run 3
0
```

3.

```
public class A
{
    5 references
    public virtual void f1()
    {
        Console.WriteLine("f1");
    }
}
2 references
public class B : A
{
    5 references
    public override void f1() => Console.WriteLine("f2");
}
3 references
class Program
{
    static int i = 0;
    public event funcPtr handler;
    public delegate void funcPtr();
    4 references
    public void destroy()
    {
        if (i == 6)
            return;
        else
        {
            Console.WriteLine(i++);
            destroy();
        }
    }
    0 references
    public static void Main(string[] args)
    {
        Program p = new Program();
        p.handler += new funcPtr((new A()).f1);
        p.handler += new funcPtr((new B()).f1);
        p.handler();
        p.handler -= new funcPtr((new B()).f1);
        p.handler -= new funcPtr((new A()).f1);
        p?.destroy(); //check here8 about ?. operator
        p = null;
        i = -6;
        p?.destroy();
        (new Program())?.destroy();
    }
}
```

Reasoning:

p.handler += A.f1 then += B.f1.

p.handler() invokes both delegates in registration order:

- Calls (new A()).f1() → prints f1
- Calls (new B()).f1() → prints f2

Removing both delegates leaves handler null.

p?.destroy() calls destroy() (p is not null):

- destroy() uses recursion printing i++ until i == 6.
- Starting static i = 0: prints 0,1,2,3,4,5 (stops when i becomes 6).

p = null; i = -6; p?.destroy(); → p is null so nothing happens.

(new Program())?.destroy() → calls destroy() on new object, but i is static and currently -6, so recursion prints -6,-5,-4,-3,-2,-1,0,1,2,3,4,5 and stops at i == 6.

Output:

```
f1
f2
0
1
2
3
4
5
-6
-5
-4
-3
-2
-1
0
1
2
3
4
5
```

Output Reasoning (Level 2)

1.

```
public class Institute
{
    internal int i = 7;
    1 reference
    public Institute()
    {
        Console.Write("1");
    }
    3 references
    public virtual void info()
    {
        Console.Write("2");
    }
}
5 references
public class IITGN : Institute
{
    public int i = 8;
    1 reference
    public IITGN()
    {
        Console.Write("3");
    }
    1 reference
    public IITGN(int i)
    {
        Console.Write("4");
    }
    3 references
    public override void info()
    {
        Console.Write("5");
    }
}
0 references
class Program
{
    0 references
    public static void Main(string[] args)
    {
        Console.Write("6");
        Institute ins1 = new Institute();
        ins1.info();
        IITGN ab101 = new IITGN(3);
        ab101 = new IITGN();
        ab101.info();
        Console.WriteLine();
        Console.WriteLine(ab101.i);
        Console.WriteLine(~(((Institute)ab101).i));
    }
}
```

Reasoning:

Prints "6".

new Institute() → constructor prints "1".

ins1.info() → prints "2".

new IITGN(3):

- Base constructor always runs → prints "1", then derived "4".

new IITGN() → base "1", derived "3".

ab101.info() → overridden → "5".
So far output: 6 1 2 1 4 1 3 5.

ab101.i = 8, ((Institute)ab101).i = 7.

- ~7 = -8

Output:

```
61214135
8
-8
```

2.

```
using System;
2 references
public class Program
{
    public delegate void mydel();
    2 references
    public void fun1()
    {
        Console.WriteLine("fun1()");
    }
    3 references
    public void fun2()
    {
        Console.WriteLine("fun2()");
    }
    0 references
    public static void Main(string[] args)
    {
        Program p = new Program();
        mydel obj1 = new mydel(p.fun1);
        obj1 += new mydel(p.fun2);
        Console.WriteLine("run 1");
        obj1();
        mydel obj2 = new mydel(p.fun2);
        obj2 += new mydel(p.fun1);
        Console.WriteLine("run 2");
        obj2();
        obj2 -= p.fun2;
        Console.WriteLine("run 3");
        obj2();
    }
}
```

Reasoning:

run 1:

- obj1 → fun1 + fun2 → prints:
- fun1()
- fun2()

run 2:

- obj2 → fun2 + fun1 → prints:
- fun2()
- fun1()

run 3:

- obj2 -= p.fun2 → now only fun1 remains → prints:

fun1()

Output:

```
run 1
fun1()
fun2()
run 2
fun2()
fun1()
run 3
fun1()
```

3.

```
using System;
using System.Collections;
7 references
public class Program
{
    int x;
    0 references
    public static void Main(string[] args)
    {
        ArrayList L = new ArrayList();
        L.Add(new Program());
        L.Add(new Program());
        for (int i = 0; i < L.Count; i++)
            Console.WriteLine(++((Program)L[i]).x);
        L[0] = L[1];
        ((Program)L[0]).x = 202;
        for (int i = 0; i < L.Count; i++)
            Console.WriteLine(((Program)L[i]).x);
        ((Program)L[0]).x = 111;
        L.RemoveAt(0);
        Console.WriteLine(L.Count);
        Console.WriteLine(((Program)L[0]).x);
    }
}
```

Reasoning:

Initially, both objects have x=0.

- First loop: ++x for both → prints:
- 1
- 1

L[0] = L[1] → both point to the same object.

- Set x = 202 for that shared object.

Second loop: both L[0] and L[1] refer to same → prints:

202

202

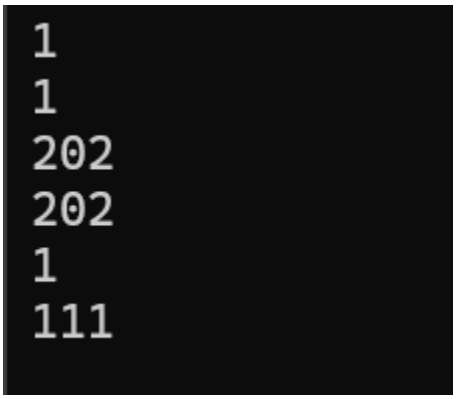
Change that object's x = 111.

- Remove first element → now only one object remains.
- L.Count = 1, x = 111.
  Prints:

1

111

Output:



Resources

• Constructors in C#

• Publisher-Subscriber Pattern

• Inheritance in C# and .NET

• Lecture 10

• Lecture 11