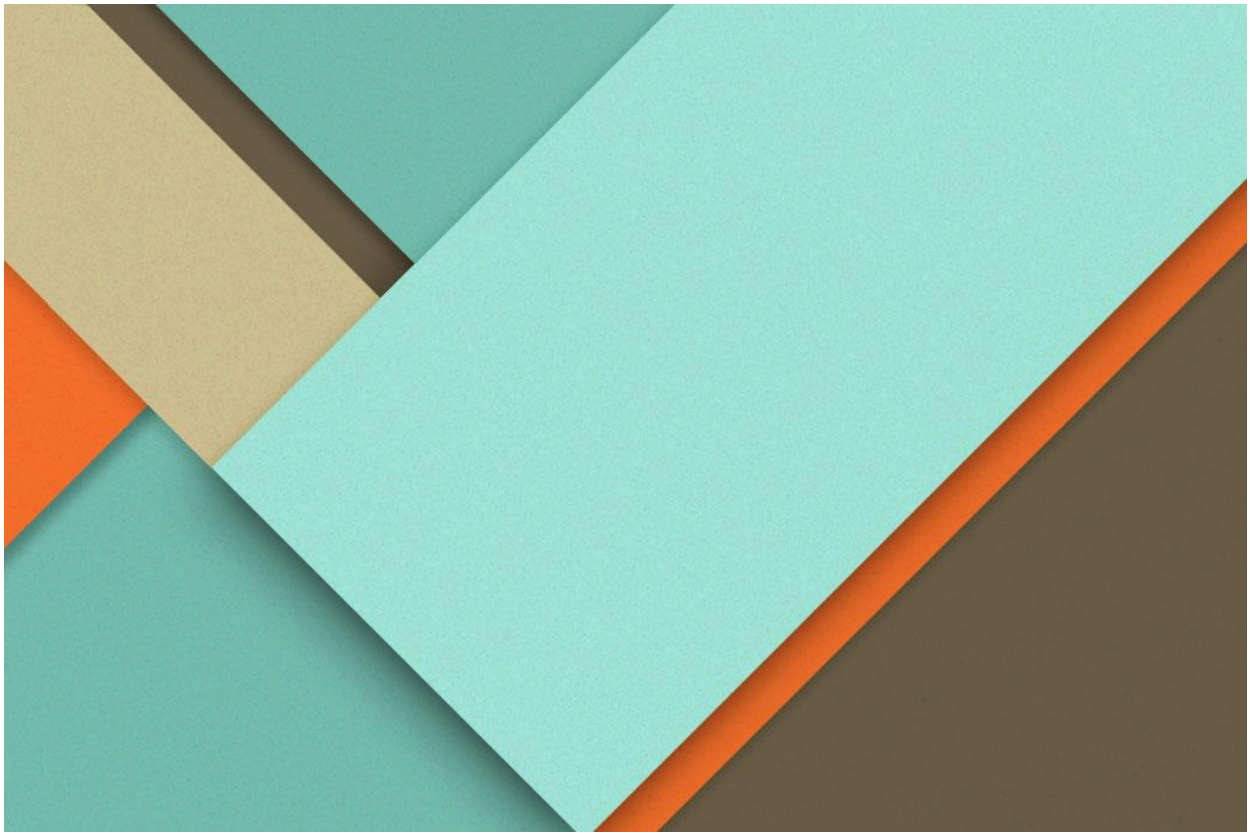


Assignment 3

Software Tools and Techniques for CSE

—

Mohit
23110207

Tanishq Bhushan Chaudhari
23110329

Github Repository

TABLE OF CONTENTS

Lab Session 9

Introduction

In this lab, I learned how to develop basic console applications using C# in Visual Studio 2022. The primary objective was to comprehend how C# operates within the .NET framework and to practice its fundamental syntax, control structures, and object-oriented programming features.

I performed various tasks, including setting up the development environment, writing programs that utilised loops, functions, and arrays, and reasoning about given code outputs. Through these exercises, I gained an understanding of how C# handles variables, increment operations, and array manipulations, as well as how code execution flow affects program results. This lab helped me gain practical experience with C# programming and improved my confidence in writing and analysing simple object-oriented programs

Tools and Setup

For this lab, I used Visual Studio 2022 Community Edition as the main development tool, along with the .NET 8.0 framework for running C# programs. Visual Studio was chosen because it provides an easy-to-use environment with built-in features like IntelliSense, debugging tools, and project templates for C# console applications.

To install the setup, I downloaded Visual Studio from the official Microsoft website and selected the ".NET desktop development" workload during installation. This included the C# compiler and the .NET SDK required to run and build console applications. After installation, I verified the setup by creating and running a sample "Hello World" program.

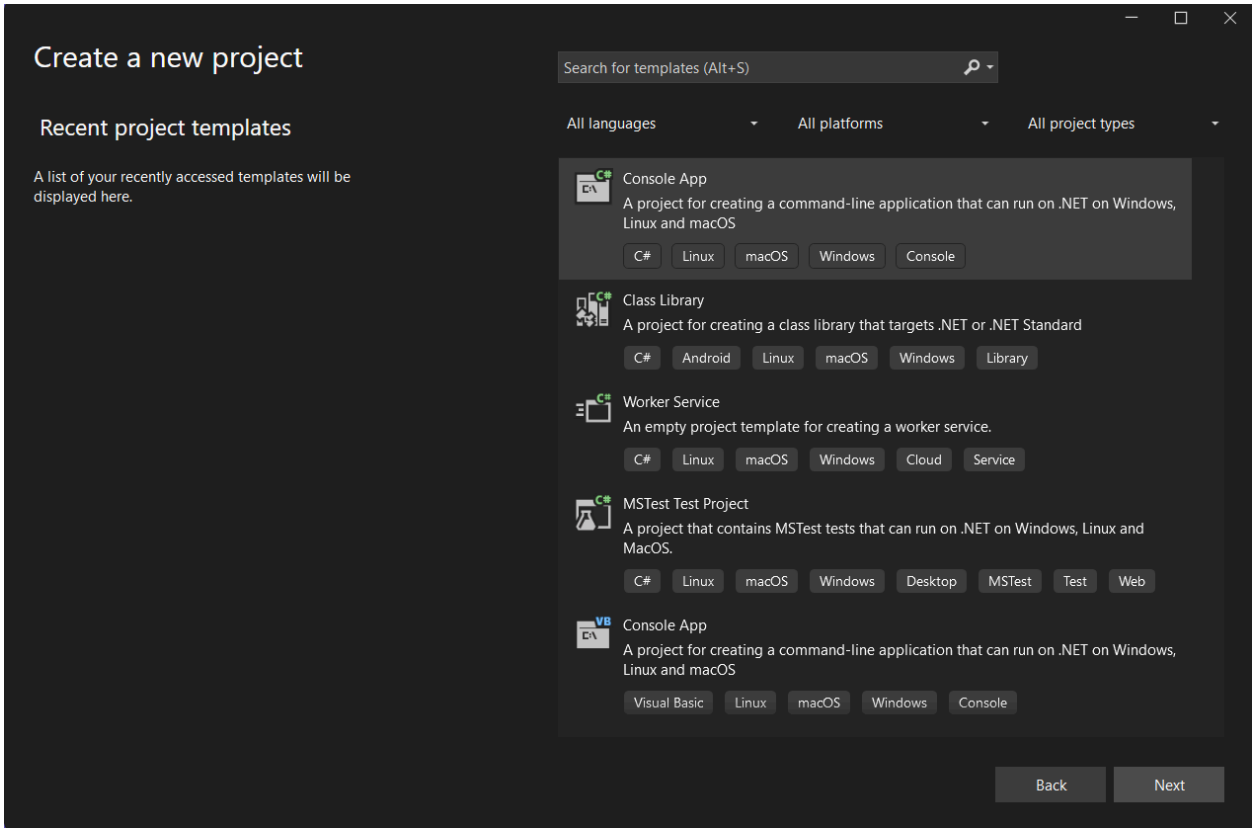
The step-by-step process of creating a new project and configuring it in Visual Studio is explained in the next section of this report.

Methodology and Execution

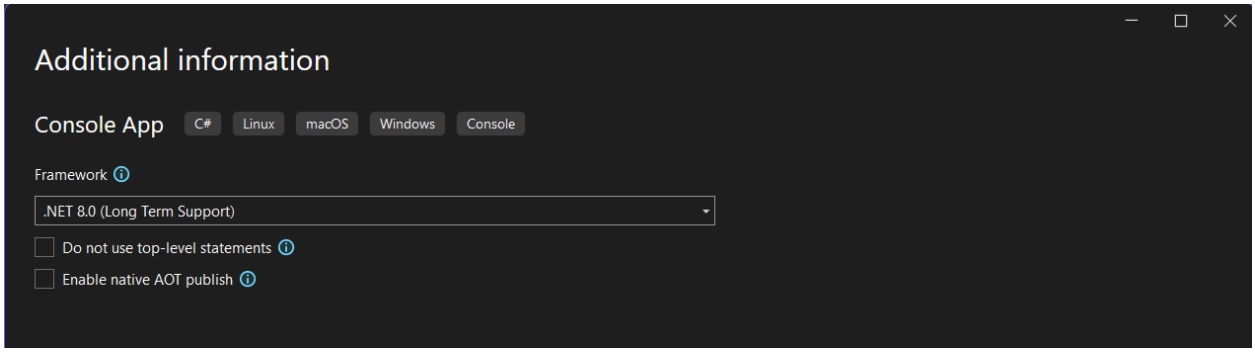
a) Setting Up .NET Development Environment

- Open Visual Studio.

- Select the Console App option with the C# tag and click on the next button.



- Browse the location, select some other options, and then, under additional information, ensure the framework is 6.0 or later; I chose 8.0.



- Write a small C# program in the Program.cs file.

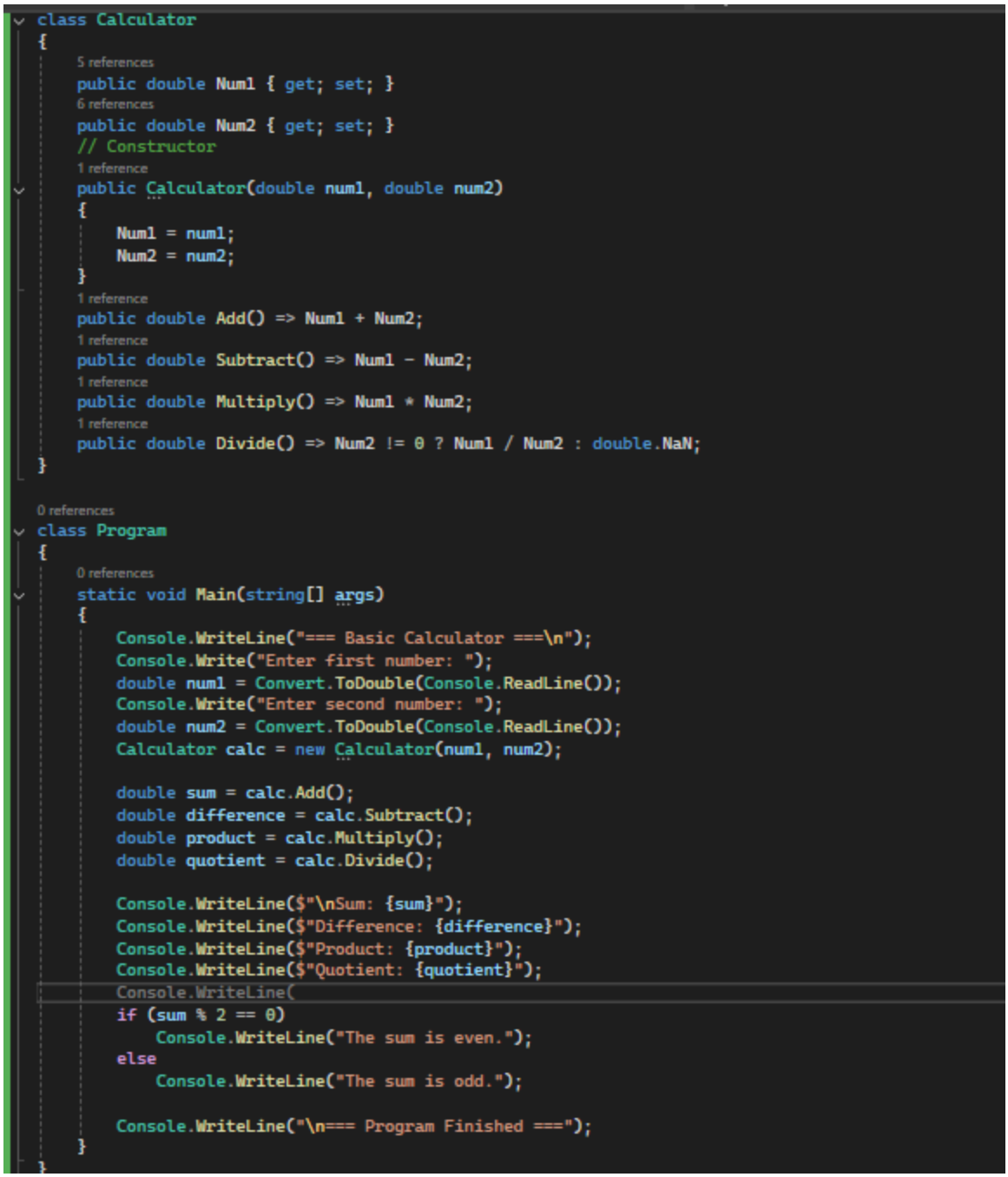
```
Program.cs*
FirstDotNetApp | Program | Main(string[] args)

using System;

0 references
class Program
{
    0 references
    static void Main(string[] args)
    {
        Console.WriteLine("=== Welcome to My First .NET App ===\n");

        Console.Write("Enter your name: ");
        string name = Console.ReadLine();

        Console.Write("Enter your age: ");
        int age = Convert.ToInt32(Console.ReadLine());

        int nextYearAge = age + 1;

        Console.WriteLine($"\nHello {name}!");
        Console.WriteLine($"You are {age} years old now.");
        Console.WriteLine($"Next year, you will be {nextYearAge} years old.");

        if (age < 18)
        {
            Console.WriteLine("You're a minor.");
        }
        else if (age < 60)
        {
            Console.WriteLine("You're an adult.");
        }
        else
        {
            Console.WriteLine("You're a senior citizen.");
        }

        Console.WriteLine("\nThank you for trying my first C# program!");
    }
}
```

- Run it by clicking the Start button on the toolbar above the code window. It will open a terminal-like window where the program will run.

```
Microsoft Visual Studio Debu

=== Welcome to My First .NET App ===

Enter your name: Mohit
Enter your age: 19

Hello Mohit!
You are 19 years old now.
Next year, you will be 20 years old.
You're an adult.

Thank you for trying my first C# program!
```

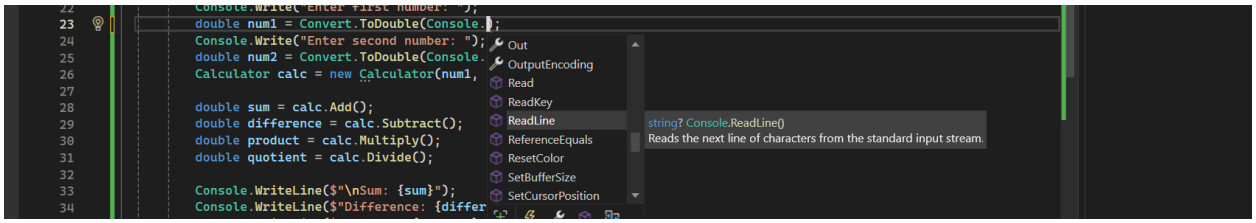
b) Understanding Basic Syntax and Control Structures

- The Program:

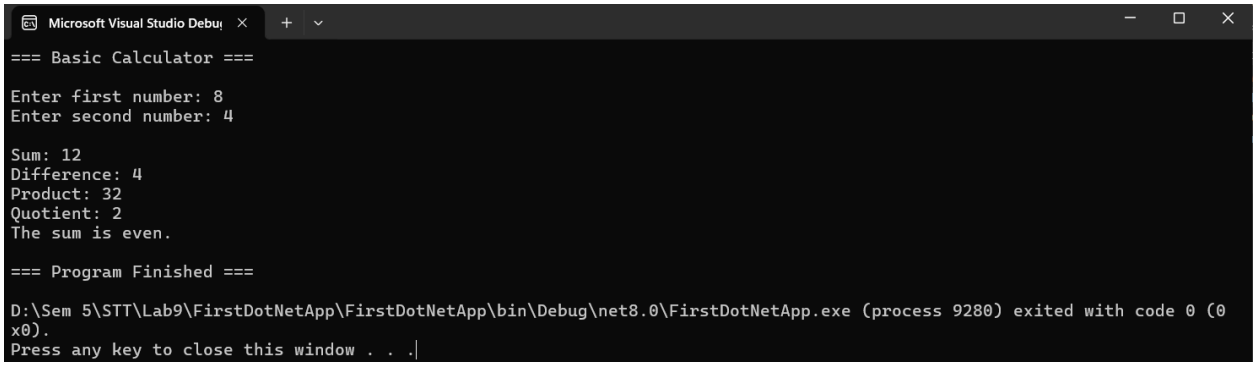


```
class Calculator
{
    5 references
    public double Num1 { get; set; }
    6 references
    public double Num2 { get; set; }
    // Constructor
    1 reference
    public Calculator(double num1, double num2)
    {
        Num1 = num1;
        Num2 = num2;
    }
    1 reference
    public double Add() => Num1 + Num2;
    1 reference
    public double Subtract() => Num1 - Num2;
    1 reference
    public double Multiply() => Num1 * Num2;
    1 reference
    public double Divide() => Num2 != 0 ? Num1 / Num2 : double.NaN;
}

0 references
class Program
{
    0 references
    static void Main(string[] args)
    {
        Console.WriteLine("=== Basic Calculator ===\n");
        Console.Write("Enter first number: ");
        double num1 = Convert.ToDouble(Console.ReadLine());
        Console.Write("Enter second number: ");
        double num2 = Convert.ToDouble(Console.ReadLine());
        Calculator calc = new Calculator(num1, num2);

        double sum = calc.Add();
        double difference = calc.Subtract();
        double product = calc.Multiply();
        double quotient = calc.Divide();

        Console.WriteLine($"\nSum: {sum}");
        Console.WriteLine($"Difference: {difference}");
        Console.WriteLine($"Product: {product}");
        Console.WriteLine($"Quotient: {quotient}");
        Console.WriteLine(
        if (sum % 2 == 0)
            Console.WriteLine("The sum is even.");
        else
            Console.WriteLine("The sum is odd.");

        Console.WriteLine("\n=== Program Finished ===");
    }
}
```

- We receive IDE suggestions as soon as we type "`Console.`" Based on these, I chose the function `ReadLine()`.



- Its outputs:



c) Using Loops and Functions

- The code:
  - For loop

```
Console.Write("Numbers from 1 to 10 using for loop: ");
for (int i = 1; i <= 10; i++)
{
    Console.Write(i + " ");
}
Console.WriteLine("\n");
```

  - Foreach loop

```
Console.Write("Numbers from 1 to 10 using foreach loop: ");
int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
foreach (int n in numbers)
{
    Console.Write(n + " ");
}
Console.WriteLine("\n");
```

  - Factorial function

```
static int Factorial(int num)
{
    int result = 1;
    for (int i = 1; i <= num; i++)
    {
        result *= i;
    }
    return result;
}
```

  - I have used the factorial function in the do-while loop itself. When a number is entered, we get its factorial value. If we instead write “exit”, we exit the loop.

- The do-while loop

```
string input;
do
{
    Console.Write("Enter a number to find its factorial (or type 'exit' to stop): ");
    input = Console.ReadLine();
    if (input.ToLower() != "exit")
    {
        if (int.TryParse(input, out int num))
        {
            int fact = Factorial(num);
            Console.WriteLine($"Factorial of {num} is {fact}\n");
        }
        else
        {
            Console.WriteLine("Please enter a valid number.\n");
        }
    }
} while (input.ToLower() != "exit");
Console.WriteLine("Program ended. Goodbye!");
}
```

- The output:

```
Microsoft Visual Studio Debu
Numbers from 1 to 10 using for loop: 1 2 3 4 5 6 7 8 9 10

Numbers from 1 to 10 using foreach loop: 1 2 3 4 5 6 7 8 9 10

Enter a number to find its factorial (or type 'exit' to stop): 6
Factorial of 6 is 720

Enter a number to find its factorial (or type 'exit' to stop): 7
Factorial of 7 is 5040

Enter a number to find its factorial (or type 'exit' to stop): exit
Program ended. Goodbye!

D:\Sem 5\STT\Lab9\FirstDotNetApp\FirstDotNetApp\bin\Debug\net8.0\FirstDotNetApp.exe (process 3724) exited with code 0 (0
x0).
Press any key to close this window . . .
```

d) Using Single and Multi-dimensional Arrays

- The code:
  - Bubble sort:

```
0 references
class ArrayOperations
{
    1 reference
    static void BubbleSort(int[] arr)
    {
        for (int i = 0; i < arr.Length - 1; i++)
            for (int j = 0; j < arr.Length - i - 1; j++)
                if (arr[j] > arr[j + 1])
                {
                    int temp = arr[j];
                    arr[j] = arr[j + 1];
                    arr[j + 1] = temp;
                }
    }
```

- Storing a 2-D array in a 1-D array:

```
1 reference
static int[] RowMajor(int[,] arr)
{
    int rows = arr.GetLength(0);
    int cols = arr.GetLength(1);
    int[] result = new int[rows * cols];
    int k = 0;
    for (int i = 0; i < rows; i++)
        for (int j = 0; j < cols; j++)
            result[k++] = arr[i, j];
    return result;
}

1 reference
static int[] ColumnMajor(int[,] arr)
{
    int rows = arr.GetLength(0);
    int cols = arr.GetLength(1);
    int[] result = new int[rows * cols];
    int k = 0;
    for (int j = 0; j < cols; j++)
        for (int i = 0; i < rows; i++)
            result[k++] = arr[i, j];
    return result;
}
```

- Matrix Multiplication:

```
static int[,] MatrixMultiply(int[,] A, int[,] B)
{
    int r1 = A.GetLength(0);
    int c1 = A.GetLength(1);
    int r2 = B.GetLength(0);
    int c2 = B.GetLength(1);
    if (c1 != r2) throw new Exception("Invalid matrix dimensions");
    int[,] C = new int[r1, c2];
    for (int i = 0; i < r1; i++)
        for (int j = 0; j < c2; j++)
            for (int k = 0; k < c1; k++)
                C[i, j] += A[i, k] * B[k, j];
    return C;
}
```

- Test Cases:

```
static void Main()
{
    int[] arr = { 5, 3, 8, 4, 2 };
    BubbleSort(arr);
    Console.WriteLine("Sorted Array:");
    PrintArray(arr);

    int[,] array2D = { { 1, 2, 3 }, { 4, 5, 6 } };
    Console.WriteLine("Row Major:");
    PrintArray(RowMajor(array2D));
    Console.WriteLine("Column Major:");
    PrintArray(ColumnMajor(array2D));

    int[,] A = { { 1, 2, 3 }, { 4, 5, 6 } };
    int[,] B = { { 7, 8 }, { 9, 10 }, { 11, 12 } };
    int[,] C = MatrixMultiply(A, B);
    Console.WriteLine("Matrix Multiplication Result:");
    PrintMatrix(C);
}
}
```

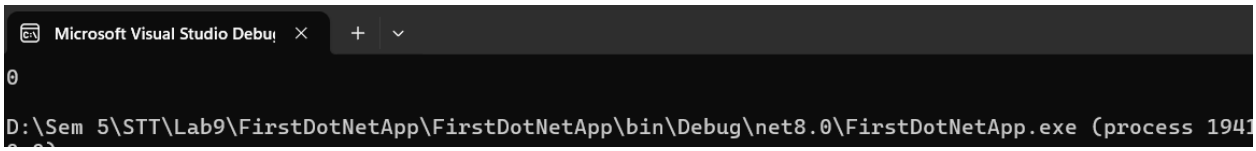
- The output:

e) Output Reasoning (Level 0)

- **Ques 1**: What will be the output of the following code?

```
using System; //namespace
class Program //default visibility⁴ is 'internal' if not specified
{
    public static void Main(string[] args)
    {
        int a = 0; //default visibility is 'private' (in a class)
        Console.WriteLine(a++);
    }
}
```

- Ans: Expected output: 0
  The post-increment operator (a++) outputs the current value before increasing it, so 0 is printed first and a becomes 1 afterwards.
- Verifying:

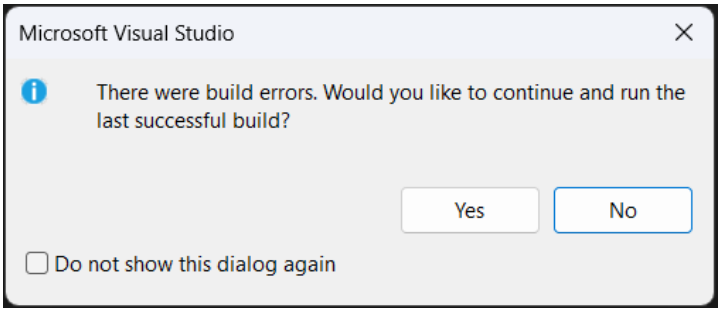


- **Ques 2**: What will be the output of the following code?

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

- Ans: Expected Output: The program should not run or give an error because the main function is not static.

- In C#, the Main method serves as the program's entry point, and it must be declared as static so that the runtime can invoke it without creating an instance of the class. A non-static Main would require an instance first, which the runtime doesn't create automatically.
- Verification:



f) Output Reasoning (Level 1)

- **Ques 1**: What will be the output of the following code?

```
class Program
{
    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);
    }
}
```

- Ans: Expected Output:

```
1
4-5
-1
```

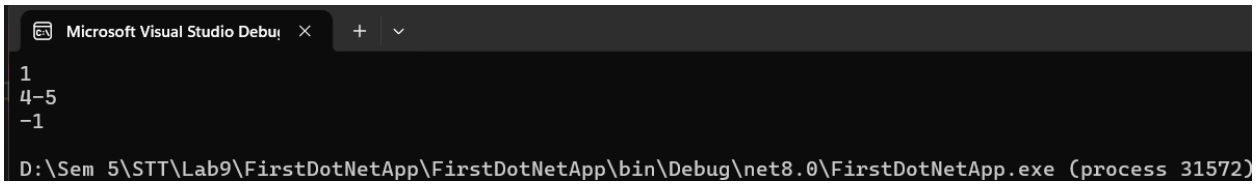
- Reason: Initially, the variable `b` is assigned the value `0`, and the value of `a` is incremented to `1` due to the post-increment operation. In C#, the `'+'` operator is used for concatenation, whereas in C++, the `','` operator can be used for separating output arguments. When the `','` operator is used in the first `WriteLine` statement, only the first argument (`a`) is printed, while the other arguments are still evaluated, and their increment operations are executed. Therefore, the first `WriteLine` call prints the value `1`, and by the end of this statement, the value of `a` becomes `4`.

  In the second `WriteLine` statement, the `'+'` operator is used, which performs concatenation. During this process, the value `4` is printed first, then `a` is incremented

to `5`. Next, `-5` is printed, and finally, `a` is incremented again to `6`. Here, the precedence of the increment operator (`++`) is higher than that of the unary minus (`-`) operator.

In the final `WriteLine` statement, the expression `~b` prints `-1`, which represents the bitwise complement of `0`.

- Verification: I verified my logic by using breakpoints and checking the values of each variable after each function call. The final output received:



- **Ques 2**: What will be the output of the following code?

```
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}");
```

- Ans: Expected Output:

  int x = 3;

  int y = 2 + ++x;

  x = 4 and y = 6

  x = 3 << 2;

  y = 10 >> 1;

  x = 12 and y = 5

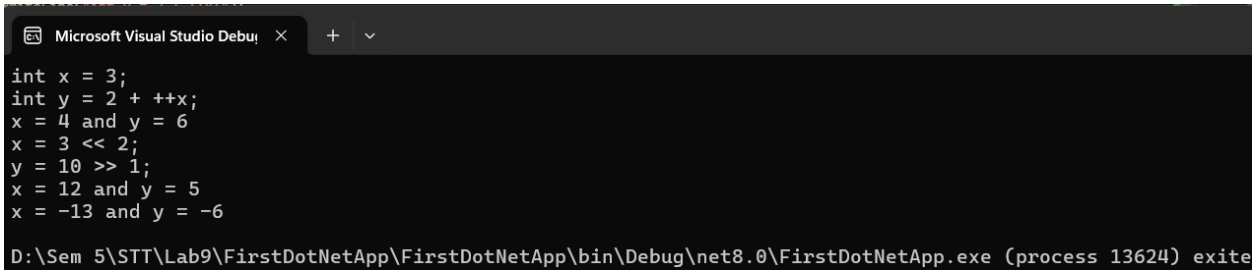
  x = -13 and y = -6

- Reason: nitially, the variable `x` is assigned the value `3`. In the expression `int y = 2 + ++x;`, the pre-increment operator (`++x`) increases the value of `x` from `3` to `4` before it is used in the calculation. Thus, the expression becomes `y = 2 + 4`, resulting in `y = 6`. At this stage, the values are `x = 4` and `y = 6`.

  Next, in the statements
  `x = 3 << 2;` and `y = 10 >> 1;`,
  the bitwise left and right shift operators are applied. The expression `3 << 2` shifts the bits of `3` two positions to the left, effectively multiplying the number by 22=42^2 = 422=4. Therefore, `x = 3 × 4 = 12`. Similarly, `10 >> 1` shifts the bits of `10` one position to the right, effectively dividing the number by 2, giving `y = 5`. After this operation, the values become `x = 12` and `y = 5`.

  Finally, the bitwise complement operator (~) is applied to both variables. The bitwise complement inverts all bits of a number. For `x = 12`, the complement `~12` results in `-13`, and for `y = 5`, the complement `~5` gives `-6`.
  Hence, the final values are `x = -13` and `y = -6`.

- Verification: Upon running the program, the output observed was the following. It matches with our expectation.



g) Output Reasoning (Level 2)

- **Ques 1**: What will be the output of the following code?

```
using System;
public class Program
{
    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());
        }
    }
}
```

- Ans: Expected output: In this program, the variable `i` is first assigned the value `int.MaxValue`, which represents the largest possible integer value in C#. The expression `-(i + 1) - i` is then evaluated.

  When `i + 1` is computed, it causes integer overflow, since the result exceeds `int.MaxValue`. In C#, such overflow in an unchecked context wraps around to `int.MinValue`. Thus, the expression becomes `-(int.MinValue) - int.MaxValue`. However, in C#, taking the negative of `int.MinValue` does not produce its positive equivalent, because `+2,147,483,648` lies outside the representable range of a 32-bit signed integer. As a result, `-int.MinValue` again equals `int.MinValue`. This means the expression effectively becomes `int.MinValue - int.MaxValue`, which, after wrapping, gives the final result `1`. Therefore, the first output is `1`.

  Next, the program enters the for loop. The loop is an empty loop that iterates a very large number of times. Crucially, when `i` is `int.MaxValue` and the loop perform `i++`, `i` wraps to `int.MinValue`. Because `int.MinValue <= int.MaxValue` is true, the loop condition remains true and the loop continues, so the loop becomes effectively infinite. I expected the loop to stop once `i` reached `int.MaxValue`, but observing the program showed it did not stop in that form.

  Changing the loop condition to stop before `int.MaxValue` (for example `i <= int.MaxValue - 1` or `i < int.MaxValue`) fixes the issue. In that case, when `i` is `int.MaxValue - 1` the final iteration runs, `i++` makes `i = int.MaxValue`, and on the next condition check `i <= int.MaxValue - 1` (or `i < int.MaxValue`) is

false, so the loop terminates normally. This is why the program stopped when you used `int.MaxValue - 1`.

- Verification: Upon running this program, the observed output was a stuck window with only '1' written on the first line, and then the window became stuck due to the infinite loop.

- **Ques 2** : What will be the output of the following code?

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

- Ans: Expected output: In this program, the `Main` method is defined as the entry point of execution with a parameter `string[] args`. Inside the method, the statement `Main(["CS202"]);` is used, which calls the same `Main` method again and passes a string array containing the value `"CS202"`.

  Since there is no terminating condition or return statement, this call results in infinite recursion; each call to `Main` invokes another call to itself. No output statements (`Console.WriteLine`) are present, so nothing is printed on the console.

  However, each recursive call occupies space on the call stack. When the stack memory becomes full, an error may be raised.

  Thus, the program is expected to produce no visible output and stay stuck in a loop.

- Actual Output:

- This indicates that the code resulted in infinite recursion, ultimately causing a stack overflow. Upon studying this error, we learnt that when the stack overflow finally occurs, the runtime attempts to print a portion of the call stack. These are not due to our code but due to the exception and error handling by VS.

Results and Analysis

After completing all the programs, I tested each one in Visual Studio 2022 using the .NET 8.0 framework. The results matched the expected theoretical outputs in almost all cases.

The first part of the lab involved creating a simple console program that took two user inputs and performed addition, subtraction, multiplication, and division, while also checking if the sum was even or odd using conditional statements. The output showed all the arithmetic results correctly, and the conditional block printed the correct parity of the sum.

In the next section, programs were written to practice loops and functions. Using `for`, `foreach`, and `do-while` loops, I verified that the program iterated through numbers from 1 to 10 and handled user input continuously until "exit" was entered. The factorial function worked correctly for positive integers, confirming that function calls and return types in C# behave as expected.

The array-based section involved implementing a bubble sort manually without using library functions. The output displayed the sorted array in ascending order. Then, a 2D array was converted into a 1D array in both row-major and column-major order. The printed results confirmed the correct sequence of data mapping. The matrix multiplication part produced the correct resulting matrix, matching the manual calculations for each cell value.

Finally, the output reasoning programs demonstrated a deeper understanding of increment operators, bitwise operations, and exception behaviour.

- Programs using post- and pre-increments produced outputs such as `1`, `4-5`, `and -1`, illustrating how operator precedence and argument evaluation work in C#.

- The bitwise program verified the correct use of shift and complement operations, printing results such as `x = -13` and `y = -6`.

- The overflow program printed `1` before becoming stuck due to an infinite loop, as predicted.

- The recursive `Main` method resulted in a stack overflow, confirming that recursion without a base case leads to runtime failure.

Overall, the practical outputs aligned with the logical and theoretical explanations, demonstrating a correct understanding of how C# executes expressions, manages memory, and handles control flow.

Discussion and Conclusion

This lab helped me understand both the syntax and runtime behaviour of C# through hands-on practice. Initially, I faced difficulty understanding the difference between pre-increment and post-increment operations and how the comma and plus operators behave differently in `Console.WriteLine()`. Step-by-step debugging using Visual Studio's breakpoints helped me visualise variable changes and confirm the expected output.

Working with arrays and loops improved my understanding of iteration control and indexing in multidimensional data structures. Implementing bubble sort without built-in methods provided a clear understanding of algorithmic logic and nested loops. Matrix multiplication required careful index management, which reinforced my understanding of row-major and column-major data access.

The later reasoning-based programs were challenging but informative. I learned how C# handles integer overflow, stack overflow, and bitwise operations, and how each of these reflects the underlying architecture of the .NET runtime.

In conclusion, this lab strengthened my skills in writing and analysing C# console applications, taught me the importance of precise logic and memory handling, and gave me confidence to debug and reason through more complex code structures in the future.

References

- C# Documentation

Lab Session 10

Introduction

The objective of this lab was to understand and apply the principles of Object-Oriented Programming (OOP) in C#. The lab covered creating and managing classes, defining constructors and destructors, implementing inheritance, overriding methods, and observing runtime polymorphism.
Before implementing OOP concepts, short output reasoning exercises were performed to understand C# operators, expressions, and evaluation order. Together, these exercises strengthened understanding of both language fundamentals and object-oriented behavior.
By completing this lab, I learned to design classes with encapsulation, apply inheritance, observe object lifecycle through constructors and destructors, and demonstrate polymorphism in C#.

Tools and Setup

Operating System: Windows
Editor: Visual Studio Code
Language: C# (.NET 8.0 SDK)

Setup Steps

1. Installed .NET SDK using the terminal.
2. Verified the installation by checking the version.
3. Created a new C# console project inside the Lab10 folder.
4. Replaced the default code file with the lab implementation.
5. Ran the project successfully using the terminal.

Methodology and Execution

Part A: Constructors and Data Control

In this part, a class was created to demonstrate constructors, destructors, and static members.
A private variable was used to store data for each object, and a static variable kept count of

all active objects. The constructor initialized values and printed messages indicating object creation, while the destructor printed messages when objects were destroyed.

```
using System;
using System.Threading;

namespace Lab10
{
    class ProgramClass
    {
        private int data;
        private static int counter = 0;

        public ProgramClass(int x)
        {
            data = x;
            counter++;
            Console.WriteLine($"Constructor Called. Object count = {counter}");
        }

        ~ProgramClass()
        {
            counter--;
            Console.WriteLine($"Object Destroyed. Remaining objects = {counter}");
        }

        public void set_data(int x) => data = x;

        public void show_data() => Console.WriteLine($"Data = {data}");
```

Three objects were created dynamically, their data values were assigned and displayed. The output showed constructor calls during creation and destructor messages during cleanup.
Since destructors in C# are managed by the garbage collector,

```
        static void TestConstructors()
        {
            ProgramClass obj1 = new ProgramClass(10);
            ProgramClass obj2 = new ProgramClass(20);
            ProgramClass obj3 = new ProgramClass(30);

            obj1.set_data(10);
            obj2.set_data(20);
            obj3.set_data(30);

            obj1.show_data();
            obj2.show_data();
            obj3.show_data();
        }
```

garbage collection was forced manually at the end of the program to observe destruction messages immediately.

```
=== Constructors and Data Control ===
Constructor Called. Object count = 1
Constructor Called. Object count = 2
Constructor Called. Object count = 3
Data = 10
Data = 20
Data = 30
Object Destroyed. Remaining objects = 2
Object Destroyed. Remaining objects = 1
Object Destroyed. Remaining objects = 0
```

This part demonstrated the lifecycle of an object, automatic constructor execution during creation, and destructor execution when memory is released. The static counter helped track how many instances were active at a time.

Part B: Inheritance and Method Overriding

```
    class Vehicle
    {
        protected int speed;
        protected int fuel;

        public Vehicle(int speed, int fuel)
        {
            this.speed = speed;
            this.fuel = fuel;
        }

        public virtual void ShowInfo()
        {
            Console.WriteLine($"Speed: {speed}, Fuel: {fuel}");
        }

        public virtual void Drive()
        {
            fuel -= 5;
            Console.WriteLine("Vehicle is moving...");
        }
    }

    class Car : Vehicle
    {
        private int passengers;

        public Car(int speed, int fuel, int passengers) : base(speed, fuel)
        {
            this.passengers = passengers;
        }

        public override void Drive()
        {
            fuel -= 10;
            Console.WriteLine("Car is moving with passengers...");
        }

        public override void ShowInfo()
        {
            Console.WriteLine($"Speed: {speed}, Fuel: {fuel}, Passengers: {passengers}");
        }
    }

    class Truck : Vehicle
    {
        private int cargoWeight;

        public Truck(int speed, int fuel, int cargoWeight) : base(speed, fuel)
        {
            this.cargoWeight = cargoWeight;
        }

        public override void Drive()
        {
            fuel -= 15;
            Console.WriteLine("Truck is moving with cargo...");
        }

        public override void ShowInfo()
        {
            Console.WriteLine($"Speed: {speed}, Fuel: {fuel}, Cargo Weight: {cargoWeight} kg");
        }
    }
}
```

This section demonstrated inheritance and runtime polymorphism using a base class and two derived classes.

```
    static void TestInheritance()
    {
        Vehicle[] vehicles = new Vehicle[3];
        vehicles[0] = new Vehicle(80, 100);
        vehicles[1] = new Car(120, 90, 4);
        vehicles[2] = new Truck(60, 150, 2000);

        foreach (var v in vehicles)
        {
            v.Drive();
            v.ShowInfo();
            Console.WriteLine();
        }
    }
}
```

```
static void Main(string[] args)
{
    Console.WriteLine("=== Constructors and Data Control ===");
    TestConstructors();

    // Force GC to show destructor output
    GC.Collect();
    GC.WaitForPendingFinalizers();
    Thread.Sleep(1000);

    Console.WriteLine("\n=== Inheritance and Method Overriding ===");
    TestInheritance();


}
```

The base class defined general properties and virtual methods for behavior, while the derived classes extended and modified these methods to create specialized behavior. Objects of each class were stored in an array of the base class type, and their methods were called through the base reference. The output showed that even when accessed through base class references, each derived class executed its own version of the method, demonstrating runtime polymorphism.

```
=== Inheritance and Method Overriding ===
Vehicle is moving...
Speed: 80, Fuel: 95

Car is moving with passengers...
Speed: 120, Fuel: 80, Passengers: 4

Truck is moving with cargo...
Speed: 60, Fuel: 135, Cargo Weight: 2000 kg
```

Explanation-

Overriding changes the behavior of a method by allowing a derived class to provide its own implementation of a method defined in the base class. When a base class method is marked as virtual, the derived class can redefine it using the override keyword. This

enables runtime polymorphism, where the actual method that executes is determined by the type of the object being referenced, not by the type of the reference variable itself.

Even though all the objects (Vehicle, Car, and Truck) were stored in a Vehicle[] array, each call to Drive() and ShowInfo() executed the correct version of the method according to the object's actual class. The Vehicle object used the base implementation, while the Car and Truck objects used their respective overridden versions. This demonstrates that C# dynamically binds method calls at runtime, ensuring that the most specific implementation (the derived one) is executed when accessed through a base-class reference.

This part explained how inheritance allows code reusability and how polymorphism ensures that the appropriate version of a method is executed based on the object type.

Results and Analysis

The results of Part A demonstrated the successful creation, initialization, and destruction of objects using constructors and destructors in C#. Each time an object was instantiated, the constructor was invoked automatically, printing a message and incrementing the static counter to reflect the total number of active objects. Similarly, when the objects went out of scope and were destroyed by the garbage collector, the destructor was executed, printing a message and decrementing the counter value. This behavior confirmed that constructors handle the setup of objects, while destructors handle cleanup during garbage collection. By observing the order and timing of these messages, it was evident that destructors in C# are not called immediately when an object goes out of scope but rather when the garbage collector decides to reclaim memory. This helped in understanding memory management and object lifecycle in managed environments like .NET.

In the inheritance and method overriding section, the results clearly showed the difference in behavior between base class and derived class methods. The base class (Vehicle) executed its own Drive() and ShowInfo() implementations, while the derived classes (Car and Truck) overrode these methods to provide specialized functionality. The Car object displayed the message "Car is moving with passengers..." and reduced fuel by 10 units, while the Truck object printed "Truck is moving with cargo..." and reduced fuel by 15 units. When these objects were stored in a Vehicle[] array and accessed through base-class references, the correct derived class methods were still executed. This confirmed the working of runtime polymorphism, where method resolution occurs dynamically based on the actual object type rather than the reference type. The experiment thus verified that overriding allows classes to modify or extend inherited behavior, enabling flexibility and extensibility in program design.

Discussion and Conclusion

This lab provided a comprehensive practical understanding of the key Object-Oriented Programming principles in C#, including encapsulation, inheritance, method overriding, and polymorphism. The first part highlighted how constructors and destructors manage the lifecycle of objects, showing that initialization and cleanup can be automated through these mechanisms. The use of a static counter illustrated how data can be shared among all instances of a class, providing a global view of object creation and destruction across the program. Observing the destructor messages emphasized how C# relies on automatic garbage collection instead of manual memory management, a key feature of the .NET environment.

The second part of the lab reinforced the concepts of inheritance and polymorphism. It demonstrated that derived classes can inherit properties and behaviors from a base class while customizing certain functionalities through overriding. This structure promotes code reusability and modularity, allowing developers to build extensible systems where future modifications or new derived classes can be easily integrated without altering existing code. The behavior of overridden methods when accessed via base-class references showed the importance of virtual and override keywords in enabling runtime polymorphism.

Overall, this lab not only strengthened the theoretical understanding of OOP concepts but also provided hands-on experience with their implementation in C#. It bridged the gap between conceptual design and actual programming, highlighting how OOP principles contribute to writing clean, maintainable, and scalable software. Through these exercises, a deeper appreciation was developed for the power of class hierarchies, encapsulation, and polymorphism in creating robust and flexible applications.

Output Reasoning (Level 0)

Given Code 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}");
```

Output

Reason

The program starts with two integer variables a and b. The value of a is set to 3. Then b is assigned the value of a++ which means that b gets the current value of a (3), and afterward a is incremented by 1. So now a becomes 4 and b is 3.

In the first Console.WriteLine statement, the expression +a++ is evaluated. The unary plus has no effect on the value, so the current value of a (4) is printed, and then a increases to 5 after the statement. The next part, -++b, first increments b before using it. Since b was 3, ++b makes it 4, and then the negative sign makes it -4. Therefore, the first output line is "a is 4, b is -4".

Next, another set of variables c and d is created. c starts at 3, and d is assigned ++c. The pre-increment operator ++c first increases c by 1, making it 4, and then assigns this value to d. So both c and d are now 4.

In the second Console.WriteLine statement, -c-- is evaluated first. The negative sign gives -4, and the post-decrement operator decreases c by 1 afterward, making c become 3. Then the expression ~d performs a bitwise NOT on d. For d = 4, the bitwise complement (~4) equals -5. Therefore, the second line prints "c is -4, d is -5".

Given Code 2

```
using System;
class Program
{
    int age;
    Program() => age=age==0?age+1:age-1;
    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]

Reason

The string "010%" is first processed by Replace('0','%'), which changes all '0' characters to '%', resulting in "%%%1%". The length of this string is 4, so k = 4.
When a new Program object is created, the instance variable age is initially 0, and the constructor sets age = age + 1, making it 1. The expression ++new Program().age increments it again to 2. The expression (k << ++new Program().age) performs a left shift of 4 by 2 bits, giving 16.

The expression "010%".Split('1')[1][0] splits the string at '1', producing ["0", "0%"], and the second part's first character is '0'.
The expression "010%".Split('0')[1][0] splits the string at '0', producing ["", "1", "%"], and the second part's first character is '1'.

Finally, "123".ToCharArray()[~-1] accesses the character at index [~-1], where ~-1 equals 0. Thus, it accesses the first element '1'. Converting this to an integer gives 1.

Hence, the output is [16][0][1][1].

Given Code 3

```
using System;

class Program
{
    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));
    }
}
```

Output

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

Reason

The array nums initially contains the values {0, 1, 0, 3, 12}. The variable pos keeps track of the position where the next non-zero element should be placed. The loop iterates over each element of the array.

When i = 0, nums[0] = 0, so nothing happens.
When i = 1, nums[1] = 1 (non-zero), it swaps nums[1] with nums[pos] (nums[0]). The array becomes {1, 0, 0, 3, 12}, and pos becomes 1.
When i = 2, nums[2] = 0, so no change.
When i = 3, nums[3] = 3 (non-zero), it swaps nums[3] with nums[pos] (nums[1]). The array becomes {1, 3, 0, 0, 12}, and pos becomes 2.
When i = 4, nums[4] = 12 (non-zero), it swaps nums[4] with nums[pos] (nums[2]). The array becomes {1, 3, 12, 0, 0}, and pos becomes 3.

At the end of the loop, all non-zero elements have been moved to the front in their original order, and zeros have been pushed to the end. Therefore, the final output is 1, 3, 12, 0, 0.

Output Reasoning (Level 1)

Given Code 1

```
using System;
class Program
{
    int age;
    Program() =>7 age=age==0?age+1:age-1;
    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]
```

Reason

The string "010%" is modified using Replace('0','%'), which changes all '0' characters to '%', giving "%%%1%". The length of this string is 4, so k = 4.

When a new Program object is created, the instance variable age starts at 0 by default. In the constructor, the expression age = age == 0 ? age + 1 : age - 1 makes age = 1. Then, in the expression ++new Program().age, the age is incremented again to 2. Therefore, (k << ++new Program().age) performs a left shift of 4 by 2 bits, giving 16.

Next, "010%".Split('1') splits the string around '1', producing ["0", "0%"]. The element at index [1] is "0%", and its first character [0] is '0'.
Then, "010%".Split('0') splits the string around '0', producing ["", "1", "%"]. The element at index [1] is "1", and its first character [0] is '1'.

Finally, "123".ToCharArray()[~-1] accesses the element at index [~-1]. Since ~-1 equals 0, it accesses the first element, which is '1'. Converting it to an integer gives 1.

Hence, the overall output is [16][0][1][1].

Given Code 2

```
using System;
class Program
{
    int f;
    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);
    }
    Program() => f = 0;
    Program(int x) => f=x;
}
```

Output

```
run 1
2
4
6
8
10
11

run 2
11

run 3
0
```

Reason

The program starts by printing run 1. The expression new int() creates a new integer with value 0, and "0".Length gives 1. When added, the result is 1, so the constructor Program(int x) is called with x = 1. Inside that constructor, the instance variable f is set to 1.

Next, the for loop runs. It has a semicolon at the end, meaning it has no body, and everything happens inside the loop header. The condition i < 5 && ++p.f >= 0 first increases f before checking the condition. Initially, f is 1, so before the first iteration it becomes 2. The last expression Console.WriteLine(p.f++) prints the current value of f and then increases it again. Because f is incremented twice in every iteration, it goes 2, 4, 6, 8, and 10 during the five iterations. After the loop ends, f has become 11.

Then the next line for (; p.f == 0;); does nothing because f is not 0. The following Console.WriteLine(p.f) prints 11. That completes run 1.

For run 2, a new Program object is created with the constructor Program(p.f). Since p.f is 11, the new object also has f = 11, which is then printed.

For run 3, the default constructor Program() runs and sets f = 0. This value is printed, completing the output.

Hence, the output shows f changing from 1 to 11 through increments in the first loop, remaining 11 in the second run, and resetting to 0 in the third run.

Given Code 3

```
public class A
{
    public virtual void f1()
    {
        Console.WriteLine("f1");
    }
}
public class B:A
{
    public override void f1() => Console.WriteLine("f2");
}


class Program
{
    static int i=0;
    public event funcPtr handler;
    public delegate void funcPtr();
    public void destroy()
    {
        if (i == 6)
            return;
        else
        {
            Console.WriteLine(i++);
            destroy();
        }
    }
    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 here[8] about ?. operator
        p = null;
        i = -6;
        p?.destroy();
        (new Program())?.destroy();
    }
}
```

Output

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

Reason

The program begins by defining two classes, A and B. Class A has a virtual method f1 that prints “f1”. Class B inherits from A and overrides f1 to print “f2”.

In class Program, a static integer i is defined and initialized to 0. There is also an event named handler of delegate type funcPtr, which is a delegate representing a method that takes no parameters and returns void.

The destroy() method works recursively. If i equals 6, it stops (returns). Otherwise, it prints the current value of i, increments i, and calls destroy() again. This causes values 0 through 5 to be printed in sequence each time destroy() runs from i = 0.

In the Main method, a Program object p is created. Two handlers are added to p.handler — one referencing f1 from an instance of A, and the other referencing f1 from an instance of B. When p.handler() is invoked, both attached functions are executed in order, so the output is “f1” followed by “f2”.

Next, both handlers are removed from the event using -=. The event now has no methods attached. Then p?.destroy() is called. The “?.“ is the null-conditional operator. It means that destroy() will only run if p is not null. Since p currently refers to an existing Program object, destroy() executes normally, printing the values 0, 1, 2, 3, 4, and 5 before stopping when i reaches 6.

After that, p is set to null, and i is assigned -6. Then p?.destroy() runs again, but this time p is null, so the “?.“ operator prevents the call from happening and nothing is printed.

Finally, a new Program object is created temporarily with (new Program())?.destroy(). Since it is not null, destroy() runs again from i = -6. The method prints numbers from -6 up to 5, because it keeps incrementing i until it reaches 6.

Overall, the output first shows f1 and f2 from the event calls, then numbers 0 to 5 from the first destroy() call, then numbers -6 to 5 from the final destroy() call.

Output Reasoning (Level 2)

Given Code 1

```
public class Institute
{
    internal int i = 7;
    public Institute()
    {
        Console.Write("1");
    }
    public virtual void info()
    {
        Console.Write("2");
    }
}
public class IITGN:Institute
{
    public int i = 8;
    public IITGN()
    {
        Console.Write("3");
    }
    public IITGN(int i)
    {
        Console.Write("4");
    }
    public override void info()
    {
        Console.Write("5");
    }
}
class Program
{
    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));
    }
}
```

Output

```
61214135
8
−8
```

Reason

The program begins execution in the Main method. The first statement prints 6. Next, an object of the class Institute is created. When this happens, the constructor of Institute runs

and prints 1. After that, the info method of this object is called, which prints 2. Up to this point, the output is 612.

Next, a new object of the derived class IITGN is created using the constructor that takes an integer argument. In C#, when an object of a derived class is created, the base class constructor is always called first. Therefore, the Institute constructor runs and prints 1, and then the IITGN constructor with an integer parameter runs and prints 4. The output now becomes 61214.

Then another object of IITGN is created using the default constructor. As before, the base constructor of Institute executes first and prints 1, and then the IITGN default constructor prints 3. The output becomes 6121413.

After that, the info method is called on this object. Since IITGN overrides the info method from Institute, the overridden version runs and prints 5. The combined output at this point is 61214135.

A new line is printed next. Then, the value of i is printed from the IITGN object. The IITGN class defines its own i variable as 8, so it prints 8.

Finally, the program casts the same object to its base type Institute and prints the bitwise complement of its i value. In the base class Institute, the variable i has the value 7. The bitwise complement of 7 is -8, so -8 is printed.

Thus, the final output is 61214135 followed by 8 and then -8.

Given Code 2

```
using System;
public class Program
{
    public delegate void mydel();
    public void fun1()
    {
        Console.WriteLine("fun1()");
    }
    public  void fun2()
    {
        Console.WriteLine("fun2()");
    }
    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();
    }
}
```

Output

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

Reason

The program defines a delegate named mydel, which can reference any method that has no parameters and no return value. The class Program contains two such methods, fun1 and fun2, both of which simply print their names.

In the Main method, an object p of Program is created. Then a delegate object obj1 is created and initialized with p.fun1. After that, another method p.fun2 is added to obj1 using the += operator. This means obj1 now holds a list of two methods, fun1 followed by fun2. When obj1 is invoked, it calls both methods in the order they were added. Hence, during run 1, it first calls fun1() and then fun2().

Next, another delegate obj2 is created and initialized with p.fun2. Then p.fun1 is added to obj2. This time, the order of methods inside obj2 is fun2 first and fun1 second. When obj2 is invoked, it calls the methods in the same sequence, so run 2 prints fun2() followed by fun1().

After that, obj2 -= p.fun2 removes the first method reference (p.fun2) from the delegate's invocation list. Now obj2 contains only p.fun1. When obj2 is called again, only fun1() executes.

Therefore, the output sequence shows the order in which delegates hold and invoke their referenced methods.

Given Code 3

```
using System;
using System.Collections[9];
public class Program
{
    int x;
    public static void Main(string[] args)
    {
        ArrayList[10] 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);
    }
}
```

Output

```
1
1
202
202
1
111
```

Reason

The program begins execution in the Main method. An ArrayList named L is created to store Program objects. Two new Program objects are added to this list. Each object has an integer variable x, which starts with the default value 0.

In the first loop, i starts from 0 and goes up to L.Count - 1, which is 1 because there are two elements. Each element in L is typecast to Program and its x value is incremented before printing. Since both objects start with x = 0, the first iteration increases x of the first object to 1 and prints 1, and the second iteration increases x of the second object to 1 and prints 1.

Next, the statement L[0] = L[1] assigns the reference of the second object to the first element. Now both L[0] and L[1] refer to the same Program object. When the line ((Program)L[0]).x = 202 executes, it changes the x value of that shared object to 202. As a result, both L[0] and L[1] now point to the same object whose x value is 202.

In the next loop, both iterations print the x value of the same object, which is 202. Therefore, two consecutive 202 values are printed.

After that, the statement ((Program)L[0]).x = 111 changes the x value of the shared object to 111. Then L.RemoveAt(0) removes the first element from the ArrayList. Since both entries pointed to the same object, removing one reference does not delete the actual object. The remaining element still points to that same object.

Next, the program prints L.Count, which is now 1 because one element was removed. Finally, it prints the x value of the remaining element, which is 111.

References

• Microsoft .NET Documentation – Classes and Objects in C#

• C# Language Reference – Constructors, Destructors, and Inheritance

• Visual Studio Code Documentation – .NET SDK Setup

• Lecture 10

• Lecture 11