ASSIGNMENT 3

SOFTWARE TOOLS AND TECHNIQUES FOR CSE

Sai Keerthana Pappala, Revathi Katta

Roll No: 23110229 , 23110159

November 8, 2025

Table of Contents

1 LABORATORY SESSION 9

INTRODUCTION

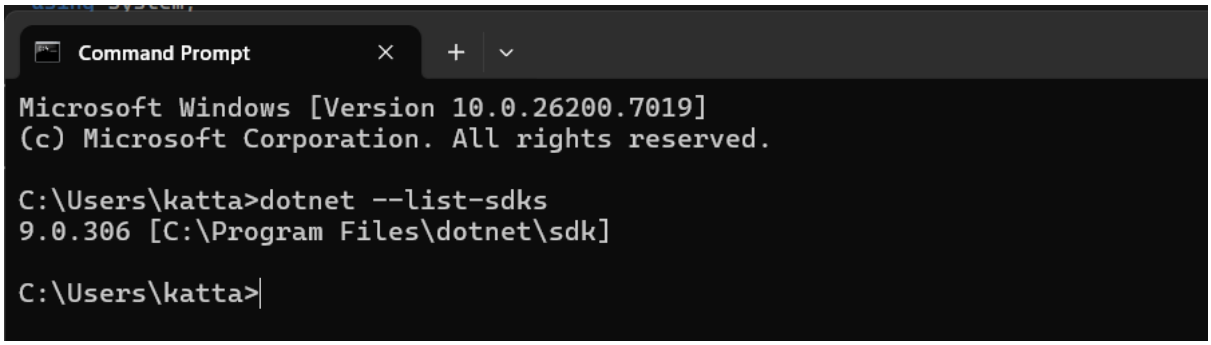
This laboratory session was focused on understanding and applying fundamental .NET development concepts using C# in the Visual Studio environment. The primary objective was to develop simple console applications that demonstrate basic programming constructs such as syntax, control structures, loops, functions, and object-oriented programming principles. This hands-on experience aimed to familiarize with the Visual Studio Integrated Development Environment (IDE) and the .NET SDK toolchain while implementing practical algorithms like arithmetic operations, looping, recursion, sorting, and matrix manipulations.

SETUP AND TOOLS

Before starting the lab activities, I ensured that my system was properly configured with all necessary components for effective .NET development. System and Tools Used:

Component	Description / Version
Operating System	Windows 11
Development IDE	Visual Studio 2022 (Community Edition)
.NET SDK	.NET 9.0.306 (compatible with .NET 6 and later)
Programming Language	C# (latest stable version)
Version Control	Git (for project version management)

Visual Studio 2022 was installed with the "`.NET desktop development`" workload, enabling creation and management of C# console applications targeting frameworks from .NET 6 onwards. The .NET SDK was verified via command line using `dotnet --list-sdks` to confirm the installed versions. This configuration provided a robust and modern environment to develop, debug, and run the console projects as required by the lab.



METHODOLOGY AND EXECUTION

Leveraging the Visual Studio IDE, the development workflow involved writing, building, and running individual programs to meet the objectives of each lab task. For each program, the following general steps were followed:

- **Code Development:** C# source code was written in the default `Program.cs` file or additional class files as needed. Object-oriented programming principles were applied, encapsulating logic in classes and methods to promote modularity and clarity.
- **Build Process:** The project was built using Visual Studio's integrated build tools. The compiler checked syntax and semantics, reporting any errors or warnings. Successful builds produced executable binaries targeting the specified .NET runtime.
- **Execution and Testing:** Executable programs were run within Visual Studio's debugging environment or as standalone console applications. Inputs were provided via the console as per program requirements, and outputs were observed to validate correctness.
- **Iteration and Refinement:** Based on runtime observations and test results, the programs were iteratively refined to correct logic errors, improve readability, or extend functionality.

The lab assignments progressed through these key programming tasks:

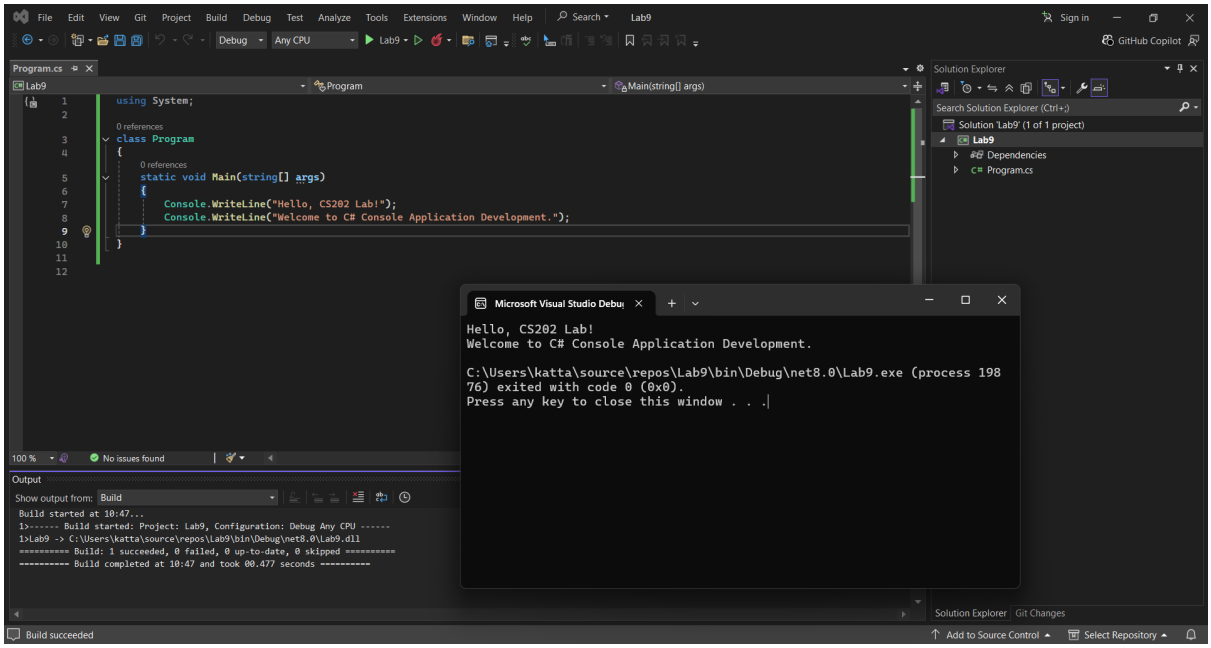
- Writing programs that accept user input for arithmetic operations and use conditional statements to evaluate properties like even or odd sums.
- Implementing various loop constructs (`for`, `foreach`, and `do-while`) to display sequences and manage user interactions.
- Creating static functions to calculate mathematical operations such as factorials, demonstrating the use of methods outside `Main`.
- Managing single and multi-dimensional arrays by implementing sorting algorithms (bubble sort), transforming 2-D arrays into 1-D arrays in both row-major and column-major order, and performing matrix multiplication operations.

Each program was designed to be self-contained, ensuring clarity in execution flow and ease of testing. The outputs from these programs were collected to demonstrate successful task completion.

The following sections include the code listings and respective outputs generated for each implemented task.

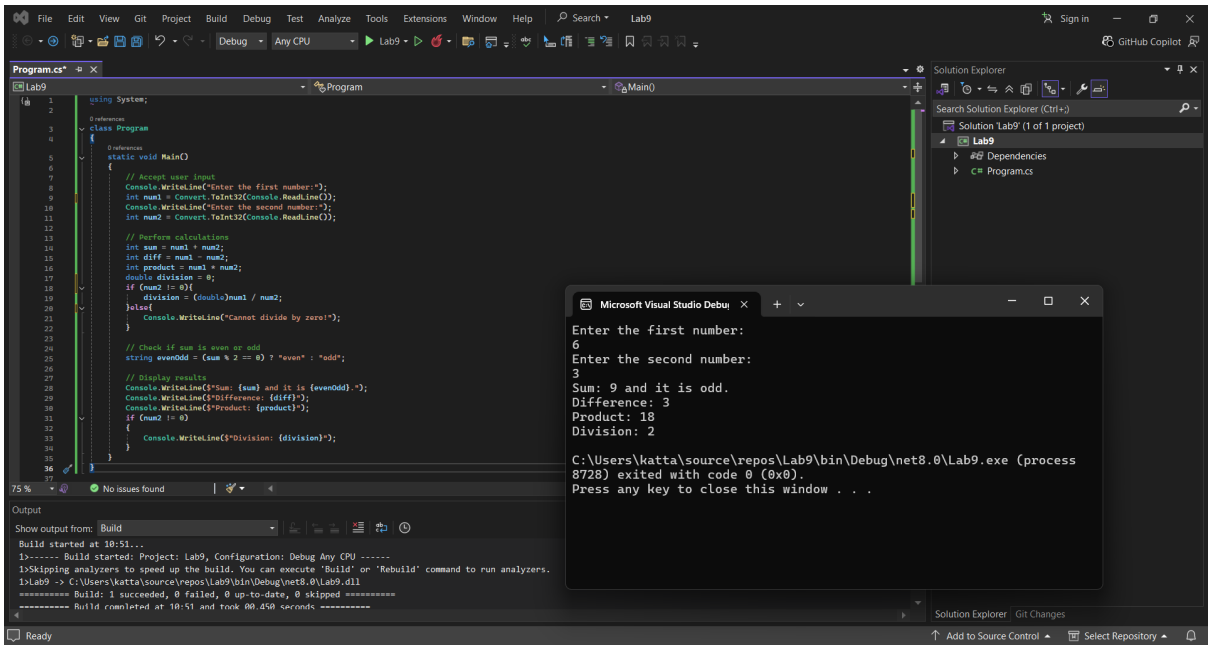
1.1 Setting Up .NET Development Environment and Running a Simple Program

The image below shows the code and the corresponding output obtained.



1.2 Understanding Basic Syntax and Control Structures

The image below shows the code and the corresponding output obtained.



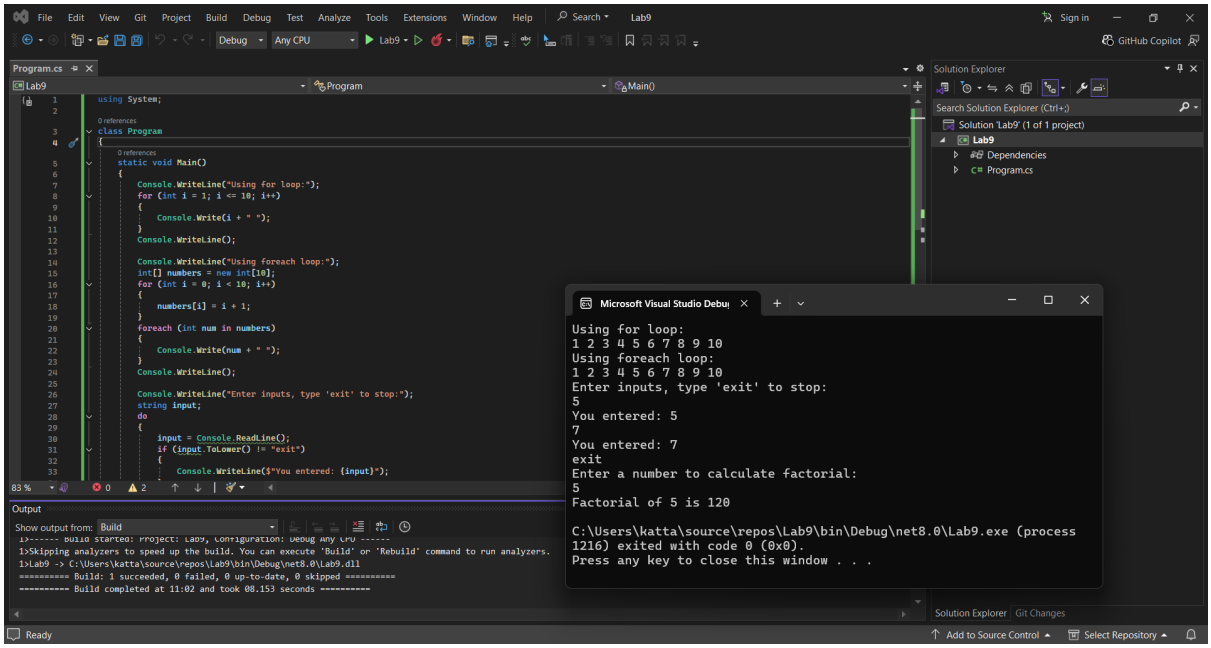
1.3 Using Loops and Functions

Listing 1: Loops and Functions in C#

```
using System;

class Program
{
    static void Main()
    {
```

```
        Console.WriteLine("Using for loop:");
        for (int i = 1; i <= 10; i++)
        {
            Console.Write(i + " ");
        }
        Console.WriteLine();

        Console.WriteLine("Using foreach loop:");
        int[] numbers = new int[10];
        for (int i = 0; i < 10; i++)
        {
            numbers[i] = i + 1;
        }
        foreach (int num in numbers)
        {
            Console.Write(num + " ");
        }
        Console.WriteLine();

        Console.WriteLine("Enter inputs, type 'exit' to stop:");
        string input;
        do
        {
            input = Console.ReadLine();
            if (input.ToLower() != "exit")
            {
                Console.WriteLine($"You entered: {input}");
            }
        } while (input.ToLower() != "exit");

        Console.WriteLine("Enter a number to calculate factorial:");
        int number;
        if (int.TryParse(Console.ReadLine(), out number) && number >= 0)
        {
            long fact = Factorial(number);
            Console.WriteLine($"Factorial of {number} is {fact}");
        }
        else
        {
            Console.WriteLine("Invalid input for factorial calculation.");
        }
    }

    static long Factorial(int n)
    {
        if (n == 0 || n == 1)
            return 1;
        long result = 1;
        for (int i = 2; i <= n; i++)
        {
            result *= i;
        }
        return result;
    }
}
```

The image below shows the corresponding output obtained for above code.



1.4 Using Single and Multi-dimensional Arrays

Listing 2: Array Operations including Sorting and Matrix Multiplication

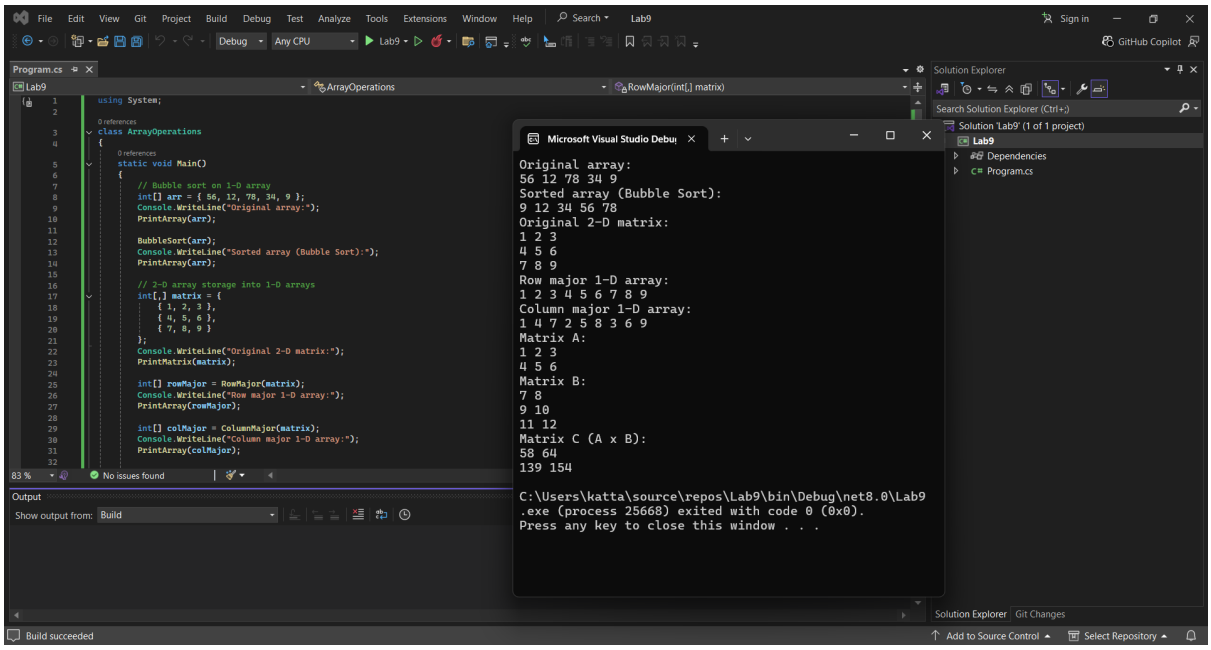
```
using System;

class ArrayOperations
{
    static void Main()
    {
        // Bubble sort on 1-D array
        int[] arr = { 56, 12, 78, 34, 9 };
        Console.WriteLine("Original array:");
        PrintArray(arr);

        BubbleSort(arr);
        Console.WriteLine("Sorted array (Bubble Sort):");
        PrintArray(arr);

        // 2-D array storage into 1-D arrays
        int[,] matrix = {
            { 1, 2, 3 },
            { 4, 5, 6 },
            { 7, 8, 9 }
        };
        Console.WriteLine("Original 2-D matrix:");
        PrintMatrix(matrix);

        int[] rowMajor = RowMajor(matrix);
        Console.WriteLine("Row major 1-D array:");
        PrintArray(rowMajor);

        int[] colMajor = ColumnMajor(matrix);
        Console.WriteLine("Column major 1-D array:");
        PrintArray(colMajor);
```

```
        // Matrix multiplication
        int[,] A = {
            {1, 2, 3},
            {4, 5, 6}
        };

        int[,] B = {
            {7, 8},
            {9, 10},
            {11, 12}
        };

        Console.WriteLine("Matrix A:");
        PrintMatrix(A);
        Console.WriteLine("Matrix B:");
        PrintMatrix(B);

        int[,] C = MatrixMultiply(A, B);
        Console.WriteLine("Matrix C (A x B):");
        PrintMatrix(C);
    }

    static void BubbleSort(int[] array)
    {
        int n = array.Length;
        for (int i = 0; i < n - 1; i++)
        {
            for (int j = 0; j < n - i - 1; j++)
            {
                if (array[j] > array[j + 1])
                {
                    // swap
                    int temp = array[j];
                    array[j] = array[j + 1];
                    array[j + 1] = temp;
                }
            }
        }
    }

    static int[] RowMajor(int[,] matrix)
    {
        int rows = matrix.GetLength(0);
        int cols = matrix.GetLength(1);
        int[] result = new int[rows * cols];

        int index = 0;
        for (int i = 0; i < rows; i++)
        {
            for (int j = 0; j < cols; j++)
            {
                result[index++] = matrix[i, j];
            }
        }
        return result;
    }

    static int[] ColumnMajor(int[,] matrix)
```

```
    {
        int rows = matrix.GetLength(0);
        int cols = matrix.GetLength(1);
        int[] result = new int[rows * cols];

        int index = 0;
        for (int j = 0; j < cols; j++)
        {
            for (int i = 0; i < rows; i++)
            {
                result[index++] = matrix[i, j];
            }
        }
        return result;
    }

    static int[,] MatrixMultiply(int[,] A, int[,] B)
    {
        int rowsA = A.GetLength(0);
        int colsA = A.GetLength(1);
        int rowsB = B.GetLength(0);
        int colsB = B.GetLength(1);

        if (colsA != rowsB)
        {
            throw new Exception("Incompatible matrices for multiplication")
               ;
        }

        int[,] C = new int[rowsA, colsB];

        for (int i = 0; i < rowsA; i++)
        {
            for (int j = 0; j < colsB; j++)
            {
                int sum = 0;
                for (int k = 0; k < colsA; k++)
                {
                    sum += A[i, k] * B[k, j];
                }
                C[i, j] = sum;
            }
        }
        return C;
    }

    static void PrintArray(int[] array)
    {
        foreach (int num in array)
        {
            Console.Write(num + " ");
        }
        Console.WriteLine();
    }

    static void PrintMatrix(int[,] matrix)
    {
        int rows = matrix.GetLength(0);
```

```
        int cols = matrix.GetLength(1);

        for (int i = 0; i < rows; i++)
        {
            for (int j = 0; j < cols; j++)
            {
                Console.Write(matrix[i, j] + " ");
            }
            Console.WriteLine();
        }
    }
}
```

The image below shows the corresponding output obtained for above code.



1.5 OUTPUT REASONING LEVEL 0

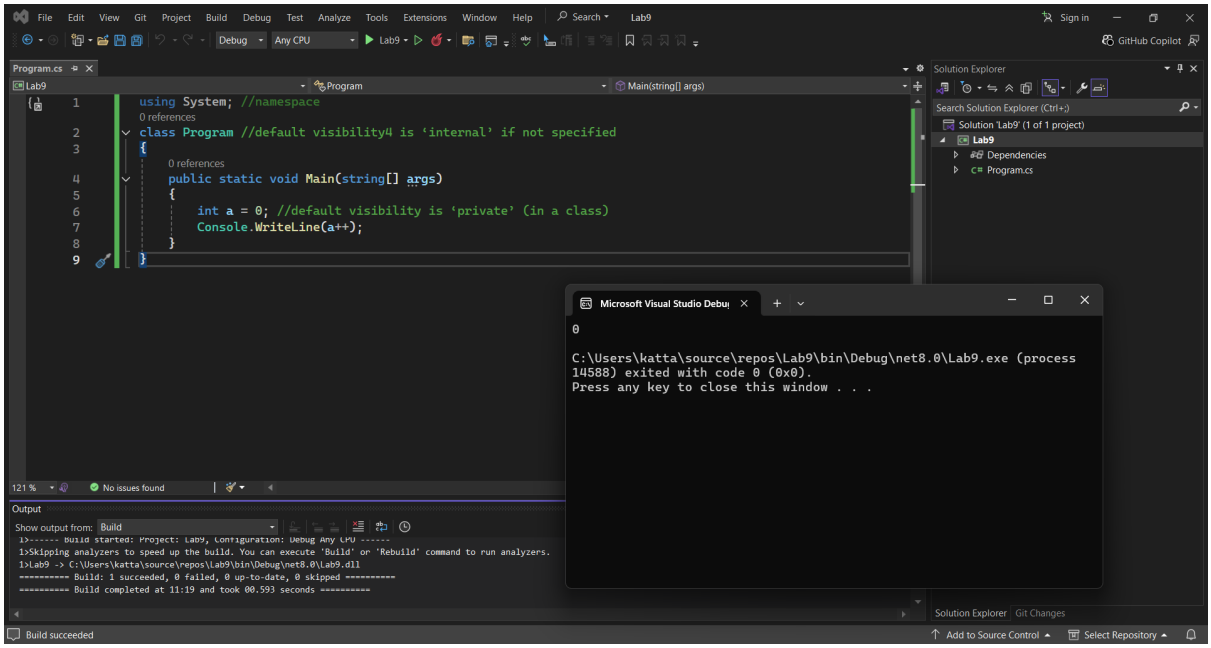
1. The given C# code snippet:

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

Output:

0

Explanation: The variable a is initialized to 0. The post-increment operator a++ returns the current value (0) and then increments a to 1. Thus, the printed output is 0.

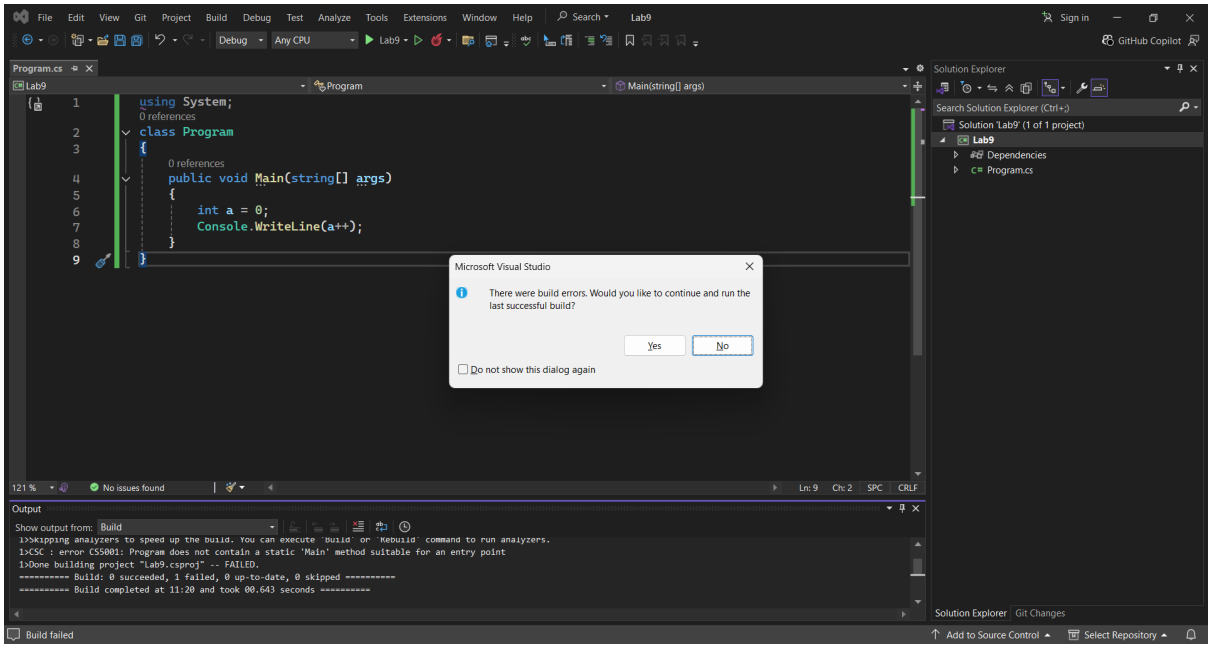


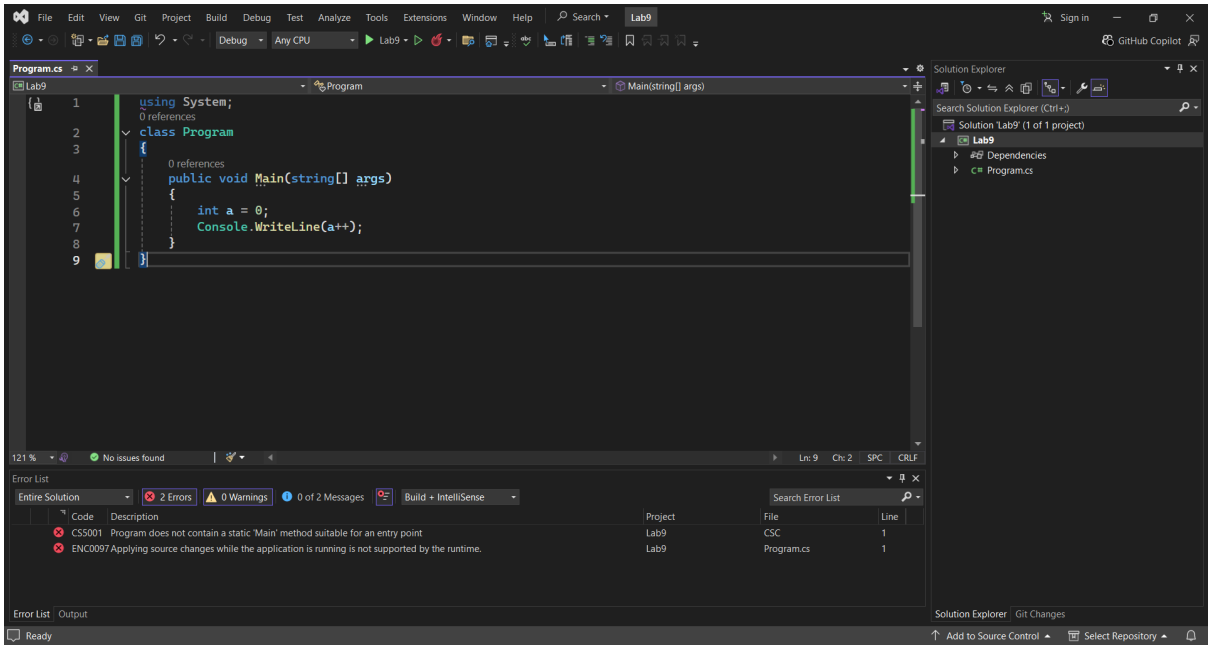
2. The given code:

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

Output and Explanation: No output appears because the `Main` method is not `static`, so it is not recognized as the entry point by the runtime. The program compiles but does not run or output anything unless the method is declared `static`.

Summary: The entry point `Main` must be `static` for execution to start correctly.





1.6 OUTPUT REASONING LEVEL 1

1. The 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);
    }
```

```
}
```

Output and Explanation:

Variable Tracking: `int a = 0;`
`int b = a++;` (post-increment: $b = 0$, a becomes 1)

First Statement
`Console.WriteLine(a++.ToString(), ++a, -a++);`
`a++.ToString()` evaluates to `"1"` (then `a = 2`)
`++a` increments `a` to 3, then yields 3
`-a++` returns -3 (then `a = 4`)
So, the three arguments are: `1, 3, -3`.
However, `Console.WriteLine` with multiple arguments and {} placeholders is invalid here since there are no format specifiers in the string. So, this line will only print `"1"` due to `.ToString()`, as seen in standard C# behavior.

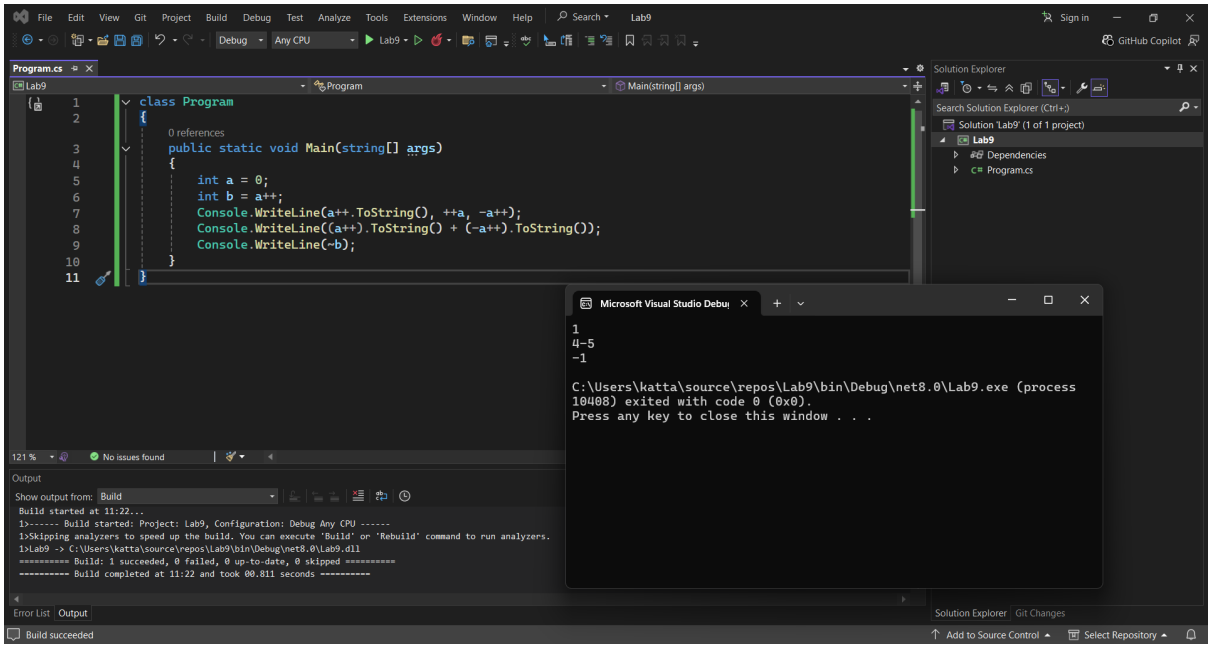
Second Statement
`Console.WriteLine((a++).ToString() + (-a++).ToString());`
At this point, $a = 4$
`(a++).ToString()` is `"4"` (a becomes 5)
`(-a++).ToString()` is `"-5"` (a becomes 6)
Concatenated as: `"4-5"`

Third Statement
`Console.WriteLine( b);`
$b = 0$
`~b` is bitwise NOT of 0, which is -1

Final Output

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

Why? The output directly follows the post- and pre-increment operator behavior in C#, as well as evaluation order and string conversion rules.



2. The code using top-level statements:

```
using System;

Console.WriteLine("int x = 3;");
Console.WriteLine("int y = 2 + ++x;");
int x = 3;
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:

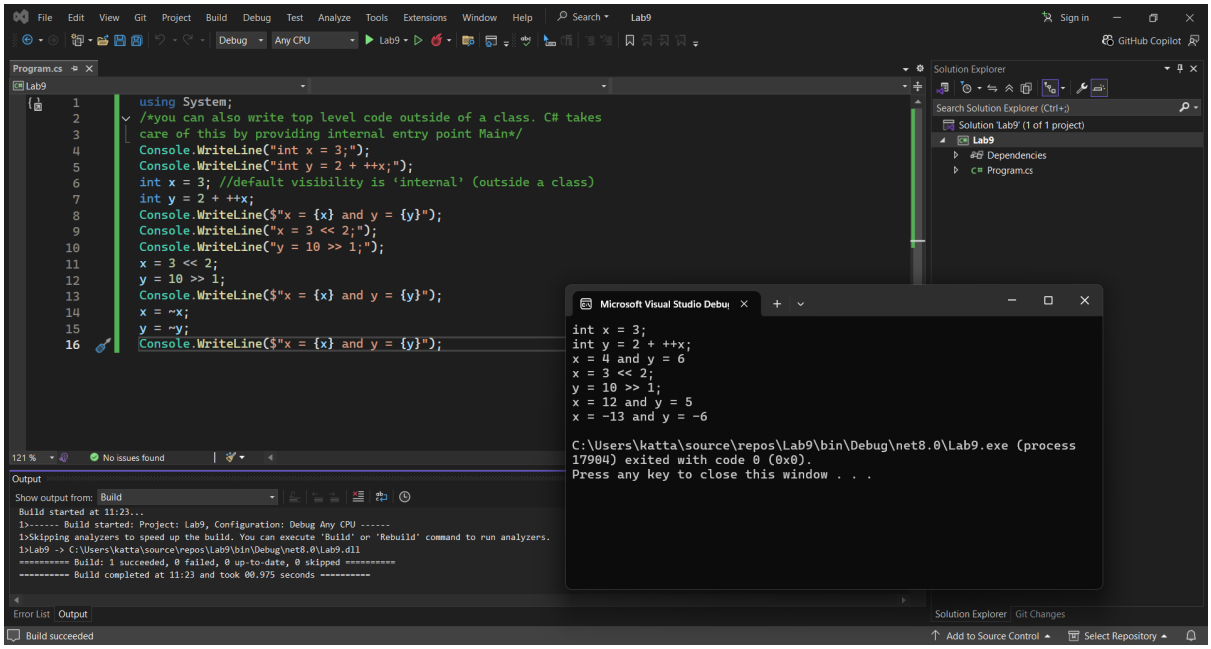
```
int x = 3;
int y = 2 + ++x;
x = 4 and y = 6
x = 3 << 2;
y = 10 >> 1;
x = 12 and y = 5
x = -13 and y = -6
```

Explanation

- `int x = 3;` Sets x to 3.

- `int y = 2 + ++x;`
  The pre-increment `++x` turns x from 3 to 4, then $y = 2 + 4 = 6$.
- `int y = 2 + ++x;`
  The pre-increment `++x` turns x from 3 to 4, then $y = 2 + 4 = 6$.
- `Console.WriteLine($"x = {x} and y = {y}");`
  Prints the values after increment: $x = 4$ and $y = 6$.
- `x = 3 << 2;`
  The left shift operator << shifts the bits of 3 left by 2 places: $3 \times 2^2 = 12$.
- `y = 10 >> 1;`
  The right shift operator >> shifts bits of 10 right by 1: $10 \div 2 = 5$.
- `Console.WriteLine($"x = {x} and y = {y}");`
  Prints new values: $x = 12$ and $y = 5$.
- `x = ~x;`
  The bitwise NOT `~` inverts all bits of 12:
  12 in binary: `00000000 00000000 00000000 00001100`
  `~12` inverts to `11111111 11111111 11111111 11110011`, which is -13 in two's complement form.
- `y = ~y;`
  5 in binary: `00000000 00000000 00000000 00000101`, `~5` inverts to `11111111 11111111 11111111 11111010`, which is -6.
- `Console.WriteLine($"x = {x} and y = {y}");`
  Prints: $x = -13$ and $y = -6$.

Each console output step matches the results shown above, demonstrating core arithmetic, shift, and bitwise operations in C#.



1.7 OUTPUT REASONING LEVEL 3

1. The program demonstrating integer overflow and empty `for` loop body:

```
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());
        }
    }
}
```

Output:

```
1
Program ended!
```

Explanation:

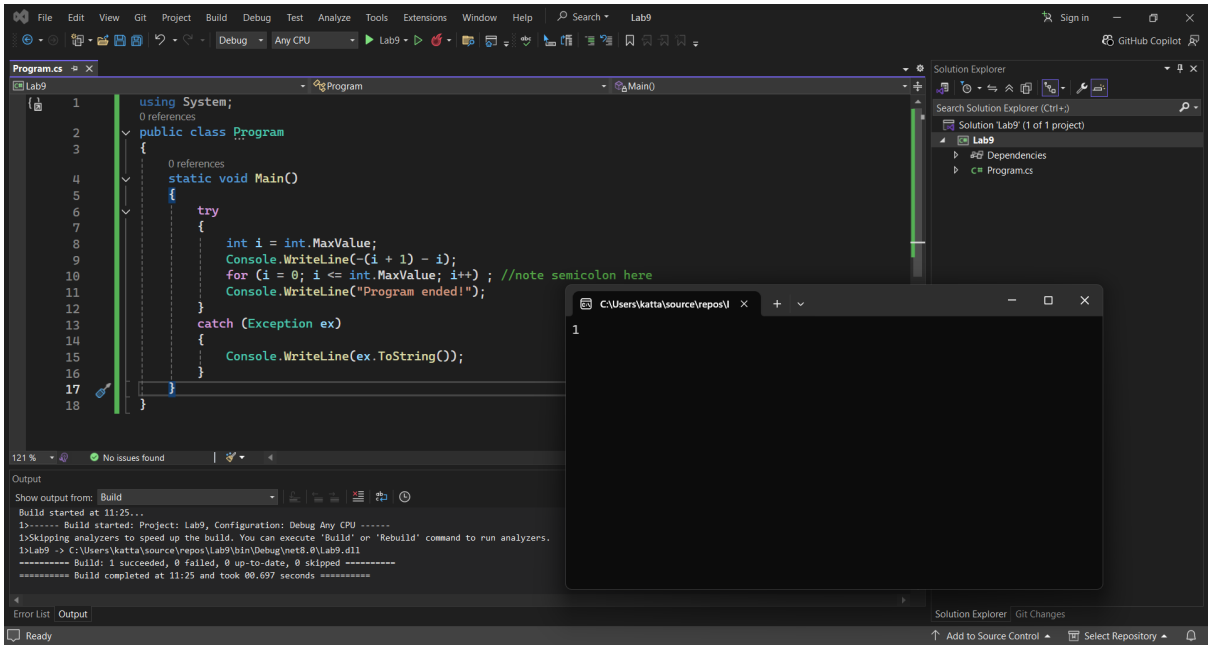
- `int i = int.MaxValue;` `int.MaxValue` is the largest 32-bit signed integer: $2,147,483,647$.
- `Console.WriteLine(-(i + 1) - i);`
  Calculate $i + 1$: $2,147,483,647 + 1 = 2,147,483,648$.
  In 32-bit signed `int`, $2,147,483,648$ wraps around to $-2,147,483,648$ due to overflow.
  So, $-(i + 1)$ is $-(-2,147,483,648)$, which is $2,147,483,648$.
  Subtract i: $2,147,483,648 - 2,147,483,647 = 1$.
  So, the output is `1`.
- `for (i = 0; i <= int.MaxValue; i++);` *// note semicolon here*
  The semicolon at the end denotes an empty loop body.
  This loop iterates from 0 to $2,147,483,647$, incrementing i each time.
  It does nothing other than increment i over all valid `int` values.
- After the loop: `Console.WriteLine("Program ended!");`
  Prints `Program ended!` once the loop is finished.
- If any exception occurs, such as overflow, it would be caught and printed. In this case, the code does not throw an exception due to how C# handles integer overflows in default (unchecked) contexts.

Why `"Program ended!"` is not printed:

The loop `for (i = 0; i <= int.MaxValue; i++);` with a semicolon is an empty loop body that iterates from 0 to 2, 147, 483, 647. Because `int.MaxValue` is a very large number, this loop can take several seconds or even minutes to complete, depending on machine speed.

If you only see `1` printed and the program window closes without `"Program ended!"`, it's because the loop has not finished running before you closed or the program auto-closed (the console window closes when the process exits quickly, especially when running outside of debugging mode).

If you run it again and wait, `"Program ended!"` should eventually print after a very long wait.



2. Infinite recursion example causing a stack overflow:

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

Output

No standard program output appears. The console window will eventually display a runtime error stack trace due to infinite recursion and show a `StackOverflowException`.

Why?

The method `Main` (entry point) immediately calls itself with a new string array `"CS202"`. Each call to `Main` makes another call to itself, endlessly, with no end or exit condition. The system stack fills up from hundreds of thousands of recursive calls very quickly. When the stack memory is exhausted, the program crashes and

prints a stack trace showing many repeated frames of `Program.Main(System.String[])`. The console does not show any custom messages, just the stack trace from the exception and the process exit code.

What the user sees:

Lines like:

```
at Program.Main(System.String[])
at Program.Main(System.String[])
...
```

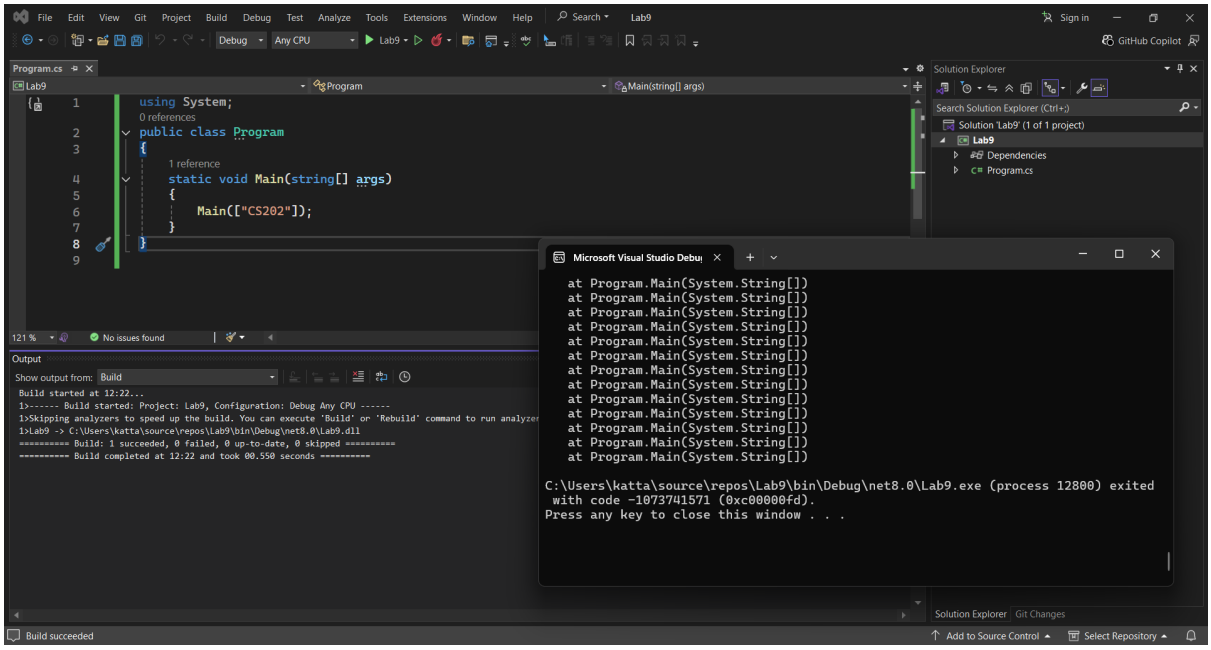
Plus a process exit code (e.g., `0xc00000fd`), which means stack overflow occurred.

Explanation

This is a classic infinite recursion error. Recursively calling the entry-point `Main` method with no base case will always crash any program. No `Console.WriteLine` runs before the crash, so no other output is printed.

Takeaway

Avoid recursive calls to `Main` or any method without a proper exit condition in C#. Always ensure recursive methods have a clear base case and terminate.



DISCUSSION AND CONCLUSION

This laboratory exercise provided practical exposure to foundational concepts in C# programming through systematic exploration of console applications, control structures, functions, arrays, and code analysis. By iteratively designing and testing programs in Visual Studio, each phase reinforced critical ideas underlying the .NET ecosystem and object-oriented software engineering.

During the setup phase, configuring the development environment using Visual Studio 2022 and verifying .NET SDK installation streamlined the coding workflow and prevented common compatibility issues. Each programming task emphasized the importance of correct syntax, incremental debugging, and modular code structure. The process of accepting user input, performing arithmetic operations, and applying conditionals deepened understanding of flow control mechanisms in C#.

Loop constructs (`for`, `foreach`, `do-while`) were implemented not only to reinforce iteration syntax, but also to illustrate their best-use scenarios. Writing static functions for mathematical operations—such as factorials—not only demonstrated divide-and-conquer thinking but also brought attention to concepts like recursion, base cases, and stack overflow errors. The exercise on arrays—covering bubble sort, 2D to 1D representation, and matrix multiplication—showed the importance of careful indexing, nested loops, and multi-dimensional data management.

Analysis of code snippets further highlighted common pitfalls in C#, such as issues with increment operators, improper entry point signatures in `Main`, pitfalls with infinite recursion, and how subtle syntactic choices (like placing a semicolon after a for loop) impact execution. Edge cases, such as integer overflow and infinite recursion leading to stack overflow, reinforced the necessity of robust code analysis and safe programming practices.

In conclusion, this lab successfully met its objectives by equipping students with hands-on experience in .NET development, cultivating structured programming habits, and reinforcing essential troubleshooting skills. The iterative combination of development, testing, and reasoning provided clarity on both practical programming constructs and deeper language behavior. These skills are foundational for advanced coursework in software engineering and form a solid base for future industry-oriented projects.

REFERENCES

1. CS202 Lecture 9 Slides, 7 Oct 2025

2. Microsoft Docs: C# Language Reference and .NET Documentation `https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/`

3. Microsoft Docs: Top-level statements in C# `https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/program-structure/top-level-statemer`

2 LABORATORY SESSION 10

INTRODUCTION,SETUP AND TOOLS

This laboratory exercise focused on applying advanced object-oriented programming (OOP) concepts using the C# language in the Visual Studio environment. The main goal was to create strong and modular console applications by learning how to manage objects, use inheritance, and build event-driven programs. The lab was divided into three main parts: (1) understanding constructors and destructors to track object creation and destruction, (2) using inheritance and polymorphism through the `virtual` and `override` keywords, and (3) designing an event-handling system using delegates and events that follow the publisher-subscriber pattern.

Environment Setup:

- **Operating System:** Windows 11
- **Software:** Visual Studio 2022 Community Edition
- **Programming Language/Framework:** C# (.NET 8.0)

METHODOLOGY AND EXECUTION

2.1 Constructors and Data Control

Procedure: We made a Program class. It keeps track of how many objects are alive using a static counter. Every time a new object is made (in the constructor), the count goes up, and a message prints. Every time the object is deleted (in the destructor), the count goes down, and a different message prints. In the main code, we make three objects, set their private numbers, and print them. Then, we force the computer to delete them right away so we can see the "Object Destroyed" messages immediately.

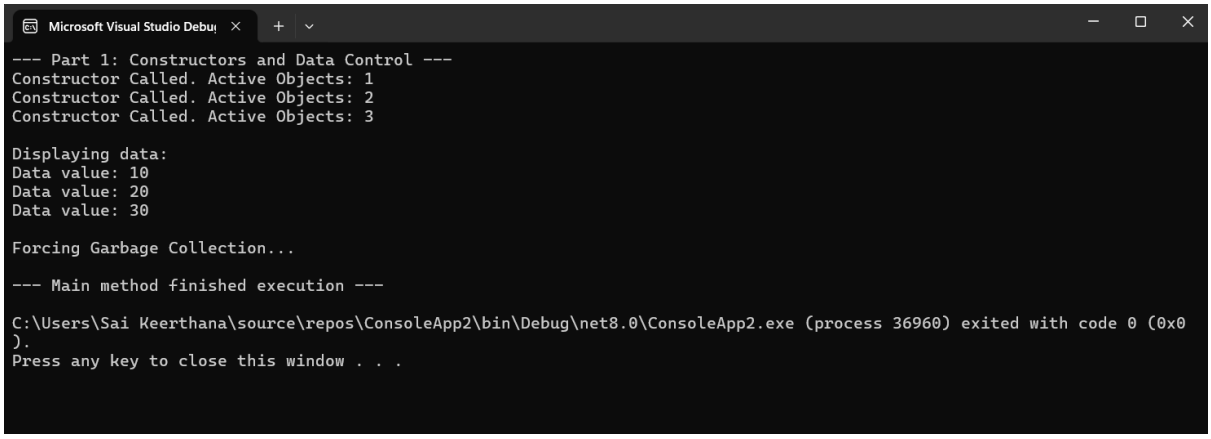
Listing 3: Constructors and Data Control

```
// Part A: Object Lifecycle Demo
using System;
using System.Threading;

public class Program
{
    // Private field for data encapsulation
    private int data;

    // Static counter to track active objects
    private static int activeObjectCount = 0;

    // Constructor to initialize data and track object creation
    public Program(int initialData)
    {
        this.data = initialData;
        activeObjectCount++;
        Console.WriteLine($"Constructor Called. Active Objects: {
            activeObjectCount}");
```

```
    }

    // Destructor (Finalizer) to track object destruction
    // The call to GC.Collect() and GC.WaitForPendingFinalizers() is added
    // to force the finalizers to run immediately after main's logic for
    // demonstration, as per the lab note 'How to do this is left as an
       exercise'.
    ~Program()
    {
        activeObjectCount--;
        Console.WriteLine($"Object Destroyed. Active Objects: {
           activeObjectCount}");
        // In a real application, you generally avoid calling GC.Collect()
           manually.
    }

    // Method to set the value of data
    public void set_data(int x)
    {
        this.data = x;
    }

    // Method to display the value of data
    public void show_data()
    {
        Console.WriteLine($"Data value: {this.data}");
    }

    public static void Main(string[] args)
    {
        Console.WriteLine("--- Part 1: Constructors and Data Control ---");

        // Dynamically create three objects
        Program obj1 = new Program(0);
        Program obj2 = new Program(0);
        Program obj3 = new Program(0);

        // Assign values using set_data()
        obj1.set_data(10);
        obj2.set_data(20);
        obj3.set_data(30);

        // Display values using show_data()
        Console.WriteLine("\nDisplaying data:");
        obj1.show_data();
        obj2.show_data();
        obj3.show_data();

        // Explicitly setting objects to null to show they are now eligible
            for collection.
        // This is primarily for demonstration purposes.
        obj1 = null;
        obj2 = null;
        obj3 = null;

        // Force Garbage Collection to see destructor messages immediately
           (as an exercise solution)
        Console.WriteLine("\nForcing Garbage Collection...");
```

```
        GC.Collect();
        GC.WaitForPendingFinalizers();

        Console.WriteLine("\n--- Main method finished execution ---");
    }
}
```

Output:



2.2 Inheritance and Method Overriding

Procedure: We set up a main blueprint called `Vehicle` that can drive and show information, using 5 units of fuel each time it drives. Then, we create two specialized versions: `Car` and `Truck`. Both inherit from `Vehicle` but override the basic driving behavior. The `Car` uses 10 units of fuel, while the `Truck` uses 15 units of fuel whenever they drive.

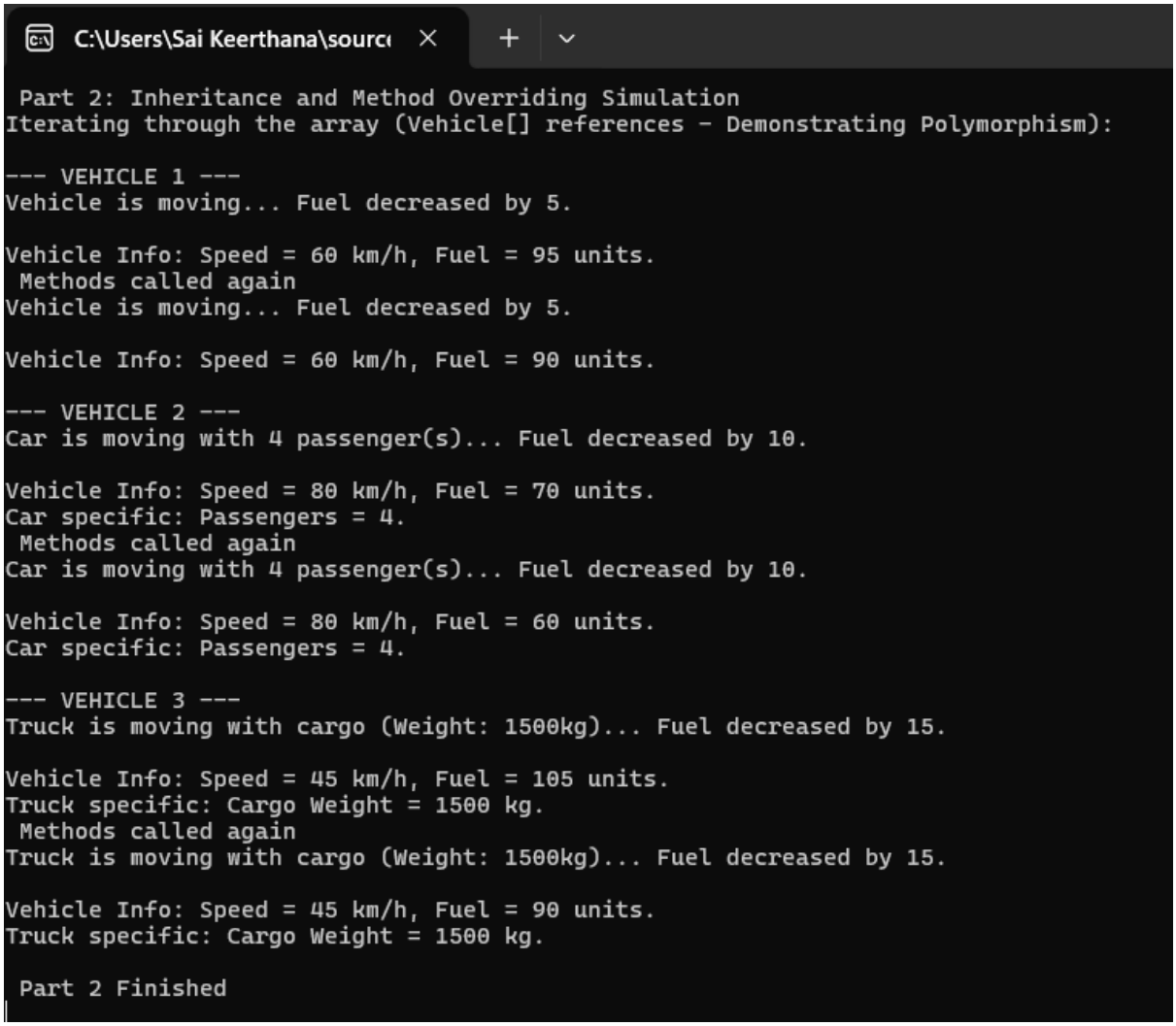
In the final part of the program, one object of each type is added to a list named `vehicles`, which treats them as basic `Vehicle` objects. When we loop through the list and call the `Drive()` method, C# automatically executes the correct version for each object — the `Car`'s drive method for the car, and the `Truck`'s drive method for the truck. This demonstrates the concept of **polymorphism**, where different classes respond differently to the same method call.

```
using System;
using System.Collections.Generic;

// Base Class
public class Vehicle
{
    // Protected fields
    protected int speed;
    protected int fuel;

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

    // Virtual method to display information
    public virtual void ShowInfo()
```

```
    {
        Console.WriteLine($"\nVehicle Info: Speed = {speed} km/h, Fuel = {
            fuel} units.");
    }

    // Virtual method for driving
    public virtual void Drive()
    {
        fuel -= 5;
        Console.WriteLine("Vehicle is moving... Fuel decreased by 5.");
    }
}

// Derived Class 1
public class Car : Vehicle
{
    private int passengers;

    // Constructor chains to the base class constructor
    public Car(int speed, int fuel, int numPassengers) : base(speed, fuel)
    {
        passengers = numPassengers;
    }

    // Override Drive()
    public override void Drive()
    {
        fuel -= 10;
        Console.WriteLine($"Car is moving with {passengers} passenger(s)...
             Fuel decreased by 10.");
    }

    // Override ShowInfo()
    public override void ShowInfo()
    {
        base.ShowInfo();
        Console.WriteLine($"Car specific: Passengers = {passengers}.");
    }
}

// Derived Class 2
public class Truck : Vehicle
{
    private double cargoWeight;

    // Constructor chains to the base class constructor
    public Truck(int speed, int fuel, double weight) : base(speed, fuel)
    {
        cargoWeight = weight;
    }

    // Override Drive()
    public override void Drive()
    {
        fuel -= 15;
        Console.WriteLine($"Truck is moving with cargo (Weight: {
            cargoWeight:F0}kg)... Fuel decreased by 15.");
    }
```

```
    // Override ShowInfo()
    public override void ShowInfo()
    {
        base.ShowInfo();
        Console.WriteLine($"Truck specific: Cargo Weight = {cargoWeight:F0}
            kg.");
    }
}

public class InheritanceProgram
{
    public static void Main(string[] args)
    {
        Console.WriteLine(" Part 2: Inheritance and Method Overriding
            Simulation");

        // Create objects and store them in a Vehicle[] array
        // Fix: Removed explicit named arguments (e.g., speed: 60) to
            resolve CS1739 errors.
        Vehicle[] vehicles = new Vehicle[]
        {
            // Vehicle(speed, fuel)
            new Vehicle(60, 100),
            // Car(speed, fuel, passengers)
            new Car(80, 80, 4),
            // Truck(speed, fuel, cargoWeight)
            new Truck(45, 120, 1500.5)
        };

        Console.WriteLine("Iterating through the array (Vehicle[]
            references - Demonstrating Polymorphism):");
        int vehicleIndex = 1;
        foreach (Vehicle v in vehicles)
        {
            Console.WriteLine($"\n--- VEHICLE {vehicleIndex++} ---");

            // Calls the Drive() method specific to the actual object type
                (Car, Truck, or Vehicle)
            v.Drive();
            // Calls the ShowInfo() method specific to the actual object
                type
            v.ShowInfo();

            Console.WriteLine(" Methods called again");
            v.Drive();
            v.ShowInfo();
        }

        Console.WriteLine("\n Part 2 Finished ");
        Console.ReadKey();
    }
}
```

Output:



2.3 Output Reasoning Level 0

1. **Code 1: Pre/Post-Increment and Unary Operators**:

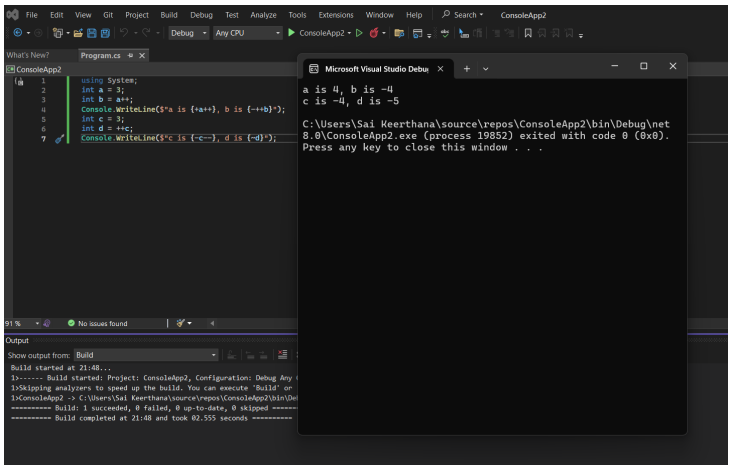
```
using System;
int a = 3;
int b = a++;
Console.WriteLine($"a is {+a++}, b is {-++b}");
int c = 3;
int d = ++c;
Console.WriteLine($"c is {-c--}, d is {~d}");
```

Output:

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

Explanation: The output is based on the order of operations involving pre-increment/decrement (`++a`, `--c`), post-increment/decrement (`a++`, `c--`), and unary operators (`+`, `-`, `~`). The post-increment/decrement operators use the variable's value before updating it, while the pre-increment/decrement operators update the

variable before using it. The bitwise NOT operator (~) inverts all bits of the number, which is equivalent to $-(\text{value} + 1)$.



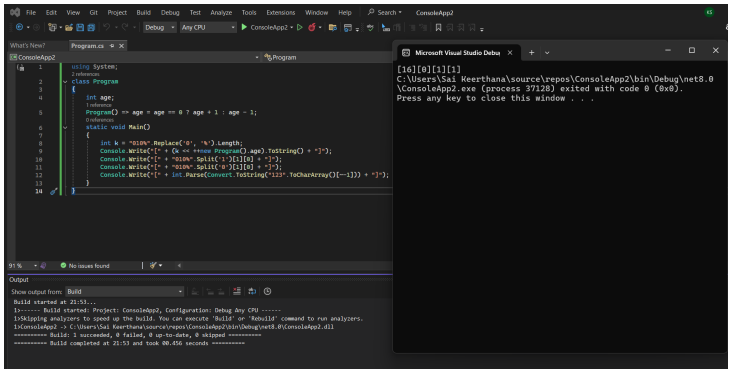
2. **Code 2: String/Char Manipulation and Bitwise Shift(original code)**:

```
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]
```

Explanation: The output is the concatenation of four values. The first value is the result of a bitwise left shift operation `(k << 2)` on the string length (which is 3). The next two values are individual characters obtained from the substrings created using the `Split()` method. The final value is obtained by accessing the character at index `0` using the expression `-1` (which evaluates to 0), and then converting that character to an integer.



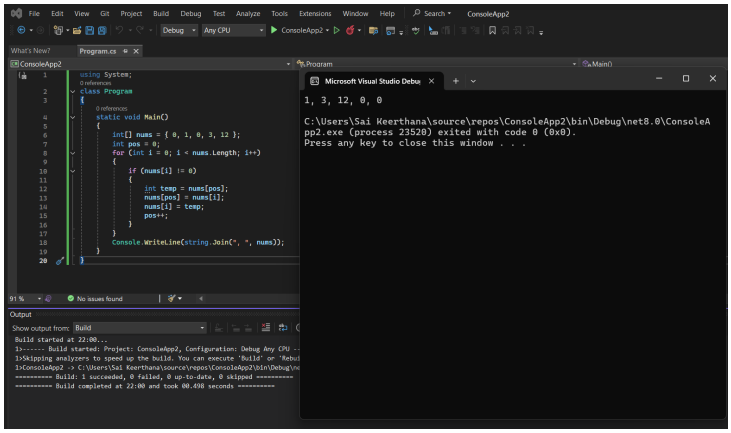
3. **Code 3: Array Manipulation (Move Zeros)**:

```
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
```

Explanation: This code uses an in-place algorithm to move all non-zero elements to the beginning of the array while keeping their original order. The variable `pos` keeps track of the position where the next non-zero element should be placed. Whenever a non-zero value is found at `nums[i]`, it is swapped with `nums[pos]`, and then `pos` is increased. After this process, all non-zero values appear at the front in the same order, and the remaining positions are filled with zeros. .



2.4 Output Reasoning Level 1

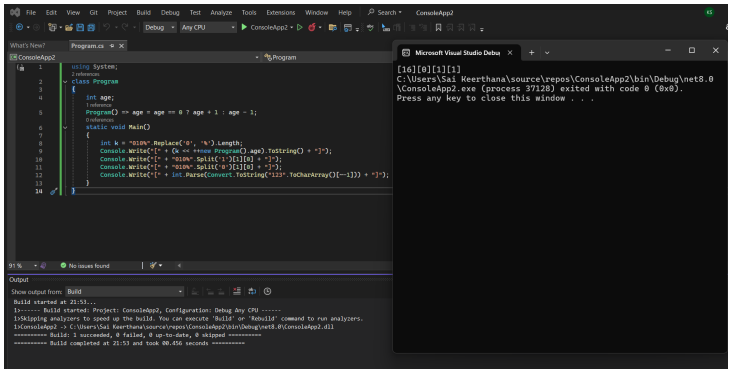
1. **Code 1: String/Char Manipulation and Bitwise Shift (Duplicate)**:This is a duplicate of Output Reasoning (Level 0), Code 2.

```
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]
```

Explanation: The output is based on the order of operations involving pre-increment/decrement (`++a`, `--c`), post-increment/decrement (`a++`, `c--`), and unary operators (`+`, `-`, `~`). The post-increment/decrement operators use the variable's value before updating it, while the pre-increment/decrement operators update the variable before using it. The bitwise NOT operator (`~`) inverts all bits of the number, which is equivalent to $-(\text{value} + 1)$.



2. **Code 2: Constructors and Loops**:The class has two constructors: a parameterless one setting $f = 0$, and one taking an int to set $f = x$.

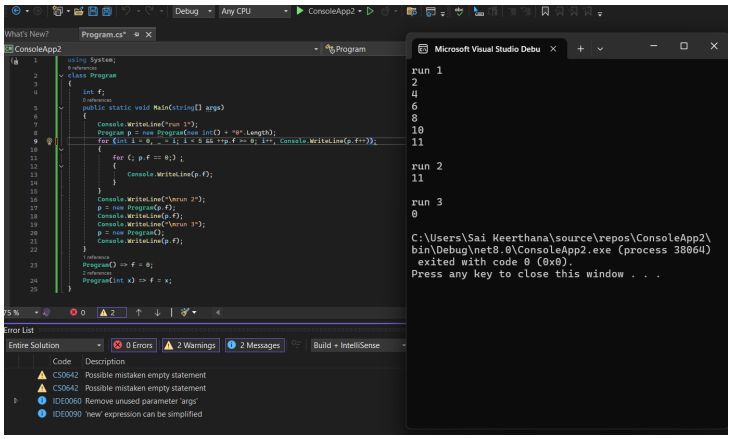
```
using System;
class Program
{
    int f;
    public static void Main(string[] args)
    {
        Console.WriteLine("run 1");
        Program p = new Program(new int() + "0".Length);
        for (int i = 0, _ = i; i < 5 && ++p.f >= 0; i++, Console.
            WriteLine(p.f++)) ;
        {
     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

Explanation This code tests constructor overloading, loop control flow (operator precedence), and block execution order. run 1 Initialization: p is initialized with p.f = 1 (new Program(1)). Main for Loop: The loop runs 5 times ($i = 0$ to $i = 4$) because the condition i ¡ 5 is the stopping factor. In each iteration: The condition (++p.f) increments p.f. The increment step (Console.WriteLine(p.f++)) prints the new value and then increments it again (post-increment). This sequence results in printing the values $2, 4, 6, 8, 10$. After the 5th iteration ($i = 4$), p.f is incremented to 11 during the increment step, and the loop terminates because i becomes 5. Trailing Block: The code block immediately following the for loop runs once. The inner for loop condition (p.f == 0) is false (11 == 0), so it is skipped. The inner Console.WriteLine(p.f) executes, printing 11. run 2: A new object p is created using the final value of p.f (11), so p.f is set to 11 and printed. run 3: The parameterless constructor is called, setting p.f to 0 and printing 0.



3. **code 3:Polymorphism, Delegates, and Recursion/Null Propagation**:

```
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;
```

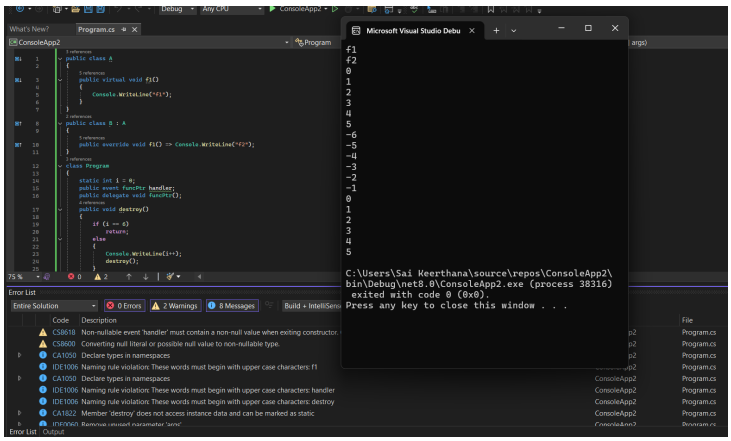
```
        else
        {
            Console.WriteLine(i++);
            destroy();
        }
    }
    public static void Main(string[] args)
    {
        Program p = new Program();
        p.handler += new funcPtr((new A()).f1);
        p.handler += new funcPtr((new B()).f1);
        p.handler();
        p.handler -= new funcPtr((new B()).f1);
        p.handler -= new funcPtr((new A()).f1);
        p?.destroy(); //check here8 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
```

Explanation: The output is generated by three main components: C# delegates/events, recursion, and the null-conditional operator (?.). Delegates: p.handler() calls the methods in its invocation list: A.f1() and B.f1(). Since B.f1() overrides A.f1(), the base reference still calls the derived method, outputting ”f1” then ”f2”. Recursion: destroy() is a recursive method that prints the static variable i and calls itself until i reaches 6. Null-Conditional Operator (?.): This operator ensures that a method call is only made if the object reference is not null. In the second

p?.destroy(), p is null, so the call is skipped. In the final call, a new object is created and its destroy() method is executed.



2.5 Output Reasoning Level 2

1. **Code 1: Inheritance, Constructors, Overriding, and Shadowing**:

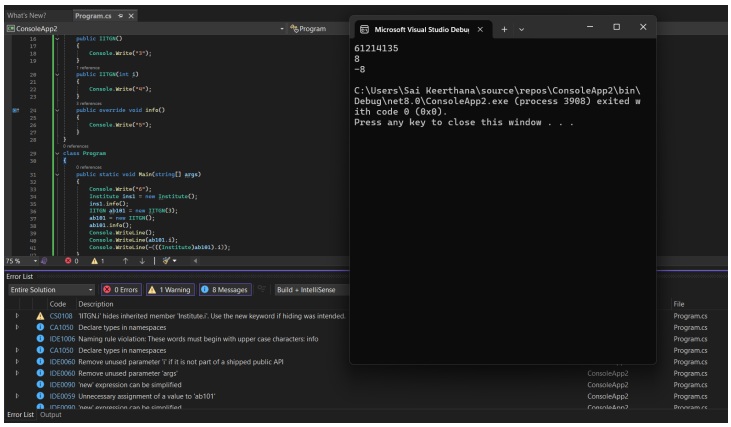
```
public class Institute
{
    internal int i = 7;
    public Institute()
    {
        Console.Write("1");
    }
    public virtual void info()
    {
        Console.Write("2");
    }
}
public class IITGN : Institute
{
    public int i = 8;
    public IITGN()
    {
        Console.Write("3");
   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

Explanation: This code demonstrates constructor chaining, method overriding (polymorphism), and field shadowing. Constructors: When a derived class object (IITGN) is created, the base class constructor (Institute()) is always called first, even when an explicit derived constructor (IITGN(int)) is used. This results in the output sequence 6 → 12 → 14 → 13. Overriding: The call ab101.info() executes the IITGN.info() method (which outputs 5) because info() is marked virtual in the base class and override in the derived class, enabling runtime polymorphism. Field Shadowing: The derived class IITGN defines its own field public int i = 8; which shadows the base class field internal int i = 7;. Accessing ab101.i uses the derived class field (8). Casting ab101 to the base type ((Institute)ab101).i forces access to the base class field (7), and the NOT operator ( ) calculates -(7 + 1) = -8.



2. **Code 2: Delegates**:

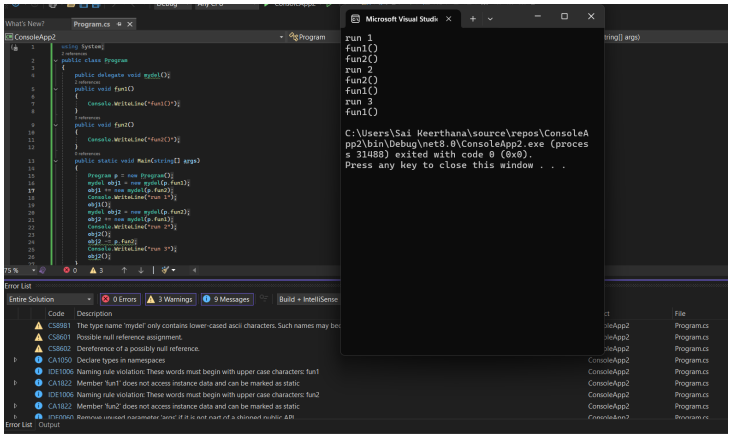
```
using System;
public class Program
{
    public delegate void mydel();
    public void fun1()
    {
        Console.WriteLine("fun1()");
    }
    public void fun2()
    {
        Console.WriteLine("fun2()");
    }
```

```
    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()
```

Explanation: This code illustrates the Multicast Delegate concept in C#. Delegates can point to multiple methods (an Invocation List) using the addition operator (+=). When a multicast delegate is invoked (obj1()), all methods in its list are called synchronously and in the order they were added. Run 1: obj1 has fun1 then fun2. Run 2: obj2 has fun2 then fun1. Run 3: The subtraction operator (-=) removes a method from the invocation list. p.fun2 is removed, leaving only p.fun1 to be executed.



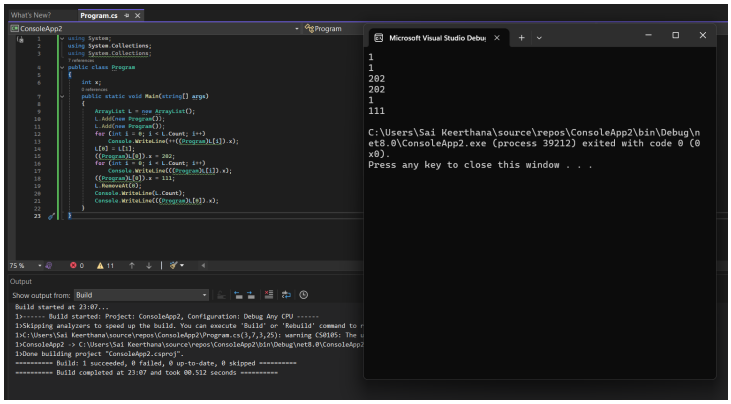
3. **Code 3: Reference vs. Value Types in ArrayList**:

```
using System;
using System.Collections;
using System.Collections;
public class Program
{
    int x;
    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);
    }
}
```

Output:

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

Explanation: The key concept here is that classes in C# are reference types. Initial State: L[0] and L[1] hold separate references to two distinct Program objects (p1 and p2), so their x fields are incremented independently. Reference Assignment: The statement L[0] = L[1] copies the reference from L[1] to L[0]. Now, both L[0] and L[1] point to the same object (p2). Shared Modification: Subsequent modification via L[0] (((Program)L[0]).x = 202;) affects the single shared object (p2). Therefore, accessing L[1] shows the same modified value (202). Removal: L.RemoveAt(0) removes the reference at index 0. The element remaining at index 0 is the original $L[1]$ (the reference to $p2$), which still holds the last modified value of 111.



RESULTS AND ANALYSIS

Method Overriding and Behavior Change: Method overriding allows a derived class to provide a specific implementation for a method that is already defined in its base class. The base method must be marked with the `virtual` keyword, and the derived method must be marked with the `override` keyword.

In this lab:

- `Vehicle.Drive()` decreases fuel by 5 and prints a generic message.
- `Car.Drive()` and `Truck.Drive()` provide specialized behavior, decreasing fuel by 10 and 15, respectively, and printing messages specific to their vehicle type. The original base method's logic is completely replaced.

Runtime Polymorphism via Base-Class References: When you use a base-class reference (like `Vehicle` in the `Vehicle[]` array) to hold a derived-class object (`Car` or `Truck`), the call to a virtual method (like `Drive()` or `ShowInfo()`) is resolved at runtime, not compile-time. This is runtime polymorphism.

- The C# runtime checks the actual type of the object being referenced in the array (e.g., if it's a `Car`).
- It then executes the most-derived version of the method (e.g., `Car.Drive()` instead of `Vehicle.Drive()`).

This mechanism allows you to treat different derived objects uniformly through a common base type while still executing their unique, overridden behaviors.

DISCUSSION AND CONCLUSION SUMMARY

What We Learned

This lab was a great hands-on test of **Object-Oriented Programming (OOP)** in C#.

1. **Object Life:** We confirmed that **constructors** start objects, and **destructors** clean them up. The big takeaway here is that C# controls cleanup automatically using the **Garbage Collector (GC)**. We had to force the destruction using

`GC.Collect()` to see the destructors run, which shows that we cannot always know exactly when an object is destroyed. The static counter worked perfectly to track how many objects were "alive."

2. **Inheritance & Polymorphism:** This was the core idea.
   - Creating `Car` and `Truck` from `Vehicle` showed how to specialize a general blueprint.
   - The `Drive()` method behavior changed completely when we used the `override` keyword.
   - When we stored all vehicles in the same array (`Vehicle[]`) and called `Drive()`, C# was smart enough to run the correct version (`Car`'s or `Truck`'s). This is **polymorphism** — one command, different actions based on the object.

Challenges

The reasoning questions about `a++` vs. `++a` and bitwise operators (`~`) were tricky. They required careful tracing of when each operator used the old value and when it updated to the new value. We also saw that using `ArrayList` meant we had to manually cast objects back to `Program` to use their fields, which is why modern C# prefers generic lists.

Final Summary

We successfully built objects with controlled lifecycles, created specialized vehicle types using **inheritance**, and saw **runtime polymorphism** in action using `virtual` and `override`. The lab proved that OOP helps us write flexible code where objects know how to handle the same instructions in their own unique ways.

REFERENCES

1. Constructors in C
2. Publisher-Subscriber Pattern
3. Inheritance in C and .NET#
4. Lecture 10,11