

Lab Report 9: Module Dependency and Cohesion Analysis

Kushal Rathod
22110128

CS202: Software Tools and Techniques for CSE

March 2025

1 Introduction and Objectives

This lab focuses on analyzing software dependencies and cohesion using `pydeps` for Python projects and `LCOM` metrics for Java projects. The objective is to assess module dependencies, detect cyclic dependencies, measure cohesion, and evaluate design flaws in real-world software.

2 Environment Setup and Tools Used

- **Operating System:** MacOS
- **Python Version:** 3.10.7
- **Java Version:** 11
- **Software Tool:** `pydeps`, `LCOM.jar`
- **Python Repository:** Alphafold
- **Java Repository:** Lombok
- **Installation**

```
1 pip install pydeps
```

```
1 java -jar LCOM.jar -i <input_path> -o <output_path>
```

3 Methodology

3.1 Repository Selection and Criteria

- The selection was based on the following criteria:
 - Number of GitHub stars: Over 1,000
 - Number of commits: Over 500
 - Active maintenance by contributors
 - Should be a real world software project
- Selected a real-world Python project with multiple modules.
- Navigated to the project directory and generated the dependency graph using:

```
1 pydeps <project_folder> --show-deps
```

- Parsed the generated JSON output to analyze module dependencies, fan-in, and fan-out.
- Selected a Java project containing at least 10 classes.
- Ran LCOM.jar to compute cohesion metrics for each class.
- Identified classes with high LCOM values and analyzed potential refactoring strategies.

4 Execution

4.1 Aphafold - Python

- We used the following command to create the JSON file:

```
1 pydeps run_alphafold.py --show-deps -Tpng -o  
  ./../alphafold.png > ./../alphafold.json  
  --max-module-depth=2
```

- The resulting JSON file is submitted in the .zip file.

4.2 Lombok - Java

- We used the following command to create the TypeMetrics.csv file:

```
1 java -jar LCOM.jar -i
    /Users/kushalrathod/Desktop/Projects/STT_lab9/lom
    bok/src -o .
```

- The resulting file is submitted in the .zip file.

5 Results and Analysis

5.1 Alphafold - Python

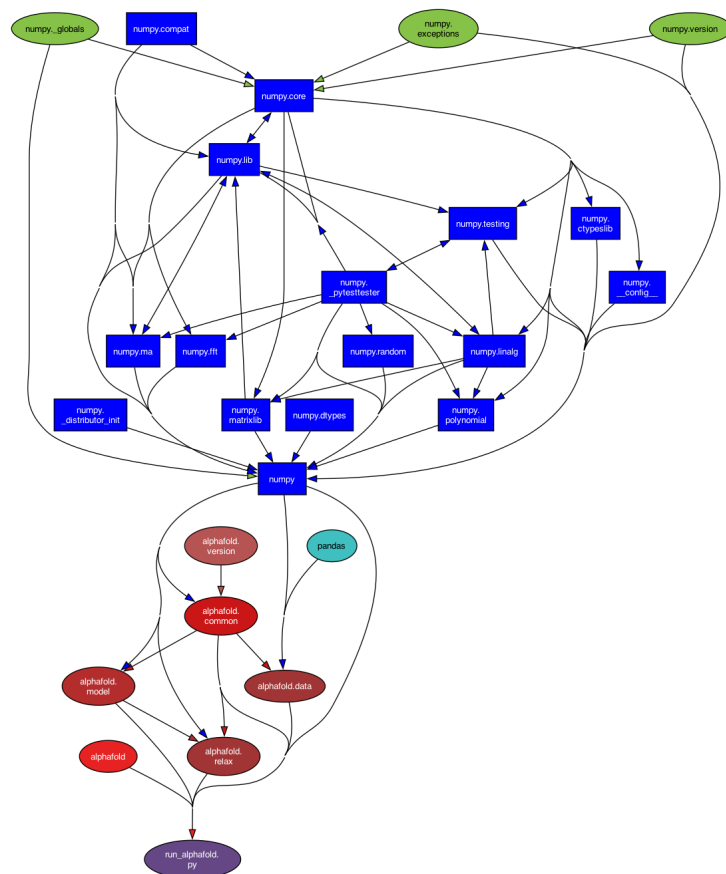


Figure 1: Dependency Graph of JSON file

- Identify highly coupled modules and their dependencies.

- Module: numpy
- Module: numpy.core
- Module: numpy.lib
- Module: numpy._pytesttester
- Module: numpy.linalg
- Detect cyclic dependencies (few examples, rest are given in the .zip file):
 - `numpy.core` → `numpy.lib` → `numpy.ma` → `numpy.core`
 - `alphafold.common` → `alphafold.model` → `alphafold.common`
- Impact on Maintainability:
 - Cyclic dependencies complicate system comprehension and modification since altering one module may trigger unexpected effects in others within the cycle.
 - They can cause runtime issues, such as import errors when a module is accessed before it has been fully initialized.
 - Refactoring becomes more difficult because isolating a module for updates is challenging without disrupting the cycle.
- Unused/Disconnected Modules:
 - Disconnected Modules (Fan-In = 0 and Fan-Out = 0): None
- Depth of Dependencies
 - Maximum Dependency Depth from `run_alphafold.py`: 8
 - Maximum Dependency Depth in the Entire Graph: 8

5.2 Lombok - Java

- Identify classes with high LCOM values (top 90 percentile submitted in .zip file)

90th percentile of LCOM1: 167.80000000000018
Classes with high LCOM1 values:

	Package Name	Type Name	LCOM1	LCOM2	LCOM3	LCOM4	LCOM5	YALCOM
0	lombok.javac.handlers	JavaHandlerUtil	15424.0	14917.0	132.0	59.0	0.978777	0.296089
1	lombok.eclipse.handlers	EclipseHandlerUtil	12660.0	11459.0	109.0	51.0	0.985341	0.287425
2	lombok.javac	JavaTreeMaker	5983.0	5750.0	81.0	73.0	0.993289	0.482143
3	lombok.delombok	PrettyPrinter	5636.0	5494.0	79.0	6.0	0.981655	0.055556
4	lombok.launch	PatchFixesHider	3828.0	0.0	88.0	88.0	1.001095	0.943182
5	lombok.eclipse.handlers	SetGeneratedByVisitor	2737.0	0.0	1.0	1.0	0.754930	0.000000
6	lombok.javac	JavaAST	2048.0	1951.0	40.0	12.0	0.981154	0.181818
7	lombok.eclipse.agent	PatchDelegate	1558.0	1463.0	37.0	30.0	0.965748	0.500000
8	lombok.eclipse	EclipseASTVisitor	1490.0	1269.0	36.0	36.0	0.882184	-1.000000
9	lombok.javac.handlers	JavaSingularsRecipes	1380.0	1329.0	34.0	33.0	0.953688	0.611111
10	lombok.javac	JavaASTVisitor	1285.0	1085.0	34.0	34.0	0.824074	-1.000000
11	lombok.eclipse.agent	EclipsePatcher	1128.0	0.0	48.0	9.0	0.000000	0.187500
12	lombok.delombok	Delombok	973.0	911.0	26.0	12.0	0.980117	0.108696
13	com.sun.tools.javac.code	Symbol	902.0	901.0	45.0	45.0	1.000000	0.976744

Figure 2: Highest LCOM values

- What does a high LCOM value suggest about a class's design?
 - Unrelated Methods: The methods inside the class do not share many common instance variables.
 - Multiple Responsibilities: The class is handling more than one concern, violating the Single Responsibility Principle (SRP).
 - Difficult Maintainability: Such classes are harder to understand, test, and modify since they try to do too many things at once.
 - Code Smell - God Class: A very high LCOM might indicate a God Class, which should be refactored into smaller, more cohesive classes.
- Is there a chance for performing functional decomposition?
 - Yes, if a class has a high LCOM value, we should perform functional decomposition,
 - * Identifying independent functionalities within the class.
 - * Splitting the class into smaller, well-defined classes, where each class focuses on a single responsibility.
 - * Grouping related methods together into separate modules.
- Side by side comparison of LCOM values:

	Project Name	Package Name	Type Name	LCOM1	LCOM2	LCOM3	LCOM4	LCOM5	YALCOM
0	lombok	lombok.ant	SimpleTestFormatter	37.0	19.0	4.0	4.0	0.65	0.363636
1	lombok	org.mapstruct.ap.spi	AstModifyingAnnotationProcessor	0.0	0.0	1.0	1.0	0.00	-1.000000
2	lombok	org.mapstruct.ap.spi	AstModifyingAnnotationProcessor	0.0	0.0	1.0	1.0	0.00	-1.000000
3	lombok	org.eclipse.jdt.internal.compiler.ast	OperatorIds	0.0	0.0	0.0	0.0	0.00	-1.000000
4	lombok	com.sun.tools.javac.tree	EndPosTable	0.0	0.0	1.0	1.0	0.00	-1.000000

Figure 3: Table of LCOM values

6 Discussion and Conclusion

- Modules with high fan-in/fan-out should be modularized to improve maintainability.
- Classes with high LCOM suggest poor cohesion and should be refactored.
- Removing cyclic dependencies improves software structure and ease of modification.

7 Challenges Faced

- Choosing the right repositories for analysis was a time taking task.
- Understanding what constitutes a "high" LCOM value was subjective.
- Some classes had high LCOM values but were still logically grouped, making decomposition tricky.

8 Lessons Learned

- High LCOM = Warning Sign - A high LCOM is often a red flag for poor design and signals the need for refactoring. However, not all high-LCOM classes need immediate decomposition—context matters!
- Functional Decomposition is Key - Breaking down large, low-cohesion classes improves code maintainability and reduces complexity.
- Code Quality Metrics Drive Better Design - Using LCOM and other software metrics can quantify design issues, helping developers make data-driven decisions for refactoring.

- Visual Representations Help in Analysis - Representing cohesion levels in tables and dependency graphs makes it easier to spot problematic classes and plan refactoring.

9 Summary

This assignment explored class cohesion using LCOM analysis and module coupling using pydeps. High LCOM values revealed poorly designed Java classes that may benefit from decomposition. Dependency graphs helped identify tightly coupled Python modules, cycles, and potential risks in the system. Together, these tools provided valuable insights into improving software modularity and maintainability.

Lab Report 10: Building C# Console Applications

Kushal Rathod

22110128

CS202: Software Tools and Techniques for CSE

March 2025

1 Introduction

This laboratory exercise focuses on developing fundamental C# programming skills using Visual Studio 2022. C# is an object-oriented language developed by Microsoft that operates on the .NET framework. The assignment emphasizes creating console-based applications to establish core programming competencies.

Through these text-based programs running in command-line environments, students gain practical experience with essential programming concepts including basic syntax, control flow structures, functions, object-oriented principles, and error handling. The lab also introduces Visual Studio's debugging capabilities, a critical skill for identifying and resolving code issues.

1.1 Learning Objectives

The key goals of this laboratory work include:

- Configuring Visual Studio for .NET development
- Creating and running basic C# console applications
- Implementing core programming constructs: loops, conditionals, and methods
- Applying object-oriented programming principles
- Utilizing debugging tools effectively

1.2 Development Environment

Tools Utilized: The following software was employed:

- OS: Windows 11
- Development Environment: Visual Studio 2022 Community Edition
- Framework: .NET 6.0
- Language: C# 10.0

Setup Process: The initial step involved configuring the development environment for C# programming using Visual Studio 2022.

1.2.1 Visual Studio 2022 Installation

1. The Visual Studio 2022 installer was obtained from Microsoft's [official download portal](#).

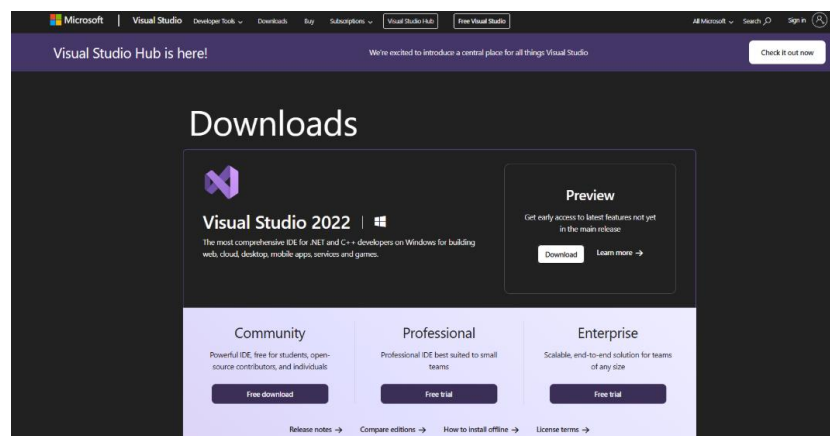


Figure 1: Visual Studio download interface

2. The installation wizard was executed with the following components selected:

- ".NET cross-platform development"
- "Desktop development with .NET"

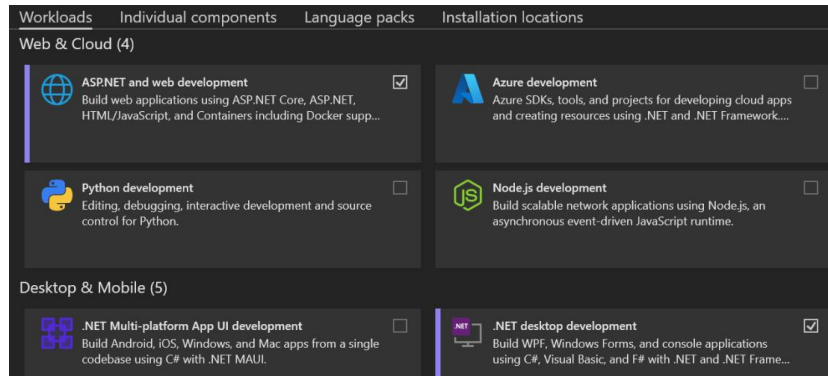


Figure 2: Component selection during installation

3. The installation process was completed following the on-screen guidance.

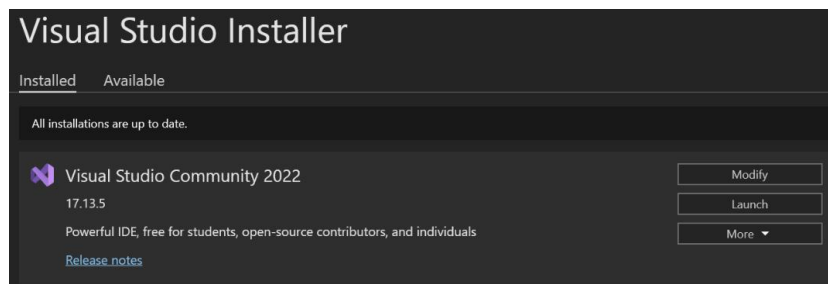


Figure 3: Successful installation confirmation

4. .NET SDK installation was verified through terminal command:

```
1 dotnet --version
```

The command output confirmed the successful installation of the required .NET SDK version.

2 Implementation Approach

2.0.1 Initializing a C# Console Project

1. Launched Visual Studio 2022 IDE.
2. Selected "Create new project" option.
3. Searched for "console" template and chose "Console App" with C# language.

4. Proceeded to configuration screen.
5. Named the project "STT_Lab10" and specified storage location.
6. Verified .NET 6.0 as target framework.
7. Finalized project creation.
8. The IDE generated a new project containing a basic Program.cs file with Main method.

2.1 Exercise 1: Hello, World! Program

The initial exercise involved creating a basic program to verify environment configuration.

The following figure contains both the glimpse of code (**all the codes are submitted properly in the github repository**) and the output:

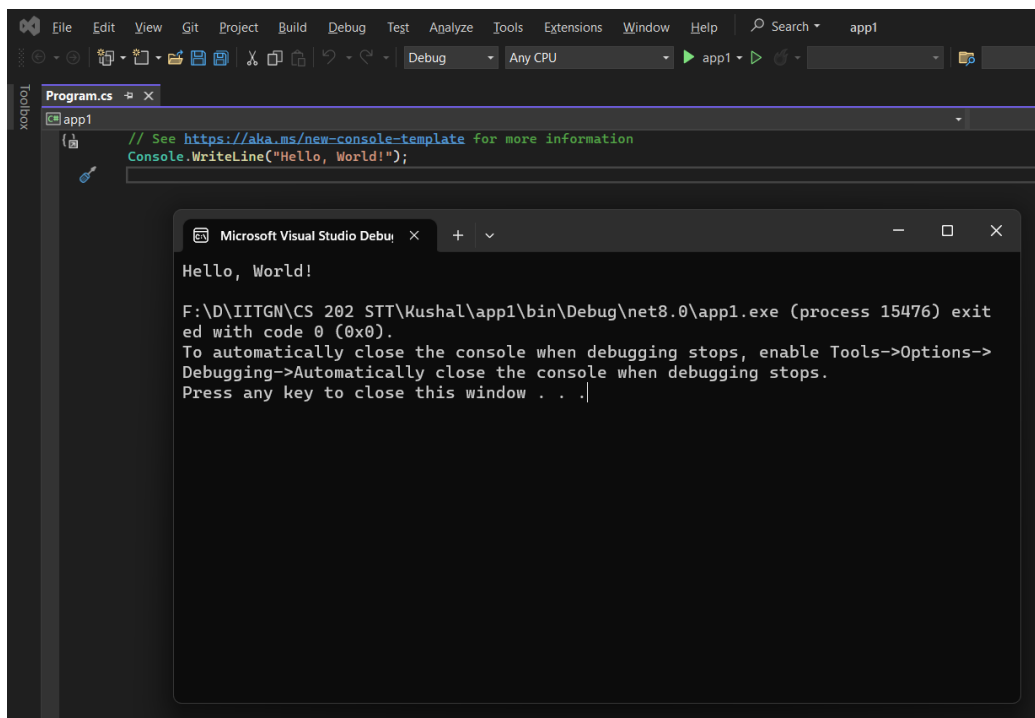
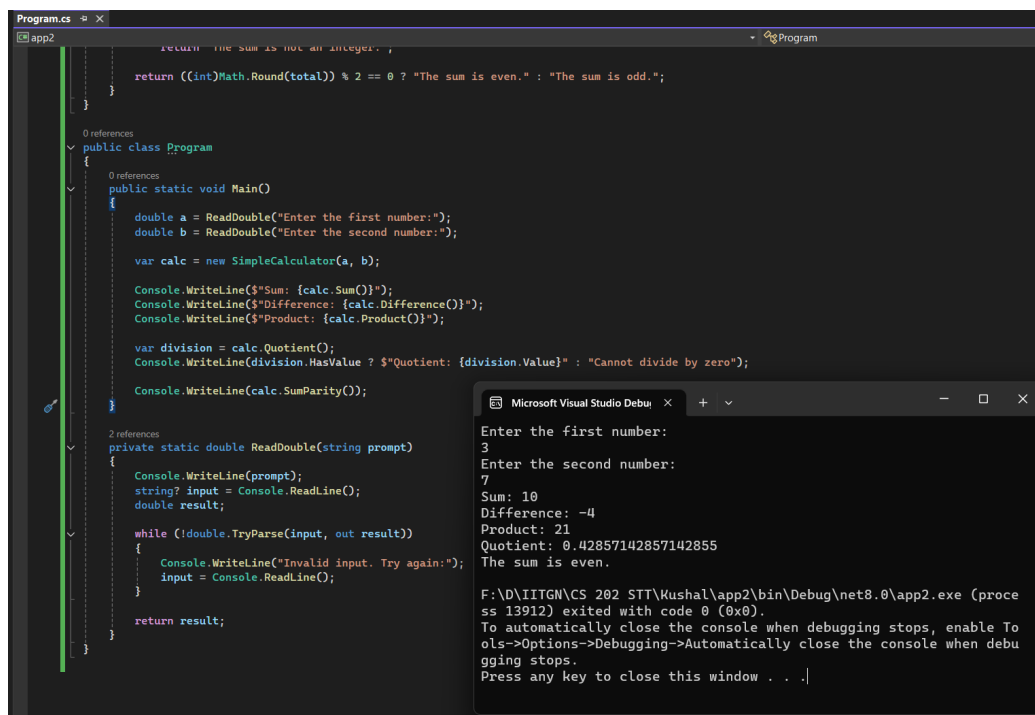


Figure 4: Program execution output

2.2 Exercise 2: Basic Syntax and Control Structures

To learn the basics of C#, I made a simple calculator that runs in the console. It asks the user to enter two numbers, then it does addition, subtraction, multiplication, and division. After adding the numbers, it also checks if the result is even or odd using if-else statements.

The following figure contains both the glimpse of code (**all the codes are submitted properly in the github repository**) and the output:



The screenshot displays a Visual Studio IDE with a C# program named 'Program.cs'. The code defines a 'Program' class with a 'Main' method. It uses 'ReadDouble' to get user input, calculates sum, difference, product, and quotient, and checks if the sum is even or odd. A 'ReadDouble' helper method is also shown. The console output shows the program running, prompting for two numbers (3 and 7), and displaying the results: Sum: 10, Difference: -4, Product: 21, Quotient: 0.42857142857142855, and 'The sum is even.'.

```
Program.cs x
app2

return "The sum is not an integer.";
}
return ((int)Math.Round(total)) % 2 == 0 ? "The sum is even." : "The sum is odd.";
}
}

0 references
public class Program
{
    0 references
    public static void Main()
    {
        double a = ReadDouble("Enter the first number:");
        double b = ReadDouble("Enter the second number:");

        var calc = new SimpleCalculator(a, b);

        Console.WriteLine($"Sum: {calc.Sum()}");
        Console.WriteLine($"Difference: {calc.Difference()}");
        Console.WriteLine($"Product: {calc.Product()}");

        var division = calc.Quotient();
        Console.WriteLine(division.HasValue ? $"Quotient: {division.Value}" : "Cannot divide by zero");

        Console.WriteLine(calc.SumParity());
    }
}

2 references
private static double ReadDouble(string prompt)
{
    Console.WriteLine(prompt);
    string? input = Console.ReadLine();
    double result;

    while (!double.TryParse(input, out result))
    {
        Console.WriteLine("Invalid input. Try again:");
        input = Console.ReadLine();
    }

    return result;
}
```

Microsoft Visual Studio Debu x + v - □ x

Enter the first number:
3
Enter the second number:
7
Sum: 10
Difference: -4
Product: 21
Quotient: 0.42857142857142855
The sum is even.

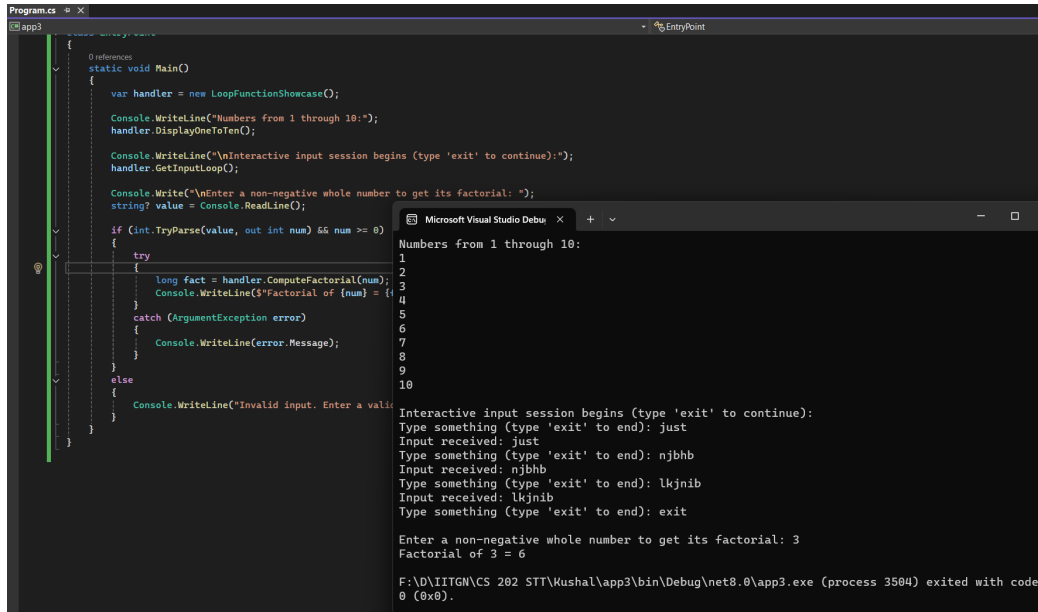
F:\D\IITGN\CS 202 STT\Kushal\app2\bin\Debug\net8.0\app2.exe (proce
ss 13912) exited with code 0 (0x0).
To automatically close the console when debugging stops, enable To
ols->Options->Debugging->Automatically close the console when debu
gging stops.
Press any key to close this window . . .|

Figure 5: Output of calculator program

2.3 Exercise 3: Implementing Loops and Functions

To get a better understanding of loops and functions in C#, I created a console program that prints numbers from 1 to 10. I also made another program using a while loop that keeps taking input from the user and prints it back until the user types “exit”. To practice writing and using functions, I also built a simple program that calculates the factorial of a number using an iterative method.

The following figure contains both the glimpse of code (all the codes are submitted properly in the github repository) and the output:



The screenshot shows a Visual Studio IDE with a C# program and its output. The code in Program.cs defines a static Main method that creates a LoopFunctionShowcase object, displays numbers 1 through 10, and enters an interactive loop for calculating factorials. The console output shows the program's execution, including the list of numbers, the start of the interactive session, and the calculation of the factorial of 3.

```
Program.cs
using System;

static void Main()
{
    var handler = new LoopFunctionShowcase();

    Console.WriteLine("Numbers from 1 through 10:");
    handler.DisplayOneToTen();

    Console.WriteLine("\nInteractive input session begins (type 'exit' to continue):");
    handler.GetInputLoop();

    Console.WriteLine("\nEnter a non-negative whole number to get its factorial: ");
    string? value = Console.ReadLine();

    if (int.TryParse(value, out int num) && num >= 0)
    {
        try
        {
            long fact = handler.ComputeFactorial(num);
            Console.WriteLine($"Factorial of {num} = {fact}");
        }
        catch (ArgumentException error)
        {
            Console.WriteLine(error.Message);
        }
    }
    else
    {
        Console.WriteLine("Invalid input. Enter a valid number.");
    }
}
```

Microsoft Visual Studio Debug Console Output:

```
Numbers from 1 through 10:
1
2
3
4
5
6
7
8
9
10

Interactive input session begins (type 'exit' to continue):
Type something (type 'exit' to end): just
Input received: just
Type something (type 'exit' to end): njbhb
Input received: njbhb
Type something (type 'exit' to end): lkjnib
Input received: lkjnib
Type something (type 'exit' to end): exit

Enter a non-negative whole number to get its factorial: 3
Factorial of 3 = 6

F:\D\IITGN\CS 202 STT\Mushal\app3\bin\Debug\net8.0\app3.exe (process 3504) exited with code 0 (0x0).
```

Figure 6: Output of loops and functions program

2.4 Exercise 4: Object-Oriented Programming in C#

To understand the basics of object-oriented programming in C#, I created a simple Student class with attributes like Name, ID, and Marks. It had methods to assign values, calculate grades, and display student details. I also explored constructor overloading and copy constructors, which helped me copy data from one student object to another. Later, I made a subclass called StudentIITGN that inherited from Student and added an extra detail — the name of the hostel. Both classes had their own Main() methods to test the code. Here are a few things I noticed while working on this:

- The Main() method in C# must be static because it's the first thing the program runs, and it doesn't depend on any object being created.
- If you call Main() from itself repeatedly, it causes a stack overflow and crashes the program.
- C# is case-sensitive, so main() (lowercase) won't work unless you change the program's entry point manually.

- You can have separate Main() methods in both classes, but only one can act as the starting point of the program. If both are present, the compiler may throw an error.
- To run one Main() method, the other one has to be commented out. For example, to test the subclass, I had to comment out Student.Main() and keep StudentIITGN.Main() active.

The following figure contains both the glimpse of code (**all the codes are submitted properly in the github repository**) and the output:

The screenshot shows a Visual Studio IDE with two windows. The top window, titled 'Program.cs', displays the following C# code:

```

else
    return "F";
}

0 references
public static void Main()
{
    // Original object
    Student student1 = new Student("Kushal", 22110128, 100);
    Console.WriteLine("Original Student:");
    Console.WriteLine($"Name: {student1.Name}, ID: {student1.ID}, Marks: {student1.Marks}, Grade: {student1.GetGrade()}");

    // Using overloaded constructor
    Student student2 = new Student("Digvijay");
    Console.WriteLine("\nStudent Created with Overloaded Constructor:");
    Console.WriteLine($"Name: {student2.Name}, ID: {student2.ID}, Marks: {student2.Marks}, Grade: {student2.GetGrade()}");

    // Using copy constructor
    Student student3 = new Student(student1);
    Console.WriteLine("\nCopied Student:");
    Console.WriteLine($"Name: {student3.Name}, ID: {student3.ID}, Marks: {student3.Marks}, Grade: {student3.GetGrade()}");
}

```

The bottom window, titled 'Microsoft Visual Studio Debug Console', shows the output of the program:

```

Name: Kushal, ID: 22110128, Marks: 100, Grade: A
Student Created with Overloaded Constructor:
Name: Digvijay, ID: 0, Marks: 0, Grade: F
Copied Student:
Name: Kushal, ID: 22110128, Marks: 100, Grade: A
F:\D\IITGN\CS 202 STT\Kushal\app4\bin\Debug\net8.0\app4.exe (process 14240) exited with code 0 (0x0).
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .

```

Figure 7: Program testing Student.Main() (StudentIITGN.Main() commented)

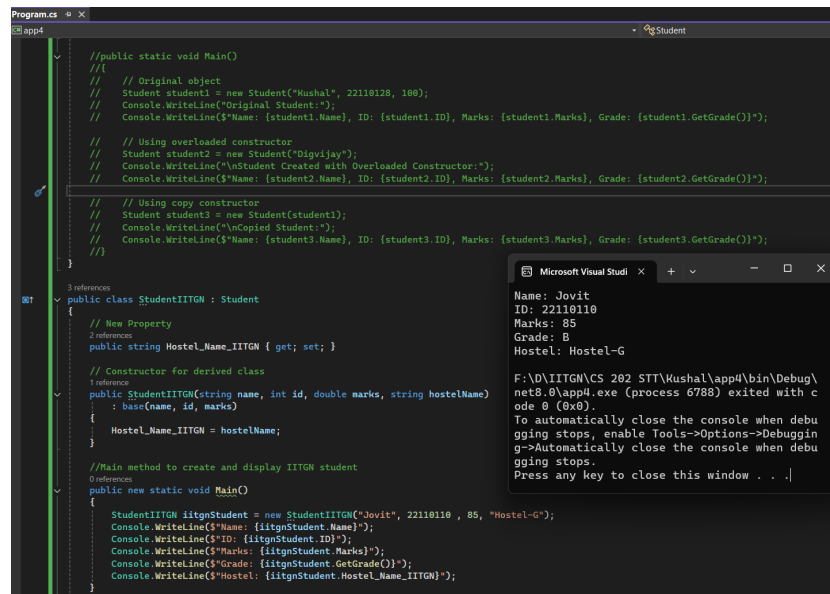


Figure 8: Program testing StudentIITGN.Main() (Student.Main() commented)

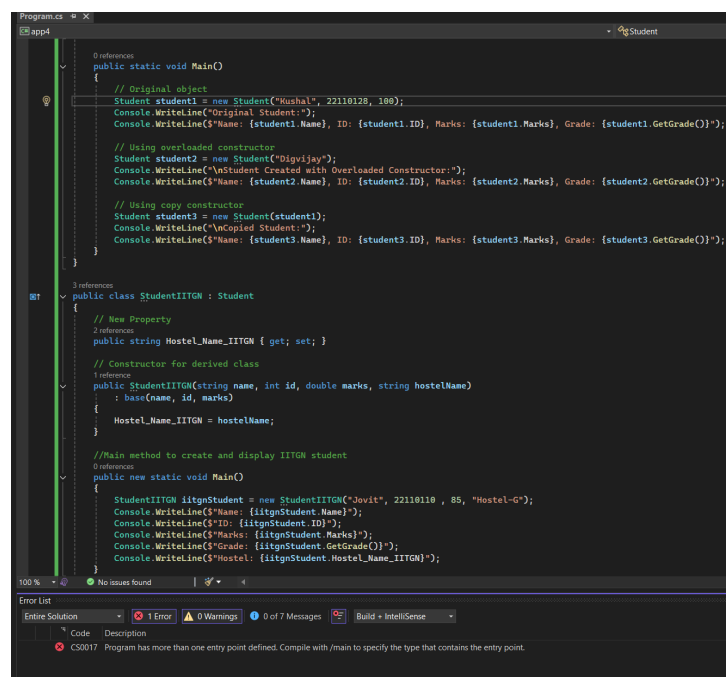


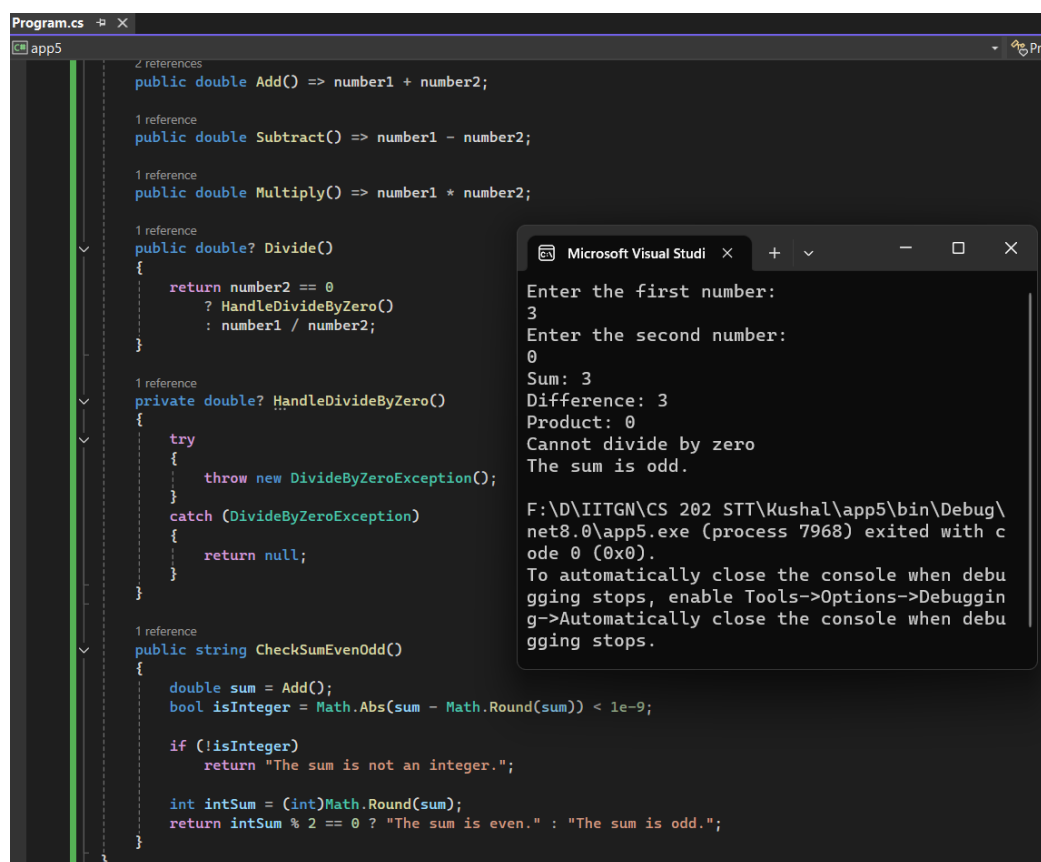
Figure 9: Program testing, 2 main together

- This will definitely throw an error as there are two main functions.

2.5 Exercise 5: Exception Handling

To learn how exception handling works in C#, I updated the console-based calculator I had built earlier. I added a try-catch block, especially for the division part, so that if the user tries to divide by zero, the program doesn't crash. Instead, it shows a proper message and keeps running smoothly.

The following figure contains both the glimpse of code (**all the codes are submitted properly in the github repository**) and the output:



The screenshot displays the Visual Studio IDE with a C# file named `Program.cs` open. The code implements a calculator with methods for addition, subtraction, multiplication, and division. The `Divide()` method uses a try-catch block to handle a `DivideByZeroException` if the divisor is zero. The `HandleDivideByZero()` method catches the exception and returns `null`. The `CheckSumEvenOdd()` method checks if the sum of two numbers is even or odd. The console output shows the results of these operations: Sum: 3, Difference: 3, Product: 0, Cannot divide by zero, and The sum is odd. A message box also displays the file path and exit code of the application.

```
Program.cs x
app5

2 references
public double Add() => number1 + number2;

1 reference
public double Subtract() => number1 - number2;

1 reference
public double Multiply() => number1 * number2;

1 reference
public double? Divide()
{
    return number2 == 0
        ? HandleDivideByZero()
        : number1 / number2;
}

1 reference
private double? HandleDivideByZero()
{
    try
    {
        throw new DivideByZeroException();
    }
    catch (DivideByZeroException)
    {
        return null;
    }
}

1 reference
public string CheckSumEvenOdd()
{
    double sum = Add();
    bool isInteger = Math.Abs(sum - Math.Round(sum)) < 1e-9;

    if (!isInteger)
        return "The sum is not an integer.";

    int intSum = (int)Math.Round(sum);
    return intSum % 2 == 0 ? "The sum is even." : "The sum is odd.";
}

Microsoft Visual Studio x + - □ x

Enter the first number:
3
Enter the second number:
0
Sum: 3
Difference: 3
Product: 0
Cannot divide by zero
The sum is odd.

F:\D\IITGN\CS 202 STT\Kushal\app5\bin\Debug\net8.0\app5.exe (process 7968) exited with code 0 (0x0).
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
```

Figure 10: Output of exception handling program

- Here we can clearly that "Cannot divide by zero" comes in the output.

2.6 Exercise 6: Debugging using Visual Studio Debugger

2.6.1 Overview of Debugging in Visual Studio

Setting Breakpoints

- Open the code file you want to debug (like `Program.cs`).
- Click on the left side margin next to the line where you want the program to pause.
- A red dot will appear showing that the breakpoint is set.

Starting the Debugging Session

- Press **F5** to start debugging.
- Use these shortcut keys to control how the program runs:
 - **F5 (Continue)** Runs the program until the next breakpoint is hit.
 - **F10 (Step Over)** Runs the current line, skipping over method details.
 - **F11 (Step Into)** Goes into method calls so you can see what's happening inside.
 - **Shift+F11 (Step Out)** Comes out of the current method and goes back to the caller.
 - **Shift+F5 (Stop Debugging)** Ends the debugging session right away.

Inspecting Program Behavior

- **Variables/Locals Pane:** Lets you see the current values of variables.
- **Call Stack Window:** Shows the list of method calls leading to the current point.
- **Watch Window:** Helps you keep an eye on specific variables or expressions.
- **Breakpoints Pane:** Lets you manage all the breakpoints you've set.

Note: Rather than using screenshots, video demonstrations showing the debugging process at key breakpoints for all the programs mentioned above have been provided. These videos (.mkv format) are available in the media folder of the STT_lab10 branch in the shared GitHub repository. Please refer to them for a clearer understanding.

3 Results and Observations

- **Activity 1** focused on writing a simple "Hello, World!" program to get started with C# syntax and structure. The `Main()` method needed to be marked `static` as it's the entry point and doesn't rely on object instances. Adding `Console.ReadKey()` helped keep the console open after execution, which was especially helpful while testing the output.
- In **Activity 2**, basic arithmetic operations and an even/odd number checker were implemented in a `Calculator` class. It made use of constructors, methods, and properties to neatly organize the logic. Input was taken using `Convert.ToDouble()`, although at this stage, error handling for bad inputs was not yet added.
- **Activity 3** introduced the use of loops and functions. A `for` loop printed numbers from 1 to `n`, a `while` loop kept accepting user input until "exit" was typed, and a separate method computed factorials using iteration. These examples helped understand control flow and how functions return values.
 - Using `string.ToLower()` allowed users to enter "exit" in any case (like "Exit", "EXIT"), improving usability and avoiding unnecessary input issues.
- **Activity 4** was about learning object-oriented programming. A base class `Student` and a subclass `StudentIITGN` were created. This helped understand inheritance and how methods from the parent class can be reused or overridden. The `DisplayDetails()` method in the subclass called `base.DisplayDetails()` to avoid rewriting the same code.
 - Constructors were used to set values when objects were created. A grading system based on marks was implemented using `if-else`, showing how logic can be built around object properties.

- **Activity 5** improved upon the calculator program by adding proper error handling. A helper method `GetValidNumber()` was introduced to manage invalid inputs and catch exceptions like `FormatException` and `OverflowException`.
 - The division logic included a `try-catch` block specifically for `DivideByZeroException`, preventing the program from crashing and allowing customized error messages.
 - Methods were reused efficiently. For example, the `CheckEvenOdd()` method worked across multiple activities without changes, showing how reusable code can simplify development.
- Debugging was done using Visual Studio’s breakpoints. Breakpoints inside `catch` blocks helped observe how control flowed when exceptions were caught. Tools like the Watch and Locals windows made it easy to track variables during program execution.
- Throughout all activities, access modifiers like `public`, `private`, and `internal` were used appropriately to define scope and maintain encapsulation.

Discussion and Conclusion

Challenges Faced

Each activity came with its own learning curve. Setting up the development environment and understanding the structure of a basic C# project was tricky at first. As tasks got more complex, input validation and error prevention became important, especially while handling user-driven data. In **Activity 5**, implementing exception handling was particularly challenging—it required thinking ahead about what kind of problems might occur during runtime. Debugging division-by-zero and format errors meant placing breakpoints thoughtfully and testing different cases. It was also necessary to differentiate between user errors and logical mistakes in the code. Navigating the code using debugging shortcuts like `F5`, `F10`, and `F11` made identifying problems easier. Designing and testing class hierarchies in **Activity 4** helped understand how inheritance and encapsulation really work in practice. These challenges encouraged writing cleaner and more structured code.

Reflections

Working through these tasks offered a gradual and effective introduction to C#. Starting from basics and building up to object-oriented and exception-handling concepts, each activity added something new. With each step, confidence in writing C# code grew, and so did the appreciation for its structure and rules. Handling real-time input and errors helped highlight the importance of writing user-friendly and fault-tolerant programs.

Lessons Learned

1. **Start with the Basics:** Understanding things like how to print to the console and read user input lays a strong foundation.
2. **Reusability Helps:** Breaking code into reusable methods and classes keeps things cleaner and easier to manage.
3. **Always Handle Errors:** Using `try-catch` blocks can save a program from crashing and provide better user feedback.
4. **Object-Oriented Concepts Matter:** Working with base and derived classes showed how inheritance and polymorphism can simplify and organize code.
5. **Validate Input:** Never assume user input is valid. Defensive programming goes a long way in building robust software.

Summary

Overall, the five activities provided a practical and hands-on introduction to programming in C#. Starting from the simplest "Hello, World!" example and moving towards more advanced concepts like classes, inheritance, and exception handling, each step built on the last. By the end, the exercises offered a deeper understanding of how to write clean, efficient, and reliable C# code, while also developing logical thinking and problem-solving skills.