

LAB 9 : Analysis of Module Dependencies and Cohesion

Aayush

April 7, 2025

1 Introduction, Setup, and Tools

1.1 Overview and Objectives

This lab focuses on analyzing software dependencies and cohesion using pydeps (for Python) and LCOM (Lack of Cohesion of Methods) (for Java). The objectives are:

- Use pydeps to generate and analyze dependency graphs for Python projects.
- Use LCOM to measure class cohesion in Java projects.
- Identify design flaws and code smells (anti-patterns).
- Implement modularization and refactoring strategies to optimize software structure.

1.2 Environment Setup

- Operating System: Windows/Linux/macOS
- Programming Languages: Python (3.7 or above) and Java (11 or later)
- Required Tools: pydeps, LCOM

2 Methodology and Execution

2.1 Selecting a Real-World Project

A real-world Python project with multiple modules and at least 500+ lines of source code (Flask and Praxis) was chosen. A Java project with at least 10 classes was also selected (Galosphere-Main).

2.2 Running pydeps

Navigating to the project directory and running the following command:

```
pydeps <project_directory>  
pydeps --show-deps <project_directory>
```

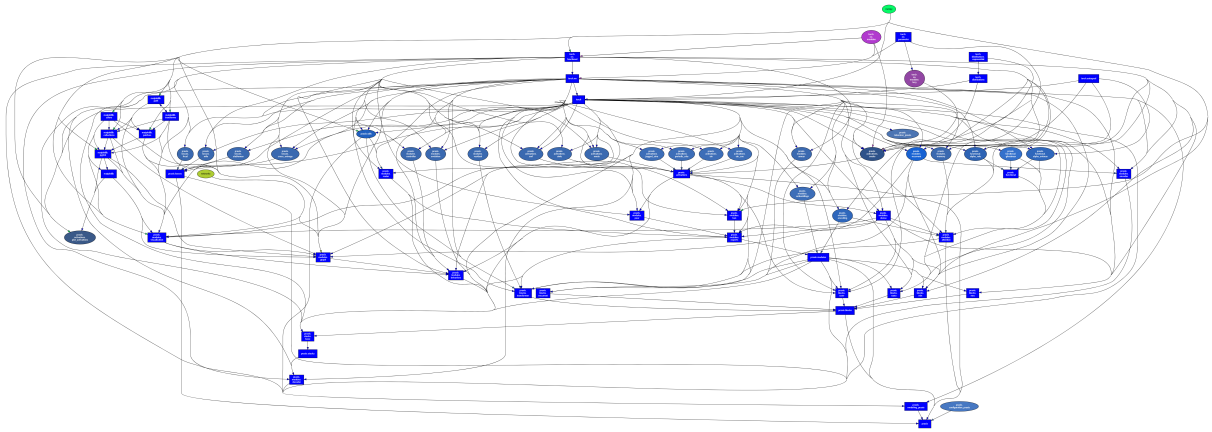


Figure 1: Praxis Dependency Graph

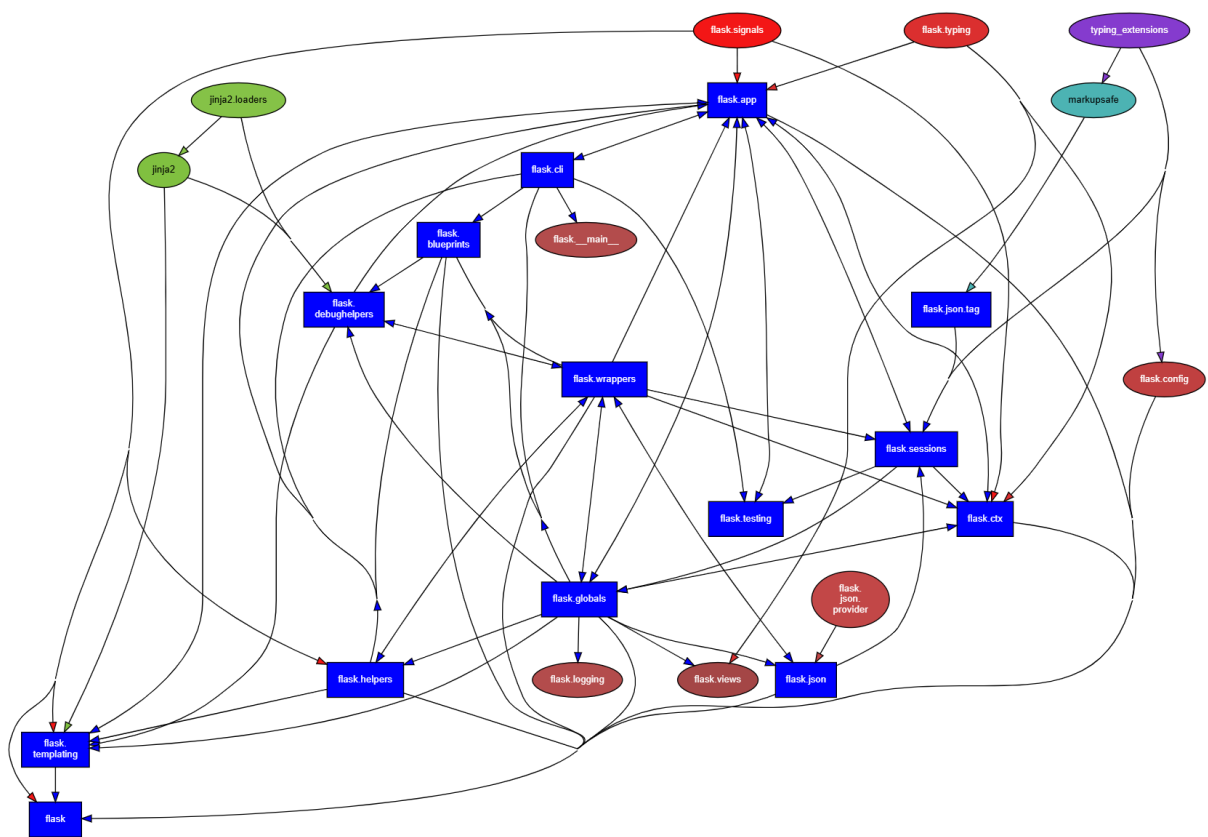


Figure 2: Flask Dependency Graph

2.3 Analyzing Dependency Graph

2.3.1 Highly Coupled Modules and Their Dependencies

Highly coupled modules have many incoming and outgoing edges, indicating deep inter-connections.

flask.app

- Most highly coupled node.
- Depends on: `flask.signals`, `flask.typing`, `flask.ctx`, `flask.helpers`, `flask.json`, `flask.sessions`, etc.
- Is used by: `flask`, `flask.cli`, `flask.blueprints`, `flask.wrappers`, `flask.debughelpers`, etc.

flask.ctx

- Many incoming and outgoing dependencies.
- Used by: `flask.app`, `flask.sessions`, `flask.json`, `flask.testing`, `flask.helpers`, etc.

flask.helpers

- Widely used utility module.
- Dependencies: `flask.templating`, `flask`, `flask.cli`, `flask.app`, `flask.debughelpers`.

2.3.2 Cyclic Dependencies

Cyclic dependencies are loops where modules indirectly depend on themselves through a chain.

`flask.app` → `flask.helpers` → `flask.templating` → `flask` → `flask.app`

Such cycles can:

- Make debugging and maintenance harder.
- Lead to unexpected bugs during refactoring.
- Prevent separation of concerns or modular builds.

2.3.3 Unused and Disconnected Modules

Modules with no or few connections:

- `flask.__main__`: Likely used for CLI/startup.
- `flask.signals`, `flask.typing`, `flask.config`: Only used by `flask.app`.

These modules are likely used for:

- Initialization (`__main__`)
- Typing and configuration utilities

2.3.4 Depth of Dependencies

Core Module: `flask.app`

- Directly or indirectly influences most of the system.
- At the root of the dependency tree.

Long Chains of Dependencies

`flask` \rightarrow `flask.templating` \rightarrow `flask.helpers` \rightarrow `flask.ctx` \rightarrow `flask.sessions`

Observation: More layers = more fragile and harder-to-debug behavior.

2.4 Dependency Impact Assessment

2.4.1 Dependency Impact Assessment

How would changes in `flask.app` affect the system?

- High impact across the board.
- Affects `flask`, `flask.cli`, `flask.sessions`, `flask.ctx`, and more.

Mitigation: Use abstraction layers or interfaces to minimize blast radius.

Modules at High Risk of Breaking the System if Modified

- `flask.app`: Central hub with high coupling.
- `flask.ctx`: Core to request/response lifecycle.
- `flask.helpers`: Widely used across the project.
- `flask.globals`: Tied to app state and shared data.

Action: Modifications require careful testing and full dependency awareness.

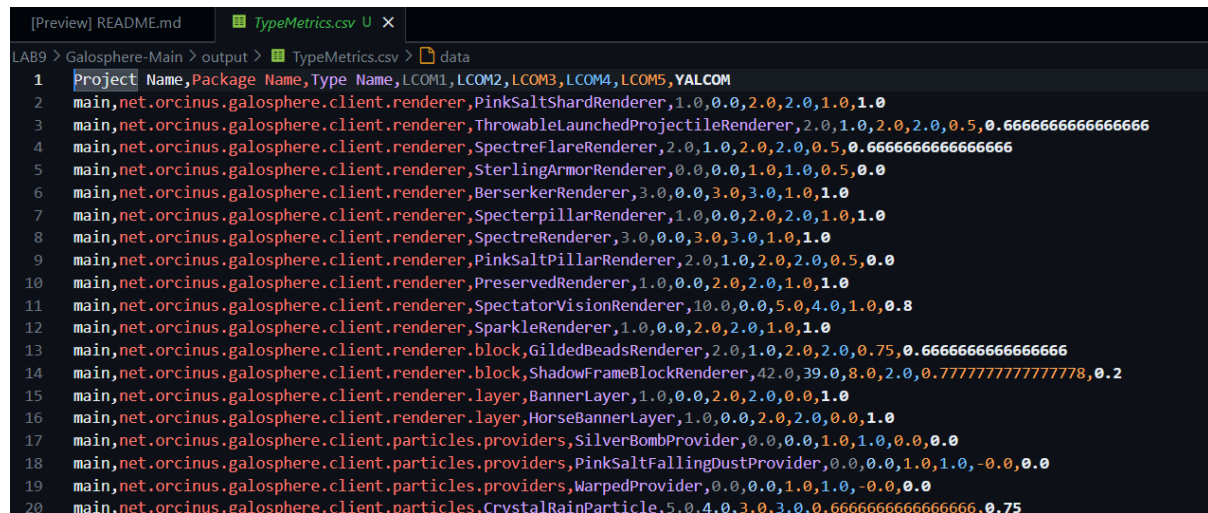
Summary Table

Aspect	Module(s)
Highly Coupled	<code>flask.app</code> , <code>flask.ctx</code> , <code>flask.helpers</code>
Cyclic Dependencies	<code>flask.app</code> \rightarrow <code>flask.helpers</code> \rightarrow <code>flask.templating</code> \rightarrow <code>flask</code> \rightarrow <code>flask.app</code>
Disconnected / Unused	<code>flask.__main__</code> , <code>flask.typing</code> , <code>flask.signals</code> , <code>flask.config</code>
High Depth / Core Modules	<code>flask.app</code> , <code>flask.ctx</code> , <code>flask.helpers</code>
Impact Risk if Modified	<code>flask.app</code> , <code>flask.ctx</code> , <code>flask.helpers</code> , <code>flask.globals</code>

2.5 Running LCOM Analysis on Java Classes

Navigate to the Java project directory and run:

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



```
[Preview] README.md | TypeMetrics.csv U X
LAB9 > Galosphere-Main > output > TypeMetrics.csv > data
1 Project Name,Package Name,Type Name,LCOM1,LCOM2,LCOM3,LCOM4,LCOM5,YALCOM
2 main,net.orcinus.galosphere.client.renderer,PinkSaltShardRenderer,1.0,0.0,2.0,2.0,1.0,1.0
3 main,net.orcinus.galosphere.client.renderer,ThrowableLaunchedProjectileRenderer,2.0,1.0,2.0,2.0,0.5,0.6666666666666666
4 main,net.orcinus.galosphere.client.renderer,SpectreFlareRenderer,2.0,1.0,2.0,2.0,0.5,0.6666666666666666
5 main,net.orcinus.galosphere.client.renderer,SterlingArmorRenderer,0.0,0.0,1.0,1.0,0.5,0.0
6 main,net.orcinus.galosphere.client.renderer,BerserkerRenderer,3.0,0.0,3.0,3.0,1.0,1.0
7 main,net.orcinus.galosphere.client.renderer,SpecterpillarRenderer,1.0,0.0,2.0,2.0,1.0,1.0
8 main,net.orcinus.galosphere.client.renderer,SpectreRenderer,3.0,0.0,3.0,3.0,1.0,1.0
9 main,net.orcinus.galosphere.client.renderer,PinkSaltPillarRenderer,2.0,1.0,2.0,2.0,0.5,0.0
10 main,net.orcinus.galosphere.client.renderer,PreservedRenderer,1.0,0.0,2.0,2.0,1.0,1.0
11 main,net.orcinus.galosphere.client.renderer,SpectatorVisionRenderer,10.0,0.0,5.0,4.0,1.0,0.8
12 main,net.orcinus.galosphere.client.renderer,SparkleRenderer,1.0,0.0,2.0,2.0,1.0,1.0
13 main,net.orcinus.galosphere.client.renderer.block,GildedBeadsRenderer,2.0,1.0,2.0,2.0,0.75,0.6666666666666666
14 main,net.orcinus.galosphere.client.renderer.block,ShadowFrameBlockRenderer,42.0,39.0,8.0,2.0,0.7777777777777778,0.2
15 main,net.orcinus.galosphere.client.renderer.layer,BannerLayer,1.0,0.0,2.0,2.0,0.0,1.0
16 main,net.orcinus.galosphere.client.renderer.layer,HorseBannerLayer,1.0,0.0,2.0,2.0,0.0,1.0
17 main,net.orcinus.galosphere.client.particles.providers,SilverBombProvider,0.0,0.0,1.0,1.0,0.0,0.0
18 main,net.orcinus.galosphere.client.particles.providers,PinkSaltFallingDustProvider,0.0,0.0,1.0,1.0,-0.0,0.0
19 main,net.orcinus.galosphere.client.particles.providers,WarpedProvider,0.0,0.0,1.0,1.0,-0.0,0.0
20 main,net.orcinus.galosphere.client.particles,CrystalRainParticle,5.0,4.0,3.0,3.0,0.6666666666666666,0.75
```

Figure 3: Generated LCOM csv

2.6 Identifying High LCOM Classes

2.6.1 What does a high LCOM value suggest about a class's design?

A high LCOM (Lack of Cohesion of Methods) value indicates **low cohesion** within a class. This metric helps assess how related the methods in a class are to each other based on shared data (i.e., class attributes).

It suggests that:

- Methods in the class operate on different subsets of instance variables, implying that the methods are functionally unrelated.
- The class might be trying to encapsulate multiple responsibilities or behaviors, thus violating the *Single Responsibility Principle (SRP)*.
- Such a design can increase complexity, reduce maintainability, and hinder reusability across the system.
- Classes with low cohesion tend to have more defects and are harder to understand or extend.
- It may also indicate poor encapsulation, where internal data is not logically grouped with the operations that act on it.

Therefore, a high LCOM value is a red flag that encourages a critical review of the class's responsibilities and structure.

2.6.2 Is there a chance for performing functional decomposition?

Yes. High LCOM classes are prime candidates for **functional decomposition**. By analyzing the correlation between methods and the attributes they use, such classes can be refactored into smaller, more cohesive units. Functional decomposition promotes the separation of concerns and simplifies class behavior.

Decomposition benefits include:

- Smaller classes with a focused scope, each performing a well-defined task.
- Improved encapsulation, as each new class will likely have stronger internal consistency.
- Enhanced readability and ease of maintenance, as each class will be simpler and more intuitive.
- Better testability, since classes with high cohesion are easier to write unit tests for.
- Facilitates code reuse, as responsibilities are more modular and less entangled.
- Improved compliance with SOLID principles, particularly SRP and the Open/Closed Principle.

This decomposition process is a step toward refactoring the codebase for better object-oriented design.

2.6.3 Examples of High LCOM Classes from the Dataset

The dataset reveals several classes with high LCOM scores, indicating a need for structural improvement. Some notable examples include:

- **BerserkerRenderer (LCOM1 = 3.0, LCOM3 = 3.0)**: This class likely contains rendering logic for multiple independent features or effects that do not share internal data. Such separation of concerns can be moved into dedicated renderer components.
- **SpectreFlareRenderer (LCOM1 = 2.0)**: Suggests bifurcated behavior, possibly involving multiple rendering modes or object types. Decomposition into individual flare rendering modules may be beneficial.
- **ThrowableLaunchedProjectileRenderer (LCOM1 = 2.0)**: May be handling various projectile types or effects, each with distinct rendering strategies. Consider isolating each projectile type into its own renderer class.

In these examples, applying cohesion-based refactoring would not only simplify class structure but also enhance rendering system flexibility by isolating specific concerns into modular components.

2.7 Visualizing Cohesion

2.7.1 LCOM Metrics Table

Java Class	LCOM1	LCOM2	LCOM3	LCOM4	LCOM5	YALCOM
PinkSaltShardRenderer	1.0	0.0	2.0	2.0	1.0	1.000
ThrowableLaunchedProject	2.0	1.0	2.0	2.0	0.5	0.667
SpectreFlareRenderer	2.0	1.0	2.0	2.0	0.5	0.667
SterlingArmorRenderer	0.0	0.0	1.0	1.0	0.5	0.000
BerserkerRenderer	3.0	0.0	3.0	3.0	1.0	1.000

Table 1: LCOM Metrics for Selected Java Classes

3 Results and Analysis

3.1 Outputs and Observations

- Dependency graphs revealed highly coupled modules such as `flask.app`, `flask.ctx`, and `flask.helpers`.
- Cyclic dependencies indicate tight coupling and potential architectural flaws.
- Disconnected or lightly used modules point toward possible redundancies or specialized responsibilities.
- LCOM values showed that classes like `BerserkerRenderer` and `SpectreFlareRenderer` had poor cohesion, which may hinder maintainability and clarity.

3.2 Key Insights and Comparisons

- High coupling increases system fragility, making the project prone to regression bugs during refactoring.
- Cycles can trap teams into monolithic behaviors even in modular codebases.
- High LCOM scores reveal hidden technical debt and opportunities to split responsibilities.
- Combining visual tools (like `pydeps`) with quantitative metrics (like LCOM) provides a robust strategy for early detection of architectural issues.

3.3 Real-World Implications

- Teams maintaining Flask-like architectures should implement boundaries using design patterns like Facade or Mediator to reduce ripple effects of change.
- In Java ecosystems, cohesion metrics can guide legacy code refactoring, especially in game engines or renderers where logic often gets duplicated or scattered.
- Applying this methodology during code reviews or CI pipelines can enforce quality gates beyond style checks.

3.4 Recommendations

- Refactor highly coupled modules by introducing interfaces or abstract base classes.
- Split classes with high LCOM into smaller units dedicated to single responsibilities.
- Avoid circular dependencies by re-architecting using dependency inversion or observer patterns.
- Automate pydeps/LCOM runs in CI/CD for continuous design validation.

4 Discussion and Conclusion

4.1 Challenges and Lessons Learned

- Understanding large graphs without annotations or filters was overwhelming initially.
- Mapping LCOM values back to meaningful class behavior required domain familiarity.
- Circular dependencies are often introduced unintentionally over time and go unnoticed without regular analysis.

4.2 Broader Takeaways

- Structural analysis tools offer critical insight into maintainability beyond what code coverage or linting provides.
- Metrics like LCOM help enforce the Single Responsibility Principle in an objective manner.
- Dependency graphs offer powerful visual feedback but require careful interpretation and filtering to avoid misjudgments.

4.3 Conclusion

This lab demonstrated a holistic workflow to analyze, interpret, and improve modularization and class design in real-world codebases using pydeps and LCOM tools. It revealed how metrics can be applied both visually and quantitatively to identify architectural weaknesses, guide refactoring efforts, and ensure scalable, maintainable software development.

Future Work: Explore more granular metrics like RFC (Response for a Class) and coupling-based clustering algorithms. Tools like Graphviz or Gephi may help enhance the visual feedback further. Integrating such analysis early in the software lifecycle (design or review phase) could drastically reduce technical debt downstream.

LAB 10 : Introduction to .NET Development using C#

Aayush

April 7, 2025

1 Introduction, Setup, and Tools

1.1 Overview and Objectives

This lab introduces US to .NET development using C# in Visual Studio. The focus is on creating simple console applications to understand basic syntax, control structures, and object-oriented programming principles in C#.

The objectives are:

- Set up and use Visual Studio for .NET development.
- Write and execute basic C# console applications.
- Implement fundamental programming constructs: loops, conditionals, and functions.
- Apply object-oriented programming concepts in C#.
- Understand the benefit of the Visual Studio Debugger.

1.2 Environment Setup

- Operating System: Windows
- Software: Visual Studio 2022 (Community Edition), Visual Studio with .NET SDK
- Programming Language: C# (latest stable version)

2 Methodology and Execution

2.1 Setting Up .NET Development Environment

- Opened Visual Studio and created a new C# Console Application project.
- Ensured the target framework is .NET 6 or later.
- Wrote and executed a simple program.

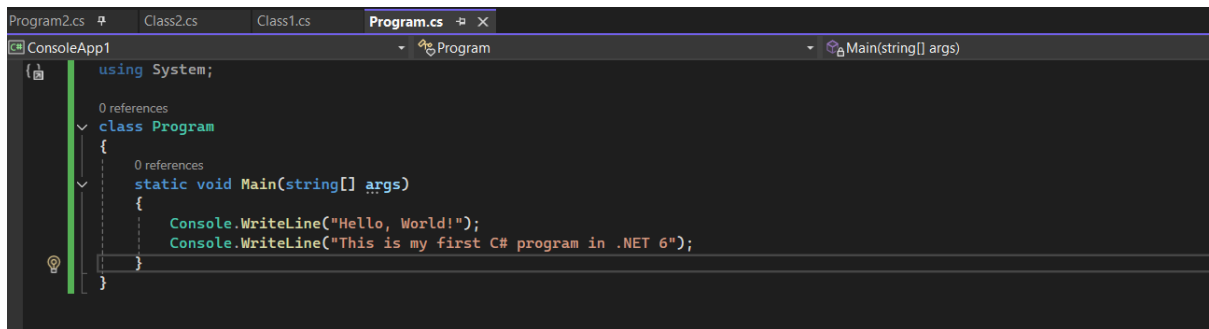


Figure 1: Hello World Programme

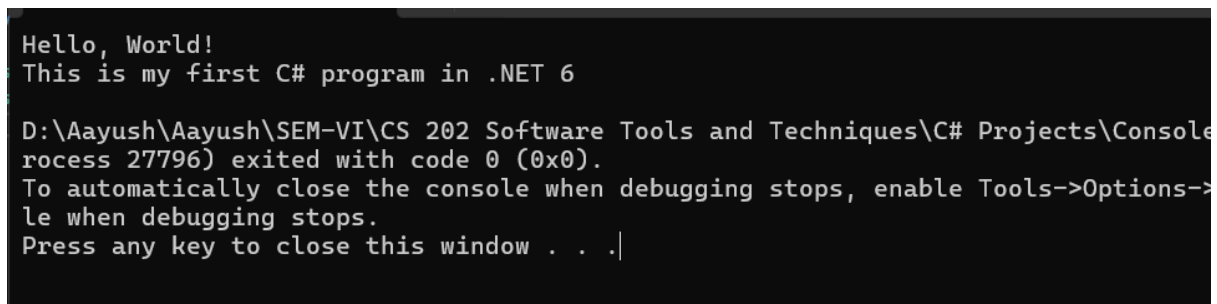


Figure 2: Executing Programme

2.2 Understanding Basic Syntax and Control Structures

Write an object-oriented C# program that:

- Accepts user input for two numbers.
- Performs addition, subtraction, multiplication, and division.
- Uses if-else conditions to determine if the sum is even or odd.
- Displays the results using `Console.WriteLine()`.

```

string even_odd;
// Check if sum is even or odd
if (sum % 2 == 0) {
    even_odd = "even";
}
else
{
    even_odd = "odd";
}

// Display results
Console.WriteLine($"Results:");
Console.WriteLine($"Sum: {sum} (This is {even_odd})");
Console.WriteLine($"Difference: {difference}");
Console.WriteLine($"Product: {product}");

if (num2 != 0)
{
    Console.WriteLine($"Quotient: {quotient}");
}
else

```

Figure 3: Code Block

```

Enter first number: 5
Enter second number: 5

Results:
Sum: 10 (This is even)
Difference: 0
Product: 25
Quotient: 1

```

Figure 4: Executing the Code

2.3 Implementing Loops and Functions

Design an object-oriented C# program that:

- Uses a for loop to print numbers from 1 to 10.
- Uses a while loop to keep asking the user for input until they enter "exit".
- Defines and calls a function that calculates the factorial of a number provided by the user.

```
// For loop to print numbers 1-10
Console.WriteLine("Numbers 1 to 10:");
for (int i = 1; i <= 10; i++)
{
    Console.Write(i + " ");
}
Console.WriteLine("\n");

// While loop for user input
string userInput = "";
while (userInput.ToLower() != "exit")
{
    Console.Write("Enter a number to calculate factorial or 'exit' to quit: ");
    userInput = Console.ReadLine();

    if (userInput.ToLower() != "exit")
    {
        if (int.TryParse(userInput, out int number))
        {
            long factorial = CalculateFactorial(number);
            Console.WriteLine($"Factorial of {number} is {factorial}");
        }
        else
        {
            Console.WriteLine("Invalid input. Please enter a number or 'exit'.");
        }
    }
}
```

Figure 5: Code Block

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

Enter a number to calculate factorial or 'exit' to quit: 10
Factorial of 10 is 3628800
Enter a number to calculate factorial or 'exit' to quit: 20
Factorial of 20 is 2432902008176640000
Enter a number to calculate factorial or 'exit' to quit: |
```

Figure 6: Executing Programme

2.4 Object-Oriented Programming in C#

Created a class `Student` with:

- Properties: Name, ID, Marks.
- A constructor to initialize these values.
- A method `getGrade()` that returns the grade based on marks (A, B, C, etc.).
- A `Main()` method to create and display student details.

Create a subclass `StudentIITGN` from `Student` with:

- A new property: `Hostel_Name_IITGN`.
- A `Main()` method to create and display IITGN student details.

```
class Student
{
    // Properties
    2 references
    public string Name { get; set; }
    2 references
    public string ID { get; set; }
    6 references
    public double Marks { get; set; }

    // Constructor
    2 references
    public Student(string name, string id, double marks)
    {
        Name = name;
        ID = id;
        Marks = marks;
    }

    // Method to get grade
    1 reference
    public virtual string GetGrade()
    {
        if (Marks >= 90) return "A";
        else if (Marks >= 80) return "B";
        else if (Marks >= 70) return "C";
        if (Marks < 70) return "D";
    }
}
```

Figure 7: Student Class

```

// Derived StudentIITGN class
3 references
class StudentIITGN : Student
{
    // Additional property
    2 references
    public string HostelName { get; set; }

    // Constructor
    1 reference
    public StudentIITGN(string name, string id, double marks, string hostelName)
        : base(name, id, marks)
    {
        HostelName = hostelName;
    }

    // Override display details method
    4 references
    public override void DisplayDetails()
    {
        base.DisplayDetails();
        Console.WriteLine($"Hostel Name: {HostelName}");
    }
}

```

Figure 8: Student IITGn SubClass

```

Regular Student:
Student Name: John Doe
Student ID: S001
Marks: 85.5
Grade: B

IITGN Student:
Student Name: Jane Smith
Student ID: IIT001
Marks: 92.3
Grade: A
Hostel Name: Kameng

```

Figure 9: Executing Programme

2.5 Exception Handling

Modified the program from Activity 2 to handle exceptions:

- Used try-catch to handle division-by-zero errors.
- Ensured the program does not crash on invalid input.

```

1reference
public static void one()
{
    try
    {
        // Accept user input for two numbers
        Console.Write("Enter first number: ");
        double num1 = GetValidNumber();

        Console.Write("Enter second number: ");
        double num2 = GetValidNumber();

        // Perform operations
        double sum = num1 + num2;
        double difference = num1 - num2;
        double product = num1 * num2;
        double quotient = 0;

        // Division with error handling
        try
        {
            quotient = num1 / num2;
        }
        catch (DivideByZeroException)
        {
            Console.WriteLine("Warning: Division by zero attempted");
        }
    }
}

```

Figure 10: Exception Handling

```

Enter first number: Hello
Invalid input. Please enter a valid number: 5
Enter second number: 10

Results:
Sum: 15 (This is odd)
Difference: -5
Product: 50
Quotient: 0.5

Program execution completed.

```

Figure 11: Executing Programme

2.6 Debugging using Visual Studio Debugger

For Activities (2) to (5), visually illustrated program execution in Debug mode by inserting breakpoints at appropriate places

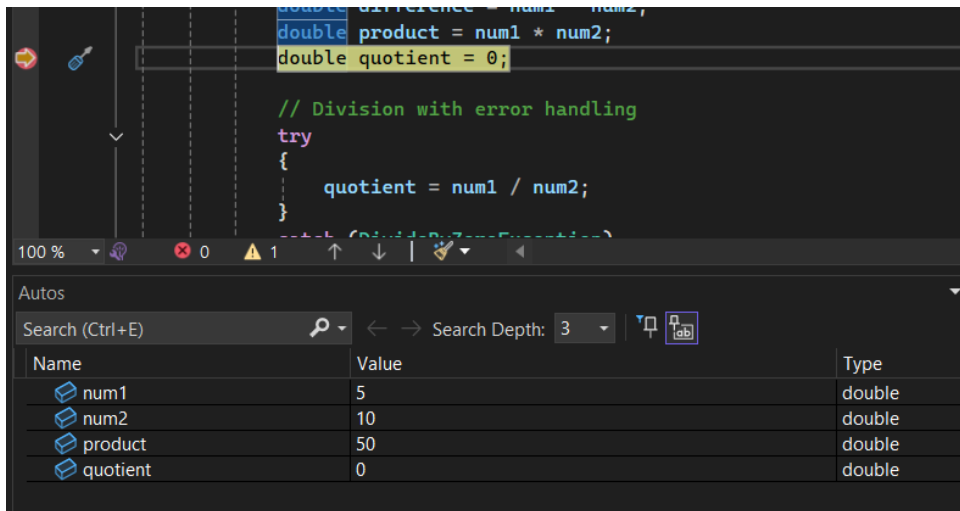


Figure 12: Breakpoint at variable quotient

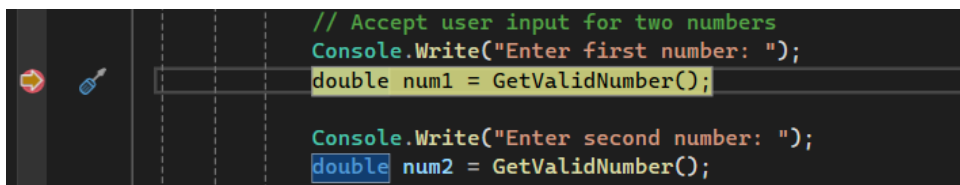


Figure 13: Breakpoint at Method Step Into

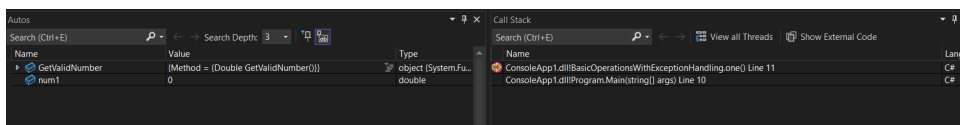


Figure 14: Executing the current method in stack Step Out

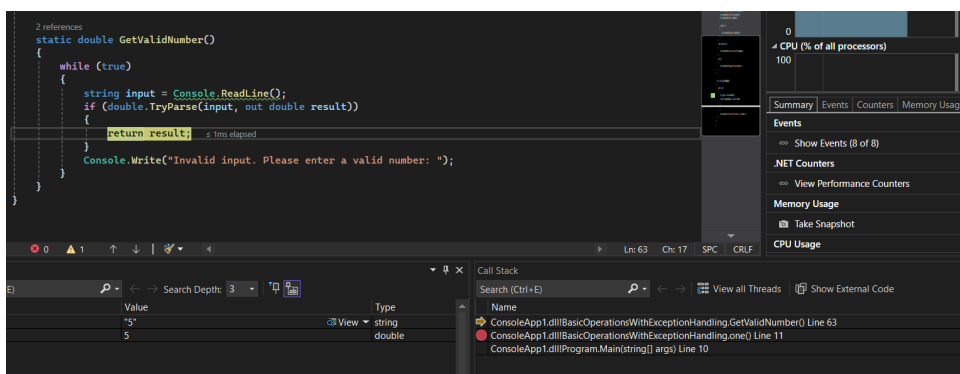


Figure 15: Stepping Into the Method

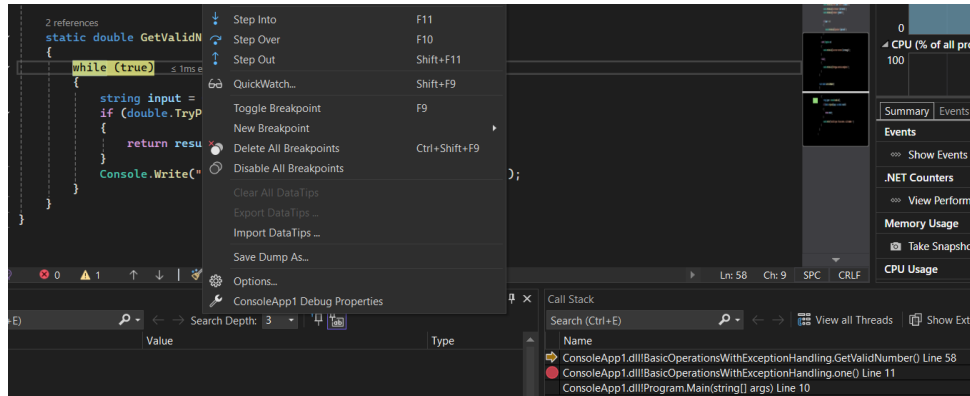


Figure 16: Using Step Over, Step into, Step Out

- **Step Into:** Allows you to enter the body of a function or method call line-by-line. Use this when you want to debug the internal implementation of a method.
- **Step Over:** Executes the current line of code but skips over method calls, treating them as a single step. Use this when you trust the method and don't want to debug inside it.
- **Step Out:** Runs the rest of the current method and breaks once it returns to the caller. Use this when you've stepped into a method and want to quickly exit and return to the previous context.

3 Results and Analysis

3.1 Outputs and Observations

- Successfully executed a C# console application in Visual Studio with appropriate input/output handling.
- Demonstrated the use of conditional statements (if-else) and loops (for, while) to build interactive logic.
- Applied object-oriented design principles, including encapsulation and inheritance, in the creation of **Student** and **StudentIITGN** classes.
- Ensured user input validation and safe execution through exception handling mechanisms using try-catch blocks.
- Explored debugging techniques, including breakpoints, step-in, step-out, and variable watch to inspect the execution state.
- Observed that well-structured code with proper class design is easier to debug, extend, and maintain.

3.2 Key Insights and Comparisons

- **Code Modularity:** Creating separate methods for each functionality (e.g., factorial calculation, input handling) increased modularity and made the codebase cleaner and more readable.

- **Error Handling:** Try-catch blocks played a crucial role in preventing runtime crashes and improving program robustness—especially in scenarios involving user input and division operations.
- **Debugging Process:** The Visual Studio debugger allowed a deeper understanding of program flow and logical errors. It helped analyze variable values in real-time and greatly reduced trial-and-error-based bug fixing.
- **Comparison with Other IDEs:** Visual Studio offers a more intuitive and feature-rich experience compared to lightweight IDEs like VS Code or online compilers, especially for .NET applications.

3.3 Performance and Usability

- The C# programs executed with minimal latency and were responsive to user inputs.
- The design of the programs (structured classes, clean methods) contributed to easier testing and result verification.
- OOP-based structure will facilitate the extension of functionality, such as integrating a GUI or database in future iterations.

4 Discussion and Conclusion

Challenges and Lessons Learned

- Initially, it was challenging to understand the syntax differences between C# and other languages like Java or Python, particularly in areas like property declarations and exception handling.
- Grasping Visual Studio's advanced debugging features required experimentation, but once mastered, they became invaluable for effective troubleshooting.
- Understanding the correct use of access modifiers (`public`, `private`) and object instantiation helped in reinforcing object-oriented concepts.
- Exception handling forced a mindset shift from reactive coding to proactive error prevention.

Summary and Takeaways

This lab provided a solid foundation in .NET development using C#. It allowed for hands-on experience with:

- The setup and execution of console applications.
- Core programming constructs like variables, loops, conditionals, and functions.
- Object-oriented principles, including encapsulation, inheritance, and method design.
- Debugging strategies and error handling mechanisms in a robust IDE environment.

Future Scope:

- Extend the console applications to Windows Forms or ASP.NET projects.
- Integrate file handling, database access (using Entity Framework), and unit testing frameworks.
- Explore advanced C# features such as LINQ, asynchronous programming with `async/await`, and generics.