Assignment 1: Multi-Stage Calculator

1. The Mission

 Welcome to TechStart Industries!

You've just been hired as a Junior Developer at TechStart Industries, a cutting-edge software company. Your manager, Sarah, has an urgent request: the company's old calculator system crashed during a critical client demo, and they need a replacement FAST!

But here's the twist - different departments need different features:

- The **Accounting Department** needs basic calculations for quick financial checks
- The **Engineering Team** wants validation and error handling (they're tired of crashes!)
- The **Research Division** requires scientific operations for their experiments
- And **Management** wants it polished and professional for client presentations

Sarah hands you a coffee and says: "Build this in stages. Show me progress after each department's requirements are met. We're counting on you!"

Due Date: [To be announced by instructor]
Submission: GitHub repository URL via course platform

In this assignment, you will build a console-based calculator application that gradually increases in complexity, meeting each department's needs. You'll apply the concepts learned in Session 1: variables, data types, console I/O, control flow (if/else, switch/case), and version control with Git.

2. Learning Objectives

By completing this assignment, you will:

- Practice working with variables and data types
- Implement control flow using if/else and switch/case statements
- Handle user input and output formatting
- Perform input validation and error handling
- Structure your code with meaningful comments
- Use Git for version control with incremental commits
- Document your work professionally

3. Prerequisites

Before starting, ensure you have:

- .NET 9.0 SDK installed
- JetBrains Rider (or another IDE)
- Git installed and configured
- A GitHub account

4. Assignment Structure

This assignment consists of **four stages** that build upon each other. Complete them in order, committing your progress after each stage.

Do not skip stages! Each stage teaches important concepts needed for the next.

5. Stage 1: Basic Calculator - Accounting Department Needs

"First things first," Sarah explains, "the Accounting team just needs the basics - addition, subtraction, multiplication, and division. They're doing quick calculations all day, so keep it simple!"

5.1. Requirements

Create a console application that:

1. Asks the user for the first number
2. Asks the user for the second number
3. Asks the user for an operation (+, -, *, /)
4. Performs the calculation using **if/else statements**
5. Displays the result in a clear format

5.2. Example Output

```
Welcome to the Basic Calculator!

Enter first number: 10
Enter second number: 5
Enter operation (+, -, *, /): +
```

```
Result: 10 + 5 = 15
```

5.3. Hints & APIs to Explore

- `Console.WriteLine()` - Output text to console
- `Console.ReadLine()` - Read user input as string
- `int.Parse()` or `Convert.ToInt32()` - Convert string to integer
- `double.Parse()` or `Convert.ToDouble()` - Convert string to double (for decimal numbers)

Use `double` instead of `int` for your numbers to handle decimal results (e.g., 5 / 2 = 2.5)

5.4. Git Tasks

After completing Stage 1:

```
git add .
git commit -m "Stage 1: Basic calculator with if/else"
git push
```

6. Stage 2: Enhanced Calculator - Engineering Team Requirements

The Engineering lead, Marcus, catches you in the hallway: "Hey, great start! But we need this to be more robust. The old calculator crashed every time someone divided by zero - embarrassing during client calls! Also, can you make it so we don't have to restart the app for every calculation? And we could really use the modulo operator for our work."

6.1. Requirements

Enhance your calculator to:

1. Replace if/else with **switch/case** for operation selection
2. Add the modulo operation (%)
3. Validate input:
   - Check for division by zero
   - Handle invalid operation symbols
4. Add a loop to allow multiple calculations without restarting
   - Ask "Do you want to perform another calculation? (y/n)"
   - Exit when user enters 'n'

6.2. Example Output

```
Welcome to the Enhanced Calculator!

Enter first number: 10
Enter second number: 0
Enter operation (+, -, *, /, %): /

Error: Division by zero is not allowed!

Do you want to perform another calculation? (y/n): y

Enter first number: 17
Enter second number: 5
Enter operation (+, -, *, /, %): %

Result: 17 % 5 = 2

Do you want to perform another calculation? (y/n): n

Thank you for using the calculator!
```

6.3. Hints & APIs to Explore

- `switch/case` statement - Better than multiple if/else for fixed options
- `while` loop or `do-while` loop - Repeat calculations
- `char.ToLower()` - Convert input to lowercase for 'y'/'n' comparison
- Comparison operators: `==`, `!=`
- Logical operators: `&&` (and), `||` (or)

Structure your switch/case like this:

```
switch(operation)
{
    case '+':
        result = num1 + num2;
        break;
    case '-':
        result = num1 - num2;
        break;
    // ... more cases
    default:
        Console.WriteLine("Invalid operation!");
        break;
}
```

6.4. Git Tasks

```
git add .
git commit -m "Stage 2: Add switch/case, validation, and loop"
git push
```

7. Stage 3: Scientific Features - Research Division Specifications

Dr. Chen from the Research Division sends you an email: "Excellent progress! Our team needs more advanced mathematical functions for our experiments. We're calculating square roots, powers, and we need to keep track of our calculation history. Could you add a menu system so we can easily switch between different operation types? This will be a game-changer for our lab work!"

7.1. Requirements

Add advanced features:

1. Create a **menu system** with the following options:
    - 1: Basic operations (+, -, *, /, %)
    - 2: Square root
    - 3: Power (x^y)
    - 4: View calculation history
    - 5: Exit
2. Implement square root operation:
    - Use only one number as input
    - Validate that the number is not negative
    - Display error message for negative numbers
3. Implement power operation:
    - Ask for base and exponent
    - Calculate base^exponent
4. Add calculation history:
    - Store the last 5 calculations as strings
    - Display them when user selects "View history"
    - Use variables (e.g., `history1`, `history2`, etc.)

7.2. Example Output

```
=== Scientific Calculator ===

Menu:
1. Basic Operations (+, -, *, /, %)
2. Square Root
3. Power (x^y)
4. View History
5. Exit

Choose an option: 2

Enter a number: 16

Result: √16 = 4

Menu:
1. Basic Operations (+, -, *, /, %)
2. Square Root
3. Power (x^y)
4. View History
5. Exit

Choose an option: 3

Enter base: 2
Enter exponent: 8

Result: 2^8 = 256

Menu:
1. Basic Operations (+, -, *, /, %)
2. Square Root
3. Power (x^y)
4. View History
5. Exit

Choose an option: 4

=== Calculation History ===
1. √16 = 4
2. 2^8 = 256

Menu:
1. Basic Operations (+, -, *, /, %)
2. Square Root
3. Power (x^y)
4. View History
5. Exit
```

```
Choose an option: 5

Thank you for using the Scientific Calculator!
```

7.3. Hints & APIs to Explore

- `Math.Sqrt(number)` - Calculate square root
- `Math.Pow(base, exponent)` - Calculate power
- String concatenation or interpolation for history: `$"{number} = {result}"`
- Nested switch/case or if/else for menu structure

For storing history without arrays (which you'll learn later), use separate variables:

```
string history1 = "";
string history2 = "";
string history3 = "";
string history4 = "";
string history5 = "";

// After each calculation, shift history:
history5 = history4;
history4 = history3;
history3 = history2;
history2 = history1;
history1 = $"{num1} + {num2} = {result}";
```

7.4. Git Tasks

```
git add .
git commit -m "Stage 3: Add menu system, scientific operations, and history"
git push
```

8. Stage 4: Polish & Documentation - Management Presentation Ready

Sarah calls you into her office with a smile: "The teams love it! Now we need to get this ready for a client presentation next week. Let's make sure the code is clean, well-documented, and handles every edge case gracefully. Can you also create professional documentation? We want to show this off as an example of our development quality!"

8.1. Requirements

1. **Code Quality:**
   - Add meaningful comments explaining each section
   - Use proper naming conventions (camelCase for variables)
   - Format numbers to 2 decimal places where appropriate
   - Improve error messages to be user-friendly
2. **Input Validation:**
   - Handle non-numeric input gracefully
   - Try using `TryParse` instead of `Parse` to avoid crashes
3. **README.md:**
   - Create a README.md file in your repository
   - Include:
     - Project title and description
     - Features list
     - How to run the program
     - Example usage
     - Your name and date
4. **Final Testing:**
   - Test all operations thoroughly
   - Try to break your program with invalid inputs
   - Fix any bugs you find

8.2. Example README.md Structure

```
# Multi-Stage Calculator

A console-based scientific calculator built with C# and .NET 9.0.

## Features

- Basic arithmetic operations (+, -, *, /, %)
- Scientific operations (square root, power)
- Input validation and error handling
- Calculation history (last 5 calculations)
- User-friendly menu system

## How to Run

1. Clone this repository
```

```
2. Open the solution in JetBrains Rider
3. Press F5 to run
4. OR use command line: `dotnet run`

## Example Usage

```
Enter first number: 10
Enter second number: 5
Enter operation (+, -, *, /, %): +
Result: 10 + 5 = 15
```

Author

[Your Name]
Date: [Date]
Session: 1 - Professional Programmer Software Engineering I
```

8.3. Hints & APIs to Explore

- `double.TryParse(input, out result)` - Safer than Parse, returns bool
- `Math.Round(number, 2)` - Round to 2 decimal places
- String formatting: `string.Format("{0:F2}", number)` or `$"{number:F2}"`
- `Console.Clear()` - Clear console for better UX

Use TryParse to handle invalid input:

```
Console.Write("Enter a number: ");
string input = Console.ReadLine();

if (double.TryParse(input, out double number))
{
    // Valid number, use 'number' variable
    Console.WriteLine($"You entered: {number}");
}
else
{
    // Invalid input
    Console.WriteLine("Error: Please enter a valid number!");
}
```

8.4. Git Tasks

```
git add .
git commit -m "Stage 4: Polish code, add documentation and validation"
git push
```

9. Submission Requirements

Submit your assignment by providing:

1. **GitHub Repository URL** containing:
   - Your complete calculator code
   - README.md file
   - Multiple commits showing progression through stages
   - .gitignore file (should be created automatically by Rider)
2. **Ensure your repository:**
   - Is public (or grant instructor access if private)
   - Has a descriptive repository name (e.g., `calculator-assignment1`)
   - Contains clear commit messages
   - Includes all required files

10. Grading Criteria

Your assignment will be evaluated on:

Criteria	Points	Description
Stage 1 Completion	15	Basic calculator works with if/else
Stage 2 Completion	20	Switch/case, validation, and loop implemented
Stage 3 Completion	25	Menu system, scientific operations, history work
Stage 4 Completion	15	Code quality, documentation, input validation
Git Usage	10	Meaningful commits at each stage, good commit messages
Code Quality	10	Proper naming, comments, formatting
README Documentation	5	Clear, complete, professional
Total	**100**	

11. Tips for Success

- **Start early** - Don't wait until the last minute

- **Commit often** - After each stage or significant change
- **Test thoroughly** - Try different inputs, including invalid ones
- **Ask for help** - If stuck for more than 30 minutes, reach out to instructor or classmates
- **Read error messages** - They often tell you exactly what's wrong
- **Use descriptive variable names** - `firstNumber` is better than `n1`

12. Common Pitfalls to Avoid

- Forgetting to use `double` instead of `int` for division
- Not handling division by zero
- Using `Parse` instead of `TryParse` (causes crashes on invalid input)
- Forgetting `break;` statements in switch/case
- Not committing to Git regularly
- Copying code without understanding it

13. Optional Challenges (Bonus)

If you finish early and want extra practice:

1. Add percentage calculation (e.g., "What is 20% of 150?")
2. Add memory functions (M+, M-, MR, MC)
3. Support negative numbers in all operations
4. Add factorial calculation
5. Create a "theme" with colored console output using `Console.ForegroundColor`

14. Resources

- C# Documentation
- Math Class Documentation
- Console Class Documentation
- Git Documentation

15. Questions?

If you have questions about the assignment:

1. Check the lecture notes and slides
2. Review the hints and API suggestions above

3. Search the Microsoft documentation
4. Ask in the course forum or during office hours
5. Email the instructor

Remember: You're not just completing an assignment - you're saving TechStart Industries from calculator chaos! Each stage represents real requirements from real departments. Show them what you can do!

Good luck and happy coding!