INSTITUTE OF AERONAUTICAL ENGINEERING
(Autonomous)
Dundigal - 500 043, Hyderabad, Telangana

COURSE CONTENT

PROGRAMMING WITH OBJECTS LABORATORY								
III Semester: CSE / CSE (AI & ML) / CSE (DS) / CSE (CS) / IT								
Course Code	**Category**	**Hours / Week**			**Credits**	**Maximum Marks**		
		L	**T**	**P**	**C**	**CIA**	**SEE**	**Total**
AITD02	**Core**	-	-	2	1	40	60	100
Contact Classes: Nil	**Tutorial Classes: Nil**	**Practical Classes: 45**				**Total Classes: 45**		
Prerequisite: Essentials of Problem Solving								

I. COURSE OVERVIEW:

Windows, a dominant OS, provides a user-friendly platform and robust development environ- ment via the .NET Framework. Developers leverage C#, a versatile language on .NET, for diverse applications across desktops, servers, web, mobile, and IoT. C# accelerates development across the software lifecycle, from enterprise solutions to responsive web apps. This lab course equips students with core C# skills and practical experience in real-world scenarios, preparing them for success in software development.

II. COURSE OBJECTIVES:

The students will try to learn

I. The basic concepts of object oriented programming.
II. The application of object oriented features for developing flexible and extensible applications.
III. The Graphical User Interface (GUI) with database connectivity to develop web applications.

III. COURSE OUTCOMES:

At the end of the course students should be able to:

CO 1	**Core Programming Skills:** Develop core programming skills in C# for basic applica- tion development.
CO 2	**Function Design and Implementation:** Design and implement functions to enhance code modularity and reusability.
CO 3	**Advanced Text and Service Integration:** Utilize advanced techniques for pattern matching and integrating external services.
CO 4	**Object-Oriented Design:** Apply principles of object-oriented design to create struc- tured and scalable applications.
CO 5	**Error Handling and File Management:** Implement robust error handling and manage file operations effectively.
CO 6	**Concurrency Management:** Create and manage concurrent processes to improve ap- plication performance.

Introduction to the Course C# & .NET

Note: Students are encouraged to bring their own laptops for laboratory practice sessions.

Course Overview

Windows, a dominant OS, provides a user-friendly platform and robust development environment via the .NET Framework. Developers leverage C#, a versatile language on .NET, for diverse applications across desktops, servers, web, mobile, and IoT. C# accelerates development across the software lifecycle, from enterprise solutions to responsive web apps. This lab course equips students with core C# skills and practical experience in real-world scenarios, preparing them for success in software development.

Software Requirements

Windows 10 Pro or Enterprise is recommended for optimal performance.
While Windows 10 Home is sufficient for introductory C# courses, advanced courses re- quire Windows 10 Pro or higher.

Visual Studio Community Edition 2017 or later is required and available for free download from Microsoft's official website: `https://www.visualstudio.com/`

Hardware Requirements

- Intel/AMD multi-core processor (i3 or better)
- Minimum 8GB RAM, 16GB preferred
- At least 50GB of free hard drive space for Visual Studio and project files

Prerequisites

Completion of "Visual Studio" or equivalent experience is recommended. No prior programming knowledge or experience is necessary.

Next Steps

Upon completion of this course, consider taking C# Programming certification

Getting Started Exercises

Note: Students are encouraged to bring their own laptops for laboratory practice sessions.

1.1 Installing Visual Studio and Setting up the .NET Environ- ment

The objective of this lab is to guide students through the installation of Visual Studio and the setup of the .NET environment necessary for C# programming.

Requirements

- A computer with internet access.
- Windows operating system (recommended) or macOS.
- Administrative privileges to install software.

Steps to Install Visual Studio

1. **Download Visual Studio:**
   - Visit the official Visual Studio website: `https://visualstudio.microsoft. com/`
   - Click on the "Download" button for the Community edition (free version).
2. **Run the Installer:**
   - Once the download is complete, run the installer.
   - Follow the on-screen instructions to continue with the installation.
3. **Select Workloads:**
   - In the installer, you will see a list of workloads. Select *".NET desktop development"*.
   - Optionally, you can select other workloads if needed for future use, such as *"ASP.NET and web development"*.
4. **Install:**
   - Click on the "Install" button to begin the installation process.
   - Wait for the installation to complete. This may take some time depending on your internet speed and system performance.
5. **Launch Visual Studio:**
   - Once the installation is complete, launch Visual Studio from the Start menu or desk- top shortcut.
   - Sign in with your Microsoft account if prompted.
   - Choose the default environment settings when prompted.

Setting up the .NET Environment

1. **Create a New Project:**
   - Click on “Create a new project” from the Visual Studio start page.
   - Select the template *“Console App (.NET Core)”*.
   - Click “Next”.
2. **Configure Your Project:**
   - Enter a name for your project (e.g., *“HelloWorld”*).
   - Choose a location to save your project.
   - Click “Create”.
3. **Verify the Setup:**
   - Once the project is created, you will see the *“Program.cs”* file open in the code editor.
   - Replace the existing code with the following:

```
using System;

namespace HelloWorld
{
 class Program
 {
  static void Main(string[] args)
  {
   Console.WriteLine("Hello, World!");
  }
 }
}
```

   - Press `Ctrl + F5` to run the application.
   - Verify that the output window displays *“Hello, World!”*.

By completing this lab, students will have successfully installed Visual Studio, set up the .NET environment, and created their first C# console application. This foundational setup is crucial for all future labs and projects in this course.

1.2 Distance based on Speed and Time

Calculate the distance traveled by an object based on the speed (v) and time (t) entered by the user. The formula to calculate distance (d) is given by:

$$d = v \times t$$

Input

Your program should prompt the user to enter the following:

- The speed of the object in meters per second (m/s).
- The time duration in seconds (s) for which the object traveled at that speed. The program should then compute and display the distance traveled by the object.

Requirements

- Use appropriate data types for variables.
- Ensure the program handles decimal values for speed and time.
- Include error handling for non-positive values of speed or time (optional).

Sample Output

Example 1:

```
Enter the speed of the object
(m/s): 10.5 Enter the time
duration (s): 8.2

Distance traveled: 86.1 meters
```

Example 2:

```
Enter the speed of the object
(m/s): 0 Enter the time
duration (s): 5.5

Invalid input: Speed must be greater than zero.
```

1.3 Cube Root

Calculate the cube root of a given number.

Input

The program should:

- Prompt the user to enter a number.
- Calculate the cube root of the number.
- Display the result.

Requirements

- Implement a function `CalculateCubeRoot(double number)` that returns the cube root of the given number.
- Ensure the program handles valid numerical inputs.

Sample Output

Example 1:

```
Enter the
number: 27
Cube root
of 27 is 3
```

Example 2:

```
Enter the
number: 64
Cube root
of 64 is 4
```

Example 3:

```
Enter the
number: -8
Cube root
of -8 is -2
```

1.4 Random Number Genrator

Generate and display a sequence of random numbers.

Input

- Use the `System.Random` class for generating random numbers.
- Prompt the user to enter the number of random numbers to generate.
- Specify a range for the random numbers (e.g., between 1 and 100).
- Display each generated random number in the sequence.

Your program should ensure that each random number generated is unique within the spec- ified range.

Requirements

- Use appropriate data types and structures to store and display the random numbers.
- Handle user input validation to ensure the number of random numbers is positive.
- Implement error handling for any unexpected inputs or conditions (optional).

Sample Output

Example:

```
Enter the number of random numbers to
generate: 5 Enter the minimum value
of the range: 1
Enter the maximum value of the range: 100

Generated random numbers:
45 12 89 27 56
```

1.5 Nullable Data Types

Demonstrates the use of Nullable data types. Nullable data types allow variables to have an additional value, `null`, which represents undefined or unknown values.

Input

- Declare and initialize nullable variables of different data types (`int?`, `float?`, `bool?`).
- Assign both `null` values and non-null values to these nullable variables.
- Display the values of these nullable variables using formatted output.
- Illustrate how nullable types handle initialization and assignment.

Ensure your program includes comments or documentation to explain the purpose of each variable and the behavior of nullable data types.

Requirements

- Use appropriate syntax for declaring and initializing nullable data types in C#.
- Demonstrate the advantages of using nullable data types, such as handling database null values or optional parameters.
- Include formatted output to clearly display the values of nullable variables.

Sample Output

```
Nullable
Integers : ,
786 Nullable
Floats      :
3.14,
Nullable
boolean  :
```

1.6 Permutations (nPr)

Calculate the number of permutations (nPr) of a given set of `n` items taken `r` at a time. The formula to calculate permutations is given by:

$$nPr = \frac{n!}{(n-r)!}$$

where `n!` denotes the factorial of `n`.

Input

- Prompt the user to enter two integers `n` and `r`.
- Calculate the number of permutations using the provided formula.
- Display the result.

Requirements

- Implement a function `Factorial(int x)` that returns the factorial of `x`.
- Implement a function `CalculatePermutations(int n, int r)` that returns the number of permutations.
- Ensure the program handles edge cases such as `r > n` by displaying an appropriate message.

Sample Output

Example 1:

```
Enter the value of n: 5
Enter the value of r: 3
Number of permutations (5P3) = 60
```

Example 2:

```
Enter the value of n: 6
Enter the value of r: 6
Number of permutations (6P6) = 720
```

Example 3:

```
Enter the value of n: 4
Enter the value of r: 5
Error: r cannot be greater than n.
```

1.7 Binary Sum

Write a C# program to add two binary numbers.

Input

- Prompt the user to enter two binary numbers as strings.
- Validate the input to ensure that it contains only binary digits (0 and 1).
- Perform the binary addition.
- Display the sum as a binary number.

Requirements

- Implement a function `IsBinary(string binary)` that returns `true` if the string contains only binary digits, otherwise returns `false`.
- Implement a function `AddBinary(string a, string b)` that performs the binary addition and returns the result as a string.
- Ensure the program handles cases where the input is not valid binary numbers by displaying an appropriate message.

Sample Output

Example 1:

```
Enter the first binary number: 101
Enter the second binary number: 110
Sum of 101 and 110 is 1011
```

Example 2:

```
Enter the first binary number: 1001
Enter the second binary number: 11
Sum of 1001 and 11 is 1100
```

Example 3:

```
Enter the first binary number: 1010
Enter the second binary number: 102
Error: One or both inputs are not valid binary numbers.
```

1.8 Explore Bitwise Operators

Demonstrate the use of bitwise operators.

Input

The program should:

- Prompt the user to enter two integers.
- Perform and display the result of the following bitwise operations on the entered integers:
  - AND (&)
  - OR (|)
  - XOR (^)
  - NOT (~)
  - Left Shift (<<)
  - Right Shift (>>)

Requirements

- Implement functions for each of the bitwise operations.
- Ensure the program handles valid integer inputs.

Sample Output

Example:

```
Enter the first integer: 12
Enter the second integer: 5

Bitwise AND (12 & 5) = 4
Bitwise OR (12 | 5) = 13
Bitwise XOR (12 ^ 5) = 9
Bitwise NOT (~12) = -13
Left Shift (12 << 2) = 48
Right Shift (12 >> 2) = 3
```

1.9 No Math

The program should

- Calculate the square root of a given number without using the built-in `Math.Sqrt()` method.
- Find the absolute value of a given number without using the built-in `Math.Abs()` method.

Requirements

- Implement a function `CalculateSquareRoot(double number)` that returns the square root of the given number using an iterative approach (e.g., Newton's method).
- Implement a function `FindAbsoluteValue(double number)` that returns the absolute value of the given number.
- Ensure the program handles valid numerical inputs.

Sample Output

Example 1:

```
Enter a number to find its square root: 16
Square root of 16 is approximately: 4

Enter a number to find its absolute value: -8.5
Absolute value of -8.5 is: 8.5
```

Example 2:

```
Enter a number to find its square root: 20
Square root of 20 is approximately: 4.4721

Enter a number to find its absolute value: 3.7
Absolute value of 3.7 is: 3.7
```

1.10 Edge cases

Explore edge cases using the `Math.Pow()` method. The program should:

- Calculate and display the results for the following scenarios:
  1. Calculate `Math.Pow(double.PositiveInfinity, 2)`.
  2. Calculate `Math.Pow(double.NegativeInfinity, 2)`.
  3. Calculate `Math.Pow(double.MinValue, 0)`.
  4. Calculate `Math.Pow(double.NaN, 2)`.
- Print the results of each calculation.

Requirements

- Implement a C# program that performs the above calculations using the `Math.Pow()` method.
- Ensure the program handles special cases such as infinity and NaN appropriately.

Sample Output

Example Output:

```
Number1 : Infinity
Number2 : Infinity
Number3 : 1
Number4 : NaN
```

Try

1. Calculate `Math.Pow(0, 0)`.
2. Calculate `Math.Pow(10, -1)`.
3. Calculate `Math.Pow(1, double.MaxValue)`.

Arrays

Note: Students are encouraged to bring their own laptops for laboratory practice sessions.

2.1 Arrow Pattern

Create a function `Arrow(num)` that generates a pattern as a 2D array for a given number `num`.

- For `Arrow(3)`, the function should return `[">", ">>", ">>>", ">>", ">"]`.
- For `Arrow(4)`, the function should return `[">", ">>", ">>>", ">>>>", ">>>>", ">>>", ">>", ">"]`.

Criteria

- The function argument `num` will always be a number greater than 0.
- Odd numbers will produce a pattern with a single “peak”.
- Even numbers will produce a pattern with two “peaks”.

Hints

- Identify that the pattern consists of increasing and then decreasing sequences of arrows.
- Create a loop to generate the increasing part of the pattern from 1 up to `num`.
- Create a second loop to generate the decreasing part of the pattern from `num-1` down to 1.
- Use a list to collect the pattern strings and then convert it to an array before returning.
- Consider how the pattern behaves for the smallest values of `num` (like 1).

2.2 Working 9 to 5

Write a function that calculates overtime and pay associated with overtime.

Working Hours

- Working hours: 9 AM to 5 PM (regular hours).
- After 5 PM is considered overtime.

Input

Your function receives an array with 4 values:

- Start of working day, in decimal format (24-hour day notation).
- End of working day (same format).
- Hourly rate.
- Overtime multiplier.

Output

Your function should return the total earnings for that day, rounded to the nearest hundredth, formatted as:

$$\$ \textit{ earned that day}$$

Testcases

- `OverTime([9, 17, 30, 1.5])` ⇒ `"$240.00"`
- `OverTime([16, 18, 30, 1.8])` ⇒ `"$84.00"`
- `OverTime([13.25, 15, 30, 1.5])` ⇒ `"$52.50"`

Explanation of the Second Example

- From 16:00 to 17:00 is regular hours: $1 \times 30 = 30$
- From 17:00 to 18:00 is overtime: $1 \times 30 \times 1.8 = 54$
- Total earnings: $30 + 54 = 84 \Rightarrow \$84.00$

Hints

- Identify the regular working hours (9 AM to 5 PM) and determine if the work falls within these hours or extends into overtime.
- Calculate the total number of regular hours and overtime hours based on the start and end times.
- Compute the earnings for regular hours and overtime hours separately.
- Use the hourly rate for regular hours and apply the overtime multiplier for overtime hours to calculate the total earnings.
- Round the final earnings to the nearest hundredth and format the result as a monetary value.

2.3 New Driving License

You have to get a new driver's license. You show up at the office at the same time as four other people. The office says they will see everyone in alphabetical order and it takes 20 minutes for them to process each new license.

All of the agents are available now, and they can each see one customer at a time. How long will it take for you to walk out with your new license?

Your input will be a string of your name me, an integer of the number of available agents, and a string of the other four names separated by spaces others.

Return the number of minutes it will take to get your license.

Examples

```
License("Eric", 2, "Adam Caroline Rebecca
                Frank") ⇒ 40
```

As you are in the second group of people.

```
License("Zebediah", 1, "Bob Jim Becky Pat")
                    ⇒ 100
```

As you are the last person.

```
License("Aaron", 3, "Jane Max Olivia Sam") ⇒ 20
```

As you are the first.

Hints

- Combine your name with the list of other names and sort them alphabetically to determine the order of processing.
- Determine your position in the sorted list to find out when you will be processed.
- Calculate the number of people before you in the sorted list to determine how many people will be processed before you.
- Given the number of available agents, distribute the workload so that each agent handles one person at a time.
- Compute the time it will take for you to be processed based on the number of people ahead of you and the processing time per person.

2.4 Who Won the League?

The 2019/20 season of the English Premier League (EPL) saw Liverpool FC win the title for the first time despite their bitter rivals, Manchester United, having won 13 titles!

Create a function that receives an alphabetically sorted array of the results achieved by each team in that season and the name of one of the teams. The function should return a string giving the final position of the team, the number of points achieved, and the goal difference (see examples for precise format).

The results table is given in the following format:

Team	Played	Won	Drawn	Lost	G/Diff
Arsenal	38	14	14	10	8
Aston Villa	38	9	8	21	-26
Bournemouth	38	9	7	22	-26
Brighton	38	9	14	15	-15
Burnley	38	15	9	14	-7
Chelsea	38	20	6	12	15
Crystal Palace	38	11	10	17	-19
Everton	38	13	10	15	-12
Leicester City	38	18	8	12	26
Liverpool	38	32	3	3	52
Manchester City	38	26	3	9	67
Manchester Utd	38	18	12	8	30
Newcastle	38	11	11	16	-20
Norwich	38	5	6	27	-49
Sheffield Utd	38	14	12	12	0
Southampton	38	15	7	16	-9
Tottenham	38	16	11	11	14
Watford	38	8	10	20	-28
West Ham	38	10	9	19	-13
Wolves	38	15	14	9	11

Examples

```
string[] table = [
"Arsenal, 38, 14, 14, 10, 8",
"Aston Villa, 38, 9, 8, 21, -26",
...
"West Ham, 38, 10, 9, 19, -13",
"Wolves, 38, 15, 14, 9, 11"
]

EPLResult(table, "Liverpool") -->
"Liverpool came 1st with 99 points and a goal difference of 52!"

EPLResult(table, "Manchester Utd")-->
"Manchester Utd came 3rd with 66 points and a goal difference of
30!"

EPLResult(table, "Norwich") -->
"Norwich came 20th with 21 points and a goal difference of -49!"
```

Hints

- Each result in the source table is a comma separated string.

- Score 3 points for a win and 1 point for a draw.
- If teams are tied on points, their position is determined by better goal difference.
- The input table should be considered immutable - do not make any changes to it!

2.5 The Day Rob Was Born in Dutch

I was born on the 21st of September in the year of 1970. That was a Monday. Where I was born that weekday is called *maandag*.

Write a method that, when passed a date as `year/month/day`, returns the date's weekday name in the Dutch culture. The culture identifier to use is `"nl-NL"`, not `"nl-BE"`.

You can assume the specified date is valid. Looking at the tests and doing a switch statement or similar is considered cheating. `System.Globalization.CultureInfo` should be used.

The method may be used to get the name of the Dutch weekday of other memorable days too, like in the examples below:

Examples

```
WeekdayRobWasBornInDutch(1970, 9, 21) --> "maandag"

WeekdayRobWasBornInDutch(1945, 9, 2) --> "zondag"

WeekdayRobWasBornInDutch(2001, 9, 11) --> "dinsdag"
```

Hints

- Use the `System.Globalization.CultureInfo` class to handle Dutch culture settings.
- Create a `DateTime` object using the provided year, month, and day.
- Use the `ToString` method of the `DateTime` object with the Dutch culture to get the weekday name in Dutch.
- Ensure that you specify the culture identifier `"nl-NL"` to get the correct Dutch weekday names.

2.6 Smallest Missing Positive Integer

Given an array of integers, return the smallest positive integer not present in the array. Here is a representative example. Consider the array:

```
{ -2, 6, 4, 5, 7, -1, 7, 1, 3, 6, 6, -2, 9, 10, 2, 2 }
```

After reordering, the array becomes:

```
{ -2, -2, -1, 1, 2, 2, 3, 4, 5, 6, 6, 6, 7, 7, 9, 10 }
```

... from which we see that the smallest missing positive integer is 8.

Examples

```
MinMissPos({ -2, 6, 4, 5, 7, -1, 1, 3, 6, -2, 9, 10, 2, 2 })
-> 8
// After sorting, the array becomes
// { -2, -2, -1, 1, 2, 2, 3, 4, 5, 6, 6, 7, 9, 10 }
// So the smallest missing positive integer is 8

MinMissPos({ 5, 9, -2, 0, 1, 3, 9, 3, 8, 9 })
-> 2
// After sorting, the array becomes
// { -2, 0, 1, 3, 3, 5, 8, 9, 9, 9 }
// So the smallest missing positive integer is 2

MinMissPos({ 0, 4, 4, -1, 9, 4, 5, 2, 10, 7, 6, 3, 10, 9 })
-> 1
// After sorting, the array becomes
// { -1, 0, 2, 3, 4, 4, 4, 5, 6, 7, 9, 9, 10, 10 }
// So the smallest missing positive integer is 1
```

Hints

For the sake of clarity, recall that 0 is not considered to be a positive number.

2.7 Briefcase Lock

A briefcase has a 4-digit rolling-lock. Each digit is a number from 0-9 that can be rolled either forwards or backwards.

Create a function that returns the smallest number of turns it takes to transform the lock from the current combination to the target combination. One turn is equivalent to rolling a number forwards or backwards by one.

To illustrate:

```
current-lock: 4089
target-lock: 5672
```

What is the minimum number of turns it takes to transform 4089 to 5672?

```
4 -> 5
4 -> 5  // Forward Turns: 1 <- Min
4 -> 3 -> 2 -> 1 -> 0 -> 9 -> 8 -> 7 -> 6 -> 5  // Backward Turns:
9

0 -> 6
0 -> 1 -> 2 -> 3 -> 4 -> 5 -> 6  // Forward Turns: 6
0 -> 9 -> 8 -> 7 -> 6  // Backward Turns:  4  <- Min

8 -> 7
8 -> 9 -> 0 -> 1 -> 2 -> 3 -> 4 -> 5 -> 6
```

```
                                                -> 7  // Forward Turns:
                                                9
8 -> 7  // Backward Turns: 1  <- Min

9 -> 2
9 -> 0 -> 1 -> 2  // Forward Turns: 3  <- Min
9 -> 8 -> 7 -> 6 -> 5 -> 4 -> 3 -> 2  // Backward Turns: 7

It takes 1 + 4 + 1 + 3 = 9 minimum turns to change the lock from
4089 to 5672.
```

Examples

```
MinTurns("4089", "5672") -> 9

MinTurns("1111", "1100") -> 2

MinTurns("2391", "4984") -> 10
```

Hints

- Both locks are in string format.
- A 9 rolls forward to 0, and a 0 rolls backwards to a 9.

2.8 Keeping Count

Given a number, create a function which returns a new number based on the following rules:

- For each digit, replace it by the number of times it appears in the number.
- The final instance of the number will be an

  integer, not a string. The following is a working

  example:

```
DigitCount(136116) -> 312332
```

Explanation:

- The number 1 appears thrice, so replace all 1s with 3s.
- The number 3 appears only once, so replace all 3s with 1s.
- The number 6 appears twice, so replace all 6s with 2s.

Examples

- DigitCount(221333) -> 221333

- DigitCount(136116) -> 312332
- DigitCount(2) -> 1

Hints

- Each test will have a positive whole number in its parameter.

2.9 Break Point

A number has a breakpoint if it can be split in a way where the digits on the left side and the digits on the right side sum to the same number.

For instance, the number 35190 can be sliced between the digits 351 and 90, since $3 + 5 + 1 = 9$ and $9 + 0 = 9$. On the other hand, the number 555 does not have a breakpoint (you must split between digits).

Create a function that returns true if a number has a breakpoint, and false otherwise.

Examples

- BreakPoint(159780) → true
- BreakPoint(112) → true
- BreakPoint(1034) → true
- BreakPoint(10) → false
- BreakPoint(343) → false

Hints

- Read each digit as only one number.
- Check the Resources tab for a hint.

2.10 ATM Transactions

Implement an ATM (Automated Teller Machine) transaction program in C# that simulates basic banking operations. The program should:

- Begin by prompting the user to enter a PIN to access the ATM services.
- Validate the entered PIN against a predefined PIN (e.g., 4040).
- If the PIN is correct, provide the following options to the user:
    1. Check balance: Display the current balance in the account.
    2. Withdraw money: Prompt the user to enter the amount to withdraw. Ensure withdrawal amount is in multiples of 100 and does not exceed the available balance.
    3. Deposit money: Prompt the user to enter the amount to deposit and update the current balance accordingly.
    4. Change PIN: Allow the user to change their PIN by verifying the current PIN and entering a new PIN.

- Handle incorrect PIN entries and invalid user inputs appropriately.

Requirements

- Implement the ATM program in C# with proper validation and error handling.
- Ensure that withdrawal amounts are multiples of 100 and do not exceed the available balance.
- Display appropriate messages for each transaction and user interaction.

Sample Output

```
Enter the pin:
> 4040
1. To check balance
2. To withdraw money
3. To deposit money
4. To
change
the pin
Enter
your
choice:
> 1
The current balance in your account is 10000

Enter the pin:
> 4040
1. To check balance
2. To withdraw money
3. To deposit money
4. To
change
the pin
Enter
your
choice:
> 2
Enter the amount to withdraw:
> 5000
Please collect the
cash: 5000 The
current balance is
now: 5000

Enter the pin:
> 4040
1. To check balance
2. To withdraw money
3. To deposit money
4. To
change
the pin
```

```
Enter
your
choice:
> 3
Enter the amount to be deposited:
> 2000
The current balance in the account is 7000

Enter the pin:
> 4040
1. To check balance
2. To withdraw money
3. To deposit money
4. To
change
the pin
Enter
your
choice:
> 4
Want to change
your pin Enter
your previous
pin:
> 4040
Enter your new pin:
> 1234
Your pin is changedEnter the pin:
> 1234
1. To check balance
2. To withdraw money
3. To deposit money
4. To
change
the pin
Enter
your
choice:
> 5
Please select correct option
```

Hints

- Test the program with incorrect PIN entries.
- Test withdrawing amounts that are not multiples of 100.
- Test depositing different amounts and verifying the updated balance.

Strings Manipulations

Note: Students are encouraged to bring their own laptops for laboratory practice sessions.

3.1 Find the Bomb

Create a function that finds the word *"bomb"* in the given string (not case sensitive). If found, return **"Duck!!!"**, otherwise, return **"There is no bomb, relax."**.

Examples

```
Bomb("There is a bomb.")
-> "Duck!!!"

Bomb("Hey, did you think there is a bomb?")
-> "Duck!!!"

Bomb("This goes boom!!!")
-> "There is no bomb, relax."
```

Hints

- "bomb" may appear in different cases (i.e., uppercase, lowercase, mixed).

3.2 Finding Nemo

Finding Nemo

You're given a string of words. You need to find the word *"Nemo"*, and return a string like this:
"I found Nemo at [the order of the word you find Nemo]!". If you can't find Nemo, return **"I can't find Nemo :("**.

Examples

```
FindNemo("I am finding Nemo !")
-> "I found Nemo at 4!"

FindNemo("Nemo is me")
-> "I found Nemo at 1!"
```

```
FindNemo("I Nemo am")
-> "I found Nemo at 2!"
```

Hints

- ! , ? . are always separated from the last word.
- Nemo will always look like Nemo, and not NeMo or other capital variations.
- Nemo's, or anything that says Nemo with something behind it, doesn't count as Finding Nemo.
- If there are multiple Nemo's in the sentence, only return the first one.

3.3 A Week Later

Create a function which takes in a date as a string, and returns the date a week after.

Examples

```
WeekAfter("12/03/2020")
-> "19/03/2020"

WeekAfter("21/12/1989")
-> "28/12/1989"

WeekAfter("01/01/2000")
-> "08/01/2000"
```

Hints

Note that dates will be given in **day/month/year** format. Single digit numbers should be zero padded.

3.4 Track the Robot

A robot has been given a list of movement instructions. Each instruction is either *left*, *right*, *up* or *down*, followed by a distance to move. The robot starts at [0, 0]. You want to calculate where the robot will end up and return its final position as an array.

To illustrate, if the robot is given the following instructions:

```
new string[] { "right 10", "up 50", "left 30", "down 10" }
```

It will end up 20 left and 40 up from where it started, so we return `int[] { -20, 40 }`.

Examples

```
TrackRobot(new string[] { "right 10", "up 50", "left 30", "down 10" })
-> int[] { -20, 40 }

TrackRobot(new string[] { })
```

```
-> int[] { 0, 0 }
// If there are no instructions, the robot doesn't move.

TrackRobot(new string[] { "right 100", "right 100", "up 500", "up 10000" })
-> int[] { 200, 10500 }
```

Hints

- The only instructions given will be *left*, *right*, *up* or *down*.
- The distance after the instruction is always a positive whole number.

3.5 True Alphabetical Order

Create a function which takes every letter in every word, and puts it in alphabetical order. Note how the original word lengths must stay the same.

Examples

```
TrueAlphabetic("hello world")
-> "dehll loorw"

TrueAlphabetic("edabit is awesome")
-> "aabdee ei imosstw"

TrueAlphabetic("have a nice day")
-> "aaac d eehi nvy"
```

Hints

- All sentences will be in lowercase.
- No punctuation or numbers will be included in the Tests.

3.6 Longest Common Ending

Write a function that returns the longest common ending between two strings.

Examples

```
LongestCommonEnding("multiplication", "ration")
-> "ation"

LongestCommonEnding("potent", "tent")
-> "tent"

LongestCommonEnding("skyscraper", "carnivore")
-> ""
```

Hints

[label=-] Return an empty string if there exists no common ending.

3.7 Clear Brackets

Create a function `Brackets()` that takes a string and checks that the brackets in the math expression are correct. The function should return `true` or `false`.

Examples

```
Brackets("(a*(b-c).....)")
-> true

Brackets(")(a-b-45/7*(a-34))")
-> false

Brackets("sin(90...)+.....cos1)")
-> false
```

Hints

- The string may not contain brackets, then return `true`.
- The string may contain spaces.
- The string may be empty.

3.8 Count the Number of Duplicate Characters

Create a function that takes a string and returns the number of alphanumeric characters that occur more than once.

Examples

```
DuplicateCount("abcde")
-> 0

DuplicateCount("aabbcde")
-> 2

DuplicateCount("Indivisibilities")
-> 2

DuplicateCount("Aa")
-> 0
// Case sensitive
```

Hints

- Duplicate characters are case sensitive.
- The input string will contain only alphanumeric characters.

3.9 Uban Numbers

A number *n* is called *uban* if its name (in English) does not contain the letter "u". In particular, it cannot contain the terms "four", "hundred", and "thousand", so the uban number following 99 is 1,000,000.

Write a function to determine if the given integer is uban.

Examples

```
IsUban(456)
-> false

IsUban(25)
-> true

IsUban(1098)
-> false
```

Hints

- Convert the integer to its English word representation. Use a library or a function to achieve this.
- Check if the word representation contains the letter "u".
- If the letter "u" is present in the word representation, the number is not uban.
- If the letter "u" is not present, the number is uban.
- Consider edge cases like very large numbers or zero.

Functions and Function Overloading

Note: Students are encouraged to bring their own laptops for laboratory practice sessions.

4.1 Climbing Stairs

You are climbing a staircase. It takes *n* steps to reach the top. Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

Specifications:

1. **Function Signature:**

   ```
   int climbStairs(int n);
   ```

2. **Input:**

   - An integer *n* representing the total number of steps to reach the top.

3. **Output:**
    - An integer representing the number of distinct ways to reach the top.
4. **Constraints:**
    - $1 \leq n \leq 45$

Examples:

Example 1:

```
Input: n = 2
Output: 2
Explanation: There are two ways to climb to the top.
1. 1 step + 1 step
2. 2 steps
```

Example 2:

```
Input: n = 3
Output: 3
Explanation: There are three ways to climb to the top.
1. 1 step + 1 step + 1 step
2. 1 step + 2 steps
3. 2 steps + 1 step
```

Implement the function `climbStairs` and also provide driver code to solve the problem.

Hints:

- Consider the base cases: $n = 1$ and $n = 2$.
- Think about the recursive relationship. If you are on the nth step, you could have arrived from the $(n-1)$th step or the $(n-2)$th step.

Try

- Test the function with small values of n such as 1, 2, 3, 4, and larger values up to 45 to verify correctness and performance.

4.2 Prime Number Utility

Create a program that offers three prime-related functionalities using three different functions, invoked based on user input. Specifically:

1. **Input:** Prompt the user to select an option:
    - Check if a number is prime.
    - Find the nth prime number.
    - Display a series of prime numbers up to a given number.
2. **Function Design:**

- Implement a function `bool isPrime(int number)` that:
  - Returns `true` if the number is prime.
  - Returns `false` otherwise.
- Implement a function `int findNthPrime(int n)` that:
  - Uses the `isPrime` function to find and return the nth prime number.
- Implement a function `void printPrimeSeries(int limit)` that:
  - Uses the `isPrime` function to print all prime numbers up to the given limit.

3. **Output:**
   - Based on the user's selection, the program will:
     - Print "Prime" or "Not Prime" if the first option is selected.
     - Print the nth prime number if the second option is selected.
     - Print the series of prime numbers up to the given limit if the third option is selected.

Write the C# code for the Prime Number Utility program using the described functions.

Hints:

- A prime number is a number greater than 1 that has no divisors other than 1 and itself.
- Use a loop to check for divisors up to the square root of the number for efficiency.
- To find the nth prime, iterate through numbers, checking for primality, and count primes found until the nth prime is reached.
- To display a series of prime numbers, iterate through numbers up to the limit and print each prime number.

4.3 Zodiac Sign Calculator

`ZodiacSignCalculator` determines the zodiac sign based on a given birthdate using func- tions. The program should:

1. **Input**: Prompt the user to enter their birthdate (DD/MM/YYYY).
2. **Functionality**: Implement a function to determine the zodiac sign based on the entered birthdate.
3. **Output**: Display the calculated zodiac sign corresponding to the entered birthdate.

Hints:

- Use functions to encapsulate the logic for determining the zodiac sign.
- Validate the input format and range of dates to ensure accurate zodiac sign determination.

Try:

- Test the program with various birthdates to verify correct zodiac sign calculation.
- Ensure the program handles edge cases such as leap years and valid input ranges.

4.4 Days in a Month Calculator

`DaysInMonthCalculator` determines the number of days in a given month of a specific year using functions. The program should:

1. **Input**: Prompt the user to enter a month (1 for January, 2 for February, etc.) and a year.
2. **Functionality**: Implement a function to calculate and return the number of days in the specified month.
3. **Output**: Display the calculated number of days in the entered month and year.

Hints:

- Use functions to encapsulate the logic for handling leap years and month-specific day counts.
- Validate the input to ensure valid month and year ranges.

Try:

- Test the program with different months and years, including leap years, to ensure accurate day count calculation.
- Verify that the program correctly handles edge cases such as February in leap years.

4.5 Date Validator

`DateValidator` checks if a given date is valid using functions. The program should:

1. **Input**: Prompt the user to enter a date in the format (DD/MM/YYYY).
2. **Functionality**: Implement a function to validate if the entered date is valid, considering leap years and month-specific day counts.
3. **Output**: Display a message indicating whether the entered date is valid or invalid.

Hints:

- Use functions to encapsulate the logic for validating date entries.
- Ensure proper validation of day ranges based on month and leap year calculations.

Try:

- Test the program with various dates to verify accurate validation of both valid and invalid dates.
- Ensure the program handles edge cases such as February 29th in leap years.

4.6 Age Difference Calculator

Develop a program named `AgeDifferenceCalculator` that calculates the age difference between two individuals using functions. The program should:

1. **Input**: Prompt the user to enter the birthdates of two people in the format (DD/MM/YYYY).
2. **Functionality**: Implement a function to calculate and return the difference in years, months, and days between the two birthdates.
3. **Output**: Display the calculated age difference in a readable format (e.g., "X years Y months Z days").

Hints

:

- Use functions to encapsulate the logic for age calculation and handling date differences.
- Validate the input to ensure accurate date formatting and valid date ranges.

Try

- Test the program with different birthdate pairs to verify accurate age difference calculation.
- Ensure the program correctly handles edge cases such as identical birthdates and reversed input order.

4.7 Elapsed Time Calculator

`ElapsedTimeCalculator` calculates the elapsed time between two timestamps within the same day using functions. The program should:

1. **Input**: Prompt the user to enter two timestamps in hours, minutes, and seconds format.
2. **Functionality**: Implement a function to calculate and return the elapsed time in hours, minutes, and seconds.
3. **Output**: Display the calculated elapsed time in a readable format (e.g., "X hours Y minutes Z seconds").

Hints:

- Use functions to encapsulate the logic for time difference calculation and handling time rollover.
- Validate the input to ensure valid time entries (0 ¡= hours ¡ 24, 0 ¡= minutes ¡ 60, 0 ¡= seconds ¡ 60).

Try:

- Test the program with different timestamp pairs to verify accurate calculation of elapsed time.
- Ensure the program handles edge cases such as timestamps crossing over midnight or identical timestamps.

4.8 Count of Specific Weekdays

`CountSpecificWeekdays` counts the number of specific weekdays (e.g., Mondays) between two given dates using functions. The program should:

1. **Input**: Prompt the user to enter two dates in the format (DD/MM/YYYY) and specify the weekday to count.
2. **Functionality**: Implement a function to iterate through each day between the two dates and count occurrences of the specified weekday.
3. **Output**: Display the total count of the specified weekday within the date range.

Hints:

- Use functions to encapsulate the logic for date iteration and weekday comparison.
- Validate the input to ensure accurate date formatting and valid date ranges.

Try:

- Test the program with various date ranges and weekdays to verify accurate counting.
- Ensure the program handles edge cases such as spanning across different months or years.

4.9 Polymorphic Printing

Function overloading is a feature that allows multiple functions with the same name but different parameter types or numbers, enabling the same function name to perform different tasks based on how it is called.

Create a program that demonstrates function overloading in C#. Specifically, define multiple versions of functions that handle different types of calculations or tasks but share the same function name.

1. **Function Design:**
   - Define a function `print()` that prints different types of data (e.g., integer, floating point number, string) to the console.
   - Overload the `print()` function to handle different data types and formats.
   - Demonstrate the use of function overloading with at least two different versions of the `print()` function, each accepting different parameters but sharing the same name.
2. **Output:** Output the results of calling the overloaded `print()` functions with different data types to demonstrate their functionality.

Hints:

- Ensure that each overloaded function has a distinct parameter list to differentiate them during function calls.
- Implement error handling or default behavior for unexpected or unsupported data types.

Try:

- Test the program with various data types (e.g., integer, float, string) to validate the functionality of

each overloaded `print()` function.

- Verify the handling of edge cases such as empty strings or invalid input values.

Motley Coding Tasks

Note: Students are encouraged to bring their own laptops for laboratory practice sessions.

5.1 Is the Phone Number Formatted Correctly?

Create a function that accepts a string and returns `true` if it's in the format of a proper phone number and `false` if it's not. Assume any number between 0-9 (in the appropriate spots) will produce a valid phone number.

Input

- A string representing the phone number.

Requirements

- The phone number must be in the format: (123) 456-7890
- Each digit can be any number between 0-9.
- There should be a space after the closing parenthesis.

Output

- Return `true` if the phone number is correctly formatted.
- Return `false` if the phone number is not correctly formatted.

Examples

1. `IsValidPhoneNumber("(123) 456-7890")` → `true`
2. `IsValidPhoneNumber("1111)555 2345")` → `false`
3. `IsValidPhoneNumber("098) 123 4567")` → `false`

Hints

- Consider using regular expressions to match the phone number format.
- Ensure that the string has exactly 14 characters.
- Check for the specific positions of characters: parentheses, space, and hyphen.

5.2 Basic E-Mail Validation

Create a function that accepts a string, checks if it's a valid email address, and returns either `true` or `false`, depending on the evaluation.

Input

- A string representing the email address to be validated.

Requirements

- The string must contain an `@` character.
- The string must contain a `.` character.
- The `@` must have at least one character in front of it.
- The `.` and the `@` must be in the appropriate places:
  - The `.` must come after the `@`.
  - There should be characters between the `@` and `.`.

Output

- Return `true` if the email address is valid.
- Return `false` if the email address is not valid.

Examples

1. `ValidateEmail("@gmail.com")` → `false`
2. `ValidateEmail("hello.gmail@com")` → `false`
3. `ValidateEmail("gmail")` → `false`
4. `ValidateEmail("hello@gmail")` → `false`
5. `ValidateEmail("hello@edabit.com")` → `true`

Hints

- Check the Tests tab to see exactly what's being evaluated.
- You can solve this challenge with RegEx, but it's intended to be solved with logic. Solutions using RegEx will be accepted but frowned upon.

5.3 Valid C# Comments

In C#, consider two types of comments:

- Single-line comments start with `//`
- Multi-line or block comments start with `/*` and end with `*/`

The input will be a sequence of `//`, `/*`, and `*/`. Every `/*` must have a `*/` that immediately follows it. Additionally, there can be no single-line comments in between multi-line comments.

Create a function that returns `true` if the comments are properly formatted, and `false` otherwise.

Input

- A string representing the sequence of comments.

Requirements

- Every `/*` must be followed by a `*/`.
- There must be no single-line comments (`//`) within multi-line comments (`/*` and `*/`).

Output

- Return `true` if the comments are properly formatted.
- Return `false` if the comments are not properly formatted.

Examples

1. `CommentsCorrect("//////") → true`
   - 3 single-line comments: `"//", "//", "//"`
2. `CommentsCorrect("/**//**////**/") → true`
   - 3 multi-line comments + 1 single-line comment: `"/*", "*/", "/*", "*/", "//", "/*", "*/"`
3. `CommentsCorrect("///*/**/") → false`
   - The first `/*` is missing a `*/`
4. `CommentsCorrect("/////") → false`
   - The 5th `/` is single, not a double `//`

5.4 IPv4 Validation

Problem Description

Create a function that takes a string (an IPv4 address in standard dot-decimal format) and returns `true` if the IP is valid, or `false` if it's not.

Input

- A string representing the IPv4 address.

Requirements

- The IPv4 address must be in the standard dot-decimal format: `X.X.X.X`, where `X` is a number.
- Each `X` must be a number between 0 and 255.
- Each `X` must not have leading zeros (e.g., `045` is invalid).
- The last digit may not be zero (e.g., `1.2.3.0` is invalid).
- IPv6 addresses are not valid.
- The address must contain exactly four octets.

Output

- Return `true` if the IPv4 address is valid.
- Return `false` if the IPv4 address is not valid.

Examples

1. `IsValidIP("1.2.3.4")` → `true`
2. `IsValidIP("1.2.3")` → `false`
3. `IsValidIP("1.2.3.4.5")` → `false`
4. `IsValidIP("123.45.67.89")` → `true`
5. `IsValidIP("123.456.78.90")` → `false`
6. `IsValidIP("123.045.067.089")` → `false`

Hints

- IPv6 addresses are not valid.
- Leading zeros are not valid (e.g., `123.045.067.089` should return false).
- Numbers must be between 1 and 255.
- The last digit must not be zero, but any other octet may have a zero.

5.5 Password Validation

Create a function that validates a password based on the following rules:

- Length must be between 6 and 24 characters.
- Must contain at least one uppercase letter (A-Z).
- Must contain at least one lowercase letter (a-z).
- Must contain at least one digit (0-9).
- Maximum of 2 repeated characters.

- Supported special characters include:
  `! @ # $ % & * ( ) + = _ - { } [ ] : ; " ' ? < > , .`

Create a function that returns `true` if the password is valid and `false` otherwise.

Input

- A string representing the password to be validated.

Requirements

- The password must be between 6 and 24 characters long.
- It must contain at least one uppercase letter.
- It must contain at least one lowercase letter.
- It must contain at least one digit.
- The password can have a maximum of 2 repeated characters.
- Special characters are supported but not required.

Output

- Return `true` if the password meets all the requirements.
- Return `false` if the password does not meet the requirements.

Examples

1. `ValidatePassword("P1zz@") → false`
   - Too short.
2. `ValidatePassword("iLoveYou") → false`
   - Missing a number.
3. `ValidatePassword("Fhg93@") → true`
   - OK!

Hints

- Ensure the password meets all the length, character, and repetition requirements.
- Special characters are optional but valid.

5.6 Valid Hex Code

Create a function that determines whether a string is a valid hex code.

A hex code must begin with a pound key # and is exactly 6 characters in length. Each character must be a digit from 0-9 or an alphabetic character from A-F. All alphabetic characters may be uppercase or lowercase.

Examples

```
IsValidHexCode("#CD5C5C")
-> true

IsValidHexCode("#EAECEE")
-> true

IsValidHexCode("#eaecee")
-> true

IsValidHexCode("#CD5C58C")
-> false
// Length exceeds 6

IsValidHexCode("#CD5C5Z")
-> false
// Not all alphabetic characters in A-F

IsValidHexCode("#CD5C&C")
-> false
// Contains unacceptable character

IsValidHexCode("CD5C5C")
-> false
// Missing #
```

Hints

- Check if the string starts with a pound key (#).
- Verify that the length of the string is exactly 7 characters (including the pound key).
- Ensure that all characters following the pound key are either digits (0-9) or letters in the range A-F (case insensitive).
- Use string methods or regular expressions to validate the characters.
- Consider converting the string to uppercase for a uniform comparison.

5.7 Retrieve Host Name

Create a function that retrieves the host name of the local machine. The function should return the host name of the machine on which the code is running.

Input

- No input is required for this function.

Requirements

- The function should retrieve the host name of the local machine.
- The function should return the host name as a string.

Output

- Return a string containing the host name of the local machine.

Examples

1. `GetLocalHostName()` → `"MyComputerName"`

Hints

- The host name is typically the name assigned to the computer by the operating system.
- Use the `Dns.GetHostName()` method to retrieve the host name.

5.8 Retrieve Host Name and IP Address

Create a function that retrieves the host name and IP addresses of the local machine. The function should display the host name and all associated IP addresses.

Input

- No input is required for this function.

Requirements

- The function should retrieve the host name of the local machine.
- The function should retrieve all associated IP addresses of the local machine.
- The function should display the host name and each IP address on a new line.

Output

- Display the host name and all associated IP addresses of the local machine.

Examples

```
Host Name of machine = MyComputerName
IP Address of Machine is
192.168.1.2
2001:0db8:85a3:0000:0000:8a2e:0370:7334
```

Hints

- The host name is typically the name assigned to the computer by the operating system.
- The IP addresses can be both IPv4 and IPv6.
- Use the `Dns.GetHostName()` method to retrieve the host name.
- Use the `Dns.GetHostAddresses(string)` method to retrieve the IP addresses.
- Import necessary packages: `System`, `System.Net`.

5.9 Retrieve IP Address of a Domain Name

Create a function that retrieves the IP addresses of a given domain name. The function should display the IP addresses associated with the provided domain name.

Input

- A string `domainName` representing the domain name to be resolved.

Requirements

- The function should retrieve all IP addresses associated with the given domain name.
- The function should display each IP address on a new line.
- Handle any potential exceptions gracefully and provide an appropriate error message.

Output

- Display the IP addresses associated with the provided domain name.

Examples

```
GetIPAddresses("www.bing.com");

Output:
IPAddress of www.bing.com is
13.107.21.200
204.79.197.200
```

Hints

- The domain name should be a valid domain name string.
- The IP addresses can be both IPv4 and IPv6.
- Use the `Dns.GetHostAddresses(string)` method to retrieve the IP addresses.
- Import necessary packages: `System`, `System.Net`.
- Handle exceptions using a try-catch block to ensure graceful error handling.

5.10 Retrieve IPv4 and IPv6 Addresses - Check Address Family

Create a function that retrieves and displays both IPv4 and IPv6 addresses of the local machine separately by checking the Address Family. The function should first output the host name of the machine, followed by separate lists of IPv4 and IPv6 addresses.

Input

- No input is required for this function.

Requirements

- The function should retrieve the host name of the local machine using `Dns.GetHostName()`.
- The function should retrieve all associated IP addresses of the local machine using `Dns.GetHostAddress`
- The function should filter and display IPv4 addresses separately.
- The function should filter and display IPv6 addresses separately.
- Display the host name followed by the lists of IPv4 and IPv6 addresses.

Output

- Display the host name of the local machine.
- Display the IPv4 addresses of the local machine, each on a new line.
- Display the IPv6 addresses of the local machine, each on a new line.

Examples

```
Host Name of machine = MyComputerName
IPv4 of Machine is
192.168.1.2
IPv6 of Machine is
2001:0db8:85a3:0000:0000:8a2e:0370:7334
```

Hints

- Ensure that you filter IP addresses by `AddressFamily` to distinguish between IPv4 and IPv6 addresses.
- The function should handle both types of addresses and display them clearly.
- Import the necessary packages: `System`, `System.Net`.
- Use the `Dns.GetHostName()` method to get the host name.
- Use the `Dns.GetHostAddresses(string)` method to get all IP addresses associated with the host name.
- Use LINQ to filter IP addresses based on `AddressFamily` to separate IPv4 and IPv6 addresses.

5.11 Retrieve Host Name Based on IP Address

Create a function that takes an IP address as input and returns the host name associated with that IP address. The function should use the `Dns.GetHostEntry()` method to obtain the host information and then extract and display the host name.

Input

- A string representing an IP address (e.g., "`204.79.197.200`").

Requirements

- The function should use the `Dns.GetHostEntry(string)` method to retrieve the `IPHostEntry` for the given IP address.
- Extract the host name from the `IPHostEntry` object.
- Display the host name associated with the given IP address.

Output

- Display the host name of the IP address.

Examples

```
Input: "204.79.197.200"
Output: "example.microsoft.com"
```

Hints

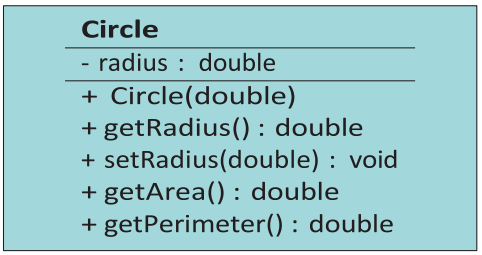
- Ensure that the IP address provided is valid and reachable.
- The host name may not always be resolvable depending on network configurations and the validity of the IP address.
- Import the necessary packages: `System`, `System.Net`.
- Use the `Dns.GetHostEntry(string)` method to retrieve the `IPHostEntry` for the given IP address.
- Access the `HostName` property of the `IPHostEntry` object to get the host name.

Classes & Objects

Note: Students are encouraged to bring their own laptops for laboratory practice sessions.

6.1 Make a Circle with OOP

Your task is to create a `Circle` constructor that creates a circle with a radius provided by an argument. The circles constructed must have two getters `GetArea()` ($\pi \cdot r^2$) and `GetPerimeter()` ($2 \cdot \pi \cdot r$) which give both respective areas and perimeter (circumference).



Examples

```
Circle circy = new Circle(11);
circy.getArea();

// Should return 380.132711084365.

Circle circy = new
Circle(4.44);
circy.getPerimeter();

// Should return 27.897342763877365
```

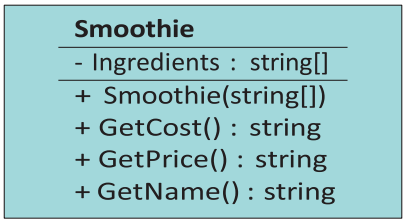
Hints

- Don't worry about floating point precision - the solution should be accurate enough.
- This is more of a tutorial than a challenge, so the topic covered may be considered ad- vanced.

6.2 Fruit Smoothie

Create a class `Smoothie` and do the following:

- Create a property called `Ingredients`.
- Create a `GetCost` method which calculates the total cost of the ingredients used to make the smoothie.
- Create a `GetPrice` method which returns the cost of the smoothie plus 50% markup. Round to two decimal places.
- Create a `GetName` method which gets the ingredients and puts them in alphabetical order into a nice descriptive sentence:
  - If there are multiple ingredients, add the word "Fusion" to the end.
  - If there is only one ingredient, add "Smoothie".
  - Change "-berries" to "-berry" for individual berry names.



Ingredient Prices:

Ingredient	Price
Strawberries	1.50
Banana	0.50
Mango	2.50
Blueberries	1.00
Raspberries	1.00
Apple	1.75
Pineapple	3.50

Examples:

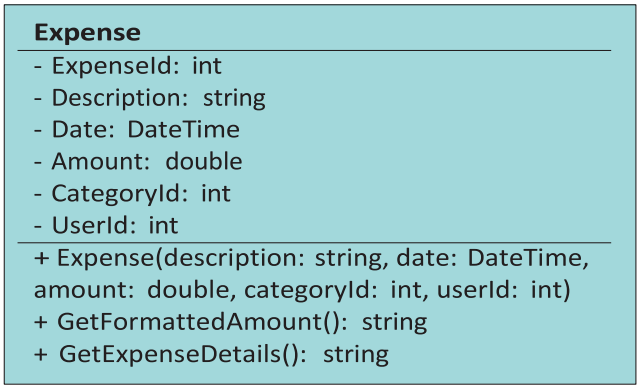
```
s1 = Smoothie(new string[] {
"Banana" }) s1.Ingredients ->
{ "Banana" } s1.GetCost() ->
"£0.50"
s1.GetPrice() ->
"£1.25" s1.GetName() ->
"Banana Smoothie"

s2 = Smoothie(new string[] { "Raspberries", "Strawberries",
"Blueberries" }) s2.ingredients -> { "Raspberries",
"Strawberries", "Blueberries" } s2.GetCost() -> "£3.50"
s2.GetPrice() -> "£8.75"
s2.GetName() -> "Blueberry Raspberry Strawberry Fusion"
```

Hints:

Feel free to check out the Resources tab for a refresher on classes.

6.3 Expense Tracker

Design and implement an `Expense` class to manage individual expenses within an Expense Tracker application. The class should provide functionalities to log, retrieve, and display expense details effectively.



Output:

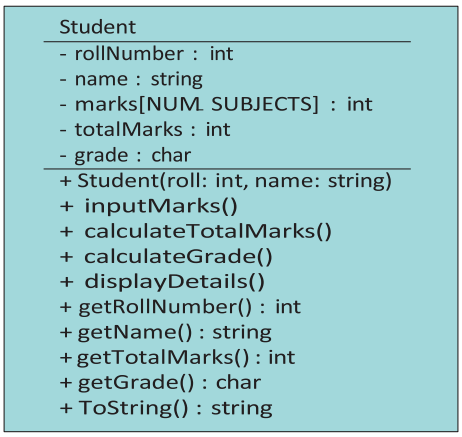
- `Constructor`: No direct output. Initializes the `Expense` object with the provided data.
- `GetFormattedAmount()`: Returns a formatted string representing the monetary amount of the expense.
- `GetExpenseDetails()`: Returns a formatted string with detailed information about the expense.

Hints:

- **Encapsulation**: Ensure attributes are properly encapsulated (private or protected access modifiers) to maintain data integrity.
- **Date Formatting**: Use appropriate formatting methods to display the date in a readable format.
- **String Formatting**: Utilize C#'s string interpolation or formatting methods (`ToString()`, format specifiers) for consistent and clear output.

6.4 Student Grades

The Student Grades class is designed to manage student details including roll number, name, and marks in multiple subjects. It calculates total marks and grades for each student and provides functionalities to input student details, process marks, calculate grades, and display student information.



Inputs

- Roll number and name for each student.
- Marks in multiple subjects (e.g., Math, Science, English) for each student.

Process Requirements

- Initialize student details and marks using the constructor.
- Input marks for each subject using the `inputMarks()` method.
- Calculate total marks using the `calculateTotalMarks()` method.
- Calculate the grade based on total marks using the `calculateGrade()` method.
- Display student details including roll number, name, marks in each subject, total marks, and grade using the `displayDetails()` method.

Output

- Display student details including roll number, name, marks in each subject, total marks, and grade.

Hints

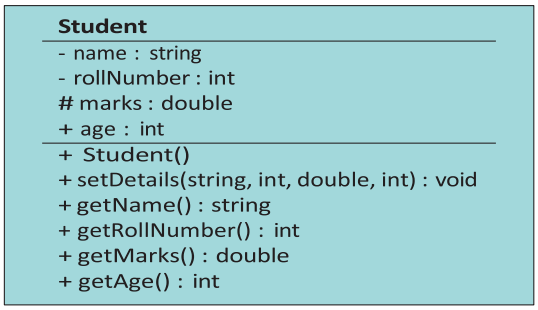
Grades can be assigned based on the total marks as follows:

- **A:** 90 - 100
- **B:** 80 - 89
- **C:** 70 - 79
- **D:** 60 - 69
- **F:** below 60

6.5 Control Your Code: Access Specifiers

Data hiding is a key feature of Object-Oriented Programming that restricts direct access to the internal details of a class. This prevents functions from directly manipulating the internal state of a class type. Access to class members is controlled using public, private, and protected labels within the class definition. These labels, known as access specifiers, dictate the level of access permitted for the class members.

Demonstrate the concepts of data hiding and access specifiers in a class `Student`. The `Student` class should have private, protected, and public data members. Implement member functions to set and get these details, ensuring that the internal representation of the class type is not directly accessible outside the class.



The `Student` class encapsulates the properties of a student while ensuring that the internal data (name, roll number, marks) is hidden from direct access outside the class. This is achieved using access specifiers: private, protected, and public.

Inputs

Details (`name`, `rollNumber`, `marks`, `age`) for a student.

Process Requirements

1. Define the `Student` class with:

   Private data members: `name`, `rollNumber`.

   Protected data member: `marks`.

   Public data member: `age`.

2. Implement public member functions `setDetails`, `getName`, `getRollNumber`, `getMarks`, and `getAge` to set and retrieve the values of the data members.

3. Ensure that the private data members cannot be accessed directly outside the class.

Output

- Print the details (`name`, `rollNumber`, `marks`, `age`) of the student using the public member functions.

Hints

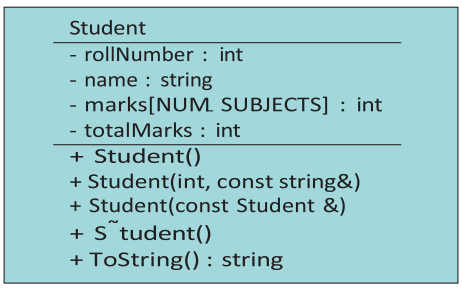
- Use private access specifier to hide sensitive data members.
- Use protected access specifier for data members that should be accessible in derived classes.
- Use public access specifier for data members that can be accessed directly.
- Example usage of the Student class with different access specifiers:

6.6 Craft Constructors

Constructors are special member functions responsible for initializing objects of a class. They are invoked automatically when an object is created. Constructors can initialize data members, allocate resources, and set default values based on parameters provided during object creation. They are defined with the same name as the class and can be overloaded to accept different parameter sets.

Destructors are also special member functions, that are invoked automatically when an ob- ject goes out of scope or is explicitly deleted. They perform cleanup tasks such as releasing resources (like memory or handles) acquired by the object during its lifetime. Destructors are named with a tilde ( ) followed by the class name and cannot have parameters or return types. They are essential for managing dynamic memory and ensuring proper resource deallocation.

Implement constructors for default initialization, parameterized initialization, copy opera- tion, and a destructor to manage resources for the following class



Input:

- **rollNumber**: Integer value representing the student's roll number.
- **name**: String representing the student's name.
- **marks[NUM SUBJECTS]**: Array of integers representing marks in various subjects.

Process Requirements:

- Implement a default constructor to initialize the 'Student' object with default values.
- Create a parameterized constructor to set initial values for 'rollNumber' and 'name'.
- Define a copy constructor to copy data from another 'Student' object.
- Implement a destructor to clean up resources when the object is destroyed.

Output:

- Display methods to show student details including 'rollNumber', 'name', 'marks', and 'totalMarks'.

Hints:

```
// Create Student objects using different constructors

// Display student details
```

```
// Clean up resources

// Destructor automatically called when objects go out of scope
```

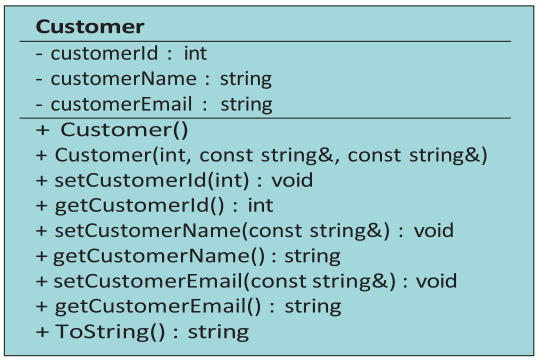
Try:

- Use initialization lists and proper memory management techniques.
- Ensure proper handling of dynamic memory if applicable.

6.7 Clearing Up Ambiguity: The Role of this Pointer

A special pointer, 'this' that refers to the current object instance within non-static member functions. It allows these functions to access and manipulate the object's data members and call other member functions of the same object.

Design a `Customer` class to manage customer information in a retail application. The class should include:



Input

- **Customer Information:** Inputs for customer details including `customerId`, `customerName`, and `customerEmail`.

Process

Implement a `Customer` class with:

- Private member variables: `customerId` (int), `customerName` (string), `customerEmail` (string).
- Constructors: Default constructor and parameterized constructor using the `this` pointer for initialization.
- Methods:
  - Setters and getters for each member variable, demonstrating **self-reference** us- ing the `this` pointer within methods.
  - Resolve **disambiguation** between instance variables and parameters/local vari- ables with the same name using the `this` pointer.
  - Enable **method chaining** by returning the `this` pointer from appropriate meth- ods.
  - Illustrate **passing object as argument** by implementing a method that accepts another `Customer` object.
  - Show how to **return the current object** (`this`) from methods to maintain method chaining or return the object state.

Output

- Display customer details including `customerId`, `customerName`, and `customerEmail` using the implemented methods.

Hints

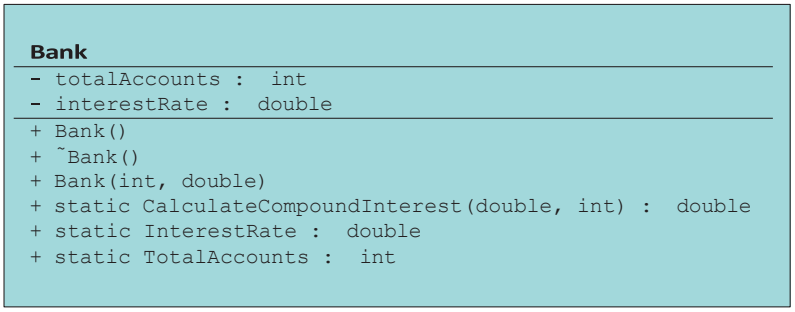
- Use the `this` pointer to reference the current object within its own methods and con- structors.
- Demonstrate how the `this` pointer resolves naming conflicts and ensures clarity in variable usage.
- Showcase the versatility of the `this` pointer in enabling efficient method chaining and object-oriented practices.

Try

- Create instances of the `Customer` class and demonstrate setting customer details using method chaining.
- Call methods sequentially on the same object to observe method chaining in action.
- Pass a `Customer` object as an argument to another method to illustrate object passing.
- Return the current object from a method and use it in a method chain to verify function- ality.

6.8 Consistent Behavior Across Objects

Demonstrate the usage of various `static` members, including static fields, properties, methods, and a static constructor. Explore their behaviors and interactions within the context of a simple application.



Requirements

1. **Static Fields**:
   - Implement a static field to maintain a count of instances created for the class.
   - Display the current count of instances whenever a new instance is created.
2. **Static Properties**:
   - Define a static property to store and retrieve a global configuration value.
   - Ensure the property is accessible and modifiable from different parts of the applica- tion.
3. **Static Methods**:

- Create a static method to perform a common utility task (e.g., calculating compound interest).
- Demonstrate calling this method without creating an instance of the class.

4. **Static Constructor**:
   - Implement a static constructor to initialize any static data or perform setup tasks.
   - Print a message confirming the static constructor's execution when the application starts.
5. **Application Interaction**:
   - Create instances of the class to verify the behavior of static fields.
   - Access and modify the static property to demonstrate its global accessibility.
   - Invoke the static method to perform a utility task.
   - Observe the sequence of static constructor execution.

Example Output

Upon running the application, the output should demonstrate:

- Creation of instances and display of instance count.
- Setting and retrieving global configuration values using the static property.
- Execution of the static method to perform a task.
- Confirmation message from the static constructor indicating initialization.

Hints

- Use the `static` keyword appropriately for fields, properties, methods, and constructors.
- Consider thread safety considerations when modifying shared static data.
- Utilize console output or logging to track execution flow and demonstrate expected behavior.

Try

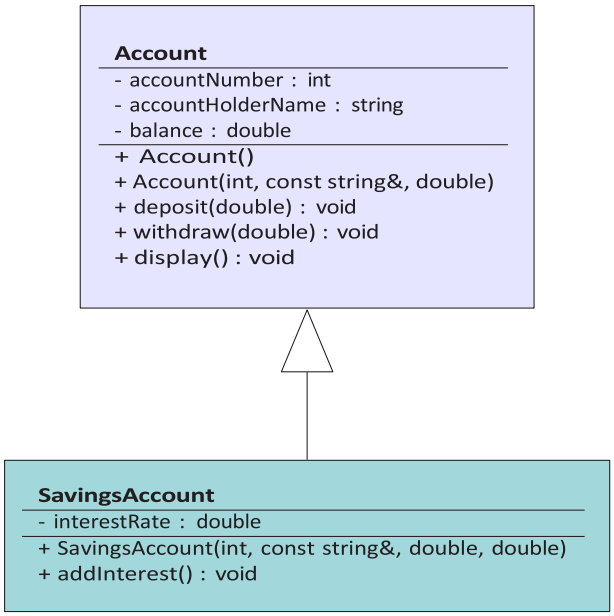
Implement the `Main` method to instantiate the `Bank` class, access static members, and verify the outputs as per the requirements.

Inheritance

Note: Students are encouraged to bring their own laptops for laboratory practice sessions.

7.1 Single Inheritance

You are tasked with designing a program to model a simple banking system using single inheritance. The program will involve creating a base class `Account` and a derived class `SavingsAccount`.



Input

- Initialize `Account` and `SavingsAccount` objects with appropriate data.
- Perform deposit and withdrawal operations on `SavingsAccount`.
- Calculate and add interest to `SavingsAccount`.

Process Requirements

1. **Account Class**:
    - Define the `Account` class with data members `accountNumber`, `accountHolderName`, and `balance`.

- Implement constructors for initializing the account.
- Methods include `deposit`, `withdraw`, and `display` to perform operations and display account details.

2. **SavingsAccount Class (Derived from Account)**:
   - Derive `SavingsAccount` from `Account`.
   - Add an additional data member `interestRate`.
   - Implement a constructor to initialize `SavingsAccount` with all parameters includ- ing `interestRate`.
   - Implement `addInterest` method to calculate and add interest to the balance based on the `interestRate`.

Output

- Display account details after each operation (deposit, withdrawal, adding interest).

Hints

- Use inheritance (`public`) to inherit `Account` properties into `SavingsAccount`.
- Implement methods to ensure proper encapsulation and validation of account operations.
- Test the program with multiple scenarios to validate the inheritance and functionality.

7.2 Multiple Inheritance

Develop a bank application using multiple inheritance to manage various types of accounts. Implement two base classes, `Account` and `Interest`, and derive a `SavingsAccount` class from both. This structure will allow efficient management of basic account functionalities and interest calculations.

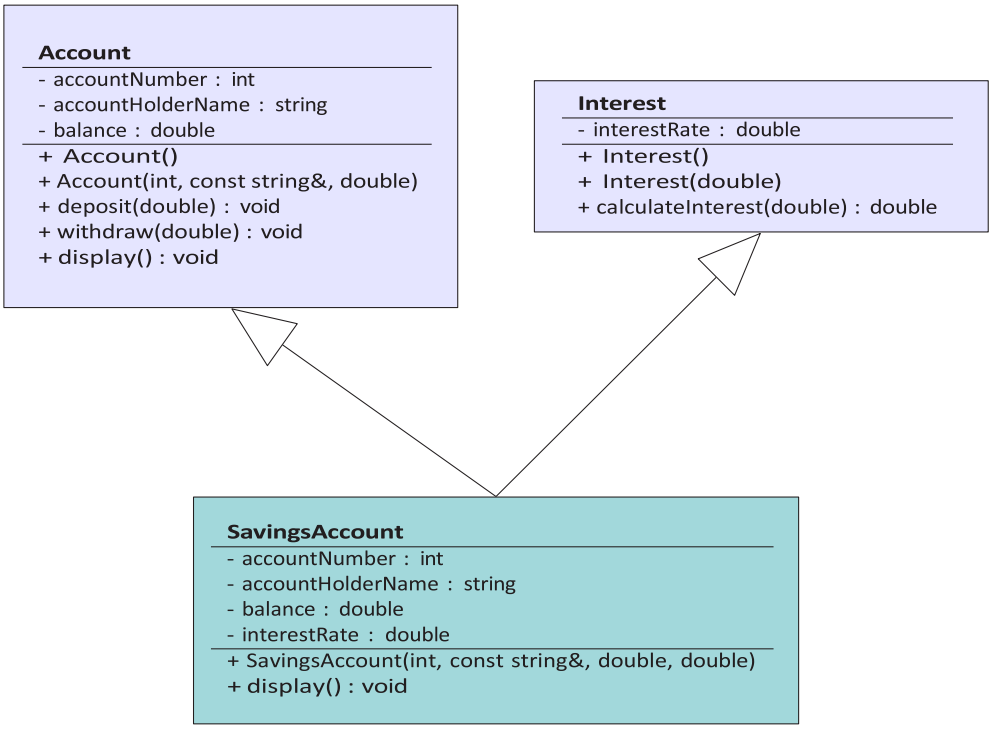
Input

- Account details: Account number, account holder name, initial balance.
- Interest rate for calculating interest.

Process Requirements

- Define the `Account` class with data members and member functions for basic account operations.
- Define the `Interest` class with data members and member functions for interest-related calculations.
- Implement the `SavingsAccount` class inheriting from both `Account` and `Interest`, integrating functionalities from both base classes.

- Implement constructors, including default and parameterized, and appropriate member functions in each class.



Output

Display account details and perform operations such as deposits, withdrawals, and interest calculations for savings accounts.

Hints

```
// Define base class Account
// Define base class Interest
// Define derived class SavingsAccount

// Usage example
SavingsAccount sa(1001, "Rama Krishna", 5000.0, 5.0);
sa.deposit(1000.0);
sa.display();
double interest = sa.calculateInterest(sa.balance);
cout << "Interest Earned: $" << interest << endl;
```

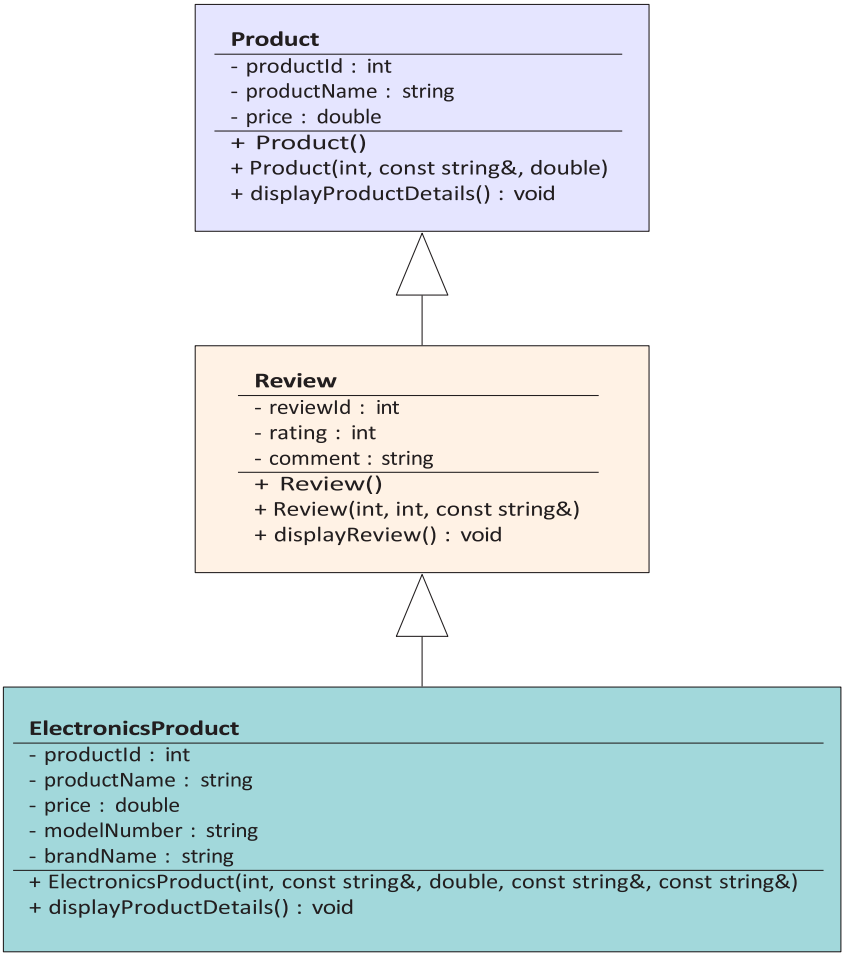
Try

Enhance the application by adding support for different types of accounts (e.g., checking accounts, fixed deposit accounts) with specific functionalities and interest calculations.

7.3 Multi Level Inheritance

Develop an online shopping application that utilizes multi-level inheritance to manage different types of products efficiently. Implement two base classes, `Product` and `Review`, and derive an `ElectronicsProduct` class from

both. This structure allows for comprehensive management of product information and customer reviews within a single class hierarchy.



Input

- Product details: product ID, product name, price.
- Review details: review ID, rating, comment.
- Additional details for electronics products: model number, brand name.

Process Requirements

- Define the `Product` class with data members and member functions for managing basic product information.
- Define the `Review` class with data members and member functions for handling customer reviews.
- Implement the `ElectronicsProduct` class inheriting from both `Product` and `Review`, integrating functionalities from both base classes.
- Implement constructors, including default and parameterized, and appropriate member functions in each class.

Output

Display product details, customer reviews, and specific information for electronics products.

Hints

```
// Define base class Product
// Define base class Review
// Define derived class ElectronicsProduct

// Usage example
ElectronicsProduct ep(1001, "Laptop", 1500.0, "Model X", "Brand Y");
ep.displayProductDetails();
ep.displayReview();
```

Try

Enhance the application by adding support for different types of products (e.g., clothing, books) with specific functionalities and review management.

7.4 Hierarchical Inheritance

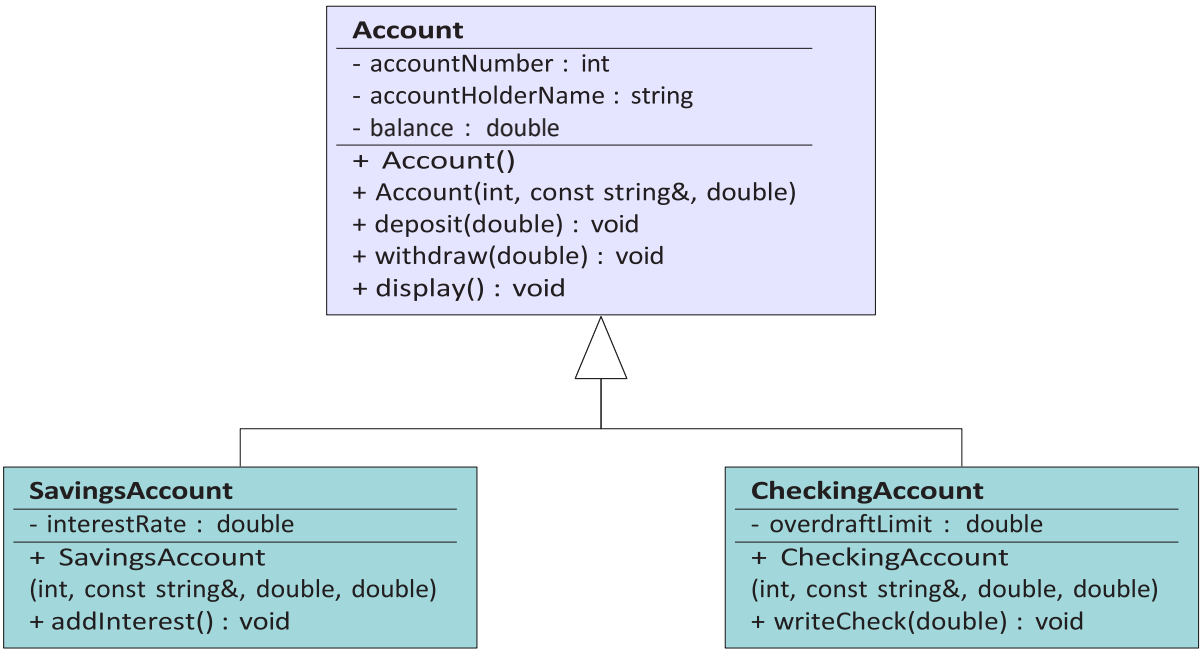
Develop a bank application using hierarchical inheritance to manage various types of accounts and related operations. Implement a base class, `Account`, and derive two classes, `SavingsAccount` and `CheckingAccount`, from it. This structure will allow efficient management of basic account functionalities and specific operations for different account types.

Input

- Account details: Account number, account holder name, initial balance.
- Interest rate for savings accounts.
- Overdraft limit for checking accounts.

Process Requirements

- Define the `Account` class with data members and member functions for basic account operations.
- Implement the `SavingsAccount` class inheriting from `Account`, with additional data members and member functions for interest-related operations.
- Implement the `CheckingAccount` class inheriting from `Account`, with additional data members and member functions for check-writing operations.
- Implement constructors, including default and parameterized, and appropriate member functions in each class.



Output

Display account details and perform operations such as deposits, withdrawals, interest calculations for savings accounts, and check writing for checking accounts.

Hints

```
// Define base class Account
// Define derived class SavingsAccount
// Define derived class CheckingAccount

// Usage example
SavingsAccount sa(1001, "Rama Krishna", 5000.0, 5.0);
sa.deposit(1000.0);
sa.display();
sa.addInterest();
sa.display();

CheckingAccount ca(2001, "Sita Devi", 3000.0, 1000.0);
ca.deposit(500.0);
ca.display();
ca.writeCheck(2000.0);
ca.display();
```

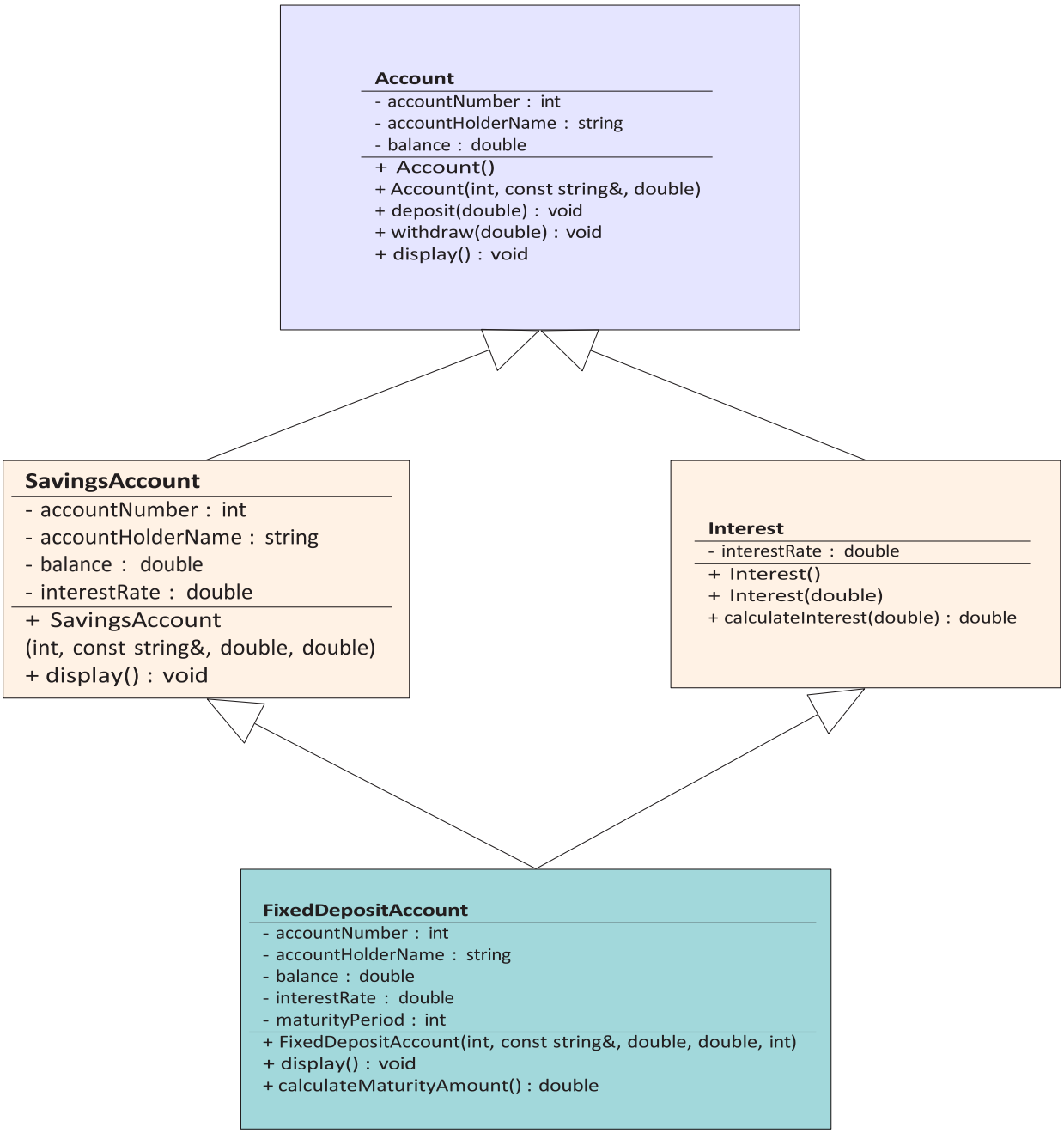
Try

Enhance the application by adding support for different types of accounts (e.g., fixed deposit accounts, recurring deposit accounts) with specific functionalities and interest calculations.

7.5 Hybrid Inheritance

Develop a bank application using hybrid inheritance to manage various types of accounts and related operations. Implement two base classes, `Account` and `Interest`, and derive two

classes, `SavingsAccount` and `FixedDepositAccount`, from these. This structure will allow efficient management of basic account functionalities and interest calculations.



Input

- Account details: Account number, account holder name, initial balance.
- Interest rate for calculating interest.
- Maturity period for fixed deposit accounts.

Process Requirements

- Define the `Account` class with data members and member functions for basic account operations.

- Define the `Interest` class with data members and member functions for interest-related calculations.
- Implement the `SavingsAccount` class inheriting from both `Account` and `Interest`, integrating functionalities from both base classes.
- Implement the `FixedDepositAccount` class inheriting from both `Account` and `Interest`, integrating functionalities from both base classes.
- Implement constructors, including default and parameterized, and appropriate member functions in each class.

Output

Display account details and perform operations such as deposits, withdrawals, and interest calculations for savings accounts and fixed deposit accounts.

Hints

```
// Define base class Account
// Define base class Interest
// Define derived class SavingsAccount
// Define derived class FixedDepositAccount

// Usage example
SavingsAccount sa(1001, "Rama Krishna", 5000.0, 5.0);
sa.deposit(1000.0);
sa.display();
double interest = sa.calculateInterest(sa.balance);
cout << "Interest Earned: $" << interest << endl;

FixedDepositAccount fda(2001, "Sita Devi", 10000.0, 6.5, 5);
fda.display();
double maturityAmount = fda.calculateMaturityAmount();
cout << "Maturity Amount: $" << maturityAmount << endl;
```

Try

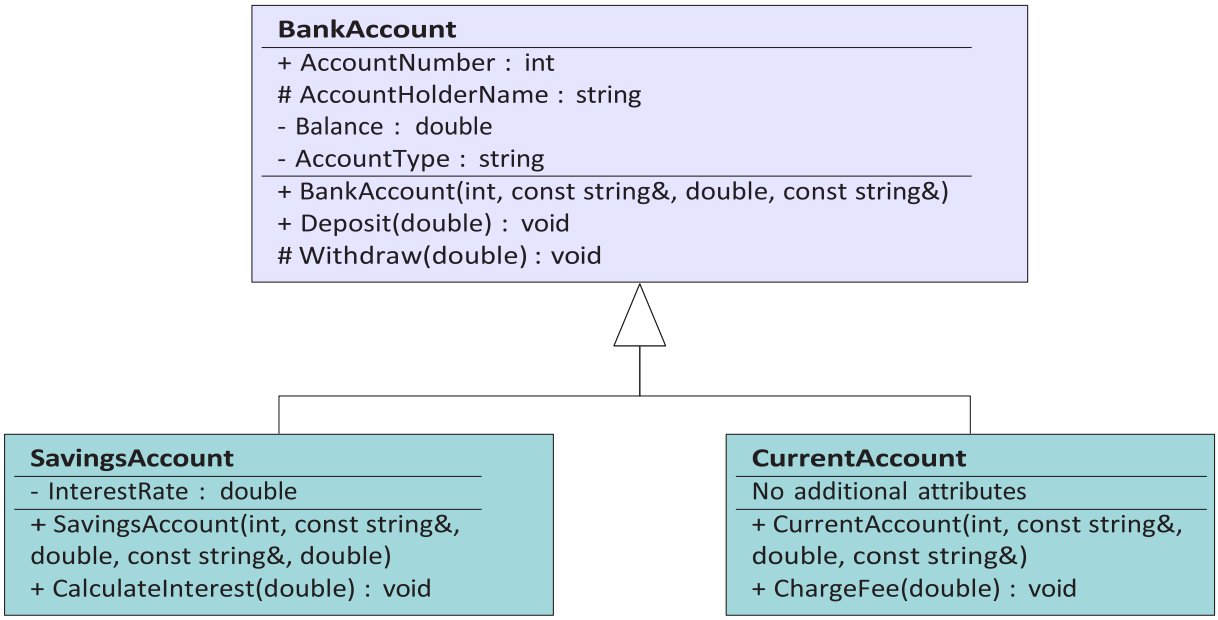
Enhance the application by adding support for different types of accounts (e.g., checking accounts, recurring deposit accounts) with specific functionalities and interest calculations.

7.6 Access Modifiers & Inheritance

Design and implement a bank application in C# to explore how different access modifiers (`public`, `protected`, `private`, etc.) affect inheritance. This exercise will enhance your understanding of object-oriented principles related to inheritance and access control in C#.

Input

- Bank account details such as account number, account holder's name, initial balance, and account type.
- Interest rate for savings accounts.
- Fees for current accounts.



Process

- Demonstrate the usage of different access modifiers (public, protected, private) in the base and derived classes.
- Show how protected members in the base class (Balance) are accessible to derived classes (SavingsAccount and CurrentAccount) but not accessible outside of them.
- Implement functionality to deposit, withdraw, calculate interest, and charge fees based on the account type.

Output

- Display the account details including account number, account holder's name, account type, and balance after performing deposit, withdrawal, interest calculation, and fee de- duction operations.

Hints

- Use access modifiers appropriately to control the accessibility of members in base and derived classes.
- Use constructors in derived classes to initialize base class data members.
- Use protected members in the base class to facilitate behavior in derived classes while keeping them encapsulated from external access.

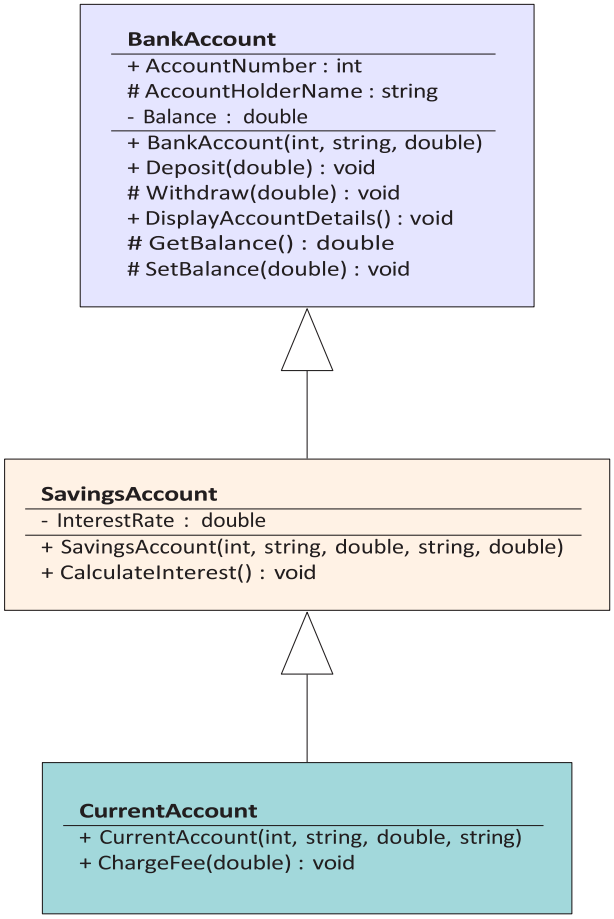
Try

- Extend the application by adding more account types (e.g., `FixedDepositAccount`) with specific functionalities and explore how access modifiers affect these new derived classes.

- Implement additional features such as account transfer and balance inquiry, ensuring proper access control using appropriate access modifiers.

7.7 Casting and Type Checking

Design and implement a bank application in C# to explore the use of the `is`, `as`, and `typeof` operators for checking and casting types in an inheritance hierarchy. This exercise will enhance your understanding of type checking and casting in object-oriented programming with C#.



Input

- Bank account details such as account number, account holder's name, initial balance, account type, and specific attributes for derived account types (e.g., interest rate for savings accounts).
- Operations to perform on different account types, such as deposit, withdraw, calculate interest, and charge fees.

Process

1. Create a base class `BankAccount` with common attributes and methods:
   - `AccountNumber` (public): a unique identifier for the account.
   - `AccountHolderName` (protected): the name of the account holder.

- `Balance` (private): the current balance of the account.
- Constructor to initialize `AccountNumber`, `AccountHolderName`, and `Balance`.
- Methods:
  - `Deposit(double amount)` (public): allows depositing money into the account.
  - `Withdraw(double amount)` (protected): allows withdrawing money from the account.
  - `DisplayAccountDetails()` (public): displays account details.

2. Create derived classes `SavingsAccount` and `CurrentAccount`:
   - `SavingsAccount` should include an `InterestRate` attribute and a method to `CalculateInterest()`.
   - `CurrentAccount` should include a method to `ChargeFee(double fee)`.
3. Implement a method to perform operations based on the type of the account using the `is`, `as`, and `typeof` operators:
   - Use the `is` operator to check if an object is of a particular type before performing operations.
   - Use the `as` operator to safely cast types and perform type-specific operations.
   - Use the `typeof` operator to display the type of the account.

Output

- Display the account details, including the account type, and results of operations performed on the account.

Hints

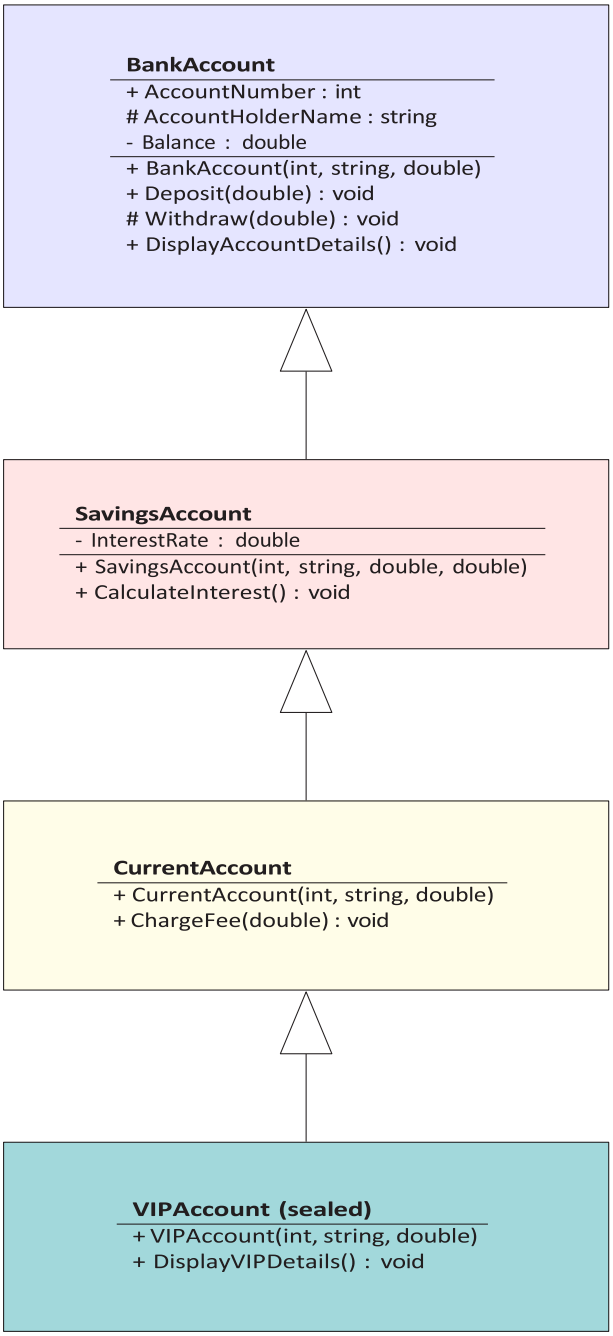
- Use the `is` operator to ensure that an object is of a specific type before performing type-specific operations.
- Use the `as` operator for safe casting, and check for `null` to avoid runtime exceptions.
- Use the `typeof` operator to get the type of an object at runtime.

Try

- Extend the application to include more account types (e.g., `FixedDepositAccount`) and implement additional type-specific operations.
- Implement a method to list all accounts and perform operations based on their types using type checking and casting.

7.8 Sealed Classes and Methods

Design and implement a bank application in C# to explore the use of sealed classes and methods to prevent further inheritance and method overriding. This exercise will help you understand how to restrict inheritance and control method overriding in object-oriented programming.



Input

- Bank account details such as account number, account holder's name, initial balance, account type, and specific attributes for derived account types (e.g., interest rate for savings accounts).
- Operations to perform on different account types, such as deposit, withdraw, calculate interest, and charge fees.

Process

1. Create a base class `BankAccount` with common attributes and methods:
    - `AccountNumber` (public): a unique identifier for the account.
    - `AccountHolderName` (protected): the name of the account holder.
    - `Balance` (private): the current balance of the account.

- Constructor to initialize `AccountNumber`, `AccountHolderName`, and `Balance`.
- Methods:
  - `Deposit(double amount)` (public): allows depositing money into the account.
  - `Withdraw(double amount)` (protected): allows withdrawing money from the account.
  - `DisplayAccountDetails()` (public): displays account details.

2. Create a derived class `SavingsAccount`:
   - `SavingsAccount` should include an `InterestRate` attribute and a method to `CalculateInterest()`.
3. Create a derived class `CurrentAccount`:
   - `CurrentAccount` should include a method to `ChargeFee(double fee)`.
4. Implement a `sealed` class `VIPAccount` that inherits from `BankAccount`:
   - `VIPAccount` should include additional privileges or features for VIP customers.
   - Ensure no other classes can inherit from `VIPAccount`.
5. Implement a `sealed` method in `BankAccount`:
   - Mark the `Withdraw` method as `sealed` to prevent overriding in derived classes.

Output

- Display the account details, including the account type, and results of operations performed on the account.

Hints

- Use the `sealed` keyword to prevent a class from being inherited.
- Use the `sealed` keyword in a method to prevent it from being overridden in derived classes.

Try

- Extend the application to include more account types (e.g., `FixedDepositAccount`) and implement additional type-specific operations without allowing inheritance from `VIPAccount`.
- Implement methods in the `VIPAccount` class that take advantage of its sealed nature.

Polymorphism, Abstract Classes and Interfaces

Note: Students are encouraged to bring their own laptops for laboratory practice sessions.

8.1 Complex Number Arithmetic

Demonstrate operator overloading by implementing addition for complex numbers. The program should define a `ComplexNumber` class, overload the `+` operator to perform addition of complex numbers, and display the result.

Program Requirements

1. Define a `ComplexNumber` class with properties for the real and imaginary parts.
2. Implement a constructor to initialize the real and imaginary parts of a complex number.
3. Overload the `+` operator to allow addition of two `ComplexNumber` objects.
4. Override the `ToString` method to provide a string representation of a complex number in the format "real + imaginaryi".
5. In the `Main` method, create instances of `ComplexNumber` and demonstrate the addition of two complex numbers using the overloaded `+` operator.
6. Print the result of the addition to the console.

Expected Output

The program should output the following:

```
Sum: 1.5 + 2.5i + 2.5 + 3.5i = 4 + 6i
```

8.2 Polynomial Arithmetic

Develop a C# program that demonstrates operator overloading by implementing multiplication for polynomials. The program should define a `Polynomial` class, overload the `*` operator to perform multiplication of polynomials, and display the result.

Requirements

1. Define a `Polynomial` class with a list to store the coefficients.
2. Implement a constructor to initialize the coefficients.
3. Overload the `*` operator to allow multiplication of two `Polynomial` objects.
4. Override the `ToString` method to provide a string representation of a polynomial in the standard mathematical format.
5. In the `Main` method, create instances of `Polynomial` and demonstrate the multiplication of two polynomials using the overloaded `*` operator.

6. Print the original polynomials and their product to the console.

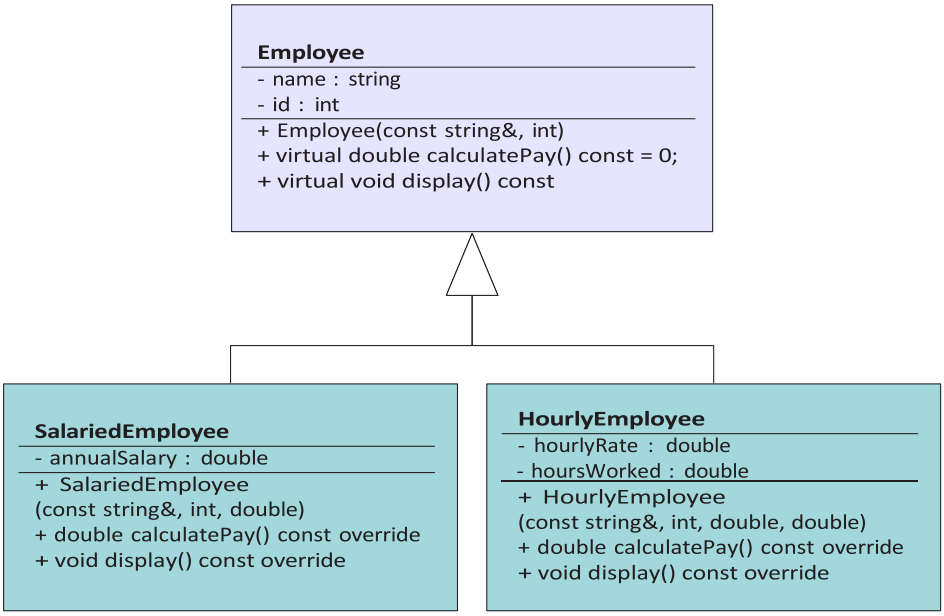
Expected Output

The program should output the following:

```
Polynomial 1:
3x^2 + 2x + 1
Polynomial 2:
5x^2 + 4x + 3
Product: 15x^4 + 22x^3 + 19x^2 + 10x + 3
```

8.3 Workforce Segments

The Workforce segment manages various types of employees and their specific payroll calculations. Implement a base class, `Employee`, and derive two classes, `SalariedEmployee` and `HourlyEmployee`, from it. This structure will allow efficient management of general employee information and specific payroll calculations for different types of employees.



Input

Employee details: Name, ID, and specific salary-related information.

Annual salary for salaried employees.

Hourly rate and hours worked for hourly employees.

Process Requirements

Define the `Employee` class with data members and member functions for general employee information.

Implement the `SalariedEmployee` class inheriting from `Employee`, with additional data members and member functions for salaried employees.

Implement the `HourlyEmployee` class inheriting from `Employee`, with additional data members and member functions for hourly employees.

Implement constructors, including default and parameterized, and appropriate member functions in each class.

Use polymorphism to calculate and display the pay for different types of employees.

Output

Display employee details and calculate their pay based on their type (salaried or hourly).

Hints

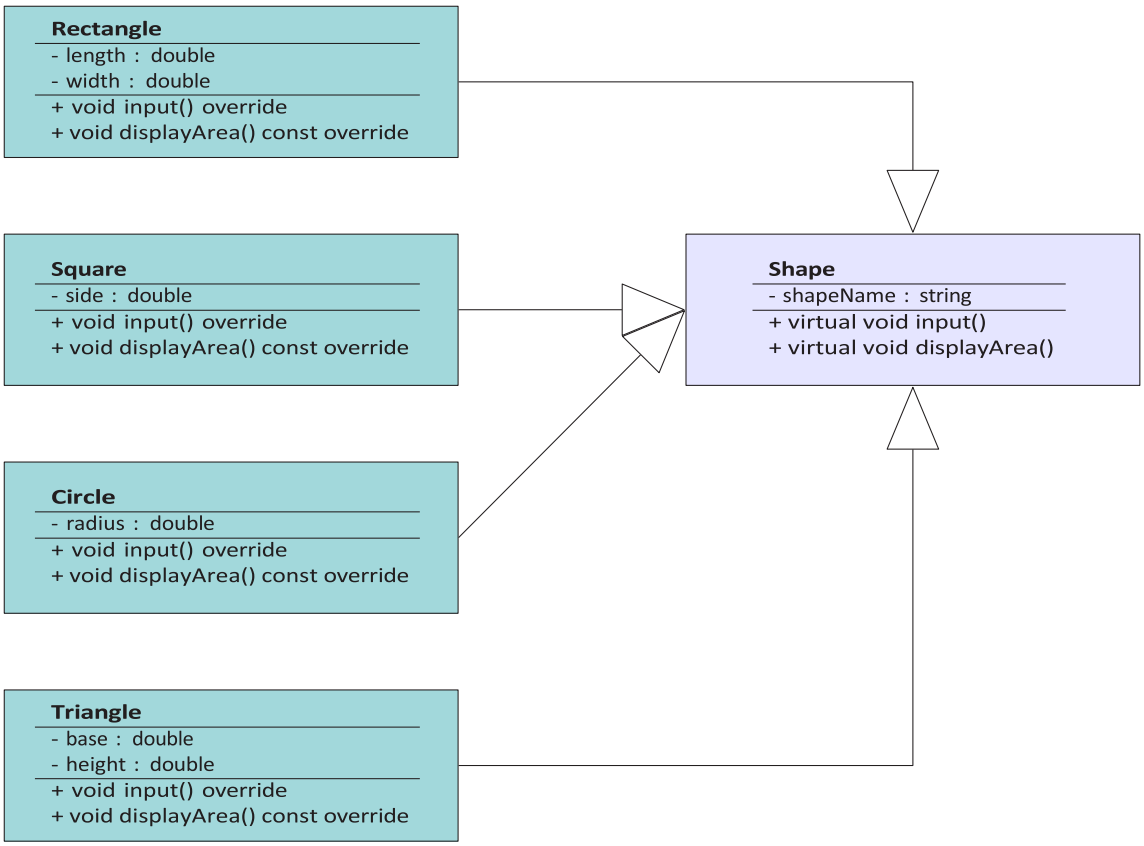
```
// Define base class Employee
// Define derived class SalariedEmployee
// Define derived class HourlyEmployee

// Usage example
SalariedEmployee se("Radha", 1001, 50000.0);
se.display();
cout << "Pay: " << se.calculatePay() << endl;

HourlyEmployee he("Smitha", 2002, 20.0, 40);
he.display();
cout << "Pay: " << he.calculatePay() << endl;
```

Try

Enhance the application by adding support for different types of employees (e.g., commission-based employees, contract employees) with specific payroll calculations.

8.4 Geometric Shapes: Abstract Class and Methods

Model different geometric shapes using object-oriented principles. Implement an abstract class `Shape` with the following specifications:



Shape Class:

- Define an abstract method `input()` to input dimensions specific to each shape.
- Define an abstract method `displayArea()` to calculate and display the area of the shape.
- The class should include a protected field `shapeName` to store the name of the shape.

Derived Classes:

Implement the following derived classes inheriting from `Shape`:

- `Rectangle`: Represents a rectangle with attributes for length and width.
- `Square`: Represents a square with a side length attribute.
- `Circle`: Represents a circle with a radius attribute.
- `Triangle`: Represents a triangle with attributes for base length and height.

Each derived class should override the `input()` method to accept user input for its specific dimensions and the `displayArea()` method to calculate and print its area.

Main Program:

In the `Main` method, demonstrate the usage of these shape classes:

- Create instances of each shape class.
- Prompt the user to input dimensions for each shape.
- Display the calculated area for each shape using the overridden `displayArea()` method.

Expected Output

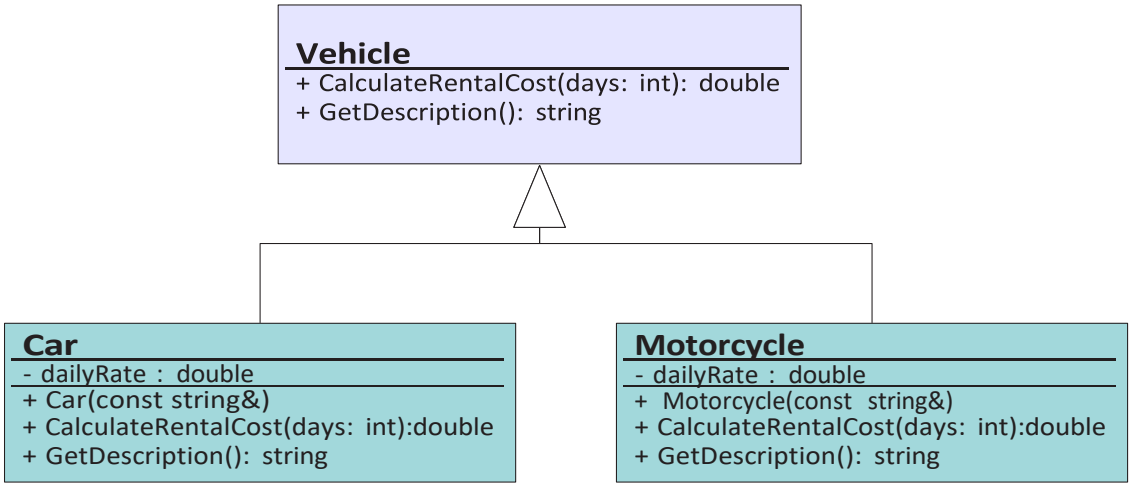
The program should output the calculated area for each shape based on user-provided dimensions.

Hints

- Use inheritance and method overriding to implement shape-specific behaviors.
- Ensure each derived class provides its own implementation of `input()` and `displayArea()`.
- Demonstrate polymorphism by treating all shapes uniformly through the base class reference.

8.5 Rental Vehicle Cost Calculator

You are developing a vehicle rental system where different types of vehicles can be rented for a specified number of days. Your task is to create a class hierarchy in C# to represent the vehicles and calculate the rental costs based on the type of vehicle and the number of rental days.



Implementation Requirements

1. Define the abstract class `Vehicle` with the required abstract methods.
2. Define the `Car` class that extends `Vehicle` and implements the required methods.
3. Define the `Motorcycle` class that extends `Vehicle` and implements the required methods.
4. Ensure that the `Car` and `Motorcycle` classes correctly calculate the rental cost based on the number of days and their respective daily rates.
5. The `GetDescription()` method should return a string describing the vehicle, including its type and daily rate.

Example Usage

```
Vehicle car = new Car("Sedan", 50.0);
// Output: "Sedan with a daily rate of $50.00"
Console.WriteLine(car.GetDescription());
// Output: 250.0
Console.WriteLine(car.CalculateRentalCost(5));

Vehicle motorcycle = new Motorcycle("Sport Bike", 30.0);
// Output: "Sport Bike with a daily rate of $30.00"
Console.WriteLine(motorcycle.GetDescription());
// Output: 90.0
Console.WriteLine(motorcycle.CalculateRentalCost(3));
```

Hints

- Define an abstract class `Vehicle` with abstract methods `CalculateRentalCost` and `GetDescription`.
- Implement the `Car` and `Motorcycle` classes to inherit from `Vehicle`.
- Use the constructor in `Car` and `Motorcycle` to initialize the `dailyRate`.
- Ensure the `CalculateRentalCost` method in each derived class multiplies the `dailyRate` by the number of days.
- The `GetDescription` method should return a formatted string describing the vehicle.

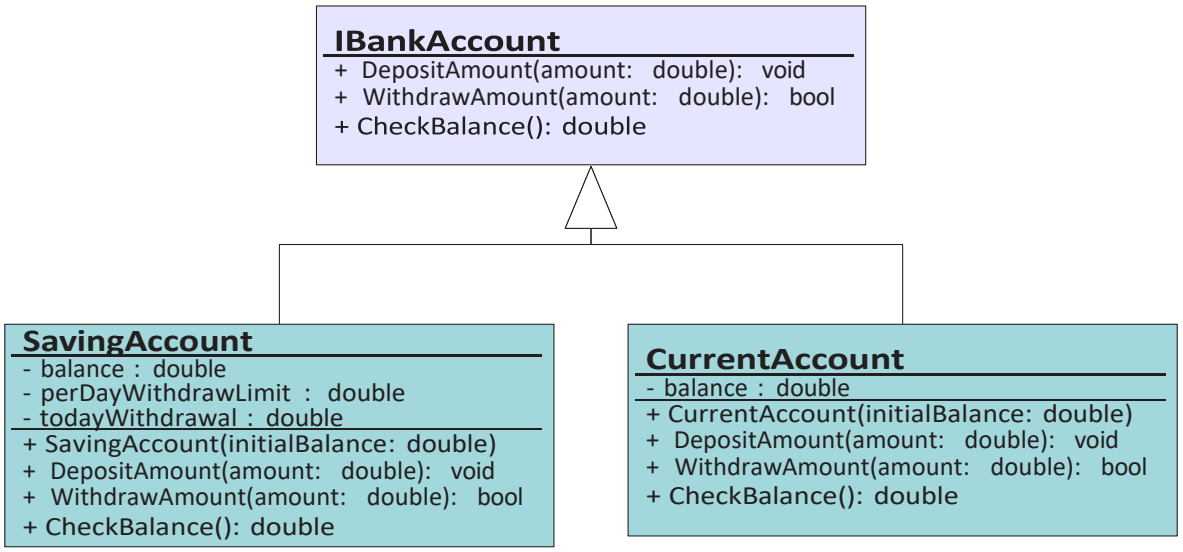
Note: Ensure that the implementation follows the principles of object-oriented programming, including encapsulation, inheritance, and polymorphism.

8.6 IBankAccount

Develop a bank application that supports the creation and management of the following account types:

Saving Account

Current Account



Requirements:

The application should provide the following services for each account:

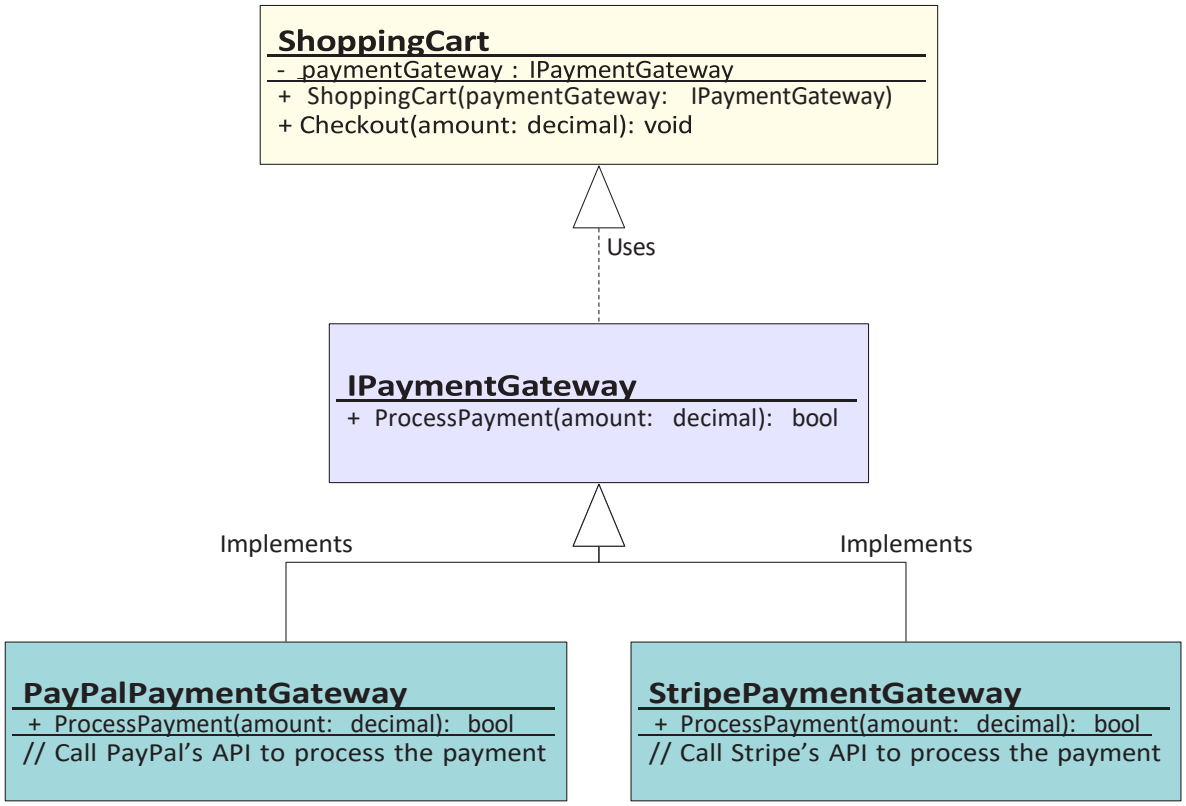
- **DepositAmount(amount: double): void** - Deposits the specified amount into the account.
- **WithdrawAmount(amount: double): bool** - Withdraws the specified amount from the account if sufficient funds are available.
- **CheckBalance(): double** - Returns the current balance of the account.

Hints

- Define the `IBankAccount` interface with `DepositAmount`, `WithdrawAmount`, and `CheckBalance` methods.
- Implement the `SavingAccount` class to handle balance, per-day withdrawal limit, and today's withdrawal.
- Implement the `CurrentAccount` class to handle balance without any withdrawal limit.
- Use the interface methods to perform deposit, withdrawal, and balance checking operations.
- Display the balance after each operation.
- Ensure that withdrawal operations check for sufficient funds before proceeding.

8.7 Payment Gateway

Develop a payment processing system using the `IPaymentGateway` interface. This interface should include a method `ProcessPayment` to handle payments. Implement this interface with two classes: `PayPalPaymentGateway` and `StripePaymentGateway`, each simulating pay- ment processing for their respective platforms.



Create a `ShoppingCart` class that uses an `IPaymentGateway` to process payments. The system should be able to handle payments through different providers by swapping implementations.

Implement the `IPaymentGateway` interface and the associated payment provider classes. Ensure the `ShoppingCart` class can interact with any payment provider without modification. Test the implementation with both PayPal and Stripe providers.

Hints:

- The `ShoppingCart` class should not be concerned with the specifics of payment process- ing.
- Adding new payment providers should require minimal changes to the existing code.

Exception Handling

Note: Students are encouraged to bring their own laptops for laboratory practice sessions.

9.1 DivideByZeroException

Performs division of two numbers provided by the user. The program should handle exceptions gracefully, particularly when the user inputs non-numeric values.

Input:

Two inputs from the user representing the numbers to be divided:

- The first number (numerator).
- The second number (denominator).

Operations:

1. **Input Handling:**
   - Prompt the user to enter the first number (numerator).
   - Prompt the user to enter the second number (denominator).
2. **Exception Handling:**
   - Use a try-catch block to handle exceptions.
   - Catch `FormatException` to handle non-numeric inputs.
   - Catch `DivideByZeroException` to handle division by zero.
   - Optionally, catch other general exceptions to handle unexpected errors.
3. **Division Operation:**

- Perform the division operation if both inputs are valid numbers and the denominator is not zero.

4. **Output:**

   - If inputs are valid, display the result of the division.
   - If a `FormatException` occurs, display an error message indicating the input was not a valid number.
   - If a `DivideByZeroException` occurs, display an error message indicating that division by zero is not allowed.
   - Optionally, handle other exceptions and display a generic error message.

Hints:

- Use `Convert.ToDouble` or `double.TryParse` to convert the user input to a double type.
- Encapsulate the division logic within a try-catch block to handle potential exceptions.
- Consider edge cases like zero as a denominator and non-numeric inputs.

Sample Output:

- **Case 1: Valid Input**

```
Enter the first
number: 10
Enter the second
number: 2
Result: 10 / 2 = 5
```

- **Case 2: Non-Numeric Input**

```
Enter the first
number: abc
Enter the
second number:
2
Error: Input is not a valid number.
```

- **Case 3: Division by Zero**

```
Enter the first
number: 10
Enter the
second number:
0
Error: Division by zero is not allowed.
```

9.2 Handling Negative Number Exception

Define a method to check if a given integer is negative. If the integer is negative, the method should throw an exception. The calling code should handle this exception and provide appropriate feedback to the user.

Input:

- An integer value provided by the user.

Operations:

1. **Method Definition:**
   - Define a method `CheckIfNegative` that takes an integer as a parameter.
   - If the integer is negative, throw an exception with an appropriate message.
2. **Exception Handling:**
   - In the calling code, prompt the user to enter an integer.
   - Call the `CheckIfNegative` method with the user's input.
   - Use a `try-catch` block to handle the exception thrown by the method.
   - If an exception is caught, display an error message to the user.

Output:

- If the input integer is non-negative, display a message indicating that the number is valid.
- If the input integer is negative, catch the exception and display the error message.

Hints:

- Use the `throw` statement to raise an exception within the `CheckIfNegative` method.
- Use a `try-catch` block in the calling code to handle the exception and display an ap- propriate message to the user.
- Test the program with both positive and negative integers to ensure proper exception handling.

Sample Output:

- **Case 1: Non-Negative Input**

```
Enter an
integer:
5 The
number 5
is
valid.
```

- **Case 2: Negative Input**

```
Enter an integer: -3
Error: The number -3 is negative.
```

9.3 Validating Integer Input Range

Prompt the user to input a numeric integer. The program should throw an exception if the

number is less than 0 or greater than 1000. The calling code should handle the exception and display appropriate error messages.

Input:

- A numeric integer provided by the user.

Operations:

1. **Input Handling:**
   - Prompt the user to enter an integer.
   - Read the user input.
2. **Validation:**
   - Define a method `ValidateNumber` that takes an integer as a parameter.
   - If the integer is less than 0 or greater than 1000, throw an exception with an appro- priate message.
3. **Exception Handling:**
   - Use a `try-catch` block to handle the exception thrown by the `ValidateNumber` method.
   - If an exception is caught, display an error message to the user.

Output:

- If the input integer is between 0 and 1000 (inclusive), display a message indicating that the number is valid.
- If the input integer is less than 0 or greater than 1000, catch the exception and display the error message.

Hints:

- Use the `throw` statement to raise an exception within the `ValidateNumber` method.
- Use a `try-catch` block in the calling code to handle the exception and display an ap- propriate message to the user.
- Test the program with edge cases, such as -1, 0, 1000, and 1001, to ensure proper exception handling.

Sample Output:

- **Case 1: Valid Input**

```
Enter an
integer:
500 The
number 500
is valid.
```

- **Case 2: Less than 0**

```
Enter an integer: -5
Error: The number -5 is out of the allowed range (0-1000).
```

- **Case 3: Greater than 1000**

```
Enter an integer: 1500
Error: The number 1500 is out of the allowed range (0-1000).
```

9.4 Array Type Mismatch Exception

You are required to write a program that demonstrates the occurrence of an **ArrayTypeMismatchException**. This exception is thrown when an array cannot store a given element because the actual type of the element is incompatible with the declared type of the array.

Input:

- Declare an array of integers.
- Attempt to assign a string value to an element in the integer array.

Process:

- Use a try-catch block to handle the exception.
- Catch the `ArrayTypeMismatchException` and display an error message indicating the attempted operation.

Output:

- Upon encountering the exception, output a descriptive error message explaining the nature of the exception.

Hints:

- Initialize an array of integers.
- Try assigning a string value or another incompatible data type to an element in the integer array.
- Encapsulate the assignment operation within a try block and handle the exception in the catch block.

9.5 Format Exception

Develop a program that illustrates the occurrence of a **FormatException**. This exception is thrown when an invalid format is detected in the input provided by the user or within the program.

Input:

- Prompt the user to enter a number as a string.
- Attempt to convert this string input to an integer using `int.Parse()` or `Convert.ToInt32()`.

Process:

- Use a try-catch block to handle the `FormatException`.
- Catch the exception and display an appropriate error message explaining the format error.

Output:

- Upon encountering the exception, output a clear error message indicating that the input string was not in a correct format.

Hints:

- Encourage users to provide valid numeric inputs.
- Use error handling techniques to gracefully manage incorrect format inputs.

9.6 Index Out of Range Exception

Design a program that demonstrates the occurrence of an **IndexOutOfRangeException**. This exception is thrown when an attempt is made to access an element of an array or a collection with an index that is outside the bounds of the array or collection.

Input:

- Declare an array with a specific size.
- Attempt to access an element using an index that is out of the array's bounds (e.g., accessing at an index greater than or equal to the array length).

Process:

- Use a try-catch block to handle the `IndexOutOfRangeException`.
- Catch the exception and display an informative error message explaining the index out of range error.

Output:

- Upon encountering the exception, output a descriptive error message indicating the at- tempted operation that caused the index out of range.

Hints:

- Initialize an array with a specific size and populate it with elements.
- Attempt to access an element using an index that exceeds the array's size.
- Utilize error handling techniques to gracefully manage index-related errors.

9.7 Invalid Cast Exception

Develop a program that demonstrates the occurrence of an **InvalidCastException**. This exception is thrown when an explicit conversion is attempted between incompatible data types, such as from a base type to a derived type or from one type to another that cannot be converted.

Input:

- Declare an object of a base type.
- Attempt to cast this object to a derived type that it is not compatible with.

Process:

- Use a try-catch block to handle the `InvalidCastException`.
- Catch the exception and display an informative error message explaining the invalid cast operation.

Output:

- Upon encountering the exception, output a descriptive error message indicating the at- tempted invalid cast.

Hints:

- Define classes or objects where an invalid cast scenario can be simulated.
- Attempt to cast an object to a type that it is not assignable to.
- Use error handling techniques to gracefully manage type casting errors.

9.8 Null Reference Exception

Design a program that demonstrates the occurrence of a **NullReferenceException**. This exception is thrown when an attempt is made to access or call a member (method or property) on an object reference that is currently null.

Input:

- Declare an object reference variable without initializing it (set it to null).
- Attempt to call a method or access a property on this null object reference.

Process:

- Use a try-catch block to handle the `NullReferenceException`.
- Catch the exception and display an informative error message explaining the null reference access.

Output:

- Upon encountering the exception, output a descriptive error message indicating the at- tempted null reference access.

Hints:

- Initialize an object reference variable to null.
- Attempt to call a method or access a property on this null reference without assigning an instance to it.
- Use error handling techniques to gracefully manage null reference errors.

9.9 Overflow Exception

Develop a program that demonstrates the occurrence of an **OverflowException**. This exception is thrown when an arithmetic operation exceeds the range of the data type used to store the result, resulting in an overflow.

Input:

- Declare variables of appropriate data types (e.g., `int`, `double`) and assign values that may cause overflow during arithmetic operations.
- Perform arithmetic operations that are likely to exceed the maximum or minimum value limits of the data type.

Process:

- Use a try-catch block to handle the `OverflowException`.
- Catch the exception and display an informative error message explaining the arithmetic overflow.

Output:

- Upon encountering the exception, output a descriptive error message indicating the arith- metic operation that caused the overflow.

Hints:

- Initialize variables with values that are near the maximum or minimum limits of their data type.
- Perform operations like addition, multiplication, or division that could exceed the data type's range.
- Use error handling techniques to gracefully manage arithmetic overflow errors.

9.10 Catching Multiple Exceptions

Develop a program that demonstrates how to catch multiple specific exceptions and use a general catch block for any other exceptions that are not explicitly handled.

Input:

- Prompt the user to enter two numbers for a division operation.
- Optionally, allow the user to choose an invalid operation to intentionally trigger specific exceptions.

Operations:

1. **Input Handling:**
   - Prompt the user to enter the first number (numerator).
   - Prompt the user to enter the second number (denominator).
2. **Exception Handling:**
   - Use a try-catch block to handle exceptions.
   - Catch `DivideByZeroException` to handle division by zero.
   - Catch `FormatException` to handle non-numeric inputs.
   - Catch `OverflowException` to handle arithmetic overflow.
   - Use a general catch block to handle any other unexpected exceptions.
3. **Division Operation:**
   - Perform the division operation if both inputs are valid numbers and the denominator is not zero.

Output:

- If inputs are valid, display the result of the division.
- If a `DivideByZeroException` occurs, display an error message indicating that division by zero is not allowed.
- If a `FormatException` occurs, display an error message indicating the input was not a valid number.
- If an `OverflowException` occurs, display an error message indicating an arithmetic overflow.
- If any other exception occurs, display a generic error message.

Hints:

- Use `Convert.ToInt32` or `int.TryParse` to convert the user input to an integer type.
- Encapsulate the division logic within a try-catch block to handle potential exceptions.
- Consider edge cases like zero as a denominator, non-numeric inputs, and very large numbers that may cause overflow.

File Handling

Note: Students are encouraged to bring their own laptops for laboratory practice sessions.

10.1 File Creation and Text Writing

You are required to develop a program that creates a new file named mytest.txt and writes some text content into this file. The program should ensure that the file is created successfully and the specified text is written into it.

Input:

No direct input from the user is required for this problem.

Operations:

1. Create a new file named `mytest.txt` in the specified directory.
2. Write the text `"Hello, this is a test file."` into the file.
3. Ensure the file is saved properly with the written content.
4. Display a message indicating the successful creation of the file and the writing of the content.

Output:

The output should indicate the successful creation of the file and the addition of the text, as follows:

```
File mytest.txt created successfully with the following content:
Hello and Welcome to the C# Lab
In this lab we are working with Files. It's really interesting to
work with C#
```

Hints:

- Use the `System.IO` namespace to handle file operations.
- Use the `File.WriteAllText` method to create the file and write the specified content to it.
- Handle any potential exceptions using a `try-catch` block to ensure the program does not crash in case of an error.

10.2 Reading the content of the File

Demonstrate file reading operations. The program should read the content from the file and then display the content on the console. This will involve using file handling methods provided by the `System.IO` namespace to ensure proper reading of the file.

Input:

No direct input from the user is required for this problem.

Operations:

1. Search for the file named `mytest.txt` in the specified directory.
2. Read the content from the created file.
3. Display the content of the file on the console.

Output:

The output should display the content of the file, as follows:

```
Here is the content of the file mytest.txt:
Hello and Welcome to the C# Lab
In this lab we are working
with Files. It's really
interesting to work with C#
```

Hints:

- Use the `System.IO` namespace to handle file operations.
- Use the `File.WriteAllText` method to create the file and write the specified content to it.
- Use the `File.ReadAllText` method to read the content from the file.
- Handle any potential exceptions using a `try-catch` block to ensure the program does not crash in case of an error.

10.3 Writing an Array of Strings to a File

Create a text file and write an array of strings into it. The program should prompt the user to input the number of lines and the content for each line. It will then save this content to a file and display the file's content.

Input:

- The number of lines to write to the file.
- The content for each line.

Operations/Instructions:

1. Prompt the user to input the number of lines to write in the file.
2. Prompt the user to input the content for each line.
3. Create a new text file named `myfile.txt`.
4. Write the user-provided lines into the file.
5. Read the content from the file.
6. Display the content of the file on the console.

Output:

The output should display the content of the file, as follows:

```
 The content of the file is:
-----------------------------------------

 Explore new horizons every day.
 Embrace challenges and seize
 opportunities. Learn, grow, and
 inspire others.
 Dream big, work hard, achieve greatness.
```

Hints:

- Use the `System.IO` namespace to handle file operations.
- Use the `File.WriteAllLines` method to write the array of strings to the file.
- Use the `File.ReadAllLines` method to read the content from the file.
- Handle any potential exceptions using a `try-catch` block to ensure the program does not crash in case of an error.

10.4 Exclude lines that contain specific string

Create a text file and writes specified lines into it, excluding any line containing a specified string. The program should prompt the user to input the string to ignore, the number of lines to write, and the content for each line. It will then save this content to a file, excluding lines that contain the specified string, and display the remaining content.

Input:

- The string to ignore (lines containing this string will be excluded).
- The number of lines to write to the file.
- The content for each line.

Operations/Instructions:

1. Prompt the user to input the string to ignore in the lines.
2. Prompt the user to input the number of lines to write in the file.
3. Prompt the user to input the content for each line.
4. Create a new text file named `myfile.txt`.
5. Write the user-provided lines into the file, excluding any line that contains the specified string.
6. Read the content from the file.
7. Display the remaining content of the file on the console.

Output:

The output should display the remaining content of the file, as follows:

```
Input the string to ignore the line:
journey Input number of lines to
write in the file: 4
Input line 1: Happiness is a journey, not a
destination. Input line 2: Kindness costs
nothing but means everything.
Input line 3: Success is the sum of small efforts repeated day in
and day out. Input line 4: Courage is not the absence of fear;
it's the triumph over it.

The lines have been ignored which contain the string
'journey'. The content of the file is:
-----------------------------------------

Kindness costs nothing but means everything.
Success is the sum of small efforts repeated day in and
day out. Courage is not the absence of fear; it's the
triumph over it.
```

Hints:

- Use the `System.IO` namespace to handle file operations.
- Use LINQ or simple loop with conditionals to filter out lines containing the specified string.
- Use the `File.WriteAllLines` method to write the lines to the file.
- Use the `File.ReadAllLines` method to read the content from the file.
- Handle any potential exceptions using a `try-catch` block to ensure the program does not crash in case of an error.
-

10.5 Add some more

Demonstrate appending additional text to an existing file. Initially, the program will create a new text file and write some lines of text into it. Next, it will append more lines of text to the same file. The program should display the content of the file before and after appending.

Input:

Initial content to write to the file.

Additional content to append to the file.

Operations:

1. Create a new text file named `myfile.txt`.
2. Write the initial content provided by the user into the file.

3. Append the additional content provided by the user to the same file.
4. Read and display the content of the file before and after appending.

Output:

The output should display the content of the file before and after appending, as follows:

```
 Initial content of the file:
----------------------------------------
 This is the initial content of

 the file. Content of the file

 after appending:
----------------------------------------
This is the initial content of the file. Now appending some more
text to the file.
```

Hints:

- Use the `System.IO` namespace to handle file operations.
- Use `File.WriteAllText` to write the initial content to the file.
- Use `File.AppendAllText` to append additional content to the file.
- Use `File.ReadAllText` to read the content from the file.
- Handle any potential exceptions using a `try-catch` block to ensure the program does not crash in case of an error.

10.6 Snapshot

Create a snapshot by copying the entire content of one text file to another. The program should prompt the user to specify the source file and the destination file. It will then read the content from the source file and write it into the destination file, preserving the original content. Finally, the program should display the content of the copied file.

Input:

- Source file path (from which content will be copied).
- Destination file path (where content will be copied).

Operations:

1. Prompt the user to input the path of the source file.
2. Prompt the user to input the path of the destination file.
3. Read the entire content from the source file.
4. Write the read content into the destination file.
5. Read the content from the destination file.
6. Display the content of the copied file on the console.

Output:

The program should output the content of the copied file on the console after copying the content from the source file to the destination file.

```
 Input the source file path: C:\myfolder\sourcefile.txt
 Input the destination file path:

 C:\myfolder\destinationfile.txt Successfully copied contents

 from 'sourcefile.txt' to 'destinationfile.txt'.

 Content of 'destinationfile.txt':
-----------------------------------------

 This is the content of the source file that has been copied.
```

Hints:

- Use the `System.IO` namespace to handle file operations.
- Use `File.ReadAllText` to read the content from the source file.
- Use `File.WriteAllText` to write the content to the destination file.
- Use `File.ReadAllText` again to read the content from the destination file for display.
- Handle any potential exceptions using a `try-catch` block to ensure the program does not crash in case of an error.

10.7 Count, Top & Tail

Count the number of lines in a text file and displays both the first (top) and last (tail) lines of the file. The program should prompt the user to input the file path. It will then read the content from the file, count the lines, and output the first and last lines along with the total number of lines in the file.

Input:

- File path of the text file.

Operations/Instructions:

1. Prompt the user to input the path of the text file.
2. Read the entire content of the file.
3. Count the number of lines in the file.
4. Display the first line of the file.
5. Display the last line of the file.
6. Display the total number of lines in the file.

Output:

```
 Input the file path: C:\myfolder\sample.txt

 Content of the file is:
------------------------------------------------------------------

 1. Positivity means focusing on the good in every situation.
 2. It involves staying optimistic and hopeful.
 3. Positivity helps in overcoming challenges with resilience.
 4. It promotes a sense of inner peace and calmness.
 5. Cultivating positivity leads to a happier and more fulfilling
    life.


 First line of the file:
------------------------------------------

 1. Positivity means focusing on the good in every situation.

 Last line of the file:
------------------------------------------

 5. Cultivating positivity leads to a happier and more fulfilling
    life.

 Total number of lines in the file: 5
```

Hints:

- Use the `System.IO` namespace to handle file operations.
- Use `File.ReadAllLines` to read all lines from the file into an array.
- Access the first element (`lines[0]`) and last element (`lines[lines.Length - 1]`) of the array to get the first and last lines.
- Use `lines.Length` to get the total number of lines.
- Handle any potential exceptions using a `try-catch` block to ensure the program does not crash in case of an error.

10.8 Selective File Reading

Read a specific line or set of lines from a text file based on user input. The program should prompt the user to input the file path and the line number(s) they wish to retrieve. It will then open the file, read the specified line(s), and display them to the user.

Input:

- File path of the text file.
- Line number(s) to read from the file (e.g., single line, multiple lines).

Operations:

1. Prompt the user to input the path of the text file.
2. Prompt the user to input the line number(s) they want to read.
3. Read the specified line(s) from the file.
4. Display the content of the selected line(s) to the user.

Output:

The program should output the content of the selected line(s) from the file based on the user's input.

```
 Input the file path: C:\myfolder\sample.txt

 Input the line number(s) to read (e.g., 1,

 3, 5): 2 Content of line 2 in

 'sample.txt':
-----------------------------------------

 2. It involves staying optimistic and hopeful.
```

Hints:

- Use the `System.IO` namespace to handle file operations.
- Use `StreamReader` to read the file and `ReadLine()` method to navigate to specific lines.
- Handle exceptions such as file not found or invalid line numbers using `try-catch` blocks.

10.9 n-Tail Extract

Read and display the last *n* lines from a text file. The program should prompt the user to input the file path and the number of lines they wish to retrieve from the end of the file. It will then read the file, extract the specified number of lines from the end, and display them.

Input:

File path of the text file.

Number of lines *n* to read from the end of the file.

Operations:

1. Prompt the user to input the path of the text file.

2. Prompt the user to input the number of lines *n* they want to read from the end of the file.
3. Read the entire content of the file.
4. Extract the last *n* lines from the file content.
5. Display the extracted lines to the user.

Output:

The program should output the last *n* lines from the specified text file.

```
 Input the file path: C:\myfolder\sample.txt
 Input the number of lines to read from the end of the file: 3

 Content of the last 3 lines in 'sample.txt':
-----------------------------------------

 3. Positivity helps in overcoming challenges with resilience.
 4. It promotes a sense of inner peace and calmness.
 5. Cultivating positivity leads to a happier and more fulfilling
    life.
```

Hints:

Use the `System.IO` namespace to handle file operations.

Use `File.ReadAllLines` to read all lines from the file into an array.

Use array slicing to access the last *n* elements of the array.

Handle any potential exceptions using a `try-catch` block to ensure the program does not crash in case of an error.

10.10 Strip Words

Remove specific words from a text file based on a list of stop words provided in another file. The program should prompt the user to input the file paths of both the text file and the stop words file. It will then read both files, remove all instances of the stop words from the text content, and save the cleaned content to a new file.

Input:

- File path of the input text file.
- File path of the stop words file.
- File path to save the cleaned text file.

Operations:

1. Prompt the user to input the path of the input text file.

2. Prompt the user to input the path of the stop words file.
3. Prompt the user to input the path to save the cleaned text file.
4. Read the content of the input text file.
5. Read the list of stop words from the stop words file.
6. Remove all instances of the stop words from the text content.
7. Write the cleaned text to the specified output file.
8. Display a message indicating that the process is complete and the cleaned text has been saved.

Output:

The program should output the cleaned text to the specified output file, excluding all words listed in the stop words file.

```
Input the file path of the input text file:
C:\myfolder\input.txt Input the file path of the stop
words file: C:\myfolder\stopwords.txt
Input the file path to save the cleaned text file:
C:\myfolder\output.txt

Stop words removed
successfully. Cleaned
content saved to
'output.txt'.
```

Hints:

- Use the `System.IO` namespace to handle file operations.
- Use `File.ReadAllText` to read the file content and `File.ReadAllLines` to read the stop words.
- Use string manipulation methods to remove the stop words from the text.
- Handle exceptions using a `try-catch` block to manage potential errors, such as file not found or access issues.

Note:

Verify the files**Foundations of LINQ and Generic Types**

Note: Students are encouraged to bring their own laptops for laboratory practice sessions.

11.1 Filtering with LINQ

Showcase the use of LINQ to filter even numbers from a given list of integers.

Input:

You are provided with a list of integers.

Operations:

1. Implement a LINQ query to filter out only the even numbers from the list.
2. Display or output the filtered list of even numbers.

Output:

For example, given the input list `[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]`,
the program should output:

```
Even numbers: 2, 4, 6, 8, 10
```

Hints:

- Use LINQ's `Where` method along with a lambda expression to filter numbers.
- Ensure to handle edge cases such as an empty list or no even numbers present in the list.

11.2 Sorting Objects with LINQ

Implement a LINQ query to sort a list of objects by a property in ascending order.

Input:

You are provided with a list of objects.

Operations:

1. Implement a LINQ query to sort the list of objects by a specified property in ascending order.
2. Display or output the sorted list of objects.

Output:

For example, given a list of objects with a property `Age`, the program should output:

```
Sorted list by Age: [Object1, Object2, Object3, ...]
```

Hints:

- Use LINQ's `OrderBy` method to sort objects by the specified property.
- Handle cases where the list may be empty or the property values are null.

11.3 Grouping and Counting with LINQ

Group a list of students by their age and count the number of students in each group using LINQ's `GroupBy` and `Count` methods.

Input:

You are provided with a list of student objects, each containing an `Age` property.

Operations:

1. Implement a LINQ query to group students by their `Age` property.
2. Calculate and display the count of students in each age group.

Output:

For example, given a list of students:

```
[
{ Name: "Surya", Age: 21 },
{ Name: "Keerthi", Age: 22 },
{ Name: "Venkatesh", Age: 21 },
{ Name: "Anusha", Age: 22 },
{ Name: "Padma", Age: 23 }
]
```

The program should output:

```
Age 21: 2 students
Age 22: 2 students
Age 23: 1 student
```

Hints:

- Use LINQ's `GroupBy` method to group objects by a property.
- Use LINQ's `Count` method within the grouped result to determine the number of items in each group.
- Handle cases where the list may be empty or where age values may be null.

11.4 Aggregation with LINQ

Calculate the sum and average of a list of numeric values using LINQ's aggregate functions.

Input:

You are provided with a list of numeric values.

Operations:

1. Implement a LINQ query to calculate the sum of the numeric values.
2. Implement a LINQ query to calculate the average of the numeric values.

Output:

For example, given a list of numeric values:

```
[10, 20, 30, 40, 50]
```

The program should output:

```
Sum: 150
Average: 30
```

Hints:

- Use LINQ's `Sum` method to compute the sum of elements in the list.
- Use LINQ's `Average` method to compute the average of elements in the list.
- Ensure to handle cases where the list may be empty or where values are null.

11.5 Distinct Values with LINQ

Remove duplicate strings from an array using LINQ's `Distinct` method.

Input:

You are provided with an array of strings.

Operations:

1. Implement a LINQ query to remove duplicate strings from the array.

Output:

For example, given an array of strings:

```
["apple", "orange", "banana", "apple", "grapes", "banana"]
```

The program should output:

```
Distinct strings: ["apple", "orange", "banana", "grapes"]
```

Hints:

- Use LINQ's `Distinct` method to filter out duplicate elements from the array.
- Maintain the order of elements in the resulting sequence.
- Handle cases where the array may be empty or contain null values.

11.6 Pagination with LINQ

Implement paging functionality to retrieve a specific number of records from a collection using LINQ's `Skip` and `Take` methods.

Input:

You are provided with a collection of records (e.g., a list or array).

Operations:

1. Implement a LINQ query to skip a specified number of records (offset) and take a specified number of records (page size).

Output:

For example, given a collection of records:

```
["Record1", "Record2", "Record3", "Record4", "Record5", "Record6"]
```

If you want to display records for page 2 with page size 2, the program should output:

```
["Record3", "Record4"]
```

Hints:

- Use LINQ's `Skip` method to skip a specified number of elements.
- Use LINQ's `Take` method to retrieve a specified number of elements after skipping.
- Ensure to handle cases where the collection may be empty or the specified page and page size exceed the collection length.

11.7 Conditional Filtering with LINQ

Filter out strings containing a specific substring from a list of strings using LINQ.

Input:

You are provided with a list of strings.

Operations:

1. Implement a LINQ query to filter strings that contain a specified substring.

Output:

For example, given a list of strings:

```
["apple", "orange", "banana", "pineapple", "grapes", "kiwi"]
```

If the substring to filter is "apple", the program should output:

```
Filtered strings: ["apple", "pineapple"]
```

Hints:

- Use LINQ's `Where` method with a lambda expression to filter strings based on the sub- string condition.
- Ensure to handle cases where the list may be empty or where the substring might not match any strings in the list.

11.8 Projecting Data with LINQ

Select only the names of students from a list of student objects using LINQ's `Select`

clause.

Input:

You are provided with a list of student objects, each containing a `Name` property.

Operations:

1. Implement a LINQ query to extract and project only the names of students from the list.

Output:

For example, given a list of student objects:

```
[
{ Name: "Surya", Age: 21 },
{ Name: "Keerthi", Age: 22 },
{ Name: "Venkatesh", Age: 21 },
{ Name: "Anusha", Age: 22 },
{ Name: "Padma", Age: 23 }
]
```

The program should output:

```
Student names: ["Surya", "Keerthi", "Venkatesh", "Anusha",
"Padma"]
```

Hints:

- Use LINQ's `Select` method to project the `Name` property from each student object.
- Ensure to handle cases where the list may be empty or where student names might be null or empty strings.

11.9 Join Operations with LINQ

Perform an inner join between two lists of objects based on a common property using LINQ.

Input:

You are provided with two lists of objects, each containing a common property for joining (e.g., Student objects with a common property like `Age`).

Operations:

1. Implement a LINQ query to perform an inner join between the two lists based on the common property.

Output:

For example, given two lists of objects:

```
List A:
[
{ Name: "Surya", Age: 21 },
{ Name: "Keerthi", Age: 22 },
```

```
{ Name: "Venkatesh", Age: 21 },
{ Name: "Anusha", Age: 22 },
{ Name: "Padma", Age: 23 }
]

List B:
[
{ Name: "Ravi", Age: 21 },
{ Name: "Kiran", Age: 22 },
{ Name: "Vijaya", Age: 23 },
{ Name: "Ramya", Age: 24 }
]
```

If joining based on the `Age` property, the program should output:

```
Inner join result:
[
{ NameA: "Surya", NameB: "Ravi", Age: 21 },
{ NameA: "Keerthi", NameB: "Kiran", Age: 22 },
{ NameA: "Padma", NameB: "Vijaya", Age: 23 }
]
```

Hints:

- Use LINQ's `Join` method along with a lambda expression to perform the inner join.
- Ensure to handle cases where there are no common elements between the lists or where the lists may be empty.

11.10 Handling Null Values with LINQ

Safely handle null values when selecting data using LINQ's null-conditional operators (`?.` and `??`).

Input:

You are provided with a collection of objects, some of which may contain null values for certain properties.

Operations:

1. Implement a LINQ query to select data from the collection, ensuring safe navigation through potentially null properties using `?.`.
2. Use the null-coalescing operator `??` to provide default values or handle null cases gracefully.

Output:

For example, given a collection of objects:

```
[
{ Name: "Ravi", Age: 30 },
```

```
{ Name: "Priya", Age: null },
{ Name: "Arun", Age: 25 },
{ Name: null, Age: 28 }
]
```

The program should output:

```
Selected data with handling null values:
[
{ Name: "Ravi", Age: 30 },
{ Name: "Priya", Age: 0 },       % Assume default value 0 for Age
{ Name: "Arun", Age: 25 },
{ Name: "", Age: 28 }            % Assume empty string "" for Name
]
```

Hints:

- Use LINQ's null-conditional operator `?.` to safely navigate through potentially null prop- erties.
- Use the null-coalescing operator `??` to provide default values or handle null cases.
- Handle scenarios where entire objects might be null in the collection.

Advanced Methods and Collections

Note: Students are encouraged to bring their own laptops for laboratory practice sessions.

12.1 Advanced Sorting with LINQ

Sort a list of objects by multiple criteria using LINQ's `OrderBy` and `ThenBy` methods.

Input:

You are provided with a list of objects, each containing multiple properties for sorting (e.g., Student objects with properties like `Name`, `Age`, and `Grade`).

Operations:

1. Implement a LINQ query to sort the list of objects primarily by one property using `OrderBy` and secondarily by another property using `ThenBy`.

Output:

For example, given a list of student objects:

```
[
{ Name: "Surya", Age: 21, Grade: "A" },
{ Name: "Keerthi", Age: 22, Grade: "B" },
{ Name: "Venkatesh", Age: 21, Grade: "C" },
{ Name: "Anusha", Age: 22, Grade: "A" },
{ Name: "Padma", Age: 23, Grade: "B" }
```

```
]
```

The program should output:

```
Sorted list:
[
{ Name: "Anusha", Age: 22, Grade: "A" },
{ Name: "Keerthi", Age: 22, Grade: "B" },
{ Name: "Padma", Age: 23, Grade: "B" },
{ Name: "Surya", Age: 21, Grade: "A" },
{ Name: "Venkatesh", Age: 21, Grade: "C" }
]
```

Hints:

- Use LINQ's `OrderBy` method to sort by the primary property and `ThenBy` method to sort by the secondary property.
- Handle cases where the list may be empty or where sorting criteria may include null values or unexpected data types.

12.2 Grouping and Aggregation with LINQ

Group a list of sales transactions by month and calculate the total sales amount for each month using LINQ.

Input:

You are provided with a list of sales transactions, each containing a `Date` (representing the transaction date) and `Amount` (representing the sales amount).

Operations:

1. Implement a LINQ query to group the sales transactions by month based on the `Date` property.
2. Calculate the total sales amount for each month using LINQ's aggregation functions.

Output:

For example, given a list of sales transactions:

```
[
{ Date: "2024-01-05", Amount: 100.00 }
                                      ,
{ Date: "2024-02-15", Amount: 150.00 }
                                      ,
{ Date: "2024-01-20", Amount: 120.00 }
                                      ,
{ Date: "2024-03-10", Amount: 200.00 }
                                      ,
{ Date: "2024-02-28", Amount: 180.00 }
]
```

The program should output:

```
Sales totals by month:
January: 220.00
February: 330.00
March: 200.00
```

Hints:

- Use LINQ's `GroupBy` method to group transactions by month.
- Use LINQ's aggregation methods like `Sum` to calculate the total sales amount for each group.
- Handle cases where there are no transactions for a particular month or where the list may be empty.

12.3 Joining Complex Data with LINQ

Perform an outer join between two lists of objects with a fallback value for missing matches using LINQ's `Join` and `DefaultIfEmpty` methods.

Input:

You are provided with two lists of objects, each containing properties that can be matched for joining (e.g., Employee objects with a common property like `DepartmentId`).

Operations:

1. Implement a LINQ query to perform an outer join between the two lists based on the common property.
2. Use `DefaultIfEmpty` to provide a fallback value when no matches are found in the second list.

Output:

For example, given two lists of objects:

```
List A:
[
{ EmployeeId: 1, Name: "Ravi", DepartmentId: 101 },
{ EmployeeId: 2, Name: "Priya", DepartmentId: 102 },
{ EmployeeId: 3, Name: "Arun", DepartmentId: 103 }
]

List B:
[
{ DepartmentId: 101, DepartmentName: "IT" },
{ DepartmentId: 103, DepartmentName: "HR" }
]
```

If joining based on the `DepartmentId` property, the program should output:

```
Outer join result:
[
```

```
{ EmployeeId: 1, Name: "Ravi", DepartmentId: 101, DepartmentName:
"IT" },
{ EmployeeId: 2, Name: "Priya", DepartmentId: 102, DepartmentName:
null },
{ EmployeeId: 3, Name: "Arun", DepartmentId: 103, DepartmentName:
"HR" }
]
```

Hints:

- Use LINQ's `Join` method to perform the outer join and `DefaultIfEmpty` method to handle missing matches.
- Ensure to handle cases where there are no common elements between the lists or where the lists may be empty.

12.4 Union and Concatenation with LINQ

Combine two lists into one unique list using LINQ's `Union` and `Concat` methods.

Input:

You are provided with two lists of objects.

Operations:

1. Implement a LINQ query to combine the two lists into one unique list.
2. Use `Union` to ensure that duplicate elements are removed and `Concat` to concatenate both lists while preserving duplicates.

Output:

For example, given two lists:

```
List A: [1, 2, 3, 4]
List B: [3, 4, 5, 6]
```

The program should output:

```
Unique combined list (Union):
[1, 2, 3, 4, 5, 6]

Concatenated list (Concat):
[1, 2, 3, 4, 3, 4, 5, 6]
```

Hints:

- Use LINQ's `Union` method to merge the lists and remove duplicates.
- Use LINQ's `Concat` method to concatenate both lists while preserving duplicates.
- Handle cases where the lists may be empty or contain null elements.

12.5 Element Manipulation with LINQ

Use LINQ to reverse the order of elements in a list and select specific elements based on index.

Input:

You are provided with a list of elements (e.g., integers, strings).

Operations:

1. Implement a LINQ query to reverse the order of elements in the list.
2. Select specific elements from the reversed list based on given indices (e.g., select elements at indices 0, 2, 4).

Output:

For example, given a list of integers:

```
Input list: [1, 2, 3, 4, 5, 6, 7, 8, 9]
Indices to select: [0, 2, 4]
```

The program should output:

```
Reversed list: [9, 8, 7, 6, 5, 4, 3, 2, 1]
Selected elements: [9, 7, 5]
```

Hints:

- Use LINQ's `Reverse` method to reverse the order of elements in the list.
- Use LINQ's `Where` method with a condition based on index to select specific elements.
- Handle cases where the list may be empty or where the indices provided may exceed the list length.

12.6 Subqueries with LINQ

Write a LINQ query that includes a subquery to retrieve data from related tables or nested collections.

Input:

You are provided with two related collections or tables, where one collection/table contains objects that reference objects in the other collection/table.

Operations:

1. Implement a LINQ query that includes a subquery to retrieve data from the related tables or nested collections.
2. Use appropriate LINQ operators to perform operations such as filtering, projecting, or aggregating data from the subquery.

Output:

For example, given two collections of objects:

```
List A:
[
```

```
{ EmployeeId: 1, Name: "Ravi", DepartmentId: 101 },
{ EmployeeId: 2, Name: "Priya", DepartmentId: 102 },
{ EmployeeId: 3, Name: "Arun", DepartmentId: 103 }
]

List B:
[
{ DepartmentId: 101, DepartmentName: "IT" },
{ DepartmentId: 102, DepartmentName: "Finance" },
{ DepartmentId: 103, DepartmentName: "HR" }
]
```

Implement a LINQ query to retrieve employees along with their department names:

```
LINQ query result:
[
{ EmployeeId: 1, Name: "Ravi", DepartmentId: 101, DepartmentName:
"IT" },
{ EmployeeId: 2, Name: "Priya", DepartmentId: 102, DepartmentName:
"Finance" },
{ EmployeeId: 3, Name: "Arun", DepartmentId: 103, DepartmentName:
"HR" }
]
```

Hints:

- Use LINQ's `Join` or `SelectMany` methods to perform the subquery operation.
- Ensure to handle cases where there are no common elements between the collections or where the collections may be empty.

12.7 Error Handling in LINQ Queries

Handle exceptions and errors gracefully in LINQ queries, particularly when dealing with null values or unexpected data.

Input:

You are provided with data that may include null values or unexpected formats.

Operations:

1. Implement a LINQ query that gracefully handles potential exceptions or errors that may arise during data processing.
2. Use error-handling techniques such as null checks, exception handling blocks, or default value assignments to manage unexpected scenarios.

Output:

For example, given a list of integers with potential null values:

```
Input list: [1, null, 3, 4, null, 6, 7, 8, null]
```

The program should handle null values and unexpected data formats gracefully, ensuring the query operates without throwing exceptions.

Hints:

- Use LINQ's `Where` method with null checks or `Select` method with conditional operators to filter or transform data while handling nulls.
- Consider using try-catch blocks around LINQ queries to catch and manage exceptions that may occur during data processing.

12.8 Custom Filtering using LINQ and Lambda Expressions

Implement custom filtering of a generic collection of objects dynamically based on user-defined criteria using LINQ and lambda expressions.

Input:

You are provided with a generic collection of objects.

Operations:

1. Implement a LINQ query that allows dynamic filtering of objects based on user-defined criteria.
2. Use lambda expressions to construct flexible filtering conditions, such as filtering by specific properties or applying complex logical operations.

Output:

For example, given a list of `Product` objects:

```
List of Products:
[
{ Id: 1, Name: "Laptop", Price: 1200 },
{ Id: 2, Name: "Smartphone", Price: 800 },
{ Id: 3, Name: "Tablet", Price: 500 }
]
```

Implement a LINQ query that allows filtering products based on user-defined criteria, such as products with a price greater than 600:

```
Filtered list of Products (Price > 600):
[
{ Id: 1, Name: "Laptop", Price: 1200 },
{ Id: 2, Name: "Smartphone", Price: 800 }
]
```

Hints:

- Use LINQ's `Where` method with lambda expressions to apply dynamic filtering conditions.
- Allow users to input filtering criteria interactively or define them programmatically.

12.9 Dynamic Queries with LINQ

Build dynamic LINQ queries based on runtime conditions and user inputs using expression trees and Queryable extensions.

Input:

You are provided with a dataset that needs to be queried dynamically based on user-defined conditions.

Operations:

1. Implement a LINQ query that can dynamically adjust its filtering, sorting, or projection based on runtime conditions.
2. Utilize expression trees and Queryable extensions to construct LINQ queries programmat- ically.

Output:

For example, given a dataset of `Employee` objects:

```
List of Employees:
[
{ Id: 1, Name: "Ravi", Department: "HR", Salary: 50000 },
{ Id: 2, Name: "Priya", Department: "IT", Salary: 60000 },
{ Id: 3, Name: "Arun", Department: "Finance", Salary: 55000 }
]
```

Implement a LINQ query that can dynamically filter employees based on user inputs (e.g., filter by department and salary range):

```
Filtered list of Employees (Department: IT, Salary > 55000): [
{ Id: 2, Name: "Priya", Department: "IT", Salary: 60000 }
]
```

Hints:

- Use expression trees to build LINQ queries dynamically based on conditions such as user input.
- Explore LINQ's Queryable extensions to handle sorting, paging, and complex filtering scenarios dynamically.

12.10 Parallel LINQ (PLINQ)

Write LINQ queries that leverage PLINQ for parallel execution and improved performance.

Input:

You are provided with a dataset that can benefit from parallel processing.

Operations:

1. Implement LINQ queries using PLINQ to parallelize operations such as filtering, sorting, or aggregation.

2. Utilize PLINQ's parallel execution capabilities to enhance performance on large datasets.

Output:

For example, given a dataset of `Product` objects:

```
List of Products:
[
{ Id: 1, Name: "Laptop", Price: 1200 },
{ Id: 2, Name: "Smartphone", Price: 800 },
{ Id: 3, Name: "Tablet", Price: 500 }
]
```

Implement a PLINQ query that parallelizes operations like filtering or aggregation:

```
Filtered list of Products (Price > 600
using PLINQ): [
{ Id: 1, Name: "Laptop", Price: 1200 },
{ Id: 2, Name: "Smartphone", Price: 800 }
]
```

Hints:

- Use PLINQ's `AsParallel` method to enable parallel execution of LINQ queries.
- Consider the benefits and limitations of parallel processing, such as thread safety and overhead.

Explore Threads

Note: Students are encouraged to bring their own laptops for laboratory practice sessions.

13.1 Thread Life Cycle

Develop a C# program that demonstrates the lifecycle management of a worker thread, including starting, suspending, resuming, and terminating the thread.

Requirements

1. Create a new thread and start its execution.
2. Pause the main thread for a specific duration to simulate a wait period.
3. Suspend the worker thread after a designated time interval.
4. Resume the suspended thread after another wait period.
5. Terminate the worker thread gracefully using appropriate thread abort mechanisms.
6. Display messages to indicate each phase of the thread's lifecycle.

Expected Output

The program should output messages indicating the various stages of the thread's lifecycle, including starting, suspending, resuming, and terminating:

```
Main thread: Thread
started. Worker
thread: Starting
work... Worker
thread: Working...
Part 1 Worker
thread: Working...
Part 2
Main thread: Suspending the worker
thread... Main thread: Resuming
the worker thread...
Worker thread:
Working... Part 3 Worker
thread: Working... Part
4
Main thread: Aborting the
worker thread... Worker
thread: Working... Part 5
Exception thrown: 'System.Threading.ThreadAbortException' in
mscorlib.dll Worker thread: Aborted.
Exception thrown: 'System.Threading.ThreadAbortException' in
CSharpLabProblems. Worker thread: Exiting.
```

```
Main thread: Thread lifecycle management completed.
```

13.2 Thread Naming and Priority

Create multiple threads, assigns names to each thread, and sets different priorities. Display the thread names and priorities during execution.

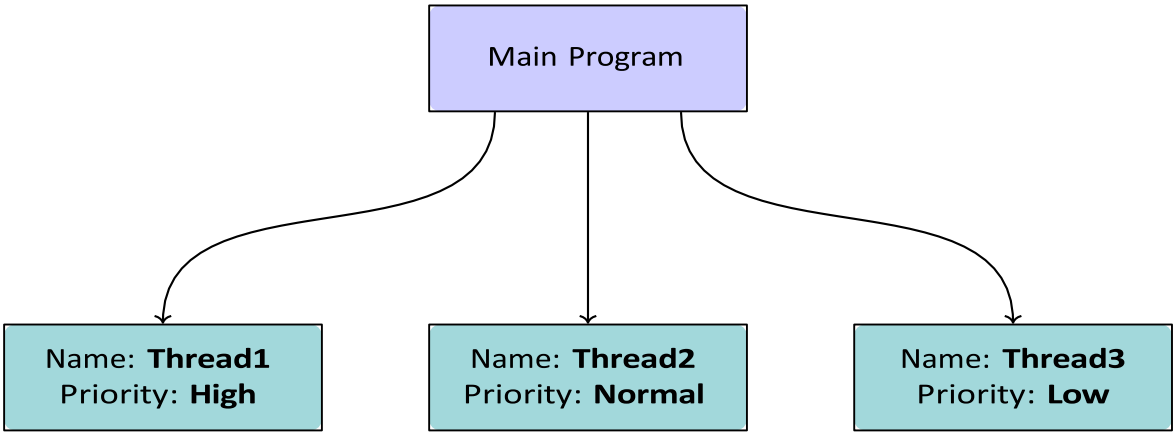


Figure 13.1: Thread naming and priorities

Instructions:

1. Create a list of tasks to be executed by different threads.
2. Assign a unique name to each thread.
3. Set different priority levels (High, Normal, Low) for each thread.
4. Start all threads and display their names and priorities during execution.
5. Ensure all threads complete their tasks.

Output:

```
Thread Name: Thread1, Priority: High, Status:
Running Thread Name: Thread2, Priority:
Normal, Status: Running Thread Name: Thread3,
Priority: Low, Status: Running
```

13.3 Multiple Threads

Develop a program that demonstrates the lifecycle management of multiple threads, including starting each thread, waiting for them to complete their work, and printing specific messages at different stages of their execution.

Program Requirements

1. Create three separate threads, each performing a simulated task (e.g.,

printing messages, performing calculations).

2. Print messages when each thread starts its work and when each thread completes its task.
3. Ensure the main thread waits for all threads to finish their tasks before proceeding further.
4. Display a message indicating when all threads have completed their tasks.

Output

The program should output messages in the following sequence:

```
Thread      1:
Starting
work...
Thread      2:
Starting
work...
Thread      3:
Starting
work...
Main thread: Waiting for all threads to
complete... Thread 1: Work completed.
Thread 2: Work
completed.
Thread 3: Work
completed.
Main thread: All threads have completed.
```

13.4 Thread Synchronization with Mutex

Develop an application where multiple threads increment a shared counter. Use a mutex to synchronize access to the counter and ensure thread safety.

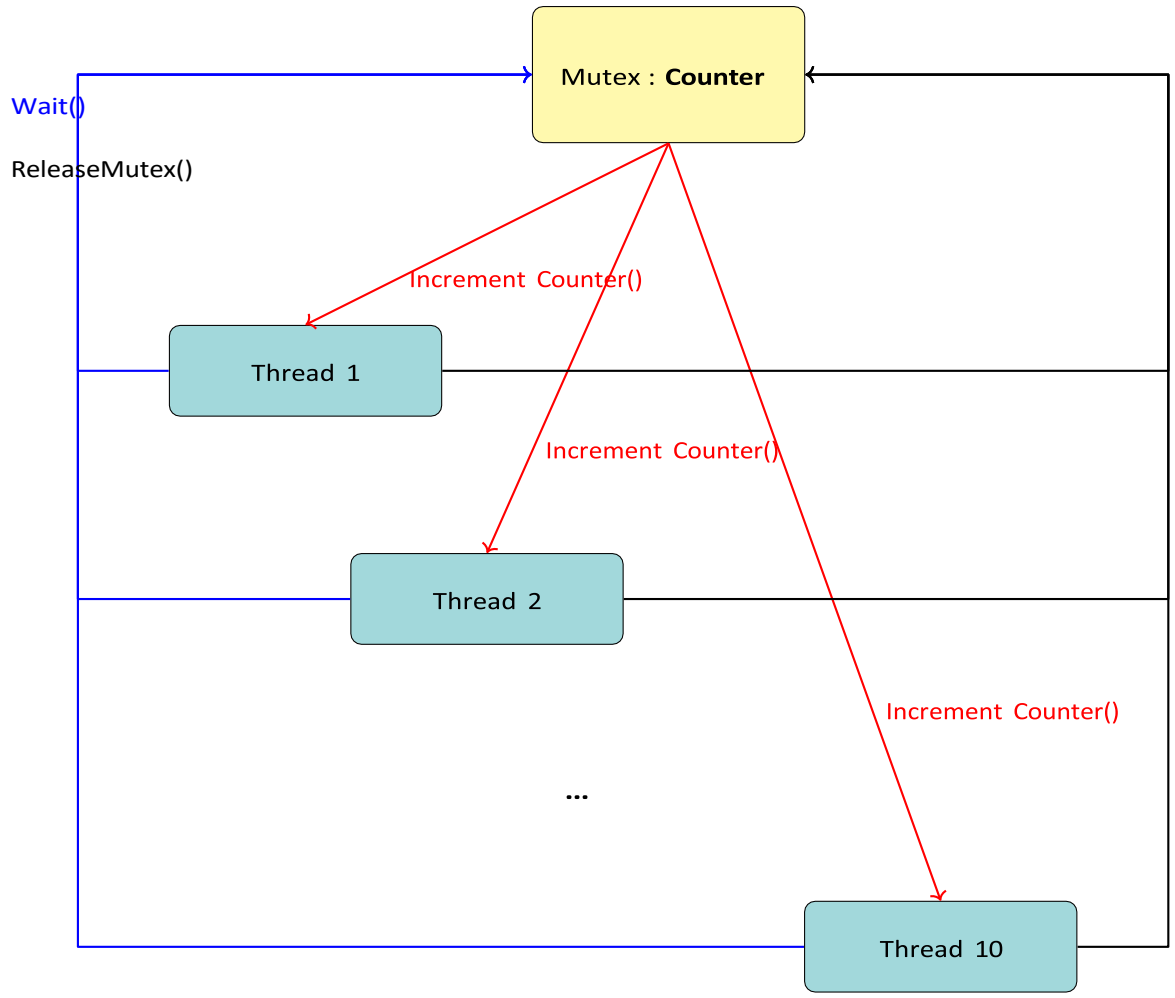


Figure 13.2: Thread synchronization with mutex, illustrating Wait(), Increment Counter, and ReleaseMutex() actions

Instructions:

Create a shared counter variable.

Create multiple threads that increment the counter.

Use a mutex to ensure only one thread can increment the counter at a time.

Start all threads and wait for their completion.

Display the final value of the counter.

Output:

```
Final Counter Value: 1000
```

13.5 Producer-Consumer Problem

Implement the producer-consumer problem using threads. Use condition variables to synchronize producer and consumer actions.

Instructions:

- Create a shared buffer (queue).
- Implement a producer thread that generates items and adds them to the buffer.
- Implement a consumer thread that removes and processes items from the buffer.
- Use condition variables to notify the consumer when new items are available and the producer when there is space in the buffer.
- Ensure proper synchronization to avoid race conditions.

Output:

```
Producer: Produced item 1
Buffer is full. Waiting for
consumer... Consumer: Consumed
item 1
Buffer is empty. Waiting for
producer... Producer: Produced
item 2
Buffer is full. Waiting for
consumer... Consumer: Consumed
item 2
Buffer is empty. Waiting for
producer... Producer: Produced
item 3
Buffer is full. Waiting for
consumer... Consumer: Consumed
item 3
...
```

13.6 Thread Pool Implementation

Design a thread pool manager that manages a fixed number of worker threads. Implement task submission and ensure tasks are executed concurrently.

Instructions:

- Create a thread pool manager with a fixed number of threads.
- Implement a method to submit tasks to the thread pool.
- Ensure tasks are executed concurrently by available threads in the pool.
- Use synchronization techniques to manage task execution and completion.
- Display the status of tasks and thread pool utilization.

Output:

```
Task 1 started by Thread Thread 1.
Task 2 started by Thread Thread 2.
Task 2 completed by Thread Thread 2.
Task 1 completed by Thread Thread 1.
Task 3 started by Thread Thread 2.
Task 3 completed by Thread Thread 2.
```

Mini Projects

Note: Students are encouraged to bring their own laptops for laboratory practice sessions.

14.1 Desktop Applications

1. **Student Management System:** Manage student records, courses, grades, and attendance.
2. **Inventory Management System:** Track inventory levels, orders, and sales.
3. **Library Management System:** Manage books, patrons, and borrowing records.
4. **Hospital Management System:** Manage patient records, appointments, and medical history.
5. **Employee Management System:** Manage employee information, payroll, and leave applications.
6. **Task Management System:** Track tasks, assignees, and project deadlines.
7. **Chat Application:** Implement real-time messaging between users or groups.

14.2 Web Applications

1. **Online Quiz Application:** Create quizzes with multiple-choice questions.
2. **E-commerce Website:** Develop a fully functional online store with product listings and shopping cart.
3. **Blog or Content Management System:** Manage blog posts, articles, and multimedia content.
4. **Customer Relationship Management (CRM) System:** Manage customer interac- tions, sales leads, and support tickets.
5. **Personal Finance Manager:** Track income, expenses, budgets, and savings goals. On- line Learning Platform: Host courses, quizzes, and educational resources.

14.3 Mobile Applications

1. **Fitness Tracker:** Track workouts, set goals, and monitor progress.
2. **Recipe Manager:** Organize recipes, create shopping lists, and meal plans.
3. **Expense Tracker App:** Track expenses, categorize spending, and set budgets.
4. **Language Learning App:** Provide lessons, quizzes, and interactive exercises for learning languages.
5. **Tourist Guide App:** Offer information on attractions, maps, and local recommendations.

14.4 Game Development

1. **2D Platformer Game:** Create a classic side-scrolling platform game with levels and obstacles.
2. **Puzzle Game:** Design and implement various puzzles with logic challenges and levels.
3. **RPG (Role-Playing Game):** Develop a role-playing game with quests, characters, and combat mechanics.
4. **Simulation Game:** Build a simulation game around a specific theme like city-building or farming.
5. **Card Game:** Implement a card game with rules, multiplayer support, and AI opponents.

14.5 IoT Applications

1. **Smart Home Control System:** Create an application to control IoT devices at home (lights, temperature, security cameras).
2. **Environmental Monitoring System:** Develop a system to collect and display data from sensors (temperature, humidity, air quality).
3. **Health Monitoring Device Integration:** Integrate wearable health devices (like fitness trackers) to track and display health metrics.
4. **Industrial IoT Dashboard:** Build a dashboard to monitor and manage industrial equip- ment and processes.
5. **Smart Agriculture System:** Develop an IoT solution for monitoring soil moisture, temperature, and crop health.

14.6 Data Analysis and Reporting

1. **Dashboard for Business Analytics:** Create a dashboard to visualize sales data, trends, and key performance indicators (KPIs).
2. **Real-time Data Streaming Application:** Build an application to process and visualize real-time data streams (IoT sensor data, social media feeds).
3. **Predictive Analytics Tool:** Develop a tool to analyze historical data and make predic- tions using machine learning algorithms.
4. **Financial Portfolio Tracker:** Track investments, analyze portfolio performance, and provide recommendations.
5. **Healthcare Data Visualization:** Visualize patient data, medical trends, and health outcomes for healthcare professionals.

14.7 Guidelines for Mini Projects

This section provides guidelines for students to undertake various mini projects such as web applications, mobile applications, gaming applications, IoT applications, and data mining applications. These guidelines include the necessary knowledge and tools required for each project type, considering that students have covered up to multi-threading in their course.

14.7.1 Desktop Applications

Knowledge Required:

- Basics of desktop application development
- Understanding of Windows Forms and WPF (Windows Presentation Foundation)
- Event-driven programming
- Multi-threading for responsive UI
- Data binding and MVVM (Model-View-ViewModel) architecture

Tools and Technologies:

- Visual Studio
- .NET Framework or .NET Core
- Windows Forms or WPF
- Git for version control

Guidelines:

- Start by designing the user interface using Windows Forms or WPF.
- Implement event handlers for user interactions (e.g., button clicks, form inputs).
- Use data binding to connect UI elements to data sources.
- Apply the MVVM pattern for better separation of concerns and maintainability.
- Implement multi-threading to ensure the UI remains responsive during long-running tasks.

- Test the application thoroughly for functionality and performance.
- Ensure the application handles exceptions and provides appropriate error messages to users.

14.7.2 Web Applications

Knowledge Required:

- Basics of web development (HTML, CSS, JavaScript)
- C# and ASP.NET Core for backend development
- Understanding of MVC (Model-View-Controller) architecture
- Working with databases (SQL Server, Entity Framework)
- Asynchronous programming with async and await

Tools and Technologies:

- Visual Studio or Visual Studio Code
- ASP.NET Core
- SQL Server or other relational databases
- Git for version control

Guidelines:

- Start by designing the database schema.
- Create the backend using ASP.NET Core and Entity Framework.
- Develop the frontend using HTML, CSS, and JavaScript.
- Integrate frontend and backend using MVC architecture.
- Implement asynchronous operations for improved performance.
- Test the application thoroughly before deployment.

14.7.3 Mobile Applications

Knowledge Required:

- Basics of mobile development (Xamarin for C#)
- Understanding of mobile UI/UX design
- Knowledge of RESTful services for backend integration
- Asynchronous programming for handling background tasks

Tools and Technologies:

- Visual Studio with Xamarin
- RESTful APIs
- SQLite for local data storage
- Git for version control

Guidelines:

- Design the mobile application’s UI/UX.
- Use Xamarin.Forms for cross-platform development.
- Integrate with RESTful APIs for backend data.

- Implement local data storage using SQLite.
- Use asynchronous programming to handle network calls.
- Test the application on multiple devices and screen sizes.

14.7.4 Gaming Applications

Knowledge Required:

- Basics of game development (Unity with C#)
- Understanding of game physics and animations
- Multi-threading for handling game loops and rendering

Tools and Technologies:

- Unity Engine
- Visual Studio or Visual Studio Code
- Git for version control

Guidelines:

- Start by designing the game concept and mechanics.
- Use Unity to develop the game environment and characters.
- Implement game logic and physics using C# scripts.
- Utilize multi-threading to optimize game performance.
- Test the game thoroughly for bugs and performance issues.

14.7.5 IoT Applications

Knowledge Required:

- Basics of IoT (Internet of Things) concepts
- Working with sensors and microcontrollers (e.g., Arduino, Raspberry Pi)
- Networking basics for IoT device communication
- Asynchronous programming for sensor data handling

Tools and Technologies:

- Arduino IDE or Raspberry Pi setup
- Visual Studio for C# development
- MQTT or HTTP for IoT communication
- Git for version control

Guidelines:

- Start by designing the IoT system architecture.
- Connect and configure sensors and microcontrollers.
- Develop the C# application to process and display sensor data.
- Implement communication protocols (MQTT/HTTP) for data transmission.
- Use asynchronous programming for real-time data handling.

- Test the IoT system in different environments and conditions.

14.7.6 Data Mining Applications

Knowledge Required:

- Basics of data mining and machine learning
- Understanding of data preprocessing and cleaning
- C# libraries for data mining (e.g., ML.NET)
- Multi-threading for handling large datasets

Tools and Technologies:

- Visual Studio with ML.NET
- SQL Server or other data storage solutions
- Git for version control

Guidelines:

- Start by collecting and preprocessing the dataset.
- Use ML.NET to develop data mining models.
- Implement the models in a C# application.
- Use multi-threading to process large datasets efficiently.
- Evaluate and validate the data mining results.
- Test the application with different datasets and scenarios.