**Task 08**

**Part01**

**Problem 01**

**Why is it better to code against an interface rather than a concrete class?**

When I code against an **interface** instead of a **concrete class**, I make the code more flexible and easier to change later. The interface only defines a contract without giving details, which means any class that implements the interface can replace another one without changing the main code.

This helps in:

* Reducing tight coupling between classes.
* Making it easier to extend the code (add new implementations without touching old code).
* Easier testing, because I can use mock objects.
* Following the Dependency Inversion principle from SOLID.

In short: coding against interfaces makes the code cleaner, more flexible, and easier to maintain.

**Problem 02**

**When should you prefer an abstract class over an interface?**

I prefer using an **abstract class** when there is some shared behavior between different classes, or when I need to define **fields and constructors** which interfaces don’t support.

For example, if I have Animal as an abstract class, I can put a method like Eat() that works the same for all animals, but leave MakeSound() abstract so each animal defines its own sound.

The main cases to use abstract classes are:

* There is common code between classes.
* There is a clear “is-a” relationship (like Dog is an Animal).
* I need to share data or state between classes.
* I want to provide a base constructor.

In short: I use an abstract class when I need **shared behavior + state**, and an interface when I only need a **contract for implementation**.

**Problem 03**

**How does implementing IComparable improve flexibility in sorting?**

When a class implements IComparable, it defines how its objects can be compared with others of the same type. This makes sorting easier and more flexible, because I don’t need to write external comparison logic.

The advantages are:

* I can sort objects the way I want (by name, ID, GPA, etc.).
* I can directly use built-in methods like List.Sort() or Array.Sort().
* The code becomes clearer and more reusable.

**Problem 04**

**What is the primary purpose of a copy constructor in C#?**

The main purpose of a copy constructor in C# is to create a **new object** that is a copy of an existing object. It allows me to duplicate the state (values of fields and properties) of one object into another, without directly referencing the same memory.

This is useful when:

* I want a new object with the same data, but independent from the original one.
* I need to avoid side effects that can happen if two objects share the same reference.
* I want to make code cleaner and safer when dealing with object duplication.

**Problem 05**

**How does explicit interface implementation help in resolving naming conflicts?**

Explicit interface implementation is useful when two or more interfaces have methods with the same name. If I implement them normally, there will be a conflict because the compiler won’t know which method belongs to which interface. By using explicit implementation, I can separate them and make it clear which method belongs to which interface.

For example:

interface IReadable{ void Read(); }

interface IWritable{ void Read(); }

class File : IReadable, IWritable{

void IReadable.Read()

{

Console.WriteLine("Reading from file...");

}

void IWritable.Read()

{

Console.WriteLine("Writing to file...");

}

}

**Problem 06**

**What is the key difference between encapsulation in structs and classes?**

The key difference is that **classes are reference types** while **structs are value types**. Both can use encapsulation (private fields with public properties or methods), but the way the data is handled in memory is different.

* In a **class**, encapsulated data is stored on the heap, and multiple variables can reference the same object.
* In a **struct**, encapsulated data is stored on the stack (or inline in memory), and every copy of a struct has its own separate copy of the data.

**what is abstraction as a guideline, what’s its relation with encapsulation?**

**Abstraction** is about focusing on the essential features of an object while hiding unnecessary details. As a guideline in programming, it means I should design classes and methods to show only what is important for the user, and leave out the complex implementation details. For example, when I use a List, I just call methods like Add() or Remove() without knowing how the list manages memory internally.

**Encapsulation**, on the other hand, is about hiding the internal state of an object and controlling access to it, usually by using private fields and exposing them through public methods or properties.

The relation between them is that **encapsulation is one way to achieve abstraction**. Encapsulation hides the internal data and implementation, while abstraction decides *which* details should be shown to the outside world and *which* should be hidden.

**Problem 07**

**How do default interface implementations affect backward compatibility in C#?**

Default interface implementations (introduced in C# 8) allow an interface to provide a **default body** for its methods. This means if I add a new method to an interface, I can give it a default implementation, and existing classes that already implement the interface will not break.

Before this feature, adding a new method to an interface forced all implementing classes to update and provide an implementation, which could cause a lot of code changes. With default implementations, old code keeps working, and only classes that want custom behavior need to override the method.

**Problem 08**

**How does constructor overloading improve class usability?**

Constructor overloading makes a class easier to use by allowing multiple ways to create an object with different sets of input data. Each constructor can take different parameters, so the user of the class can choose the most convenient option depending on the situation.

For example, one constructor might only need a single required value, while another constructor allows passing more details. This flexibility improves readability, reduces the need for setting properties manually after creating the object, and makes the class more intuitive to work with.

**Part02**

**Problem 02**

**What we mean by coding against interface rather than class ?**

Coding against an interface means that I write my code to depend on an **interface (a contract)** instead of depending directly on a specific class. The interface only defines *what* operations are available, not *how* they are implemented. This way, any class that implements the interface can be used without changing the main code.

For example, if I have an ILogger interface, my code will only call Log(), without caring if the actual implementation writes to a file, console, or database.

**What we mean by code against abstraction not concreteness ?**

This is the same idea but in a broader sense. Abstraction refers to **general definitions** like interfaces or abstract classes, while concreteness refers to **specific implementations**. By coding against abstraction, I make my code flexible, reusable, and easy to extend, because I can replace implementations without rewriting the logic that uses them.

Problem 03

**What is abstraction as a guideline and how we can implement this through what we have studied ?**

**Abstraction** as a guideline means focusing on the important features of an object and hiding the unnecessary details. The idea is to expose only what is needed for the user of the class, and hide the complex logic that works behind the scenes. This makes code easier to understand, use, and maintain.

From what we have studied, we can implement abstraction in two main ways:

1. **Abstract classes** → they allow us to define abstract methods (without implementation) and also provide shared code for all subclasses. For example, an Animal abstract class can have a method Eat() with common logic and an abstract method MakeSound() that each animal overrides.
2. **Interfaces** → they define contracts that classes must follow without providing implementation details. For example, an IShape interface can define CalculateArea() and Draw(), and each shape class (like Rectangle or Triangle) implements them in its own way.