Task Day08

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

Coding against an interface rather than a concrete class is better because it makes your code:

1. Flexible / Loosely Coupled – You can easily swap implementations (e.g., Car → Bike) without changing the code.

2. Testable – Interfaces can be mocked in unit tests, so you don't need real objects.

3. Polymorphic – Multiple classes implementing the same interface can be treated uniformly.

4. Separation of Concerns – The interface defines *what* a class should do, while the class defines *how*.

5. Maintainable – Adding or changing classes does not require rewriting existing code that depends on the interface.

Question: When should you prefer an abstract class over an interface?

You should prefer an abstract class when:

1. You Want to Share Common Code
   Abstract classes can have fields, constructors, and implemented methods.
   Example: a Shape class with a Display() method that all shapes share.
   Interfaces cannot (before C# 8.0) provide implementation.

2. You Need to Maintain State
   -Abstract classes can have fields or properties that store data for all derived classes.
   -Interfaces only define behavior, not data.

3. You Want to Control Inheritance
   -A class can inherit from only one base class (abstract or concrete) but can implement multiple interfaces.
   -If your design requires a single controlled hierarchy, an abstract class is

better.

4. You Expect Extensibility with Default Behavior
   Abstract classes let you provide default implementations that derived classes can override.
   Example: Shape.Display() can be the same for all shapes, while GetArea() is abstract.

5. You Need Constructors or Destructors
   Abstract classes can have constructors, which are not allowed in interfaces.
   Useful when initializing fields in the base class.

Question: How does implementing IComparable improve flexibility in sorting?

Implementing IComparable in a class allows objects of that class to define their natural ordering. This improves flexibility in sorting because:

1. Standardized Comparison: Each object knows how to compare itself with another (e.g., by Price).

2. Automatic Sorting: Collections like Array.Sort() or List<T>.Sort() can sort objects without extra code.

3. Multiple Criteria: You can easily change the comparison logic to sort by different properties (Price, Name, etc.).

4. Reusability: Any code or library that sorts objects can use the class directly without knowing its internal details.

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

The primary purpose of a copy constructor in C# is to create a new object as a copy of an existing object. It allows you to:

1. Create deep copies so the new object has its own independent data.

2. Simplify object duplication without manually copying each field.

3. Encapsulate copy logic for complex objects to ensure consistency.

Question: How does explicit interface implementation help in resolving naming conflicts?

Explicit interface implementation helps resolve naming conflicts by:

1. Separating interface methods from class methods – the interface method is only accessible when the object is referenced as the interface.

2. Allowing multiple interfaces with the same method names to be implemented in the same class without conflicts.

3. Controlling method calls – you can decide whether to call the class's own method or the interface method depending on the reference type.

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

The key difference between encapsulation in structs and classes is:

-Structs are value types: assigning or passing a struct creates a copy of the data. Changes to one copy do not affect the original.
-Classes are reference types: assigning or passing a class creates a reference to the same object. Changes through any reference affect the original object.

Question: what is abstraction as a guideline, what's its relation with encapsulation?

Abstraction is a design principle that focuses on **hiding unnecessary implementation details** and exposing only the **essential behavior** of an object. It allows developers to interact with objects based on **what they do**, not **how they do it**.

Relation to Encapsulation:

-Encapsulation hides the internal state and protects data using private fields and public methods/properties.
-**Abstraction** defines **what an object can do**, while encapsulation defines **how the data and implementation are kept hidden**.
-Encapsulation **supports abstraction** by keeping internal details private, allowing the interface or essential behavior to be exposed safely.

Question: How does constructor overloading improve class usability?

Constructor overloading improves class usability by:

1. Providing multiple ways to initialize objects depending on the available information.

2. Offering flexibility so users can choose the most convenient constructor.

3. Reducing code duplication by handling initialization internally.

4. Enhancing readability and maintainability, making the class easier and safer to use.

What do we mean by coding against interface rather than class ? and if you get it so What do we mean by code against abstraction not concreteness ?

Coding against an interface rather than a class means writing your code to depend on an interface (a contract) instead of a specific concrete class. This allows flexibility, polymorphism, and reduces tight coupling because any class implementing the interface can be used interchangeably.

Coding against abstraction, not concreteness, means writing code to depend on abstract types (interfaces or abstract classes) rather than concrete implementations. You rely on what an object can do, not how it does it, making the system more flexible, maintainable, and extensible

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

Abstraction as a guideline is a design principle that focuses on hiding unnecessary implementation details and exposing only the essential behavior of an object. It helps manage complexity, improves maintainability, and allows programmers to focus on what an object does, not how it does it.