Questions

Part 1: -

1. Why is it better to code against an interface rather than a concrete class?
   - **Increased Flexibility:** By coding against an interface, you decouple your code from specific implementations. This allows you to easily substitute different implementations without affecting the rest of your code.
   - **Improved Testability:** It's easier to write unit tests when you can mock or stub interface implementations.
   - **Enhanced Design:** It promotes loose coupling and the principle of Dependency Inversion.

2. When should you prefer an abstract class over an interface?
   - **Partial Implementation:** Use an abstract class when you want to provide a partial implementation of functionality that subclasses can inherit and extend.
   - **Multiple Inheritance:** Interfaces allow you to inherit from multiple sources, while abstract classes do not.

3. How does implementing IComparable improve flexibility in sorting?
   - **Custom Sorting:** Implementing `IComparable` allows you to define custom sorting logic for your objects. This enables you to sort collections based on specific criteria.
   - **Framework Integration:** Many framework classes, like `Array.Sort` and `List.Sort`, use `IComparable` to sort collections.

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

   Deep Copying: A copy constructor creates a new object that is a deep copy of an existing object. This ensures that changes to the original object do not affect the copy.

5. How does explicit interface implementation help in resolving naming conflicts?
   - **Naming Conflicts:** When a class implements multiple interfaces with conflicting member names, explicit implementation helps avoid naming collisions.
   - **Interface-Specific Behavior:** It allows you to define interface-specific behavior without affecting the public interface of the class.

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

   Value vs. Reference Types: Structs are value types, and their members are typically encapsulated within the struct itself. Classes are reference types, and encapsulation is achieved through access modifiers and the object's public interface.

7. what is abstraction as a guideline, what's its relation with encapsulation?

   - **Focus on Essentials:** Abstraction involves focusing on the essential features of an object or system while hiding unnecessary details.
   - **Relation to Encapsulation:** Encapsulation is a mechanism for achieving abstraction. By hiding implementation details and exposing only the necessary public interface, encapsulation promotes abstraction.

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

   Backward Compatibility: Default interface implementations can introduce new functionality to interfaces without breaking existing implementations. However, it's important to consider potential compatibility issues when using default implementations.

9. How does constructor overloading improve class usability?

   Flexibility: Constructor overloading allows you to create multiple constructors with different parameter lists, providing flexibility in object initialization.

Part 2: -

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

   - **Abstraction over Implementation:** This means writing code that interacts with objects through their interfaces rather than their concrete implementations. This promotes loose coupling and makes your code more flexible and testable.
   - **Example:** Instead of coding against a specific `ConcreteClass`, you would code against an `Interface` that `ConcreteClass` implements.

2. What is abstraction as a guideline and how we can implement this through what we have studied ?
    - **Focus on Essential Behavior:** Abstraction involves identifying the essential behaviors and properties of an object or system and hiding the underlying implementation details.
    - **Implementation:**
        - **Interfaces:** Define interfaces to specify the contracts that objects must adhere to.
        - **Abstract Classes:** Provide partial implementations of functionality and abstract methods that subclasses must implement.
        - **Encapsulation:** Hide implementation details within classes and expose only the necessary public interface.
        - **Polymorphism:** Use polymorphism to treat objects of different types as if they were instances of a common interface or base class.