1. Why is it better to code against an interface rather than a concrete class?
Coding against an interface promotes loose coupling and flexibility in software design. By relying on an abstraction rather than a specific implementation, the code becomes more adaptable to change, easier to test, and more maintainable. It allows multiple implementations to be substituted without altering the client code, supporting extensibility and adherence to SOLID principles, especially the Dependency Inversion Principle.

---

2. When should you prefer an abstract class over an interface?
An abstract class is preferred when there is a clear hierarchical relationship and a need to share common state or behavior among related classes. Unlike interfaces, abstract classes can include fields, constructors, and partial implementations that can be reused across subclasses. They are more suitable when you want to enforce a base functionality and avoid duplicating code across multiple classes that share a common ancestor.

---

3. How does implementing IComparable improve flexibility in sorting?
Implementing `IComparable` provides a standardized way to define the natural ordering of objects. This makes objects directly sortable using built-in algorithms without requiring external logic. It increases flexibility because different types can define their own comparison logic, allowing collections of these objects to be sorted consistently and efficiently according to the desired criteria.

---

4. What is the primary purpose of a copy constructor in C#?
The primary purpose of a copy constructor is to create a new object as a copy of an existing one while ensuring proper handling of its internal state. It is particularly useful to prevent unintended side effects of shallow copying, and to provide precise control over how the new object duplicates fields, ensuring independence when needed.

---

5. How does explicit interface implementation help in resolving naming conflicts?
Explicit interface implementation is useful when multiple interfaces declare members with the same signature. It allows a class to differentiate between these members by implementing them explicitly, avoiding ambiguity and ensuring that each interface's contract is honored correctly without interfering with the public API of the class.

---

6. What is the key difference between encapsulation in structs and classes?
Encapsulation in both structs and classes hides implementation details and provides controlled access through properties or methods. The key difference is that classes are reference types,

supporting inheritance and polymorphism, while structs are value types, more lightweight, and stored differently in memory. This distinction influences how encapsulation is applied and how objects behave in terms of copying and memory management.

---

7. What is abstraction as a guideline, what's its relation with encapsulation?
Abstraction focuses on exposing only the essential features of an object while hiding unnecessary details to reduce complexity. Encapsulation, on the other hand, is the mechanism of restricting direct access to the internal state of an object. Their relation is that abstraction defines *what* should be exposed, while encapsulation enforces *how* this is achieved in practice by controlling visibility and accessibility.

---

8. How do default interface implementations affect backward compatibility in C#?
Default interface implementations allow developers to add new methods to interfaces without breaking existing implementations. This significantly improves backward compatibility by ensuring that older classes that implement the interface remain functional even when the interface evolves. It reduces the need for creating new interfaces or modifying existing classes unnecessarily.

---

9. How does constructor overloading improve class usability?
Constructor overloading enhances usability by providing multiple ways to create and initialize objects depending on the available data or use case. It makes a class more versatile by supporting different initialization scenarios without requiring external configuration methods, thereby simplifying object creation for developers.

---

10. What do we mean by coding against interface rather than class?
Coding against an interface rather than a class means designing software to depend on abstractions instead of concrete implementations. This allows developers to write flexible code where the actual implementation can change or evolve without requiring changes in the client code, thereby promoting scalability and maintainability.

---

11. What do we mean by coding against abstraction, not concreteness?
Coding against abstraction rather than concreteness is the broader principle that emphasizes depending on high-level definitions of behavior rather than tightly coupling to specific implementations. It is a design philosophy that underpins object-oriented programming and ensures that systems are modular, extensible, and adaptable to change.

12. What is operator overloading?

Operator overloading is the ability to redefine the behavior of standard operators for user-defined types. It allows developers to make their classes or structs behave in intuitive and natural ways when using operators, aligning them more closely with the semantics of the type being represented.

```
public class Complex {
    public int Real { get; set; }
    public int Imag { get; set; }

    public static Complex operator +(Complex c1, Complex c2) {
        return new Complex { Real = c1.Real + c2.Real, Imag = c1.Imag + c2.Imag };
    }
}
```

```
Complex c1 = new Complex { Real = 1, Imag = 2 };
Complex c2 = new Complex { Real = 3, Imag = 4 };
Complex c3 = c1 + c2;
```