Q1: Why is it better to code against an interface rather than a concrete class?
It provides flexibility, allows dependency inversion, makes code loosely coupled, and makes switching implementations easier without changing client code.

Q2: When should you prefer an abstract class over an interface?
Use an abstract class when you need to share **common code or state** among related classes, but use an interface when you only want to enforce a **contract** without implementation.

Q3: How does implementing IComparable improve flexibility in sorting?
It allows objects to define their own **custom comparison logic**, enabling easy sorting using built-in methods like Array.Sort() or List.Sort().

Q4: What is the primary purpose of a copy constructor in C#?
To create a **new independent copy** of an object with the same values, avoiding unintended reference sharing (deep copy).

Q5: How does explicit interface implementation help in resolving naming conflicts?
It lets a class provide a **separate implementation** for interface methods when the method names clash with class methods, avoiding ambiguity.

Q6: What is the key difference between encapsulation in structs and classes?
Structs are **value types** (copied on assignment), while classes are **reference types** (shared by reference). Encapsulation works the same, but behavior differs due to value vs reference semantics.

Q7: What is abstraction as a guideline, and what's its relation with encapsulation?
Abstraction hides **implementation details** and exposes only essential features; encapsulation hides **internal data** using access modifiers. They are related: abstraction is the **design principle**, encapsulation is the **implementation mechanism**.

Q8: How does constructor overloading improve class usability?
It gives flexibility to create objects in **multiple ways** depending on available data, improving ease of use and readability.