باقي اسئلة D7

1. **Why can't you create an instance of an interface directly?**
   Because an interface does not provide any actual implementation — it only defines a set of methods and properties that a class must implement.
   It's like a "blueprint" or "contract" without real behavior. You need a class that implements the interface to create an object.

---

2. **What are the benefits of default implementations in interfaces (C# 8.0)?**
   Default interface methods allow adding new functionality to an interface without breaking existing code.

- They provide backward compatibility (old classes still work).
- They reduce code duplication (shared logic inside the interface).
- They make interfaces more flexible, closer to abstract classes, but still support multiple inheritance.

---

3. **Why is it useful to use an interface reference to access implementing class methods?**
   Using an interface reference promotes **polymorphism**: different classes can be accessed through the same interface type.

- This leads to loose coupling (code depends on interface, not concrete class).
- Easier to change or replace classes without affecting client code.
- Great for testing and mocking (you can pass fake implementations).

---

4. **How does C# overcome the limitation of single inheritance with interfaces?**
   C# does not support multiple inheritance of classes (a class can inherit only one base class).
   But a class **can implement multiple interfaces**.
   This allows a class to combine behaviors from many sources (e.g., IReadable, IWritable, IMovable).
   So interfaces provide flexibility and achieve multiple inheritance of behavior safely, avoiding the

5. **What is the difference between a virtual method and an abstract method in C#?**
    1. A **virtual method** has a default implementation but can be overridden.
    2. An **abstract method** has no implementation and must be overridden in derived classes.

D8 : Part01

1. **Why is it better to code against an interface rather than a concrete class?**
   Because coding against an interface provides flexibility, loose coupling, and polymorphism. You can switch implementations without changing the code that uses the interface.

---

2. **When should you prefer an abstract class over an interface?**
   Use an abstract class when you want to share common implementation/fields across subclasses, not just method signatures. Interfaces are better when you only need a contract, without shared code.

---

3. **How does implementing IComparable improve flexibility in sorting?**
   It allows objects to be sorted directly with built-in sorting algorithms. You can define custom comparison logic (e.g., by price, by name) without modifying external code.

---

4. **What is the primary purpose of a copy constructor in C#?**
   To create a new object as a copy of an existing one, ensuring deep copy of fields so that objects don't unintentionally share references.

---

5. **How does explicit interface implementation help in resolving naming conflicts?**
   It allows a class to implement methods from multiple interfaces that have the same method names, without conflicts. Each interface's method can be handled separately.

---

6. **What is the key difference between encapsulation in structs and classes?**
   Both support encapsulation, but **structs are value types** (copied by value) while **classes are reference types** (copied by reference). This affects memory behavior and performance.

---

7. **What is abstraction as a guideline, and what's its relation with encapsulation?**
   Abstraction hides *what* the object does (focus on behavior, not implementation).
   Encapsulation hides *how* it does it (internal data, implementation details).
   They work together: abstraction defines the design, encapsulation secures the implementation.

---

8. **How do default interface implementations affect backward compatibility in C#?**
   They allow adding new methods to interfaces without breaking existing implementations. Old classes still compile even if they don't override the new method.

---

9. **How does constructor overloading improve class usability?**
   It gives flexibility when creating objects. Users can initialize an object with different sets of data (full details or partial), improving readability and convenience.

Part02+Bonus

1️⃣ What do we mean by coding against interface rather than class?

It means: when writing code, depend on the **interface (contract)** instead of the actual class.

🔷 Example:
Instead of coding directly to a concrete class like SqlDatabase, you depend on IDatabase.
This way, if tomorrow you switch from SQL to MongoDB, your main code won't break — only the implementation changes.

👉 Idea: Focus on **what the object can do** (behavior) instead of **how it does it** (implementation).

2️⃣ What do we mean by code against abstraction, not concreteness?

This is a general principle:

- **Abstraction** = interface or abstract class.
- **Concreteness** = actual implementation class.

So, "code against abstraction" means always design your systems to rely on the **general contract** and not the specific details.

3️⃣ What is abstraction as a guideline?

As a guideline, **abstraction** means:

- Hide unnecessary details.
- Focus on essential behavior.
- Provide a clear contract that other classes must follow.

How do we implement this?

From what we've studied:

- Using **Interfaces** → define only contracts.
- Using **Abstract Classes** → combine contracts + shared logic.
- Using **Polymorphism** → work with objects through their abstraction.

4️⃣ What is operator overloading?

Operator overloading allows us to redefine how operators (+, -, *, ==, etc.) work for our custom classes.

🔷 Example:
If you create a ComplexNumber class, you can overload the + operator:

ComplexNumber c1 = new ComplexNumber(1, 2);

ComplexNumber c2 = new ComplexNumber(3, 4);

ComplexNumber result = c1 + c2; // works because we overloaded +

Without operator overloading, you would need something like c1.Add(c2) instead.