1-Why can't you create an instance of an interface directly?

- An **interface** is only a **contract** (it defines methods/properties but no implementation).
- Because it has **no real code**, you **cannot create an object** from an interface directly.
- To use it, a **class must implement** the interface and provide the actual code.

2-What are the benefits of default implementations in interfaces introduced in C# 8.0?

- Default implementations in interfaces (C# 8.0) provide:

    1. **Backward compatibility** → add new methods without breaking old code.
    2. **Code reuse** → common logic can live in the interface.
    3. **Cleaner design** → interfaces can carry default behavior.
    4. **Easier API evolution** → libraries can grow without forcing rewrites.
    5. **Less boilerplate** → classes only override what's unique.

3-Why is it useful to use an interface reference to access implementing class methods?

1. **Polymorphism** → one reference type can work with many classes.
2. **Decoupling** → code depends on abstractions, not concrete classes.

3. **Extensibility** → new classes can be added without changing existing code.
4. **Testability** → easy to mock interfaces for unit testing.

4-How does C# overcome the limitation of single inheritance with interfaces?

C# only allows **single inheritance** (one base class), but it overcomes this limitation by allowing a class to **implement multiple interfaces**.

- This provides the benefits of multiple inheritance (multiple behaviors) without its problems.
- Example: A class can implement both IMovable and IFlyable to combine behaviors.

5-What is the difference between a virtual method and an abstract method in C#?

- **Virtual method** → has a default implementation; derived classes may override it but are not required to.
- **Abstract method** → has no implementation; derived classes **must** override and implement it.
- Abstract methods can only exist in **abstract classes**.

6-What is the difference between class and struct in C#?

- **Class** → Reference type, stored on the heap, supports inheritance, copying passes references (changes affect both).
- **Struct** → Value type, stored on the stack, no inheritance (only interfaces), copying creates independent copies.

- Classes are used for **complex objects**, structs for **lightweight data containers**.

7-If inheritance is relation between classes clarify other relations between classes?

Relations between classes in C#:

1. **Inheritance (is-a)** → one class derives from another.
2. **Association** → general link between two classes.
3. **Aggregation (has-a)** → whole-part relationship; parts can exist independently.
4. **Composition (strong has-a)** → whole-part relationship; parts cannot exist without the whole.
5. **Dependency (uses)** → one class depends on another for some functionality.