C# day 8

Part 01 Questions

Q1: Why is it better to code against an interface rather than a concrete class?

Coding against interfaces provides several key benefits:

- **Flexibility:** You can easily swap implementations without changing client code. If you code against `IVehicle`, you can switch between `Car` and `Bike` seamlessly.
- **Loose Coupling:** Code depends on abstractions, not concrete implementations, reducing dependencies between components.
- **Testability:** Easy to create mock objects for unit testing without affecting production code.
- **Extensibility:** New implementations can be added without modifying existing code, following the Open/Closed Principle.

Q2: When should you prefer an abstract class over an interface?

Prefer abstract classes when:

- You need to provide **common implementation** (non-abstract methods) that all derived classes can share.
- You want to define **fields or constructors** - interfaces cannot have these.
- You need **access modifiers** (protected, private) for members.
- There's an **"is-a" relationship** where derived classes share core behavior.

Use interfaces when you only need to define a contract without implementation, or when a class needs to implement multiple contracts (C# supports multiple interface implementation but not multiple inheritance).

Q3: How does implementing IComparable improve flexibility in sorting?

Implementing `IComparable<T>` provides:

- **Built-in sorting support:** Objects can be sorted using `Array.Sort()` and `List.Sort()` without custom comparers.
- **Consistent comparison logic:** The comparison rule is defined once in the class itself.
- **Reusability:** The same comparison logic works across different collections and algorithms.
- **Type safety:** Generic implementation ensures compile-time type checking.

Q4: What is the primary purpose of a copy constructor in C#?

The copy constructor creates a **deep copy** of an object. Its primary purposes are:

- **Independence:** Create a new object that is independent from the original, so changes to one don't affect the other.
- **Avoid shallow copy issues:** When copying reference types, a shallow copy only copies references, causing both objects to share the same data.
- **Data protection:** Preserve the original object's state when creating duplicates for manipulation.

Example: When `Student` contains a `Grade` object, a copy constructor ensures the new `Student` gets its own separate `Grade` instance.

Q5: How does explicit interface implementation help in resolving naming conflicts?

Explicit interface implementation allows:

- **Method disambiguation:** When a class implements multiple interfaces with same method names, you can provide different implementations for each.
- **Hiding methods:** The explicitly implemented method is only accessible through the interface reference, not through the class instance.
- **Separation of concerns:** Provide interface-specific behavior separate from the class's own public methods.

Example: `Robot` can have both `public void Walk()` and `void IWalkable.Walk()` with different implementations.

Q6: What is the key difference between encapsulation in structs and classes?

Key differences:

- **Value vs Reference:** Structs are value types (stored on stack), classes are reference types (stored on heap).
- **Copying behavior:** Assigning a struct creates a complete copy; assigning a class copies only the reference.
- **Immutability preference:** Structs are often designed to be immutable for safety, while classes can be mutable.
- **No parameterless constructor:** Structs cannot have explicit parameterless constructors (before C# 10).

Both support encapsulation through properties and access modifiers, but their memory behavior differs significantly.

Q7: What is abstraction as a guideline? What's its relation with encapsulation?

Abstraction is hiding complex implementation details and showing only essential features. It answers "what" an object does.

Encapsulation is bundling data and methods together while restricting direct access. It answers "how" to protect data.

Relationship:

- Abstraction uses encapsulation to hide implementation.
- Encapsulation is the mechanism; abstraction is the design principle.
- Example: A `Shape` class (abstraction) hides area calculation logic behind public methods, while private fields (encapsulation) protect internal data.

Q8: How do default interface implementations affect backward compatibility in C#?

Default interface implementations (C# 8.0+) improve backward compatibility by:

- **Non-breaking changes:** Adding new methods to interfaces doesn't break existing implementations - they automatically use the default.
- **Optional overriding:** Classes can choose to override the default or use it as-is.
- **Evolution support:** Interfaces can evolve over time without forcing all implementations to update immediately.
- **Migration flexibility:** Legacy code continues working while new code can leverage new features.

Q9: How does constructor overloading improve class usability?

Constructor overloading provides:

- **Flexibility:** Users can create objects with different initialization options based on available data.
- **Default values:** Provide sensible defaults when partial information is given.
- **Convenience:** Reduce code duplication by offering multiple ways to instantiate objects.
- **Clear intent:** Different constructors communicate different use cases clearly.

Part 02 Questions

Q2: What do we mean by coding against interface rather than class? Code against abstraction not concreteness?

Coding against interface rather than class:

This means depending on interface types instead of concrete class types. Benefits include:

- Your code works with any implementation of the interface
- Easy to substitute different implementations without code changes
- Example: Use `IShapeSeries` instead of `SquareSeries` - your `PrintTenShapes()` method works with any series type

Code against abstraction not concreteness:

This extends the principle to abstract classes and general abstractions:

- Depend on abstract types (`GeometricShape`) rather than concrete types (`Rectangle`)
- Use high-level contracts instead of implementation details
- Focus on "what" not "how" - define behavior contracts, not specific implementations
- Enables polymorphism: `GeometricShape shape = ShapeFactory.CreateShape("rectangle", 5, 4)`

Q3: What is abstraction as a guideline and how can we implement it through what we have studied?

Abstraction as a guideline means designing systems by:

1. **Hiding complexity:** Show only what's necessary, hide implementation details
2. **Focusing on behavior:** Define what objects do, not how they do it
3. **Creating contracts:** Establish interfaces that guarantee behavior

Implementation through our studies:

- **Interfaces** (`IVehicle`, `IShapeSeries`): Define pure contracts without implementation
- **Abstract classes** (`Shape`, `GeometricShape`): Provide partial implementation with mandatory abstract members
- **Encapsulation**: Use properties and access modifiers to hide internal state
- **Polymorphism**: Work with base types, let runtime determine actual behavior
- **Factory Pattern**: Abstract object creation - client doesn't know concrete types being created