# Self-Study

1. What Access Modifiers are allowed in an interface and why?

### Access Modifiers for Interfaces

1. **For the Interface Declaration**:
    - In C# (prior to C# 8.0), interfaces can have the following access modifiers:
        - public: The interface is accessible from any code that can reference the assembly.
        - internal: The interface is accessible only within the same assembly.
        - No access modifier (implicitly internal): If no modifier is specified, the interface defaults to internal.}

    **Note**: Interfaces cannot be private, protected, or protected internal because these modifiers restrict access in ways that don't align with the purpose of an interface, which is to define a widely accessible contract.

2. **For Interface Members (Prior to C# 8.0)**:
    - Interface members (methods, properties, events, or indexers) **do not allow explicit access modifiers**. They are **implicitly public** because interfaces define a contract that implementing types must expose to consumers.

3. **C# 8.0 and Later (Default Interface Methods)**:
    - Starting with C# 8.0, interfaces can include **default implementations** for members, and these members can have explicit access modifiers, including:
        - public (default, explicitly allowed).

        - private (for private helper methods within the interface).
        - protected or internal (less common, but allowed in specific scenarios).

      **Note**: Default implementations are useful for adding functionality to interfaces without breaking existing implementations, but they are still part of the public contract unless marked private.

**Why These Access Modifiers Are Allowed?**

The restrictions and allowances for access modifiers in interfaces are driven by their purpose in OOP, particularly their role in **abstraction**.

1. **Interface Declaration Access Modifiers (public or internal)**:
    - **Purpose of Interfaces**: Interfaces define a contract that classes or structs implement to ensure consistent behavior across different parts of a system. They are meant to be shared and used by multiple components, so their accessibility should be broad.
    - **Why public or internal?**
        - public allows the interface to be used across assemblies, supporting abstraction in large systems where different modules need to communicate through a common contract.
        - internal restricts the interface to the same assembly, which is useful for internal abstractions within a library or application.
        - **Why not private or protected?**
            - private would make the interface inaccessible outside its containing type, which contradicts the

purpose of defining a contract for other types to implement.

        - protected is relevant for inheritance hierarchies, but interfaces are not inherited in the same way classes are; they are implemented, so protected doesn't make sense.
        - protected internal is similarly incompatible because interfaces are not about inheritance-based access but about defining a public or assembly-wide contract.

2. **Interface Members (Implicitly public Prior to C# 8.0)**:
    - ***W*hy implicitly public?**
        - Interface members represent the contract that implementing types must fulfill. For the contract to be meaningful, these members must be accessible to any code that uses the interface. Making them public ensures that implementers expose these members to consumers, maintaining the abstraction.
        - For example, in our earlier ILogger example, the Log method must be callable by any code that uses an ILogger instance, so it's implicitly public.
    - **Why no other modifiers (like private or internal)?**
        - Non-public modifiers would allow implementing classes to hide parts of the contract, breaking the guarantee that the interface's behavior is fully accessible. This would undermine abstraction, as consumers wouldn't be able to rely on the contract being consistently implemented.

        - For instance, if Log were private, an implementing class like ConsoleLogger could hide it, making the ILogger contract useless to consumers.

3. **C# 8.0 and Later (Explicit Modifiers for Default Implementations)**:
    - ***Why allow public, private, etc., for default implementations?***
        - Default implementations (introduced in C# 8.0) allow interfaces to provide concrete behavior, enhancing flexibility without breaking existing implementations. For example, you can add a new method with a default implementation to an existing interface.
        - public is allowed because default implementations are part of the contract and should be accessible to consumers.
        - private is allowed for helper methods that support default implementations but aren't part of the public contract. This aligns with encapsulation (a component of abstraction) by hiding internal details.
        - protected or internal can be used in rare cases to restrict access to derived types or the same assembly, but these are less common.
    - ***Why still restrict abstract members?***
        - Members without default implementations (abstract members) remain implicitly public because they define the core contract that implementers must expose. Allowing private or internal for these would break the contract's accessibility.

## 2. How is an interface memory allocated ?

### Key Points on Interface Memory Allocation

1. **Interfaces Are Reference Types**:
    - Interfaces are stored in assembly metadata when the program loads, consuming minimal memory for type information (not per instance).
    - An interface variable is a reference (4 bytes on 32-bit, 8 bytes on 64-bit systems) that points to a concrete object on the heap.
2. **Memory for Concrete Objects**:
    - When you instantiate a class implementing an interface , memory is allocated on the **managed heap** for:
        - **Object header**: Includes a pointer to the type's method table (vtable) and a sync block for thread synchronization (~8-16 bytes).
        - **Instance fields**: Data defined in the class (e.g., a string field in ConsoleLogger).
        - **Method table**: Contains pointers to method implementations, including those for the interface. This is shared across all instances of the class and created when the type is loaded.
3. **Interface References**:
    - Assigning a concrete object to an interface variable (e.g., ILogger logger = new ConsoleLogger()) stores the object's memory address in the logger reference.
    - No additional memory is allocated for the interface itself; it's a view of the same object, using the method table to resolve interface methods.

4. **Virtual Dispatch**:
    - Method calls on an interface (e.g., logger.Log("message")) use the concrete object's method table to find the implementation (dynamic dispatch).
    - This incurs a small runtime overhead (~a few CPU cycles) but no additional memory allocation.
5. **Boxing with Value Types**:
    - If a value type (e.g., a struct) implements an interface, assigning it to an interface variable triggers **boxing**:
        - A copy of the value type is wrapped in an object on the heap.
        - Memory allocated includes the value type's data, object header, and method table reference.
        - Boxing increases memory usage and can impact performance, so use generic interfaces (e.g., IList<T>) to avoid it.
6. **Multiple Interfaces**:
    - A class implementing multiple interfaces (e.g., ILogger and IWriter) stores all method pointers in its single method table.
    - Interface references (e.g., ILogger logger and IWriter writer) point to the same object, with no extra memory per interface.
7. **Garbage Collection**:
    - The concrete object (e.g., ConsoleLogger) is eligible for garbage collection when no references (including interface references) point to it.
    - The interface variable itself does not affect garbage collection; only the underlying object's references matter.

### Example with Memory Breakdown

```
public interface ILogger
{
   void Log(string message);
}

public class ConsoleLogger : ILogger
{
   private string _prefix = "Console: ";
   public void Log(string message) => Console.WriteLine(_prefix + message);
}

class Program
{
   static void Main()
   {
       ILogger logger = new ConsoleLogger();
       logger.Log("Hello!");
   }
}
```

**Memory Allocation**:

- **On new ConsoleLogger()**:
  - Heap allocates ~24 bytes (object header + sync block) + 8 bytes (string reference for _prefix) + ~20 bytes (string "Console: ").
  - Method table (shared per type) includes pointers to Log and other methods, created when ConsoleLogger is loaded.
- **On ILogger logger = ...**:
  - logger is an 8-byte reference (64-bit system) pointing to the ConsoleLogger object.
- **On logger.Log("Hello!")**:
  - String "Hello!" allocates ~20 bytes on the heap.

    - CLR uses the method table to call ConsoleLogger.Log, with no additional memory for the interface.
- **Total memory** (approximate): ~60 bytes for the object, string, and references, excluding method table (type-level).

**Performance and Optimization Notes**

- ***V*irtual Dispatch Overhead**: Interface method calls involve a lookup in the method table, adding minimal CPU cycles (negligible for most applications).
- ***A*void Boxing**: Use generic interfaces (e.g., IEnumerable<T>) for value types to prevent boxing-related memory and performance costs.
- **Memory Efficiency**: Interfaces add no per-instance memory overhead beyond the reference itself, making them efficient for abstraction.
- **Type Metadata**: Interface and class metadata (method tables) are stored once per type in the CLR, not per instance, minimizing memory impact.

## 3. *W*hat is Dependency Inversion ?

**What is the Dependency Inversion Principle (DIP)? ✅**

The Dependency Inversion Principle (DIP) states that high-level modules should not depend directly on low-level modules; both should depend on abstractions (e.g., interfaces or abstract classes).

*A*dditionally:

*A*bstractions should not depend on details; details (concrete implementations) should depend on abstractions.

**In simple terms**:

✅ High-level logic (e.g., OrderService) shouldn't know the specifics of the implementation (e.g., MySQLRepository or FileLogger). It only interacts with an interface, allowing the implementation to change freely.

**Example**:

You're designing a robot that cuts pizza, called PizzaCutterRobot.

Initially, the robot uses a specific arm, PizzaCutterArm, to cut the pizza.

But you realize you might not always use the same arm—sometimes you may want to use a regular knife instead.

This is where the **Dependency Inversion Principle (DIP)** comes into play.

**Code that breaks DIP (direct dependency on details)**:

```
public class PizzaCutterRobot
{
   private readonly PizzaCutterArm _pizzaCutter = new PizzaCutterArm();
   public void CutPizza()
   {
       _pizzaCutter.Cut();
   }
}

public class PizzaCutterArm
{
   public void Cut()
   {
       Console.WriteLine("Cutting pizza with pizza cutter arm");
   }
}
```

❌ **The problem**:

PizzaCutterRobot directly depends on PizzaCutterArm.

If you want to use a different tool (e.g., Knife), you'd have to modify the PizzaCutterRobot class itself!

This breaks DIP because the high-level module (the robot) depends on a low-level module (the cutter arm).

✅ **The solution using DIP**:

1. Define an abstraction (interface):

```
public interface ICuttingTool
{
   void Cut();
}
```

2. Implement the interface in different tools:

```
public class PizzaCutterArm : ICuttingTool
{
   public void Cut()
   {
       Console.WriteLine("Cutting pizza with pizza cutter arm");
   }
}

public class Knife : ICuttingTool
{
   public void Cut()
   {
      Console.WriteLine("Cutting pizza with knife");
   }
}
```

3. Make PizzaCutterRobot depend on ICuttingTool instead of a specific tool:

```
public class PizzaCutterRobot
{
   private readonly ICuttingTool _tool;
   public PizzaCutterRobot(ICuttingTool tool)
   {
       _tool = tool;
   }

   public void CutPizza()
   {
```

```
        _tool.Cut();
    }
}
```

✅ **Main code**:

```
class Program
{
    static void Main()
    {
        var robot1 = new PizzaCutterRobot(new PizzaCutterArm());
        robot1.CutPizza(); // Output: Cutting pizza with pizza cutter arm
        var robot2 = new PizzaCutterRobot(new Knife());
        robot2.CutPizza(); // Output: Cutting pizza with knife
    }
}
```

## 4. What is Dependency Inversion ?

In the Specification design pattern, the core interface, often named ISpecification<T>, defines a contract for a reusable object that encapsulates a business rule or condition, returning a boolean through an IsSatisfiedBy (or similar) method to determine if an object meets the criteria. This interface allows for modularity, enabling the combination of multiple specifications using boolean logic (AND, OR, NOT) to create complex filtering and matching rules, thereby separating business logic from domain objects.

**Purpose of the Interface (ISpecification<T>)**

**Encapsulates Business Rules:**
The interface serves as a template for creating objects that represent specific business rules or conditions that an entity must meet.

**Defines Reusability:**
By creating an interface, concrete specification classes can be built to be reused across different parts of an application.

**Promotes Separation of Concerns:**
It separates the logic for defining criteria from the code that uses those criteria, making the system more maintainable and flexible.

Key Method: IsSatisfiedBy

**Checks Criteria:**
The IsSatisfiedBy(T candidate) method is fundamental to the ISpecification<T> interface.

**Returns Boolean:**
It evaluates whether a given candidate object satisfies the specific criteria defined by the implementing class, returning true or false.

**How the Interface is Used**

**1. Define the Interface:**
Create a generic ISpecification<T> interface with the IsSatisfiedBy method.

**2. Implement Concrete Specifications:**
Create classes that implement the ISpecification<T> interface to define specific rules, such as filtering by a price range or checking an employee's status.

**3. Combine Specifications:**
The ISpecification<T> interface can define methods for logical operations like AND, OR, and NOT, allowing new specifications to be built by combining existing ones.

**4. Apply Specifications:**
The combined specifications are then applied to filter data or validate objects within a repository or other part of the application, making the filtering logic cleaner and more manageable.


public interface ISpecification<T>
{
bool IsSatisfiedBy(T entity);
}
public class DiscountSpecification : ISpecification<Customer>
{
public bool IsSatisfiedBy(Customer customer)
{
// Check if the customer is eligible for a discount
return customer.TotalPurchases >= 1000 && customer.IsPremium;
}
}