Interface in Java

An interface in Java is a blueprint of a class that contains only abstract methods (before Java 8) and static or default methods (from Java 8 onwards). It is used to achieve abstraction and multiple inheritance in Java.

Why Use Interfaces in Java?

- **Achieves Abstraction:** Interfaces allow defining a contract for classes without providing implementation.
- **Supports Multiple Inheritance**: Unlike classes, a Java class can implement multiple interfaces, overcoming Java's lack of multiple inheritance.
- **Loose Coupling:** It helps achieve a low dependency between components, making code easier to maintain and test.
- **Polymorphism:** Interfaces allow different classes to be treated as the same type, promoting dynamic behavior.

Some Concepts / Rules for Interfaces in Java

1. All methods in an interface are public and abstract by default (before Java 8).
2. Interfaces cannot have instance variables, only constants (public, static, and final by default).
3. A class must use the implements keyword to use an interface.
4. From Java 8 onwards, interfaces can have:
    a. Default methods (with implementation).
    b. Static methods (with implementation).

5. From Java 9 onwards, interfaces can have private methods.

Note: The Java compiler adds public and abstract keywords before the interface method. and, it adds public, static and final keywords before data members.

Coupling in Java

In software design, coupling refers to the degree of dependency between components (classes, modules, or services).

Tight Coupling (High Dependency)

- One class directly depends on another class's implementation.
- If we change one class, it may require modifications in the dependent class.
- Not flexible → Difficult to maintain and test.

Example:

```
class Engine {
    void start() {
        System.out.println("Engine starting...");
    }
}

// Car class depends directly on Engine class
class Car {
    private Engine engine = new Engine(); // Direct dependency

    void startCar() {
        engine.start();
        System.out.println("Car started.");
```

```
    }
}
public class Main {
    public static void main(String[] args) {
        Car car = new Car();
        car.startCar();
    }
}
```

Problems with Tight Coupling

- The Car class is tightly dependent on the Engine class.
- If we need to change the Engine class (e.g., introduce ElectricEngine), we must modify the Car class.
- Difficult to replace or extend functionality without modifying the core structure.

Loose Coupling (Low Dependency)

- One class depends on an abstraction (interface) rather than a specific implementation.
- The classes can be changed independently without affecting each other.
- More flexible and easier to maintain.

Example:

```
// Step 1: Define an Interface (Abstraction)
interface Engine {
    void start();
}
// Step 2: Implement Different Engine Types
```

```
class PetrolEngine implements Engine {
    @Override
    public void start() {
        System.out.println("Petrol Engine starting...");
    }
}
class ElectricEngine implements Engine {
    @Override
    public void start() {
        System.out.println("Electric Engine starting...");
    }
}

// Step 3: Use Dependency Injection (Injecting Engine into Car)
class Car {
    private Engine engine;

    // Constructor injection
    public Car(Engine engine) {
        this.engine = engine;
    }

    void startCar() {
        engine.start();
        System.out.println("Car started.");
    }
}

public class Main {
```

```
    public static void main(String[] args) {
        // Using Petrol Engine
        Engine petrolEngine = new PetrolEngine();
        Car petrolCar = new Car(petrolEngine);
        petrolCar.startCar();

        // Using Electric Engine
        Engine electricEngine = new ElectricEngine();
        Car electricCar = new Car(electricEngine);
        electricCar.startCar();
    }
}
```

Advantages of Loose Coupling

- The Car class does not depend on a specific engine implementation; it depends on the Engine interface.
- We can easily replace the PetrolEngine with ElectricEngine without modifying the Car class.
- More flexible → New engine types can be added without changing existing code.
- Better for testing → We can mock the Engine interface in unit tests.

Question: Does Abstraction Always Lead to Loose Coupling?

Tagged / Marker Interface

An interface that does not contain any methods but is used to mark a class with some special characteristic or capability.

Key Characteristics:

- No methods are defined in the interface.
- It serves only as a tag.
- The actual behavior is handled by the system or framework based on the presence of the marker.

How it Works:

No methods = No behavior in the interface: The marker interface itself doesn't define any behavior. Its job is just to mark the class.

When a class implements a marker interface, the runtime environment or other parts of the framework will check whether a class implements the marker interface and perform actions based on that.

As developers, we understand its role through documentation and context in which the marker is used.

Example:

Serializable is a marker interface in Java. Classes implementing it are marked as serializable, and frameworks can treat them accordingly. Java's I/O system recognizes this and allows objects of those classes to be serialized (converted into a byte stream).