Interface

1. What is an interface in Java .

Ans1. In Java, an **interface** is a reference type that defines a contract for classes to follow. It contains **abstract methods** (methods without implementations) that any class implementing the interface must define. Interfaces help achieve **abstraction** and **multiple inheritance** in Java.

Key Features of Interfaces:

1. **Abstract Methods** – Methods in an interface do not have a body (until Java 8). Classes implementing the interface must provide implementations.
2. **Constants** – Variables declared in an interface are implicitly `public`, `static`, and `final`.
3. **Multiple Inheritance** – A class can implement multiple interfaces, unlike class inheritance, where a class can extend only one class.
4. **Default and Static Methods (Java 8+)** – Interfaces can have `default` methods (with a body) and `static` methods.
5. **Functional Interfaces (Java 8+)** – An interface with exactly **one** abstract method is called a **functional interface** (used in lambda expressions).

Syntax Example

```
// Defining an Interface

interface Animal {

    void makeSound(); // Abstract method (no body)
```

```
}

// Implementing the Interface
class Dog implements Animal {
    public void makeSound() { // Must define the method
        System.out.println("Woof!");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal myDog = new Dog(); // Interface reference
        myDog.makeSound(); // Outputs: Woof!
```

```
    }
}
```

Multiple Inheritance Example

```
interface Flyable {
    void fly();
}

interface Swimmable {
    void swim();
}

class Duck implements Flyable, Swimmable {
    public void fly() {
```

```
        System.out.println("Duck is flying");
    }

    public void swim() {
        System.out.println("Duck is swimming");
    }
}

public class Main {
    public static void main(String[] args) {
        Duck duck = new Duck();
        duck.fly();  // Outputs: Duck is flying
        duck.swim(); // Outputs: Duck is swimming
```

```
    }
}
```

Default and Static Methods (Java 8+)

```
interface Vehicle {
    void start();

    // Default method with implementation
    default void stop() {
        System.out.println("Vehicle is stopping");
    }

    // Static method
    static void fuelType() {
```

```
        System.out.println("Uses petrol or diesel");
    }
}

class Car implements Vehicle {
    public void start() {
        System.out.println("Car is starting");
    }
}

public class Main {
    public static void main(String[] args) {
        Car myCar = new Car();
```

```
        myCar.start();

        myCar.stop();         // Calls default method

        Vehicle.fuelType();   // Calls static method

    }

}
```

2. Which modifiers are allowed for methods in an Interface?Explain with an example

Ans2. In Java, the allowed modifiers for methods in an **interface** depend on the Java version:

1. Before Java 8 (Java 7 and earlier)

- **Only `public` and `abstract`** are allowed.
- All methods in an interface are **implicitly `public` and `abstract`**, so these modifiers can be omitted.

```
interface Animal {

    void makeSound(); // Implicitly public and abstract

}
```

2. Java 8 (Introduced `default` and `static` methods)

- **`default` methods**: Methods with a body (implementation).
- **`static` methods**: Can be called using the interface name.

```
interface Vehicle {

    void start(); // Implicitly public and abstract



    default void stop() { // Default method (has a body)

        System.out.println("Vehicle is stopping.");

    }



    static void fuelType() { // Static method

        System.out.println("Uses petrol or diesel.");

    }

}



class Car implements Vehicle {
```

```
    public void start() {
        System.out.println("Car is starting.");
    }
}

public class Main {
    public static void main(String[] args) {
        Car myCar = new Car();
        myCar.start();
        myCar.stop();        // Calls default method
        Vehicle.fuelType();  // Calls static method
    }
}
```

3. Java 9+ (Introduced `private` methods)

- `private` **methods**: Used inside the interface for code reuse.
- `private static` **methods**: Used within the interface for helper functions.

```
interface Calculator {

    default void add(int a, int b) {

        System.out.println("Sum: " + calculate(a, b));

    }



    private int calculate(int x, int y) { // Private method

        return x + y;

    }



    static void info() {

        System.out.println("Calculator Interface");

    }

}
```

```
class MathOperations implements Calculator {}



public class Main {

    public static void main(String[] args) {

        MathOperations obj = new MathOperations();

        obj.add(5, 3);  // Uses private method internally

        Calculator.info(); // Calls static method

    }

}
```

3. What is the use of interface in Java?Or, why do we use an interface in Java

Ans3. Interfaces in Java are used to achieve **abstraction**, **loose coupling**, **multiple inheritance**, and **polymorphism**. They define a contract that multiple classes can follow without enforcing a specific implementation.

Uses of an Interface in Java

1. Achieving Abstraction

An interface allows you to define a **contract** (method signatures) without specifying how the methods are implemented.

Example:

```
interface Animal {

    void makeSound(); // Abstract method

}




class Dog implements Animal {

    public void makeSound() {

        System.out.println("Woof!");

    }

}
```

```
class Cat implements Animal {

    public void makeSound() {

        System.out.println("Meow!");

    }

}



public class Main {

    public static void main(String[] args) {

        Animal myPet = new Dog();

        myPet.makeSound(); // Outputs: Woof!

    }

}
```

2. Supporting Multiple Inheritance

Java does **not** allow multiple class inheritance, but interfaces help achieve multiple inheritance by allowing a class to implement multiple interfaces.

Example:

```
interface Flyable {
    void fly();
}

interface Swimmable {
    void swim();
}

class Duck implements Flyable, Swimmable {
    public void fly() {
        System.out.println("Duck is flying.");
    }
```

```
    public void swim() {
        System.out.println("Duck is swimming.");
    }
}

public class Main {
    public static void main(String[] args) {
        Duck duck = new Duck();
        duck.fly();
        duck.swim();
    }
}
```

3. Achieving Loose Coupling

Interfaces **decouple** the implementation from the code using it. This makes it easier to change implementations without modifying dependent code.

Example:

```
interface Payment {
    void pay(int amount);
}

class CreditCardPayment implements Payment {
    public void pay(int amount) {
        System.out.println("Paid " + amount + " using Credit Card.");
    }
}

class PayPalPayment implements Payment {
    public void pay(int amount) {
        System.out.println("Paid " + amount + " using PayPal.");
```

```
    }
}

class ShoppingCart {
    private Payment paymentMethod;

    ShoppingCart(Payment paymentMethod) {
        this.paymentMethod = paymentMethod;
    }

    void checkout(int amount) {
        paymentMethod.pay(amount);
    }
```

```
}



public class Main {

    public static void main(String[] args) {

        ShoppingCart cart1 = new ShoppingCart(new CreditCardPayment());

        cart1.checkout(100); // Uses Credit Card



        ShoppingCart cart2 = new ShoppingCart(new PayPalPayment());

        cart2.checkout(200); // Uses PayPal

    }

}
```

4. Enforcing Consistency

Interfaces ensure that all implementing classes follow a common structure.

Example:

```
interface Vehicle {
```

```
    void start();

    void stop();

}



class Car implements Vehicle {

    public void start() { System.out.println("Car is starting."); }

    public void stop() { System.out.println("Car is stopping."); }

}



class Bike implements Vehicle {

    public void start() { System.out.println("Bike is starting."); }

    public void stop() { System.out.println("Bike is stopping."); }

}
```

5. Supporting Functional Programming (Java 8+)

Interfaces with a **single abstract method** are called **functional interfaces**, allowing the use of **lambda expressions**.

Example:

```
@FunctionalInterface

interface Greeting {

    void sayHello(String name);

}




public class Main {

    public static void main(String[] args) {

        Greeting greet = (name) -> System.out.println("Hello, " + name);

        greet.sayHello("Alice");

    }

}
```

4. What is the difference between abstract class and interface in Java?

Ans4.**Difference Between Abstract Class and Interface in Java**

Both **abstract classes** and **interfaces** are used for abstraction in Java, but they have different use cases and characteristics.

Feature	Abstract Class	Interface
Implementation	both **abstract** and **non-abstract (c**	nly abstract methods. Can have **default and static methods**.
Modifiers	instance variables (non-static, non-	les are **implicitly public, static, and fi** s).
:tor	constructors.	ave constructors.
Inheritance	an extend only one abstract class.	an implement multiple interfaces.
Iodifiers	`public`, `protected`, `private`	nly `public` abstract methods. Can have `private` methods.

Examples

1. Abstract Class Example

```
abstract class Animal {

    String name;



    Animal(String name) {

        this.name = name;

    }



    abstract void makeSound(); // Abstract method



    void sleep() { // Concrete method

        System.out.println(name + " is sleeping.");
```

```
    }
}

class Dog extends Animal {
    Dog(String name) {
        super(name);
    }

    public void makeSound() {
        System.out.println("Woof!");
    }
}
```

```
public class Main {

    public static void main(String[] args) {

        Dog dog = new Dog("Buddy");

        dog.makeSound(); // Outputs: Woof!

        dog.sleep(); // Outputs: Buddy is sleeping.

    }

}
```

2. Interface Example

```
interface Vehicle {

    void start(); // Abstract method



    default void stop() { // Default method (Java 8+)

        System.out.println("Vehicle is stopping.");
```

```
    }

    static void fuelType() { // Static method (Java 8+)
        System.out.println("Uses petrol or diesel.");
    }
}

class Car implements Vehicle {
    public void start() {
        System.out.println("Car is starting.");
    }
}
```

```
public class Main {

    public static void main(String[] args) {

        Car myCar = new Car();

        myCar.start();

        myCar.stop(); // Calls default method

        Vehicle.fuelType(); // Calls static method

    }

}
```