In Java, an **interface** is a blueprint that defines **what a class should do**, without necessarily specifying **how it does it**.

For example:-

```
interface Animal
 {
   void sound();
}

class Dog implements Animal {
   public void sound() {
      System.out.println("Dog barks");
   }
}
```

Here, `Animal` is an interface. It declares the method `sound()`.

The `Dog` class **implements** the interface and provides the actual code for `sound()`.

Key points

- Interfaces are declared using the `interface` keyword.
- A class uses `implements` to implement an interface.
- Interface methods are `public abstract` by default unless they are `default`, `static`, or `private` methods.
- Interface fields are implicitly `public static final` constants.
- A class can implement **multiple interfaces**, which is one way Java supports multiple inheritance of type.

1. Basic Interface Example

```
interface Vehicle {
   void start();
   void stop();
}

class Car implements Vehicle {

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

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

public class Main {
   public static void main(String[] args) {

      Car c = new Car();

      c.start();
      c.stop();
   }
}
```

What happens?

The interface says:

```
interface Vehicle {
   void start();
   void stop();
}
```

It basically says:

"Any class that implements `Vehicle` must have `start()` and `stop()` methods."

The `Car` class fulfills that contract:

```
class Car implements Vehicle
```

Output:

```
Car starts
Car stops
```

---

2. Multiple Classes Implementing the Same Interface

This is one of the most important uses of interfaces.

```
interface Animal {
   void sound();
}

class Dog implements Animal {

   public void sound() {
      System.out.println("Dog barks");
   }
```

```
}

class Cat implements Animal {

   public void sound() {
      System.out.println("Cat meows");
   }
}

public class Main {
   public static void main(String[] args) {

      Dog d = new Dog();
      d.sound();

      Cat c = new Cat();
      c.sound();
   }
}
```

Output:

```
Dog barks
Cat meows
```

Both `Dog` and `Cat` follow the same `Animal` interface, but they behave differently.
This is called **polymorphism**.

---

3. Interface Reference

You can also create a reference using the interface type:

```
interface Animal {
   void sound();
}

class Dog implements Animal {

   public void sound() {
      System.out.println("Dog barks");
   }
}

class Cat implements Animal {

   public void sound() {
      System.out.println("Cat meows");
   }
}

public class Main {
   public static void main(String[] args) {

      Animal a;

      a = new Dog();
      a.sound();

      a = new Cat();
      a.sound();
   }
}
```

Output:

Dog barks
Cat meows

Notice:

Animal a;

a is an Animal reference.

Then:

a = new Dog();

and later:

a = new Cat();

This is very useful because your code can work with the **interface** rather than being tightly connected to a particular class.

---

4. Real-World Example: Payment

Imagine an application that supports different payment methods.

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

class CreditCard implements Payment {

    public void pay(double amount) {
        System.out.println("Paid ₹" + amount + " using Credit Card");
    }
}

class UPI implements Payment {

    public void pay(double amount) {
        System.out.println("Paid ₹" + amount + " using UPI");
    }
}

public class Main {
    public static void main(String[] args) {

        Payment p;

        p = new CreditCard();
        p.pay(1000);

        p = new UPI();
        p.pay(500);
    }
}
```

Output:

Paid ₹1000.0 using Credit Card
Paid ₹500.0 using UPI

The application doesn't need to worry about the exact payment implementation.

It just knows:

Payment p;
p.pay(1000);

---

5. One Class Can Implement Multiple Interfaces

A Java class cannot extend multiple classes, but it **can implement multiple interfaces**.

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

interface Swimmable {
   void swim();
}

class Duck implements Flyable, Swimmable {

   public void fly() {
      System.out.println("Duck can fly");
   }

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

public class Main {
   public static void main(String[] args) {

      Duck d = new Duck();

      d.fly();
      d.swim();
   }
}
```

Output:

```
Duck can fly
Duck can swim
```

Here:

```
class Duck implements Flyable, Swimmable
```

means Duck follows **two contracts**.

---

6. Interface with Variables

Interface variables are automatically:

```
public static final
```

Example:

```
interface Constants {

   int MAX_SPEED = 120;
}

class Car implements Constants {

   void display() {
      System.out.println(MAX_SPEED);
   }
}
```

You don't have to write:

```
public static final int MAX_SPEED = 120;
```

Java automatically treats it that way.

---

7. Interface with `default` Method

Modern Java interfaces can also have `default` methods containing implementation.

```
interface Animal {

   void sound();

   default void sleep() {
      System.out.println("Animal is sleeping");
   }
}

class Dog implements Animal {

   public void sound() {
      System.out.println("Dog barks");
   }
}

public class Main {
   public static void main(String[] args) {

      Dog d = new Dog();

      d.sound();
      d.sleep();
   }
}
```

Output:

```
Dog barks
Animal is sleeping
```

`Dog` doesn't need to implement `sleep()` because the interface already provides a default implementation.

---

8. Interface vs Class

A simple way to remember the difference:

Interface	Class
Defines a contract	Provides implementation/state
Uses `interface`	Uses `class`
Class uses `implements`	Class uses `extends` for inheritance
A class can implement multiple interfaces	A class can extend only one class
Useful for abstraction	Used to create objects