## Java Interface

An interface in Java is a blueprint of a behaviour. A Java interface contains static constants and abstract methods.

**Key Properties of Interface:**

* The interface in Java is a mechanism to achieve [abstraction](https://www.geeksforgeeks.org/abstraction-in-java-2) and [multiple inheritance](https://www.geeksforgeeks.org/java-and-multiple-inheritance/) in Java.
* By default,**variables in an interface are public, static, and final.**
* It **supports loose coupling** (classes depend on behavior, not implementation).

**Note:** In Java, the [**abstract keyword**](https://www.geeksforgeeks.org/abstract-keyword-in-java/) applies only to classes and methods, indicating that they cannot be instantiated directly and must be implemented. When we decide on a type of entity by its behaviour and not via attribute we should define it as an interface.

A class that implements an interface must implement all the methods declared in the interface otherwise that class has to be declared abstract. To implement the interface, use the implements keyword.

**When to Use Class and Interface?**

* **Use a Class when**:
  + Use a class when you need to represent a real-world entity with attributes (fields) and behaviors (methods).
  + Use a class when you need to create objects that hold state and perform actions.
* **Use a Interface when:**
  + Use an interface when you need to define a contract for behavior that multiple classes can implement.
  + Interface is ideal for achieving abstraction and multiple inheritance.

## Multiple Inheritance in Java Using Interface

Multiple Inheritance is an OOPs concept that can't be implemented in Java using classes. But we can use multiple inheritances in Java using Interface

## New Features Added in Interfaces in JDK 8

There are certain features added to Interfaces in JDK 8 update mentioned below:

### 1. Default Methods

* Interfaces can define methods with default implementations.
* Useful for adding new methods to interfaces without breaking existing implementations.

**Example:**

// interfaces can have methods from JDK 1.8 onwards  
interface TestInterface {  
 final int *a* = 10;  
  
 default void display() {  
 System.*out*.println("hello");  
 }  
}  
  
// A class that implements the interface.  
class TestClass implements TestInterface {  
 // Driver Code  
 public static void main (String[] args) {  
 TestClass t = new TestClass();  
 t.display();  
 }  
}  
  
Output hello

### *2.*Static Methods

* These methods are called directly using the interface name and are not inherited by implementing classes.

**Example:**

interface TestInterface {  
 final int *a* = 10;  
 static void display()  
 {  
 System.*out*.println("hello");  
 }  
}  
  
// A class that implements the interface.  
class TestClass implements TestInterface {  
 // Driver Code  
 public static void main (String[] args)  
 {  
 TestInterface.*display*();  
 }  
}  
  
Output hello

### 3. Private Methods(Since java 9)

* Private methods are defined within the interface but it cannot be accessed by the implementing classes.

**Example:**

interface Vehicle {  
 // Private method for internal use  
 private void startEngine() {  
 System.*out*.println("Engine started.");  
 }  
  
 // Default method that uses the private method  
 default void drive() {  
 // Calls the private method  
 startEngine();  
 System.*out*.println("Vehicle is now driving.");  
 }  
}  
  
class Car implements Vehicle {  
 // Car class implements Vehicle interface and inherits the default method 'drive'  
}  
  
public class Main {  
 public static void main(String[] args) {  
 Car car = new Car();  
 // This will call the default method, which in turn calls the private method  
 car.drive();  
 }  
}  
  
Output  
Engine started.

Vehicle is now driving.

### 4. Functional Interface

**A functional interface in Java is an interface that contains only one abstract method. Functional interfaces can have multiple default or static methods, but only one abstract method.**[**Runnable**](https://www.geeksforgeeks.org/runnable-interface-in-java/)**,**[**ActionListener**](https://www.geeksforgeeks.org/java-actionlistener-in-awt/)**, and**[**Comparator**](https://www.geeksforgeeks.org/comparator-interface-java/)**are common examples of Java functional interfaces.**

* Functional interfaces can be used with lambda expressions or method references.
* The**@FunctionalInterface annotation** can be used to indicate that an interface is a functional interface, although it’s optional.

**Example**

@FunctionalInterface  
interface MyFunctionalInterface {  
 void singleAbstractMethod();  
}

public class Geeks {  
  
 public static void main(String[] args) {  
  
 // Using lambda expression   
 // to implement Runnable  
 new Thread(() -> System.*out*.println("New thread created")).start();  
 }  
}

## Interface vs abstract class

| **Feature** | **Interface** | **Abstract Class** |
| --- | --- | --- |
| Multiple inheritance | ✅ Supported | ❌ Not supported |
| Constructors | ❌ Not allowed | ✅ Allowed |
| Access Modifiers | Only public methods | Can have any access modifier |
| Fields | public static final only | Can have any type of field |
| Methods | abstract, default, static | abstract and concrete |

## Extending Interfaces

One interface can inherit another by the use of keyword extends. When a class implements an interface that inherits another interface, it must provide an implementation for all methods required by the interface inheritance chain.

*In general, the development process is step by step:*

***Level 1*** *- interfaces: It contains the service details.****Level 2*** *- abstract classes: It contains partial implementation.****Level 3*** *- implementation classes: It contains all implementations.****Level 4*** *- Final Code / Main Method: It have access of all interfaces data.*

**Example:**

// implementation Level wise  
import java.io.\*;  
import java.lang.\*;  
import java.util.\*;  
  
// Level 1  
interface Bank {  
 void deposit();  
 void withdraw();  
 void loan();  
 void account();  
}  
  
 // Level 2  
 abstract class Dev1 implements Bank {  
 public void deposit()  
 {  
 System.*out*.println("Your deposit Amount :" + 100);  
 }  
 }  
  
 abstract class Dev2 extends Dev1 {  
 public void withdraw()  
 {  
 System.*out*.println("Your withdraw Amount :" + 50);  
 }  
 }  
  
 // Level 3  
 class Dev3 extends Dev2 {  
 public void loan() {}  
 public void account() {}  
 }  
  
 // Level 4  
 class Main  
 {  
 public static void main(String[] args)  
 {  
 Dev3 d = new Dev3();  
 d.account();  
 d.loan();  
 d.deposit();  
 d.withdraw();  
 }  
 }  
  
 Try it on GfG Practice  
  
  
  
 Output  
 Your deposit Amount :100  
 Your withdraw Amount :50

* **Runnable: This interface only contains the**[**run()**](https://www.geeksforgeeks.org/run-method-in-java-thread/)**method.**
* **Comparable: This interface only contains the compareTo() method.**
* **ActionListener: This interface only contains the actionPerformed() method.**
* **Callable: This interface only contains the call() method.**

## New Features Added in Interfaces in JDK 9

From Java 9 onwards, interfaces can contain the following also:

1. Static methods
2. Private methods
3. Private Static methods

* Inside the Interface not possible to declare instance variables because by default variables are **public static final.**
* Inside the Interface, constructors are not allowed.
* From JDK 8, interfaces can have a main method for execution
* Interfaces can declare static, final, and private methods starting from JDK 8 and JDK 9.

## Marker Interface

In Java, a **marker Interface is an empty interface that has no fields or methods.** It is used just to mark or tag a class and act as metadata to provide information about the class that this class should be treated differently. Examples of marker interfaces include **Serializable, Cloneable, and Remote Interfaces.**

## Nested Interface in Java

In Java, we can declare **interfaces** as members of a class or another interface. Such an interface is called a **member interface or nested interface.**

* A nested interface can be declared public, protected, package-private (default), or private. But if we put an interface inside another interface, it is automatically public and static, it simply means that we do not need to add public or static ourselves.
* A top-level interface (not nested) can only be declared as public or package-private (default). It cannot be declared as protected or private.

**Refer to the article**: [Access Modifiers for Classes or Interfaces in Java](https://www.geeksforgeeks.org/access-modifiers-for-classes-or-interfaces-in-java/) for more details.

**Declaration of Nested Interface**

The declaration of the nested interface is:

interface i\_first{  
 interface i\_second{   
...  
 }  
}

When implementing a nested interface, we refer to it as**i\_first.i\_second**, where **i\_first**is the name of the interface in which the interface is nested, and **i\_second** is the interface's name.

There is another nested interface which is nested inside a class its syntax is as follows:

class c\_name{  
 interface i\_name{  
...  
 }  
}

When implementing a nested interface, we refer to it as **c\_name.i\_name**, where **c\_name** is the name of the class in which the interface is nested and **i\_name** is the interface's name.

**Example 1**: Let us have a look at the following code:

// Parent Class  
class Parent {  
  
 // Nested Interface  
 interface Test {  
 void show();  
 }  
}  
  
// Child Class  
class Child implements Parent.Test {  
 public void show()  
↔  
  
 Output  
 show method of interface

**Explanation**: The access specifier of the nested interface **Test** is package-private (default) since no access modifier is specified. We can also assign public, protected, or private access specifiers to nested interfaces inside a class.

**Example 2**: Below is an example of protected Nested Interface.

protected interface Test {  
 void show();  
}  
}  
  
class Child implements Parent.Test {  
 public void show(){  
 System.*out*.println("show method of interface");  
 }  
}  
  
Output  
show method of interface

**Explanation**: In the above example, if we change the access specifier to private, it will cause a compilation error because the derived class **Child** tries to access a private interface.

### Interface Nested Inside Another Interface

An interface can be declared inside another interface also. We mention the interface as **Parent.Test**where Parent is the name of the interface in which it is nested and Test is the name of the interface to be implemented.

**Example 1:**

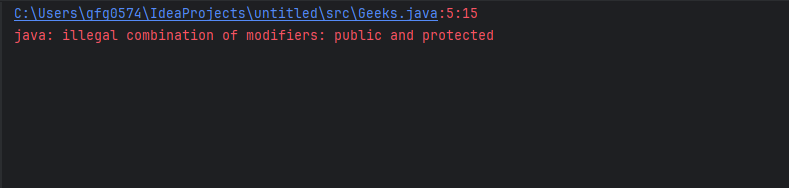
// Nested Interface-Interface  
interface Parent {  
 interface Test {  
 void show();  
 }  
}  
  
class Child implements Parent.Test {  
 public void show() {  
 System.*out*.println("show method of interface");  
 }  
}  
↔  
  
Output  
show method of interface

**Explanation**: In the above example, when we put an interface inside another interface it is automatically **public and static** even if we do not write public and if we try to make it private and protected, the compiler will give an error. Everything inside an interface is always considered public by default.

**Example 2:**

{...}  
interface Parent {  
 protected interface Test {  
 void show();  
 }  
}  
  
class Child implements Parent.Test {  
 public void show()  
 {  
 System.*out*.println("show method of interface");  
 }  
}  
  
 {...}

**Output:**



**Example 3: before jdk 8 by anonymous call.**

// Nested interface  
public interface NestedInterface {  
 public void nestedMethod();  
}  
  
public static void main(String[] args)  
{  
 // Implement nested interface  
 NestedInterface nested = new NestedInterface() {  
 public void nestedMethod()  
 {  
 System.*out*.println(  
 "Hello from nested interface!");  
 }  
 };  
  
 // Call nested interface method  
 nested.nestedMethod();  
}  
}  
  
Output  
Hello from nested interface!

**Explanation**: In this example, we have a nested interface **NestedInterface**inside the outer class. We then implement the interface using an anonymous inner class in the main method and call its method **nestedMethod()**. This is just one way to use nested interfaces in Java.

### Uses of Nested Interfaces

In Java, nested interfaces can be used for a variety of purposes, including:

* When we put one interface inside another interface it makes the code more organized and easy to understand as well.
* If we nest an interface inside a class, it limits where that interface can be used. This helps keep our code safer and reduces the chances because the interface won’t be accessible everywhere.
* Nested interfaces are great for callbacks. This means one object can pass itself to another, and the second object can call a method defined inside the nested interface.
* By using nested interfaces, we can set up a contract. Different classes can follow this contract by implementing the same interface but with their own versions

## ✅ Core Conceptual Questions

**1. What is an interface in Java? How is it different from an abstract class?**

* No constructor in interface
* Multiple inheritance support
* default, static methods in interfaces
* Use-cases: API contracts, strategy pattern, etc.

**2. What is the difference between implements and extends when working with interfaces and classes?**

**Answer:**

* A class implements an interface to provide concrete behavior.
* A class extends another class (single inheritance).
* An interface extends another interface to inherit and extend behavior.

interface A {}  
interface B extends A {}  
class C {}  
class D extends C implements A {}

**3. What are default methods in interfaces? Why were they introduced in Java 8?**

* How do they solve the **diamond problem**? – by providing default implementations
* Can you override default methods? - yes
* How to call specific interface’s default method from the subclass?- by interfaceName.super.methodName()

**4. Can you have private methods in interfaces? If yes, what is the use case?**

**Answer:**  
Yes, since Java 9, interfaces can have private methods. These are used to:

* Refactor common logic among default or static methods.
* Improve code readability without exposing helper methods to implementing classes.

**5. Can a class implement multiple interfaces with same method signatures? What if method signatures differ in return type?**

**Answer:**  
✅ Yes, as long as method **signatures and return types** are the same, there’s **no conflict**.

❌ If return types differ, it's a **compile-time error**.

If two interfaces have the **same default method**, the implementing class must **override** it to resolve ambiguity.

**6. How do interfaces support loose coupling and testability in applications?**

**Answer:**  
Interfaces allow the system to depend on **abstractions rather than implementations** (Dependency Inversion Principle SOLI**D**).

* Promotes flexibility: Swap implementations easily (e.g., EmailService, SmsService)
* Better testing: Easier to **mock interfaces** in unit tests
* Encourages use of **DI frameworks** like Spring

**7. Explain a real-time use case where you used interfaces in your current or previous projects.**

**Answer:**

In my recent project, I used an EventPublisher interface to abstract Kafka event publishing. Different services implemented this interface (EmailEventPublisher, SMSEventPublisher) depending on the notification type. This helped with clean separation of concerns and testability.

**8. Can interfaces extend other interfaces? Can they extend classes?**

**Answer:**

* ✅ Interfaces can **extend multiple interfaces**.
* ❌ Interfaces **cannot extend classes**.
* A class can implement multiple interfaces.

**9. How are functional interfaces used in Java 8+?**

**Answer:**  
Functional interfaces have exactly **one abstract method**, making them ideal for **lambda expressions** and **method references**.

**10. What happens when a class implements two interfaces with the same default method?**

**Answer:**  
This causes a **conflict**. The implementing class **must override** the method to resolve ambiguity.

**✅ Leadership & Design-Oriented Questions**

**11. How would you use interfaces to design a pluggable architecture?**

**Answer:**  
Define interfaces like Plugin, PaymentProcessor, NotificationChannel. Let different modules register their implementations. Use reflection or DI frameworks to load them dynamically.

Benefits:

* Easy to add new modules
* Minimal code change
* Interface becomes the contract

**12. How would you handle versioning in interfaces in large systems?**

**Answer:**

* Avoid changing existing interfaces directly
* Create new versions: UserServiceV2, PaymentServiceV3
* Use adapters/wrappers to support old versions
* Document deprecation plans clearly

**13. Can you explain the difference between Interface Segregation Principle and a fat interface?**

**Answer:**

* **ISP (Interface Segregation Principle):** Clients should not be forced to depend on methods they don’t use.
* A **Fat Interface** is the opposite — too many unrelated methods grouped together.

✅ Instead, split into smaller, **cohesive interfaces**.

**14. How do interfaces help in building microservices or modular monoliths?**

**Answer:**

* Define contracts between services
* Used in API client stubs (like Feign in Spring Cloud)
* Allow mocking in tests
* Interfaces decouple service consumers from implementations

**15. What are some pitfalls of overusing interfaces?**

* Unnecessary abstraction
* Too many tiny interfaces without meaning
* Makes debugging and tracing harder

## Real-time coding problems

#### Problem

interface Animal {  
 void speak();  
}  
  
class Dog implements Animal {  
 public void speak() {  
 System.*out*.println("Bark");  
 }  
  
 public void walk() {  
 System.*out*.println("Dog walking");  
 }  
}  
  
public class Test5 {  
 public static void main(String[] args) {  
 Animal a = new Dog();  
 a.speak();  
 // a.walk(); // Uncomment this  
 }  
}

##### Answer - >

 ✅ a.speak() calls Dog.speak() → "Bark" is printed.

 ❌ a.walk() → Compile-time error.

The reference is of type Animal, which **doesn’t have** a walk() method.

Even though the object is a Dog, you can only call methods declared in the **interface reference type** (Animal in this case).

#### Program

@FunctionalInterface  
interface Printer {  
 void print(String msg);  
}  
  
public class Test1 {  
 public static void main(String[] args) {  
 Printer p1 = System.*out*::println;  
 p1.print("Hello from lambda!");  
  
 Printer p2 = msg -> System.*out*.println("Printing: " + msg);  
 p2.print("Java Interface");  
 }  
}

##### Answer:

Hello from lambda!

Printing: Java Interface

 System.out::println is a **valid method reference** to a method matching void print(String).

 No error. It's a standard Java 8+ feature.

#### Program

interface Engine {  
 void start();  
}  
  
interface AdvancedEngine extends Engine {  
 default void diagnostics() {  
 System.*out*.println("Running diagnostics...");  
 }  
}  
  
class ElectricEngine implements AdvancedEngine {  
 public void start() {  
 System.*out*.println("Electric engine started");  
 }  
}  
  
public class Test2 {  
 public static void main(String[] args) {  
 Engine engine = new ElectricEngine();  
 engine.start();  
 // engine.diagnostics(); // Uncomment this  
 }  
}

##### Answer:

Electric engine started

CTE

 engine is of type Engine and Engine **doesn't declare** diagnostics().

 Even though the object is ElectricEngine, the **reference type limits access**.

Fix - ((AdvancedEngine) engine).diagnostics();

#### Program

interface Reporter {  
 void generateReport();  
}  
  
class PDFReporter implements Reporter {  
 public void generateReport() {  
 System.*out*.println("Generating PDF Report");  
 }  
  
 public void encrypt() {  
 System.*out*.println("Encrypting PDF");  
 }  
}  
  
public class Test4 {  
 public static void main(String[] args) {  
 Reporter rep = new PDFReporter();  
 rep.generateReport();  
 // rep.encrypt(); // Uncomment this  
 ((PDFReporter) rep).encrypt(); // Casted  
 }  
}

##### Answer:

Generating PDF Report

Encrypting PDF

#### Program

interface Logger {  
 default void log() {  
 System.*out*.println("Logger - Interface log");  
 }  
}  
  
class Parent {  
 public void log() {  
 System.*out*.println("Parent - Class log");  
 }  
}  
  
class Child extends Parent implements Logger {  
 // no override  
}  
  
public class Test1 {  
 public static void main(String[] args) {  
 new Child().log();  
 }  
}

##### Answer:

Parent - Class log

🧠 **Rule:**

If a class in the hierarchy provides a concrete method, the default method from the interface is **ignored** — even if both methods have the same signature.

#### Program

interface X {  
 int *VALUE* = 10;  
}  
  
class A {  
 public static final int *VALUE* = 20;  
}  
  
class B extends A implements X {  
 void print() {  
 System.*out*.println(VALUE);  
 }  
}  
  
public class Test2 {  
 public static void main(String[] args) {  
 new B().print();  
 }  
}

##### Answer:

20

 VALUE is inherited from **class A**, and since class fields are resolved **before interfaces**, class A’s field wins.

 To access the interface constant specifically: System.out.println(X.VALUE);

#### Program

interface Utils {  
 static void process() {  
 System.*out*.println("Interface Process");  
 }  
}  
  
class MyUtils implements Utils {  
 public static void process() {  
 System.*out*.println("Class Process");  
 }  
}  
  
public class Test4 {  
 public static void main(String[] args) {  
 MyUtils.*process*(); // Line A  
 Utils.*process*(); // Line B  
 // new MyUtils().process(); // Line C  
 }  
}

##### Answer:

MyUtils.process(); // Line A → Class Process

Utils.process(); // Line B → Interface Process

new MyUtils().process(); // Line C → ❌ Compile-time error

#### Program

@FunctionalInterface  
interface Operation {  
 int apply(int x, int y);  
}  
  
public class Test5 {  
 static int operate(Operation op, int a, int b) {  
 return op.apply(a, b);  
 }  
  
 static int operate(int a, int b, Operation op) {  
 return a - b;  
 }  
  
 public static void main(String[] args) {  
 int result = *operate*((x, y) -> x + y, 10, 5);  
 System.*out*.println("Result: " + result);  
 }  
}

##### Answer:

Result: 15 method chosen - operate(Operation op, int a, int b)

Java matches method based on **parameter types and order**.  
Since the lambda is first, it matches the first operate(...) signature.

#### Program

class Outer {  
 interface Inner {  
 void display();  
 }  
  
 static class InnerImpl implements Inner {  
 public void display() {  
 System.*out*.println("InnerImpl display");  
 }  
 }  
}  
  
public class Test1 {  
 public static void main(String[] args) {  
 Outer.Inner obj = new Outer.InnerImpl();  
 obj.display();  
   
}

##### Answer:

InnerImpl display

#### Program

interface Printer {  
 void print();  
}  
  
class InkjetPrinter {  
 public void print() {  
 System.*out*.println("Inkjet printing...");  
 }  
}  
  
public class Test5 {  
 public static void main(String[] args) {  
 Printer p = (Printer) new InkjetPrinter(); // Line X  
 p.print();  
 }  
}

##### Answer:

Exception in thread "main" java.lang.ClassCastException: class InkjetPrinter cannot be cast to interface Printer

 InkjetPrinter doesn't **implement** Printer.

 So casting it to Printer compiles (no method mismatch), but at **runtime**, JVM throws ClassCastException.

#### Program

interface A {  
 default void show() {  
 System.*out*.println("A");  
 }  
}  
  
interface B extends A {  
 default void show() {  
 System.*out*.println("B");  
 }  
}  
  
class C implements B {}  
  
public class Test4 {  
 public static void main(String[] args) {  
 A obj = new C();  
 obj.show();  
 }  
}

##### Answer:

B

 C implements B, which **overrides** the default method from A.

 Even though the reference is of type A, the **actual implementation is from B**.

#### Program

interface I {  
 default void hello() {  
 System.*out*.println("Hello from Interface");  
 }  
}  
  
abstract class A {  
 public void hello() {  
 System.*out*.println("Hello from Abstract Class");  
 }  
}  
  
class B extends A implements I {  
 // no override  
}  
  
public class Test1 {  
 public static void main(String[] args) {  
 new B().hello();  
 }  
}

##### Answer:

Hello from Abstract Class

When both interface and superclass provide a method with the same signature:

**Class always wins over interface.**

#### Program

interface SecretAgent {  
 default void report() {  
 authenticate();  
 System.*out*.println("Reporting status");  
 }  
  
 private void authenticate() {  
 System.*out*.println("Authenticating...");  
 }  
}  
  
public class Test2 {  
 public static void main(String[] args) {  
 SecretAgent agent = new SecretAgent() {};  
 agent.report();  
 }  
}

##### Answer:

Authenticating...

Reporting status

#### Program

interface Marker {}  
  
class Impl implements Marker {}  
  
public class Test3 {  
 public static void main(String[] args) {  
 Marker obj = new Impl();  
 System.*out*.println(obj.getClass().getInterfaces()[0].getSimpleName());  
 }  
}

##### Answer:

Marker

#### Program

interface Dangerous {  
 void explode() throws Exception;  
}  
  
class Bomb implements Dangerous {  
 public void explode() {  
 System.*out*.println("Boom!");  
 }  
}  
  
public class Test4 {  
 public static void main(String[] args) throws Exception {  
 Dangerous d = new Bomb();  
 d.explode();  
 }  
}

##### Answer:

Boom!

Java allows the overriding method to **narrow** or even **eliminate** checked exceptions.