# **🎯 What is Abstraction?**

**Abstraction** is the process of hiding **implementation details** and showing **only essential features** of an object.

**🔑 Key Points:**

* Focuses on **what** an object does rather than **how** it does it.
* Achieved using **abstract classes** and **interfaces** in Java.
* Promotes **loose coupling**, **code reuse**, and **better maintainability**.

**🏗️ What is an Abstract Class in Java?**

An **abstract class** is a class that **cannot be instantiated**. It can have:

* **Abstract methods** (without body)
* **Concrete methods** (with implementation)
* Constructors
* Variables (instance/static/final)

✅ Syntax:  
abstract class Vehicle {

abstract void start(); // abstract method

void fuelType() { // concrete method

System.out.println("Generic fuel type");

}

}  
  
**🧑‍🔧 Real-World Analogy: Remote Control**

* **RemoteControl (abstract class)** – defines methods like pressButton(), changeChannel(), but doesn’t define how each remote works.
* **TVRemote, ACRemote, ProjectorRemote (subclasses)** – provide actual implementations.

**💡 Real-World Example in Java**

**🎮 Abstract Class Example: Game**

abstract class Game {

abstract void start(); // abstract method

void play() {

System.out.println("Playing the game...");

}

}  
🎯 Subclasses:  
class Football extends Game {

@Override

void start() {

System.out.println("Starting Football Game");

}

}

class Chess extends Game {

@Override

void start() {

System.out.println("Starting Chess Game");

}

}  
✅ Test Class:  
public class Main {

public static void main(String[] args) {

Game g1 = new Football();

Game g2 = new Chess();

g1.start(); // Starting Football Game

g1.play(); // Playing the game...

g2.start(); // Starting Chess Game

g2.play(); // Playing the game...

}

}  
  
**🧾 Rules of Abstract Class in Java**

**✅ 1. Abstract class is declared using the abstract keyword**abstract class Vehicle {

// abstract or concrete methods can go here

}  
  
✅ 2. **You cannot instantiate an abstract class**Vehicle v = new Vehicle(); // ❌ Error!  
  
**✅ 3. An abstract class can have:**

* Abstract methods (without body)
* Concrete methods (with body)
* Constructors
* Fields (instance/static/final)

abstract class Shape {

int sides; // instance variable

static String type = "2D"; // static variable

Shape() { // constructor

System.out.println("Shape created");

}

abstract void draw(); // abstract method

void info() { // concrete method

System.out.println("A generic shape");

}

}  
  
  
✅ 4. **If a class has at least one abstract method, it must be declared abstract**

abstract class Animal {

abstract void makeSound(); // at least one abstract method

}  
  
✅ 5. **Subclasses must override all abstract methods, or be abstract themselves**class Dog extends Animal {

void makeSound() {

System.out.println("Bark");

}

}  
  
abstract class Cat extends Animal {

// still abstract, because makeSound() not overridden

}  
  
✅ 6. Abstract classes can extend other abstract classes

abstract class Person {

abstract void work();

}

abstract class Employee extends Person {

abstract void attendMeeting();

}  
  
**✅ 7. Abstract class can have static and final methods**

* static methods belong to the class, not instances.
* final methods can’t be overridden.

abstract class Demo {

static void display() {

System.out.println("Static method");

}

final void print() {

System.out.println("Final method");

}

}  
  
**🧾 Rules of Abstract Method in Java**

**✅ 1. Abstract methods have no body (end with a semicolon)**

abstract void run(); // ✅  
❌ Invalid:  
abstract void run() {

System.out.println("Running"); // ❌ Error! Abstract method can't have body

}  
  
✅ 2. **Abstract methods can only be declared inside abstract classes**abstract class Machine {

abstract void start(); // ✅ allowed here

}  
class Test {

abstract void run(); // ❌ Error! Only allowed in abstract class

}  
  
✅ 3. **Subclasses must override all abstract methods**abstract class Bird {

abstract void fly();

}

class Sparrow extends Bird {

void fly() {

System.out.println("Sparrow flying...");

}

}  
  
**✅ 4. Abstract methods cannot be private, static, or final**

These modifiers make overriding impossible, which breaks the purpose of abstraction.

abstract class Test {

private abstract void show(); // ❌ Error!

static abstract void run(); // ❌ Error!

final abstract void walk(); // ❌ Error!

}  
  
📌 Summary Table

| **Concept** | **Allowed?** | **Notes** |
| --- | --- | --- |
| Abstract class | ✅ | Can’t be instantiated |
| Abstract method | ✅ | Must be in abstract class |
| Abstract method with body | ❌ | Not allowed |
| Static/Final abstract methods | ❌ | Not allowed |
| Constructors in abstract class | ✅ | Used for initialization |
| Concrete methods in abstract class | ✅ | Fully allowed |
| Subclass must override abstract methods | ✅ | Or it becomes abstract |

**🧪 Quiz Time (with answers)**

**1. What is the purpose of an abstract class?**  
A) Create instances directly  
B) Force subclasses to implement specific methods ✅  
C) Provide only variables  
D) None of the above

**2. Can abstract classes have constructors?**  
A) Yes ✅  
B) No

**3. Can we instantiate an abstract class directly?**  
A) Yes  
B) No ✅

**4. Which of these is valid?**

abstract class Animal {

abstract void sound();

}

A) Animal a = new Animal(); ❌  
B) class Dog extends Animal { void sound() { ... } } ✅  
C) Both A and B  
D) None

# **🔍 What is an Interface in Java?**

An **interface** is a **blueprint** of a class. It defines **what** should be done, but not **how**. It contains **abstract methods** (by default) and **constants**.

**🧱 Syntax:**

interface InterfaceName {

void method1(); // implicitly public and abstract

}

**🧠 Why Use Interfaces?**

* To **achieve abstraction**.
* To **implement multiple inheritance**.
* To **define a contract** that implementing classes must follow

**🧑‍🏫 Real-World Analogy:**

Imagine an **ElectronicDevice interface**. It says every device must be able to:

* turnOn()
* turnOff()

But how these methods are implemented is up to the individual device (like a Fan, TV, or Laptop).

⚙️ Practical Example: Basic Interface  
interface ElectronicDevice {

void turnOn();

void turnOff();

}

class Fan implements ElectronicDevice {

public void turnOn() {

System.out.println("Fan is turning on.");

}

public void turnOff() {

System.out.println("Fan is turning off.");

}

}  
  
public class Main {

public static void main(String[] args) {

ElectronicDevice device = new Fan();

device.turnOn(); // Fan is turning on.

device.turnOff(); // Fan is turning off.

}

}  
**📏 Rules of Interfaces in Java**

| **Rule** | **Description** |
| --- | --- |
| **1** | All interface methods are implicitly public and abstract (unless they are default or static) |
| **2** | Interfaces cannot have constructors |
| **3** | All variables in interfaces are public, static, and final by default |
| **4** | A class uses implements to use an interface |
| **5** | A class can implement **multiple interfaces** |
| **6** | Interfaces can extend other interfaces |

✅ Interface with Constants and Default Methods  
interface Vehicle {

int SPEED\_LIMIT = 120; // public static final

void start();

default void displaySpeedLimit() {

System.out.println("Speed limit is: " + SPEED\_LIMIT);

}

}  
  
**🔁 Multiple Inheritance Using Interfaces**

Java does not support multiple inheritance with classes (to avoid ambiguity like the Diamond Problem), but it supports multiple inheritance with interfaces.  
  
🚀 **Example: Multiple Interfaces**  
interface Camera {

void takePhoto();

}

interface MusicPlayer {

void playMusic();

}

class SmartPhone implements Camera, MusicPlayer {

public void takePhoto() {

System.out.println("Taking a photo...");

}

public void playMusic() {

System.out.println("Playing music...");

}

}  
public class Test {

public static void main(String[] args) {

SmartPhone sp = new SmartPhone();

sp.takePhoto(); // Taking a photo...

sp.playMusic(); // Playing music...

}

}  
✅ This is how **multiple inheritance is achieved in Java**.  
  
**🧪 Quiz Time (with Answers)**

**1. Can an interface contain a constructor?**

* A) Yes
* B) No ✅

**2. Which of the following is true about variables in interfaces?**

* A) They are public
* B) They are static
* C) They are final
* D) All of the above ✅

**3. What keyword is used by a class to use an interface?**

* A) extends
* B) implements ✅
* C) inherits
* D) interface

**4. What happens if a class does not implement all methods of an interface?**

* A) It gives a warning
* B) It runs fine
* C) It must be declared abstract ✅
* D) It throws an exception

**5. Can an interface extend another interface?**

* A) No
* B) Yes ✅

**6. Guess the output**  
interface A {

void show();

}

interface B {

void display();

}

class C implements A, B {

public void show() {

System.out.println("Show from A");

}

public void display() {

System.out.println("Display from B");

}

}

public class Test {

public static void main(String[] args) {

C obj = new C();

obj.show();

obj.display();

}

}  
  
🧠 Quick Summary Before We Dive In

| **Feature** | **Abstract Class** | **Interface** |
| --- | --- | --- |
| **Purpose** | Share common base behavior | Define contracts for unrelated classes |
| **Inheritance** | Single inheritance only | Multiple inheritance supported |
| **Access Modifiers** | Can have all modifiers | All methods are public by default |
| **Constructors** | Allowed | Not allowed |
| **State (fields)** | Can hold instance variables | Only constants (public static final) |
| **Use Case** | “Is-a” relationship | “Can-do” capability |

**🧰 Real-World Use Cases**

**✅ 1. Abstract Class – Common Base Functionality**

**Scenario**: Building a PaymentGateway system

abstract class PaymentGateway {

String transactionId;

void connect() {

System.out.println("Connecting to payment server...");

}

abstract void pay(double amount); // each gateway implements its own logic

}  
class PayPal extends PaymentGateway {

void pay(double amount) {

System.out.println("Paid $" + amount + " via PayPal");

}

}

class RazorPay extends PaymentGateway {

void pay(double amount) {

System.out.println("Paid $" + amount + " via RazorPay");

}

}  
🔎 **Why Abstract Class?**

* All payment gateways share **connection logic**
* Different payment methods need their own pay() implementation
* We may want to maintain **state** (like transaction ID)

**✅ 2. Interface – Define Capability**

**Scenario**: Devices that support **Bluetooth**

interface BluetoothEnabled {

void pair();

void connect();

}  
class Headphones implements BluetoothEnabled {

public void pair() {

System.out.println("Headphones paired.");

}

public void connect() {

System.out.println("Connected to audio.");

}

}

class Speaker implements BluetoothEnabled {

public void pair() {

System.out.println("Speaker paired.");

}

public void connect() {

System.out.println("Connected to mobile.");

}

}  
🔎 **Why Interface?**

* BluetoothEnabled is a **capability**, not a base type
* These devices may have different parent classes already
* You want to **enforce a contract** for pairing/connection behavior

**✅ 3. Interface for Multiple Inheritance**

**Scenario**: A SmartPhone that acts as a **Camera** and a **MusicPlayer**

interface Camera {

void takePhoto();

}

interface MusicPlayer {

void playMusic();

}

class SmartPhone implements Camera, MusicPlayer {

public void takePhoto() {

System.out.println("Click! Photo taken.");

}

public void playMusic() {

System.out.println("Now playing: Jazz music");

}

}  
🔎 **Why Interface?**

* A smartphone **can do** camera and music
* Java classes can **only extend one class**, but **implement multiple interfaces**

**✅ 4. Abstract Class for Common Template**

**Scenario**: Building a **Report Generation System**

abstract class ReportGenerator {

void generateReport() {

fetchData();

formatData();

exportReport();

}

abstract void fetchData();

abstract void formatData();

void exportReport() {

System.out.println("Exporting report to PDF...");

}

}  
  
class SalesReport extends ReportGenerator {

void fetchData() {

System.out.println("Fetching sales data...");

}

void formatData() {

System.out.println("Formatting sales data...");

}

}  
🔎 **Why Abstract Class?**

* Enforces a **template pattern**
* Some methods are **common**, others **must be implemented**
* Keeps structure consistent across reports

🧠 When to Use Abstract Class vs Interface?

| **Use Abstract Class When:** |
| --- |
| ✅ You want to share **code** between subclasses |
| ✅ You want to define **default behavior** |
| ✅ You have **internal state/fields** to manage |
| ✅ You want to enforce a **common parent** |

| **Use Interface When:** |
| --- |
| ✅ You want to **define a capability or role** |
| ✅ You need **multiple inheritance** |
| ✅ You’re designing **plug-and-play components/APIs** |
| ✅ You want **zero implementation** – just contracts |

**📚 Quiz Time (with Answers)**

**1. Which one can have constructors?**

* A) Interface
* B) Abstract Class ✅

**2. Which one allows multiple inheritance?**

* A) Abstract Class
* B) Interface ✅

**3. Which one can have implemented methods?**

* A) Interface
* B) Abstract Class
* C) Both ✅ (via default in interface, and concrete in abstract class)

**4. What’s best for defining a plugin architecture?**

* A) Abstract class
* B) Interface ✅

**5. Which can have private methods in Java 9+?**

* A) Interface ✅
* B) Abstract class
* C) Both

**🔍 Why Private Methods in Interfaces?**

Before Java 9:

* Interfaces could only have:
  + **Abstract methods**
  + **Static methods** (Java 8)
  + **Default methods** (Java 8)

But code duplication became a problem:

* Multiple default methods might have repeated logic.

To solve this, **Java 9** introduced **private methods** in interfaces to: ✅ Remove code duplication  
✅ Increase modularity  
✅ Make helper methods available **only inside the interface**

✅ Syntax of Private Methods in Interface  
private void helperMethod() {

// reusable internal logic

}  
  
🧠 Key Points

| **Feature** | **Description** |
| --- | --- |
| private | Accessible **only inside the interface** |
| Can be used in | default and static methods |
| Cannot be | abstract or inherited by implementing classes |
| Cannot be | Accessed by implementing class or from outside |

✅ Real-World Style Example  
interface GreetingService {

default void greetInEnglish() {

log("Hello!");

}

default void greetInSpanish() {

log("¡Hola!");

}

private void log(String message) {

System.out.println("Greeting: " + message);

}

}  
class Greeter implements GreetingService {

// No need to implement anything!

}

public class Main {

public static void main(String[] args) {

Greeter g = new Greeter();

g.greetInEnglish(); // Output: Greeting: Hello!

g.greetInSpanish(); // Output: Greeting: ¡Hola!

}

}  
🔎 **What’s happening here?**

* The log() method is private
* It's used by multiple default methods to avoid code duplication
* The class Greeter does not and **cannot** access log() directly

💥 Illegal Access Example  
GreetingService service = new Greeter();

service.log("Hi"); // ❌ Compilation error: log() has private access

# **Assignments for this week**

**🧪 Assignment 1: Banking System Design**

**💡 Scenario:**

Design a system where different banks provide different interest rates on fixed deposits.

**🔧 Requirements:**

* Create an **abstract class Bank** with:
  + An abstract method getInterestRate()
  + A method displayBankInfo() (common to all banks)
* Create subclasses:
  + SBI, HDFC, and ICICI that override getInterestRate() with their specific rates.
* Create an interface OnlineBankingFeatures with:
  + void transferMoney()
  + void checkBalance()
* Implement the interface in the respective bank classes with dummy implementation

📥 Sample Input:

Bank bank = new HDFC();

bank.getInterestRate(); // 6.5%

bank.displayBankInfo();

OnlineBankingFeatures ob = new HDFC();

ob.transferMoney();  
  
**📤 Expected Output:**

Bank Name: HDFC

Interest Rate: 6.5%

Transferring Money Online...

**🧪 Assignment 2: Vehicle Rental System**

**💡 Scenario:**

A vehicle rental company rents various types of vehicles.

**🔧 Requirements:**

* Create an abstract class Vehicle with:
  + Properties: brand, fuelType
  + Abstract method calculateRentalPrice(int days)
  + Method displayInfo()
* Create subclasses:
  + Car, Bike, Truck with different rental price logic
* Create interface Maintainable with method void serviceSchedule()
* Implement the interface in each subclass with custom logic

📥 Sample Input:

Vehicle v = new Car("Toyota", "Petrol");

v.displayInfo();

System.out.println("Rent: " + v.calculateRentalPrice(5));

Maintainable m = (Maintainable) v;

m.serviceSchedule();

📤 Expected Output:

Vehicle: Toyota | Fuel: Petrol

Rent: 2500

Car service every 5000 km or 6 months  
  
**🧪 Assignment 3: E-Commerce System Interfaces**

**💡 Scenario:**

You're designing an online shopping platform.

**🔧 Requirements:**

* Create interface Product with methods:
  + String getName()
  + double getPrice()
* Create interface Discountable with method:
  + double applyDiscount(double percentage)
* Create classes Electronics, Clothing, Grocery that implement both interfaces.
* Simulate applying discounts and displaying product details.

📥 Sample Input:

Product p = new Electronics("Laptop", 60000);

Discountable d = (Discountable) p;

System.out.println(p.getName() + ": " + d.applyDiscount(10));  
  
📤 Expected Output:

Laptop: 54000.0