Abstract Classes & Interface

Abstract Class & Interface in Java

---

1. What is an Abstract Class?

Definition:

An **abstract class** in Java is a class that **cannot be instantiated** and may contain **abstract methods** (without a body) as well as **concrete methods** (with a body).

It acts as a **base class** for other classes to extend and implement the abstract behaviors.

Syntax:

```
abstract class Animal {
   abstract void makeSound();  // Abstract method
   void breathe() {            // Concrete method
      System.out.println("Breathing...");
   }
}
```

Key Points:

- Declared using the abstract keyword.
- Can have both abstract and non-abstract methods.
- Can have constructors.
- Can have instance and static variables.
- Can have access modifiers (private, protected, etc.).
- Used for **inheritance and code reuse**.

---

2. What is an Interface?

Definition:

An **interface** in Java is a blueprint of a class that contains **only abstract methods** (Java 7) or **abstract, static, default, and private methods** (Java 8+).

It is used to **achieve 100% abstraction** and **multiple inheritance** in Java.

□ Syntax:

```
interface Flyable {
   void fly();  // public and abstract by default
}
```

□ Key Points:

- Declared using the interface keyword.
- All methods are **implicitly public and abstract** (Java 7).
- Can contain **default**, **static**, and **private** methods (Java 8+).
- Variables are **public, static, and final** by default (i.e., constants).
- Cannot have constructors.
- Cannot maintain state (no instance variables).
- Supports **multiple inheritance**.

---

3. Abstract Class vs Interface (Comparison Table)

Feature	Abstract Class	Interface
Instantiation	Cannot be instantiated	Cannot be instantiated
Methods	Can be abstract or concrete	Abstract (Java 7), default/static (Java 8+)
Constructors	Yes	No
Variables	Instance/static allowed	public static final only
Access Modifiers	Any (public, private, etc.)	Only public (methods & fields)

Feature	Abstract Class	Interface
Multiple Inheritance	Not supported	Supported
Use Case	Partial abstraction, code reuse	Total abstraction, contracts

---

4. When to Use What?

Use This	When you...
Abstract Class	Need to share code among several closely related classes.
Interface	Want to define a contract for classes with **unrelated** behavior types.
	Want **multiple inheritance of type**.

Abstract Methods vs Concrete Methods in Java

---

1. What is an Abstract Method?

Definition:

An **abstract method** is a method **declared without a body**, and must be **overridden in a subclass**.

It only has a method signature — no implementation in the base class.

Syntax:

```
abstract class Shape {
   abstract void draw();  // Abstract method (no body)
}
```

 Key Points:

- Declared using the abstract keyword.
- Cannot have a method body.
- Must be in an abstract class or an interface.
- Must be **overridden** in the first **non-abstract** subclass.
- Promotes **polymorphism and abstraction**.

Example:

```
abstract class Animal {
   abstract void makeSound();  // Abstract method
}

class Dog extends Animal {
   void makeSound() {
      System.out.println("Bark");
   }
}
```

---

2. What is a Concrete Method?

Definition:

A **concrete method** is a **fully implemented** method with a body.

It performs a defined operation and can exist in any class (including abstract classes).

Syntax:

```
class Person {
   void speak() {
      System.out.println("Speaking...");
   }
}
```

 Key Points:

- Has complete implementation (method body).
- Can exist in both normal and abstract classes.
- Can be overridden (if not final).
- Supports **code reuse**.

Example:

```
abstract class Animal {
   void breathe() {
      System.out.println("Breathing...");
   }
}
```

3. Comparison Table

Feature	Abstract Method	Concrete Method
Body	No (only declaration)	Yes (full implementation)
Belongs To	Abstract class or interface	Any class
abstract Keyword	Required	Not required
Inheritance	Must be overridden in subclass	Can be inherited or overridden
Purpose	Define contract or required behavior	Provide reusable functionality

---

4. Use Case Summary

Use Abstract Method When...	Use Concrete Method When...
You want to force subclasses to implement specific behavior.	You want to share common logic among subclasses.
You are designing a **template/base class**.	You are implementing **actual functionality**.

Difference Between Abstract Class and Interface in Java

Feature	Abstract Class	Interface
Keyword	abstract	interface
Methods	Can have **abstract** and **concrete** methods	Only **abstract** methods (Java 7) From Java 8: default, static, private methods
Method Modifiers	Can use any modifier (public, protected, private)	All methods are **public abstract** by default
Variables	Can have instance, static, or final variables	Only **public static final** (constants)
Constructors	Yes, can have constructors	No, cannot have constructors
Object Creation	Cannot instantiate directly	Cannot instantiate directly
Multiple Inheritance	Not supported	Supported (Java supports multiple interfaces)

Feature	Abstract Class	Interface
Inheritance Type	extends keyword (single inheritance only)	implements keyword (can implement multiple interfaces)
Access Modifiers for Members	Can use any access modifier	Members are implicitly public
Use Case	Used for **code sharing**, partial abstraction	Used for defining **contracts**, full abstraction
Speed	Slightly faster (as some implementation is present)	Slightly slower (more abstraction and indirection)
State (instance variables)	Can maintain state via instance variables	Cannot maintain state

---

Example Summary:

Abstract Class

```
abstract class Animal {
   abstract void makeSound();
   void eat() {
      System.out.println("Eating...");
   }
}
```

Interface

```
interface Animal {
   void makeSound();  // abstract method
}
```

 When to Use

Scenario	Prefer
You need to define **base behavior + structure**	Abstract Class
You want to define a **contract for unrelated classes**	Interface
You need **multiple inheritance**	Interface
You want to **share some default code**	Abstract Class

Interface Changes After Java 1.8

With the release of **Java 8**, interfaces in Java became **more powerful** and **flexible** than ever before. Prior to Java 8, interfaces could only have **public abstract methods** and **public static final variables**. But after Java 8, interfaces can also contain **default**, **static**, and **private** methods.

---

1. Default Methods (Java 8)

Purpose:

To allow method implementation in interfaces **without breaking** existing implementing classes.

 Why?

Earlier, if you added a method in an interface, all implementing classes had to override it. default methods solve that issue.

 Syntax:

```
interface Vehicle {
   default void start() {
      System.out.println("Vehicle starting...");
   }
```

```
}
```

 Key Points:

- Marked with the default keyword.
- Can be overridden by implementing class.
- Cannot be abstract, final, strictfp, or native.

---

2. Static Methods (Java 8)

Purpose:

To define utility/helper methods in the interface itself.

Syntax:

```
interface MathUtils {
   static int add(int a, int b) {
      return a + b;
   }
}
```

 Key Points:

- Belongs to the interface, not instances.
- Cannot be overridden.
- Called using the interface name: MathUtils.add(5, 3);

---

3. Private Methods (Java 9+)

Purpose:

To help **reuse common code** within default and static methods of interfaces.

Syntax:

```
interface Logger {
   private void log(String msg) {
```

```
        System.out.println("LOG: " + msg);
    }

    default void info(String info) {
        log(info);
    }
}
```

 Key Points:

- Accessed only **inside the interface**.
- Cannot be abstract, static, final, or synchronized.
- Helps to avoid code duplication.

 Benefits of Java 8+ Interface Enhancements

Benefit	Explanation
Backward Compatibility	Adding methods won't break existing code
Multiple Interface Behaviors	Enables more flexible multiple inheritance
Code Reusability	default and private allow cleaner code
Utility Methods in Interface	static methods can be used like helpers

Remaining Core Concepts (Before Java 8) – Abstract Classes & Interfaces

---

1. Partial vs Full Abstraction

- **Abstract Class**: Allows **partial abstraction** (can have both abstract and concrete methods).
- **Interface**: Provides **100% abstraction** (only abstract methods allowed in Java 7).

---

2. Multiple Inheritance Problem

- Java **does not support multiple inheritance with classes** to avoid ambiguity.
- But Java **supports multiple inheritance with interfaces**:

```
interface A {
   void show();
}

interface B {
   void display();
}

class C implements A, B {
   public void show() { System.out.println("A"); }
   public void display() { System.out.println("B"); }
}
```

 No ambiguity since interface methods are abstract (no body).

---

3. Interface Extending Another Interface

- An interface can **extend** another interface using the extends keyword.
- It can also extend **multiple interfaces** (multiple inheritance).

```
interface A {
   void methodA();
}

interface B extends A {
   void methodB();
}
```

---

4. Class Implementing Multiple Interfaces

- A class can **implement multiple interfaces**, enabling multiple inheritance of types:

```
interface A { void methodA(); }
interface B { void methodB(); }

class C implements A, B {
   public void methodA() { System.out.println("A"); }
   public void methodB() { System.out.println("B"); }
}
```

---

5. Interface vs Abstract Class in Memory & Compilation

Concept	Explanation
Interface	Cannot have instance variables (no state), hence stateless.
Abstract Class	Can maintain state with instance variables.
Compilation	Interface methods are **implicitly public abstract**, so access modifiers are restricted.
Memory	Interfaces do not hold memory unless implemented by a class.

---

6. Interface Constants

- Variables declared in an interface are implicitly:

  public static final

```
interface Test {
   int VALUE = 10; // same as: public static final int VALUE = 10;
}
```

---

7. Polymorphism with Abstract Class and Interface

Both abstract classes and interfaces support **runtime polymorphism**.

```
abstract class Shape {
   abstract void draw();
}

class Circle extends Shape {
   void draw() { System.out.println("Drawing Circle"); }
}

Shape s = new Circle(); // Polymorphism
s.draw();
```

Same applies to interfaces:

```
interface Drawable {
   void draw();
}

class Circle implements Drawable {
   public void draw() { System.out.println("Circle"); }
}

Drawable d = new Circle(); // Polymorphism
d.draw();
```

---

8. Nested Interfaces (Interface inside a class or interface)

```
class Outer {
   interface Nested {
      void message();
   }
}

class Test implements Outer.Nested {
   public void message() {
      System.out.println("Nested Interface Example");
   }
```

```
}
```