
# DIGITAL NOTES
# ON
# JAVA PROGRAMMING

FOR

COMPUTER ENGG. STUDENT

PROVIEDED BY: -

SHAKTI RAJ SINGH, PROGRAMMER

**UNIT IV**

**Abstract Class & Interface in Java**

**1. Abstract Class**

**Definition:**

An **abstract class** in Java is a class that cannot be instantiated directly. It is meant to be inherited by other classes. The abstract class can have both **abstract methods** (methods without implementation) and **concrete methods** (methods with implementation). The purpose of an abstract class is to provide a common template for other classes to implement, while leaving some methods abstract so that subclasses can provide specific implementations for them.

- An **abstract class** can have:
  - **Abstract methods**: These are methods without implementation (no method body).
  - **Concrete methods**: These are regular methods that provide implementation.
  - **Fields**: Can have instance variables (fields).
  - **Constructors**: Abstract classes can have constructors, which are invoked when a subclass is instantiated.

**Example of Abstract Class:**

```
// Abstract class
abstract class Animal {
   // Abstract method (does not have a body)
   abstract void sound();

   // Regular method
   void sleep() {
      System.out.println("This animal is sleeping");
   }
}

// Subclass (inherited from Animal)
class Dog extends Animal {
   // Providing implementation for the abstract method
   void sound() {
      System.out.println("Barking...");
   }
}

public class Main {
   public static void main(String[] args) {
      Dog dog = new Dog();  // Creating an object of the subclass
      dog.sound();  // Calls the method implemented in the subclass
      dog.sleep();  // Calls the inherited method from Animal class
   }
}
```

**Output**:

```
Barking...
This animal is sleeping
```

**Advantages of Abstract Classes:**

1. **Code Reusability**: Abstract classes allow you to define methods that multiple subclasses can inherit. Concrete methods in the abstract class can be reused without modification.
2. **Partial Implementation**: Abstract classes allow a class to define some methods with implementation, and leave others abstract. This allows subclasses to focus only on specific parts of functionality that need customization.
3. **Common Base for Related Classes**: Abstract classes are used when we want to define a common base for a group of related classes that share some common functionality.
4. **Flexibility in Method Implementation**: Abstract classes allow you to provide default behavior (concrete methods), while also allowing subclasses to implement their own specialized behavior (abstract methods).

**Disadvantages of Abstract Classes:**

1. **Single Inheritance**: Java does not support **multiple inheritance** for classes. If a class extends one abstract class, it cannot extend another. This limitation restricts flexibility in certain designs.
2. **Tight Coupling**: Using abstract classes can cause tight coupling between classes. If one class is tightly coupled to another, it may be harder to modify and extend the system in the future.

**2. Interface**

**Definition:**

An **interface** in Java is a completely abstract class that is used to define a set of methods that a class must implement. It provides a way to define a contract or behavior that classes must adhere to. In Java, interfaces can contain:

- **Abstract methods** (methods without a body).
- **Default methods** (methods with a body) introduced in Java 8.
- **Static methods** (methods with a body) introduced in Java 8.
- **Constants** (public static final variables).

Unlike abstract classes, interfaces cannot contain instance variables or constructors, and all methods are implicitly public and abstract (except default and static methods).

- A class can implement multiple interfaces, which is a way to achieve **multiple inheritance**.

**Example of Interface:**

```
// Interface
interface Animal {
   // Abstract method
   void sound();

   // Default method (Java 8 onwards)
   default void breathe() {
      System.out.println("Breathing...");
   }
}

// Implementing class
class Dog implements Animal {
   // Implementing the abstract method
   public void sound() {
      System.out.println("Barking...");
   }
```

```
}

public class Main {
   public static void main(String[] args) {
      Dog dog = new Dog();  // Creating an object of Dog class
      dog.sound();  // Calls the method implemented in Dog class
      dog.breathe(); // Calls the default method from the Animal interface
   }
}
```

**Output**:
Barking...
Breathing...

**Advantages of Interfaces:**

1. **Multiple Inheritance**: Interfaces allow a class to inherit behavior from multiple sources, whereas classes only support single inheritance. A class can implement multiple interfaces, allowing greater flexibility in design.
2. **Loose Coupling**: Interfaces decouple the specification of behavior (interface) from the implementation of behavior (class). This improves the modularity of the program.
3. **Flexible Design**: Interfaces can define contracts for any class to implement, regardless of the class hierarchy. This allows different classes to implement the same interface and provides polymorphic behavior.
4. **Default Methods**: Since Java 8, interfaces can have default methods that provide a default implementation. This allows interfaces to evolve without breaking existing implementations.

**Disadvantages of Interfaces:**

1. **No Implementation (until Java 8)**: Before Java 8, interfaces could not provide any method implementation, which meant that implementing classes had to implement all methods. This sometimes led to redundant code.
2. **No Instance Variables**: Interfaces cannot have instance variables, which means they cannot maintain any state. This may limit their usefulness in some designs.
3. **Increased Complexity**: Overusing interfaces, especially in large systems, can lead to code that is difficult to follow. Too many interfaces can complicate the design and make it harder to maintain.

**3. Key Differences Between Abstract Class and Interface**

Here's a detailed comparison between **Abstract Classes** and **Interfaces** to help you understand their respective features and when to use them:

| Feature | Abstract Class | Interface |
|---|---|---|
| **Instantiation** | Cannot be instantiated directly. | Cannot be instantiated directly. |
| **Methods** | Can have both abstract and concrete methods. | Can only have abstract methods (prior to Java 8), but can have default and static methods (from Java 8 onwards). |
| **Constructor** | Can have constructors. | Cannot have constructors. |
| **Multiple Inheritance** | Supports single inheritance only. | Supports multiple inheritance (a class can implement multiple interfaces). |
| **Access Modifiers** | Can have access modifiers (public, private, protected). | Methods are implicitly public. |

| Feature | Abstract Class | Interface |
|---|---|---|
| **State (Fields)** | Can have instance variables (fields). | Cannot have instance variables, only constants (static final variables). |
| **Default Methods** | Cannot have default methods (before Java 8). | Can have default methods (from Java 8 onwards). |
| **Implementation** | A subclass must implement abstract methods, but can also use inherited concrete methods. | A class must implement all methods of the interface. |
| **Use Case** | Used when a class needs to share common behavior and state. | Used when a class needs to provide specific behavior without enforcing any state. |
| **Inheritance Type** | Supports inheritance through extends. | Supports inheritance through implements. |

**4. Implementing Multiple Inheritance Through Interface**

Java does not support **multiple inheritance** (i.e., a class cannot inherit from more than one class), but it allows **multiple inheritance** through interfaces. This means that a class can implement multiple interfaces, effectively allowing it to inherit behavior from several sources.

**Example of Multiple Inheritance Using Interfaces:**

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

interface Mammal {
   void breathe();
}

class Dog implements Animal, Mammal {
   // Implementing methods from both interfaces
   public void sound() {
      System.out.println("Barking...");
   }

   public void breathe() {
      System.out.println("Breathing...");
   }
}

public class Main {
   public static void main(String[] args) {
      Dog dog = new Dog();
      dog.sound();   // Output: Barking...
      dog.breathe(); // Output: Breathing...
   }
}
```

**Output**:
Barking...
Breathing...
In this example, the Dog class implements two interfaces (Animal and Mammal), inheriting the methods from both. This demonstrates **multiple inheritance** through interfaces.

**Advantages of Multiple Inheritance Using Interfaces:**

1. **Increased Flexibility**: A class can implement multiple interfaces, allowing it to inherit behavior from more than one source, which cannot be achieved using class inheritance alone.
2. **Avoids Diamond Problem**: The diamond problem, which occurs in languages supporting multiple inheritance, does not occur in Java's interface implementation, because Java interfaces can be implemented but not inherited directly. If multiple interfaces define the same method, the class implementing the interface can provide its own method implementation.

**Disadvantages:**

1. **Complexity**: With the ability to implement multiple interfaces, the design can become more complex. A class implementing many interfaces might end up with many methods that need to be implemented.
2. **Ambiguity in Method Resolution**: If two interfaces define the same method, the implementing class must provide its own implementation. This may lead to ambiguity if not handled properly.

**5. When to Use Abstract Classes vs Interfaces**
**Use Abstract Classes When:**

- You want to provide some common functionality or state (fields) to all derived classes.
- You want to give a default implementation of some methods but require subclasses to implement others.
- You need to share code between related classes but still need to allow different behaviors for some methods.

**Use Interfaces When:**

- You need to specify a **contract** or **behavior** that can be implemented by any class, regardless of its position in the class hierarchy.
- You want to provide **multiple inheritance** of behavior, which Java allows through interfaces.
- You are designing a system where classes can perform a common set of operations, but they don't need to share any state (fields).

**6. Conclusion**
Both **abstract classes** and **interfaces** are vital in **Java's object-oriented programming model**. They enable different aspects of **abstraction** and **polymorphism**:

- **Abstract classes** provide a foundation for related classes and allow for code reuse and partial implementation.
- **Interfaces** provide a way to define a common behavior for unrelated classes, and they support **multiple inheritance**.

By choosing **abstract classes** or **interfaces** correctly, you can make your system more **modular**, **flexible**, and **extensible**, and manage complexity effectively in large applications.