INTRODUCTION TO PROGRAMMING LANGUAGE II(JAVA)

Fariha Zahin

What is Abstraction?

- **Definition:**

Abstraction is the process of hiding implementation details and showing only functionality to the user.

- **Purpose:**

Focus on *what* an object does instead of *how* it does it.

- **Achieved By:**
  - Abstract Classes
  - Interfaces

Abstract Class – Definition & Features

- Can have both abstract (without body) and concrete methods.
- Can have member variables (fields).
- Can have constructors.
- Cannot be instantiated directly.
- Partial abstraction(0-100%)

```java
abstract class Animal {
    // Abstract method (no body)
    abstract void sound();

    // Concrete method
    void sleep() {
        System.out.println("Animal is sleeping...");
    }
}

// Concrete subclass
class Dog extends Animal {
    @Override
    void sound() {
        System.out.println("Dog barks");
    }
    public static void main(String[] args) {
        Dog d = new Dog();
        d.sound();  // Output: Dog barks
        d.sleep();  // Output: Animal is sleeping...
    }
}
```

Animal is an abstract class.
It has an abstract method sound() with no body.
It also has a concrete method sleep() with implementation.
Dog is a non-abstract subclass of Animal.
Since Dog is not abstract, it must override the abstract method sound().
The method sleep() is inherited from Animal and used directly.
In the main() method, an object d of class Dog is created.
Calling d.sound() prints "Dog barks".
Calling d.sleep() prints "Animal is sleeping...".

```java
abstract class Animal {
    abstract void sound();

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

abstract class Dog extends Animal {
    void bark() {
        System.out.println("Dog barking behavior");
    }
}
```

```java
class Beagle extends Dog {
    @Override
    void sound() {
        System.out.println("Beagle barks");
    }
}

public class MainClass {
    public static void main(String[] args) {
        Dog d = new Beagle();  // Dog reference to Beagle object
        d.sound();  // Output: Beagle barks
        d.bark();   // Output: Dog barking behavior
        d.sleep();  // Output: Animal is sleeping...
    }
```

Animal is abstract with abstract sound() and concrete sleep(). Dog is an abstract subclass with bark(). Beagle is a concrete subclass overriding sound(). In main(), a Dog reference points to a Beagle object. Abstract class references can refer to subclass objects. Calling d.sound() runs Beagle's method; d.bark() and d.sleep() come from Dog and Animal.

Dog class is abstract, **you cannot create a direct object of Dog-Java will give a compile-time error.

- One abstract class can have **multiple abstract methods**.
- Concrete subclasses must implement **all** abstract methods.

```
Car starts with key
Car stops with brakes
Uses fuel
```

```
abstract class Vehicle {
    abstract void start();
    abstract void stop();
    void fuelType() {
        System.out.println("Uses fuel");
    }
}
class Car extends Vehicle {
    void start() {
        System.out.println("Car starts with key");
    }
    void stop() {
        System.out.println("Car stops with brakes");
    }
}
public class Main {
    public static void main(String[] args) {
        Vehicle v = new Car();
        v.start();
        v.stop();
        v.fuelType();
    }
}
```

- Abstract classes **can have constructors**.
- Constructor is invoked when a subclass object is created.

```
abstract class Shape {
    String color;
    Shape(String color) {
        this.color = color;
    }
    abstract double area();
    void displayColor() {
        System.out.println("Color: " + color);
    }
}


class Circle extends Shape {
    double radius;
    Circle(String color, double radius) {
        super(color);
        this.radius = radius;
    }
    double area() {
        return Math.PI * radius * radius;
    }
}
```

- Abstract class references enable **runtime method dispatch**.
- Method executed depends on **object type**, not reference type.

```
C:\Users\Farah\.jdks\openjdk-2
Salary: Fixed monthly salary
Salary: Hourly based salary
```

```
abstract class Employee {
    abstract void calculateSalary();
}
class FullTimeEmployee extends Employee {
    void calculateSalary() {
        System.out.println("Salary: Fixed monthly salary");
    }
}
class PartTimeEmployee extends Employee {
    void calculateSalary() {
        System.out.println("Salary: Hourly based salary");
    }
}
public class Main {
    public static void main(String[] args) {
        Employee e1 = new FullTimeEmployee();
        Employee e2 = new PartTimeEmployee();


        e1.calculateSalary();
        e2.calculateSalary();
    }
}
```

Interface – Definition & Features

- All methods are implicitly public and abstract.
- Variables will be constants (public static final).
- Supports multiple inheritance.
- Cannot have constructors or instance fields
- 100% abstraction.
- A class can implement an interface

```
interface Flyable {
    // Variable must be static and final
    int MAX_ALTITUDE = 10000;  // implicitly public static final

    void fly();  // implicitly public and abstract
}
```

- Interface variables are **public static final by default**.
- Methods must be implemented as **public**.

```
interface Bank {
    double INTEREST_RATE = 5.5;
    void calculateInterest();
}


class CityBank implements Bank {
    public void calculateInterest() {
        System.out.println("Interest Rate: " + INTEREST_RATE + "%");
    }
}
```

- Every field in an interface is implicitly **public static final** (constant)
- • Constants can be accessed using **interface name** (preferred) or **implementing class name**

```
12.5664
3.1416
3.1416
```

```
interface Shape{
   double pi = 3.1416;
   double area();
}
class Circle implements Shape{
   double r=2;
   public double area()
   {
      return pi*r*r;
   }
}

public class Main {
   public static void main(String[] args)
   {
      Circle c = new Circle();
      System.out.println(c.area());;
      System.out.println(Circle.pi);
      System.out.println(Shape.pi);
   }
}
```

- Java supports **multiple inheritance through interfaces**.

```
interface Printable {
    void print();
}
interface Scannable {
    void scan();
}
class Printer implements Printable, Scannable {
    public void print() {
        System.out.println("Printing document");
    }
    public void scan() {
        System.out.println("Scanning document");
    }
}
```

- **Interface references** behave like abstract class references.

```
Paid using credit card
Paid using mobile banking
```

```java
interface Payment {
    void pay();
}
class CreditCardPayment implements Payment {
    public void pay() {
        System.out.println("Paid using credit card");
    }
}
class MobileBankingPayment implements Payment {
    public void pay() {
        System.out.println("Paid using mobile banking");
    }
}
public class Main {
    public static void main(String[] args) {
        Payment p1 = new CreditCardPayment();
        p1.pay();
        Payment p2 = new MobileBankingPayment();
        p2.pay();
    }
}
```

Key Differences: Abstract Class vs Interface

Interface	Abstract
Java interface are implicitly abstract and cannot have implementations	A Java abstract class can have instance methods that implements a default behavior
Variables declared in a Java interface is by default final	An abstract class may contain non-final variables.
Members of a Java interface are public by default	A Java abstract class can have the usual flavors of class members like private, protected, etc
Java interface should be implemented using keyword “implements”	A Java abstract class should be extended using keyword “extends”
An interface can extend another Java interface only	an abstract class can extend another Java class and implement multiple Java interfaces.
Interface is absolutely abstract and cannot be instantiated	A Java abstract class also cannot be instantiated, but can be invoked if a main() exists.
java interfaces are slow as it requires extra indirection	Comparatively fast

```java
abstract class Animal {
    abstract void sound();
    void sleep() {
        System.out.println("Sleeping...");
    }
}

// Interface
interface Pet {
    void play();
}
```

```java
class Dog extends Animal implements Pet {
    public void sound() {
        System.out.println("Bark");
    }
    public void play() {
        System.out.println("Plays fetch");
    }

    public static void main(String[] args) {
        Dog d = new Dog();
        d.sound();  // Output: Bark
        d.play();   // Output: Plays fetch
        d.sleep();  // Output: Sleeping...
    }
```

Animal is an abstract class with abstract method sound() and concrete method sleep(). Pet is an interface with method play(). Dog is a concrete class that extends Animal and implements Pet. Dog overrides sound() and play() methods. In main(), a Dog object d is created. Calling d.sound() prints "Bark". Calling d.play() prints "Plays fetch". Calling d.sleep() prints "Sleeping...".

What is the Diamond Problem?

- The **Diamond Problem** occurs in multiple inheritance when a class inherits the same method from two different parent classes or interfaces.
- This causes ambiguity: which parent's method should the child inherit or execute?
- In languages like C++, this can lead to conflicts and complexity.
- **Java avoids the diamond problem with classes by not allowing multiple class inheritance.**
- However, with **interfaces**, default methods can cause a similar ambiguity.
- Java requires the implementing class to explicitly override conflicting default methods to resolve this.

```java
class A {
    void show() {
        System.out.println("A's show");
    }
}

class B extends A {
    void show() {
        System.out.println("B's show");
    }
}
```

```java
class C extends A {
    void show() {
        System.out.println("C's show");
    }
}

// Compile error: Java does not allow multiple inheritance with classes
class D extends B, C {
    // Ambiguous: which show() method to inherit?
}
```

Solution to Diamond Problem

- Java forces explicit override to resolve **default method conflicts**.

```
Interface A
```

```
interface A {
    default void show() {
        System.out.println("Interface A");
    }
}
interface B {
    default void show() {
        System.out.println("Interface B");
    }
}
class Demo implements A, B {
    public void show() {
        A.super.show();
    }
}
public class Main {
    public static void main(String[] args)
    {
        Demo d = new Demo();
        d.show();
    }
}
```

Why Interfaces Support Multiple Inheritance

- Java **does not allow multiple inheritance with classes** to avoid complexity and ambiguity (like the "Diamond Problem").
- However, Java allows a class to implement **multiple interfaces** because interfaces only declare method signatures without implementation (except default methods).
- This means a class can inherit method contracts from multiple interfaces without conflicts in code behavior.
- Interfaces provide a way to achieve **multiple inheritance of type** safely and clearly.

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

interface Swimmable {
    void swim();
}
```

```java
class Duck implements Flyable, Swimmable {
    public void fly() {
        System.out.println("Duck is flying");
    }

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

    public static void main(String[] args) {
        Duck d = new Duck();
        d.fly();    // Output: Duck is flying
        d.swim();   // Output: Duck is swimming
    }
}
```

Flyable and **Swimmable** are **interfaces** declaring methods. **Duck** class **implements** both **interfaces**, providing implementations for **fly()** and **swim()**. This demonstrates **multiple inheritance** of behavior signatures via **interfaces**. If **Java** allowed multiple **class** inheritance, conflicts could arise. **Interfaces** avoid that by having no method body to conflict.