Java Unit - 3

Inheritance and Polymorphism : Inheritance, Method overriding, super keyword, Object class and overriding toString() and equals() method, Using super and final keywords, instanceof operator

Abstract Class and Interface : Abstract method and abstract class, Interfaces, static and default methods.

Inheritance and Polymorphism

Inheritance

Inheritance is a fundamental concept of Object-Oriented Programming (OOP) in Java that allows **a class to inherit properties (fields) and behavior (methods) from another class.** It helps in **code reusability**, **method overriding**, and creating a hierarchical classification of objects.

The class that **inherits** from another class is called the **subclass (child class)**, and the class from which it inherits is called the **superclass (parent class)**.

Key Benefits of Inheritance

- **Code Reusability**: Common properties and methods can be defined in a parent class and reused in child classes.
- **Method Overriding**: Allows a subclass to provide a specific implementation of a method already defined in the parent class.
- **Polymorphism Support**: Helps in achieving dynamic method dispatch (runtime polymorphism).
- **Better Organization**: Creates a logical hierarchy of classes in a program.

Types of Inheritance in Java

Java supports different types of inheritance:

1. **Single Inheritance**
2. **Multilevel Inheritance**
3. **Hierarchical Inheritance**
4. **Multiple Inheritance using Interfaces**

Note: Java does not support **Multiple Inheritance** using classes to avoid ambiguity/diamond problem , but it can be achieved using **interfaces**.

---

1. Single Inheritance

Definition: In Single Inheritance, a child class inherits from a single parent class.

Example:

```
class Vehicle {                                          Ummed Singh
    String brand = "Toyota";                                  ByteXL
    void honk() {
        System.out.println("Vehicle is honking...");
    }
}

// Child class (Sub class)
class Car extends Vehicle {
    int speed = 100;
    void display() {
        System.out.println("Brand: " + brand);
        System.out.println("Speed: " + speed + " km/h");
    }
}

// Main class
public class SingleInheritanceExample {
    public static void main(String[] args) {
        Car myCar = new Car();
        myCar.display();
        myCar.honk();
    }
}
Output:
Brand: Toyota
Speed: 100 km/h
Vehicle is honking...
```

Real-World Use Case:

💡 **Banking System (User Account Inheritance)**

- **Parent Class**: `BankAccount` (common properties like balance, account number)
- **Child Class**: `SavingsAccount`, `CurrentAccount`

```
class BankAccount {                                    Ummed Singh
    String accountHolder;
    double balance;

    BankAccount(String accountHolder, double balance) {
        this.accountHolder = accountHolder;
        this.balance = balance;
    }

    void deposit(double amount) {
        balance += amount;
        System.out.println(amount + " deposited. New balance: " +
balance);
    }
}
// Child class inheriting from BankAccount
class SavingsAccount extends BankAccount {
    SavingsAccount(String accountHolder, double balance) {
        super(accountHolder, balance);
    }
    void withdraw(double amount) {
        if (balance >= amount) {
            balance -= amount;
            System.out.println(amount + " withdrawn. Remaining
balance: " + balance);
        } else {
            System.out.println("Insufficient balance.");
        }
    }
}

// Main class to test the implementation
public class BankingSystem {
    public static void main(String[] args) {
```

```
        SavingsAccount myAccount = new SavingsAccount("Ummed
Singh", 500.0);
        myAccount.deposit(200);
        myAccount.withdraw(100);
        myAccount.withdraw(700);
    }
}
```

Problem Without Inheritance

If every account type (`SavingsAccount`, `CurrentAccount`) defines balance and deposit separately, it leads to duplicate code.

Solution Through Inheritance

By making `BankAccount` a superclass, all account types can inherit common properties, avoiding redundancy.

---

2. Multilevel Inheritance

Definition: In Multilevel Inheritance, a class inherits from another class, which in turn is inherited by another class.

Example:

```
class Animal {                                          Ummed Singh
    void eat() {
        System.out.println("This animal eats food.");
    }
}
// Derived class
class Mammal extends Animal {
    void walk() {
        System.out.println("Mammals can walk.");
    }
}
// Derived from Mammal
class Human extends Mammal {
    void speak() {
        System.out.println("Humans can speak.");
```

```
    }
}
// Main class
public class MultilevelInheritanceExample {
    public static void main(String[] args) {
        Human human = new Human();
        human.eat();
        human.walk();
        human.speak();
    }
}

Output:
This animal eats food.
Mammals can walk.
Humans can speak.
```

Real-World Use Case:

💡 **Educational Management System**

- **Parent Class**: `Person` (common properties like name, age)
- **Child Class**: `Student` (adds roll number)
- **Grandchild Class**: `EngineeringStudent` (specializes further)

Problem Without Inheritance

Redundant fields for each role like `name`, `age`, `gender`.

Solution Through Inheritance

By inheriting `Person`, `Student`, and `EngineeringStudent`, we create a structured hierarchy.

```
class Person {                                                    Ummed Singh

    String name;

    Person(String name) {

        this.name = name;
```

```
    }

    void displayPersonInfo() {
        System.out.println("Name: " + name);
    }
}

// Child class inheriting from Person
class Student extends Person {
    int rollNumber;

    Student(String name, int rollNumber) {
        super(name);
        this.rollNumber = rollNumber;
    }

    void displayStudentInfo() {
        displayPersonInfo(); // Calling parent class method
        System.out.println("Roll Number: " + rollNumber);
    }
}

// Grandchild class inheriting from Student
class EngineeringStudent extends Student {
    String specialization;

    EngineeringStudent(String name, int rollNumber, String
specialization) {
        super(name, rollNumber);
```

```
        this.specialization = specialization;
    }
    void displayEngineeringStudentInfo() {
        displayStudentInfo(); // Calling student class method
        System.out.println("Specialization: " + specialization);
    }
}
// Main class to test the hierarchy
public class EducationSystem {
    public static void main(String[] args) {
        EngineeringStudent engStudent = new
EngineeringStudent("Ummed Singh", 1001, "Computer Science");
        engStudent.displayEngineeringStudentInfo();
    }
}
```

---

3. Hierarchical Inheritance

Definition: In Hierarchical Inheritance, multiple child classes inherit from a single parent class.

Example:

```
// Parent class                                          Ummed Singh
class Shape {
    void area() {
        System.out.println("Calculating area...");
    }
}
```

```java
// Child class 1
class Circle extends Shape {
    double radius = 5;
    void area() {
        System.out.println("Area of Circle: " + (3.14 * radius *
radius));
    }
}

// Child class 2
class Rectangle extends Shape {
    double length = 4, width = 6;
    void area() {
        System.out.println("Area of Rectangle: " + (length *
width));
    }
}

// Main class
public class HierarchicalInheritanceExample {
    public static void main(String[] args) {
        Circle c = new Circle();
        c.area();
        Rectangle r = new Rectangle();
        r.area();
    }
}
```

Real-World Use Case:

💡 **University Management System**

- **Parent Class**: `LpuCollege`(common variable like `director name`)
- **Child Classes**: `EngDpart, ManagementDepart`

Problem Without Inheritance

Each display method must implement `directorName` separately.

Solution Through Inheritance

Create a `LpuCollege` base class to enforce consistency.

```
class LpuCollege{                                          Ummed Singh
   String directorName = "Mr Dr.Ashok Kumar Mittal Sir";
}
class EngineeringDepart extends LpuCollege{
   String hod = "";
   void display(){
       System.out.println(directorName);
   }
}


class ManagementDepartment extends LpuCollege{
   void display(){
       System.out.println(directorName);
   }
}
public class LpuManagementSystem {
   public static void main(String[] args) {
       ManagementDepartment obj = new ManagementDepartment();
       obj.display();
   }
```

```
}
```

---

4. Multiple Inheritance using Interfaces

Java does **not support multiple inheritance** with classes to avoid ambiguity but allows it with **interfaces**.

Example:

```
interface Printable {                                    Ummed Singh
    void print();
}

interface Showable {
    void show();
}

// Multiple inheritance through interfaces
class Document implements Printable, Showable {
    public void print() {
        System.out.println("Printing document...");
    }

    public void show() {
        System.out.println("Showing document...");
    }
}

public class MultipleInheritanceExample {
    public static void main(String[] args) {
        Document doc = new Document();
        doc.print();
        doc.show();
    }
}
```

Real-World Use Case:

💡 Messaging System

- **Interface 1**: `Sendable` (defines `send()`)
- **Interface 2**: `Receivable` (defines `receive()`)
- **Class**: `Email` (implements both interfaces)

Problem Without Inheritance

Emails and SMS need to implement `send()` and `receive()` separately.

Solution Through Inheritance

A single class implements both interfaces for **code reusability**.

```
// Interface for sending messages                    Ummed Singh
interface Sendable {
    void send(String message);
}
// Interface for receiving messages
interface Receivable {
    void receive();
}
// Class implementing both interfaces
class Email implements Sendable, Receivable {
    private String inboxMessage;
    @Override
    public void send(String message) {
        System.out.println("Sending Email: " + message);
        inboxMessage = message; // Simulating sending an email
    }
```

```
    @Override
    public void receive() {
        if (inboxMessage != null) {
            System.out.println("Received Email: " +
inboxMessage);
        } else {
            System.out.println("No new emails.");
        }
    }
}
// Main class to test the implementation
public class MessagingSystem {
    public static void main(String[] args) {
        Email email = new Email();
        email.send("Hello, this is a test email.");
        email.receive();
    }
}
```

Diamond Problem in Java and Its Solution

The **diamond problem** occurs in programming languages that support **multiple inheritance**, leading to ambiguity when a class inherits from multiple classes that have methods with the same name.

Java **avoids this problem** by **not allowing multiple inheritance with classes** but allows it with **interfaces**.

Diamond Problem in Multiple Inheritance (Not Allowed in Java)

Problem Example (If Java Allowed Multiple Inheritance)

```
// Parent Class 1                                          Ummed Singh
class A {
    void show() {
        System.out.println("A's show method");
    }
}

// Parent Class 2
class B {
    void show() {
        System.out.println("B's show method");
    }
}

// Child Class trying to inherit both A and B (Not Allowed in
Java)
class C extends A, B { // ❌ ERROR: Java does not support
multiple class inheritance
    public static void main(String[] args) {
        C obj = new C();
        obj.show(); // ❓ Ambiguity: Which show() method to call?
A's or B's?
    }
}
```

Problem:

- If Java allowed multiple inheritance with classes, class `C` would **inherit the `show()` method from both `A` and `B`**, causing **ambiguity**.

Solution: Use Interfaces in Java

Java solves the diamond problem by **allowing multiple inheritance using interfaces**, where ambiguity is resolved explicitly.

Example Using Interfaces (Allowed in Java)

```
// Interface 1                                                    Ummed Singh
interface A {
    default void show() {
        System.out.println("A's show method");
    }
}

// Interface 2
interface B {
    default void show() {
        System.out.println("B's show method");
    }
}

// Class implementing both interfaces
class C implements A, B {
    // Overriding show() method to resolve ambiguity
    @Override
    public void show() {
        System.out.println("Resolving ambiguity...");
        A.super.show(); // Explicitly calling A's show()
        B.super.show(); // Explicitly calling B's show()
    }

    public static void main(String[] args) {
        C obj = new C();
        obj.show();
    }
}

Output:
Resolving ambiguity...
```

```
A's show method
B's show method
```

How This Works in Java

1. **Interfaces allow default methods** (Java 8+) to define behavior.
2. If multiple interfaces have the **same method**, Java forces the child class to **override it** and specify which version to use.
3. The child class can use `A.super.methodName()` or `B.super.methodName()` to resolve ambiguity explicitly.

---

1. What is Polymorphism?

Polymorphism is a core concept in Object-Oriented Programming (OOP) that **allows a single interface or a method to be used for different types.** It enables code reusability and flexibility, making it easier to manage and extend.

Types of Polymorphism in Java:

1. **Compile-Time Polymorphism/static type (Method Overloading):** Method Overloading allows multiple methods in the same class with the same name but different parameters. The compiler determines which method to invoke based on the method signature.
2. **Run-Time Polymorphism/dynamic type (Method Overriding)**

Run-Time Polymorphism (Method Overriding)

Method Overriding allows a subclass to provide a specific implementation of a method that is already defined in its superclass. This is achieved using **dynamic method dispatch**, where the method call is resolved at runtime.

Example of Method Overriding in a Real-World Software (Ride Booking App like Uber):

```
// Parent class                                                    Ummed Singh
class Ride {
    void calculateFare(int distance) {
        System.out.println("Ride fare for " + distance + " km is
calculated.");
    }
}
// Child class for Economy ride
class EconomyRide extends Ride {
    @Override
    void calculateFare(int distance) {
        System.out.println("Economy ride fare for " + distance +
" km: $" + (distance * 2));
    }
}

// Child class for Luxury ride
class LuxuryRide extends Ride {
    @Override
    void calculateFare(int distance) {
        System.out.println("Luxury ride fare for " + distance + "
km: $" + (distance * 5));

    }
}

// Main application class
public class RideBookingAppOverriding {
    public static void main(String[] args) {
        Ride myRide; // Reference variable of parent class

        myRide = new EconomyRide();
        myRide.calculateFare(10); // Calls EconomyRide's method
```

```
        myRide = new LuxuryRide();
        myRide.calculateFare(10); // Calls LuxuryRide's method
    }
}
```

Advantages of Polymorphism in Java

✅ **Code Reusability** – We can write a common method with multiple implementations.
✅ **Scalability** – Easy to add new features (new ride types in the Uber example).
✅ **Maintainability** – Clean and structured code with better separation of concerns.

`super` Keyword

The `super` keyword in Java is used to refer to the **parent class** (superclass) of a subclass. It is primarily used for the following purposes:

1. **Access Parent Class Methods** (Method Overriding): When a subclass overrides a method from its parent class, the overridden method can still be accessed using `super.methodName()`.
2. **Call Parent Class Constructor:** When a subclass object is created, the parent class constructor runs before the subclass constructor or It must be the first statement in the subclass constructor. The **super()** keyword allows us to explicitly call a specific parent constructor.
3. **Access Parent Class Variables** (When Hidden by Subclass): If a subclass defines a variable with the same name as its parent class, it **hides** the parent's variable. The `super` keyword allows access to the parent class variable.

   `super` is commonly used to eliminate ambiguity between parent and child class members when they have the same name.

Example:

```
class ParentSuper {                                    Ummed Singh
    String name = "Parent";
```

```
    ParentSuper(){
        System.out.println("Executed the ParentClass
constructor");
    }
    void display() {
        System.out.println("This is the parent class.");
    }
}
class ChildSuper extends ParentSuper {
    String name = "Child";

    ChildSuper(){
        super();

    }

    void display() {


        System.out.println("This is the child class.");
    }
    void show() {
        System.out.println("Name in Child class: " + name);
        System.out.println("Name in Parent class: " +
super.name);
        display();
// Calls Child's display method
        super.display(); // Calls Parent's display method
    }
}
public class SuperDemo {
    public static void main(String[] args) {
        ChildSuper child = new ChildSuper();
        child.show();
    }
}
```

Introduction to Object Class

In Java, the `Object` class is the **superclass of all classes**. Every class in Java **implicitly** extends the `Object` class (except when explicitly extending another class).

Key Points:

- The `Object` class is part of `java.lang` package.
- It provides basic methods that all Java objects inherit.
- It allows Java to implement key object-oriented programming concepts like **polymorphism** and **inheritance**.

---

2. Methods of Object Class

The `Object` class provides several important methods that can be overridden in subclasses for customization.

List of Important Methods in the Object Class

Method	Description
`equals(Object obj)`	Checks if two objects are equal.
`hashCode()`	Returns a hash code value for the object.
`toString()`	Returns a string representation of the object.
`clone()`	Creates a duplicate (shallow copy) of an object.

`finalize()`	Called by the garbage collector before destroying an object.
`getClass()`	Returns the runtime class of the object.
`wait()`, `notify()`, `notifyAll()`	Used in thread synchronization.

Understanding Object Class Methods with Examples

(i) equals() Method

The `equals()` method compares two objects for equality. By default, it uses the `==` operator, which checks **memory address (reference equality)**.

Example Without Overriding `equals()`

```
class Car {                                                    Ummed Singh
    String model;

    Car(String model) {
        this.model = model;
    }
}

public class ObjectEqualsExample {
    public static void main(String[] args) {
        Car car1 = new Car("Tesla");
        Car car2 = new Car("Tesla");

        System.out.println(car1.equals(car2)); // false
(different memory locations)
    }
}
```

Overriding `equals()` for Logical Comparison

```
class Bike {                                                    Ummed Singh
    String model;
    Bike(String model) {
        this.model = model;
    }
    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true; // Same reference
        if (obj == null || getClass() != obj.getClass()) return
false;
        Bike bike = (Bike) obj;
        return model.equals(bike.model);
    }
}
public class ObjectEqualsExample1 {
    public static void main(String[] args) {
        Bike bike1 = new Bike("Tesla");
        Bike bike2 = new Bike("Tesla");

        System.out.println(bike1.equals(bike2)); // true (based
on model name)
    }
```

```
}
```

(ii) hashCode() Method

The `hashCode()` method returns an integer that represents the **memory address** or a **logical value**. It is important when storing objects in **hash-based collections (e.g., HashMap, HashSet)**.

Example Without Overriding `hashCode()`

```
class BMW {                                                    Ummed Singh
    String model;
    BMW(String model) {
        this.model = model;
    }
}
public class ObjectHashCodeExample {
    public static void main(String[] args) {
        BMW bmw1 = new BMW("BMW");
        BMW bmw2 = new BMW("BMW");

        System.out.println(bmw1.hashCode()); // Different hash
codes
        System.out.println(bmw2.hashCode()); // Different hash
codes
    }
}
```

Overriding `hashCode()` for Consistent Hashing

```
class Car3 {                                                    Ummed Singh
    String model;
    Car3(String model) {
        this.model = model;
    }
    @Override
    public int hashCode() {
        return model.hashCode();
    }
}

public class ObjectHashCodeExample1 {
    public static void main(String[] args) {
        Car3 car1 = new Car3("BMW");
        Car3 car2 = new Car3("BMW");

        System.out.println(car1.hashCode()); // Same hash code
        System.out.println(car2.hashCode()); // Same hash code
    }
}
```

Relation Between `equals()` and `hashCode()`

- If `equals()` returns `true`, then `hashCode()` must return the same value.
- If `equals()` returns `false`, `hashCode()` can be different.

(iii) toString() Method

The `toString()` method returns a **string representation** of an object.

Default `toString()` Implementation

```
class Thar {                                                    Ummed Singh

}



public class ObjectToStringExample {

   public static void main(String[] args) {

        Thar thar = new Thar();

        System.out.println(thar.toString()); // Output:
Thar@1b28cdfa (memory reference)

   }

}
```

Overriding `toString()` for Better Output

```
class Mahindra {                                                 Ummed Singh

   String model;
```

```
    Mahindra(String model) {
        this.model = model;
    }
    @Override
    public String toString() {
        return "Car Model: " + model;
    }
}
public class ObjectToStringExample1 {
    public static void main(String[] args) {
        Mahindra mahindra = new Mahindra("Mahindra");
        System.out.println(mahindra.toString()); // Output: Car
Model: Mahindra
    }
}
```

(iv) clone() Method (Shallow Copy)

The `clone()` method creates a **copy of an object**. It requires the **`Cloneable`** interface.

Example of `clone()`

```
class Car implements Cloneable {                    Ummed Singh
    String model;
```

```
    Car(String model) {
        this.model = model;
    }

    @Override
    protected Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}

public class ObjectCloneExample {
    public static void main(String[] args) throws
CloneNotSupportedException {
        Car car1 = new Car("Tesla");
        Car car2 = (Car) car1.clone();

        System.out.println(car1.model); // Tesla
        System.out.println(car2.model); // Tesla
    }
}
```

(v) finalize() Method

The `finalize()` method is called before an object is garbage collected.

Example of `finalize()`

```
class Car {                                                    Ummed Singh
   @Override
   protected void finalize() throws Throwable {
        System.out.println("Car object is destroyed");
   }
}


public class ObjectFinalizeExample {
   public static void main(String[] args) {
        Car car = new Car();
        car = null;
        System.gc(); // Suggest garbage collection
   }
}
```

`instanceof` Operator in Java

The `instanceof` operator is a **binary operator** in Java used to **test whether an object is an instance of a specific class or a subclass of that class**. It is used to check **type compatibility** at runtime.

Syntax:

`object instanceof ClassName`

- **Returns `true`** → If the object is an instance of the specified class or its subclass.
- **Returns `false`** → If the object is not an instance of the specified class.

Example of `instanceof` Operator:

```
class Animal1 {}                                        Ummed Singh
class Dog extends Animal1 {}

public class InstanceofExample {
   public static void main(String[] args) {
       Dog dog = new Dog();
       System.out.println(dog instanceof Dog);     // true
       System.out.println(dog instanceof Animal1); // true
       System.out.println(dog instanceof Object); // true
   }
}
```

✅ `instanceof` confirms that **dog** is an instance of **Dog**, **Animal**, and **Object**.

Use `instanceof` Operator

- **Avoid ClassCastException** → Prevents invalid type casting.
- **Supports Runtime Type Checking** → Useful when dealing with polymorphism.
- **Ensures Safe Downcasting** → Checks before performing a downcast.

Final Keyword

The `final` keyword in Java is used to apply restrictions on variables, methods, and classes. It ensures that the variable's value cannot be changed, the method cannot be overridden, and the class cannot be inherited.

1. Final Variable

A **`final`** variable in Java is a constant; once assigned, its value cannot be changed.

Syntax:

```
final int MAX_USERS = 100;
```

Real-World Example: Database Connection URL

In a web application, we often define database connection details as constants since they should not be modified.

```
public class DatabaseConfig {                                    Ummed Singh

   public static final String DB_URL =
"jdbc:mysql://localhost:3306/mydb";

   public static final String USERNAME = "root";

   public static final String PASSWORD = "password";

}
```

Here, **`DB_URL`**, **`USERNAME`**, and **`PASSWORD`** should remain unchanged throughout the application.

---

2. Final Method

A `final` method cannot be overridden by subclasses. This is useful when we want to prevent changes to the core logic of a method.

Syntax:

```
class Parent {                                                   Ummed Singh
```

```
    public final void display() {
        System.out.println("This is a final method.");
    }
}
class Child extends Parent {
    // Cannot override display() method
}
```

---

3. Final Class

A `final` class cannot be inherited. This is useful for security and utility classes where modification is undesirable.

Syntax:

```
final class Utility {                                    Ummed Singh
    public static void printMessage(String msg) {
        System.out.println(msg);
    }
}
class ExtendedUtility extends Utility { } // This would cause an
error
```

Real-World Example: Java's Built-in String Class

In Java, the `String` class is `final` to prevent subclassing and potential security risks.

```
public final class String {                                    Ummed Singh

    // Internal implementation

}
```

If `String` were not `final`, anyone could extend it and modify its behavior, which could lead to security vulnerabilities.

Use case of `final` in Software Development

- **`final` variables**: For constants like database URLs, API keys, or mathematical constants (`PI = 3.14159`).
- **`final` methods**: When you want to ensure that core logic (e.g., logging, authentication) remains unchanged.
- **`final` classes**: For utility or security-sensitive classes (e.g., `String`, `Math`).

Abstract class and Interface

Abstraction in Java

Abstraction is one of the core concepts of Object-Oriented Programming (OOP) in Java. It is the process of **hiding implementation details and exposing only the necessary parts to the user.** Abstraction helps in reducing complexity and increasing code maintainability.

Java achieves abstraction using:

1. **Abstract Classes**
2. **Interfaces**

---

1. Abstract Classes

An **abstract class** in Java is a class that cannot be instantiated. It can contain both abstract methods (methods without implementation) and concrete methods (methods with implementation).

Example of Abstract Class

```
abstract class Vehicle {                                    Ummed Singh
    abstract void start(); // Abstract method (no implementation)

    void stop() { // Concrete method (with implementation)
        System.out.println("Vehicle is stopping.");
    }
}

class Car extends Vehicle {
    @Override
    void start() {
        System.out.println("Car is starting with a key.");
    }
}

class Bike extends Vehicle {
    @Override
    void start() {
        System.out.println("Bike is starting with a self-start
button.");
    }
}

public class AbstractionExample {
    public static void main(String[] args) {
        Vehicle myCar = new Car();
        myCar.start(); // Outputs: Car is starting with a key.
        myCar.stop();  // Outputs: Vehicle is stopping.

        Vehicle myBike = new Bike();
        myBike.start(); // Outputs: Bike is starting with a
self-start button.
    }
}
```

2. Interfaces

An **interface** in Java is a completely abstract class that only contains abstract methods (before Java 8). From Java 8 onwards, interfaces can also have default and static methods.

Example of Interface

```
interface Payment {                                    Ummed Singh
    void makePayment(); // Abstract method
}

class CreditCardPayment implements Payment {
    @Override
    public void makePayment() {
        System.out.println("Payment made using Credit Card.");
    }
}

class PayPalPayment implements Payment {
    @Override
    public void makePayment() {
        System.out.println("Payment made using PayPal.");
    }
}

public class InterfaceExample {
    public static void main(String[] args) {
        Payment payment1 = new CreditCardPayment();
        payment1.makePayment(); // Outputs: Payment made using
Credit Card.

        Payment payment2 = new PayPalPayment();
        payment2.makePayment(); // Outputs: Payment made using
PayPal.
    }
}
```

Real-World Software Example: Banking System

Scenario: Online Payment System

Imagine an **Online Banking System** where different payment methods (Credit Card, PayPal, UPI) are available. Instead of implementing separate payment logic in multiple places, we define an **interface (Abstraction)** called `PaymentGateway`. Any new payment method can implement this interface without affecting existing code.

Implementation:

```
// Abstracting the payment system using an interface
interface PaymentGateway {
    void processPayment(double amount);
}

// Concrete implementations
class CreditCard implements PaymentGateway {
    @Override
    public void processPayment(double amount) {
        System.out.println("Processing Credit Card Payment of $"
+ amount);
    }
}

class PayPal implements PaymentGateway {
    @Override
    public void processPayment(double amount) {
        System.out.println("Processing PayPal Payment of $" +
amount);
    }
}

class UPI implements PaymentGateway {
    @Override
    public void processPayment(double amount) {
        System.out.println("Processing UPI Payment of $" +
amount);
    }
}

// Main Class
public class PaymentSystem {
```

```
    public static void main(String[] args) {          Ummed Singh
        PaymentGateway payment1 = new CreditCard();
        payment1.processPayment(250.00);

        PaymentGateway payment2 = new PayPal();
        payment2.processPayment(100.50);

        PaymentGateway payment3 = new UPI();
        payment3.processPayment(50.00);
    }
}
```

Another Example:

```
abstract class PaymentPoly {
    double amount;                                    Ummed Singh
    PaymentPoly(double amount) {
        this.amount = amount;
    }
    abstract void processPayment(); // Abstract method to be
implemented by subclasses

}

// Child class: CreditCardPayment
class CreditCardPaymentPoly extends PaymentPoly {
    String cardNumber;
    CreditCardPaymentPoly(double amount, String cardNumber) {
        super(amount);
        this.cardNumber = cardNumber;
    }
    @Override
    void processPayment() {
        System.out.println("Processing Credit Card Payment of $"
+ amount + " using card: " + cardNumber);
    }
}

// Child class: UPIPayment
class UPIPaymentPoly extends PaymentPoly {
    String upiId;
```

```
    UPIPaymentPoly(double amount, String upiId) {
        super(amount);
        this.upiId = upiId;
    }
    @Override
    void processPayment() {
        System.out.println("Processing UPI Payment of $" + amount
+ " using UPI ID: " + upiId);
    }
}
// Main class to test the implementation
public class ECommercePaymentSystemPoly extends Object {
    public static void main(String[] args) {
        // Using a list of Payment objects instead of separate
object creation
        List<PaymentPoly> payments = new ArrayList<>();
        payments.add(new CreditCardPaymentPoly(100.0,
"1234-5678-9876-5432"));
        payments.add(new UPIPaymentPoly(50.0, "ummed@upi"));
        // Processing all payments dynamically using a loop
        for (PaymentPoly payment : payments) {
            payment.processPayment();
        }
    }
}
```

Advantages of Abstraction in This Example

1. **Code Maintainability:** New payment methods can be added without modifying existing code.
2. **Scalability:** If a new payment method like "Crypto Payment" is introduced, we can create a new class implementing `PaymentGateway` without changing existing classes.
3. **Security:** The internal logic of each payment method is hidden from the users, preventing unauthorized changes.

- **Abstraction** in Java allows us to define a blueprint and hide unnecessary details.

- **Abstract classes** allow partial abstraction, while **interfaces** provide full abstraction.
- In real-world applications like banking, e-commerce, and authentication systems, abstraction helps improve modularity and maintainability.

Difference Between Abstract Class and Interface

Both **Abstract Classes** and **Interfaces** are used to achieve **abstraction**, but they have key differences in how they work and when they should be used.

Feature	Abstract Class	Interface
Definition	A class that cannot be instantiated and may contain abstract and concrete methods.	A completely abstract type that only defines method signatures (until Java 8) and allows multiple implementations.
Method Types	Can have **both** abstract methods (without body) and concrete methods (with body).	Can have only **abstract methods** (before Java 8). From Java 8, can have **default** and **static** methods.
Method Implementation	Can provide partial implementation.	Cannot provide method implementation (except default/static methods in Java 8+).
Access Modifiers	Methods can have **any** access modifier (public, private, protected, default).	Methods are **public** by default.
Variable Types	Can have instance variables (fields) with any access modifier.	Only **public, static, final** constants (no instance variables).
Constructors	Can have constructors.	Cannot have constructors.
Multiple Inheritance	Supports single inheritance only (a class can extend only one abstract class).	Supports multiple inheritance (a class can implement multiple interfaces).
Use Case	Used when classes share common behavior, but require some methods to be implemented by subclasses.	Used when different classes need to follow the same contract but may have completely different implementations.

Static and Default Methods in Abstraction (Java 8 and Above)

In Java, abstraction is mainly achieved using **abstract classes** and **interfaces**. However, before Java 8, interfaces could only have abstract methods (methods without implementations). Java 8 introduced **default** and **static** methods in interfaces to enhance flexibility and backward compatibility.

---

1. Static Methods in Interfaces

A **static method** in an interface belongs to the interface itself, not to the instances of the implementing classes. These methods cannot be overridden by implementing classes.

Why Are Static Methods Introduced?

- To avoid utility/helper method duplication across implementing classes.
- To provide utility functions related to an interface without requiring an implementing class.

Example

Let's say we have an **E-commerce application** where multiple payment gateways (HDFC, SBI card, etc.) are integrated. Each payment gateway implements a common `PaymentGatewayDemo` interface. A utility method for validating credit card numbers can be implemented as a **static method** inside the interface.

```
interface PaymentGatewayDemo {                                  Ummed Singh
    void processPayment(double amount);

    // Static method for card validation
    static boolean validateCard(String cardNumber) {
        return cardNumber.matches("\\d{16}"); // Checks if card
number is 16 digits
    }
}

class HDFC implements PaymentGateway {
    @Override
    public void processPayment(double amount) {
        System.out.println("Processing payment through HDFC: $" +
amount);
    }
```

```
}

class SBI implements PaymentGateway {
    @Override
    public void processPayment(double amount) {
        System.out.println("Processing payment through SBI: $" +
amount);
    }
}

// Testing static method
public class PaymentTestStatic {
    public static void main(String[] args) {
        boolean isValid =
PaymentGatewayDemo.validateCard("1234567812345678");
        System.out.println("Is card valid? " + isValid);
    }
}
```

📌 **Key Benefit:** The `validateCard()` method can be used directly without needing an instance of `PaymentGatewayDemo`, preventing duplicate validation logic in multiple classes.

---

2. Default Methods in Interfaces

A **default method** provides a default implementation inside an interface. It allows new methods to be added to interfaces without breaking existing implementations.

Why Are Default Methods Introduced?

- To **extend interfaces without breaking existing classes** that implement them.
- To provide **common behavior** that can be overridden when necessary.

Example

Consider an **E-commerce Discount System** where different vendors apply different discount rules. Instead of forcing every vendor to implement a discount calculation method, we can provide a **default discount calculation** in the interface and allow vendors to override it when needed.

```
interface DiscountService {                                    Ummed Singh
    // Default method providing a common discount calculation
    default double calculateDiscount(double price) {
        return price * 0.10; // Default 10% discount
    }
}

class Pnb implements DiscountService {
    // Uses default discount calculation
}

class Icici implements DiscountService {
    @Override
    public double calculateDiscount(double price) {
        return price * 0.15; // Walmart gives a 15% discount
    }
}

public class DiscountTestDefault {
    public static void main(String[] args) {
        DiscountService pnb = new Pnb();
        DiscountService icici = new Icici();

        System.out.println("PNB Discounted Price: $" +
pnb.calculateDiscount(100));
        System.out.println("ICICI Discounted Price: $" +
icici.calculateDiscount(100));
    }
}
```

📌 **Key Benefit:**

- `Pnb` uses the default 10% discount without extra code.
- `Icici` overrides the method to apply a different discount.