Case Study Title: E-Commerce Product Management System (OOP in Java)

□ Objective:

Implement and understand core OOP concepts by simulating a real-world e-commerce product and order management system.

Scenario:

An e-commerce company wants to manage different types of products (Electronics, Clothing), customer orders, and payment processing. They want a system where:

- All products share common features but have category-specific details.
- Orders are associated with products and customers.
- Payment should be processed via different methods (UPI, Credit Card, COD).
- Discounts can be applied using lambda expressions.

OOP Concepts to Practice:

Concept	Implementation in the Project
Class & Object	Product, Customer, Order
Inheritance	`Electronics`, `Clothing` extends `Product`
Abstraction	`Payment` is an abstract class
Interface	`Discountable` interface
Polymorphism	Multiple payment types override `processPayment()`
Lambda Expression	Apply discount logic

Class Design Overview:

1. Abstract Class: Product

```
public abstract class Product {
    protected String id;
    protected String name;
    protected double price;

    public Product(String id, String name, double price) {
        this.id = id;
        this.name = name;
        this.price = price;
    }

    public abstract void displayDetails();

    public double getPrice() {
        return price;
    }
}
```

---

2. Subclasses: Electronics & Clothing

```
public class Electronics extends Product {
    private String brand;
    private int warrantyInMonths;

    public Electronics(String id, String name, double price, String brand,
int warrantyInMonths) {
        super(id, name, price);
        this.brand = brand;
        this.warrantyInMonths = warrantyInMonths;
    }

    @Override
    public void displayDetails() {
        System.out.println("Electronics: " + name + " | Brand: " + brand + "
| Warranty: " + warrantyInMonths + " months | Price: " + price);
    }
}

public class Clothing extends Product {
    private String size;
    private String fabric;

    public Clothing(String id, String name, double price, String size, String
fabric) {
        super(id, name, price);
        this.size = size;
        this.fabric = fabric;
    }
    @Override
    public void displayDetails() {
        System.out.println("Clothing: " + name + " | Size: " + size + " |
Fabric: " + fabric + " | Price: " + price);
    }
}
```

---

3. Interface: Discountable

```
@FunctionalInterface
public interface Discountable {
    double applyDiscount(double price);
}
```

---

4. Abstract Class: Payment

```
public abstract class Payment {
    protected double amount;

    public Payment(double amount) {
        this.amount = amount;
    }

    public abstract void processPayment();
}
```

---

5. Subclasses: UpiPayment, CardPayment

```
public class UpiPayment extends Payment {
    private String upiId;

    public UpiPayment(double amount, String upiId) {
        super(amount);
        this.upiId = upiId;
    }

    @Override
    public void processPayment() {
        System.out.println("Paid ₹" + amount + " via UPI: " + upiId);
    }
}

public class CardPayment extends Payment {
    private String cardNumber;

    public CardPayment(double amount, String cardNumber) {
        super(amount);
        this.cardNumber = cardNumber;
    }
    @Override
    public void processPayment() {
        System.out.println("Paid ₹" + amount + " using Card ending with: " +
cardNumber.substring(cardNumber.length()-4));
    }
}
```

---

6. Customer and Order Classes

```
public class Customer {
    private String name;
    private String email;

    public Customer(String name, String email) {
        this.name = name;
        this.email = email;
    }

    public void displayCustomer() {
        System.out.println("Customer: " + name + " | Email: " + email);
    }
}
import java.util.List;

public class Order {
    private Customer customer;
    private List<Product> products;
    private double totalAmount;

    public Order(Customer customer, List<Product> products) {
        this.customer = customer;
        this.products = products;
        this.totalAmount = calculateTotal();
    }

    private double calculateTotal() {
        return products.stream().mapToDouble(Product::getPrice).sum();
    }

    public void placeOrder(Payment payment, Discountable discountable) {
        customer.displayCustomer();
        products.forEach(Product::displayDetails);

        double discounted = discountable.applyDiscount(totalAmount);
        System.out.println("Total after discount: ₹" + discounted);

        payment.amount = discounted;
        payment.processPayment();
    }
}
```

Sample Main Method (Test Scenario)

```
import java.util.Arrays;

public class ECommerceApp {
    public static void main(String[] args) {
        Product phone = new Electronics("P1001", "iPhone 15", 79999, "Apple",
12);
        Product tshirt = new Clothing("C1002", "Polo T-shirt", 1499, "L",
"Cotton");

        Customer customer = new Customer("Ravi Kumar", "ravi@example.com");

        Order order = new Order(customer, Arrays.asList(phone, tshirt));

        // Lambda Expression for 10% Discount
        Discountable discount = (price) -> price * 0.9;

        // Payment Mode
        Payment payment = new UpiPayment(0, "ravi@upi");

        order.placeOrder(payment, discount);
    }
}
```

- Understand how abstraction and interfaces help build extensible systems.
- Practice inheritance to model specialized product types.
- Apply polymorphism in payment and display logic.
- Use lambda expressions for dynamic discounting.
- Develop a mini real-world application using Java OOP principles.

Tasks:

1. Add more product categories (Books, Furniture).
2. Add new payment methods (CashOnDelivery).
3. Implement order cancellation.
4. Add functionality to update stock quantity.
5. Store and retrieve orders using file handling or database.