**Exercise 1:**

**1.** **Two use cases to demonstrate two behavioural design pattern**

**Use Case 1: Chain of Responsibility Pattern**

**Problem:** In a banking system, we want to create a system that processes a customer's request to deposit or withdraw money. The request should be routed to the correct handler based on the amount of money involved.

**Solution:** We can use the Chain of Responsibility pattern to achieve this

// Abstract Handler interface

public abstract class Handler {

protected Handler nextHandler;

public void setNextHandler(Handler nextHandler) {

this.nextHandler = nextHandler;

}

public abstract void handleRequest(int amount);

}

// Concrete Handler classes

public class ATMHandler extends Handler {

@Override

public void handleRequest(int amount) {

if (amount <= 1000) {

System.out.println("ATM handling: " + amount);

} else {

if (nextHandler != null) {

nextHandler.handleRequest(amount);

}

}

}

}

public class TellerHandler extends Handler {

@Override

public void handleRequest(int amount) {

if (amount > 1000 && amount <= 5000) {

System.out.println("Teller handling: " + amount);

} else {

if (nextHandler != null) {

nextHandler.handleRequest(amount);

}

}

}

}

public class ManagerHandler extends Handler {

@Override

public void handleRequest(int amount) {

if (amount > 5000) {

System.out.println("Manager handling: " + amount);

}

}

}

// Client code

public class BankSystem {

public static void main(String[] args) {

ATMHandler atmHandler = new ATMHandler();

TellerHandler tellerHandler = new TellerHandler();

ManagerHandler managerHandler = new ManagerHandler();

atmHandler.setNextHandler(tellerHandler);

tellerHandler.setNextHandler(managerHandler);

atmHandler.handleRequest(800); // Output: ATM handling: 800

atmHandler.handleRequest(4000); // Output: Teller handling: 4000

atmHandler.handleRequest(6000); // Output: Manager handling: 6000

}

}

**Use Case 2: Iterator Pattern**

**Problem:** In a e-commerce system, we want to iterate over a list of products and display their details.

**Solution:** We can use the Iterator pattern to achieve this.

// Abstract Iterator interface

public interface Iterator {

boolean hasNext();

Product getNext();

}

// Concrete Iterator class

public class ProductIterator implements Iterator {

private Product[] products;

private int position;

public ProductIterator(Product[] products) {

this.products = products;

this.position = 0;

}

@Override

public boolean hasNext() {

return position < products.length;

}

@Override

public Product getNext() {

return products[position++];

}

}

// Concrete Aggregate class

public class ProductList implements Iterable<Product> {

private Product[] products;

public ProductList(Product[] products) {

this.products = products;

}

@Override

public Iterator<Product> iterator() {

return new ProductIterator(products);

}

}

// Client code

public class ECommerceSystem {

public static void main(String[] args) {

Product product1 = new Product("Product 1", 10.99);

Product product2 = new Product("Product 2", 9.99);

Product product3 = new Product("Product 3", 14.99);

ProductList productList = new ProductList(new Product[]{product1, product2, product3});

for (Product product : productList) {

System.out.println(product.getName() + ": " + product.getPrice());

}

}

}