Coding Skill Assessment

Part 1: System Design

Problem Statement:
Design a simplified **e-commerce system** that handles users, products, orders, and payments.

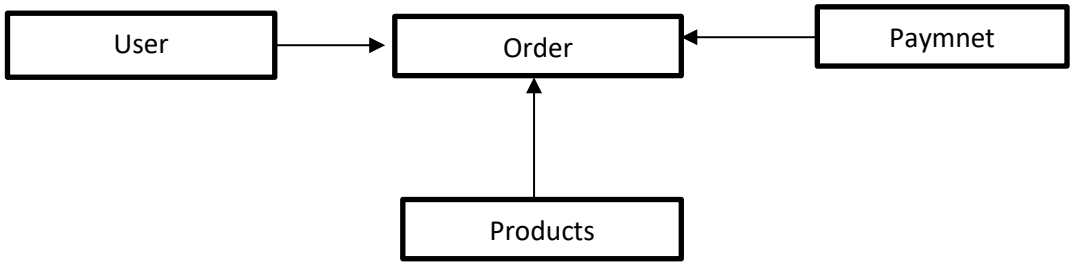
Requirements:

- The system should support multiple users with the ability to create, view, and manage orders.
- Each order can contain multiple products.
- A payment can be made for each order, and an order can have different statuses (e.g., pending, completed, shipped).

Deliverables:

1. **Class Diagram** that outlines the relationships between User, Product, Order, and Payment.
2. Write code stubs for each of the main components, ensuring that relationships (e.g., Order contains multiple Products) are appropriately handled.

Problem Solution:



Explanation

- User can create an Order that contains a list of Products.
- Order keeps track of its status (pending, completed) and links with a Payment.
- Each class has a unique identifier and relationships are managed through lists and objects.

Java Code:

```
import java.util.ArrayList;
import java.util.List;
class Main {
    public static void main(String[] args) {
        // Creating some sample products
        Product product1 = new Product(1, "Laptop", 65000.20);
        Product product2 = new Product(2, "Smartphone", 18000.04);
        Product product3 = new Product(3, "Headphones", 15000.25);
        // Creating a user
        User user1 = new User(1, "Vikas");
        // Creating an order with a list of products
        List<Product> productList1 = new ArrayList<>();
        productList1.add(product1);
        productList1.add(product2);
        // User creates an order
        Order order1 = user1.createOrder(productList1);
        System.out.println("Order created with status: " + order1.getStatus());
        // Adding payment to the order
        Payment payment1 = new Payment(order1, 83000.24); // total amount of products
        order1.addPayment(payment1);
        System.out.println("Payment added. Order status: " + order1.getStatus());
        // Creating another order with more products
        List<Product> productList2 = new ArrayList<>();
        productList2.add(product3);
        Order order2 = user1.createOrder(productList2);
        System.out.println("Second order created with status: " + order2.getStatus());
```

```
        // Adding payment to the second order
        Payment payment2 = new Payment(order2, 15000.25);
        order2.addPayment(payment2);
        System.out.println("Second order payment added. Order status: " + order2.getStatus());
    }
}

class User {
    private int userId;
    private String name;
    private List<Order> orders;

    public User(int userId, String name) {
        this.userId = userId;
        this.name = name;
        this.orders = new ArrayList<>();
    }

    public Order createOrder(List<Product> products) {
        Order order = new Order(this, products);
        this.orders.add(order);
        return order;
    }
}
```

```
class Product {
    private int productId;
    private String name;
    private double price;

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

class Order {
    private static int idCounter = 1;
    private int orderId;
    private User user;
    private List<Product> products;
    private String status;
    private Payment payment;

    public Order(User user, List<Product> products) {
        this.orderId = idCounter++;
        this.user = user;
        this.products = products;
        this.status = "pending";
    }
```

```
    public void addPayment(Payment payment) {
        this.payment = payment;
        this.status = "completed";
    }
    public String getStatus() {
        return status;
    }
}
class Payment {
    private static int idCounter = 1;
    private int paymentId;
    private Order order;
    private double amount;

    public Payment(Order order, double amount) {
        this.paymentId = idCounter++;
        this.order = order;
        this.amount = amount;
    }
}
```

OUTPUT:

```
Order created with status: pending
Payment added. Order status: completed
Second order created with status: pending
Second order payment added. Order status: completed
```

Part 2: Business Logic Implementation

Problem Statement:
You are tasked with implementing an **inventory management system** for a warehouse. The system should be able to track stock levels and manage restocking.

Requirements:

1. Implement a function that:
    - Takes a list of products with their current stock levels and a list of incoming sales orders.
    - Reduces the stock levels based on the orders.
    - If the stock level of any product drops below a certain threshold (e.g., 10 units), an alert should be triggered to restock the item.
2. Implement a function to **restock** items. The function should:
    - Take a list of products that need restocking and their required quantities.
    - Update the stock levels accordingly.

Deliverables:

- Provide the code implementation for the two functions: process_orders() and restock_items().
- Ensure error handling is in place for invalid input (e.g., trying to process an order when the product is out of stock).

<u>Problem Solution:</u>

Explanation:

1. **Products Initialization**: We have three products in the inventory: Laptop, Smartphone, and Headphones with initial stock levels.
2. **Sales Orders**: We created three sales orders:
    - Selling 3 laptops.
    - Selling 5 smartphones.
    - Selling 4 headphones (this will drop the headphone stock below 10, triggering a restock alert).
3. **Restocking**: The system restocks 15 headphones after the sales order processing.

4. **Final Stock Levels**: After processing orders and restocking, the final stock levels for each product are displayed.

Java Code:

```
import java.util.ArrayList;

import java.util.HashMap;

import java.util.List;


public class Main {

    public static void main(String[] args) {

        // Step 1: Initialize the product inventory

        HashMap<Integer, ProductStock> products = new HashMap<>();

        products.put(1, new ProductStock("Laptop", 50));

        products.put(2, new ProductStock("Smartphone", 20));

        products.put(3, new ProductStock("Headphones", 5));


        // Step 2: Create a list of sales orders

        List<SalesOrder> salesOrders = new ArrayList<>();

        salesOrders.add(new SalesOrder(1, 3)); // Selling 3 Laptops

        salesOrders.add(new SalesOrder(2, 5)); // Selling 5 Smartphones

        salesOrders.add(new SalesOrder(3, 4)); // Selling 4 Headphones (will trigger restock alert)


        // Step 3: Process sales orders and reduce stock

        System.out.println("Processing sales orders...");

        InventoryManager.processOrders(products, salesOrders);


        // Step 4: Create a list of restock items

        List<RestockItem> restockItems = new ArrayList<>();
```

```
        restockItems.add(new RestockItem(3, 15)); // Restocking 15 Headphones

        // Step 5: Restock items
        System.out.println("\nRestocking items...");
        InventoryManager.restockItems(products, restockItems);

        // Step 6: Display final stock levels
        System.out.println("\nFinal Stock Levels:");
        products.forEach((id, product) ->
            System.out.println("Product ID: " + id + ", Name: " + product.getName() + ", Stock: "
+ product.getStock()));
    }
}

class InventoryManager {
    public static void processOrders(HashMap<Integer, ProductStock> products,
List<SalesOrder> salesOrders) {
        for (SalesOrder order : salesOrders) {
            int productId = order.getProductId();
            int quantity = order.getQuantity();

            if (!products.containsKey(productId)) {
                throw new IllegalArgumentException("Product ID " + productId + " does not
exist.");
            }

            ProductStock product = products.get(productId);
            if (product.getStock() < quantity) {
```

```
            throw new IllegalArgumentException("Insufficient stock for product ID " +
productId);
        }

        product.reduceStock(quantity);

        if (product.getStock() < 10) {
            System.out.println("Alert: Product " + productId + " stock below threshold,
restocking needed");
        }
    }
  }

  public static void restockItems(HashMap<Integer, ProductStock> products,
List<RestockItem> restockList) {
    for (RestockItem restockItem : restockList) {
        int productId = restockItem.getProductId();
        int quantity = restockItem.getQuantity();

        if (!products.containsKey(productId)) {
            throw new IllegalArgumentException("Product ID " + productId + " does not
exist.");
        }

        products.get(productId).increaseStock(quantity);
    }
  }
}
```

```
class ProductStock {
    private String name;
    private int stock;

    public ProductStock(String name, int stock) {
        this.name = name;
        this.stock = stock;
    }

    public String getName() {
        return name;
    }

    public int getStock() {
        return stock;
    }

    public void reduceStock(int quantity) {
        this.stock -= quantity;
    }

    public void increaseStock(int quantity) {
        this.stock += quantity;
    }
}
```

```
class SalesOrder {
    private int productId;
    private int quantity;

    public SalesOrder(int productId, int quantity) {
        this.productId = productId;
        this.quantity = quantity;
    }

    public int getProductId() {
        return productId;
    }

    public int getQuantity() {
        return quantity;
    }
}

class RestockItem {
    private int productId;
    private int quantity;

    public RestockItem(int productId, int quantity) {
        this.productId = productId;
        this.quantity = quantity;
    }
```

```
    public int getProductId() {
        return productId;
    }
    public int getQuantity() {
        return quantity;
    }
}
```

OUTPUT:

Processing sales orders...

Alert: Product 3 stock below threshold, restocking needed

Restocking items...

Final Stock Levels:

Product ID: 1, Name: Laptop, Stock: 47

Product ID: 2, Name: Smartphone, Stock: 15

Product ID: 3, Name: Headphones, Stock: 16

Part 3: Database Query Handling

Problem Statement:
You are given a relational database schema for an online bookstore with the following tables:

Tables:

Customers (customer_id, name, email)
Books (book_id, title, author, price)
Orders (order_id, customer_id, order_date)
OrderDetails (order_id, book_id, quantity)

Requirements:

1. Write a SQL query to retrieve the top 5 customers who have purchased the most books (by total quantity) over the last year.
2. Write a SQL query to calculate the total revenue generated from book sales by each author.
3. Write a SQL query to retrieve all books that have been ordered more than 10 times, along with the total quantity ordered for each book.

Deliverables:

- Provide the SQL queries for the three requirements.
- Ensure that the queries are optimized for performance, considering indexing where necessary.

1. Top 5 customers who have purchased the most books over the last year (since today is 2024-10-16)

```
SELECT C.customer_id, C.name, SUM(OD.quantity) AS total_books_purchased

FROM Customers C

JOIN Orders O ON C.customer_id = O.customer_id

JOIN OrderDetails OD ON O.order_id = OD.order_id

WHERE O.order_date >= '2023-10-16'

GROUP BY C.customer_id, C.name

ORDER BY total_books_purchased DESC

LIMIT 5;
```

2. Total revenue generated from book sales by each author

```
SELECT B.author, SUM(OD.quantity * B.price) AS total_revenue
FROM Books B
JOIN OrderDetails OD ON B.book_id = OD.book_id
GROUP BY B.author
ORDER BY total_revenue DESC;
```

3. Books that have been ordered more than 10 times, along with the total quantity ordered for each book

```
SELECT B.book_id, B.title, SUM(OD.quantity) AS total_quantity_ordered
FROM Books B
JOIN OrderDetails OD ON B.book_id = OD.book_id
GROUP BY B.book_id, B.title
HAVING SUM(OD.quantity) > 10;
```