**Module 1 - Design Patterns and Principles**

**Exercise 1: Implementing the Singleton Pattern**

**Scenario:**

You need to ensure that a logging utility class in your application has only one instance throughout the application lifecycle to ensure consistent logging.

**Program:**

import java.util.\*;

public class Main

{

static class Logger

{

private static Logger instance;

private Logger()

{

System.out.println("Logger Instance Created");

}

public static Logger getInstance()

{

if (instance == null)

{

instance = new Logger();

}

return instance;

}

public void log(String message)

{

System.out.println("LOG: " + message);

}

}

public static void main(String[] args)

{

Logger logger1 = Logger.getInstance();

Logger logger2 = Logger.getInstance();

logger1.log("First log message.");

logger2.log("Second log message.");

if (logger1 == logger2)

{

System.out.println("Both logger1 and logger2 point to the same instance.");

} else

{

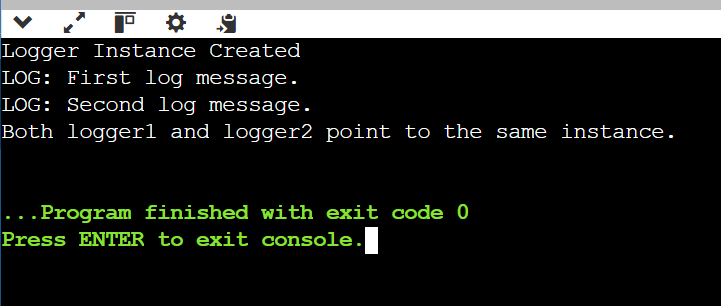
System.out.println("Different instances were created. Singleton failed.");

}

}

}

**Output:**



**Exercise 2: Implementing the Factory Method Pattern**

**Scenario:**

You are developing a document management system that needs to create different types of documents (e.g., Word, PDF, Excel). Use the Factory Method Pattern to achieve this.

**Program:**

import java.util.\*;

interface Document

{

void open();

}

class WordDocument implements Document

{

public void open()

{

System.out.println("Opening Word Document");

}

}

class PdfDocument implements Document

{

public void open()

{

System.out.println("Opening PDF Document");

}

}

class ExcelDocument implements Document

{

public void open()

{

System.out.println("Opening Excel Document");

}

}

abstract class DocumentFactory

{

public abstract Document createDocument();

}

class WordFactory extends DocumentFactory

{

public Document createDocument()

{

return new WordDocument();

}

}

class PdfFactory extends DocumentFactory

{

public Document createDocument()

{

return new PdfDocument();

}

}

class ExcelFactory extends DocumentFactory

{

public Document createDocument()

{

return new ExcelDocument();

}

}

class Main {

public static void main(String[] args)

{

Scanner sc = new Scanner(System.in);

System.out.print("Enter document type (word/pdf/excel): ");

String type = sc.next().toLowerCase();

DocumentFactory factory = switch (type)

{

case "word" -> new WordFactory();

case "pdf" -> new PdfFactory();

case "excel" -> new ExcelFactory();

default -> null;

};

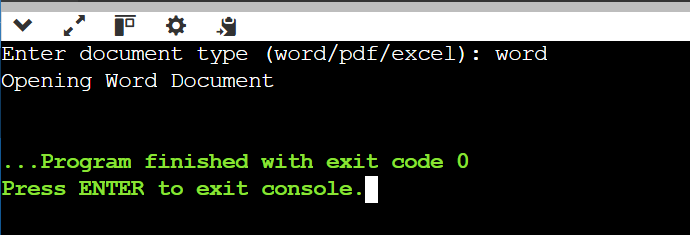
if (factory != null) factory.createDocument().open();

else System.out.println("Invalid type");

}

}

**Output:**



**Exercise 3: Implementing the Builder Pattern**

**Scenario:**

You are developing a system to create complex objects such as a Computer with multiple optional parts. Use the Builder Pattern to manage the construction process.

**Program:**

import java.util.Scanner;

class Computer

{

private String cpu;

private String ram;

private String storage;

private Computer(Builder builder)

{

this.cpu = builder.cpu;

this.ram = builder.ram;

this.storage = builder.storage;

}

public static class Builder

{

private String cpu;

private String ram;

private String storage;

public Builder setCpu(String cpu)

{

this.cpu = cpu;

return this;

}

public Builder setRam(String ram)

{

this.ram = ram;

return this;

}

public Builder setStorage(String storage)

{

this.storage = storage;

return this;

}

public Computer build()

{

return new Computer(this);

}

}

@Override

public String toString()

{

return "Computer [CPU=" + cpu + ", RAM=" + ram + ", Storage=" + storage + "]";

}

}

class Main {

public static void main(String[] args)

{

Scanner scanner = new Scanner(System.in);

System.out.print("Enter CPU: ");

String cpu = scanner.nextLine();

System.out.print("Enter RAM: ");

String ram = scanner.nextLine();

System.out.print("Enter Storage: ");

String storage = scanner.nextLine();

Computer customPC = new Computer.Builder()

.setCpu(cpu)

.setRam(ram)

.setStorage(storage)

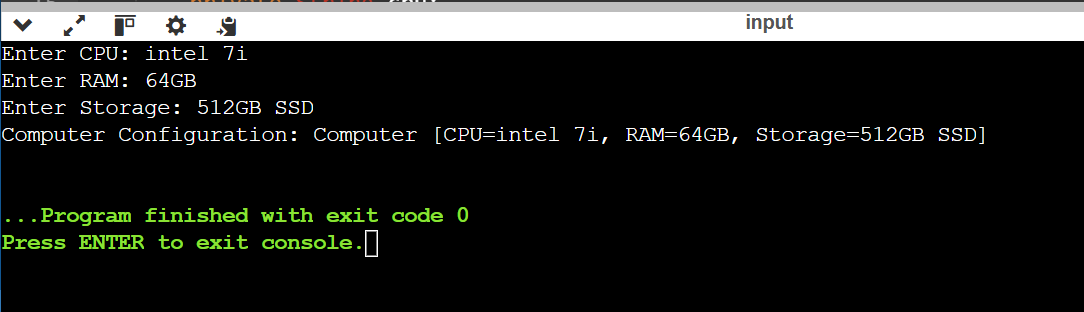
.build();

System.out.println("Computer Configuration: " + customPC);

}

}

**Output:**



**Exercise 4: Implementing the Adapter Pattern**

**Scenario:**

You are developing a payment processing system that needs to integrate with multiple third-party payment gateways with different interfaces. Use the Adapter Pattern to achieve this.

**Program:**

import java.util.Scanner;

interface PaymentProcessor

{

void processPayment(double amount);

}

class PayPalGateway

{

public void sendPayment(double amount)

{

System.out.println("Paid via PayPal: $" + amount);

}

}

class StripeGateway

{

public void makePayment(double amount)

{

System.out.println("Paid via Stripe: $" + amount);

}

}

class PayPalAdapter implements PaymentProcessor

{

private PayPalGateway gateway = new PayPalGateway();

public void processPayment(double amount)

{

gateway.sendPayment(amount);

}

}

class StripeAdapter implements PaymentProcessor

{

private StripeGateway gateway = new StripeGateway();

public void processPayment(double amount)

{

gateway.makePayment(amount);

}

}

class Main

{

public static void main(String[] args)

{

Scanner scanner = new Scanner(System.in);

System.out.print("Choose payment method (paypal/stripe): ");

String method = scanner.next().toLowerCase();

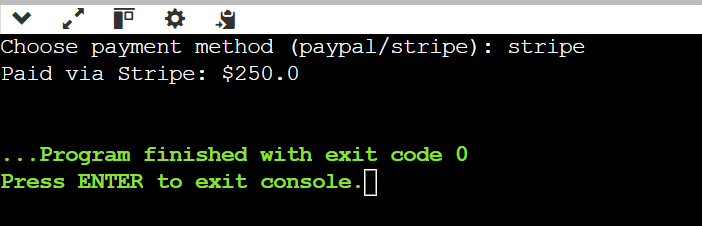
PaymentProcessor processor = method.equals("paypal") ? new PayPalAdapter() : new StripeAdapter();

processor.processPayment(250.00);

}

}

**Output:**



**Exercise 5: Implementing the Decorator Pattern**

**Scenario:**

You are developing a notification system where notifications can be sent via multiple channels (e.g., Email, SMS). Use the Decorator Pattern to add functionalities dynamically.

**Program:**

import java.util.Scanner;

interface Notifier

{

void send(String message);

}

class EmailNotifier implements Notifier

{

public void send(String message)

{

System.out.println("Sending Email: " + message);

}

}

abstract class NotifierDecorator implements Notifier

{

protected Notifier notifier;

public NotifierDecorator(Notifier notifier)

{

this.notifier = notifier;

}

}

class SMSNotifier extends NotifierDecorator

{

public SMSNotifier(Notifier notifier)

{

super(notifier);

}

public void send(String message)

{

notifier.send(message);

System.out.println("Sending SMS: " + message);

}

}

class Main

{

public static void main(String[] args)

{

Scanner scanner = new Scanner(System.in);

System.out.print("Enter notification message: ");

String message = scanner.nextLine();

Notifier email = new EmailNotifier();

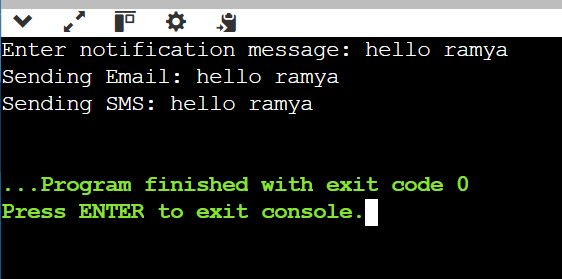
Notifier sms = new SMSNotifier(email);

sms.send(message);

}

}

**Output:**



**Exercise 6: Implementing the Proxy Pattern**

**Scenario:**

You are developing an image viewer application that loads images from a remote server. Use the Proxy Pattern to add lazy initialization and caching.

**Program:**

import java.util.Scanner;

interface Image

{

void display();

}

class RealImage implements Image

{

private String filename;

public RealImage(String filename)

{

this.filename = filename;

loadFromDisk();

}

private void loadFromDisk()

{

System.out.println("Loading image: " + filename);

}

public void display()

{

System.out.println("Displaying: " + filename);

}

}

class ProxyImage implements Image

{

private RealImage realImage;

private String filename;

public ProxyImage(String filename)

{

this.filename = filename;

}

public void display()

{

if (realImage == null)

{

realImage = new RealImage(filename);

}

realImage.display();

}

}

class Main

{

public static void main(String[] args)

{

Scanner scanner = new Scanner(System.in);

System.out.print("Enter image filename: ");

String name = scanner.nextLine();

Image image = new ProxyImage(name);

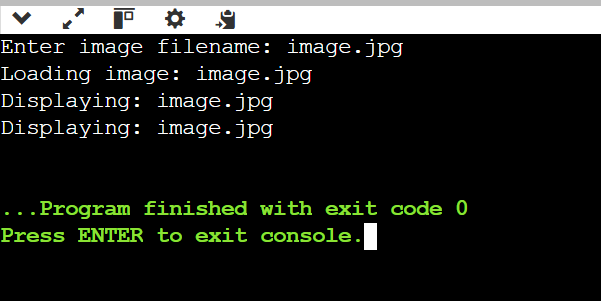
image.display();

image.display();

}

}

**Output:**



**Exercise 7: Implementing the Observer Pattern**

**Scenario:**

You are developing a stock market monitoring application where multiple clients need to be notified whenever stock prices change. Use the Observer Pattern to achieve this.

**Program:**

import java.util.\*;

interface Observer

{

void update(float price);

}

interface Stock

{

void register(Observer o);

void unregister(Observer o);

void notifyObservers();

}

class StockMarket implements Stock

{

private List<Observer> observers = new ArrayList<>();

private float stockPrice;

public void setPrice(float price)

{

this.stockPrice = price;

notifyObservers();

}

public void register(Observer o)

{

observers.add(o);

}

public void unregister(Observer o)

{

observers.remove(o);

}

public void notifyObservers()

{

for (Observer o : observers)

{

o.update(stockPrice);

}

}

}

class MobileApp implements Observer

{

public void update(float price)

{

System.out.println("Mobile App - New Price: $" + price);

}

}

class WebApp implements Observer

{

public void update(float price)

{

System.out.println("Web App - New Price: $" + price);

}

}

class Main

{

public static void main(String[] args)

{

Scanner scanner = new Scanner(System.in);

StockMarket market = new StockMarket();

Observer mobile = new MobileApp();

Observer web = new WebApp();

market.register(mobile);

market.register(web);

System.out.print("Enter new stock price: ");

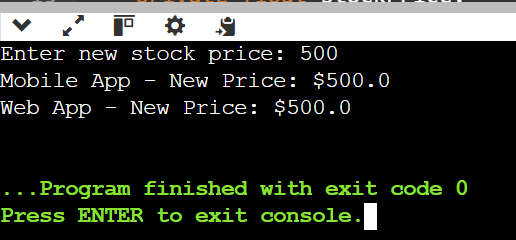
float price = scanner.nextFloat();

market.setPrice(price);

}

}

**Output:**



**Module 2 - Data Structures and Algorithms**

**Exercise 1: Inventory Management System**

**Scenario:**

You are developing an inventory management system for a warehouse. Efficient data storage and retrieval are crucial.

**Program:**

import java.util.HashMap;

import java.util.Scanner;

public class Main

{

static class Product

{

private String productId;

private String productName;

private int quantity;

private double price;

public Product(String productId, String productName, int quantity, double price)

{

this.productId = productId;

this.productName = productName;

this.quantity = quantity;

this.price = price;

}

public String getProductId()

{

return productId;

}

public void setProductName(String productName)

{

this.productName = productName;

}

public void setQuantity(int quantity)

{

this.quantity = quantity;

}

public void setPrice(double price)

{

this.price = price;

}

@Override

public String toString()

{

return "Product{productId='" + productId + "', name='" + productName + "', quantity=" + quantity + ", price=" + price + "}";

}

}

static class Inventory

{

private HashMap<String, Product> products;

public Inventory()

{

products = new HashMap<>();

}

public void addProduct(Product p)

{

products.put(p.getProductId(), p);

}

public void updateProduct(String productId, String newName, int newQuantity, double newPrice)

{

Product p = products.get(productId);

if (p != null)

{

p.setProductName(newName);

p.setQuantity(newQuantity);

p.setPrice(newPrice);

}

else

{

System.out.println("Product with ID " + productId + " not found.");

}

}

public void deleteProduct(String productId)

{

if (products.remove(productId) == null)

{

System.out.println("Product with ID " + productId + " not found.");

}

}

public void listAllProducts()

{

if (products.isEmpty())

{

System.out.println("Inventory is empty.");

return;

}

for (Product p : products.values())

{

System.out.println(p);

}

}

}

public static void main(String[] args)

{

Inventory inventory = new Inventory();

Scanner scanner = new Scanner(System.in);

while (true)

{

System.out.println("\n1. Add Product\n2. Update Product\n3. Delete Product\n4. List Products\n5. Exit");

System.out.print("Enter choice: ");

int choice = scanner.nextInt();

scanner.nextLine();

if (choice == 1)

{

System.out.print("Enter Product ID: ");

String id = scanner.nextLine();

System.out.print("Enter Product Name: ");

String name = scanner.nextLine();

System.out.print("Enter Quantity: ");

int quantity = scanner.nextInt();

System.out.print("Enter Price: ");

double price = scanner.nextDouble();

scanner.nextLine();

inventory.addProduct(new Product(id, name, quantity, price));

}

else if (choice == 2)

{

System.out.print("Enter Product ID to Update: ");

String id = scanner.nextLine();

System.out.print("Enter New Name: ");

String name = scanner.nextLine();

System.out.print("Enter New Quantity: ");

int quantity = scanner.nextInt();

System.out.print("Enter New Price: ");

double price = scanner.nextDouble();

scanner.nextLine();

inventory.updateProduct(id, name, quantity, price);

}

else if (choice == 3)

{

System.out.print("Enter Product ID to Delete: ");

String id = scanner.nextLine();

inventory.deleteProduct(id);

}

else if (choice == 4)

{

inventory.listAllProducts();

}

else if (choice == 5)

{

System.out.println("Exiting.");

break;

}

else

{

System.out.println("Invalid choice.");

}

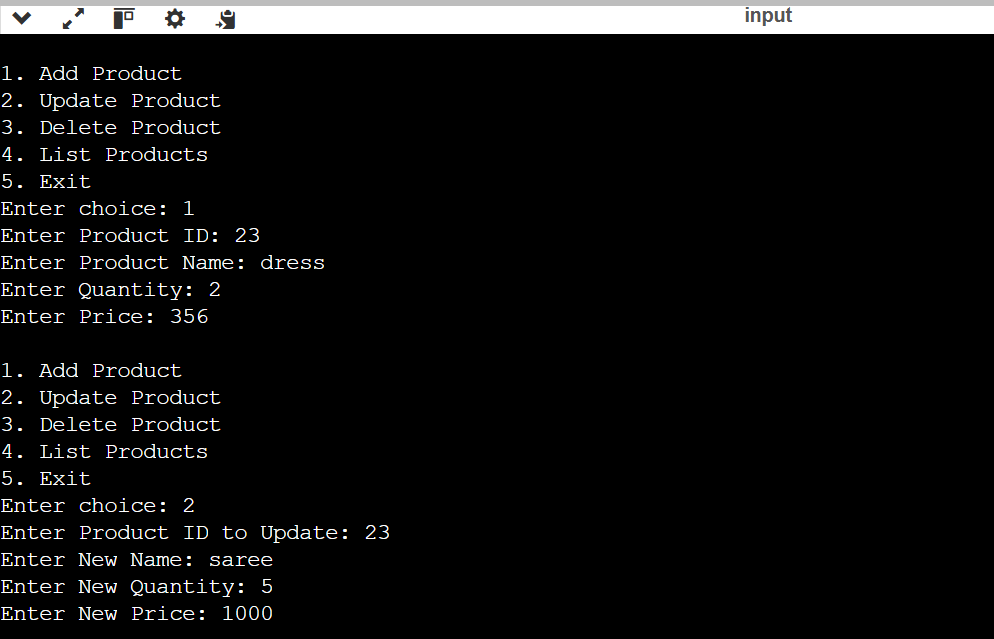
}

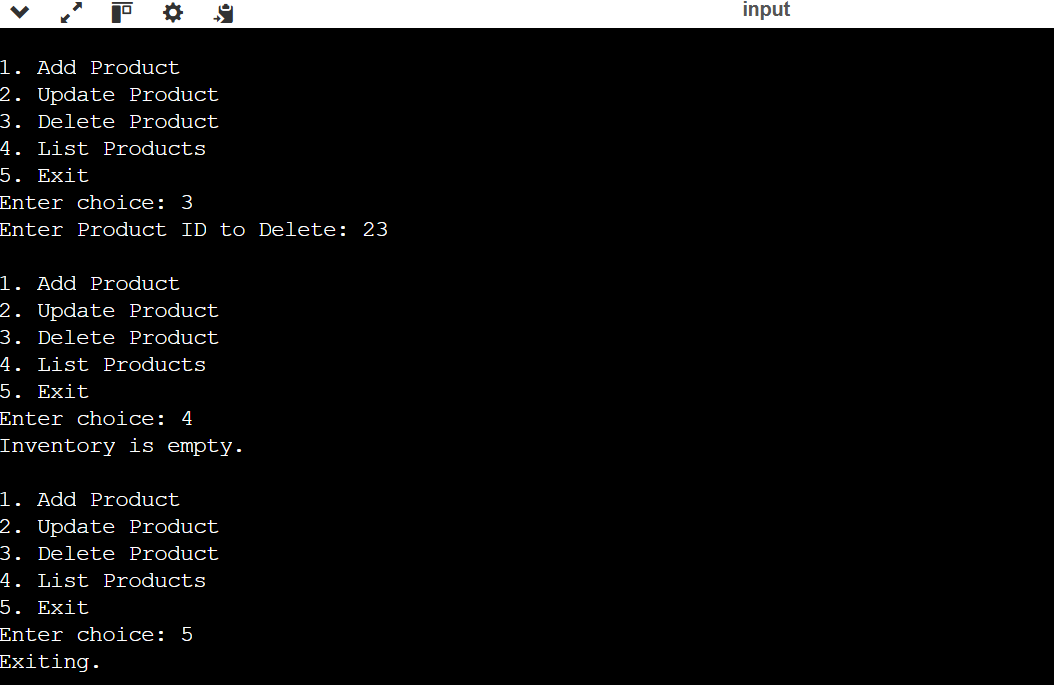
scanner.close();

}

}

**Output:**





**Exercise 2: E-commerce Platform Search Function**

**Scenario:**

You are working on the search functionality of an e-commerce platform. The search needs to be optimized for fast performance.

**Program:**

import java.util.Arrays;

public class Main

{

static class Product

{

private String productId;

private String productName;

private String category;

public Product(String productId, String productName, String category)

{

this.productId = productId;

this.productName = productName;

this.category = category;

}

public String getProductId()

{

return productId;

}

public String getProductName()

{

return productName;

}

public String getCategory()

{

return category;

}

@Override

public String toString()

{

return "Product{productId='" + productId + "', name='" + productName + "', category='" + category + "'}";

}

}

public static Product linearSearch(Product[] products, String targetId)

{

for (Product p : products)

{

if (p.getProductId().equals(targetId))

{

return p;

}

}

return null;

}

public static Product binarySearch(Product[] products, String targetId)

{

int left = 0;

int right = products.length - 1;

while (left <= right)

{

int mid = left + (right - left) / 2;

int compare = products[mid].getProductId().compareTo(targetId);

if (compare == 0) {

return products[mid];

}

else if (compare < 0)

{

left = mid + 1;

}

else

{

right = mid - 1;

}

}

return null;

}

public static void main(String[] args)

{

Product[] products = {

new Product("P001", "Laptop", "Electronics"),

new Product("P002", "Phone", "Electronics"),

new Product("P003", "Watch", "Accessories"),

new Product("P004", "Headphones", "Audio"),

new Product("P005", "Keyboard", "Computer Accessories")

};

Arrays.sort(products, (a, b) -> a.getProductId().compareTo(b.getProductId()));

String targetId = "P004";

System.out.println("\nPerforming Linear Search:");

Product resultLinear = linearSearch(products, targetId);

System.out.println(resultLinear != null ? resultLinear : "Product not found");

System.out.println("\nPerforming Binary Search:");

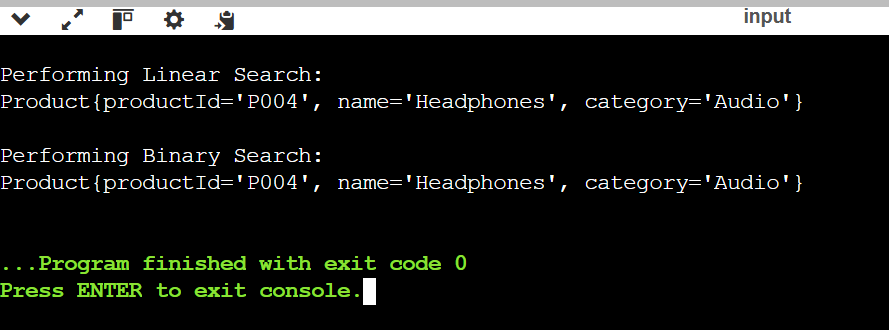
Product resultBinary = binarySearch(products, targetId);

System.out.println(resultBinary != null ? resultBinary : "Product not found");

}

}

**Output:**



**Exercise 7: Financial Forecasting**

**Scenario:**

You are developing a financial forecasting tool that predicts future values based on past data.

**Program:**

import java.util.Scanner;

public class Main

{

public static double futureValue(double principal, double rate, int years)

{

if (years == 0)

{

return principal;

}

else

{

return futureValue(principal, rate, years - 1) \* (1 + rate);

}

}

public static void main(String[] args)

{

double principal = 1000.0;

double rate = 0.05;

int years = 10;

double result = futureValue(principal, rate, years);

System.out.println("Future Value after " + years + " years: " + result);

double optimizedResult = principal \* Math.pow(1 + rate, years);

System.out.println("Optimized (Direct) Result: " + optimizedResult);

}

}

**Output:**

