**Exercise 1: Implementing the Singleton Pattern (Mandatory hands-on)**

Code

public class Logger {

    private static Logger singleInstance;

    private Logger() {

        System.out.println("Logger instance created.");

    }

    public static Logger getInstance() {

        if (singleInstance == null) {

            singleInstance = new Logger();

        }

        return singleInstance;

    }

    public void log(String message) {

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

    }

}

public class Main {

    public static void main(String[] args) {

        Logger logger1 = Logger.getInstance();

        logger1.log("First log message");

        Logger logger2 = Logger.getInstance();

        logger2.log("Second log message");

        // To Check if both logger1 and logger2 are the same instance

        if (logger1 == logger2) {

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

        } else {

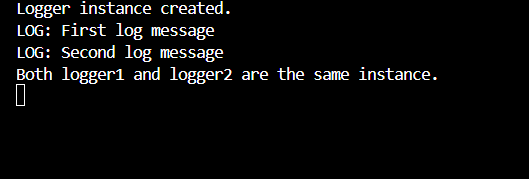
            System.out.println("Different Logger instances exist.");

        }

    }

}

Output



**Exercise 2: Implementing the Factory Method Pattern (Mandatory hands-on)**

Code

interface Document {

    void open();

}

class WordDocument implements Document {

    public void open() {

        System.out.println("Opening a Word document.");

    }

}

class PdfDocument implements Document {

    public void open() {

        System.out.println("Opening a PDF document.");

    }

}

class ExcelDocument implements Document {

    public void open() {

        System.out.println("Opening an Excel document.");

    }

}

abstract class DocumentFactory {

    public abstract Document createDocument();

}

class WordDocumentFactory extends DocumentFactory {

    public Document createDocument() {

        return new WordDocument();

    }

}

class PdfDocumentFactory extends DocumentFactory {

    public Document createDocument() {

        return new PdfDocument();

    }

}

class ExcelDocumentFactory extends DocumentFactory {

    public Document createDocument() {

        return new ExcelDocument();

    }

}

public class Main {

    public static void main(String[] args) {

        DocumentFactory wordFactory = new WordDocumentFactory();

        Document wordDoc = wordFactory.createDocument();

        wordDoc.open();

        DocumentFactory pdfFactory = new PdfDocumentFactory();

        Document pdfDoc = pdfFactory.createDocument();

        pdfDoc.open();

        DocumentFactory excelFactory = new ExcelDocumentFactory();

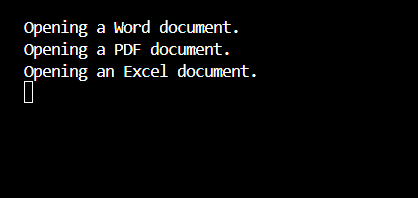
        Document excelDoc = excelFactory.createDocument();

        excelDoc.open();

    }

}

Output



**Exercise 3: Implementing the Builder Pattern**

Code

public class Computer {

    private String CPU;

    private String RAM;

    private String storage;

    private String graphicsCard;

    private boolean hasWiFi;

    private boolean hasBluetooth;

    private Computer(Builder builder) {

        this.CPU = builder.CPU;

        this.RAM = builder.RAM;

        this.storage = builder.storage;

        this.graphicsCard = builder.graphicsCard;

        this.hasWiFi = builder.hasWiFi;

        this.hasBluetooth = builder.hasBluetooth;

    }

    public String toString() {

        return "Computer Configuration:\n"

             + "CPU: " + CPU + "\n"

             + "RAM: " + RAM + "\n"

             + "Storage: " + storage + "\n"

             + "Graphics Card: " + graphicsCard + "\n"

             + "WiFi: " + hasWiFi + "\n"

             + "Bluetooth: " + hasBluetooth;

    }

    public static class Builder {

        private String CPU;

        private String RAM;

        private String storage;

        private String graphicsCard;

        private boolean hasWiFi;

        private boolean hasBluetooth;

        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 Builder setGraphicsCard(String graphicsCard) {

            this.graphicsCard = graphicsCard;

            return this;

        }

        public Builder setWiFi(boolean hasWiFi) {

            this.hasWiFi = hasWiFi;

            return this;

        }

        public Builder setBluetooth(boolean hasBluetooth) {

            this.hasBluetooth = hasBluetooth;

            return this;

        }

        public Computer build() {

            return new Computer(this);

        }

    }

}

public class Main {

    public static void main(String[] args) {

        // Build a high-end computer

        Computer gamingPC = new Computer.Builder()

            .setCPU("Intel i9")

            .setRAM("32GB")

            .setStorage("1TB SSD")

            .setGraphicsCard("NVIDIA RTX 4090")

            .setWiFi(true)

            .setBluetooth(true)

            .build();

        System.out.println(gamingPC);

        // Build a budget computer

        Computer officePC = new Computer.Builder()

            .setCPU("Intel i5")

            .setRAM("8GB")

            .setStorage("512GB HDD")

            .setWiFi(false)

            .setBluetooth(false)

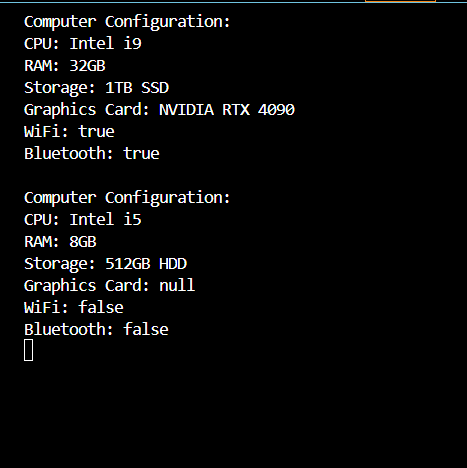
            .build();

        System.out.println("\n" + officePC);

    }

}

Output



**Exercise 4: Implementing the Adapter Pattern**

Code

// Target Interface

interface PaymentProcessor {

    void processPayment(double amount);

}

// PayPal Gateway

class PayPalGateway {

    public void makePayment(double amountInUSD) {

        System.out.println("Payment of $" + amountInUSD + " made using PayPal.");

    }

}

// Stripe Gateway

class StripeGateway {

    public void pay(double money) {

        System.out.println("Payment of $" + money + " made using Stripe.");

    }

}

// Adapter for PayPal

class PayPalAdapter implements PaymentProcessor {

    private PayPalGateway payPal;

    public PayPalAdapter(PayPalGateway payPal) {

        this.payPal = payPal;

    }

    @Override

    public void processPayment(double amount) {

        payPal.makePayment(amount);

    }

}

// Adapter for Stripe

class StripeAdapter implements PaymentProcessor {

    private StripeGateway stripe;

    public StripeAdapter(StripeGateway stripe) {

        this.stripe = stripe;

    }

    @Override

    public void processPayment(double amount) {

        stripe.pay(amount);

    }

}

// Main class to test the Adapter Pattern

public class Main {

    public static void main(String[] args) {

        // Using PayPal through Adapter

        PaymentProcessor paypalProcessor = new PayPalAdapter(new PayPalGateway());

        paypalProcessor.processPayment(150.75);

        // Using Stripe through Adapter

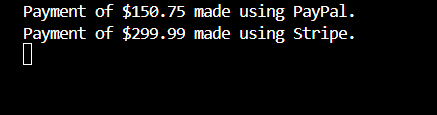
        PaymentProcessor stripeProcessor = new StripeAdapter(new StripeGateway());

        stripeProcessor.processPayment(299.99);

    }

}

Output



**Exercise 5: Implementing the Decorator Pattern**

Code

interface Notifier {

    void send(String message);

}

class EmailNotifier implements Notifier {

    @Override

    public void send(String message) {

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

    }

}

abstract class NotifierDecorator implements Notifier {

    protected Notifier wrappedNotifier;

    public NotifierDecorator(Notifier notifier) {

        this.wrappedNotifier = notifier;

    }

    @Override

    public void send(String message) {

        wrappedNotifier.send(message);

    }

}

class SMSNotifierDecorator extends NotifierDecorator {

    public SMSNotifierDecorator(Notifier notifier) {

        super(notifier);

    }

    @Override

    public void send(String message) {

        super.send(message);

        sendSMS(message);

    }

    private void sendSMS(String message) {

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

    }

}

class SlackNotifierDecorator extends NotifierDecorator {

    public SlackNotifierDecorator(Notifier notifier) {

        super(notifier);

    }

    @Override

    public void send(String message) {

        super.send(message);

        sendSlack(message);

    }

    private void sendSlack(String message) {

        System.out.println("Slack Notification Sent: " + message);

    }

}

// Main class to test the Decorator Pattern

public class Main {

    public static void main(String[] args) {

        // Base notifier

        Notifier emailNotifier = new EmailNotifier();

        // Add SMS notification

        Notifier emailAndSMS = new SMSNotifierDecorator(emailNotifier);

        // Add Slack on top of Email + SMS

        Notifier multiChannelNotifier = new SlackNotifierDecorator(emailAndSMS);

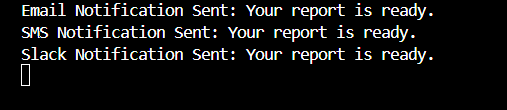
        // Send notification

        multiChannelNotifier.send("Your report is ready.");

    }

}

Output



**Exercise 6: Implementing the Proxy Pattern**

Code

interface Image {

    void display();

}

class RealImage implements Image {

    private String fileName;

    public RealImage(String fileName) {

        this.fileName = fileName;

        loadFromRemoteServer();

    }

    private void loadFromRemoteServer() {

        System.out.println("Loading image from remote server: " + fileName);

    }

    @Override

    public void display() {

        System.out.println("Displaying image: " + fileName);

    }

}

// Proxy Class

class ProxyImage implements Image {

    private RealImage realImage;

    private String fileName;

    public ProxyImage(String fileName) {

        this.fileName = fileName;

    }

    @Override

    public void display() {

        if (realImage == null) {

            realImage = new RealImage(fileName); // Lazy initialization

        } else {

            System.out.println("Image loaded from cache: " + fileName);

        }

        realImage.display();

    }

}

// Main class to test the Proxy Pattern

public class Main {

    public static void main(String[] args) {

        Image image1 = new ProxyImage("nature\_photo.jpg");

        // Image will be loaded from remote server

        image1.display();

        // Image will be loaded from cache

        image1.display();

        // Another image - will be fetched again

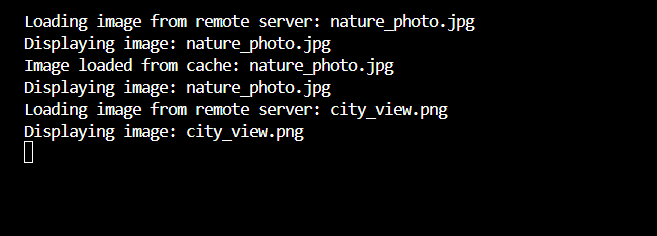
        Image image2 = new ProxyImage("city\_view.png");

        image2.display();

    }

}

Output



**Exercise 7: Implementing the Observer Pattern**

Code

import java.util.ArrayList;

import java.util.List;

interface Observer {

    void update(String stockName, double stockPrice);

}

interface Stock {

    void registerObserver(Observer o);

    void removeObserver(Observer o);

    void notifyObservers();

}

class StockMarket implements Stock {

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

    private String stockName;

    private double stockPrice;

    public void setStockData(String stockName, double stockPrice) {

        this.stockName = stockName;

        this.stockPrice = stockPrice;

        notifyObservers();

    }

    @Override

    public void registerObserver(Observer o) {

        observers.add(o);

    }

    @Override

    public void removeObserver(Observer o) {

        observers.remove(o);

    }

    @Override

    public void notifyObservers() {

        for (Observer o : observers) {

            o.update(stockName, stockPrice);

        }

    }

}

class MobileApp implements Observer {

    private String appName;

    public MobileApp(String appName) {

        this.appName = appName;

    }

    @Override

    public void update(String stockName, double stockPrice) {

        System.out.println("[" + appName + "] Stock Update - " + stockName + ": $" + stockPrice);

    }

}

class WebApp implements Observer {

    private String siteName;

    public WebApp(String siteName) {

        this.siteName = siteName;

    }

    @Override

    public void update(String stockName, double stockPrice) {

        System.out.println("[" + siteName + "] Stock Update - " + stockName + ": $" + stockPrice);

    }

}

public class Main {

    public static void main(String[] args) {

        // Create subject

        StockMarket stockMarket = new StockMarket();

        // Create observers

        Observer mobileClient = new MobileApp("StockMobile");

        Observer webClient = new WebApp("StockWeb");

        // Register observers

        stockMarket.registerObserver(mobileClient);

        stockMarket.registerObserver(webClient);

        // Simulate stock price updates

        stockMarket.setStockData("TCS", 3750.50);

        stockMarket.setStockData("INFY", 1489.10);

        // Remove one observer and simulate another update

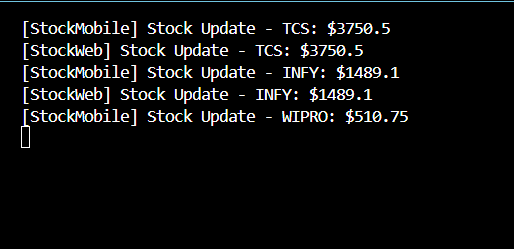
        stockMarket.removeObserver(webClient);

        stockMarket.setStockData("WIPRO", 510.75);

    }

}

Output



**Exercise 8: Implementing the Strategy Pattern**

Code

interface PaymentStrategy {

    void pay(double amount);

}

// Credit Card

class CreditCardPayment implements PaymentStrategy {

    private String cardNumber;

    private String cardHolderName;

    public CreditCardPayment(String cardNumber, String cardHolderName) {

        this.cardNumber = cardNumber;

        this.cardHolderName = cardHolderName;

    }

    @Override

    public void pay(double amount) {

        System.out.println("Paid $" + amount + " using Credit Card (Holder: " + cardHolderName + ").");

    }

}

// PayPal

class PayPalPayment implements PaymentStrategy {

    private String email;

    public PayPalPayment(String email) {

        this.email = email;

    }

    @Override

    public void pay(double amount) {

        System.out.println("Paid $" + amount + " using PayPal (Email: " + email + ").");

    }

}

class PaymentContext {

    private PaymentStrategy strategy;

    public void setPaymentStrategy(PaymentStrategy strategy) {

        this.strategy = strategy;

    }

    public void executePayment(double amount) {

        if (strategy != null) {

            strategy.pay(amount);

        } else {

            System.out.println("Payment strategy not set.");

        }

    }

}

// Main class to test the Strategy Pattern

public class Main {

    public static void main(String[] args) {

        PaymentContext context = new PaymentContext();

        // Pay using Credit Card

        context.setPaymentStrategy(new CreditCardPayment("1234-5678-9012-3456", "Sujitha R"));

        context.executePayment(250.00);

        // Pay using PayPal

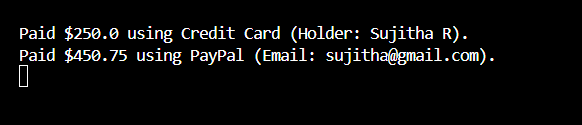
        context.setPaymentStrategy(new PayPalPayment("sujitha@gmail.com"));

        context.executePayment(450.75);

    }

}

Output



**Exercise 9: Implementing the Command Pattern**

Code

interface Command {

    void execute();

}

class Light {

    public void turnOn() {

        System.out.println("Light is ON");

    }

    public void turnOff() {

        System.out.println("Light is OFF");

    }

}

// Command to turn on the light

class LightOnCommand implements Command {

    private Light light;

    public LightOnCommand(Light light) {

        this.light = light;

    }

    @Override

    public void execute() {

        light.turnOn();

    }

}

//  Command to turn off the light

class LightOffCommand implements Command {

    private Light light;

    public LightOffCommand(Light light) {

        this.light = light;

    }

    @Override

    public void execute() {

        light.turnOff();

    }

}

// Invoker Class

class RemoteControl {

    private Command command;

    public void setCommand(Command command) {

        this.command = command;

    }

    // Execute the command

    public void pressButton() {

        if (command != null) {

            command.execute();

        } else {

            System.out.println("No command set.");

        }

    }

}

// Main class to test the Command Pattern

public class Main {

    public static void main(String[] args) {

        // Receiver

        Light livingRoomLight = new Light();

        // Commands

        Command lightOn = new LightOnCommand(livingRoomLight);

        Command lightOff = new LightOffCommand(livingRoomLight);

        // Invoker

        RemoteControl remote = new RemoteControl();

        // Turn on the light

        remote.setCommand(lightOn);

        remote.pressButton();

        // Turn off the light

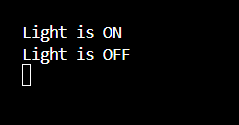
        remote.setCommand(lightOff);

        remote.pressButton();

    }

}

Output



**Exercise 10: Implementing the MVC Pattern**

Code

// Model Class

class Student {

    private String name;

    private String id;

    private String grade;

    public Student(String name, String id, String grade) {

        this.name = name;

        this.id = id;

        this.grade = grade;

    }

    // Getters and Setters

    public String getName() {

        return name;

    }

    public void setName(String name) {

        this.name = name;

    }

    public String getId() {

        return id;

    }

    public String getGrade() {

        return grade;

    }

    public void setGrade(String grade) {

        this.grade = grade;

    }

}

// View Class

class StudentView {

    public void displayStudentDetails(String name, String id, String grade) {

        System.out.println("Student Details:");

        System.out.println("Name: " + name);

        System.out.println("ID: " + id);

        System.out.println("Grade: " + grade);

    }

}

// Controller Class

class StudentController {

    private Student model;

    private StudentView view;

    public StudentController(Student model, StudentView view) {

        this.model = model;

        this.view = view;

    }

    public void setStudentName(String name) {

        model.setName(name);

    }

    public void setStudentGrade(String grade) {

        model.setGrade(grade);

    }

    public void updateView() {

        view.displayStudentDetails(model.getName(), model.getId(), model.getGrade());

    }

}

// Main class to test the MVC Pattern

public class Main {

    public static void main(String[] args) {

        // Create the model

        Student student = new Student("Sujitha", "AID123", "A");

        // Create the view

        StudentView view = new StudentView();

        // Create the controller

        StudentController controller = new StudentController(student, view);

        // Initial display

        controller.updateView();

        // Update student data

        controller.setStudentName("Sujitha R");

        controller.setStudentGrade("A+");

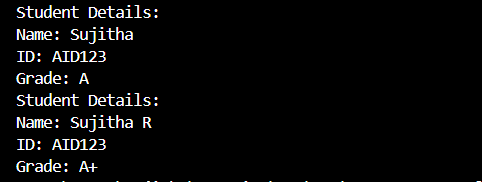
        // Updated display

        controller.updateView();

    }

}

Output



**Exercise 11: Implementing Dependency Injection**

Code

class Customer {

    private String id;

    private String name;

    public Customer(String id, String name) {

        this.id = id;

        this.name = name;

    }

    public String getId() {

        return id;

    }

    public String getName() {

        return name;

    }

}

// Repository Interface

interface CustomerRepository {

    Customer findCustomerById(String id);

}

class CustomerRepositoryImpl implements CustomerRepository {

    @Override

    public Customer findCustomerById(String id) {

        return new Customer(id, "Sujitha R");

    }

}

class CustomerService {

    private CustomerRepository repository;

    public CustomerService(CustomerRepository repository) {

        this.repository = repository;

    }

    public void getCustomerDetails(String id) {

        Customer customer = repository.findCustomerById(id);

        System.out.println("Customer Details:");

        System.out.println("ID: " + customer.getId());

        System.out.println("Name: " + customer.getName());

    }

}

// Main class to test Dependency Injection

public class Main {

    public static void main(String[] args) {

        // Create repository

        CustomerRepository repo = new CustomerRepositoryImpl();

        // Inject repository into service

        CustomerService service = new CustomerService(repo);

        // Use service to get customer details

        service.getCustomerDetails("C123");

    }

}

Output

