**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.

class Logger {

    // Volatile keyword ensures that multiple threads handle the loggerInstance variable correctly

    private static volatile Logger loggerInstance;

    // Private constructor to prevent instantiation

    private Logger() {}

    // Double-checked locking mechanism to ensure thread safety and lazy initialization

    public static Logger getInstance() {

        if (loggerInstance == null) {

            synchronized (Logger.class) {

                if (loggerInstance == null) {

                    loggerInstance = new Logger();

                }

            }

        }

        return loggerInstance;

    }

    public void log(String message) {

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

    }

}

public class SingletonPattern {

    public static void main(String[] args) {

        Logger logger1 = Logger.getInstance();

        Logger logger2 = Logger.getInstance();

        logger1.log("This is the first log message.");

        logger2.log("This is the second log message.");

        if (logger1 == logger2) {

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

        } else {

            System.out.println("Logger instances are different.");

        }

    }

}

**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.

interface Document {

    void open();

    void close();

}

class WordDocument implements Document {

    public void open() {

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

    }

    public void close() {

        System.out.println("Closing Word document...");

    }

}

class PdfDocument implements Document {

    public void open() {

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

    }

    public void close() {

        System.out.println("Closing PDF document...");

    }

}

class ExcelDocument implements Document {

    public void open() {

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

    }

    public void close() {

        System.out.println("Closing 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 FactoryMethodPattern {

    public static void main(String[] args) {

        DocumentFactory wordDocFactory = new WordDocumentFactory();

        Document wordDoc = wordDocFactory.createDocument();

        wordDoc.open();

        wordDoc.close();

        DocumentFactory pdfDocFactory = new PdfDocumentFactory();

        Document pdfDoc = pdfDocFactory.createDocument();

        pdfDoc.open();

        pdfDoc.close();

        DocumentFactory excelDocFactory = new ExcelDocumentFactory();

        Document excelDoc = excelDocFactory.createDocument();

        excelDoc.open();

        excelDoc.close();

    }

}

**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.

public class BuilderPattern {

    static class Computer {

        private String processor;

        private String memory;

        private String disk;

        private Computer(Builder builder) {

            this.processor = builder.processor;

            this.memory = builder.memory;

            this.disk = builder.disk;

        }

        // Getters

        public String getProcessor() {

            return processor;

        }

        public String getMemory() {

            return memory;

        }

        public String getDisk() {

            return disk;

        }

        public static class Builder {

            private String processor;

            private String memory;

            private String disk;

            public Builder setProcessor(String processor) {

                this.processor = processor;

                return this;

            }

            public Builder setMemory(String memory) {

                this.memory = memory;

                return this;

            }

            public Builder setDisk(String disk) {

                this.disk = disk;

                return this;

            }

            public Computer build() {

                return new Computer(this);

            }

        }

    }

    public static void main(String[] args) {

        Computer gamingComputer = new Computer.Builder()

                .setProcessor("Intel Core i9")

                .setMemory("32GB")

                .setDisk("1TB SSD")

                .build();

        System.out.println("Processor: " + gamingComputer.getProcessor());

        System.out.println("Memory: " + gamingComputer.getMemory());

        System.out.println("Disk: " + gamingComputer.getDisk());

    }

}

**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.

interface PaymentProcessor {

    void processPayment(double amount);

}

class PayPal {

    public void makePayment(double amount) {

        System.out.println("Processing payment of Rs." + amount + " through PayPal.");

    }

}

class Stripe {

    public void pay(double amount) {

        System.out.println("Processing payment of Rs." + amount + " through Stripe.");

    }

}

class AmazonPay {

    public void processTransaction(double amount) {

        System.out.println("Processing payment of Rs." + amount + " through Amazon Pay.");

    }

}

class PayPalAdapter implements PaymentProcessor {

    private PayPal paypalService;

    public PayPalAdapter(PayPal paypalService) {

        this.paypalService = paypalService;

    }

    public void processPayment(double amount) {

        paypalService.makePayment(amount);

    }

}

class StripeAdapter implements PaymentProcessor {

    private Stripe stripeService;

    public StripeAdapter(Stripe stripeService) {

        this.stripeService = stripeService;

    }

    public void processPayment(double amount) {

        stripeService.pay(amount);

    }

}

class AmazonPayAdapter implements PaymentProcessor {

    private AmazonPay amazonPayService;

    public AmazonPayAdapter(AmazonPay amazonPayService) {

        this.amazonPayService = amazonPayService;

    }

    public void processPayment(double amount) {

        amazonPayService.processTransaction(amount);

    }

}

public class AdapterPattern {

    public static void main(String[] args) {

        PayPal paypal = new PayPal();

        Stripe stripe = new Stripe();

        AmazonPay amazonPay = new AmazonPay();

        PaymentProcessor paypalAdapter = new PayPalAdapter(paypal);

        PaymentProcessor stripeAdapter = new StripeAdapter(stripe);

        PaymentProcessor amazonPayAdapter = new AmazonPayAdapter(amazonPay);

        paypalAdapter.processPayment(100.00);

        stripeAdapter.processPayment(200.00);

        amazonPayAdapter.processPayment(300.00);

    }

}

**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

interface Notifier {

    void send(String message);

}

class EmailNotifier implements Notifier {

    public void send(String message) {

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

    }

}

abstract class NotifierDecorator implements Notifier {

    protected Notifier baseNotifier;

    public NotifierDecorator(Notifier baseNotifier) {

        this.baseNotifier = baseNotifier;

    }

    public void send(String message) {

        baseNotifier.send(message);

    }

}

class SmsNotifierDecorator extends NotifierDecorator {

    public SmsNotifierDecorator(Notifier baseNotifier) {

        super(baseNotifier);

    }

    public void send(String message) {

        baseNotifier.send(message);

        sendSmsNotification(message);

    }

    private void sendSmsNotification(String message) {

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

    }

}

class SlackNotifierDecorator extends NotifierDecorator {

    public SlackNotifierDecorator(Notifier baseNotifier) {

        super(baseNotifier);

    }

    public void send(String message) {

        baseNotifier.send(message);

        sendSlackNotification(message);

    }

    private void sendSlackNotification(String message) {

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

    }

}

public class DecoratorPattern {

    public static void main(String[] args) {

        Notifier emailNotifier = new EmailNotifier();

        Notifier smsNotifier = new SmsNotifierDecorator(emailNotifier);

        Notifier slackNotifier = new SlackNotifierDecorator(smsNotifier);

        slackNotifier.send("Hello, this is a test notification!");

    }

}

**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

interface Image {

    void display();

}

class RealImage implements Image {

    private String filename;

    public RealImage(String filename) {

        this.filename = filename;

        loadImageFromDisk();

    }

    private void loadImageFromDisk() {

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

    }

    @Override

    public void display() {

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

    }

}

class ProxyImage implements Image {

    private String filename;

    private RealImage realImage;

    public ProxyImage(String filename) {

        if (filename == null || filename.isEmpty()) {

            throw new IllegalArgumentException("Filename cannot be null or empty.");

        }

        this.filename = filename;

    }

    @Override

    public void display() {

        if (realImage == null) {

            realImage = new RealImage(filename);

        }

        realImage.display();

    }

}

public class ProxyPattern {

    public static void main(String[] args) {

        Image image1 = new ProxyImage("image1.jpg");

        Image image2 = new ProxyImage("image2.jpg");

        System.out.println("First display for image1:");

        image1.display();

        System.out.println("");

        System.out.println("Second display for image1 (should use cached image):");

        image1.display();

        System.out.println("");

        System.out.println("First display for image2:");

        image2.display();

        System.out.println("");

        System.out.println("Second display for image2 (should use cached image):");

        image2.display();

    }

}

**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.

import java.util.ArrayList;

import java.util.List;

interface Stock {

    void registerObserver(Observer observer);

    void deregisterObserver(Observer observer);

    void notifyObservers();

}

class StockMarket implements Stock {

    private List<Observer> observerList;

    private double currentStockPrice;

    public StockMarket() {

        this.observerList = new ArrayList<>();

    }

    public void registerObserver(Observer observer) {

        observerList.add(observer);

    }

    public void deregisterObserver(Observer observer) {

        observerList.remove(observer);

    }

    public void notifyObservers() {

        for (Observer observer : observerList) {

            observer.update(currentStockPrice);

        }

    }

    public void setStockPrice(double stockPrice) {

        this.currentStockPrice = stockPrice;

        notifyObservers();

    }

}

interface Observer {

    void update(double stockPrice);

}

class MobileApp implements Observer {

    private String applicationName;

    public MobileApp(String applicationName) {

        this.applicationName = applicationName;

    }

    public void update(double stockPrice) {

        System.out.println(applicationName + " received stock price update: " + stockPrice);

    }

}

class WebApp implements Observer {

    private String applicationName;

    public WebApp(String applicationName) {

        this.applicationName = applicationName;

    }

    public void update(double stockPrice) {

        System.out.println(applicationName + " received stock price update: " + stockPrice);

    }

}

public class ObserverPattern {

    public static void main(String[] args) {

        StockMarket stockMarket = new StockMarket();

        Observer mobileApp = new MobileApp("MobileApp");

        Observer webApp = new WebApp("WebApp");

        stockMarket.registerObserver(mobileApp);

        stockMarket.registerObserver(webApp);

        stockMarket.setStockPrice(100.00);

        stockMarket.setStockPrice(101.50);

        stockMarket.deregisterObserver(webApp);

        stockMarket.setStockPrice(102.75);

    }

}

**Exercise 8: Implementing the Strategy Pattern**

**Scenario:** You are developing a payment system where different payment methods (e.g., Credit Card, PayPal) can be selected at runtime. Use the Strategy Pattern to achieve this.

interface PaymentStrategy {

    void pay(double amount);

}

class CreditCardPayment implements PaymentStrategy {

    private String name;

    private String cardNumber;

    private String cvv;

    private String expiryDate;

    public CreditCardPayment(String name, String cardNumber, String cvv, String expiryDate) {

        this.name = name;

        this.cardNumber = cardNumber;

        this.cvv = cvv;

        this.expiryDate = expiryDate;

    }

    @Override

    public void pay(double amount) {

        // Simulating payment process

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

        // Additional validation or processing can be added here

    }

}

class PayPalPayment implements PaymentStrategy {

    private String email;

    private String password;

    public PayPalPayment(String email, String password) {

        this.email = email;

        this.password = password;

    }

    @Override

    public void pay(double amount) {

        // Simulating payment process

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

        // Additional validation or processing can be added here

    }

}

class PaymentContext {

    private PaymentStrategy paymentStrategy;

    public void setPaymentStrategy(PaymentStrategy paymentStrategy) {

        this.paymentStrategy = paymentStrategy;

    }

    public void executePayment(double amount) {

        if (paymentStrategy == null) {

            throw new IllegalStateException("Payment strategy not set");

        }

        paymentStrategy.pay(amount);

    }

}

public class StrategyPattern {

    public static void main(String[] args) {

        PaymentContext paymentContext = new PaymentContext();

        // Using CreditCardPayment strategy

        paymentContext.setPaymentStrategy(new CreditCardPayment("John Doe", "1234567890123456", "123", "12/23"));

        paymentContext.executePayment(100.0);

        // Using PayPalPayment strategy

        paymentContext.setPaymentStrategy(new PayPalPayment("john.doe@example.com", "password123"));

        paymentContext.executePayment(200.0);

    }

}

**Exercise 9: Implementing the Command Pattern**

**Scenario:** You are developing a home automation system where commands can be issued to turn devices on or off. Use the Command Pattern to achieve this.

interface Command {

    void execute();

}

class LightOnCommand implements Command {

    private Light targetLight;

    public LightOnCommand(Light targetLight) {

        this.targetLight = targetLight;

    }

    @Override

    public void execute() {

        targetLight.turnOn();

    }

}

class LightOffCommand implements Command {

    private Light targetLight;

    public LightOffCommand(Light targetLight) {

        this.targetLight = targetLight;

    }

    @Override

    public void execute() {

        targetLight.turnOff();

    }

}

class Light {

    public void turnOn() {

        System.out.println("The light is on");

    }

    public void turnOff() {

        System.out.println("The light is off");

    }

}

class RemoteControl {

    private Command currentCommand;

    public void setCommand(Command command) {

        this.currentCommand = command;

    }

    public void pressButton() {

        currentCommand.execute();

    }

}

public class CommandPattern {

    public static void main(String[] args) {

        Light livingRoomLight = new Light();

        Command lightOnCommand = new LightOnCommand(livingRoomLight);

        Command lightOffCommand = new LightOffCommand(livingRoomLight);

        RemoteControl remoteControl = new RemoteControl();

        remoteControl.setCommand(lightOnCommand);

        remoteControl.pressButton();

        remoteControl.setCommand(lightOffCommand);

        remoteControl.pressButton();

    }

}

**Exercise 10: Implementing the MVC Pattern**

**Scenario:** You are developing a simple web application for managing student records using the MVC pattern.

class Student {

    private String studentId;

    private String studentName;

    private String studentGrade;

    public Student(String studentId, String studentName, String studentGrade) {

        this.studentId = studentId;

        this.studentName = studentName;

        this.studentGrade = studentGrade;

    }

    public String getStudentId() {

        return studentId;

    }

    public void setStudentId(String studentId) {

        this.studentId = studentId;

    }

    public String getStudentName() {

        return studentName;

    }

    public void setStudentName(String studentName) {

        this.studentName = studentName;

    }

    public String getStudentGrade() {

        return studentGrade;

    }

    public void setStudentGrade(String studentGrade) {

        this.studentGrade = studentGrade;

    }

}

class StudentView {

    public void displayStudentDetails(String studentName, String studentId, String studentGrade) {

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

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

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

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

    }

}

class StudentController {

    private Student model;

    private StudentView view;

    public StudentController(Student model, StudentView view) {

        this.model = model;

        this.view = view;

    }

    public void setStudentName(String studentName) {

        model.setStudentName(studentName);

    }

    public String getStudentName() {

        return model.getStudentName();

    }

    public void setStudentId(String studentId) {

        model.setStudentId(studentId);

    }

    public String getStudentId() {

        return model.getStudentId();

    }

    public void setStudentGrade(String studentGrade) {

        model.setStudentGrade(studentGrade);

    }

    public String getStudentGrade() {

        return model.getStudentGrade();

    }

    public void updateView() {

        view.displayStudentDetails(model.getStudentName(), model.getStudentId(), model.getStudentGrade());

    }

}

public class MVCPattern {

    public static void main(String[] args) {

        Student student = new Student("1", "John Doe", "A");

        StudentView view = new StudentView();

        StudentController controller = new StudentController(student, view);

        controller.updateView();

        controller.setStudentName("Jane Doe");

        controller.setStudentGrade("B");

        controller.updateView();

    }

}

**Exercise 11: Implementing Dependency Injection**

**Scenario:** You are developing a customer management application where the service class depends on a repository class. Use Dependency Injection to manage these dependencies.

interface CustomerRepository {

    String findCustomerById(String customerId);

}

class CustomerRepositoryImpl implements CustomerRepository {

    @Override

    public String findCustomerById(String customerId) {

        if (customerId.equals("1")) {

            return "John Doe";

        } else {

            return "Customer not found";

        }

    }

}

class CustomerService {

    private CustomerRepository customerRepo;

    public CustomerService(CustomerRepository customerRepo) {

        this.customerRepo = customerRepo;

    }

    public String getCustomerDetails(String customerId) {

        return customerRepo.findCustomerById(customerId);

    }

}

public class DependencyInjection {

    public static void main(String[] args) {

        var customerRepo = new CustomerRepositoryImpl();

        var customerService = new CustomerService(customerRepo);

        var customerDetails = customerService.getCustomerDetails("1");

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

    }

}