**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 {

    private static Logger loggerInstance;

    private Logger() {}

    public static Logger getInstance() {

        if (loggerInstance == null) {

            loggerInstance = new Logger();

        }

        return loggerInstance;

    }

    public void log(String message) {

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

    }

}

public class SingletonPatternExample {

    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.

// FactoryMethodPatternExample.java

public class FactoryMethodPatternExample {

    // Document interface

    public interface Document {

        void open();

        void close();

    }

    // Concrete Document Classes

    public static class WordDocument implements Document {

        @Override

        public void open() {

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

        }

        @Override

        public void close() {

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

        }

    }

    public static class PdfDocument implements Document {

        @Override

        public void open() {

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

        }

        @Override

        public void close() {

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

        }

    }

    public static class ExcelDocument implements Document {

        @Override

        public void open() {

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

        }

        @Override

        public void close() {

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

        }

    }

    // Document Factory

    public abstract static class DocumentFactory {

        public abstract Document createDocument();

    }

    // Concrete Factory Classes

    public static class WordDocumentFactory extends DocumentFactory {

        @Override

        public Document createDocument() {

            return new WordDocument();

        }

    }

    public static class PdfDocumentFactory extends DocumentFactory {

        @Override

        public Document createDocument() {

            return new PdfDocument();

        }

    }

    public static class ExcelDocumentFactory extends DocumentFactory {

        @Override

        public Document createDocument() {

            return new ExcelDocument();

        }

    }

    // Test the Factory Method Implementation

    public static void main(String[] args) {

        DocumentFactory wordFactory = new WordDocumentFactory();

        Document wordDocument = wordFactory.createDocument();

        wordDocument.open();

        wordDocument.close();

        DocumentFactory pdfFactory = new PdfDocumentFactory();

        Document pdfDocument = pdfFactory.createDocument();

        pdfDocument.open();

        pdfDocument.close();

        DocumentFactory excelFactory = new ExcelDocumentFactory();

        Document excelDocument = excelFactory.createDocument();

        excelDocument.open();

        excelDocument.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 BuilderPatternExample {

    public static class Computer {

        private String CPU;

        private String RAM;

        private String storage;

        private String graphicsCard;

        private String powerSupply;

        // Private constructor to enforce object creation through Builder

        private Computer(Builder builder) {

            this.CPU = builder.CPU;

            this.RAM = builder.RAM;

            this.storage = builder.storage;

            this.graphicsCard = builder.graphicsCard;

            this.powerSupply = builder.powerSupply;

        }

        @Override

        public String toString() {

            return "Computer{" +

                    "CPU='" + CPU + '\'' +

                    ", RAM='" + RAM + '\'' +

                    ", Storage='" + storage + '\'' +

                    ", Graphics Card='" + graphicsCard + '\'' +

                    ", Power Supply='" + powerSupply + '\'' +

                    '}';

        }

        // Static nested Builder class

        public static class Builder {

            private String CPU;

            private String RAM;

            private String storage;

            private String graphicsCard;

            private String powerSupply;

            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 setPowerSupply(String powerSupply) {

                this.powerSupply = powerSupply;

                return this;

            }

            public Computer build() {

                return new Computer(this);

            }

        }

    }

    // Test the Builder Implementation

    public static void main(String[] args) {

        // Create a Computer with specific configurations

        Computer gamingPC = new Computer.Builder()

                .setCPU("Intel i9")

                .setRAM("32GB")

                .setStorage("1TB SSD")

                .setGraphicsCard("NVIDIA RTX 3080")

                .setPowerSupply("750W")

                .build();

        Computer officePC = new Computer.Builder()

                .setCPU("Intel i5")

                .setRAM("16GB")

                .setStorage("512GB SSD")

                .build();

        // Display the configurations

        System.out.println("Gaming PC: " + gamingPC);

        System.out.println("Office PC: " + officePC);

    }

}

**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.

// AdapterPatternExample.java

public class AdapterPatternExample {

    // Target Interface

    public interface PaymentProcessor {

        void processPayment(double amount);

    }

    // Adaptee Classes

    public static class PayPal {

        public void makePayment(double amount) {

            System.out.println("Payment of $" + amount + " processed through PayPal.");

        }

    }

    public static class Stripe {

        public void pay(double amount) {

            System.out.println("Payment of $" + amount + " processed through Stripe.");

        }

    }

    // Adapter Classes

    public static class PayPalAdapter implements PaymentProcessor {

        private PayPal payPal;

        public PayPalAdapter(PayPal payPal) {

            this.payPal = payPal;

        }

        @Override

        public void processPayment(double amount) {

            payPal.makePayment(amount);

        }

    }

    public static class StripeAdapter implements PaymentProcessor {

        private Stripe stripe;

        public StripeAdapter(Stripe stripe) {

            this.stripe = stripe;

        }

        @Override

        public void processPayment(double amount) {

            stripe.pay(amount);

        }

    }

    // Test the Adapter Implementation

    public static void main(String[] args) {

        // Create instances of the payment gateways

        PayPal payPal = new PayPal();

        Stripe stripe = new Stripe();

        // Create adapter instances

        PaymentProcessor payPalProcessor = new PayPalAdapter(payPal);

        PaymentProcessor stripeProcessor = new StripeAdapter(stripe);

        // Use the adapters to process payments

        payPalProcessor.processPayment(100.00);

        stripeProcessor.processPayment(150.50);

    }

}

**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

// DecoratorPatternExample.java

public class DecoratorPatternExample {

    // Component Interface

    public interface Notifier {

        void send(String message);

    }

    // Concrete Component

    public static class EmailNotifier implements Notifier {

        @Override

        public void send(String message) {

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

        }

    }

    // Abstract Decorator Class

    public static abstract class NotifierDecorator implements Notifier {

        protected Notifier notifier;

        public NotifierDecorator(Notifier notifier) {

            this.notifier = notifier;

        }

        @Override

        public void send(String message) {

            notifier.send(message);

        }

    }

    // Concrete Decorator Classes

    public static class SMSNotifierDecorator extends NotifierDecorator {

        public SMSNotifierDecorator(Notifier notifier) {

            super(notifier);

        }

        @Override

        public void send(String message) {

            super.send(message); // Call the original notifier

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

        }

    }

    public static class SlackNotifierDecorator extends NotifierDecorator {

        public SlackNotifierDecorator(Notifier notifier) {

            super(notifier);

        }

        @Override

        public void send(String message) {

            super.send(message); // Call the original notifier

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

        }

    }

    // Test the Decorator Implementation

    public static void main(String[] args) {

        // Create a basic email notifier

        Notifier emailNotifier = new EmailNotifier();

        // Decorate it with SMS notification capability

        Notifier smsNotifier = new SMSNotifierDecorator(emailNotifier);

        // Decorate it further with Slack notification capability

        Notifier slackNotifier = new SlackNotifierDecorator(smsNotifier);

        // Send a message

        System.out.println("Sending notifications:");

        slackNotifier.send("Hello, World!");

    }

}

**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

// ProxyPatternExample.java

public class ProxyPatternExample {

    // Subject Interface

    public interface Image {

        void display();

    }

    // Real Subject Class

    public static class RealImage implements Image {

        private String filename;

        public RealImage(String filename) {

            loadImageFromServer(); // Simulating loading image from a remote server

            this.filename = filename;

        }

        private void loadImageFromServer() {

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

        }

        @Override

        public void display() {

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

        }

    }

    // Proxy Class

    public static 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

            }

            realImage.display(); // Display the image

        }

    }

    // Test the Proxy Implementation

    public static void main(String[] args) {

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

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

        // The image will be loaded from the server when displayed for the first time

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

        image1.display();

        // The image will be cached, so it won't be loaded from the server again

        System.out.println("\nSecond display of image1:");

        image1.display();

        // The image will be loaded from the server when displayed for the first time

        System.out.println("\nFirst display of image2:");

        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;

// ObserverPatternExample.java

// Subject Interface

interface Stock {

    void registerObserver(Observer observer);

    void deregisterObserver(Observer observer);

    void notifyObservers();

}

// Concrete Subject

class StockMarket implements Stock {

    private List<Observer> observers;

    private double stockPrice;

    public StockMarket() {

        observers = new ArrayList<>();

    }

    public void setStockPrice(double stockPrice) {

        this.stockPrice = stockPrice;

        notifyObservers();

    }

    @Override

    public void registerObserver(Observer observer) {

        observers.add(observer);

    }

    @Override

    public void deregisterObserver(Observer observer) {

        observers.remove(observer);

    }

    @Override

    public void notifyObservers() {

        for (Observer observer : observers) {

            observer.update(stockPrice);

        }

    }

}

// Observer Interface

interface Observer {

    void update(double stockPrice);

}

// Concrete Observers

class MobileApp implements Observer {

    @Override

    public void update(double stockPrice) {

        System.out.println("Mobile App: Stock price updated to $" + stockPrice);

    }

}

class WebApp implements Observer {

    @Override

    public void update(double stockPrice) {

        System.out.println("Web App: Stock price updated to $" + stockPrice);

    }

}

// Test the Observer Implementation

public class ObserverPatternExample {

    public static void main(String[] args) {

        StockMarket stockMarket = new StockMarket();

        MobileApp mobileApp = new MobileApp();

        WebApp webApp = new WebApp();

        // Register observers

        stockMarket.registerObserver(mobileApp);

        stockMarket.registerObserver(webApp);

        // Update stock price

        System.out.println("Updating stock price to $100.00");

        stockMarket.setStockPrice(100.00);

        // Update stock price again

        System.out.println("\nUpdating stock price to $150.00");

        stockMarket.setStockPrice(150.00);

        // Deregister mobile app

        stockMarket.deregisterObserver(mobileApp);

        // Update stock price again

        System.out.println("\nUpdating stock price to $200.00");

        stockMarket.setStockPrice(200.00);

    }

}

**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.

// StrategyPatternExample.java

// Strategy Interface

interface PaymentStrategy {

    void pay(String amount);

}

// Concrete Strategy for Credit Card Payment

class CreditCardPayment implements PaymentStrategy {

    private String cardNumber;

    public CreditCardPayment(String cardNumber) {

        this.cardNumber = cardNumber;

    }

    @Override

    public void pay(String amount) {

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

    }

}

// Concrete Strategy for PayPal Payment

class PayPalPayment implements PaymentStrategy {

    private String email;

    public PayPalPayment(String email) {

        this.email = email;

    }

    @Override

    public void pay(String amount) {

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

    }

}

// Context Class

class PaymentContext {

    private PaymentStrategy paymentStrategy;

    public void setPaymentStrategy(PaymentStrategy paymentStrategy) {

        this.paymentStrategy = paymentStrategy;

    }

    public void executePayment(String amount) {

        if (paymentStrategy != null) {

            paymentStrategy.pay(amount);

        } else {

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

        }

    }

}

// Test the Strategy Implementation

public class StrategyPatternExample {

    public static void main(String[] args) {

        PaymentContext context = new PaymentContext();

        // Paying with Credit Card

        System.out.println("Using Credit Card Payment:");

        PaymentStrategy creditCard = new CreditCardPayment("1234-5678-9876-5432");

        context.setPaymentStrategy(creditCard);

        context.executePayment("$100");

        // Paying with PayPal

        System.out.println("\nUsing PayPal Payment:");

        PaymentStrategy paypal = new PayPalPayment("user@example.com");

        context.setPaymentStrategy(paypal);

        context.executePayment("$150");

        // Trying without setting a payment strategy

        System.out.println("\nTrying with no Payment Strategy:");

        context.executePayment("$200");

    }

}

**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.

// CommandPatternExample.java

// Command Interface

interface Command {

    void execute();

}

// Receiver Class

class Light {

    public void turnOn() {

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

    }

    public void turnOff() {

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

    }

}

// Concrete 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();

    }

}

// Concrete 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;

    }

    public void pressButton() {

        command.execute();

    }

}

// Test the Command Implementation

public class CommandPatternExample {

    public static void main(String[] args) {

        Light livingRoomLight = new Light();

        // Create command objects

        Command lightOn = new LightOnCommand(livingRoomLight);

        Command lightOff = new LightOffCommand(livingRoomLight);

        // Create invoker

        RemoteControl remote = new RemoteControl();

        // Turn on the light

        remote.setCommand(lightOn);

        remote.pressButton();

        // Turn off the light

        remote.setCommand(lightOff);

        remote.pressButton();

    }

}

**Exercise 10: Implementing the MVC Pattern**

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

// Model Class

class Student {

    private String name;

    private int id;

    private String grade;

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

        this.name = name;

        this.id = id;

        this.grade = grade;

    }

    public String getName() {

        return name;

    }

    public void setName(String name) {

        this.name = name;

    }

    public int getId() {

        return id;

    }

    public void setId(int id) {

        this.id = id;

    }

    public String getGrade() {

        return grade;

    }

    public void setGrade(String grade) {

        this.grade = grade;

    }

}

// View Class

class StudentView {

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

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

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

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

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

    }

}

// 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 String getStudentName() {

        return model.getName();

    }

    public void setStudentId(int id) {

        model.setId(id);

    }

    public int getStudentId() {

        return model.getId();

    }

    public void setStudentGrade(String grade) {

        model.setGrade(grade);

    }

    public String getStudentGrade() {

        return model.getGrade();

    }

    public void updateView() {

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

    }

}

// Main Class to Test the MVC Implementation

public class MVCPatternExample {

    public static void main(String[] args) {

        // Create a Student model

        Student student = new Student("Alice", 1, "A");

        // Create a View for the Student

        StudentView view = new StudentView();

        // Create a Controller

        StudentController controller = new StudentController(student, view);

        // Display initial student details

        controller.updateView();

        // Update student details

        controller.setStudentName("Bob");

        controller.setStudentId(2);

        controller.setStudentGrade("B");

        // Display updated student details

        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.

// Repository Interface

interface CustomerRepository {

    String findCustomerById(int id);

}

// Concrete Repository Implementation

class CustomerRepositoryImpl implements CustomerRepository {

    @Override

    public String findCustomerById(int id) {

        // In a real application, this might query a database

        // For this example, we will just return a dummy customer

        return "Customer" + id; // Dummy customer data

    }

}

// Service Class

class CustomerService {

    private final CustomerRepository customerRepository;

    // Constructor Injection

    public CustomerService(CustomerRepository customerRepository) {

        this.customerRepository = customerRepository;

    }

    public String getCustomer(int id) {

        return customerRepository.findCustomerById(id);

    }

}

// Main Class to Test the Dependency Injection Implementation

public class DependencyInjectionExample {

    public static void main(String[] args) {

        // Create the concrete repository

        CustomerRepository customerRepository = new CustomerRepositoryImpl();

        // Inject the repository into the service using constructor injection

        CustomerService customerService = new CustomerService(customerRepository);

        // Use the service to find a customer

        String customer = customerService.getCustomer(1);

        System.out.println("Found Customer: " + customer);

    }

}