**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.

**Code:**

public class Logger {

    private static Logger l;

    private Logger(){

    }

    public static Logger getInstance(){

        if(l==null){

            l=new Logger();

        }

        return l;

    }

}

public class Test {

    public static void main(String args[]){

        Logger user1=Logger.getInstance();

        Logger user2=Logger.getInstance();

        if(user1==user2){

            System.out.println("Both the users are same");

        }

        else{

        System.out.println("Both are not same");

        }

    }

}

**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.

public interface Document {

    public void Writing();

}

public class DocumentFactory {

    public Document create(String s){

        if(s.equals("Word")){

             new WordDocument().Writing();;

        }

        if(s.equals("Pdf")){

             new PdfDocument().Writing();

        }

        if(s.equals("Excel")){

             new ExcelDocument().Writing();;

        }

        return new WordDocument();

    }

}

public class ExcelDocument implements Document{

    public void Writing(){

        System.out.println("I am Writing in Excel Documents");

    }

}

public class FactoryExcelDocument extends DocumentFactory {

    public Document createDocument(){

        return new ExcelDocument();

    }

}

public class FactoryTest {

    public static void main(String args[]){

        FactoryWordDocument w=new FactoryWordDocument();

        w.createDocument();

    }

}

public class WordDocument implements Document{

    public void Writing(){

        System.out.println("I am Writing in Word Documents");

    }

}

public class Test {

    public static void main(String[] args) {

        DocumentFactory doc =new DocumentFactory();

        doc.create("Pdf");

        doc.create("Word");

        doc.create("Excel");

    }

}

public class PdfDocument implements Document{

    public void Writing(){

        System.out.println("I am Writing in Pdf Documents");

    }

}

public class FactoryWordDocument extends DocumentFactory {

    public Document createDocument(){

        return new WordDocument();

    }

}

**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.

**Code:**

public class Builder {

    public String CPU;

    public String RAM;

    public String Storage;

    public String Processor;

    public Builder(String s){

        this.CPU=s;

    }

    public Builder setRAM(String s){

        this.RAM=s;

        return this;

    }

    public Builder setStorage(String s){

        this.Storage=s;

        return this;

    }

    public Builder setProcessor(String s){

        this.Processor=s;

        return this;

    }

    public Computer build(){

        return new Computer(this);

    }

}

public class BuilderImplementation {

    public static void main(String args[]){

        Computer c=new Builder("Intel")

                    .setProcessor("i7")

                    .setRAM("32")

                    .setStorage("512")

                    .build();

        c.getDetails();

    }

}

public class Computer {

    private String CPU;

    private String RAM;

    private String Storage;

    private String Processor;

    Computer(Builder builder){

        this.CPU=builder.CPU;

        this.RAM=builder.RAM;

        this.Storage=builder.Storage;

        this.Processor=builder.Processor;

    }

    public void getDetails(){

        System.out.println("RAM:"+this.RAM);

        System.out.println("CPU:"+this.CPU);

        System.out.println("Storage:"+this.Storage);

        System.out.println("Processor:"+this.Processor);

    }

}

**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 PaymentHandler {

void handlePayment(double amount);

}

class PayPalService {

public void executePayment(double amount) {

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

}

}

class StripeService {

public void execute(double amount) {

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

}

}

class AmazonPayService {

public void process(double amount) {

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

}

}

class PayPalHandlerAdapter implements PaymentHandler {

private PayPalService payPalService;

public PayPalHandlerAdapter(PayPalService payPalService) {

this.payPalService = payPalService;

}

public void handlePayment(double amount) {

payPalService.executePayment(amount);

}

}

class StripeHandlerAdapter implements PaymentHandler {

private StripeService stripeService;

public StripeHandlerAdapter(StripeService stripeService) {

this.stripeService = stripeService;

}

public void handlePayment(double amount) {

stripeService.execute(amount);

}

}

class AmazonPayHandlerAdapter implements PaymentHandler {

private AmazonPayService amazonPayService;

public AmazonPayHandlerAdapter(AmazonPayService amazonPayService) {

this.amazonPayService = amazonPayService;

}

public void handlePayment(double amount) {

amazonPayService.process(amount);

}

}

public class AdapterPatternExample {

public static void main(String[] args) {

PayPalService payPalService = new PayPalService();

StripeService stripeService = new StripeService();

AmazonPayService amazonPayService = new AmazonPayService();

PaymentHandler payPalAdapter = new PayPalHandlerAdapter(payPalService);

PaymentHandler stripeAdapter = new StripeHandlerAdapter(stripeService);

PaymentHandler amazonPayAdapter = new AmazonPayHandlerAdapter(amazonPayService);

payPalAdapter.handlePayment(400.00);

stripeAdapter.handlePayment(500.00);

amazonPayAdapter.handlePayment(600.00);

}

}

**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);

        sendSMS(message);

    }

    private void sendSMS(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);

        sendSlack(message);

    }

    private void sendSlack(String message) {

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

    }

}

public class DecoratorPatternExample {

    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 imageFilename;

    public RealImage(String imageFilename) {

        this.imageFilename = imageFilename;

        loadImageFromDisk();

    }

    private void loadImageFromDisk() {

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

    }

    public void display() {

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

    }

}

class ProxyImage implements Image {

    private String imageFilename;

    private RealImage realImageInstance;

    public ProxyImage(String imageFilename) {

        this.imageFilename = imageFilename;

    }

    public void display() {

        if (realImageInstance == null) {

            realImageInstance = new RealImage(imageFilename);

        }

        realImageInstance.display();

    }

}

public class ProxyPatternExample {

    public static void main(String[] args) {

        Image proxyImage1 = new ProxyImage("image1.jpg");

        Image proxyImage2 = new ProxyImage("image2.jpg");

        proxyImage1.display();

        System.out.println("");

        proxyImage1.display();

        System.out.println("");

        proxyImage2.display();

        System.out.println("");

        proxyImage2.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 addObserver(Observer observer);

    void removeObserver(Observer observer);

    void notifyObservers();

}

class StockExchange implements Stock {

    private List<Observer> observers;

    private double currentStockPrice;

    public StockExchange() {

        this.observers = new ArrayList<>();

    }

    @Override

    public void addObserver(Observer observer) {

        observers.add(observer);

    }

    @Override

    public void removeObserver(Observer observer) {

        observers.remove(observer);

    }

    @Override

    public void notifyObservers() {

        for (Observer observer : observers) {

            observer.update(currentStockPrice);

        }

    }

    public void updateStockPrice(double newStockPrice) {

        this.currentStockPrice = newStockPrice;

        notifyObservers();

    }

}

interface Observer {

    void update(double stockPrice);

}

class MobileNotification implements Observer {

    private String appName;

    public MobileNotification(String appName) {

        this.appName = appName;

    }

    @Override

    public void update(double stockPrice) {

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

    }

}

class WebNotification implements Observer {

    private String appName;

    public WebNotification(String appName) {

        this.appName = appName;

    }

    @Override

    public void update(double stockPrice) {

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

    }

}

public class ObserverPatternExample {

    public static void main(String[] args) {

        StockExchange stockExchange = new StockExchange();

        Observer mobileNotification = new MobileNotification("MobileApp");

        Observer webNotification = new WebNotification("WebApp");

        stockExchange.addObserver(mobileNotification);

        stockExchange.addObserver(webNotification);

        stockExchange.updateStockPrice(105.00);

        stockExchange.updateStockPrice(109.50);

       stockExchange.removeObserver(webNotification);

        stockExchange.updateStockPrice(102.50);

    }

}

**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);

}

// Implement Concrete Strategies

class CreditCardPayment implements PaymentStrategy {

    private String cardHolderName;

    private String cardNumber;

    private String cvv;

    private String expiryDate;

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

        this.cardHolderName = cardHolderName;

        this.cardNumber = cardNumber;

        this.cvv = cvv;

        this.expiryDate = expiryDate;

    }

    @Override

    public void pay(double amount) {

        // Here, you would normally process the payment through a payment gateway

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

    }

}

class PayPalPayment implements PaymentStrategy {

    private String emailAddress;

    private String password;

    public PayPalPayment(String emailAddress, String password) {

        this.emailAddress = emailAddress;

        this.password = password;

    }

    @Override

    public void pay(double amount) {

        // Here, you would normally interact with the PayPal API to process the payment

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

    }

}

// Implement Context Class

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);

    }

}

// Test the Strategy Implementation

public class StrategyPatternExample {

    public static void main(String[] args) {

        PaymentContext paymentContext = new PaymentContext();

        // Pay using Credit Card

        paymentContext.setPaymentStrategy(new CreditCardPayment("Ankitha", "1234567898", "123", "12/25 "));

        paymentContext.executePayment(500.0);

        // Pay using PayPal

        paymentContext.setPaymentStrategy(new PayPalPayment("Ankitha@example.com", "password294"));

        paymentContext.executePayment(300.0);

    }

}

**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 TurnOnLightCommand implements Command {

    private Light light;

    public TurnOnLightCommand(Light light) {

        this.light = light;

    }

    @Override

    public void execute() {

        light.turnOn();

    }

}

class TurnOffLightCommand implements Command {

    private Light light;

    public TurnOffLightCommand(Light light) {

        this.light = light;

    }

    @Override

    public void execute() {

        light.turnOff();

    }

}

class RemoteControl {

    private Command command;

    public void setCommand(Command command) {

        this.command = command;

    }

    public void pressButton() {

        if (command != null) {

            command.execute();

        }

    }

}

class Light {

    public void turnOn() {

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

    }

    public void turnOff() {

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

    }

}

class TestCommandPattern {

    public static void main(String[] args) {

        Light livingRoomLight = new Light();

        Command turnOn = new TurnOnLightCommand(livingRoomLight);

        Command turnOff = new TurnOffLightCommand(livingRoomLight);

        RemoteControl remote = new RemoteControl();

        remote.setCommand(turnOn);

        remote.pressButton();

        remote.setCommand(turnOff);

        remote.pressButton();

    }

}

**Exercise 10: Implementing the MVC Pattern**

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

class Learner {

    private String identifier;

    private String fullName;

    private String classGrade;

    public Learner(String identifier, String fullName, String classGrade) {

        this.identifier = identifier;

        this.fullName = fullName;

        this.classGrade = classGrade;

    }

    public String getIdentifier() {

        return identifier;

    }

    public void setIdentifier(String identifier) {

        this.identifier = identifier;

    }

    public String getFullName() {

        return fullName;

    }

    public void setFullName(String fullName) {

        this.fullName = fullName;

    }

    public String getClassGrade() {

        return classGrade;

    }

    public void setClassGrade(String classGrade) {

        this.classGrade = classGrade;

    }

}

class LearnerView {

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

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

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

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

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

    }

}

class LearnerController {

    private Learner model;

    private LearnerView view;

    public LearnerController(Learner model, LearnerView view) {

        this.model = model;

        this.view = view;

    }

    public void setLearnerName(String name) {

        model.setFullName(name);

    }

    public String getLearnerName() {

        return model.getFullName();

    }

    public void setLearnerId(String id) {

        model.setIdentifier(id);

    }

    public String getLearnerId() {

        return model.getIdentifier();

    }

    public void setLearnerGrade(String grade) {

        model.setClassGrade(grade);

    }

    public String getLearnerGrade() {

        return model.getClassGrade();

    }

    public void updateView() {

        view.showLearnerDetails(model.getFullName(), model.getIdentifier(), model.getClassGrade());

    }

}

public class MVCPatternExample {

    public static void main(String[] args) {

        Learner model = new Learner("1", "Ankitha", "A");

        LearnerView view = new LearnerView();

        LearnerController controller = new LearnerController(model, view);

        controller.updateView();

        controller.setLearnerName("Sirisha");

        controller.setLearnerGrade("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 ClientRepository {

    String findClientById(String id);

}

class ClientRepositoryImpl implements ClientRepository {

    public String findClientById(String id) {

        if (id.equals("1")) {

            return "Sirisha";

        } else {

            return "Client not found";

        }

    }

}

class ClientService {

    private ClientRepository clientRepository;

    public ClientService(ClientRepository clientRepository) {

        this.clientRepository = clientRepository;

    }

    public String getClientDetails(String id) {

        return clientRepository.findClientById(id);

    }

}

public class DependencyInjectionDemo {

    public static void main(String[] args) {

        ClientRepository clientRepository = new ClientRepositoryImpl();

        ClientService clientService = new ClientService(clientRepository);

        String clientDetails = clientService.getClientDetails("1");

        System.out.println("Client Details: " + clientDetails);

    }

}