**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);

    }

}

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 WordDoc implements Document {

    public void open() {

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

    }

    public void close() {

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

    }

}

class PdfDoc implements Document {

    public void open() {

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

    }

    public void close() {

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

    }

}

class ExcelDoc implements Document {

    public void open() {

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

    }

    public void close() {

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

    }

}

abstract class DocFactory {

    public abstract Document createDocument();

}

class WordDocFactory extends DocFactory {

    public Document createDocument() {

        return new WordDoc();

    }

}

class PdfDocFactory extends DocFactory {

    public Document createDocument() {

        return new PdfDoc();

    }

}

class ExcelDocFactory extends DocFactory {

    public Document createDocument() {

        return new ExcelDoc();

    }

}

public class FactoryMethodPattern {

    public static void main(String[] args) {

        DocFactory wordFactory = new WordDocFactory();

        Document wordDoc = wordFactory.createDocument();

        wordDoc.open();

        wordDoc.close();

        DocFactory pdfFactory = new PdfDocFactory();

        Document pdfDoc = pdfFactory.createDocument();

        pdfDoc.open();

        pdfDoc.close();

        DocFactory excelFactory = new ExcelDocFactory();

        Document excelDoc = excelFactory.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 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);

            }

        }

    }

    public static void main(String[] args) {

        Computer gamingPc = new Computer.Builder()

                .setCpu("Intel Core i9")

                .setRam("32GB")

                .setStorage("1TB SSD")

                .build();

        System.out.println("CPU: " + gamingPc.cpu);

        System.out.println("RAM: " + gamingPc.ram);

        System.out.println("Storage: " + gamingPc.storage);

    }

}

**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 payPal;

    public PayPalAdapter(PayPal payPal) {

        this.payPal = payPal;

    }

    public void processPayment(double amount) {

        payPal.makePayment(amount);

    }

}

class StripeAdapter implements PaymentProcessor {

    private Stripe stripe;

    public StripeAdapter(Stripe stripe) {

        this.stripe = stripe;

    }

    public void processPayment(double amount) {

        stripe.pay(amount);

    }

}

class AmazonPayAdapter implements PaymentProcessor {

    private AmazonPay amazonPay;

    public AmazonPayAdapter(AmazonPay amazonPay) {

        this.amazonPay = amazonPay;

    }

    public void processPayment(double amount) {

        amazonPay.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 notifier;

    public NotifierDecorator(Notifier notifier) {

        this.notifier = notifier;

    }

    public void send(String message) {

        notifier.send(message);

    }

}

class SmsNotifierDecorator extends NotifierDecorator {

    public SmsNotifierDecorator(Notifier notifier) {

        super(notifier);

    }

    public void send(String message) {

        notifier.send(message);

        sendSms(message);

    }

    private void sendSms(String message) {

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

    }

}

class SlackNotifierDecorator extends NotifierDecorator {

    public SlackNotifierDecorator(Notifier notifier) {

        super(notifier);

    }

    public void send(String message) {

        notifier.send(message);

        sendSlack(message);

    }

    private void sendSlack(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);

    }

    public void display() {

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

    }

}

class ProxyImage implements Image {

    private String filename;

    private RealImage realImage;

    public ProxyImage(String filename) {

        this.filename = filename;

    }

    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");

        image1.display();

        System.out.println("");

        image1.display();

        System.out.println("");

        image2.display();

        System.out.println("");

        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> observers;

    private double stockPrice;

    public StockMarket() {

        this.observers = new ArrayList<>();

    }

    public void registerObserver(Observer observer) {

        observers.add(observer);

    }

    public void deregisterObserver(Observer observer) {

        observers.remove(observer);

    }

    public void notifyObservers() {

        for (Observer observer : observers) {

            observer.update(stockPrice);

        }

    }

    public void setStockPrice(double stockPrice) {

        this.stockPrice = stockPrice;

        notifyObservers();

    }

}

interface Observer {

    void update(double stockPrice);

}

class MobileApp implements Observer {

    private String appName;

    public MobileApp(String appName) {

        this.appName = appName;

    }

    public void update(double stockPrice) {

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

    }

}

class WebApp implements Observer {

    private String appName;

    public WebApp(String appName) {

        this.appName = appName;

    }

    public void update(double stockPrice) {

        System.out.println(appName + " 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;

}

public void pay(double amount) {

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

}

}

class PayPalPayment implements PaymentStrategy {

private String email;

private String password;

public PayPalPayment(String email, String password) {

this.email = email;

this.password = password;

}

public void pay(double amount) {

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

}

}

class PaymentContext {

private PaymentStrategy paymentStrategy;

public void setPaymentStrategy(PaymentStrategy paymentStrategy) {

this.paymentStrategy = paymentStrategy;

}

public void executePayment(double amount) {

paymentStrategy.pay(amount);

}

}

public class StrategyPattern {

public static void main(String[] args) {

PaymentContext paymentContext = new PaymentContext();

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

paymentContext.executePayment(100.0);

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 light;

public LightOnCommand(Light light) {

this.light = light;

}

@Override

public void execute() {

light.turnOn();

}

}

class LightOffCommand implements Command {

private Light light;

public LightOffCommand(Light light) {

this.light = light;

}

@Override

public void execute() {

light.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 command;

public void setCommand(Command command) {

this.command = command;

}

public void pressButton() {

command.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 id;

private String name;

private String grade;

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

this.id = id;

this.name = name;

this.grade = grade;

}

public String getId() {

return id;

}

public void setId(String id) {

this.id = id;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public String getGrade() {

return grade;

}

public void setGrade(String grade) {

this.grade = grade;

}

}

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);

}

}

class StudentController {

private Student student;

private StudentView view;

public StudentController(Student student, StudentView view) {

this.student = student;

this.view = view;

}

public void setStudentName(String name) {

student.setName(name);

}

public String getStudentName() {

return student.getName();

}

public void setStudentId(String id) {

student.setId(id);

}

public String getStudentId() {

return student.getId();

}

public void setStudentGrade(String grade) {

student.setGrade(grade);

}

public String getStudentGrade() {

return student.getGrade();

}

public void updateView() {

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

}

}

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 id);

}

class CustomerRepositoryImpl implements CustomerRepository {

@Override

public String findCustomerById(String id) {

if (id.equals("1")) {

return "John Doe";

} else {

return "Customer not found";

}

}

}

class CustomerService {

private CustomerRepository repository;

public CustomerService(CustomerRepository repository) {

this.repository = repository;

}

public String getCustomerDetails(String id) {

return repository.findCustomerById(id);

}

}

public class DependencyInjection {

public static void main(String[] args) {

var repository = new CustomerRepositoryImpl();

var service = new CustomerService(repository);

var details = service.getCustomerDetails("1");

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

}

}