Assignment 1: Design Patterns.

**Exercise 1: Implementing the Singleton Pattern.**

**Code:**

class Logger {

private static Logger instance;

private Logger() {

System.out.println("Constructor is initialised");

}

public static Logger getInstance() {

if (instance == null) {

instance = new Logger();

}

return instance;

}

public void log(String message) {

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

}}

public class Main {

public static void main(String[] args) {

Logger logger1 = Logger.getInstance();

Logger logger2 = Logger.getInstance();

logger1.log("Instance one of SingleTon");

logger2.log("Instance Two of SingleTon");

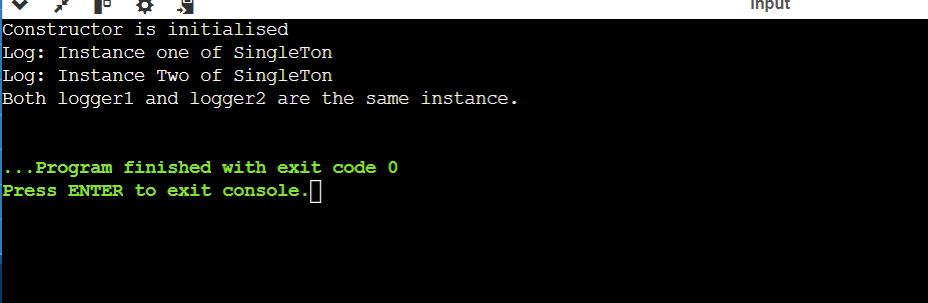
if (logger1 == logger2) {

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**

Code:

interface Document {

void open();

void close();

}

class WordDocument implements Document {

public void open() {

System.out.println("Document Opened.");

}

public void close() {

System.out.println("Document Closed.");

}

}

class PdfDocument implements Document {

public void open() {

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

}

public void close() {

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

}

}

class ExcelDocument implements Document {

public void open() {

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

}

public void close() {

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

}

}

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();

}

}

class FactoryMethodTest {

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();

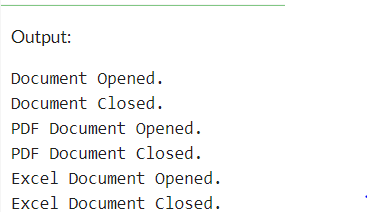
Document excelDocument = excelFactory.createDocument();

excelDocument.open();

excelDocument.close();

}

}



**Exercise 3: Implementing the Builder Pattern**:

Code:

class Computer {

private String cpu;

private int ram;

private int storage;

private Computer(Builder builder) {

this.cpu = builder.cpu;

this.ram = builder.ram;

this.storage = builder.storage;

}

public String getCpu() {

return cpu;

}

public int getRam() {

return ram;

}

public int getStorage() {

return storage;

}

public static class Builder {

private String cpu;

private int ram;

private int storage;

public Builder setCpu(String cpu) {

this.cpu = cpu;

return this;

}

public Builder setRam(int ram) {

this.ram = ram;

return this;

}

public Builder setStorage(int storage) {

this.storage = storage;

return this;

}

public Computer build() {

return new Computer(this);

}

}

}

class BuilderPatternTest {

public static void main(String[] args) {

// Create a computer with default configuration

Computer defaultComputer = new Computer.Builder().build();

System.out.println("Default Computer: " + defaultComputer.getCpu() + ", " + defaultComputer.getRam() + "GB, " + defaultComputer.getStorage() + "GB");

// Create a computer with custom configuration

Computer customComputer = new Computer.Builder()

.setCpu("Intel Core i7")

.setRam(16)

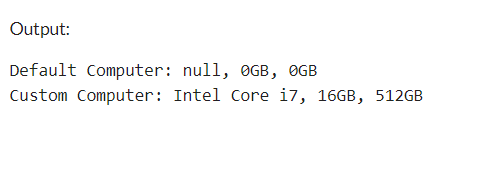
.setStorage(512)

.build();

System.out.println("Custom Computer: " + customComputer.getCpu() + ", " + customComputer.getRam() + "GB, " + customComputer.getStorage() + "GB");

}

}



**Exercise 4: Implementing the Adapter Pattern**

**Code:**

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 charge(double amount) {

System.out.println("Charging Rs" + amount + " using Stripe.");

}

}

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.charge(amount);

}

}

class PaymentProcessorTest {

public static void main(String[] args) {

PayPal payPal = new PayPal();

Stripe stripe = new Stripe();

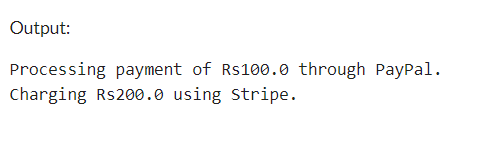
PaymentProcessor payPalAdapter = new PayPalAdapter(payPal);

PaymentProcessor stripeAdapter = new StripeAdapter(stripe);

payPalAdapter.processPayment(100);

stripeAdapter.processPayment(200);

}

}

**Exercise 5: Implementing the Decorator Pattern**

Code:

interface Notifier {

void send(String message);

}

class EmailNotifier implements Notifier {

public void send(String message) {

System.out.println("Received 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) {

super.send(message);

sendSMS(message);

}

private void sendSMS(String message) {

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

}

}

class SlackNotifierDecorator extends NotifierDecorator {

public SlackNotifierDecorator(Notifier notifier) {

super(notifier);

}

public void send(String message) {

super.send(message);

sendSlackMessage(message);

}

private void sendSlackMessage(String message) {

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

}

}

class NotifierTest {

public static void main(String[] args) {

Notifier emailNotifier = new EmailNotifier();

emailNotifier.send("Hello from email!");

Notifier smsNotifier = new SMSNotifierDecorator(emailNotifier);

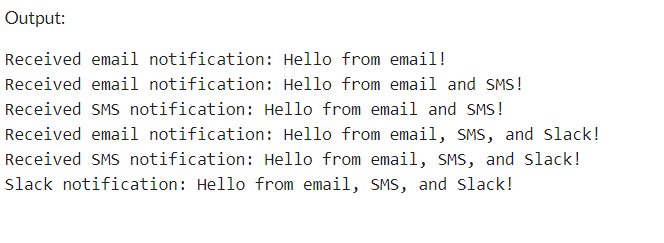
smsNotifier.send("Hello from email and SMS!");

Notifier slackNotifier = new SlackNotifierDecorator(smsNotifier);

slackNotifier.send("Hello from email, SMS, and Slack!");

}

}



**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;

loadImageFromDisk();

}

private void loadImageFromDisk() {

System.out.println("Loading " + fileName);

}

public void display() {

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

}

}

class ProxyImage implements Image {

private RealImage realImage;

private String fileName;

public ProxyImage(String fileName) {

this.fileName = fileName;

}

public void display() {

if (realImage == null) {

realImage = new RealImage(fileName);

}

realImage.display();

}

}

class ProxyPatternDemo {

public static void main(String[] args) {

Image image1 = new ProxyImage("HiRes\_10MB\_Photo1.jpg");

Image image2 = new ProxyImage("HiRes\_10MB\_Photo2.jpg");

image1.display();

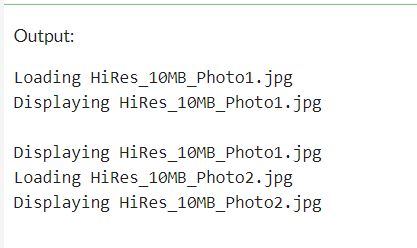
System.out.println("");

image1.display();

image2.display();

}

}



**Exercise 7: Implementing the Observer Pattern**

**Code:**

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 String stockName;

private float stockPrice;

public StockMarket(String stockName) {

observers = new ArrayList<>();

this.stockName = stockName;

}

public void setStockPrice(float stockPrice) {

this.stockPrice = stockPrice;

notifyObservers();

}

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(stockName, stockPrice);

}

}

}

interface Observer {

void update(String stockName, float stockPrice);

}

class MobileApp implements Observer {

public void update(String stockName, float stockPrice) {

System.out.println("Mobile App: The price of " + stockName + " is now $" + stockPrice);

}

}

class WebApp implements Observer {

public void update(String stockName, float stockPrice) {

System.out.println("Web App: The price of " + stockName + " is now $" + stockPrice);

}

}

class ObserverPatternDemo {

public static void main(String[] args) {

StockMarket stockMarket = new StockMarket("AAPL");

MobileApp mobileApp = new MobileApp();

WebApp webApp = new WebApp();

stockMarket.registerObserver(mobileApp);

stockMarket.registerObserver(webApp);

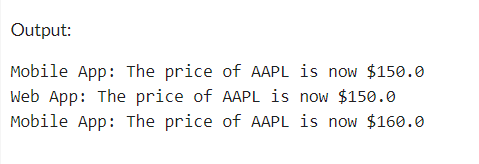
stockMarket.setStockPrice(150.00f);

stockMarket.deregisterObserver(webApp);

stockMarket.setStockPrice(160.00f);

}

}



**Exercise 8: Implementing the Strategy Pattern**

**Code:**

interface PaymentStrategy {

void pay(int amount);

}

class CreditCardPayment implements PaymentStrategy {

private String cardNumber;

private String name;

public CreditCardPayment(String cardNumber, String name) {

this.cardNumber = cardNumber;

this.name = name;

}

public void pay(int amount) {

System.out.println(amount + " paid using credit card: " + cardNumber);

}

}

class PayPalPayment implements PaymentStrategy {

private String email;

public PayPalPayment(String email) {

this.email = email;

}

public void pay(int amount) {

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

}

}

class PaymentContext {

private PaymentStrategy paymentStrategy;

public void setPaymentStrategy(PaymentStrategy paymentStrategy) {

this.paymentStrategy = paymentStrategy;

}

public void executePayment(int amount) {

paymentStrategy.pay(amount);

}

}

class StrategyPatternDemo {

public static void main(String[] args) {

PaymentContext context = new PaymentContext();

context.setPaymentStrategy(new CreditCardPayment("1234-5678-9012-3456", "John Doe"));

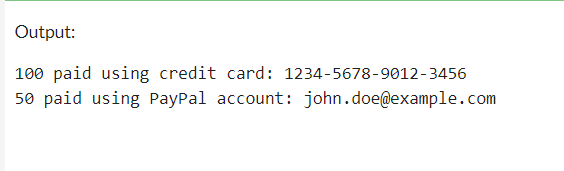
context.executePayment(100);

context.setPaymentStrategy(new PayPalPayment("john.doe@example.com"));

context.executePayment(50);

}

}



**Exercise 9: Implementing the Command Pattern**

**Code:**

interface Command {

void execute();

}

class LightOnCommand implements Command {

private Light light;

public LightOnCommand(Light light) {

this.light = light;

}

public void execute() {

light.turnOn();

}

}

class LightOffCommand implements Command {

private Light light;

public LightOffCommand(Light light) {

this.light = light;

}

public void execute() {

light.turnOff();

}

}

class RemoteControl {

private Command command;

public void setCommand(Command command) {

this.command = command;

}

public void pressButton() {

command.execute();

}

}

class Light {

public void turnOn() {

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

}

public void turnOff() {

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

}

}

class CommandPatternTest {

public static void main(String[] args) {

Light light = new Light();

Command lightOnCommand = new LightOnCommand(light);

Command lightOffCommand = new LightOffCommand(light);

RemoteControl remoteControl = new RemoteControl();

// Turn on the light

remoteControl.setCommand(lightOnCommand);

remoteControl.pressButton();

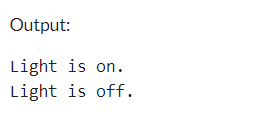
// Turn off the light

remoteControl.setCommand(lightOffCommand);

remoteControl.pressButton();

}

}



**Exercise 10: Implementing the MVC Pattern**

**Code:**

class Student {

private String name;

private int id;

private int 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 int getGrade() {

return grade;

}

public void setGrade(int grade) {

this.grade = grade;

}

}

class StudentView {

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

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

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

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

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

}

}

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(int grade) {

model.setGrade(grade);

}

public int getStudentGrade() {

return model.getGrade();

}

public void updateView() {

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

}

}

class MVCPatternTest {

public static void main(String[] args) {

Student model = new Student();

StudentView view = new StudentView();

StudentController controller = new StudentController(model, view);

controller.setStudentName("John");

controller.setStudentId(1);

controller.setStudentGrade(85);

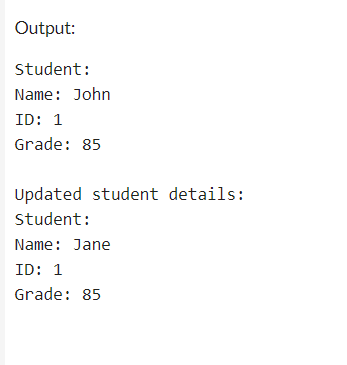
controller.updateView();

controller.setStudentName("Jane");

System.out.println("\nUpdated student details:");

controller.updateView();

}

}

**Exercise 11: Implementing Dependency Injection**

**Code:**

interface CustomerRepository {

String findCustomerById(int id);

}

class CustomerRepositoryImpl implements CustomerRepository {

public String findCustomerById(int id) {

return "Customer with ID: " + id;

}

}

class CustomerService {

private final CustomerRepository customerRepository;

public CustomerService(CustomerRepository customerRepository) {

this.customerRepository = customerRepository;

}

public String getCustomer(int id) {

return customerRepository.findCustomerById(id);

}

}

class DependencyInjectionTest {

public static void main(String[] args) {

CustomerRepository customerRepository = new CustomerRepositoryImpl();

CustomerService customerService = new CustomerService(customerRepository);

String customer = customerService.getCustomer(1);

System.out.println(customer);

}

}

