ASSIGNMENT-1

DESIGN PATTERNS & PRINCIPLES

**Exercise 1: Implementing the Singleton Pattern**

**CODE:**

class Logger {

private static Logger instance;

private Logger() {

// Initialization

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

}

public static Logger getInstance() {

if (instance == null) {

instance = new Logger();

}

return instance;

}

public void log(String message) {

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

}}

public class SingletonTest {

public static void main(String[] args) {

Logger logger1 = Logger.getInstance();

Logger logger2 = Logger.getInstance();

logger1.log("Hello World.");

logger2.log("Singleton instance created.");

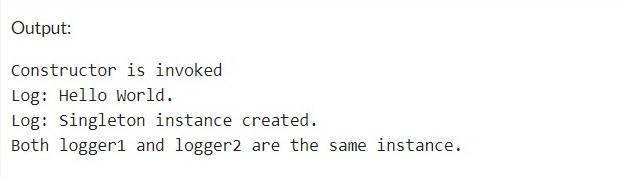
if (logger1 == logger2) {

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 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 DocumentFactory {

public abstract Document createDocument();

}

class WordDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new WordDoc();

}

}

class PdfDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new PdfDoc();

}

}

class ExcelDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new ExcelDoc();

}

}

class FactoryMethodTest {

public static void main(String[] args) {

DocumentFactory wordFactory = new WordDocumentFactory();

Document wordDoc = wordFactory.createDocument();

wordDoc.open();

wordDoc.close();

DocumentFactory pdfFactory = new PdfDocumentFactory();

Document pdfDoc = pdfFactory.createDocument();

pdfDoc.open();

pdfDoc.close();

DocumentFactory excelFactory = new ExcelDocumentFactory();

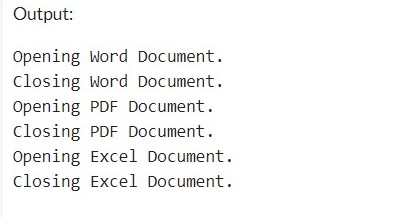
Document excelDoc = excelFactory.createDocument();

excelDoc.open();

excelDoc.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) {

// default configuration

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

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

// custom configuration

Computer customComputer = new Computer.Builder()

.setCpu("Intel Core i7")

.setRam(32)

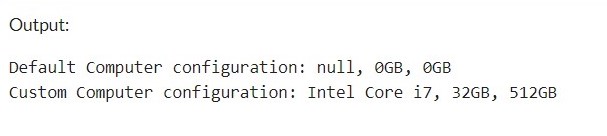
.setStorage(512)

.build();

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

}

}



**Exercise 4: Implementing the Adapter Pattern**

CODE:

interface PaymentProcessor { //Target Interface

void processPayment(double amount);

}

//Adaptee Classes

class PayPal {

public void makePayment(double amount) {

System.out.println("Making payment of rupees " + amount + " using PayPal.");

}

}

class Stripe {

public void collectPayment(double amount) {

System.out.println("Collecting payment of rupees " + amount + " using Stripe.");

}

}

//Adapter Class

class PayPalAdapter implements PaymentProcessor {

private PayPal payPal;

public PayPalAdapter() {

payPal = new PayPal();

}

public void processPayment(double amount) {

payPal.makePayment(amount);

}

}

class StripeAdapter implements PaymentProcessor {

private Stripe stripe;

public StripeAdapter() {

stripe = new Stripe();

}

public void processPayment(double amount) {

stripe.collectPayment(amount);

}

}

//Adapter Implementation

class AdapterPatternTest {

public static void main(String[] args) {

// payment using PayPal

PaymentProcessor payPalProcessor = new PayPalAdapter();

payPalProcessor.processPayment(2000.0);

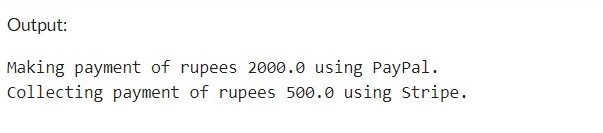
// payment using Stripe

PaymentProcessor stripeProcessor = new StripeAdapter();

stripeProcessor.processPayment(500.0);

}

}



**Exercise 5: Implementing the Decorator Pattern**

CODE:

//Component Interface

interface Notifier {

void send(String message);

}

//Concrete Component

class EmailNotifier implements Notifier {

public void send(String message) {

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

}

}

// Decorator Classes

abstract class NotifierDecorator implements Notifier {

protected Notifier notifier;

public NotifierDecorator(Notifier notifier) {

this.notifier = notifier;

}

public void send(String message) {

notifier.send(message);

}

}

//SMSNotifierDecorator

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("You got SMS notification: " + message);

}

}

//SlackNotifierDecorator

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("You Slack notification: " + message);

}

}

//Decorator Implementation

class DecoratorPatternTest {

public static void main(String[] args) {

// Send email notification

Notifier emailNotifier = new EmailNotifier();

emailNotifier.send("Hello!");

// Send email and SMS notifications

Notifier emailAndSMSNotifier = new SMSNotifierDecorator(emailNotifier);

emailAndSMSNotifier.send("This is a test notification.");

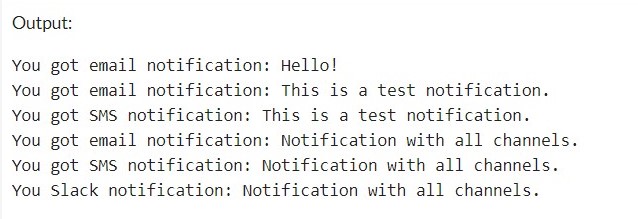
// Send email, SMS, and Slack notifications

Notifier allNotifier = new SlackNotifierDecorator(emailAndSMSNotifier);

allNotifier.send("Notification with all channels.");

}

}



**Exercise 6: Implementing the Proxy Pattern**

**CODE:**

//Subject Interface

interface Image {

void display();

}

//Real Subject Class

class RealImage implements Image {

private String filename;

public RealImage(String filename) {

loadImageFromServer();

this.filename = filename;

}

private void loadImageFromServer() {

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

}

public void display() {

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

}

}

//Proxy Class

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();

}

}

//Proxy Implementation

public class ProxyPatternTest {

public static void main(String[] args) {

Image image1 = new ProxyImage("img1.jpg");

Image image2 = new ProxyImage("img2.jpg");

// Image will be loaded from the server

image1.display();

System.out.println();

// Image will be loaded from the server again

image1.display();

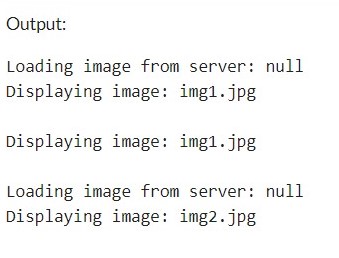
System.out.println();

// Image will be loaded from the server

image2.display();

}

}



**Exercise 7: Implementing the Observer Pattern**

CODE:

import java.util.ArrayList;

import java.util.List;

//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();

}

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);

}

}

}

//Observer Interface

interface Observer {

void update(double stockPrice);

}

//Concrete Observers

class MobileApp implements Observer {

public void update(double stockPrice) {

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

}

}

class WebApp implements Observer {

public void update(double stockPrice) {

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

}

}

//Observer Implementation

class ObserverPatternTest {

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);

// Set stock price and notify observers

stockMarket.setStockPrice(100.50);

stockMarket.setStockPrice(102.75);

// Deregister WebApp

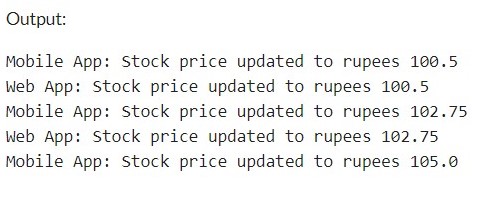
stockMarket.deregisterObserver(webApp);

// Set stock price and notify remaining observers

stockMarket.setStockPrice(105.00);

}

}



**Exercise 8: Implementing the Strategy Pattern**

**CODE:**

//Strategy Interface

interface PaymentStrategy {

void pay(double amount);

}

//Concrete Strategies

class CreditCardPayment implements PaymentStrategy {

public void pay(double amount) {

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

}

}

class PayPalPayment implements PaymentStrategy {

public void pay(double amount) {

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

}

}

//Context Class

class PaymentContext {

private PaymentStrategy paymentStrategy;

public void setPaymentStrategy(PaymentStrategy paymentStrategy) {

this.paymentStrategy = paymentStrategy;

}

public void makePayment(double amount) {

paymentStrategy.pay(amount);

}

}

//Strategy Implementation

class StrategyPatternTest {

public static void main(String[] args) {

PaymentContext paymentContext = new PaymentContext();

// Use Credit Card payment strategy

paymentContext.setPaymentStrategy(new CreditCardPayment());

paymentContext.makePayment(100.0);

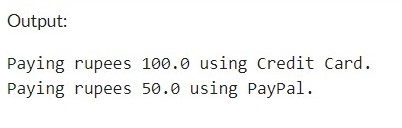
// Use PayPal payment strategy

paymentContext.setPaymentStrategy(new PayPalPayment());

paymentContext.makePayment(50.0);

}

}



**Exercise 9: Implementing the Command Pattern**

CODE:

//Command Interface

interface Command {

void execute();

}

//Concrete Commands

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();

}

}

//Invoker Class

class RemoteControl {

private Command command;

public void setCommand(Command command) {

this.command = command;

}

public void pressButton() {

command.execute();

}

}

//Receiver Class

class Light {

public void turnOn() {

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

}

public void turnOff() {

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

}

}

//Command Implementation

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();

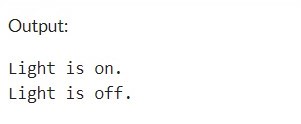
// Turn off the light

remoteControl.setCommand(lightOffCommand);

remoteControl.pressButton();

}

}



**Exercise 10: Implementing the MVC Pattern**

CODE:

//Model Class

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;

}

}

//View Class

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);

}

}

//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(int grade) {

model.setGrade(grade);

}

public int getStudentGrade() {

return model.getGrade();

}

public void updateView() {

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

}

}

//MVC Implementation

class MVCPatternTest {

public static void main(String[] args) {

// Create a student model

Student model = new Student();

// Create a view

StudentView view = new StudentView();

// Create a controller

StudentController controller = new StudentController(model, view);

// Set student details

controller.setStudentName("John");

controller.setStudentId(1);

controller.setStudentGrade(85);

// Update the view

controller.updateView();

// Update student name

controller.setStudentName("Jane");

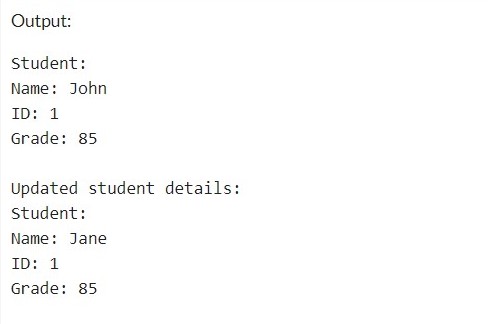
System.out.println("\nUpdated student details:");

// Update the view

controller.updateView();

}

}



**Exercise 11: Implementing Dependency Injection**

CODE:

//Repository Interface

interface CustomerRepository {

String findCustomerById(int id);

}

// Concrete Repository

class CustomerRepositoryImpl implements CustomerRepository {

@Override

public String findCustomerById(int id) {

// Simulating a database lookup

return "Customer with ID: " + id;

}

}

//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);

}

}

//Dependency Injection Implementation

class DependencyInjectionTest {

public static void main(String[] args) {

// Create an instance of the repository

CustomerRepository customerRepository = new CustomerRepositoryImpl();

// Inject the repository into the service

CustomerService customerService = new CustomerService(customerRepository);

// Use the service to find a customer

String customer = customerService.getCustomer(1);

System.out.println(customer);

}

}

