**HANDSON EXERCISES - WEEK 1**

**Skill : Design Patterns and Principles**

**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**

**Logger.java**

package com.example.singleton;

public class Logger {

private static Logger instance;

private Logger() {

System.out.println("Logger initialized.");

}

public static Logger getInstance() {

if (instance == null) {

instance = new Logger();

}

return instance;

}

public void log(String message) {

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

}

}

**Main.java**

package com.example.singleton;

public class Main {

public static void main(String[] args) {

Logger logger1 = Logger.*getInstance*();

Logger logger2 = Logger.*getInstance*();

logger1.log("First log message");

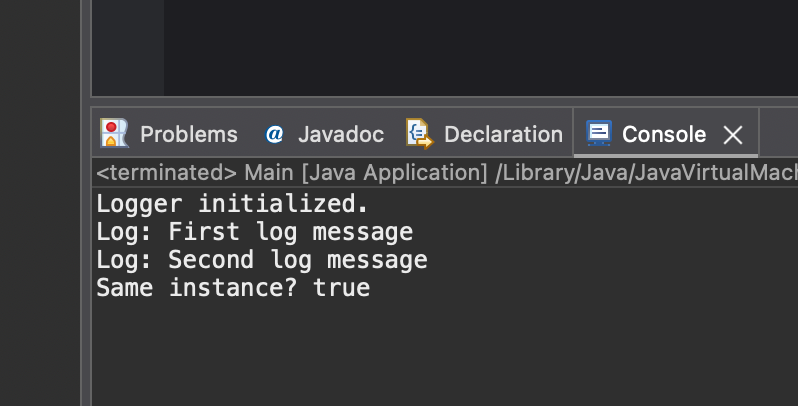
logger2.log("Second log message");

System.***out***.println("Same instance? " + (logger1 == logger2));

}

}

**OUTPUT :**



**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.

**CODE :**

**Document.java :**package com.example.Factory\_Method\_Pattern;

public interface Document {

void open();

}

**WordDocument.java :**

package com.example.Factory\_Method\_Pattern;

public class WordDocument implements Document {

public void open() {

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

}

}

**PdfDocument.java :**

package com.example.Factory\_Method\_Pattern;

public class PdfDocument implements Document {

public void open() {

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

}

}

**DocumentFactory.java :**

package com.example.Factory\_Method\_Pattern;

public abstract class DocumentFactory {

public abstract Document createDocument();

}

**WordFactory.java :**

package com.example.Factory\_Method\_Pattern;

public class WordFactory extends DocumentFactory {

public Document createDocument() {

return new WordDocument();

}

}

**PdfFactory.java :**

package com.example.Factory\_Method\_Pattern;

public class PdfFactory extends DocumentFactory {

public Document createDocument() {

return new PdfDocument();

}

}

**Main.java :**

package com.example.Factory\_Method\_Pattern;

public class Main {

public static void main(String[] args) {

DocumentFactory wordFactory = new WordFactory();

Document doc1 = wordFactory.createDocument();

doc1.open();

DocumentFactory pdfFactory = new PdfFactory();

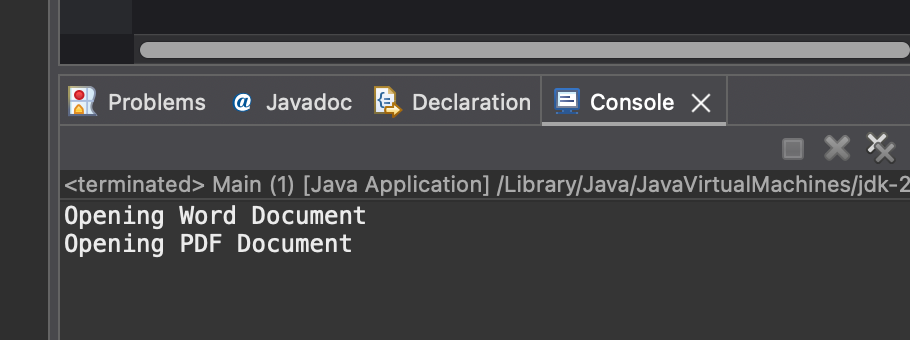
Document doc2 = pdfFactory.createDocument();

doc2.open();

}

}

**OUTPUT :**



**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**

**Computer.java :**

package com.example.BuilderPattern;

public class Computer {

private String cpu, ram, storage;

private Computer(Builder builder) {

this.cpu = builder.cpu;

this.ram = builder.ram;

this.storage = builder.storage;

}

public void showSpecs() {

System.out.println("CPU: " + cpu + ", RAM: " + ram + ", Storage: " + storage);

}

public static class Builder {

private String cpu, ram, 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);

}

}

}

**Main.java :**

package com.example.BuilderPattern;

public class Main {

public static void main(String[] args) {

Computer myPc = new Computer.Builder()

.setCpu("Intel i9")

.setRam("32GB")

.setStorage("1TB SSD")

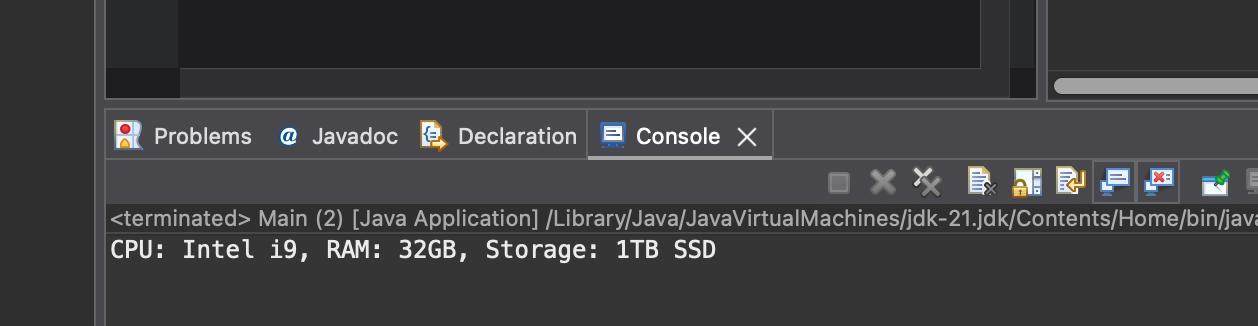
.build();

myPc.showSpecs();

}

}

**OUTPUT :**



**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.

**PaymentProcessor.java :**

package com.example.AdapterPattern;

public interface PaymentProcessor {

void processPayment(double amount);

}

**ThirdPartyGateway.java :**

package com.example.AdapterPattern;

public class ThirdPartyGateway {

public void makeTransaction(double amount) {

System.***out***.println("Paid using third party: ₹" + amount);

}

}

**GatewayAdapter.java :**

package com.example.AdapterPattern;

public class GatewayAdapter implements PaymentProcessor {

private ThirdPartyGateway gateway;

public GatewayAdapter(ThirdPartyGateway gateway) {

this.gateway = gateway;

}

public void processPayment(double amount) {

gateway.makeTransaction(amount);

}

}

**Main.java :**

package com.example.AdapterPattern;

public class Main {

public static void main(String[] args) {

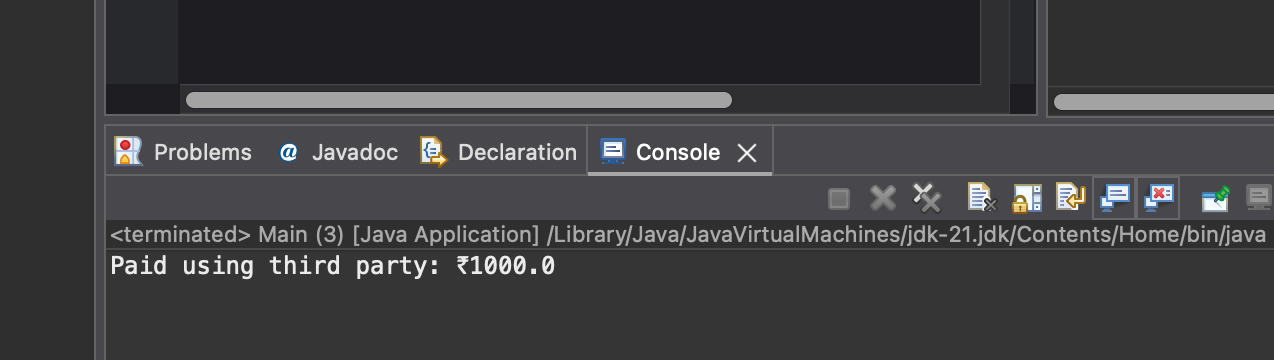
PaymentProcessor adapter = new GatewayAdapter(new ThirdPartyGateway());

adapter.processPayment(1000);

}

}

**OUTPUT :**



**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.

**CODE**

**Notifier.java :**

package com.example.DecoratorPattern;

public interface Notifier {

void send(String message);

}

**EmailNotifier.java :**

package com.example.DecoratorPattern;

public class EmailNotifier implements Notifier {

public void send(String message) {

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

}

}

**NotifierDecorator.java :**

package com.example.DecoratorPattern;

public abstract class NotifierDecorator implements Notifier {

protected Notifier notifier;

public NotifierDecorator(Notifier notifier) {

this.notifier = notifier;

}

public void send(String message) {

notifier.send(message);

}

}

**SMSNotifierDecorator.java :**

package com.example.DecoratorPattern;

public class SMSNotifierDecorator extends NotifierDecorator {

public SMSNotifierDecorator(Notifier notifier) {

super(notifier);

}

public void send(String message) {

super.send(message);

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

}

}

**Main.java :**

package com.example.DecoratorPattern;

public class Main {

public static void main(String[] args) {

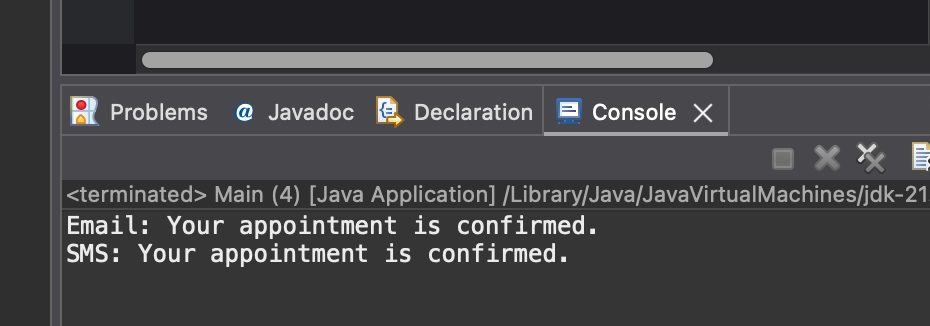
Notifier notifier = new SMSNotifierDecorator(new EmailNotifier());

notifier.send("Your appointment is confirmed.");

}

}

**OUTPUT :**



**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.

**CODE**

**Image.java :**

package com.example.ProxyPattern;

public interface Image {

void display();

}

**RealImage.java :**

package com.example.ProxyPattern;

public class RealImage implements Image {

private String fileName;

public RealImage(String fileName) {

this.fileName = fileName;

loadFromDisk();

}

private void loadFromDisk() {

System.***out***.println("Loading image: " + fileName);

}

public void display() {

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

}

}

**ProxyImage.java :**

package com.example.ProxyPattern;

public 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();

}

}

**Main.java :**

package com.example.ProxyPattern;

public class Main {

public static void main(String[] args) {

Image image = new ProxyImage("Remainder.jpeg");

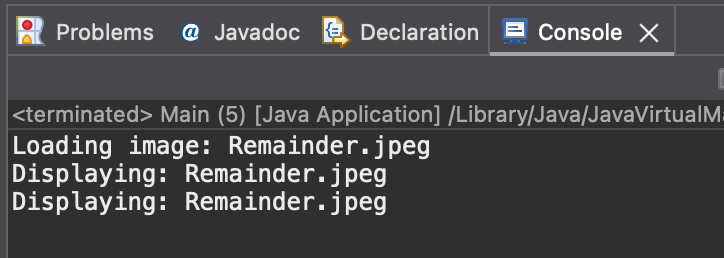
image.display();

image.display();

}

}

**OUTPUT :**



**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.

**CODE**

**Observer.java :**

package com.example.ObserverPattern;

public interface Observer {

void update(String stock);

}

**Stock.java :**

package com.example.ObserverPattern;

public interface Stock {

void register(Observer o);

void remove(Observer o);

void notifyObservers();

}

**StockMarket.java :**

package com.example.ObserverPattern;

import java.util.\*;

public class StockMarket implements Stock {

private List<Observer> observers = new ArrayList<>();

private String stock;

public void setStock(String stock) {

this.stock = stock;

notifyObservers();

}

public void register(Observer o) {

observers.add(o);

}

public void remove(Observer o) {

observers.remove(o);

}

public void notifyObservers() {

for (Observer o : observers) {

o.update(stock);

}

}

}

**MobileApp.java :**

package com.example.ObserverPattern;

public class MobileApp implements Observer {

public void update(String stock) {

System.***out***.println("Mobile App: Stock update → " + stock);

}

}

**WebApp.java :**

package com.example.ObserverPattern;

public class WebApp implements Observer {

public void update(String stock) {

System.***out***.println("Web App: Stock update → " + stock);

}

}

**Main.java :**

package com.example.ObserverPattern;

public class Main {

public static void main(String[] args) {

StockMarket market = new StockMarket();

Observer mobile = new MobileApp();

Observer web = new WebApp();

market.register(mobile);

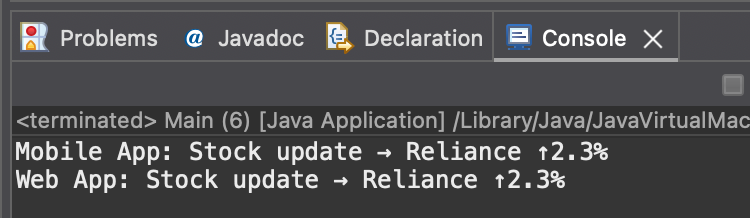
market.register(web);

market.setStock("Reliance ↑2.3%");

}

}

**OUTPUT :**



**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.

**CODE**

**PayementStrategy.java :**

package com.example.StrategyPattern;

public interface PaymentStrategy {

void pay(double amount);

}

**CreditCardPayment.java :**

package com.example.StrategyPattern;

public class CreditCardPayment implements PaymentStrategy {

public void pay(double amount) {

System.***out***.println("Paid ₹" + amount + " via Credit Card");

}

}

**PayPalPayment.java :**

package com.example.StrategyPattern;

public class PayPalPayment implements PaymentStrategy {

public void pay(double amount) {

System.***out***.println("Paid ₹" + amount + " via PayPal");

}

}

**PaymentContext.java :**

package com.example.StrategyPattern;

public class PaymentContext {

private PaymentStrategy strategy;

public void setStrategy(PaymentStrategy strategy) {

this.strategy = strategy;

}

public void pay(double amount) {

strategy.pay(amount);

}

}

**Main.java :**

package com.example.StrategyPattern;

public class Main {

public static void main(String[] args) {

PaymentContext context = new PaymentContext();

context.setStrategy(new CreditCardPayment());

context.pay(5000);

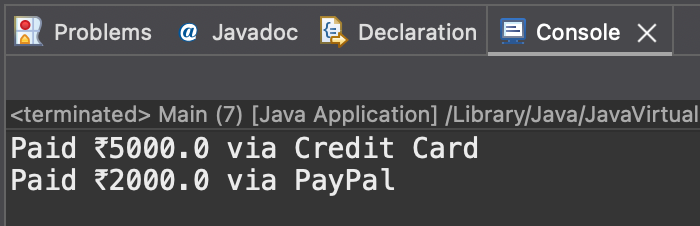
context.setStrategy(new PayPalPayment());

context.pay(2000);

}

}

**OUTPUT :**



**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.

**CODE**

**Command.java :**

package com.example.CommandPattern;

public interface Command {

void execute();

}

**Light.java :**

package com.example.CommandPattern;

public class Light {

public void on() {

System.***out***.println("Light turned ON");

}

public void off() {

System.***out***.println("Light turned OFF");

}

}

**LightOnCommand.java :**

package com.example.CommandPattern;

public class LightOnCommand implements Command {

private Light light;

public LightOnCommand(Light light) {

this.light = light;

}

public void execute() {

light.on();

}

}

**LightOffCommand.java :**

package com.example.CommandPattern;

public class LightOffCommand implements Command {

private Light light;

public LightOffCommand(Light light) {

this.light = light;

}

public void execute() {

light.off();

}

}

**RemoteControl.java :**

package com.example.CommandPattern;

public class RemoteControl {

private Command command;

public void setCommand(Command command) {

this.command = command;

}

public void pressButton() {

command.execute();

}

}

**Main.java :**

package com.example.CommandPattern;

public class Main {

public static void main(String[] args) {

Light livingRoomLight = new Light();

Command lightOn = new LightOnCommand(livingRoomLight);

Command lightOff = new LightOffCommand(livingRoomLight);

RemoteControl remote = new RemoteControl();

remote.setCommand(lightOn);

remote.pressButton();

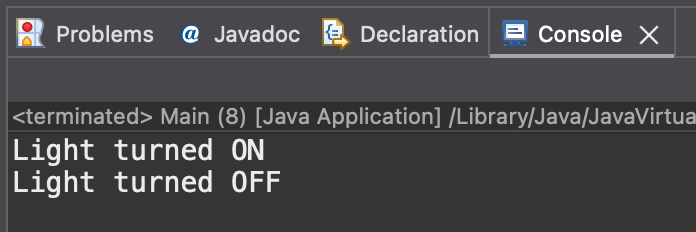
remote.setCommand(lightOff);

remote.pressButton();

}

}

**OUTPUT :**



**Exercise 10: Implementing the MVC Pattern**

**Scenario:**

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

**CODE**

**Student.java :**

package com.example.MVCPattern;

public class Student {

private String name;

private String id;

public Student(String name, String id) {

this.name = name;

this.id = id;

}

public String getName() { return name; }

public String getId() { return id; }

public void setName(String name) { this.name = name; }

}

**StudentView.java :**

package com.example.MVCPattern;

public class StudentView {

public void displayStudentDetails(String name, String id) {

System.***out***.println("Student Name: " + name + ", ID: " + id);

}

}

**StudentController.java :**

package com.example.MVCPattern;

public class StudentController {

private Student model;

private StudentView view;

public StudentController(Student model, StudentView view) {

this.model = model;

this.view = view;

}

public void updateView() {

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

}

public void changeName(String name) {

model.setName(name);

}

}

**Main.java :**

package com.example.MVCPattern;

public class Main {

public static void main(String[] args) {

Student student = new Student("Vasanthi", "101");

StudentView view = new StudentView();

StudentController controller = new StudentController(student, view);

controller.updateView();

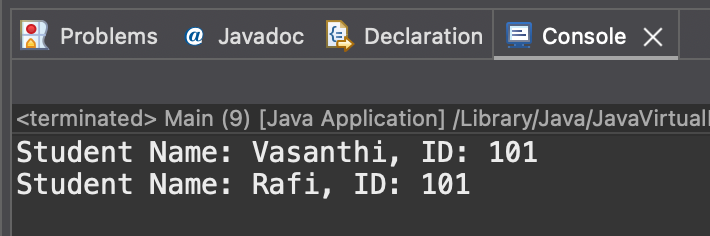
controller.changeName("Rafi");

controller.updateView();

}

}

**OUTPUT :**



**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.

**CODE**

**CustomerRepository.java :**

package com.example.DependencyInjection;

public interface CustomerRepository {

String findCustomerById(String id);

}

**CustomerRepositoryImpl.java :**

package com.example.DependencyInjection;

public class CustomerRepositoryImpl implements CustomerRepository {

public String findCustomerById(String id) {

return "Customer: Rafi | ID: " + id;

}

}

**CustomerService.java :**

package com.example.DependencyInjection;

public class CustomerService {

private CustomerRepository repo;

public CustomerService(CustomerRepository repo) {

this.repo = repo;

}

public void showCustomer(String id) {

System.***out***.println(repo.findCustomerById(id));

}

}

**Main.java :**

package com.example.DependencyInjection;

public class Main {

public static void main(String[] args) {

CustomerRepository repo = new CustomerRepositoryImpl();

CustomerService service = new CustomerService(repo);

service.showCustomer("007");

}

}

**OUTPUT :**

