**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:**

**public class SingletonPatternExample {**

**static 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);**

**}**

**}**

**public static void main(String[] args) {**

**Logger logger1 = Logger.getInstance();**

**logger1.log("This is the first message.");**

**Logger logger2 = Logger.getInstance();**

**logger2.log("This is the second message.");**

**if (logger1 == logger2) {**

**System.out.println("Both loggers are the same instance.");**

**} else {**

**System.out.println("Different instances.");**

**}**

**}**

**}**

**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 :**

**public class FactoryMethodPatternExample {**

**interface Document {**

**void open();**

**}**

**static class WordDocument implements Document {**

**public void open() {**

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

**}**

**}**

**static class PdfDocument implements Document {**

**public void open() {**

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

**}**

**}**

**static class ExcelDocument implements Document {**

**public void open() {**

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

**}**

**}**

**abstract static class DocumentFactory {**

**abstract Document createDocument();**

**}**

**static class WordDocumentFactory extends DocumentFactory {**

**public Document createDocument() {**

**return new WordDocument();**

**}**

**}**

**static class PdfDocumentFactory extends DocumentFactory {**

**public Document createDocument() {**

**return new PdfDocument();**

**}**

**}**

**static class ExcelDocumentFactory extends DocumentFactory {**

**public Document createDocument() {**

**return new ExcelDocument();**

**}**

**}**

**public static void main(String[] args) {**

**DocumentFactory wordFactory = new WordDocumentFactory();**

**Document word = wordFactory.createDocument();**

**word.open();**

**DocumentFactory pdfFactory = new PdfDocumentFactory();**

**Document pdf = pdfFactory.createDocument();**

**pdf.open();**

**DocumentFactory excelFactory = new ExcelDocumentFactory();**

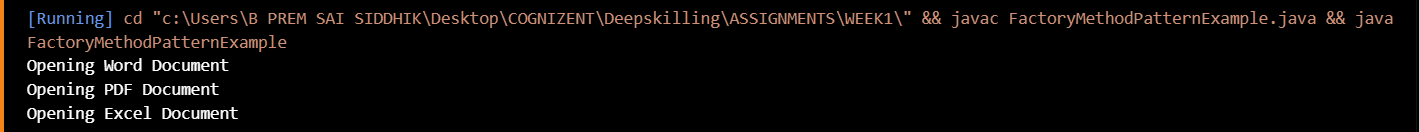
**Document excel = excelFactory.createDocument();**

**excel.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:**

**public class Computer {**

**private String cpu;**

**private String ram;**

**private String storage;**

**private String gpu;**

**private Computer(Builder builder) {**

**this.cpu = builder.cpu;**

**this.ram = builder.ram;**

**this.storage = builder.storage;**

**this.gpu = builder.gpu;**

**}**

**public void showConfig() {**

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

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

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

**System.out.println("GPU: " + gpu);**

**System.out.println("-------------------");**

**}**

**public static class Builder {**

**private String cpu;**

**private String ram;**

**private String storage;**

**private String gpu;**

**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 Builder setGpu(String gpu) {**

**this.gpu = gpu;**

**return this;**

**}**

**public Computer build() {**

**return new Computer(this);**

**}**

**}**

**public static void main(String[] args) {**

**Computer pc1 = new Computer.Builder()**

**.setCpu("Intel i7")**

**.setRam("16GB")**

**.setStorage("512GB SSD")**

**.setGpu("NVIDIA GTX 1660")**

**.build();**

**Computer pc2 = new Computer.Builder()**

**.setCpu("AMD Ryzen 5")**

**.setRam("8GB")**

**.setStorage("1TB HDD")**

**.build();**

**Computer pc3 = new Computer.Builder()**

**.setCpu("Intel i3")**

**.build();**

**pc1.showConfig();**

**pc2.showConfig();**

**pc3.showConfig();**

**}**

**}**

**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.

**CODE:**

**public class AdapterPatternExample {**

**interface PaymentProcessor {**

**void processPayment(double amount);**

**}**

**class PayPalGateway {**

**void makePayment(double amountInDollars) {**

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

**}**

**}**

**class StripeGateway {**

**void payAmount(double amount) {**

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

**}**

**}**

**class PayPalAdapter implements PaymentProcessor {**

**private PayPalGateway payPal;**

**public PayPalAdapter(PayPalGateway payPal) {**

**this.payPal = payPal;**

**}**

**public void processPayment(double amount) {**

**double amountInDollars = amount / 83.0;**

**payPal.makePayment(amountInDollars);**

**}**

**}**

**class StripeAdapter implements PaymentProcessor {**

**private StripeGateway stripe;**

**public StripeAdapter(StripeGateway stripe) {**

**this.stripe = stripe;**

**}**

**public void processPayment(double amount) {**

**stripe.payAmount(amount);**

**}**

**}**

**public static void main(String[] args) {**

**AdapterPatternExample app = new AdapterPatternExample();**

**PaymentProcessor processor1 = app.new PayPalAdapter(app.new PayPalGateway());**

**PaymentProcessor processor2 = app.new StripeAdapter(app.new StripeGateway());**

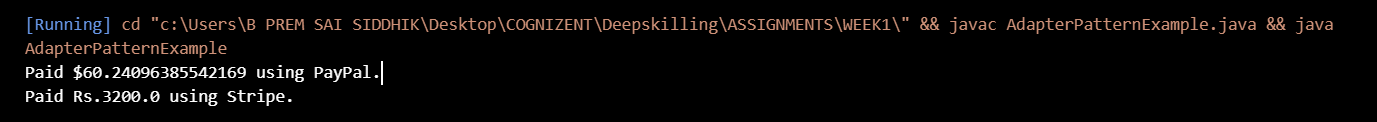
**processor1.processPayment(5000);**

**processor2.processPayment(3200);**

**}**

**}**

**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:**

**public class DecoratorPatternExample {**

**interface Notifier {**

**void send(String message);**

**}**

**class EmailNotifier implements Notifier {**

**public void send(String message) {**

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

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

**}**

**}**

**class SlackNotifierDecorator extends NotifierDecorator {**

**public SlackNotifierDecorator(Notifier notifier) {**

**super(notifier);**

**}**

**public void send(String message) {**

**super.send(message);**

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

**}**

**}**

**public static void main(String[] args) {**

**DecoratorPatternExample app = new DecoratorPatternExample();**

**Notifier baseNotifier = app.new EmailNotifier();**

**Notifier smsNotifier = app.new SMSNotifierDecorator(baseNotifier);**

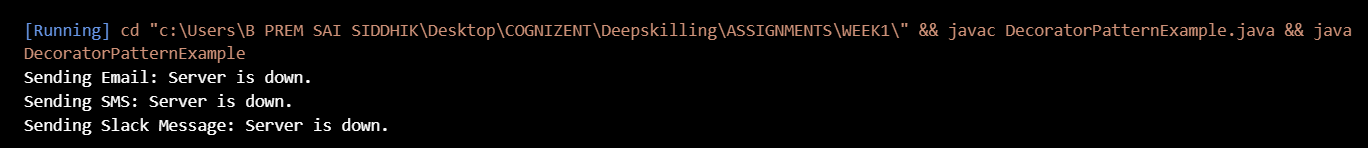
**Notifier fullNotifier = app.new SlackNotifierDecorator(smsNotifier);**

**fullNotifier.send("Server is down.");**

**}**

**}**

**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:**

**public class ProxyPatternExample {**

**interface Image {**

**void display();**

**}**

**class RealImage implements Image {**

**private String filename;**

**public RealImage(String filename) {**

**this.filename = filename;**

**loadFromRemoteServer();**

**}**

**private void loadFromRemoteServer() {**

**System.out.println("Loading " + filename + " from remote server...");**

**}**

**public void display() {**

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

**} else {**

**System.out.println("Loaded from cache: " + filename);**

**}**

**realImage.display();**

**}**

**}**

**public static void main(String[] args) {**

**ProxyPatternExample app = new ProxyPatternExample();**

**Image image1 = app.new ProxyImage("nature.jpg");**

**Image image2 = app.new ProxyImage("city.jpg");**

**image1.display();**

**image1.display();**

**image2.display();**

**image2.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:**

**public class ObserverPatternExample {**

**interface Observer {**

**void update(String stockName, double price);**

**}**

**interface Stock {**

**void register(Observer observer);**

**void deregister(Observer observer);**

**void notifyObservers();**

**}**

**class StockMarket implements Stock {**

**private String stockName;**

**private double price;**

**private java.util.List<Observer> observers = new java.util.ArrayList<>();**

**public StockMarket(String stockName, double price) {**

**this.stockName = stockName;**

**this.price = price;**

**}**

**public void register(Observer observer) {**

**observers.add(observer);**

**}**

**public void deregister(Observer observer) {**

**observers.remove(observer);**

**}**

**public void notifyObservers() {**

**for (Observer o : observers) {**

**o.update(stockName, price);**

**}**

**}**

**public void setPrice(double newPrice) {**

**this.price = newPrice;**

**notifyObservers();**

**}**

**}**

**class MobileApp implements Observer {**

**private String user;**

**public MobileApp(String user) {**

**this.user = user;**

**}**

**public void update(String stockName, double price) {**

**System.out.println("MobileApp - " + user + ": " + stockName + " updated to Rs." + price);**

**}**

**}**

**class WebApp implements Observer {**

**private String user;**

**public WebApp(String user) {**

**this.user = user;**

**}**

**public void update(String stockName, double price) {**

**System.out.println("WebApp - " + user + ": " + stockName + " updated to Rs." + price);**

**}**

**}**

**public static void main(String[] args) {**

**ObserverPatternExample app = new ObserverPatternExample();**

**StockMarket reliance = app.new StockMarket("RELIANCE", 2800.0);**

**Observer user1 = app.new MobileApp("Ravi");**

**Observer user2 = app.new WebApp("Anita");**

**reliance.register(user1);**

**reliance.register(user2);**

**reliance.setPrice(2820.5);**

**reliance.setPrice(2795.0);**

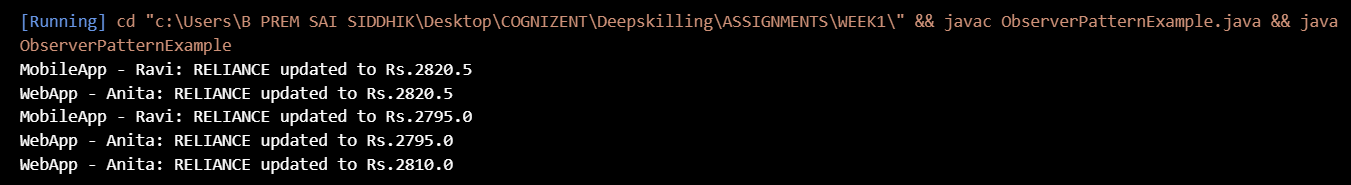
**reliance.deregister(user1);**

**reliance.setPrice(2810.0);**

**}**

**}**

**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:**

**public class StrategyPatternExample {**

**interface PaymentStrategy {**

**void pay(double amount);**

**}**

**class CreditCardPayment implements PaymentStrategy {**

**private String cardNumber;**

**public CreditCardPayment(String cardNumber) {**

**this.cardNumber = cardNumber;**

**}**

**public void pay(double amount) {**

**System.out.println("Paid Rs." + amount + " using Credit Card: " + cardNumber);**

**}**

**}**

**class PayPalPayment implements PaymentStrategy {**

**private String email;**

**public PayPalPayment(String email) {**

**this.email = email;**

**}**

**public void pay(double amount) {**

**System.out.println("Paid Rs." + amount + " using PayPal account: " + email);**

**}**

**}**

**class PaymentContext {**

**private PaymentStrategy strategy;**

**public void setPaymentStrategy(PaymentStrategy strategy) {**

**this.strategy = strategy;**

**}**

**public void processPayment(double amount) {**

**if (strategy != null) {**

**strategy.pay(amount);**

**} else {**

**System.out.println("Payment method not selected.");**

**}**

**}**

**}**

**public static void main(String[] args) {**

**StrategyPatternExample app = new StrategyPatternExample();**

**PaymentContext context = app.new PaymentContext();**

**context.setPaymentStrategy(app.new CreditCardPayment("1234-5678-9012-3456"));**

**context.processPayment(1500);**

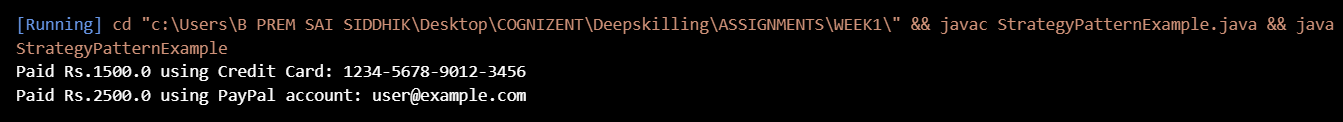
**context.setPaymentStrategy(app.new PayPalPayment("user@example.com"));**

**context.processPayment(2500);**

**}**

**}**

**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:**

**public class CommandPatternExample {**

**interface Command {**

**void execute();**

**}**

**class Light {**

**public void turnOn() {**

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

**}**

**public void turnOff() {**

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

**}**

**}**

**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() {**

**if (command != null) {**

**command.execute();**

**} else {**

**System.out.println("No command set.");**

**}**

**}**

**}**

**public static void main(String[] args) {**

**CommandPatternExample app = new CommandPatternExample();**

**Light livingRoomLight = app.new Light();**

**Command lightOn = app.new LightOnCommand(livingRoomLight);**

**Command lightOff = app.new LightOffCommand(livingRoomLight);**

**RemoteControl remote = app.new RemoteControl();**

**remote.setCommand(lightOn);**

**remote.pressButton();**

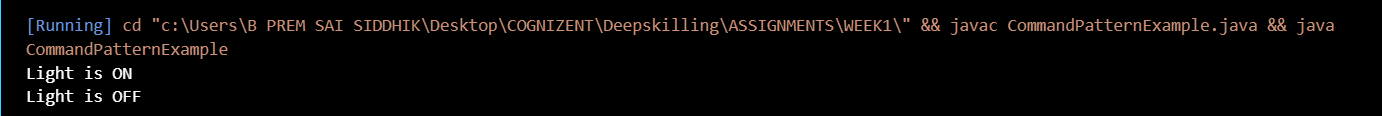
**remote.setCommand(lightOff);**

**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:**

**public class MVCPatternExample {**

**class Student {**

**private String name;**

**private String id;**

**private String grade;**

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

**this.name = name;**

**this.id = id;**

**this.grade = grade;**

**}**

**public String getName() {**

**return name;**

**}**

**public void setName(String name) {**

**this.name = name;**

**}**

**public String getId() {**

**return id;**

**}**

**public void setId(String id) {**

**this.id = id;**

**}**

**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);**

**System.out.println("--------------------");**

**}**

**}**

**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 void setStudentId(String id) {**

**student.setId(id);**

**}**

**public void setStudentGrade(String grade) {**

**student.setGrade(grade);**

**}**

**public void updateView() {**

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

**}**

**}**

**public static void main(String[] args) {**

**MVCPatternExample app = new MVCPatternExample();**

**Student student = app.new Student("Rahul", "S101", "A");**

**StudentView view = app.new StudentView();**

**StudentController controller = app.new StudentController(student, view);**

**controller.updateView();**

**controller.setStudentName("Rohit");**

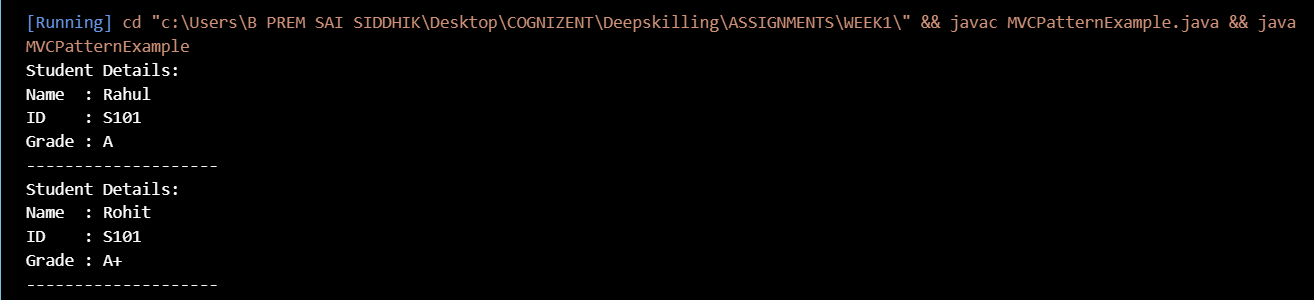
**controller.setStudentGrade("A+");**

**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:**

**public class DependencyInjectionExample {**

**class Customer {**

**private String id;**

**private String name;**

**public Customer(String id, String name) {**

**this.id = id;**

**this.name = name;**

**}**

**public String getId() {**

**return id;**

**}**

**public String getName() {**

**return name;**

**}**

**}**

**interface CustomerRepository {**

**Customer findCustomerById(String id);**

**}**

**class CustomerRepositoryImpl implements CustomerRepository {**

**public Customer findCustomerById(String id) {**

**return new Customer(id, "Arun Kumar");**

**}**

**}**

**class CustomerService {**

**private CustomerRepository repository;**

**public CustomerService(CustomerRepository repository) {**

**this.repository = repository;**

**}**

**public void showCustomer(String id) {**

**Customer customer = repository.findCustomerById(id);**

**System.out.println("Customer ID : " + customer.getId());**

**System.out.println("Customer Name : " + customer.getName());**

**}**

**}**

**public static void main(String[] args) {**

**DependencyInjectionExample app = new DependencyInjectionExample();**

**CustomerRepository repo = app.new CustomerRepositoryImpl();**

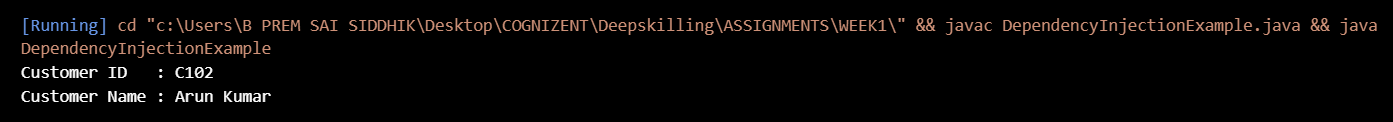
**CustomerService service = app.new CustomerService(repo);**

**service.showCustomer("C102");**

**}**

**}**

**OUTPUT :**

****