**WEEK 1: DESIGN PATTERN AND PRINCIPLES**

**Exercise 1: Implementing the Singleton Pattern**

**CODE:**

public class Main {

static class Logger {

private static volatile Logger instance;

private Logger() {

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

}

public static Logger getInstance() {

if (instance == null) {

synchronized (Logger.class) {

if (instance == null) {

instance = new Logger();

}

}

}

return instance;

}

public void log(String message) {

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

}

}

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 loggers are the same instance (Singleton confirmed).");

} else {

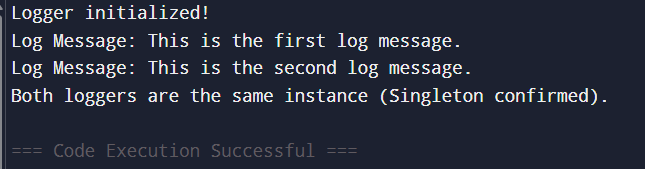
System.out.println("Different instances exist (Singleton failed).");

}

}

}

**OUTPUT:**



**Exercise 2: Implementing the Factory Method Pattern**

**CODE:**

public class Main {

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");

}

}

static abstract class DocumentFactory {

public 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 wordDoc = wordFactory.createDocument();

wordDoc.open();

DocumentFactory pdfFactory = new PdfDocumentFactory();

Document pdfDoc = pdfFactory.createDocument();

pdfDoc.open();

DocumentFactory excelFactory = new ExcelDocumentFactory();

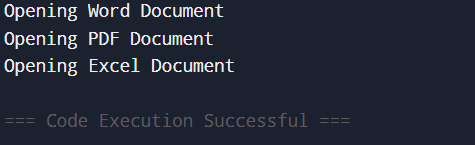
Document excelDoc = excelFactory.createDocument();

excelDoc.open();

}

}

**OUTPUT:**



**Exercise 3: Implementing the Builder Pattern**

**CODE:**

public class Main {

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;

}

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 void showSpecs() {

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

}

}

public static void main(String[] args) {

Computer pc = new Computer.Builder()

.setCPU("Intel i5")

.setRAM("8GB")

.setStorage("512GB SSD")

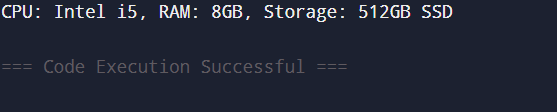
.build();

pc.showSpecs();

}

}

**OUTPUT:**



**Exercise 4: Implementing the Adapter Pattern**

**CODE:**

public class Main {

interface PaymentProcessor {

void processPayment(double amount);

}

static class PayPal {

public void makePayment(double amount) {

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

}

}

static class Stripe {

public void pay(double amount) {

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

}

}

static class PayPalAdapter implements PaymentProcessor {

private PayPal payPal = new PayPal();

public void processPayment(double amount) {

payPal.makePayment(amount);

}

}

static class StripeAdapter implements PaymentProcessor {

private Stripe stripe = new Stripe();

public void processPayment(double amount) {

stripe.pay(amount);

}

}

public static void main(String[] args) {

PaymentProcessor paypal = new PayPalAdapter();

PaymentProcessor stripe = new StripeAdapter();

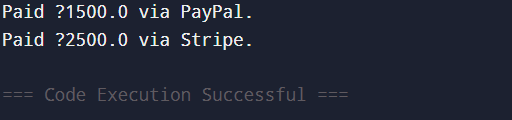
paypal.processPayment(1500);

stripe.processPayment(2500);

}

}

**OUTPUT:**



**Exercise 5: Implementing the Decorator Pattern**

**CODE:**

public class Main {

interface Notifier {

void send(String message);

}

static class EmailNotifier implements Notifier {

public void send(String message) {

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

}

}

static abstract class NotifierDecorator implements Notifier {

protected Notifier wrappee;

public NotifierDecorator(Notifier wrappee) {

this.wrappee = wrappee;

}

}

static class SMSNotifier extends NotifierDecorator {

public SMSNotifier(Notifier wrappee) {

super(wrappee);

}

public void send(String message) {

wrappee.send(message);

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

}

}

static class SlackNotifier extends NotifierDecorator {

public SlackNotifier(Notifier wrappee) {

super(wrappee);

}

public void send(String message) {

wrappee.send(message);

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

}

}

public static void main(String[] args) {

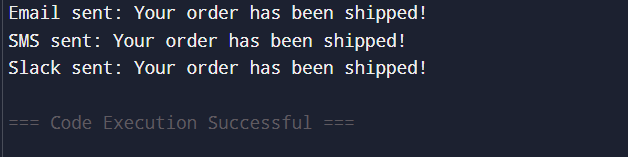
Notifier notifier = new SlackNotifier(new SMSNotifier(new EmailNotifier()));

notifier.send("Your order has been shipped!");

}

}

**OUTPUT:**



**Exercise 6: Implementing the Proxy Pattern**

**CODE:**

public class Main {

interface Image {

void display();

}

static class RealImage implements Image {

private String filename;

public RealImage(String filename) {

this.filename = filename;

loadFromDisk();

}

private void loadFromDisk() {

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

}

public void display() {

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

}

}

static 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();

}

}

public static void main(String[] args) {

Image image = new ProxyImage("flower.jpg");

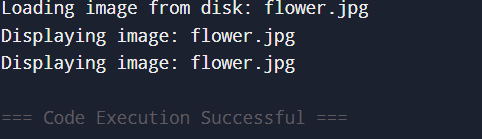
image.display(); // loads + displays

image.display(); // just displays (cached)

}

}

**OUTPUT:**



**Exercise 7: Implementing the Observer Pattern**

**CODE:**

import java.util.\*;

public class Main {

interface Observer {

void update(float price);

}

interface Stock {

void register(Observer o);

void remove(Observer o);

void notifyObservers();

}

static class StockMarket implements Stock {

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

private float price;

public void setPrice(float price) {

this.price = price;

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(price);

}

}

}

static class MobileApp implements Observer {

public void update(float price) {

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

}

}

static class WebApp implements Observer {

public void update(float price) {

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

}

}

public static void main(String[] args) {

StockMarket stock = new StockMarket();

Observer mobile = new MobileApp();

Observer web = new WebApp();

stock.register(mobile);

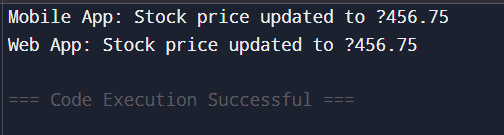
stock.register(web);

stock.setPrice(456.75f);

}

}

**OUTPUT:**



**Exercise 8: Implementing the Strategy Pattern**

**CODE:**

public class Main {

interface PaymentStrategy {

void pay(double amount);

}

static class CreditCardPayment implements PaymentStrategy {

public void pay(double amount) {

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

}

}

static class PayPalPayment implements PaymentStrategy {

public void pay(double amount) {

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

}

}

static class PaymentContext {

private PaymentStrategy strategy;

public void setStrategy(PaymentStrategy strategy) {

this.strategy = strategy;

}

public void execute(double amount) {

strategy.pay(amount);

}

}

public static void main(String[] args) {

PaymentContext context = new PaymentContext();

context.setStrategy(new CreditCardPayment());

context.execute(2500);

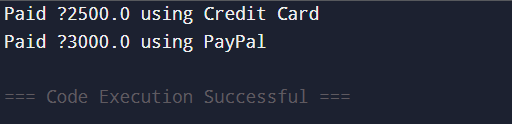
context.setStrategy(new PayPalPayment());

context.execute(3000);

}

}

**OUTPUT:**



**Exercise 9: Implementing the Command Pattern**

**CODE:**

public class Main {

interface Command {

void execute();

}

static class Light {

public void on() {

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

}

public void off() {

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

}

}

static class LightOnCommand implements Command {

private Light light;

public LightOnCommand(Light light) {

this.light = light;

}

public void execute() {

light.on();

}

}

static class LightOffCommand implements Command {

private Light light;

public LightOffCommand(Light light) {

this.light = light;

}

public void execute() {

light.off();

}

}

static class RemoteControl {

private Command command;

public void setCommand(Command command) {

this.command = command;

}

public void pressButton() {

command.execute();

}

}

public static void main(String[] args) {

Light light = new Light();

RemoteControl remote = new RemoteControl();

remote.setCommand(new LightOnCommand(light));

remote.pressButton();

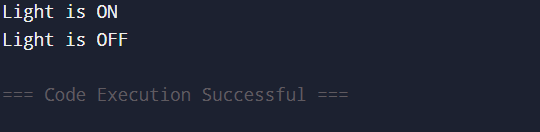
remote.setCommand(new LightOffCommand(light));

remote.pressButton();

}

}

**OUTPUT:**



**Exercise 10: Implementing the MVC Pattern**

**CODE:**

public class Main {

static 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; }

}

static class StudentView {

public void displayStudentDetails(Student student) {

System.out.println("Student: " + student.getName() + ", ID: " + student.getId());

}

}

static class StudentController {

private Student model;

private StudentView view;

public StudentController(Student model, StudentView view) {

this.model = model;

this.view = view;

}

public void updateName(String name) {

model.setName(name);

}

public void updateView() {

view.displayStudentDetails(model);

}

}

public static void main(String[] args) {

Student model = new Student("Swathi", "6412328");

StudentView view = new StudentView();

StudentController controller = new StudentController(model, view);

controller.updateView();

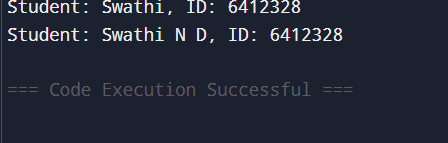
controller.updateName("Swathi N D");

controller.updateView();

}

}

**OUTPUT:**



**Exercise 11: Implementing Dependency Injection**

**CODE:**

public class Main {

interface CustomerRepository {

String findCustomerById(String id);

}

static class CustomerRepositoryImpl implements CustomerRepository {

public String findCustomerById(String id) {

return "Customer Name: Swathi (ID: " + id + ")";

}

}

static class CustomerService {

private CustomerRepository repository;

public CustomerService(CustomerRepository repository) {

this.repository = repository;

}

public void getCustomerDetails(String id) {

System.out.println(repository.findCustomerById(id));

}

}

public static void main(String[] args) {

CustomerRepository repo = new CustomerRepositoryImpl();

CustomerService service = new CustomerService(repo);

service.getCustomerDetails("6412328");

}

}

**OUTPUT:**

