**Exercise 1: Implementing the Singleton Pattern**

class Logger{

private static Logger instance;

private Logger() {

System.out.println("Logger instance created.");

}

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();

logger1.log("Initialization System...");

Logger logger2 = Logger.getInstance();

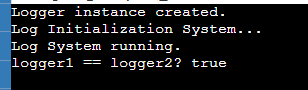
logger2.log("System running.");

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

}

}

**Output**



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

interface Document {

void open();

}

class wordDoc implements Document{

public void open(){

System.out.println("Opening word Document...");

}

}

class PdfDoc implements Document{

public void open(){

System.out.println("Opening Pdf Document...");

}

}

class ExcelDoc implements Document{

public void open(){

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

}

}

abstract class DocumentFactory{

public abstract Document createDocument();

}

class WordFactory extends DocumentFactory{

public Document createDocument(){

return new wordDoc();

}

}

class PdfFactory extends DocumentFactory{

public Document createDocument(){

return new PdfDoc();

}

}

class ExcelFactory extends DocumentFactory{

public Document createDocument(){

return new ExcelDoc();

}

}

public class Main {

public static void main(String[] args) {

DocumentFactory wordFactory = new WordFactory();

Document word = wordFactory.createDocument();

word.open();

DocumentFactory pdfFactory = new PdfFactory();

Document pdf = pdfFactory.createDocument();

pdf.open();

DocumentFactory excelFactory = new ExcelFactory();

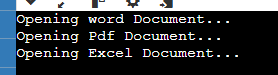
Document excel = excelFactory.createDocument();

excel.open();

}

}

**Output**



**Exercise 3: Implementing the Builder Pattern**

class Computer {

private String CPU;

private String RAM;

private String storage;

private String graphicsCard;

private boolean wifiEnabled;

private Computer(Builder builder) {

this.CPU = builder.CPU;

this.RAM = builder.RAM;

this.storage = builder.storage;

this.graphicsCard = builder.graphicsCard;

this.wifiEnabled = builder.wifiEnabled;

}

public static class Builder {

private String CPU;

private String RAM;

private String storage;

private String graphicsCard;

private boolean wifiEnabled;

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 setGraphicsCard(String graphicsCard) {

this.graphicsCard = graphicsCard;

return this;

}

public Builder setWifiEnabled(boolean wifiEnabled) {

this.wifiEnabled = wifiEnabled;

return this;

}

public Computer build() {

return new Computer(this);

}

}

public void specs() {

System.out.println("Computer Specs:");

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

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

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

System.out.println("Graphics Card: " + graphicsCard);

System.out.println("WiFi Enabled: " + wifiEnabled);

System.out.println("\n");

}

}

public class Main {

public static void main(String[] args) {

Computer gamingPC = new Computer.Builder()

.setCPU("Intel i9")

.setRAM("32GB")

.setStorage("1TB SSD")

.setGraphicsCard("NVIDIA RTX 4090")

.setWifiEnabled(true)

.build();

Computer officePC = new Computer.Builder()

.setCPU("Intel i5")

.setRAM("8GB")

.setStorage("512GB SSD")

.setWifiEnabled(false)

.build();

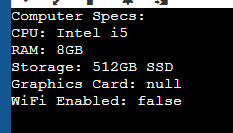
gamingPC.specs();

officePC.specs();

}

}

**Output**



**Exercise 4: Implementing the Adapter Pattern**

interface PaymentProcessor {

void processPayment(double amount);

}

class ABCGateway {

public void makePayment(double amount) {

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

}

}

class KLMGateway {

public void sendPayment(double amount) {

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

}

}

class ABCAdapter implements PaymentProcessor {

private ABCGateway abc = new ABCGateway();

public void processPayment(double amount) {

abc.makePayment(amount);

}

}

class KLMAdapter implements PaymentProcessor {

private KLMGateway klm = new KLMGateway();

public void processPayment(double amount) {

klm.sendPayment(amount);

}

}

public class Main {

public static void main(String[] args) {

PaymentProcessor abc = new ABCAdapter();

PaymentProcessor klm = new KLMAdapter();

abc.processPayment(1000.0);

klm.processPayment(2500.0);

}

}

**Output**



**Exercise 5: Implementing the Decorator Pattern**

interface Notifier {

void send(String message);

}

class EmailNotifier implements Notifier {

public void send(String message) {

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

}

}

abstract class NotifierDecorator implements Notifier {

protected Notifier wrappedNotifier;

public NotifierDecorator(Notifier notifier) {

this.wrappedNotifier = notifier;

}

public void send(String message) {

wrappedNotifier.send(message);

}

}

class SMSNotifierDecorator extends NotifierDecorator {

public SMSNotifierDecorator(Notifier notifier) {

super(notifier);

}

public void send(String message) {

super.send(message);

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

}

}

class SlackNotifierDecorator extends NotifierDecorator {

public SlackNotifierDecorator(Notifier notifier) {

super(notifier);

}

public void send(String message) {

super.send(message);

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

}

}

public class Main {

public static void main(String[] args) {

Notifier notifier = new EmailNotifier();

notifier = new SMSNotifierDecorator(notifier);

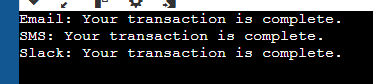
notifier = new SlackNotifierDecorator(notifier);

notifier.send("Your transaction is complete.");

}

}

**Output**



**Exercise 6: Implementing the Proxy Pattern**

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 image from remote server: " + filename);

}

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("(Cache hit) No need to reload: " + filename);

}

realImage.display();

}

}

public class Main {

public static void main(String[] args) {

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

Image image2 = new ProxyImage("city.png");

image1.display();

System.out.println();

image1.display();

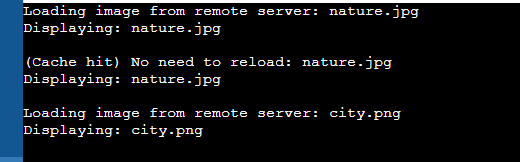
System.out.println();

image2.display();

}

}

**Output**



**Exercise 7: Implementing the Observer Pattern**

import java.util.\*;

interface Stock {

void register(Observer o);

void deregister(Observer o);

void notifyObservers();

}

interface Observer {

void update(String stockName, double price);

}

class StockMarket implements Stock {

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

private String stockName;

private double price;

public void setStockPrice(String stockName, double price) {

this.stockName = stockName;

this.price = price;

notifyObservers();

}

public void register(Observer o) {

observers.add(o);

}

public void deregister(Observer o) {

observers.remove(o);

}

public void notifyObservers() {

for (Observer o : observers) {

o.update(stockName, price);

}

}

}

class MobileApp implements Observer {

public void update(String stockName, double price) {

System.out.println("MobileApp: " + stockName + " updated to ₹" + price);

}

}

class WebApp implements Observer {

public void update(String stockName, double price) {

System.out.println("WebApp: " + stockName + " updated to ₹" + price);

}

}

public class Main {

public static void main(String[] args) {

StockMarket market = new StockMarket();

Observer mobileApp = new MobileApp();

Observer webApp = new WebApp();

market.register(mobileApp);

market.register(webApp);

market.setStockPrice("XYZ", 3560.75);

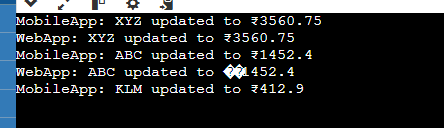
market.setStockPrice("ABC", 1452.40);

market.deregister(webApp);

market.setStockPrice("KLM", 412.90);

}}

**Output**



**Exercise 8: Implementing the Strategy Pattern**

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 ₹" + 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 ₹" + amount + " using PayPal account: " + email);

}

}

class PaymentContext {

private PaymentStrategy strategy;

public void setStrategy(PaymentStrategy strategy) {

this.strategy = strategy;

}

public void executePayment(double amount) {

if (strategy == null) {

System.out.println("Payment strategy not set!");

} else {

strategy.pay(amount);

}

}

}

public class Main {

public static void main(String[] args) {

PaymentContext context = new PaymentContext();

context.setStrategy(new CreditCardPayment("1234-5678-9876-5432"));

context.executePayment(1500.00);

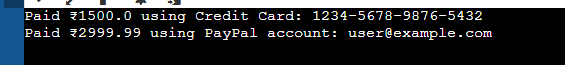
context.setStrategy(new [PayPalPayment("user@example.com](mailto:PayPalPayment("user@example.com)"));

context.executePayment(2999.99);

}

}

**Output**



**Exercise 9: Implementing the Command Pattern**

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 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**

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 String getId() { return id; }

public String getGrade() { return grade; }

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

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 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(), model.getGrade());

}

public void setStudentName(String name) { model.setName(name); }

public void setStudentGrade(String grade) { model.setGrade(grade); }

}

public class Main {

public static void main(String[] args) {

Student student = new Student("Lakshmi", "S123", "A");

StudentView view = new StudentView();

StudentController controller = new StudentController(student, view);

controller.updateView();

controller.setStudentName("Prabha Lakshmi");

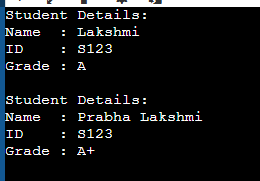
controller.setStudentGrade("A+");

controller.updateView();

}

}

**Output**



**Exercise 11: Implementing Dependency Injection**

interface CustomerRepository {

String findCustomerById(String id);

}

class CustomerRepositoryImpl implements CustomerRepository {

public String findCustomerById(String id) {

return "Customer[id=" + id + ", name=Lakshmi Prabha]";

}

}

class CustomerService {

private CustomerRepository repository;

public CustomerService(CustomerRepository repository) {

this.repository = repository;

}

public void showCustomer(String id) {

String customer = repository.findCustomerById(id);

System.out.println("Fetched: " + customer);

}

}

public class Main {

public static void main(String[] args) {

CustomerRepository repo = new CustomerRepositoryImpl();

CustomerService service = new CustomerService(repo);

service.showCustomer("C101");

}

}

**Output**

