**Exercise 1: Implementing the Singleton Pattern**

public class SingletonPattern {

// Singleton Logger class

static class Logger {

// Step 1: private static instance

private static Logger instance;

// Step 2: private constructor

private Logger() {

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

}

// Step 3: public static method to get the instance

public static Logger getInstance() {

if (instance == null) {

instance = new Logger();

}

return instance;

}

// Step 4: log method

public void log(String msg) {

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

}

}

// Main method to test the Singleton

public static void main(String[] args) {

Logger l1 = Logger.getInstance();

Logger l2 = Logger.getInstance();

l1.log("First log message.");

l2.log("Second log message.");

if (l1 == l2) {

System.out.println("Both logger instances are same. Singleton pattern is working.");

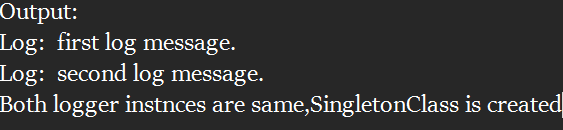
} else {

System.out.println("Logger instances are different. Singleton pattern failed.");

}

}

}



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

import java.util.\*;

interface Document {

void open();

}

class WordDocument implements Document {

public void open() {

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

}

}

class PdfDocument implements Document {

public void open() {

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

}

}

class ExcelDocument implements Document {

public void open() {

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

}

}

abstract class DocumentFactory {

public abstract Document createDocument();

}

class WordDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new WordDocument();

}

}

class PdfDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new PdfDocument();

}

}

class ExcelDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new ExcelDocument();

}

}

public class Main {

public static void main(String[] args) {

// Word document

DocumentFactory wordFactory = new WordDocumentFactory();

Document wordDoc = wordFactory.createDocument();

wordDoc.open();

// PDF document

DocumentFactory pdfFactory = new PdfDocumentFactory();

Document pdfDoc = pdfFactory.createDocument();

pdfDoc.open();

// Excel document

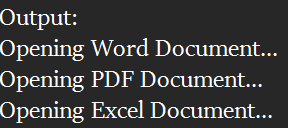
DocumentFactory excelFactory = new ExcelDocumentFactory();

Document excelDoc = excelFactory.createDocument();

excelDoc.open();

}

}



**Exercise 3: Implementing the Builder Pattern**

public class Computer {

private String cpu;

private String ram;

private String storage;

// Private constructor to enforce object creation via Builder

private Computer(Builder builder) {

this.cpu = builder.cpu;

this.ram = builder.ram;

this.storage = builder.storage;

}

// Static nested Builder class

public 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);

}

}

// Method to display computer specs

public void showSpecs() {

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

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

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

}

// Main method to test the builder

public static void main(String[] args) {

Computer myComputer = new Computer.Builder()

.setCpu("Intel i7")

.setRam("16GB")

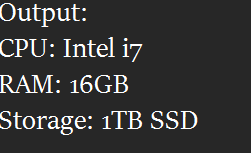
.setStorage("1TB SSD")

.build();

myComputer.showSpecs();

}

}



**Exercise 4: Implementing the Adapter Pattern**

interface PaymentProcessor {

void processPayment(double amount);

}

class PayPalGateway {

public void sendPayment(double amount) {

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

}

}

class StripeGateway {

public void makePayment(double amountInDollars) {

System.out.println("Paid ₹" + amountInDollars + " using Stripe.");

}

}

class RazorpayGateway {

public void payNow(double value) {

System.out.println("Paid ₹" + value + " using Razorpay.");

}

}

// Adapter Classes (Adapters implement the target interface)

class PayPalAdapter implements PaymentProcessor {

private PayPalGateway paypal = new PayPalGateway();

@Override

public void processPayment(double amount) {

paypal.sendPayment(amount);

}

}

class StripeAdapter implements PaymentProcessor {

private StripeGateway stripe = new StripeGateway();

@Override

public void processPayment(double amount) {

stripe.makePayment(amount);

}

}

class RazorpayAdapter implements PaymentProcessor {

private RazorpayGateway razorpay = new RazorpayGateway();

@Override

public void processPayment(double amount) {

razorpay.payNow(amount);

}

}

// Main Class to test the Adapter Pattern

public class Main {

public static void main(String[] args) {

PaymentProcessor paypalProcessor = new PayPalAdapter();

PaymentProcessor stripeProcessor = new StripeAdapter();

PaymentProcessor razorpayProcessor = new RazorpayAdapter();

System.out.println("=== Payment Processing System Using Adapter Pattern ===");

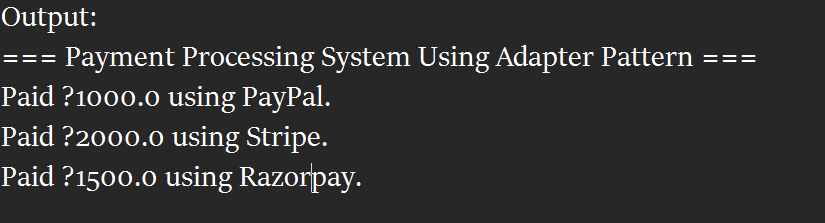
paypalProcessor.processPayment(1000.0);

stripeProcessor.processPayment(2000.0);

razorpayProcessor.processPayment(1500.0);

}

}



**Exercise 5: Implementing the Decorator Pattern**

// Component Interface

interface Notifier {

void send(String message);

}

// Concrete Component

class EmailNotifier implements Notifier {

@Override

public void send(String message) {

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

}

}

// Abstract Decorator

abstract class NotifierDecorator implements Notifier {

protected Notifier wrappedNotifier;

public NotifierDecorator(Notifier notifier) {

this.wrappedNotifier = notifier;

}

@Override

public void send(String message) {

wrappedNotifier.send(message); // Base behavior

}

}

// Concrete Decorator - SMS

class SMSNotifierDecorator extends NotifierDecorator {

public SMSNotifierDecorator(Notifier notifier) {

super(notifier);

}

@Override

public void send(String message) {

super.send(message); // Call previous notifier

sendSMS(message); // Add SMS behavior

}

private void sendSMS(String message) {

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

}

}

// Concrete Decorator - Slack

class SlackNotifierDecorator extends NotifierDecorator {

public SlackNotifierDecorator(Notifier notifier) {

super(notifier);

}

@Override

public void send(String message) {

super.send(message); // Call previous notifier

sendSlack(message); // Add Slack behavior

}

private void sendSlack(String message) {

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

}

}

// Main Class

public class Main {

public static void main(String[] args) {

// Step 1: Start with Email Notifier

Notifier baseNotifier = new EmailNotifier();

// Step 2: Add SMS on top of Email

Notifier smsNotifier = new SMSNotifierDecorator(baseNotifier);

// Step 3: Add Slack on top of SMS

Notifier fullNotifier = new SlackNotifierDecorator(smsNotifier);

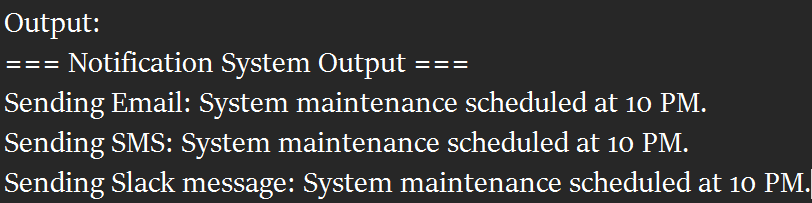
// Final Notification Chain

System.out.println("=== Notification System Output ===");

fullNotifier.send("System maintenance scheduled at 10 PM.");

}

}



**Exercise 6: Implementing the Proxy Pattern**

interface Image {

void display();

}

// Step 3: Real Subject

class RealImage implements Image {

private String filename;

public RealImage(String filename) {

this.filename = filename;

loadFromRemoteServer(); // Simulate heavy operation

}

private void loadFromRemoteServer() {

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

try {

Thread.sleep(1000); // Simulate network delay

} catch (InterruptedException e) {

e.printStackTrace();

}

}

@Override

public void display() {

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

}

}

// Step 4: Proxy Class with Lazy Loading & Caching

class ProxyImage implements Image {

private String filename;

private RealImage realImage;

public ProxyImage(String filename) {

this.filename = filename;

}

@Override

public void display() {

if (realImage == null) {

System.out.println("(First time) Creating RealImage...");

realImage = new RealImage(filename); // Lazy initialization

} else {

System.out.println("(From cache) Using existing RealImage...");

}

realImage.display();

}

}

// Step 5: Main Class for Testing

public class Main {

public static void main(String[] args) {

System.out.println("=== Proxy Pattern: Image Viewer ===");

Image image1 = new ProxyImage("beach\_photo.jpg");

// First display - should load from remote

image1.display();

System.out.println();

// Second display - should use cached image

image1.display();

System.out.println();

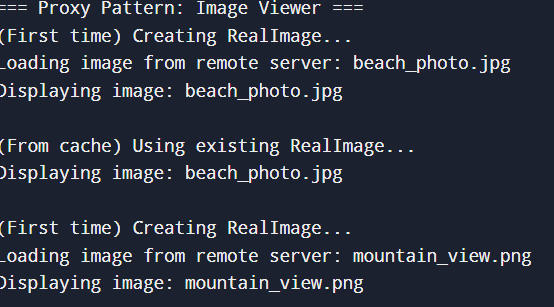
// Another new image - new load

Image image2 = new ProxyImage("mountain\_view.png");

image2.display();

}

}



**Exercise 7: Implementing the Observer Pattern**

import java.util.\*;

// Observer Interface

interface Observer {

void update(String stockName, double price);

}

// Subject Interface

interface Stock {

void registerObserver(Observer o);

void removeObserver(Observer o);

void notifyObservers();

}

// Concrete Subject

class StockMarket implements Stock {

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

private String stockName;

private double stockPrice;

public void setStockData(String stockName, double price) {

this.stockName = stockName;

this.stockPrice = price;

notifyObservers();

}

@Override

public void registerObserver(Observer o) {

observers.add(o);

}

@Override

public void removeObserver(Observer o) {

observers.remove(o);

}

@Override

public void notifyObservers() {

for (Observer o : observers) {

o.update(stockName, stockPrice);

}

}

}

// Concrete Observer: MobileApp

class MobileApp implements Observer {

private String appName;

public MobileApp(String appName) {

this.appName = appName;

}

@Override

public void update(String stockName, double price) {

System.out.println(appName + ": Stock " + stockName + " is now $" + price);

}

}

// Concrete Observer: WebApp

class WebApp implements Observer {

private String appName;

public WebApp(String appName) {

this.appName = appName;

}

@Override

public void update(String stockName, double price) {

System.out.println(appName + ": Stock " + stockName + " updated to $" + price);

}

}

// ✅ Main Class (must be 'Main' for online compilers)

public class Main {

public static void main(String[] args) {

StockMarket stockMarket = new StockMarket();

Observer mobileApp = new MobileApp("Mobile App");

Observer webApp = new WebApp("Web App");

stockMarket.registerObserver(mobileApp);

stockMarket.registerObserver(webApp);

stockMarket.setStockData("TCS", 3445.50);

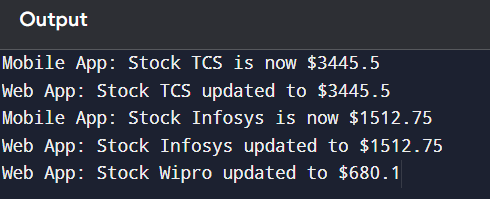
stockMarket.setStockData("Infosys", 1512.75);

stockMarket.removeObserver(mobileApp);

stockMarket.setStockData("Wipro", 680.10);

}

}



**Exercise 8: Implementing the Strategy Pattern**

interface PaymentStrategy {

void pay(double amount);

}

// Concrete Strategy: Credit Card

class CreditCardPayment implements PaymentStrategy {

private String cardNumber;

private String cardHolderName;

public CreditCardPayment(String cardNumber, String cardHolderName) {

this.cardNumber = cardNumber;

this.cardHolderName = cardHolderName;

}

@Override

public void pay(double amount) {

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

}

}

// Concrete Strategy: PayPal

class PayPalPayment implements PaymentStrategy {

private String email;

public PayPalPayment(String email) {

this.email = email;

}

@Override

public void pay(double amount) {

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

}

}

// Context Class

class PaymentContext {

private PaymentStrategy paymentStrategy;

public void setPaymentStrategy(PaymentStrategy strategy) {

this.paymentStrategy = strategy;

}

public void executePayment(double amount) {

if (paymentStrategy == null) {

System.out.println("No payment method selected!");

} else {

paymentStrategy.pay(amount);

}

}

}

public class Main {

public static void main(String[] args) {

PaymentContext context = new PaymentContext();

// Pay using Credit Card

PaymentStrategy creditCard = new CreditCardPayment("1234-5678-9876-5432", "Bindhu Mechineeni");

context.setPaymentStrategy(creditCard);

context.executePayment(1500.00);

// Pay using PayPal

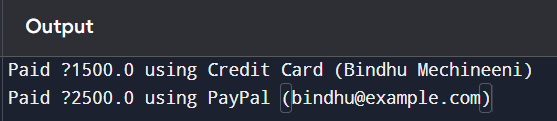
PaymentStrategy payPal = new PayPalPayment("bindhu@example.com");

context.setPaymentStrategy(payPal);

context.executePayment(2500.00);

}

}



**Exercise 9: Implementing the Command Pattern**

interface Command {

void execute();

}

// Receiver Class

class Light {

private String location;

public Light(String location) {

this.location = location;

}

public void turnOn() {

System.out.println(location + " light is ON");

}

public void turnOff() {

System.out.println(location + " light is OFF");

}

}

// Concrete Command: Turn On Light

class LightOnCommand implements Command {

private Light light;

public LightOnCommand(Light light) {

this.light = light;

}

@Override

public void execute() {

light.turnOn();

}

}

// Concrete Command: Turn Off Light

class LightOffCommand implements Command {

private Light light;

public LightOffCommand(Light light) {

this.light = light;

}

@Override

public void execute() {

light.turnOff();

}

}

// Invoker Class

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 assigned!");

}

}

}

public class Main {

public static void main(String[] args) {

// Receiver

Light livingRoomLight = new Light("Living Room");

// Concrete Commands

Command lightOn = new LightOnCommand(livingRoomLight);

Command lightOff = new LightOffCommand(livingRoomLight);

// Invoker

RemoteControl remote = new RemoteControl();

// Turn on the light

remote.setCommand(lightOn);

remote.pressButton();

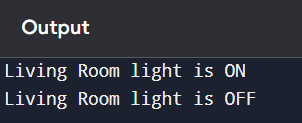
// Turn off the light

remote.setCommand(lightOff);

remote.pressButton();

}

}



**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;

}

// Getters

public String getName() { return name; }

public String getId() { return id; }

public String getGrade() { return grade; }

// Setters

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

public void setId(String id) { this.id = id; }

public void setGrade(String grade) { this.grade = grade; }

}

// View Class

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("---------------------------");

}

}

// Controller Class

class StudentController {

private Student model;

private StudentView view;

public StudentController(Student model, StudentView view) {

this.model = model;

this.view = view;

}

// Update model data

public void setStudentName(String name) {

model.setName(name);

}

public void setStudentId(String id) {

model.setId(id);

}

public void setStudentGrade(String grade) {

model.setGrade(grade);

}

// Get model data

public String getStudentName() {

return model.getName();

}

public String getStudentId() {

return model.getId();

}

public String getStudentGrade() {

return model.getGrade();

}

// Display via view

public void updateView() {

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

}

}

public class Main {

public static void main(String[] args) {

// Create model

Student student = new Student("Bindhu", "S101", "A");

// Create view

StudentView view = new StudentView();

// Create controller

StudentController controller = new StudentController(student, view);

// Initial display

controller.updateView();

// Update student data

controller.setStudentName("Mechineeni Bindhu");

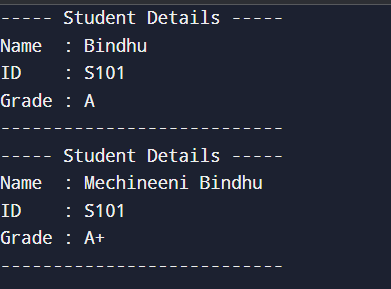
controller.setStudentGrade("A+");

// Updated display

controller.updateView();

}

}



**Exercise 11: Implementing Dependency Injection**

class Customer {

private int id;

private String name;

public Customer(int id, String name) {

this.id = id;

this.name = name;

}

public int getId() { return id; }

public String getName() { return name; }

}

// Repository Interface

interface CustomerRepository {

Customer findCustomerById(int id);

}

// Concrete Repository Implementation

class CustomerRepositoryImpl implements CustomerRepository {

@Override

public Customer findCustomerById(int id) {

// In real-world, you'd fetch from DB. Here it's hardcoded.

if (id == 1) {

return new Customer(1, "Bindhu Mechineeni");

} else if (id == 2) {

return new Customer(2, "John Doe");

}

return null;

}

}

// Service Class

class CustomerService {

private CustomerRepository customerRepository;

// Constructor-based Dependency Injection

public CustomerService(CustomerRepository customerRepository) {

this.customerRepository = customerRepository;

}

public void getCustomerDetails(int id) {

Customer customer = customerRepository.findCustomerById(id);

if (customer != null) {

System.out.println("Customer Found:");

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

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

} else {

System.out.println("Customer with ID " + id + " not found.");

}

}

}

public class Main {

public static void main(String[] args) {

// Step 1: Create repository implementation

CustomerRepository repo = new CustomerRepositoryImpl();

// Step 2: Inject dependency into service using constructor

CustomerService service = new CustomerService(repo);

// Step 3: Use the service

service.getCustomerDetails(1);

service.getCustomerDetails(2);

service.getCustomerDetails(99); // Not found

}

}

