Exercise 1: Singleton Pattern

// Logger.java

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);

}

}

// TestLogger.java

public class TestLogger {

public static void main(String[] args) {

Logger logger1 = Logger.getInstance();

Logger logger2 = Logger.getInstance();

logger1.log("First message");

logger2.log("Second message");

System.out.println(logger1 == logger2); // true

}

}

Exercise 2: Factory Method Pattern

// Document.java

public interface Document {

void open();

}

// WordDocument.java

public class WordDocument implements Document {

public void open() {

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

}

}

// PdfDocument.java

public class PdfDocument implements Document {

public void open() {

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

}

}

// ExcelDocument.java

public class ExcelDocument implements Document {

public void open() {

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

}

}

// DocumentFactory.java

public abstract class DocumentFactory {

public abstract Document createDocument();

}

// WordFactory.java

public class WordFactory extends DocumentFactory {

public Document createDocument() {

return new WordDocument();

}

}

// PdfFactory.java

public class PdfFactory extends DocumentFactory {

public Document createDocument() {

return new PdfDocument();

}

}

// ExcelFactory.java

public class ExcelFactory extends DocumentFactory {

public Document createDocument() {

return new ExcelDocument();

}

}

// TestFactory.java

public class TestFactory {

public static void main(String[] args) {

DocumentFactory factory = new WordFactory();

Document doc = factory.createDocument();

doc.open();

}

}

Exercise 3: Builder Pattern

// Computer.java

public 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;

}

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);

}

}

public void showSpecs() {

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

}

}

// TestBuilder.java

public class TestBuilder {

public static void main(String[] args) {

Computer comp = new Computer.Builder()

.setCPU("i7")

.setRAM("16GB")

.setStorage("512GB SSD")

.build();

comp.showSpecs();

}

}

Exercise 4: Adapter Pattern

// PaymentProcessor.java

public interface PaymentProcessor {

void processPayment(double amount);

}

// StripeGateway.java

public class StripeGateway {

public void makePayment(double amount) {

System.out.println("Payment made using Stripe: " + amount);

}

}

// StripeAdapter.java

public class StripeAdapter implements PaymentProcessor {

private StripeGateway stripe;

public StripeAdapter() {

this.stripe = new StripeGateway();

}

public void processPayment(double amount) {

stripe.makePayment(amount);

}

}

// TestAdapter.java

public class TestAdapter {

public static void main(String[] args) {

PaymentProcessor processor = new StripeAdapter();

processor.processPayment(2000);

}

}

Exercise 5: Decorator Pattern

// Notifier.java

public interface Notifier {

void send(String message);

}

// EmailNotifier.java

public class EmailNotifier implements Notifier {

public void send(String message) {

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

}

}

// NotifierDecorator.java

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

public class SMSNotifierDecorator extends NotifierDecorator {

public SMSNotifierDecorator(Notifier notifier) {

super(notifier);

}

public void send(String message) {

super.send(message);

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

}

}

// TestDecorator.java

public class TestDecorator {

public static void main(String[] args) {

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

notifier.send("Order placed");

}

}

Exercise 6: Proxy Pattern

// Image.java

public interface Image {

void display();

}

// RealImage.java

public class RealImage implements Image {

private String filename;

public RealImage(String filename) {

this.filename = filename;

loadFromDisk();

}

private void loadFromDisk() {

System.out.println("Loading " + filename);

}

public void display() {

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

}

}

// ProxyImage.java

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();

}

}

// TestProxy.java

public class TestProxy {

public static void main(String[] args) {

Image img = new ProxyImage("test.jpg");

img.display(); // loads and displays

img.display(); // just displays

}

}

Exercise 7: Observer Pattern

// Observer.java

public interface Observer {

void update(float price);

}

// Stock.java

public interface Stock {

void register(Observer obs);

void deregister(Observer obs);

void notifyObservers();

}

// StockMarket.java

import java.util.\*;

public class StockMarket implements Stock {

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

private float stockPrice;

public void register(Observer obs) {

observers.add(obs);

}

public void deregister(Observer obs) {

observers.remove(obs);

}

public void setStockPrice(float price) {

this.stockPrice = price;

notifyObservers();

}

public void notifyObservers() {

for (Observer obs : observers) {

obs.update(stockPrice);

}

}

}

// MobileApp.java

public class MobileApp implements Observer {

public void update(float price) {

System.out.println("Mobile App: Stock Price Updated to " + price);

}

}

// WebApp.java

public class WebApp implements Observer {

public void update(float price) {

System.out.println("Web App: Stock Price Updated to " + price);

}

}

// TestObserver.java

public class TestObserver {

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.setStockPrice(101.5f);

}

}

Exercise 8: Strategy Pattern

// PaymentStrategy.java

public interface PaymentStrategy {

void pay(double amount);

}

// CreditCardPayment.java

public class CreditCardPayment implements PaymentStrategy {

public void pay(double amount) {

System.out.println("Paid with credit card: " + amount);

}

}

// PayPalPayment.java

public class PayPalPayment implements PaymentStrategy {

public void pay(double amount) {

System.out.println("Paid with PayPal: " + amount);

}

}

// PaymentContext.java

public class PaymentContext {

private PaymentStrategy strategy;

public PaymentContext(PaymentStrategy strategy) {

this.strategy = strategy;

}

public void setStrategy(PaymentStrategy strategy) {

this.strategy = strategy;

}

public void pay(double amount) {

strategy.pay(amount);

}

}

// TestStrategy.java

public class TestStrategy {

public static void main(String[] args) {

PaymentContext context = new PaymentContext(new CreditCardPayment());

context.pay(2500);

context.setStrategy(new PayPalPayment());

context.pay(1500);

}

}

**Exercise 9: Command Pattern**

// Command.java

public interface Command {

void execute();

}

// Light.java

public class Light {

public void turnOn() {

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

}

public void turnOff() {

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

}

}

// LightOnCommand.java

public class LightOnCommand implements Command {

private Light light;

public LightOnCommand(Light light) {

this.light = light;

}

public void execute() {

light.turnOn();

}

}

// LightOffCommand.java

public class LightOffCommand implements Command {

private Light light;

public LightOffCommand(Light light) {

this.light = light;

}

public void execute() {

light.turnOff();

}

}

// RemoteControl.java

public class RemoteControl {

private Command command;

public void setCommand(Command command) {

this.command = command;

}

public void pressButton() {

command.execute();

}

}

// TestCommand.java

public class TestCommand {

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(); // Light is ON

remote.setCommand(lightOff);

remote.pressButton(); // Light is OFF

}

}

Exercise 10: MVC Pattern

// Student.java

public class Student {

private String id;

private String name;

private String grade;

public String getId() { return id; }

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

public String getName() { return name; }

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

public String getGrade() { return grade; }

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

}

// StudentView.java

public class StudentView {

public void displayStudentDetails(String name, String id, String grade) {

System.out.println("Student Details:");

System.out.println("ID: " + id);

System.out.println("Name: " + name);

System.out.println("Grade: " + grade);

}

}

// StudentController.java

public class StudentController {

private Student model;

private StudentView view;

public StudentController(Student model, StudentView view) {

this.model = model;

this.view = view;

}

public void setStudentName(String name) {

model.setName(name);

}

public void setStudentId(String id) {

model.setId(id);

}

public void setStudentGrade(String grade) {

model.setGrade(grade);

}

public void updateView() {

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

}

}

// TestMVC.java

public class TestMVC {

public static void main(String[] args) {

Student model = new Student();

model.setId("S101");

model.setName("Ananya");

model.setGrade("A");

StudentView view = new StudentView();

StudentController controller = new StudentController(model, view);

controller.updateView();

controller.setStudentGrade("A+");

controller.updateView();

}

}

**Exercise 11: Dependency Injection**

// Customer.java

public 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; }

}

// CustomerRepository.java

public interface CustomerRepository {

Customer findCustomerById(String id);

}

// CustomerRepositoryImpl.java

public class CustomerRepositoryImpl implements CustomerRepository {

public Customer findCustomerById(String id) {

// Simulated DB lookup

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

}

}

// CustomerService.java

public class CustomerService {

private CustomerRepository repository;

public CustomerService(CustomerRepository repository) {

this.repository = repository;

}

public void printCustomer(String id) {

Customer customer = repository.findCustomerById(id);

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

}

}

// TestDI.java

public class TestDI {

public static void main(String[] args) {

CustomerRepository repo = new CustomerRepositoryImpl();

CustomerService service = new CustomerService(repo);

service.printCustomer("C001");

}

}