**Exercise 1: Implementing the Singleton Pattern**

public class Main{

public static void main(String args[]){

Logger log1=Logger.getInstance();

Logger log2=Logger.getInstance();

log1.message("Logging from log1");

log2.message("Logging from log2");

System.out.println("Checking Instances "+(log1==log2));

}

}

class Logger{

private static Logger Instance;

private Logger(){

System.out.println("Instance is Created");

}

public static Logger getInstance(){

if(Instance==null){

Instance= new Logger();

}

return Instance;

}

public static void message(String text){

System.out.println(text);

}

}

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

public class Main {

public static void main(String[] args) {

WordDocumentFactory wordFactory = new WordDocumentFactory();

Document wordDoc = wordFactory.createDocument();

wordDoc.print();

DocumentFactory pdfFactory = new PdfDocumentFactory();

Document pdfDoc = pdfFactory.createDocument();

pdfDoc.print();

DocumentFactory excelFactory = new ExcelDocumentFactory();

Document excelDoc = excelFactory.createDocument();

excelDoc.print();

}

}

interface Document{

public void print();

}

class WordDocument implements Document{

public void print(){

System.out.println("This is Word Document");

}

}

class PdfDocument implements Document{

public void print(){

System.out.println("This is Pdf Document");

}

}

class ExcelDocument implements Document{

public void print(){

System.out.println("This is Excel Document");

}

}

abstract class DocumentFactory{

public abstract Document createDocument();

}

class WordDocumentFactory extends DocumentFactory {

@Override

public Document createDocument() {

return new WordDocument();

}

}

class PdfDocumentFactory extends DocumentFactory {

@Override

public Document createDocument() {

return new PdfDocument();

}

}

class ExcelDocumentFactory extends DocumentFactory {

@Override

public Document createDocument() {

return new ExcelDocument();

}

}

**Exercise 3: Implementing the Builder Pattern**

public class Main {

public static void main(String[] args) {

Computer comp1 = new Computer.Builder()

.setCPU("Intel i7")

.setRAM("16GB")

.setStorage("512GB SSD")

.build();

System.out.println("Computer 1 Configuration:");

comp1.showConfig();

Computer comp2 = new Computer.Builder()

.setCPU("AMD Ryzen 5")

.setRAM("8GB")

.setStorage("1TB HDD")

.build();

System.out.println("\nComputer 2 Configuration:");

comp2.showConfig();

}

}

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 void showConfig() {

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

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

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

}

}

}

**Exercise 4: Implementing the Adapter Pattern**

public class Main {

public static void main(String[] args) {

CreditCardAdapt creditCard = new CreditCardAdapt();

creditCard.processPayment(2500.00);

PaymentProcessor paypal = new PayPalAdapter();

paypal.processPayment(1800.00);

}

}

interface PaymentProcessor {

public void processPayment(double amount);

}

class CreditCardPayment{

public static void payWithCard(double amount){

System.out.println("Paid amount "+amount+" With CreditCard");

}

}

class PayPalPayment{

public static void payWithPal(double amount){

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

}

}

class CreditCardAdapt implements PaymentProcessor{

private CreditCardPayment creditCardPayment;

public CreditCardAdapt() {

this.creditCardPayment = new CreditCardPayment();

}

public void processPayment(double amount) {

creditCardPayment.payWithCard(amount);

}

}

class PayPalAdapter implements PaymentProcessor {

private PayPalPayment payPalPayment;

public PayPalAdapter() {

this.payPalPayment = new PayPalPayment();

}

public void processPayment(double amount) {

payPalPayment.payWithPal(amount);

}

}

**Exercise 5: Implementing the Decorator Pattern**

public class Main {

public static void main(String[] args) {

Notifier notifier = new EmailNotifier();

notifier = new SMSNotifierDecorator(notifier);

notifier = new SlackNotifierDecorator(notifier);

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

}

}

interface Notifier{

void send(String message);

}

class EmailNotifier implements Notifier{

public void send(String message){

System.out.println("Message is "+message);

}

}

abstract class NotifierDecorator implements Notifier {

protected Notifier notifier;

public NotifierDecorator(Notifier notifier) {

this.notifier = notifier;

}

@Override

public void send(String message) {

notifier.send(message);

}

}

class SMSNotifierDecorator extends NotifierDecorator {

public SMSNotifierDecorator(Notifier notifier) {

super(notifier);

}

@Override

public void send(String message) {

super.send(message);

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

}

}

class SlackNotifierDecorator extends NotifierDecorator {

public SlackNotifierDecorator(Notifier notifier) {

super(notifier);

}

@Override

public void send(String message) {

super.send(message);

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

}

}

**Exercise 6: Implementing the Proxy Pattern**

public class Main {

public static void main(String[] args) {

Image img1 = new ProxyImage("nature.jpg");

Image img2 = new ProxyImage("city.jpg");

img1.display();

img1.display();

img2.display();

}

}

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);

}

@Override

public void display() {

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

}

}

class ProxyImage implements Image {

private RealImage realImage;

private String fileName;

public ProxyImage(String fileName) {

this.fileName = fileName;

}

@Override

public void display() {

if (realImage == null) {

realImage = new RealImage(fileName);

}

realImage.display();

}

}

**Exercise 7: Implementing the Observer Pattern**

import java.util.\*;

interface Observer {

void update(String stockName, double stockPrice);

}

interface Stock {

void register(Observer o);

void deregister(Observer o);

void notifyObservers();

}

class StockMarket implements Stock {

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

private String stockName;

private double stockPrice;

public void setStock(String stockName, double stockPrice) {

this.stockName = stockName;

this.stockPrice = stockPrice;

notifyObservers();

}

@Override

public void register(Observer o) {

observers.add(o);

System.out.println("Observer Registered");

}

@Override

public void deregister(Observer o) {

observers.remove(o);

System.out.println("Observer Deregistered");

}

@Override

public void notifyObservers() {

for (Observer o : observers) {

o.update(stockName, stockPrice);

}

}

}

class MobileApp implements Observer {

@Override

public void update(String stockName, double stockPrice) {

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

}

}

class WebApp implements Observer {

@Override

public void update(String stockName, double stockPrice) {

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

}

}

public class Main {

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.setStock("TCS", 3500.50);

market.setStock("Infosys", 1450.75);

market.deregister(web);

market.setStock("HDFC Bank", 1700.00);

}

}

**Exercise 8: Implementing the Strategy Pattern**

class Main{

public static void main(String args[]){

PaymentContext payment=new PaymentContext();

payment.setPaymentContext(new CreditCardPayment());

payment.pay(2000);

payment.setPaymentContext(new PayPalPayment());

payment.pay(1000);

}

}

interface PaymentStrategy{

void pay(double amount);

}

class CreditCardPayment implements PaymentStrategy{

public void pay(double amount){

System.out.println("Paid "+amount+" With Credit card");

}

}

class PayPalPayment implements PaymentStrategy{

public void pay(double amount){

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

}

}

class PaymentContext {

PaymentStrategy paymentstrategy;

public void setPaymentContext(PaymentStrategy strategy){

this.paymentstrategy=strategy;

}

public void pay(double amount){

if(paymentstrategy==null){

System.out.println("No payment is set");

}

else{

paymentstrategy.pay(amount);

}

}

}

**Exercise 9: Implementing the Command Pattern**

interface Command{

void execute();

}

class LightOnCommand implements Command{

Light light;

LightOnCommand(Light light){

this.light=light;

}

public void execute(){

light.on();

}

}

class LightOffCommand implements Command{

Light light;

LightOffCommand(Light light){

this.light=light;

}

public void execute(){

light.off();

}

}

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.");

}

}

}

class Light {

void on(){

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

}

void off(){

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

}

}

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();

}

}

**Exercise 10: Implementing the MVC Pattern**

class Student{

String name;

String id;

String grade;

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 setId(String id) {

this.id = id;

}

public void setGrade(String grade) {

this.grade = grade;

}

}

class StudentView {

void displayStudentDetails(Student s){

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

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

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

System.out.println("Grade : " + s.getGrade());

}

}

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 String getStudentName() {

return model.getName();

}

public String getStudentId() {

return model.getId();

}

public String getStudentGrade() {

return model.getGrade();

}

public void updateView() {

view.displayStudentDetails(model);

}

}

public class Main {

public static void main(String[] args) {

Student student = new Student("Hari", "CS101", "A");

StudentView view = new StudentView();

StudentController controller = new StudentController(student, view);

controller.updateView();

controller.setStudentGrade("A+");

controller.updateView();

}

}

**Exercise 11: Implementing Dependency Injection**

interface CustomerRepository {

String findCustomerById(String id);

}

class CustomerRepositoryImpl implements CustomerRepository {

@Override

public String findCustomerById(String id) {

return "Customer[ID: " + id + ", Name: Harish, City: Hyderabad]";

}

}

class CustomerService {

private CustomerRepository repository;

public CustomerService(CustomerRepository repository) {

this.repository = repository;

}

public void displayCustomerDetails(String id) {

String customer = repository.findCustomerById(id);

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

}

}

public class Main {

public static void main(String[] args) {

CustomerRepository repo = new CustomerRepositoryImpl();

CustomerService service = new CustomerService(repo);

service.displayCustomerDetails("C102");

}

}