**Exercise 1: Singleton Pattern**

class Logger {

private static Logger instance;

private Logger() {}

public static Logger getInstance() {

if (instance == null) {

instance = new Logger();

}

return instance;

}

public void log(String message) {

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

}

}

public class LoggerTest {

public static void main(String[] args) {

Logger l1 = Logger.getInstance();

Logger l2 = Logger.getInstance();

System.out.println(l1 == l2);

}

}

**Exercise 2: Factory Method Pattern**

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 {

abstract Document createDocument();

}

class WordFactory extends DocumentFactory {

Document createDocument() {

return new WordDocument();

}

}

class PdfFactory extends DocumentFactory {

Document createDocument() {

return new PdfDocument();

}

}

class ExcelFactory extends DocumentFactory {

Document createDocument() {

return new ExcelDocument();

}

}

public class FactoryTest {

public static void main(String[] args) {

DocumentFactory factory = new PdfFactory();

Document doc = factory.createDocument();

doc.open();

}

}

**Exercise 3: Builder Pattern**

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 class BuilderTest {

public static void main(String[] args) {

Computer comp = new Computer.Builder().setCPU("i7").setRAM("16GB").setStorage("1TB").build();

}

}

**Exercise 4: Adapter Pattern**

interface PaymentProcessor {

void processPayment(double amount);

}

class PayPal {

void pay(double amount) {

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

}

}

class PayPalAdapter implements PaymentProcessor {

private PayPal paypal = new PayPal();

public void processPayment(double amount) {

paypal.pay(amount);

}

}

public class AdapterTest {

public static void main(String[] args) {

PaymentProcessor processor = new PayPalAdapter();

processor.processPayment(500);

}

}

**Exercise 5: Decorator Pattern**

interface Notifier {

void send();

}

class EmailNotifier implements Notifier {

public void send() {

System.out.println("Sending Email");

}

}

abstract class NotifierDecorator implements Notifier {

protected Notifier notifier;

NotifierDecorator(Notifier notifier) {

this.notifier = notifier;

}

}

class SMSNotifierDecorator extends NotifierDecorator {

SMSNotifierDecorator(Notifier notifier) {

super(notifier);

}

public void send() {

notifier.send();

System.out.println("Sending SMS");

}

}

public class DecoratorTest {

public static void main(String[] args) {

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

notifier.send();

}

}

**Exercise 6: Proxy Pattern**

interface Image {

void display();

}

class RealImage implements Image {

private String filename;

RealImage(String filename) {

this.filename = filename;

loadFromDisk();

}

private void loadFromDisk() {

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

}

public void display() {

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

}

}

class ProxyImage implements Image {

private RealImage realImage;

private String filename;

ProxyImage(String filename) {

this.filename = filename;

}

public void display() {

if (realImage == null) {

realImage = new RealImage(filename);

}

realImage.display();

}

}

public class ProxyTest {

public static void main(String[] args) {

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

img.display();

img.display();

}

}

**Exercise 7: Observer Pattern**

import java.util.\*;

interface Observer {

void update(float price);

}

interface Stock {

void register(Observer o);

void deregister(Observer o);

void notifyObservers();

}

class StockMarket implements Stock {

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

private float price;

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(price);

}

}

public void setPrice(float price) {

this.price = price;

notifyObservers();

}

}

class MobileApp implements Observer {

public void update(float price) {

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

}

}

public class ObserverTest {

public static void main(String[] args) {

StockMarket market = new StockMarket();

Observer app = new MobileApp();

market.register(app);

market.setPrice(123.45f);

}

}

**Exercise 8: Strategy Pattern**

interface PaymentStrategy {

void pay(double amount);

}

class CreditCardPayment implements PaymentStrategy {

public void pay(double amount) {

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

}

}

class PayPalPayment implements PaymentStrategy {

public void pay(double amount) {

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

}

}

class PaymentContext {

private PaymentStrategy strategy;

PaymentContext(PaymentStrategy strategy) {

this.strategy = strategy;

}

public void executePayment(double amount) {

strategy.pay(amount);

}

}

public class StrategyTest {

public static void main(String[] args) {

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

context.executePayment(1000);

}

}

**Exercise 9: Command Pattern**

interface Command {

void execute();

}

class Light {

void on() {

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

}

void off() {

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

}

}

class LightOnCommand implements Command {

private Light light;

LightOnCommand(Light light) {

this.light = light;

}

public void execute() {

light.on();

}

}

class LightOffCommand implements Command {

private Light light;

LightOffCommand(Light light) {

this.light = light;

}

public void execute() {

light.off();

}

}

class RemoteControl {

private Command command;

void setCommand(Command command) {

this.command = command;

}

void pressButton() {

command.execute();

}

}

public class CommandTest {

public static void main(String[] args) {

Light light = new Light();

Command on = new LightOnCommand(light);

Command off = new LightOffCommand(light);

RemoteControl remote = new RemoteControl();

remote.setCommand(on);

remote.pressButton();

remote.setCommand(off);

remote.pressButton();

}

}

**Exercise 10: MVC Pattern**

class Student {

private String name;

private int id;

private String grade;

Student(String name, int id, String grade) {

this.name = name;

this.id = id;

this.grade = grade;

}

String getName() { return name; }

int getId() { return id; }

String getGrade() { return grade; }

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

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

}

class StudentView {

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

System.out.println("Name: " + name + ", ID: " + id + ", Grade: " + grade);

}

}

class StudentController {

private Student model;

private StudentView view;

StudentController(Student model, StudentView view) {

this.model = model;

this.view = view;

}

void updateView() {

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

}

void setStudentName(String name) {

model.setName(name);

}

}

public class MVCTest {

public static void main(String[] args) {

Student model = new Student("John", 1, "A");

StudentView view = new StudentView();

StudentController controller = new StudentController(model, view);

controller.updateView();

controller.setStudentName("Mike");

controller.updateView();

}

}

**Exercise 11: Dependency Injection**

interface CustomerRepository {

String findCustomerById(int id);

}

class CustomerRepositoryImpl implements CustomerRepository {

public String findCustomerById(int id) {

return "Customer " + id;

}

}

class CustomerService {

private CustomerRepository repository;

CustomerService(CustomerRepository repository) {

this.repository = repository;

}

void getCustomer(int id) {

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

}

}

public class DIExample {

public static void main(String[] args) {

CustomerRepository repo = new CustomerRepositoryImpl();

CustomerService service = new CustomerService(repo);

service.getCustomer(101);

}

}