**Exercise 1: Implementing the Singleton Pattern**

public class Logger {

private static Logger instance;

private Logger() {} // private constructor

public static Logger getInstance() {

if (instance == null) {

instance = new Logger();

}

return instance;

}

public void log(String message) {

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

}

}

public class TestLogger {

public static void main(String[] args) {

Logger log1 = Logger.getInstance();

Logger log2 = Logger.getInstance();

System.out.println(log1 == log2); // true

}

}

**Exercise 2: Implementing the 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 {

public abstract Document createDocument();

}

class WordFactory extends DocumentFactory {

public Document createDocument() { return new WordDocument(); }

}

public class TestFactory {

public static void main(String[] args) {

DocumentFactory factory = new WordFactory();

Document doc = factory.createDocument();

doc.open();

}

}

**Exercise 3: Implementing the Builder Pattern**

public class Computer {

private String CPU, RAM, storage;

private Computer(Builder builder) {

this.CPU = builder.CPU;

this.RAM = builder.RAM;

this.storage = builder.storage;

}

public static class Builder {

private String CPU, RAM, 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 TestBuilder {

public static void main(String[] args) {

Computer comp = new Computer.Builder()

.setCPU("i7")

.setRAM("16GB")

.build();

}

}

**Exercise 4: Implementing the Adapter Pattern**

interface PaymentProcessor {

void processPayment();

}

class PayPal {

void sendMoney() { System.out.println("Paying via PayPal"); }

}

class PayPalAdapter implements PaymentProcessor {

private PayPal paypal = new PayPal();

public void processPayment() {

paypal.sendMoney();

}

}

public class TestAdapter {

public static void main(String[] args) {

PaymentProcessor processor = new PayPalAdapter();

processor.processPayment();

}

}

**Exercise 5: Implementing the 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;

public NotifierDecorator(Notifier notifier) {

this.notifier = notifier;

}

}

class SMSNotifier extends NotifierDecorator {

public SMSNotifier(Notifier notifier) {

super(notifier);

}

public void send() {

notifier.send();

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

}}

public class TestDecorator {

public static void main(String[] args) {

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

notifier.send(); // Sends Email and SMS }}

**Exercise 6: Implementing the Proxy Pattern**

interface Image {

void display();

}

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);

}

}

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();

}

}

public class TestProxy {

public static void main(String[] args) {

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

img.display(); // Loads and displays

img.display(); // Only displays

}

}

**Exercise 7: Implementing the Observer Pattern**

interface Observer {

void update(String stock);

}

interface Stock {

void register(Observer o);

void deregister(Observer o);

void notifyObservers();

}

class StockMarket implements Stock {

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

private String stock;

public void setStock(String stock) {

this.stock = stock;

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(stock);

}

}

}

class MobileApp implements Observer {

public void update(String stock) {

System.out.println("MobileApp received update: " + stock);

}

}

public class TestObserver {

public static void main(String[] args) {

StockMarket market = new StockMarket();

Observer app = new MobileApp();

market.register(app);

market.setStock("AAPL up 5%");

}

}

**Exercise 8: Implementing the Strategy Pattern**

interface PaymentStrategy {

void pay(int amount);

}

class CreditCardPayment implements PaymentStrategy {

public void pay(int amount) {

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

}

}

class PayPalPayment implements PaymentStrategy {

public void pay(int amount) {

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

}

}

class PaymentContext {

private PaymentStrategy strategy;

public void setStrategy(PaymentStrategy strategy) {

this.strategy = strategy;

}

public void pay(int amount) {

strategy.pay(amount);

}}

public class TestStrategy {

public static void main(String[] args) {

PaymentContext context = new PaymentContext();

context.setStrategy(new PayPalPayment());

context.pay(500);

}

}

**Exercise 9: Implementing the Command Pattern**

interface Command {

void execute();

}

class Light {

public void on() { System.out.println("Light On"); }

public void off() { System.out.println("Light Off"); }

}

class LightOnCommand implements Command {

private Light light;

public LightOnCommand(Light light) { this.light = light; }

public void execute() { light.on(); }

}

class RemoteControl {

private Command command;

public void setCommand(Command command) { this.command = command; }

public void pressButton() { command.execute(); }

}

public class TestCommand {

public static void main(String[] args) {

Light light = new Light();

Command lightOn = new LightOnCommand(light);

RemoteControl remote = new RemoteControl();

remote.setCommand(lightOn);

remote.pressButton();

}

}

**Exercise 10: Implementing the MVC Pattern**

class Student {

private String name;

public Student(String name) { this.name = name; }

public String getName() { return name; }

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

}

class StudentView {

public void displayStudentDetails(String name) {

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

}

}

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 updateView() {

view.displayStudentDetails(model.getName());

}

}

public class TestMVC {

public static void main(String[] args) {

Student model = new Student("John");

StudentView view = new StudentView();

StudentController controller = new StudentController(model, view);

controller.updateView();

controller.setStudentName("Alice");

controller.updateView();

}

}

**Exercise 11: Implementing Dependency Injection**

interface CustomerRepository {

String findCustomerById(int id);

}

class CustomerRepositoryImpl implements CustomerRepository {

public String findCustomerById(int id) {

return "Customer#" + id;

}

}

class CustomerService {

private CustomerRepository repo;

public CustomerService(CustomerRepository repo) {

this.repo = repo;

}

public void showCustomer(int id) {

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

}

}

public class TestDI {

public static void main(String[] args) {

CustomerRepository repo = new CustomerRepositoryImpl();

CustomerService service = new CustomerService(repo);

service.showCustomer(101);

}

}