**Exercise 1: Implementing the Singleton Pattern**

// Logger-Singleton class

public class Logger {

private static Logger instance;

private Logger() {

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

}

public static Logger getInstance() {

if (instance == null) {

instance = new Logger();

}

return instance;

}

public void log(String message) {

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

}

}

// SingletonTest-Test class

public class SingletonTest {

public static void main(String[] args) {

Logger logger1 = Logger.getInstance();

logger1.log("First log message");

Logger logger2 = Logger.getInstance();

logger2.log("Second log message");

if (logger1 == logger2) {

System.out.println("Both logger1 and logger2 are the same instance.");

} else {

System.out.println("Different instances exist (Singleton failed).");

}

} }

Output:-

Logger instance created.

Log: First log message

Log: Second log message

Both logger1 and logger2 are the same instance.

**Exercise 2: Factory Method Pattern**

package documents;

public interface Document {

void open();

}

public class WordDocument implements Document {

public void open() {

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

}

}

public class PdfDocument implements Document {

public void open() {

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

}

}

public class ExcelDocument implements Document {

public void open() {

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

}

}

public abstract class DocumentFactory {

public abstract Document createDocument();

}

public class WordDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new WordDocument();

}

}

public class PdfDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new PdfDocument();

}

}

public class ExcelDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new ExcelDocument();

}

}

public class TestFactory {

public static void main(String[] args) {

DocumentFactory factory;

factory = new WordDocumentFactory();

Document word = factory.createDocument();

word.open();

factory = new PdfDocumentFactory();

Document pdf = factory.createDocument();

pdf.open();

factory = new ExcelDocumentFactory();

Document excel = factory.createDocument();

excel.open();

}

}

Output: Opening Word document.

Opening PDF document.

Opening Excel document.

**Exercise 3: Builder Pattern**

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);

}

}

public class TestBuilder {

public static void main(String[] args) {

Computer comp = new Computer.Builder()

.setCPU("Intel i7")

.setRAM("16GB")

.setStorage("1TB SSD")

.build();

comp.showSpecs();

}

}

Output:-

CPU: Intel i7, RAM: 16GB, Storage: 1TB SSD

**Exercise 4: Adapter Pattern**

public interface PaymentProcessor {

void processPayment();

}

public class PayPal {

public void makePayment() {

System.out.println("Processing payment through PayPal.");

}

}

public class PayPalAdapter implements PaymentProcessor {

private PayPal paypal = new PayPal();

public void processPayment() {

paypal.makePayment();

}

}

public class TestAdapter {

public static void main(String[] args) {

PaymentProcessor processor = new PayPalAdapter();

processor.processPayment();

}

}

Output:-

Processing payment through PayPal.

**Exercise 5: Decorator Pattern**

public interface Notifier {

void send();

}

public class EmailNotifier implements Notifier {

public void send() {

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

}

}

public abstract class NotifierDecorator implements Notifier {

protected Notifier notifier;

public NotifierDecorator(Notifier notifier) {

this.notifier = notifier;

}

public void send() {

notifier.send();

}

}

public class SMSNotifier extends NotifierDecorator {

public SMSNotifier(Notifier notifier) {

super(notifier);

}

public void send() {

super.send();

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

}

}

public class TestDecorator {

public static void main(String[] args) {

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

notifier.send();

}

}

Output:-

Sending Email Notification

Sending SMS Notification

**Exercise 6: Proxy Pattern**

public interface Image {

void display();

}

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);

}

}

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();

}

}

public class TestProxy {

public static void main(String[] args) {

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

image.display();

image.display();

}

}

Output:-

Loading photo.jpg

Displaying photo.jpg

Displaying photo.jpg

**Exercise 7: Observer Pattern**

public interface Observer {

void update(String stock);

}

public interface Stock {

void register(Observer o);

void remove(Observer o);

void notifyObservers();

}

import java.util.\*;

public 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 remove(Observer o) {

observers.remove(o);

}

public void notifyObservers() {

for (Observer o : observers) {

o.update(stock);

}

}

}

public class MobileApp implements Observer {

public void update(String stock) {

System.out.println("Mobile App: Stock updated to " + stock);

}

}

public class TestObserver {

public static void main(String[] args) {

StockMarket market = new StockMarket();

Observer mobile = new MobileApp();

market.register(mobile);

market.setStock("Apple: 190 USD");

}

}

Output:-

Mobile App: Stock updated to Apple: 190 USD

**Exercise 8: Strategy Pattern**

public interface PaymentStrategy {

void pay(int amount);

}

public class CreditCardPayment implements PaymentStrategy {

public void pay(int amount) {

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

}

}

public class PayPalPayment implements PaymentStrategy {

public void pay(int amount) {

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

}

}

public class PaymentContext {

private PaymentStrategy strategy;

public void setPaymentStrategy(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.setPaymentStrategy(new CreditCardPayment());

context.pay(1000);

context.setPaymentStrategy(new PayPalPayment());

context.pay(1500);

}

}

Output:-

Paid 1000 using Credit Card.

Paid 1500 using PayPal.

**Exercise 9: Command Pattern**

public interface Command {

void execute();

}

public class Light {

public void turnOn() {

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

}

public void turnOff() {

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

}

}

public class LightOnCommand implements Command {

private Light light;

public LightOnCommand(Light light) {

this.light = light;

}

public void execute() {

light.turnOn();

}

}

public 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();

}

}

Output:-

Light is ON

**Exercise 10: MVC Pattern**

public class Student {

private String name;

private int id;

private String grade;

// Getters and Setters

public String getName() { return name; }

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

public int getId() { return id; }

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

public String getGrade() { return grade; }

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

}

public class StudentView {

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

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

}

}

public class StudentController {

private Student model;

private StudentView view;

public StudentController(Student model, StudentView view) {

this.model = model;

this.view = view;

}

public void updateView() {

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

}

}

public class TestMVC {

public static void main(String[] args) {

Student student = new Student();

student.setName("Tanuja");

student.setId(101);

student.setGrade("A");

StudentView view = new StudentView();

StudentController controller = new StudentController(student, view);

controller.updateView();

}

}

Output:

Student: Tanuja, ID: 101, Grade: A

**Exercise 11:Dependency Injection**

public interface CustomerRepository {

String findCustomerById(int id);

}

public class CustomerRepositoryImpl implements CustomerRepository {

public String findCustomerById(int id) {

return "Customer#" + id;

}

}

public class CustomerService {

private CustomerRepository repository;

public CustomerService(CustomerRepository repository) {

this.repository = repository;

}

public void printCustomer(int id) {

System.out.println("Found: " + repository.findCustomerById(id));

}

}

public class TestDI {

public static void main(String[] args) {

CustomerRepository repo = new CustomerRepositoryImpl();

CustomerService service = new CustomerService(repo);

service.printCustomer(42);

}

}

Output:-

Found: Customer#42