**Exercise 1: Implementing the Singleton Pattern**

**Scenario:**

You need to ensure that a logging utility class in your application has only one instance throughout the application lifecycle to ensure consistent logging.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **SingletonPatternExample**.
2. **Define a Singleton Class:**
   * Create a class named Logger that has a private static instance of itself.
   * Ensure the constructor of Logger is private.
   * Provide a public static method to get the instance of the Logger class.
3. **Implement the Singleton Pattern:**
   * Write code to ensure that the Logger class follows the Singleton design pattern.
4. **Test the Singleton Implementation:**
   * Create a test class to verify that only one instance of Logger is created and used across the application.

**Code**

public class Logger {

// Step 1: Private static instance of Logger (eager initialization or lazy — here lazy)

private static Logger instance;

// Step 2: Private constructor to prevent instantiation

private Logger() {

System.out.println("Logger Initialized");

}

// Step 3: Public static method to get the single instance

public static Logger getInstance() {

if (instance == null) {

instance = new Logger();

}

return instance;

}

// Sample log method

public void log(String message) {

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

}

}

public class Main {

public static void main(String[] args) {

// Trying to get multiple instances

Logger logger1 = Logger.getInstance();

Logger logger2 = Logger.getInstance();

// Logging messages

logger1.log("First log message.");

logger2.log("Second log message.");

// Verifying if both are the same instance

if (logger1 == logger2) {

System.out.println("Both logger instances are the same.");

} else {

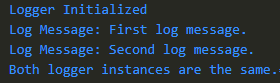
System.out.println("Different instances exist.");

}

}

}

**Output**



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

**Scenario:**

You are developing a document management system that needs to create different types of documents (e.g., Word, PDF, Excel). Use the Factory Method Pattern to achieve this.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **FactoryMethodPatternExample**.
2. **Define Document Classes:**
   * Create interfaces or abstract classes for different document types such as **WordDocument**, **PdfDocument**, and **ExcelDocument**.
3. **Create Concrete Document Classes:**
   * Implement concrete classes for each document type that implements or extends the above interfaces or abstract classes.
4. **Implement the Factory Method:**
   * Create an abstract class **DocumentFactory** with a method **createDocument()**.
   * Create concrete factory classes for each document type that extends DocumentFactory and implements the **createDocument()** method.
5. **Test the Factory Method Implementation:**
   * Create a test class to demonstrate the creation of different document types using the factory method.

Code

// Document interface

interface Document {

void open();

}

// Concrete Document Classes

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 Factory

abstract class DocumentFactory {

public abstract Document createDocument();

}

// Concrete Factories

class WordDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new WordDocument();

}

}

class PdfDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new PdfDocument();

}

}

class ExcelDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new ExcelDocument();

}

}

// Test Class

public class FactoryMethodPatternExample {

public static void main(String[] args) {

// Create Word Document

DocumentFactory wordFactory = new WordDocumentFactory();

Document wordDoc = wordFactory.createDocument();

wordDoc.open();

// Create PDF Document

DocumentFactory pdfFactory = new PdfDocumentFactory();

Document pdfDoc = pdfFactory.createDocument();

pdfDoc.open();

// Create Excel Document

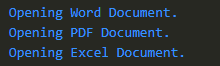
DocumentFactory excelFactory = new ExcelDocumentFactory();

Document excelDoc = excelFactory.createDocument();

excelDoc.open();

}

}



**Exercise 3: Implementing the Builder Pattern**

**Scenario:**

You are developing a system to create complex objects such as a Computer with multiple optional parts. Use the Builder Pattern to manage the construction process.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **BuilderPatternExample**.
2. **Define a Product Class:**
   * Create a class **Computer** with attributes like **CPU**, **RAM**, **Storage**, etc.
3. **Implement the Builder Class:**
   * Create a static nested Builder class inside Computer with methods to set each attribute.
   * Provide a **build()** method in the Builder class that returns an instance of Computer.
4. **Implement the Builder Pattern:**
   * Ensure that the **Computer** class has a private constructor that takes the **Builder** as a parameter.
5. **Test the Builder Implementation:**
   * Create a test class to demonstrate the creation of different configurations of Computer using the Builder pattern.

Code

// Product class

class Computer {

// Attributes

private String CPU;

private String RAM;

private String storage;

private boolean graphicsCard;

// Private constructor

private Computer(Builder builder) {

this.CPU = builder.CPU;

this.RAM = builder.RAM;

this.storage = builder.storage;

this.graphicsCard = builder.graphicsCard;

}

// Display method

public void displayConfig() {

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

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

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

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

System.out.println("Graphics Card: " + (graphicsCard ? "Yes" : "No"));

System.out.println();

}

// Static nested Builder class

public static class Builder {

private String CPU;

private String RAM;

private String storage;

private boolean graphicsCard;

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 Builder setGraphicsCard(boolean graphicsCard) {

this.graphicsCard = graphicsCard;

return this;

}

public Computer build() {

return new Computer(this);

}

}

}

// Test class

public class BuilderPatternExample {

public static void main(String[] args) {

// Building a high-end computer

Computer gamingPC = new Computer.Builder()

.setCPU("Intel i9")

.setRAM("32GB")

.setStorage("2TB SSD")

.setGraphicsCard(true)

.build();

gamingPC.displayConfig();

// Building a budget computer

Computer officePC = new Computer.Builder()

.setCPU("Intel i5")

.setRAM("16GB")

.setStorage("512GB SSD")

.setGraphicsCard(false)

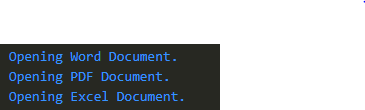
.build();

officePC.displayConfig();

}

}

Output



**Exercise 4: Implementing the Adapter Pattern**

**Scenario:**

You are developing a payment processing system that needs to integrate with multiple third-party payment gateways with different interfaces. Use the Adapter Pattern to achieve this.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **AdapterPatternExample**.
2. **Define Target Interface:**
   * Create an interface **PaymentProcessor** with methods like **processPayment()**.
3. **Implement Adaptee Classes:**
   * Create classes for different payment gateways with their own methods.
4. **Implement the Adapter Class:**
   * Create an adapter class for each payment gateway that implements PaymentProcessor and translates the calls to the gateway-specific methods.
5. **Test the Adapter Implementation:**
   * Create a test class to demonstrate the use of different payment gateways through the adapter.

Code

// Target Interface

interface PaymentProcessor {

void processPayment(double amount);

}

// Adaptee Classes (existing payment gateways)

class PayPalGateway {

public void sendPayment(double amount) {

System.out.println("Processing payment of ₹" + amount + " via PayPal.");

}

}

class StripeGateway {

public void makePayment(double amount) {

System.out.println("Processing payment of ₹" + amount + " via Stripe.");

}

}

// Adapter Classes

class PayPalAdapter implements PaymentProcessor {

private PayPalGateway payPal;

public PayPalAdapter(PayPalGateway payPal) {

this.payPal = payPal;

}

public void processPayment(double amount) {

payPal.sendPayment(amount);

}

}

class StripeAdapter implements PaymentProcessor {

private StripeGateway stripe;

public StripeAdapter(StripeGateway stripe) {

this.stripe = stripe;

}

public void processPayment(double amount) {

stripe.makePayment(amount);

}

}

// Test Class

public class AdapterPatternExample {

public static void main(String[] args) {

// Using PayPal via adapter

PaymentProcessor payPalPayment = new PayPalAdapter(new PayPalGateway());

payPalPayment.processPayment(1500);

// Using Stripe via adapter

PaymentProcessor stripePayment = new StripeAdapter(new StripeGateway());

stripePayment.processPayment(2750);

}

}

Output



**Exercise 5: Implementing the Decorator Pattern**

**Scenario:**

You are developing a notification system where notifications can be sent via multiple channels (e.g., Email, SMS). Use the Decorator Pattern to add functionalities dynamically.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **DecoratorPatternExample**.
2. **Define Component Interface:**
   * Create an interface **Notifier** with a method **send()**.
3. **Implement Concrete Component:**
   * Create a class **EmailNotifier** that implements Notifier.
4. **Implement Decorator Classes:**
   * Create abstract decorator class **NotifierDecorator** that implements **Notifier** and holds a reference to a **Notifier** object.
   * Create concrete decorator classes like **SMSNotifierDecorator**, **SlackNotifierDecorator** that extend **NotifierDecorator**.
5. **Test the Decorator Implementation:**
   * Create a test class to demonstrate sending notifications via multiple channels using decorators.

**Code**

// Component Interface

interface Notifier {

void send(String message);

}

// Concrete Component

class EmailNotifier implements Notifier {

public void send(String message) {

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

}

}

// Abstract Decorator

abstract class NotifierDecorator implements Notifier {

protected Notifier wrappedNotifier;

public NotifierDecorator(Notifier notifier) {

this.wrappedNotifier = notifier;

}

public void send(String message) {

wrappedNotifier.send(message);

}

}

// Concrete Decorators

class SMSNotifierDecorator extends NotifierDecorator {

public SMSNotifierDecorator(Notifier notifier) {

super(notifier);

}

public void send(String message) {

super.send(message);

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

}

}

class SlackNotifierDecorator extends NotifierDecorator {

public SlackNotifierDecorator(Notifier notifier) {

super(notifier);

}

public void send(String message) {

super.send(message);

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

}

}

// Test Class

public class DecoratorPatternExample {

public static void main(String[] args) {

// Basic Email notification

Notifier basicNotifier = new EmailNotifier();

// Add SMS notification

Notifier smsNotifier = new SMSNotifierDecorator(basicNotifier);

// Add Slack notification on top of SMS and Email

Notifier slackNotifier = new SlackNotifierDecorator(smsNotifier);

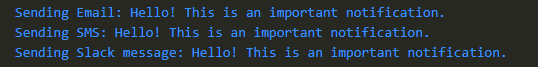
// Send notification via all channels

slackNotifier.send("Hello! This is an important notification.");

}

}

Output



**Exercise 6: Implementing the Proxy Pattern**

**Scenario:**

You are developing an image viewer application that loads images from a remote server. Use the Proxy Pattern to add lazy initialization and caching.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **ProxyPatternExample**.
2. **Define Subject Interface:**
   * Create an interface Image with a method **display()**.
3. **Implement Real Subject Class:**
   * Create a class **RealImage** that implements Image and loads an image from a remote server.
4. **Implement Proxy Class:**
   * Create a class **ProxyImage** that implements Image and holds a reference to RealImage.
   * Implement lazy initialization and caching in **ProxyImage**.
5. **Test the Proxy Implementation:**
   * Create a test class to demonstrate the use of **ProxyImage** to load and display images.

Code

// Subject Interface

interface Image {

void display();

}

// Real Subject Class

class RealImage implements Image {

private String fileName;

public RealImage(String fileName) {

this.fileName = fileName;

loadFromServer();

}

private void loadFromServer() {

System.out.println("Loading image from server: " + fileName);

}

public void display() {

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

}

}

// Proxy Class

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); // lazy initialization

}

realImage.display();

}

}

// Test Class

public class ProxyPatternExample {

public static void main(String[] args) {

Image image1 = new ProxyImage("pic1.png");

Image image2 = new ProxyImage("pic2.png");

// Image will be loaded only on first display call

image1.display(); // Loads and displays

image1.display(); // Only displays

System.out.println();

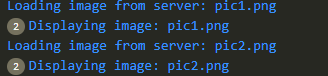
image2.display(); // Loads and displays

image2.display(); // Only displays

}

}

Output



**Exercise 7: Implementing the Observer Pattern**

**Scenario:**

You are developing a stock market monitoring application where multiple clients need to be notified whenever stock prices change. Use the Observer Pattern to achieve this.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **ObserverPatternExample**.
2. **Define Subject Interface:**
   * Create an interface **Stock** with methods to **register**, **deregister**, and **notify** observers.
3. **Implement Concrete Subject:**
   * Create a class **StockMarket** that implements **Stock** and maintains a list of observers.
4. **Define Observer Interface:**
   * Create an interface Observer with a method **update().**
5. **Implement Concrete Observers:**
   * Create classes **MobileApp**, **WebApp** that implement Observer.
6. **Test the Observer Implementation:**
   * Create a test class to demonstrate the registration and notification of observers.

Code

import java.util.\*;

// Observer Interface

interface Observer {

void update(String stockName, double price);

}

// Subject Interface

interface Stock {

void registerObserver(Observer o);

void removeObserver(Observer o);

void notifyObservers();

}

// Concrete Subject

class StockMarket implements Stock {

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

private String stockName;

private double stockPrice;

public void setStockPrice(String stockName, double price) {

this.stockName = stockName;

this.stockPrice = price;

notifyObservers();

}

public void registerObserver(Observer o) {

observers.add(o);

}

public void removeObserver(Observer o) {

observers.remove(o);

}

public void notifyObservers() {

for (Observer o : observers) {

o.update(stockName, stockPrice);

}

}

}

// Concrete Observers

class MobileApp implements Observer {

public void update(String stockName, double price) {

System.out.println("MobileApp - Stock: " + stockName + " | New Price: ₹" + price);

}

}

class WebApp implements Observer {

public void update(String stockName, double price) {

System.out.println("WebApp - Stock: " + stockName + " | New Price: ₹" + price);

}

}

// Test Class

public class ObserverPatternExample {

public static void main(String[] args) {

StockMarket stockMarket = new StockMarket();

Observer mobileApp = new MobileApp();

Observer webApp = new WebApp();

// Register observers

stockMarket.registerObserver(mobileApp);

stockMarket.registerObserver(webApp);

// Update stock prices

stockMarket.setStockPrice("TCS", 3550.75);

stockMarket.setStockPrice("Infosys", 1600.50);

// Remove one observer and update again

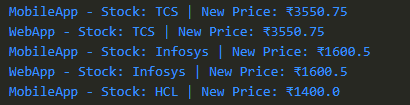
stockMarket.removeObserver(webApp);

stockMarket.setStockPrice("HCL", 1400.00);

}

}

Output



**Exercise 8: Implementing the Strategy Pattern**

**Scenario:**

You are developing a payment system where different payment methods (e.g., Credit Card, PayPal) can be selected at runtime. Use the Strategy Pattern to achieve this.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **StrategyPatternExample**.
2. **Define Strategy Interface:**
   * Create an interface PaymentStrategy with a method **pay()**.
3. **Implement Concrete Strategies:**
   * Create classes **CreditCardPayment**, **PayPalPayment** that implement **PaymentStrategy**.
4. **Implement Context Class:**
   * Create a class **PaymentContext** that holds a reference to **PaymentStrategy** and a method to execute the strategy.
5. **Test the Strategy Implementation:**
   * Create a test class to demonstrate selecting and using different payment strategies.

Code

// Strategy Interface

interface PaymentStrategy {

void pay(double amount);

}

// Concrete Strategies

class CreditCardPayment implements PaymentStrategy {

public void pay(double amount) {

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

}

}

class PayPalPayment implements PaymentStrategy {

public void pay(double amount) {

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

}

}

// Context Class

class PaymentContext {

private PaymentStrategy paymentStrategy;

// Inject strategy dynamically

public void setPaymentStrategy(PaymentStrategy paymentStrategy) {

this.paymentStrategy = paymentStrategy;

}

public void executePayment(double amount) {

if (paymentStrategy == null) {

System.out.println("No payment strategy selected.");

} else {

paymentStrategy.pay(amount);

}

}

}

// Test Class

public class StrategyPatternExample {

public static void main(String[] args) {

PaymentContext context = new PaymentContext();

// Pay with Credit Card

context.setPaymentStrategy(new CreditCardPayment());

context.executePayment(2000);

// Pay with PayPal

context.setPaymentStrategy(new PayPalPayment());

context.executePayment(3500);

}

}

Output



**Exercise 9: Implementing the Command Pattern**

**Scenario:** You are developing a home automation system where commands can be issued to turn devices on or off. Use the Command Pattern to achieve this.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **CommandPatternExample**.
2. **Define Command Interface:**
   * Create an interface Command with a method **execute()**.
3. **Implement Concrete Commands:**
   * Create classes **LightOnCommand**, **LightOffCommand** that implement Command.
4. **Implement Invoker Class:**
   * Create a class **RemoteControl** that holds a reference to a Command and a method to execute the command.
5. **Implement Receiver Class:**
   * Create a class **Light** with methods to turn on and off.
6. **Test the Command Implementation:**
   * Create a test class to demonstrate issuing commands using the **RemoteControl**.

Code

// Command Interface

interface Command {

void execute();

}

// Receiver Class

class Light {

public void turnOn() {

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

}

public void turnOff() {

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

}

}

// Concrete Command: Turn Light On

class LightOnCommand implements Command {

private Light light;

public LightOnCommand(Light light) {

this.light = light;

}

public void execute() {

light.turnOn();

}

}

// Concrete Command: Turn Light Off

class LightOffCommand implements Command {

private Light light;

public LightOffCommand(Light light) {

this.light = light;

}

public void execute() {

light.turnOff();

}

}

// Invoker Class

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.");

}

}

}

// Test Class

public class CommandPatternExample {

public static void main(String[] args) {

Light light = new Light();

// Commands

Command lightOn = new LightOnCommand(light);

Command lightOff = new LightOffCommand(light);

// Remote Control

RemoteControl remote = new RemoteControl();

// Turn light on

remote.setCommand(lightOn);

remote.pressButton();

// Turn light off

remote.setCommand(lightOff);

remote.pressButton();

}

}

Output



**Exercise 10: Implementing the MVC Pattern**

**Scenario:**

You are developing a simple web application for managing student records using the MVC pattern.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **MVCPatternExample**.
2. **Define Model Class:**
   * Create a class **Student** with attributes like **name, id, and grade**.
3. **Define View Class:**
   * Create a class **StudentView** with a method **displayStudentDetails()**.
4. **Define Controller Class:**
   * Create a class **StudentController** that handles the communication between the model and the view.
5. **Test the MVC Implementation:**
   * Create a main class to demonstrate creating a **Student**, updating its details using **StudentController**, and displaying them using **StudentView**.

**Code**

// Model Class

class Student {

private String name;

private int id;

private String grade;

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

this.name = name;

this.id = id;

this.grade = grade;

}

// Getters and setters

public String getName() {

return name;

}

public int getId() {

return id;

}

public String getGrade() {

return grade;

}

public void setName(String name) {

this.name = name;

}

public void setGrade(String grade) {

this.grade = grade;

}

}

// View Class

class StudentView {

public void displayStudentDetails(Student student) {

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

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

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

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

System.out.println();

}

}

// Controller Class

class StudentController {

private Student student;

private StudentView view;

public StudentController(Student student, StudentView view) {

this.student = student;

this.view = view;

}

public void updateView() {

view.displayStudentDetails(student);

}

public void setStudentName(String name) {

student.setName(name);

}

public void setStudentGrade(String grade) {

student.setGrade(grade);

}

}

// Test Class

public class MVCPatternExample {

public static void main(String[] args) {

// Create Model

Student student = new Student("Rohan Sharma", 101, "A");

// Create View

StudentView view = new StudentView();

// Create Controller

StudentController controller = new StudentController(student, view);

// Display initial details

controller.updateView();

// Update student data via controller

controller.setStudentName("Aarav Gupta");

controller.setStudentGrade("A+");

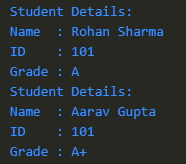
// Display updated details

controller.updateView();

}

}

**Output**



**Exercise 11: Implementing Dependency Injection**

**Scenario:**

You are developing a customer management application where the service class depends on a repository class. Use Dependency Injection to manage these dependencies.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **DependencyInjectionExample**.
2. **Define Repository Interface:**
   * Create an interface **CustomerRepository** with methods like **findCustomerById()**.
3. **Implement Concrete Repository:**
   * Create a class **CustomerRepositoryImpl** that implements **CustomerRepository**.
4. **Define Service Class:**
   * Create a class **CustomerService** that depends on **CustomerRepository**.
5. **Implement Dependency Injection:**
   * Use constructor injection to inject **CustomerRepository** into **CustomerService**.
6. **Test the Dependency Injection Implementation:**
   * Create a main class to demonstrate creating a **CustomerService** with **CustomerRepositoryImpl** and using it to find a customer.

Code

// Repository Interface

interface CustomerRepository {

String findCustomerById(int id);

}

// Concrete Repository Implementation

class CustomerRepositoryImpl implements CustomerRepository {

public String findCustomerById(int id) {

// Normally this would query a database — we’ll mock this

if (id == 1) {

return "Supreet Kamal";

} else if (id == 2) {

return "Aarav Gupta";

} else {

return "Customer Not Found";

}

}

}

// Service Class — depends on CustomerRepository

class CustomerService {

private CustomerRepository customerRepository;

// Constructor Injection

public CustomerService(CustomerRepository customerRepository) {

this.customerRepository = customerRepository;

}

public void displayCustomerName(int id) {

String customerName = customerRepository.findCustomerById(id);

System.out.println("Customer Name: " + customerName);

}

}

// Test Class

public class DependencyInjectionExample {

public static void main(String[] args) {

// Inject repository implementation into service

CustomerRepository repository = new CustomerRepositoryImpl();

CustomerService service = new CustomerService(repository);

// Use service to find and display customer names

service.displayCustomerName(1);

service.displayCustomerName(2);

service.displayCustomerName(3);

}

}

Output

