**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.

**CODE**:

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 SingletonPatternExample {

public static void main(String[] args) {

Logger logger1 = Logger.getInstance();

Logger logger2 = Logger.getInstance();

logger1.log("First log");

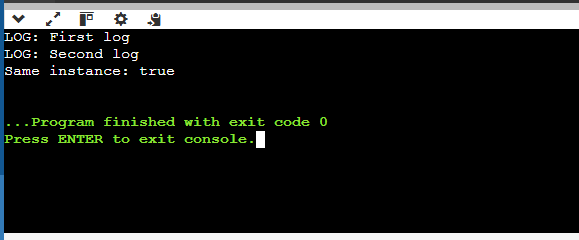
logger2.log("Second log");

System.out.println("Same instance: " + (logger1 == logger2));

}

}

**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.

**CODE:** **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 {

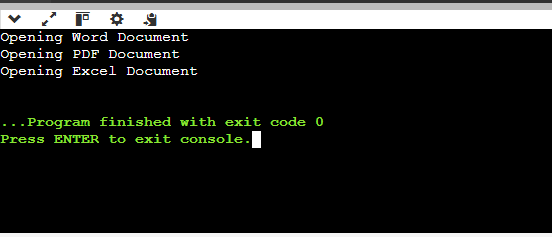
Document createDocument() {

return new PdfDocument();

}

}

**OUTPUT:**



**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.

**Code:**

class Computer {

String cpu;

String ram;

String storage;

private Computer(Builder builder) {

this.cpu = builder.cpu;

this.ram = builder.ram;

this.storage = builder.storage;

}

static class Builder {

String cpu;

String ram;

String storage;

Builder setCpu(String cpu) {

this.cpu = cpu;

return this;

}

Builder setRam(String ram) {

this.ram = ram;

return this;

}

Builder setStorage(String storage) {

this.storage = storage;

return this;

}

Computer build() {

return new Computer(this);

}

}

void showSpecs() {

System.out.println("CPU: " + cpu + ", RAM: " + ram + ", Storage: " + storage);

}

}

public class BuilderPatternExample {

public static void main(String[] args) {

Computer comp1 = new Computer.Builder().setCpu("i5").setRam("8GB").setStorage("512GB SSD").build();

comp1.showSpecs();

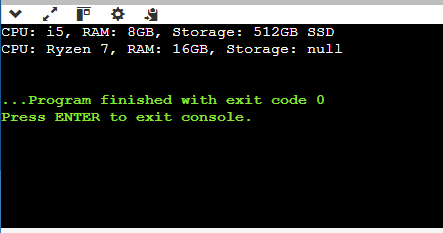
Computer comp2 = new Computer.Builder().setCpu("Ryzen 7").setRam("16GB").build();

comp2.showSpecs();

}

}

**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.

**CODE:**

interface PaymentProcessor {

void processPayment(double amount);

}

class PayPalGateway {

void sendMoney(double amount) {

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

}

}

class StripeGateway {

void makePayment(double amount) {

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

}

}

class PayPalAdapter implements PaymentProcessor {

private PayPalGateway gateway = new PayPalGateway();

public void processPayment(double amount) {

gateway.sendMoney(amount);

}

}

class StripeAdapter implements PaymentProcessor {

private StripeGateway gateway = new StripeGateway();

public void processPayment(double amount) {

gateway.makePayment(amount);

}

}

public class AdapterPatternExample {

public static void main(String[] args) {

PaymentProcessor paypal = new PayPalAdapter();

paypal.processPayment(100);

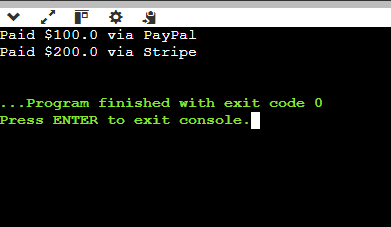
PaymentProcessor stripe = new StripeAdapter();

stripe.processPayment(200);

}

}

**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.

**CODE:**

interface Notifier {

void send();

}

class EmailNotifier implements Notifier {

public void send() {

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

}

}

abstract class NotifierDecorator implements Notifier {

protected Notifier notifier;

NotifierDecorator(Notifier notifier) {

this.notifier = notifier;

}

public void send() {

notifier.send();

}

}

class SMSNotifierDecorator extends NotifierDecorator {

SMSNotifierDecorator(Notifier notifier) {

super(notifier);

}

public void send() {

super.send();

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

}

}

class SlackNotifierDecorator extends NotifierDecorator {

SlackNotifierDecorator(Notifier notifier) {

super(notifier);

}

public void send() {

super.send();

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

}

}

public class DecoratorPatternExample {

public static void main(String[] args) {

Notifier notifier = new EmailNotifier();

Notifier smsNotifier = new SMSNotifierDecorator(notifier);

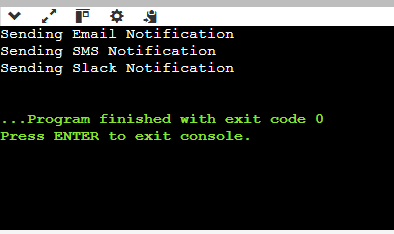
Notifier slackNotifier = new SlackNotifierDecorator(smsNotifier);

slackNotifier.send();

}

}

**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.

**CODE:**

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 image: " + filename);

}

public void display() {

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

}

}

class ProxyImage implements Image {

private String filename;

private RealImage realImage;

ProxyImage(String filename) {

this.filename = filename;

}

public void display() {

if (realImage == null) {

realImage = new RealImage(filename);

}

realImage.display();

}

}

public class ProxyPatternExample {

public static void main(String[] args) {

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

System.out.println("First call:");

img.display();

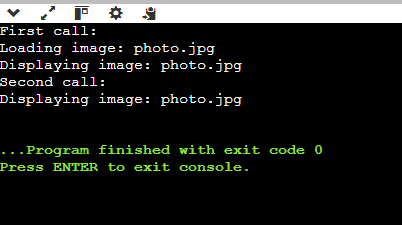
System.out.println("Second call:");

img.display();

}

}

**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.

**CODE:**

interface Observer {

void update(double price);

}

interface Stock {

void register(Observer o);

void deregister(Observer o);

void notifyObservers();

}

class StockMarket implements Stock {

private double price;

private Observer[] observers = new Observer[10];

private int count = 0;

public void register(Observer o) {

observers[count++] = o;

}

public void deregister(Observer o) {

for (int i = 0; i < count; i++) {

if (observers[i] == o) {

for (int j = i; j < count - 1; j++) {

observers[j] = observers[j + 1];

}

count--;

break;

}

}

}

public void notifyObservers() {

for (int i = 0; i < count; i++) {

observers[i].update(price);

}

}

public void setPrice(double price) {

this.price = price;

notifyObservers();

}

}

class MobileApp implements Observer {

public void update(double price) {

System.out.println("MobileApp: Stock price updated to $" + price);

}

}

class WebApp implements Observer {

public void update(double price) {

System.out.println("WebApp: Stock price updated to $" + price);

}

}

public class ObserverPatternExample {

public static void main(String[] args) {

StockMarket stock = new StockMarket();

Observer mobile = new MobileApp();

Observer web = new WebApp();

stock.register(mobile);

stock.register(web);

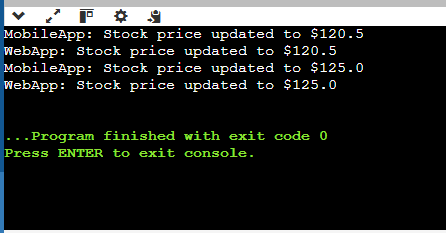
stock.setPrice(120.5);

stock.setPrice(125.0);

}

}

**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.

**CODE:**

interface PaymentStrategy {

void pay(double amount);

}

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");

}

}

class PaymentContext {

private PaymentStrategy strategy;

PaymentContext(PaymentStrategy strategy) {

this.strategy = strategy;

}

void executePayment(double amount) {

strategy.pay(amount);

}

}

public class StrategyPatternExample {

public static void main(String[] args) {

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

context.executePayment(150.0);

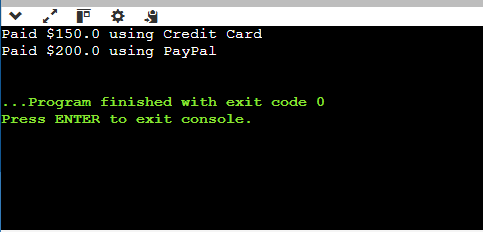
context = new PaymentContext(new PayPalPayment());

context.executePayment(200.0);

}

}

**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.

**CODE:**

interface Command {

void execute();

}

class Light {

void turnOn() {

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

}

void turnOff() {

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

}

}

class LightOnCommand implements Command {

private Light light;

LightOnCommand(Light light) {

this.light = light;

}

public void execute() {

light.turnOn();

}

}

class LightOffCommand implements Command {

private Light light;

LightOffCommand(Light light) {

this.light = light;

}

public void execute() {

light.turnOff();

}

}

class RemoteControl {

private Command command;

void setCommand(Command command) {

this.command = command;

}

void pressButton() {

command.execute();

}

}

public class CommandPatternExample {

public static void main(String[] args) {

Light light = new Light();

RemoteControl remote = new RemoteControl();

remote.setCommand(new LightOnCommand(light));

remote.pressButton();

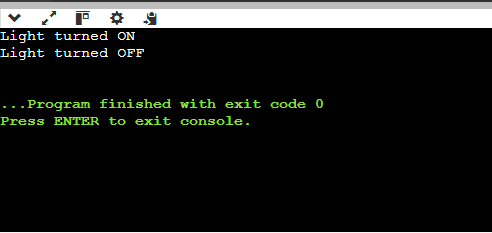
remote.setCommand(new LightOffCommand(light));

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.

**CODE:**

class Student {

String name;

int id;

String grade;

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

this.name = name;

this.id = id;

this.grade = grade;

}

}

class StudentView {

void displayStudentDetails(Student student) {

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

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

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

}

}

class StudentController {

private Student model;

private StudentView view;

StudentController(Student model, StudentView view) {

this.model = model;

this.view = view;

}

void setName(String name) {

model.name = name;

}

void setGrade(String grade) {

model.grade = grade;

}

void updateView() {

view.displayStudentDetails(model);

}

}

public class MVCPatternExample {

public static void main(String[] args) {

Student student = new Student("Alice", 101, "A");

StudentView view = new StudentView();

StudentController controller = new StudentController(student, view);

controller.updateView();

controller.setGrade("A+");

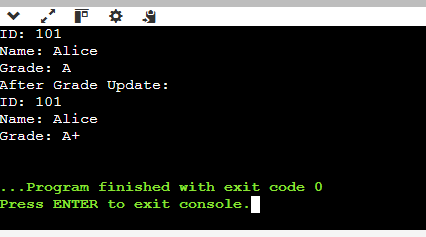
System.out.println("After Grade Update:");

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.

**CODE:**

interface CustomerRepository {

String findCustomerById(int id);

}

class CustomerRepositoryImpl implements CustomerRepository {

public String findCustomerById(int id) {

return "Customer#" + id + ": John Doe";

}

}

class CustomerService {

private CustomerRepository repository;

CustomerService(CustomerRepository repository) {

this.repository = repository;

}

void showCustomer(int id) {

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

}

}

public class DependencyInjectionExample {

public static void main(String[] args) {

CustomerRepository repo = new CustomerRepositoryImpl();

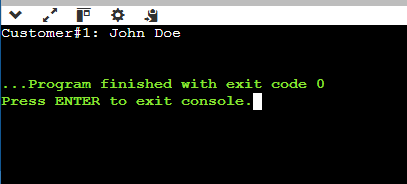
CustomerService service = new CustomerService(repo);

service.showCustomer(1);

}

}

**OUTPUT:**

****