**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 SingletonTest {

public static void main(String[] args) {

Logger logger1 = Logger.getInstance();

logger1.log("Hello, this is my first log!");

Logger logger2 = Logger.getInstance();

logger2.log("Hello again, this is my second log!");

// Checking if both are the same object

if (logger1 == logger2) {

System.out.println(" Only one Logger was created.");

} else {

System.out.println(" More than one Logger was created.");

}

}

}

// Singleton Logger class

class Logger {

private static Logger instance;

private Logger() {

}

// Method

public static Logger getInstance() {

if (instance == null) {

instance = new Logger();

}

return instance;

}

// Logging method

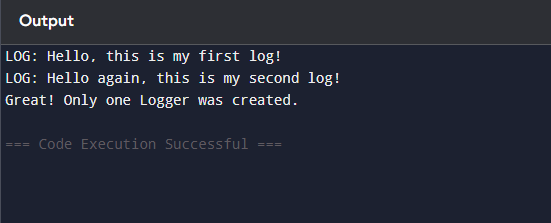
public void log(String message) {

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

}

}

Output Screenshot:



**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:**

public class DocumentFactoryExample {

// Step 1

interface Document {

void open();

}

// Step 2

static class WordDocument implements Document {

public void open() {

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

}

}

static class PdfDocument implements Document {

public void open() {

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

}

}

static class ExcelDocument implements Document {

public void open() {

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

}

}

// Step 3

static abstract class DocumentFactory {

public abstract Document createDocument();

}

// Step 4

static class WordDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new WordDocument();

}

}

static class PdfDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new PdfDocument();

}

}

static class ExcelDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new ExcelDocument();

}

}

// Step 5

public static void main(String[] args) {

DocumentFactory wordFactory = new WordDocumentFactory();

Document wordDoc = wordFactory.createDocument();

wordDoc.open();

DocumentFactory pdfFactory = new PdfDocumentFactory();

Document pdfDoc = pdfFactory.createDocument();

pdfDoc.open();

DocumentFactory excelFactory = new ExcelDocumentFactory();

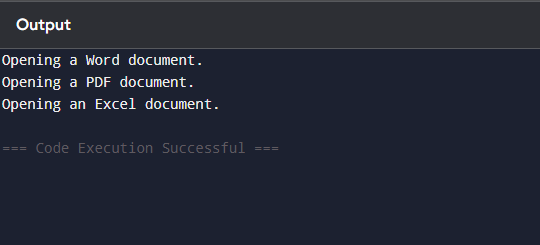
Document excelDoc = excelFactory.createDocument();

excelDoc.open();

}

}

Output Screenshot:



**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:**

public class BuilderPatternExample {

static class Computer {

private String cpu;

private String ram;

private String storage;

private String gpu;

private Computer(Builder builder) {

this.cpu = builder.cpu;

this.ram = builder.ram;

this.storage = builder.storage;

this.gpu = builder.gpu;

}

public void showSpecs() {

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

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

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

System.out.println("GPU: " + gpu);

System.out.println("-----------------------");

}

static class Builder {

private String cpu;

private String ram;

private String storage;

private String gpu;

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 setGPU(String gpu) {

this.gpu = gpu;

return this; }

public Computer build() {

return new Computer(this); }

}

}

public static void main(String[] args) {

Computer gamingPC = new Computer.Builder()

.setCPU("Intel i9")

.setRAM("32GB")

.setStorage("2TB SSD")

.setGPU("NVIDIA RTX 4090")

.build();

Computer officePC = new Computer.Builder()

.setCPU("Intel i5")

.setRAM("8GB")

.setStorage("512GB SSD")

.build(); // No GPU needed

System.out.println("Gaming PC Specs:");

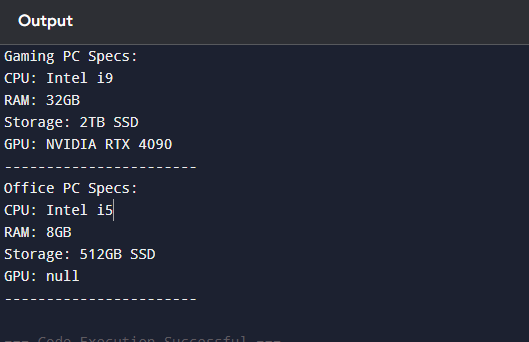
gamingPC.showSpecs();

System.out.println("Office PC Specs:");

officePC.showSpecs();

}

Output Screenshot:



**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:**

public class AdapterPatternExample {

// Step 2: Target Interface

interface PaymentProcessor {

void processPayment(double amount);

}

// Step 3: Adaptee Classes

static class PayPalGateway {

public void sendPayment(double amount) {

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

}

}

static class SupermoneyGateway {

public void makePayment(double amountInRupees) {

System.out.println("Paid " + amountInRupees + " using Supermoney.");

}

}

// Step 4: Adapter Classes

static class PayPalAdapter implements PaymentProcessor {

private PayPalGateway paypal;

public PayPalAdapter(PayPalGateway paypal) {

this.paypal = paypal;

}

public void processPayment(double amount) {

paypal.sendPayment(amount); }

}

static class SupermoneyAdapter implements PaymentProcessor {

private SupermoneyGateway supermoney;

public SupermoneyAdapter(SupermoneyGateway supermoney) {

this.supermoney = supermoney;

}

public void processPayment(double amount) {

supermoney.makePayment(amount);

}

}

// Step 5: Test the Adapters

public static void main(String[] args) {

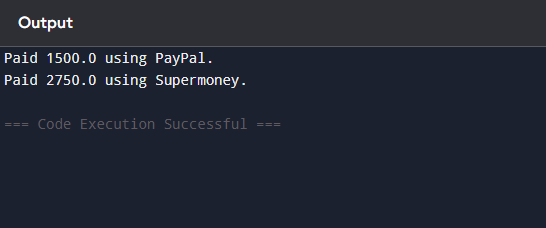
PaymentProcessor paypalProcessor = new PayPalAdapter(new PayPalGateway());

paypalProcessor.processPayment(1500);

PaymentProcessor supermoneyProcessor = new SupermoneyAdapter(new SupermoneyGateway());

supermoneyProcessor.processPayment(2750); }

Output Sreenshot:



**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:**

public class DecoratorPatternExample {

// Step 2: Component Interface

interface Notifier {

void send(String message);

}

// Step 3: Concrete Component

static class EmailNotifier implements Notifier {

public void send(String message) {

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

}

}

// Step 4.1: Base Decorator

static abstract class NotifierDecorator implements Notifier {

protected Notifier wrappedNotifier;

public NotifierDecorator(Notifier notifier) {

this.wrappedNotifier = notifier;

}

public void send(String message) {

wrappedNotifier.send(message); }

}

// Step 4.2: Concrete Decorators

static class SMSNotifierDecorator extends NotifierDecorator {

public SMSNotifierDecorator(Notifier notifier) {

super(notifier); }

public void send(String message) {

super.send(message);

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

}

static 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); }

}

// Step 5: Test Class

public static void main(String[] args) {

Notifier emailNotifier = new EmailNotifier();

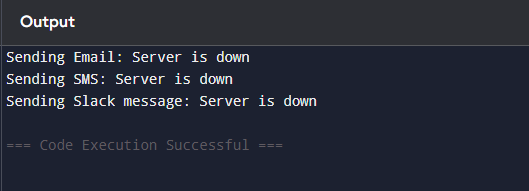
Notifier smsAndEmail = new SMSNotifierDecorator(emailNotifier);

Notifier multiChannel = new SlackNotifierDecorator(smsAndEmail);

multiChannel.send("Server is down");}

}

Output Screenshot:



**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:**

public class ProxyPatternExample {

interface Image {

void display();

}

static class RealImage implements Image {

private String filename;

public RealImage(String filename) {

this.filename = filename;

loadFromRemoteServer();

}

private void loadFromRemoteServer() {

System.out.println("Loading " + filename + " from remote server...");

}

public void display() {

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

}

}

static class ProxyImage implements Image {

private String filename;

private RealImage realImage;

public ProxyImage(String filename) {

this.filename = filename;

}

public void display() {

if (realImage == null) {

realImage = new RealImage(filename); // Lazy loading

} else {

System.out.println("(Cached) Image already loaded.");

}

realImage.display();

}

}

public static void main(String[] args) {

Image img1 = new ProxyImage("cute-puppy.jpg");

img1.display();

img1.display();

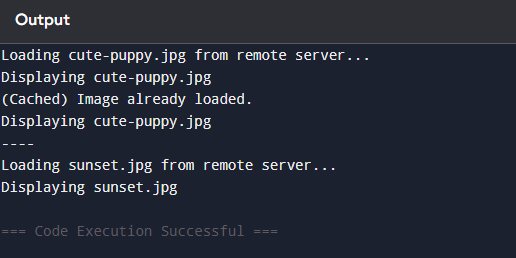
System.out.println("----");

Image img2 = new ProxyImage("sunset.jpg");

img2.display(); // Loads new image

}

**Output Screenshot:**

****

**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:**

public class ObserverPatternExample {

interface Observer {

void update(String stock, double price);

}

interface Stock {

void register(Observer o);

void deregister(Observer o);

void notifyObservers();

}

static class StockMarket implements Stock {

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

private String stockName;

private double price;

public StockMarket(String stockName) {

this.stockName = stockName;

}

public void register(Observer o) {

observers.add(o);

}

public void deregister(Observer o) {

observers.remove(o);

}

public void setPrice(double price) {

this.price = price;

notifyObservers();

}

public void notifyObservers() {

for (Observer o : observers) {

o.update(stockName, price);

}

}

}

static class MobileApp implements Observer {

public void update(String stock, double price) {

System.out.println("MobileApp: " + stock + " is now ₹" + price);

}

}

static class WebApp implements Observer {

public void update(String stock, double price) {

System.out.println("WebApp: " + stock + " is now ₹" + price);

}

}

public static void main(String[] args) {

StockMarket stock = new StockMarket("TCS");

Observer mobile = new MobileApp();

Observer web = new WebApp();

stock.register(mobile);

stock.register(web);

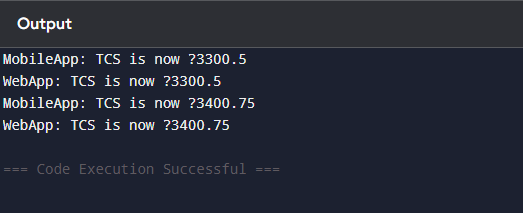
stock.setPrice(3300.50);

stock.setPrice(3400.75);

}

}

**Output Screenshot:**



**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:

public class StrategyPatternExample {

interface PaymentStrategy {

void pay(double amount);

}

static class CreditCardPayment implements PaymentStrategy {

public void pay(double amount) {

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

}

}

static class PayPalPayment implements PaymentStrategy {

public void pay(double amount) {

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

}

}

static class PaymentContext {

private PaymentStrategy strategy;

public void setStrategy(PaymentStrategy strategy) {

this.strategy = strategy;

}

public void pay(double amount) {

strategy.pay(amount);

}

}

public static void main(String[] args) {

PaymentContext context = new PaymentContext();

context.setStrategy(new CreditCardPayment());

context.pay(1200);

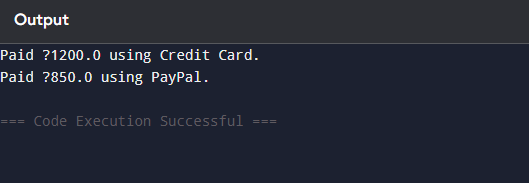
context.setStrategy(new PayPalPayment());

context.pay(850);

}

}

Output Screenshot:



**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:

public class CommandPatternExample {

interface Command {

void execute();

}

static class Light {

public void on() {

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

}

public void off() {

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

}

}

static class LightOnCommand implements Command {

private Light light;

public LightOnCommand(Light light) {

this.light = light;

}

public void execute() {

light.on();

}

}

static class LightOffCommand implements Command {

private Light light;

public LightOffCommand(Light light) {

this.light = light;

}

public void execute() {

light.off();

}

}

static class RemoteControl {

private Command command;

public void setCommand(Command command) {

this.command = command;

}

public void pressButton() {

command.execute();

}

}

public static void main(String[] args) {

Light light = new Light();

RemoteControl remote = new RemoteControl();

remote.setCommand(new LightOnCommand(light));

remote.pressButton();

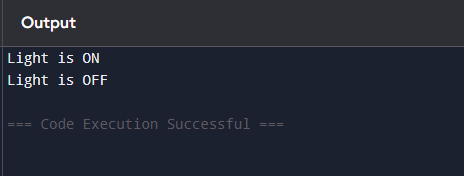
remote.setCommand(new LightOffCommand(light));

remote.pressButton();

}

}

Output Screenshot:



**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:**

public class MVCPatternExample {

static class Student {

private String name;

private String id;

private String grade;

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

this.name = name;

this.id = id;

this.grade = grade;

}

public String getName() { return name; }

public String getId() { return id; }

public String getGrade() { return grade; }

public void setGrade(String grade) {

this.grade = grade;

}

}

static class StudentView {

public void displayStudentDetails(Student student) {

System.out.println("Student: " + student.getName() + ", ID: " + student.getId() + ", Grade: " + student.getGrade());

}

}

static class StudentController {

private Student student;

private StudentView view;

public StudentController(Student student, StudentView view) {

this.student = student;

this.view = view;

}

public void updateGrade(String grade) {

student.setGrade(grade);

}

public void showStudent() {

view.displayStudentDetails(student);

}

}

public static void main(String[] args) {

Student student = new Student("Raju", "S101", "A");

StudentView view = new StudentView();

StudentController controller = new StudentController(student, view);

controller.showStudent();

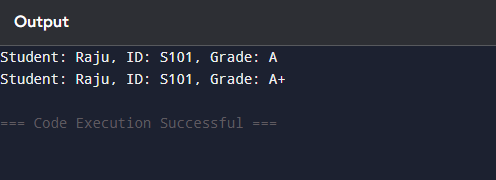
controller.updateGrade("A+");

controller.showStudent();

}

}

**Output Screenshot:**

****

**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:**

public class DependencyInjectionExample {

interface CustomerRepository {

String findCustomerById(String id);

}

static class CustomerRepositoryImpl implements CustomerRepository {

public String findCustomerById(String id) {

return "Customer[id=" + id + ", name=Rani]";

}

}

static class CustomerService {

private CustomerRepository repository;

public CustomerService(CustomerRepository repository) {

this.repository = repository;

}

public void showCustomer(String id) {

String customer = repository.findCustomerById(id);

System.out.println("Found: " + customer);

}

}

public static void main(String[] args) {

CustomerRepository repo = new CustomerRepositoryImpl();

CustomerService service = new CustomerService(repo);

service.showCustomer("C001");

}

}

**Output Screenshot:**

