**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.

**SOLUTION:**

Logger.java

package singleton;

public class Logger {

*//Private static instance of the same class*

    private static Logger instance;

*//Private constructor*

    private Logger() {

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

    }

*//Public static method to return the same instance every time*

public static Logger getInstance() {

        if (instance == null) {

            instance = new Logger();

        }

        return instance;

    }

*//logging method*

    public void log(String message) {

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

    }

}

TestSingleton.java

package singleton;

public class TestSingleton {

    public static void main(String[] args) {

*// Logger instances*

        Logger logger1 = Logger.getInstance();

        Logger logger2 = Logger.getInstance();

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

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

*// Check if both references point to the same object*

        if (logger1 == logger2) {

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

        } else {

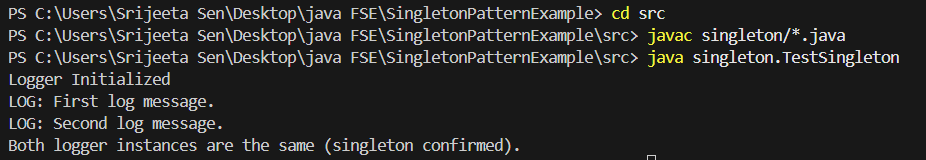
            System.out.println("Logger instances are different (singleton violated).");

        }

    }

}

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.

**SOLUTION:**

Document.java

package documents;

public interface Document {

void open();

}

WordDocument.java

package documents;

public class WordDocument implements Document {

@Override

public void open() {

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

}

}

PdfDocument.java

package documents;

public class PdfDocument implements Document {

@Override

public void open() {

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

}

}

ExcelDocument.java

package documents;

public class ExcelDocument implements Document {

@Override

public void open() {

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

}

}

WordDocumentFactory.java

package documents;

public abstract class DocumentFactory {

public abstract Document createDocument();

}

PdfDocumentFactory.java

package documents;

public class PdfDocumentFactory extends DocumentFactory {

@Override

public Document createDocument() {

return new PdfDocument();

}

}

ExcelDocumentFactory.java

package documents;

public class ExcelDocumentFactory extends DocumentFactory {

@Override

public Document createDocument() {

return new ExcelDocument();

}

}

TestFactoryMethod.java

package documents;

public class TestFactoryMethod {

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();

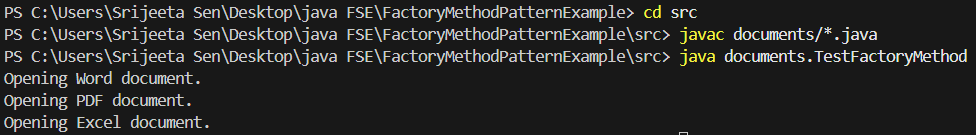
Document excelDoc = excelFactory.createDocument();

excelDoc.open();

}

}

Output:

****

**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.

**SOLUTION**

Computer.java

package builder;

public class Computer {

    private final String cpu;

    private final String ram;

    private final String storage;

    private final String graphicsCard;

*// Private constructor to enforce the builder*

    private Computer(Builder builder) {

        this.cpu = builder.cpu;

        this.ram = builder.ram;

        this.storage = builder.storage;

        this.graphicsCard = builder.graphicsCard;

    }

*// Static nested Builder class*

    public static class Builder {

        private final String cpu;

        private final String ram;

        private String storage;

        private String graphicsCard;

        public Builder(String cpu, String ram) {

            this.cpu = cpu;

            this.ram = ram;

        }

        public Builder setStorage(String storage) {

            this.storage = storage;

            return this;

        }

        public Builder setGraphicsCard(String graphicsCard) {

            this.graphicsCard = graphicsCard;

            return this;

        }

*// Build method to return the final product*

        public Computer build() {

            return new Computer(this);

        }

    }

*// Display method to show configuration*

    public void displayConfiguration() {

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

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

        System.out.println("Storage: " + (storage != null ? storage : "Not Included"));

        System.out.println("Graphics Card: " + (graphicsCard != null ? graphicsCard : "Not Included"));

    }

}

TestBuilderPattern.java

package builder;

public class TestBuilderPattern {

    public static void main(String[] args) {

*// Basic computer*

        Computer basicComputer = new Computer.Builder("Intel i3", "4GB").build();

        basicComputer.displayConfiguration();

*// High-end gaming computer*

        Computer gamingComputer = new Computer.Builder("Intel i9", "32GB")

                .setStorage("1TB SSD")

                .setGraphicsCard("NVIDIA RTX 4090")

                .build();

        gamingComputer.displayConfiguration();

*// Office computer with moderate specifications*

        Computer officeComputer = new Computer.Builder("AMD Ryzen 5", "16GB")

                .setStorage("512GB SSD")

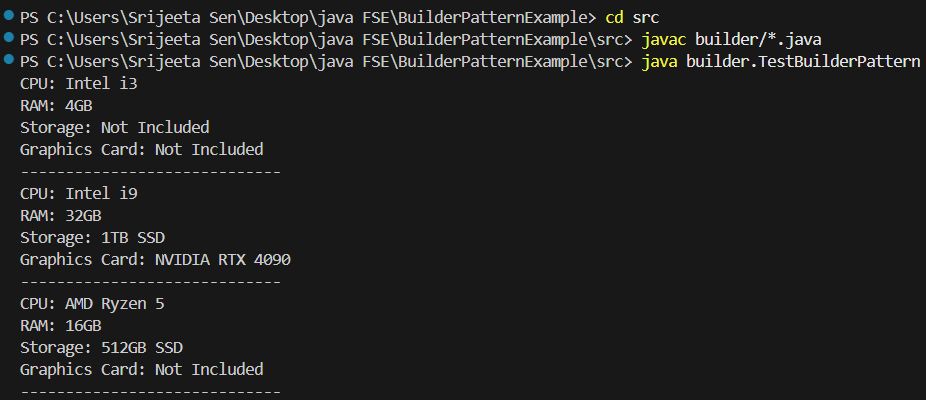
                .build();

        officeComputer.displayConfiguration();

    }

}

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.

**SOLUTION**

PaymentProcessor.java

package adapter;

public interface PaymentProcessor {

    void processPayment(double amount);

}

PayPalGateway.java

package adapter;

public class PayPalGateway {

    public void makePayment(double amountInDollars) {

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

    }

}

StripeGateway.java

package adapter;

public class StripeGateway {

     public void payAmount(double amt) {

        System.out.println("Paid $" + amt + " using Stripe.");

    }

}

PayPalAdapter.java

package adapter;

public class PayPalAdapter implements PaymentProcessor {

    private PayPalGateway paypal;

    public PayPalAdapter() {

        this.paypal = new PayPalGateway();

    }

    @Override

    public void processPayment(double amount) {

        paypal.makePayment(amount);

    }

}

StripeAdapter.java

package adapter;

public class StripeAdapter implements PaymentProcessor {

    private StripeGateway stripe;

    public StripeAdapter() {

        this.stripe = new StripeGateway();

    }

    @Override

    public void processPayment(double amount) {

        stripe.payAmount(amount);

    }

}

TestAdapterPattern.java

package adapter;

public class TestAdapterPattern {

    public static void main(String[] args) {

        PaymentProcessor paypalProcessor = new PayPalAdapter();

        paypalProcessor.processPayment(250.0);

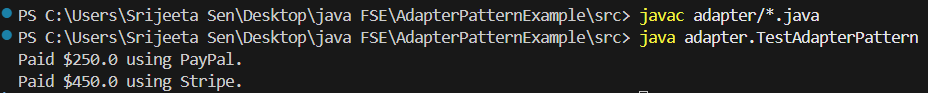
        PaymentProcessor stripeProcessor = new StripeAdapter();

        stripeProcessor.processPayment(450.0);

    }

}

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.

**SOLUTION**

Notifier.java

package decorator;

public interface Notifier {

    void send(String message);

}

EmailNotifier.java

package decorator;

public class EmailNotifier implements Notifier {

    @Override

    public void send(String message) {

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

    }

}

NotifierDecorator.java

package decorator;

public abstract class NotifierDecorator implements Notifier {

    protected Notifier wrappedNotifier;

    public NotifierDecorator(Notifier not) {

        this.wrappedNotifier = not;

    }

    public void send(String message) {

        wrappedNotifier.send(message);

    }

}

SMSNotifierDecorator.java

package decorator;

public class SMSNotifierDecorator extends NotifierDecorator {

    public SMSNotifierDecorator(Notifier not) {

        super(not);

    }

    @Override

    public void send(String message) {

        super.send(message);

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

    }

}

SlackNotifierDecorator.java

package decorator;

public class SlackNotifierDecorator extends NotifierDecorator {

    public SlackNotifierDecorator(Notifier notifier) {

        super(notifier);

    }

    @Override

    public void send(String message) {

        super.send(message);

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

    }

}

TestDecoratorPattern.java

package decorator;

public class TestDecoratorPattern {

    public static void main(String[] args) {

*// Base Notifier*

        Notifier not = new EmailNotifier();

*// Wrap with SMS decorator*

        not = new SMSNotifierDecorator(notifier);

*// Wrap with Slack decorator*

        not = new SlackNotifierDecorator(notifier);

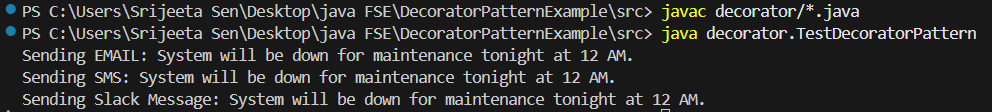
*// Send notification*

        not.send("System will be down for maintenance tonight at 12 AM.");

    }

}

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.

**SOLUTION**

Image.java

package proxy;

public interface Image {

    void display();

}

RealImage.java

package proxy;

public class RealImage implements Image {

    private String filename;

    public RealImage(String filename) {

        this.filename = filename;

        loadFromRemoteServer();

    }

    private void loadFromRemoteServer() {

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

    }

    @Override

    public void display() {

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

    }

}

ProxyImage.java

package proxy;

public class ProxyImage implements Image {

    private RealImage realImage;

    private String filename;

    public ProxyImage(String filename) {

        this.filename = filename;

    }

    @Override

    public void display() {

        if (realImage == null) {

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

        }

        realImage.display();

    }

}

TestProxyPattern.java

package proxy;

public class TestProxyPattern {

    public static void main(String[] args) {

        Image img1 = new ProxyImage("nature1.jpg");

        Image img2 = new ProxyImage("nature2.jpg");

*// Image is loaded only on first display*

        img1.display();  *// Loads + displays*

        img1.display();  *// Only displays*

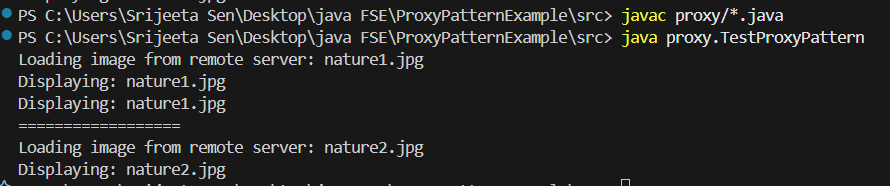
        System.out.println("==================");

        img2.display();  *// Loads + 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.

**SOLUTION**

Stock.java

package observer;

public interface Stock {

    void registerObserver(Observer o);

    void removeObserver(Observer o);

    void notifyObservers();

}

Observer.java

package observer;

public interface Observer {

    void update(String stockName, double price);

}

StockMarket.java

package observer;

import java.util.ArrayList;

import java.util.List;

public class StockMarket implements Stock {

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

    private String stockName;

    private double stockPrice;

    @Override

    public void registerObserver(Observer o) {

        ob.add(o);

    }

    @Override

    public void removeObserver(Observer o) {

        ob.remove(o);

    }

    @Override

    public void notifyObservers() {

        for (Observer obs : ob) {

            obs.update(stockName, stockPrice);

        }

    }

    public void setStockPrice(String name, double price) {

        this.stockName = name;

        this.stockPrice = price;

        notifyObservers();

    }

}

MobileApp.java

package observer;

public class MobileApp implements Observer {

    private String appName;

    public MobileApp(String appName) {

        this.appName = appName;

    }

    @Override

    public void update(String stockName, double price) {

        System.out.println(appName + " - Stock Update: " + stockName + " is now $" + price);

    }

}

WebApp.java

package observer;

public class WebApp implements Observer {

    private String siteName;

    public WebApp(String siteName) {

        this.siteName = siteName;

    }

    @Override

    public void update(String stockName, double price) {

        System.out.println(siteName + " - Stock Update: " + stockName + " is now $" + price);

    }

}

TestObserverPattern.java

package observer;

public class TestObserverPattern {

    public static void main(String[] args) {

        StockMarket st = new StockMarket();

        Observer mobile1 = new MobileApp("MobileApp1");

        Observer web1 = new WebApp("WebApp1");

        st.registerObserver(mobile1);

        st.registerObserver(web1);

        st.setStockPrice("AAPL", 200.00);

        System.out.println();

        st.setStockPrice("GOOGL", 2500.00);

        System.out.println();

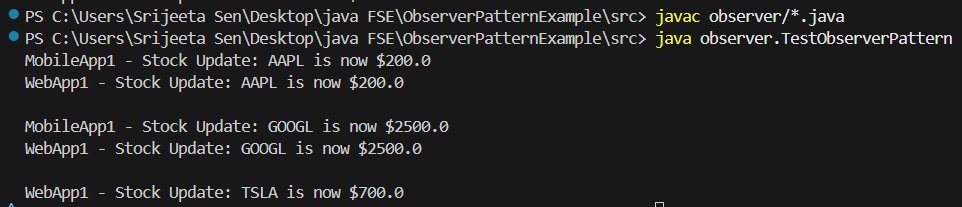
        st.removeObserver(mobile1);

        st.setStockPrice("TSLA", 700.00);

    }

}

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.

**SOLUTION**

PaymentStrategy.java

package strategy;

public interface PaymentStrategy {

    void pay(double amount);

}

CreditCardPayment.java

package strategy;

public class CreditCardPayment implements PaymentStrategy {

    private String cardNumber;

    private String cardHolderName;

    public CreditCardPayment(String cardNumber, String cardHolderName) {

        this.cardNumber = cardNumber;

        this.cardHolderName = cardHolderName;

    }

    @Override

    public void pay(double amount) {

        System.out.println("Paid $" + amount + " using Credit Card (CardHolder: " + cardHolderName + ")");

    }

}

PayPalPayment.java

package strategy;

public class PayPalPayment implements PaymentStrategy {

    private String email;

    public PayPalPayment(String email) {

        this.email = email;

    }

    @Override

    public void pay(double amount) {

        System.out.println("Paid $" + amount + " using PayPal (Email: " + email + ")");

    }

}

PaymentContext.java

package strategy;

public class PaymentContext {

    private PaymentStrategy paymentStrategy;

    public void setPaymentStrategy(PaymentStrategy paymentStrategy) {

        this.paymentStrategy = paymentStrategy;

    }

    public void processPayment(double amount) {

        if (paymentStrategy == null) {

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

        } else {

            paymentStrategy.pay(amount);

        }

    }

}

TestStrategyPattern.java

package strategy;

public class TestStrategyPattern {

    public static void main(String[] args) {

        PaymentContext context = new PaymentContext();

*// Use PayPal*

        context.setPaymentStrategy(new PayPalPayment("srijeetasen@gmail.com"));

        context.processPayment(250.00);

        System.out.println("-----");

*// Use Credit Card*

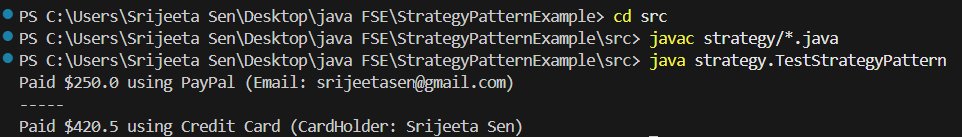
        context.setPaymentStrategy(new CreditCardPayment("1234-5678", "Srijeeta Sen"));

        context.processPayment(420.50);

    }

}

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**.

**SOLUTION**

Command.java

package command;

public interface Command {

    void execute();

}

Light.java

package command;

public class Light {

    public void turnOn() {

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

    }

    public void turnOff() {

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

    }

}

LightOn.java

package command;

public class LightOn implements Command {

    private Light light;

    public LightOn (Light light) {

        this.light = light;

    }

    @Override

    public void execute() {

        light.turnOn();

    }

}

LightOff.java

package command;

public class LightOff implements Command {

    private Light light;

    public LightOff (Light light) {

        this.light = light;

    }

    @Override

    public void execute() {

        light.turnOff();

    }

}

RemoteControl.java

package command;

public 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 set.");

        }

    }

}

TestCommandPattern.java

package command;

public class TestCommandPattern {

    public static void main(String[] args) {

        Light livingRoomLight = new Light();

        Command lightOn = new LightOn (livingRoomLight);

        Command lightOff = new LightOff (livingRoomLight);

        RemoteControl remote = new RemoteControl();

        remote.setCommand(lightOn);

        remote.pressButton();

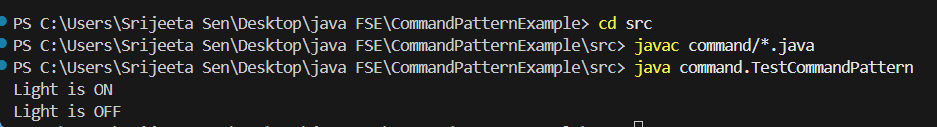
        remote.setCommand(lightOff);

        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**.

**SOLUTION**

Student.java

package mvc;

public 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;

    }

*// Getters and Setters*

    public String getName() { return name; }

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

    public String getId() { return id; }

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

    public String getGrade() { return grade; }

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

}

StudentView.java

package mvc;

public class StudentView {

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

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

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

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

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

    }

}

StudentController.java

package mvc;

public 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 setStudentId(String id) {

        model.setId(id);

    }

    public void setStudentGrade(String grade) {

        model.setGrade(grade);

    }

    public String getStudentName() {

        return model.getName();

    }

    public String getStudentId() {

        return model.getId();

    }

    public String getStudentGrade() {

        return model.getGrade();

    }

    public void updateView() {

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

    }

}

MVCTest.java

package mvc;

public class MVCTest {

    public static void main(String[] args) {

*// Create the model*

        Student student = new Student("Emily", "S1", "A");

*// Create the view*

        StudentView view = new StudentView();

*// Create controller*

        StudentController controller = new StudentController(student, view);

*// Displaying initial details*

        controller.updateView();

*// Updating model data via controller*

        controller.setStudentName("Chris");

        controller.setStudentGrade("B");

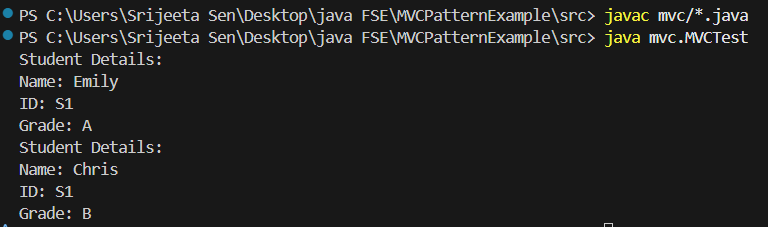
*// Displaying updated details*

        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.