**Design Patterns Solutions**

**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**

public class Logger {

    private static Logger l;

    private Logger() {

    }

    public static Logger getInstance() {

        if (l == null) {

            l = new Logger();

        }

        return l;

    }

    public void log(String message) {

        System.out.println(message);

    }

}

**Loggertest.java**

public class LoggerTest {

    public static void main(String[] args) {

        Logger l1 = Logger.getInstance();

        Logger l2 = Logger.getInstance();

        if (l1 == l2) {

            System.out.println("Logger is a singleton. Both references point to the same instance.");

        } else {

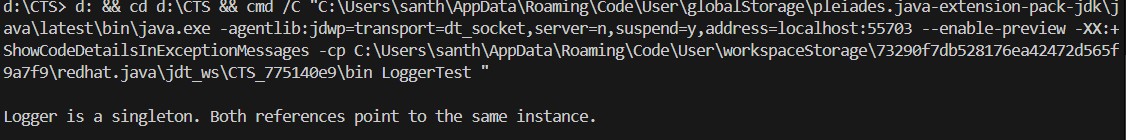
            System.out.println("Logger is not a singleton. Different instances were created.");

        }

    }

}

**Output**



**SingletonPatternExample.java**

public class SingletonPatternExample {

    public static void main(String[] args) {

        Logger l1 = Logger.getInstance();

        Logger l2 = Logger.getInstance();

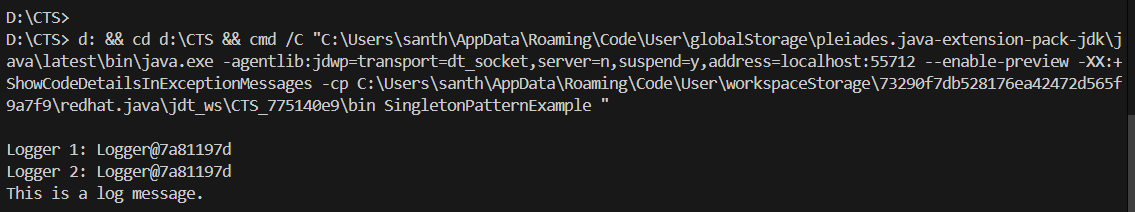
        System.out.println("Logger 1: " + l1);

        System.out.println("Logger 2: " + l2);

        l1.log("This is a log message.");

    }

}

**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**

public interface Document {

    String getContent();

    String getType();

}

**ExcelDocument.java**

public class ExcelDocument implements Document {

    @Override

    public String getContent() {

        return "This is an Excel document.";

    }

    @Override

    public String getType() {

        return "Excel";

    }

}

**PdfDocument.java**

public class PdfDocument implements Document {

    @Override

    public String getContent() {

        return "This is the content of a PDF document.";

    }

    @Override

    public String getType() {

        return "PDF Document";

    }

}

**WordDocument.java**

public class WordDocument implements Document {

    @Override

    public String getContent() {

        return "This is the content of a Word document.";

    }

    @Override

    public String getType() {

        return "Word Document";

    }

}

**DocumentFactory.java**

public abstract class DocumentFactory {

    public abstract Document createDocument();

}

**ExcelDocumentFactory.java**

public class ExcelDocumentFactory extends DocumentFactory {

    @Override

    public Document createDocument() {

        return new ExcelDocument();

    }

}

**PdfDocumentFactory.java**

public class PdfDocumentFactory extends DocumentFactory {

    @Override

    public Document createDocument() {

        return new PdfDocument();

    }

}

**WordDocumentFactory.java**

public class WordDocumentFactory extends DocumentFactory {

    @Override

    public Document createDocument() {

        return new WordDocument();

    }

}

**FactoryMethodPatternExample.java**

public class FactoryMethodPatternExample {

    public static void main(String[] args) {

        DocumentFactory wordFactory = new WordDocumentFactory();

        Document wordDoc = wordFactory.createDocument();

        System.out.println("Created: " + wordDoc.getType() + " with content: " + wordDoc.getContent());

        DocumentFactory pdfFactory = new PdfDocumentFactory();

        Document pdfDoc = pdfFactory.createDocument();

        System.out.println("Created: " + pdfDoc.getType() + " with content: " + pdfDoc.getContent());

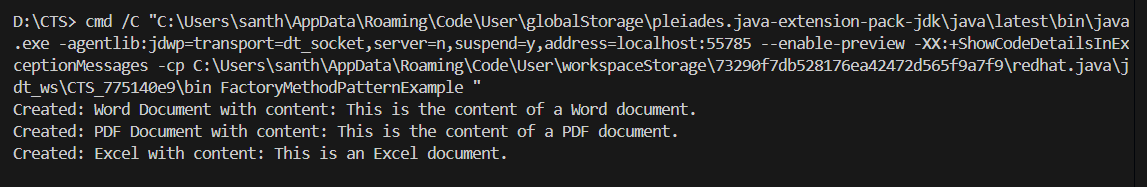
        DocumentFactory excelFactory = new ExcelDocumentFactory();

        Document excelDoc = excelFactory.createDocument();

        System.out.println("Created: " + excelDoc.getType() + " with content: " + excelDoc.getContent());

    }

}

**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**

public class Computer {

    private String cpu;

    private String ram;

    private String storage;

    private Computer(Builder b) {

        this.cpu = b.cpu;

        this.ram = b.ram;

        this.storage = b.storage;

    }

    public static class Builder {

        private String cpu;

        private String ram;

        private String storage;

        public Builder setCpu(String cpu) {

            this.cpu = cpu;

            return this;

        }

        public Builder setRam(String ram) {

            this.ram = ram;

            return this;

        }

        public Builder setStorage(String storage) {

            this.storage = storage;

            return this;

        }

        public Computer build() {

            return new Computer(this);

        }

    }

    public void display() {

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

    }

}

**Test.java**

public class Test {

    public static void main(String[] args) {

        Computer c1 = new Computer.Builder()

                .setCpu("Intel i5")

                .setRam("8GB")

                .setStorage("256GB SSD")

                .build();

        Computer c2 = new Computer.Builder()

                .setCpu("AMD Ryzen")

                .setRam("16GB")

                .build();

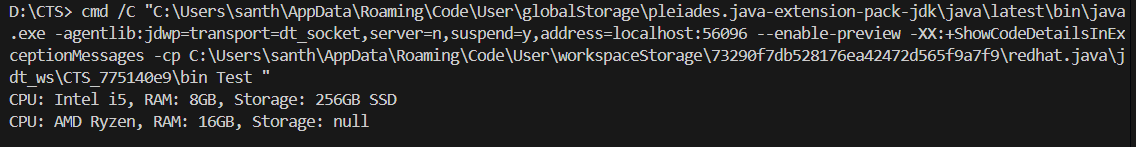
        c1.display();

        c2.display();

    }

}

**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**

public interface PaymentProcessor {

    void processPayment(double amount);

}

**PayPalGateway.java**

public class PayPalGateway {

    public void payWithPayPal(double amount) {

        System.out.println("Processing payment of $" + amount + " through PayPal.");

    }

}

**StripeGateway.java**

public class StripeGateway {

    public void payWithStripe(double amount) {

        System.out.println("Processing payment of $" + amount + " through Stripe.");

    }

}

**PayPalAdapter.java**

public class PayPalAdapter implements PaymentProcessor {

    private PayPalGateway payPalGateway;

    public PayPalAdapter(PayPalGateway payPalGateway) {

        this.payPalGateway = payPalGateway;

    }

    @Override

    public void processPayment(double amount) {

        payPalGateway.payWithPayPal(amount);

    }

}

**StripeAdapter.java**

public class StripeAdapter implements PaymentProcessor {

    private StripeGateway stripeGateway;

    public StripeAdapter() {

        this.stripeGateway = new StripeGateway();

    }

    public StripeAdapter(StripeGateway stripeGateway) {

        this.stripeGateway = stripeGateway;

    }

    @Override

    public void processPayment(double amount) {

        stripeGateway.payWithStripe(amount);

    }

}

**Test.java**

public class AdapterPatternTest {

    public static void main(String[] args) {

        PayPalGateway pg = new PayPalGateway();

        StripeGateway sg = new StripeGateway();

        PaymentProcessor pa = new PayPalAdapter(pg);

        PaymentProcessor sa = new StripeAdapter(sg);

        System.out.println("Processing payment with PayPal:");

        pa.processPayment(100.0);

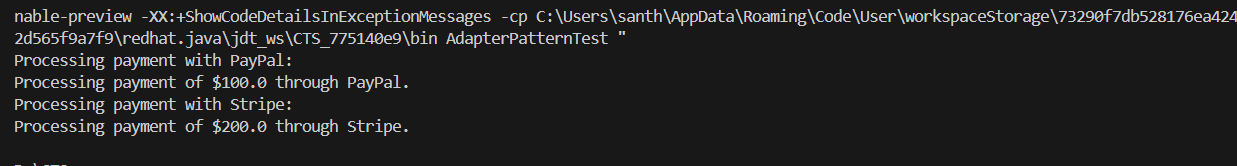
        System.out.println("Processing payment with Stripe:");

        sa.processPayment(200.0);

    }

}

**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**

public interface Notifier {

    void send(String message);

}

**EmailNotificationDecorator.java**

public class EmailNotificationDecorator implements Notifier {

    protected Notifier notifier;

    public EmailNotificationDecorator(Notifier notifier) {

        this.notifier = notifier;

    }

    @Override

    public void send(String message) {

        notifier.send(message);

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

    }

}

**NotificationDecorator.java**

public abstract class NotificationDecorator implements Notifier {

    protected Notifier decoratedNotifier;

    public NotificationDecorator(Notifier decoratedNotifier) {

        this.decoratedNotifier = decoratedNotifier;

    }

    @Override

    public void send(String msg) {

        decoratedNotifier.send(msg);

    }

}

**SMSNotificationDecorator.java**

public class SMSNotificationDecorator extends NotificationDecorator {

    public SMSNotificationDecorator(Notifier notifier) {

        super(notifier);

    }

   private void sendSMS(String message) {

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

    }

}

**BasicNotification.java**

public class BasicNotification implements Notifier {

    @Override

    public void send(String message) {

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

    }

}

**Test.java**

public class Test {

    public static void main(String[] args) {

        Notifier basicNotification = new BasicNotification();

        Notifier emailNotification = new EmailNotificationDecorator(basicNotification);

        Notifier smsNotification = new SMSNotificationDecorator(basicNotification);

        System.out.println("Sending Basic Notification:");

        basicNotification.send("hello");

        System.out.println("\n Sending Email Notification:");

        emailNotification.send("hello1");

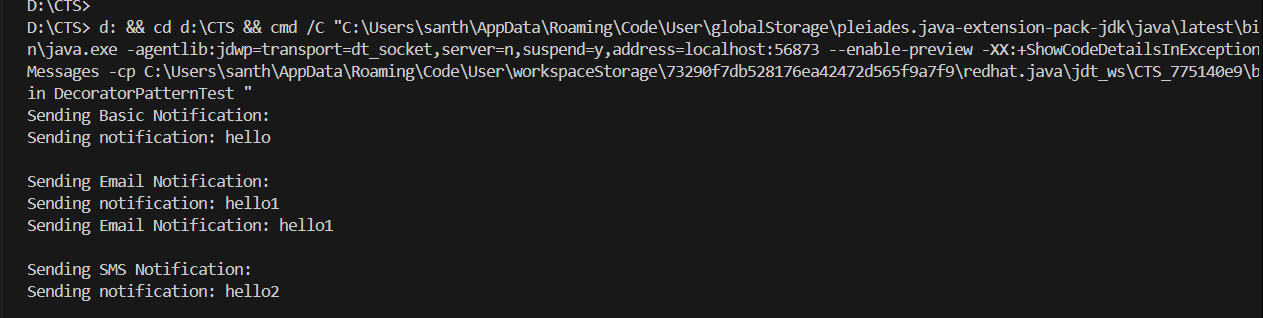
        System.out.println("\nSending SMS Notification:");

        smsNotification.send("hello2");

    }

}

**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**

public interface Image {

    void display();

}

**RealImage.java**

public class RealImage implements Image {

    private String filename;

    public RealImage(String filename) {

        this.filename = filename;

        loadImageFromDisk();

    }

    private void loadImageFromDisk() {

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

    }

    @Override

    public void display() {

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

    }

}

**ProxyImage.java**

public class ProxyImage implements Image {

    private RealImage realImage;

    private String imageName;

    public ProxyImage(String imageName) {

        this.imageName = imageName;

    }

    @Override

    public void display() {

        if (realImage == null) {

            realImage = new RealImage(imageName);

        }

        realImage.display();

    }

}

**ProxyPatternTest.java**

public class ProxyPatternTest {

    public static void main(String[] args) {

        Image image1 = new ProxyImage("image1.jpg");

        Image image2 = new ProxyImage("image2.jpg");

        image1.display();

        System.out.println();

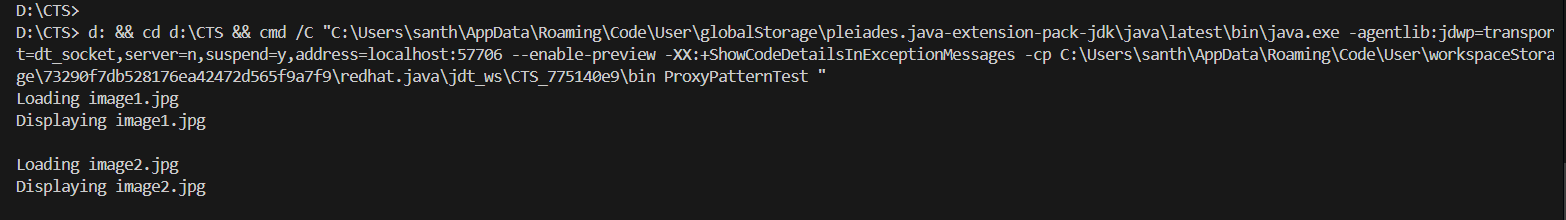
        image2.display();

        System.out.println();

    }

}

**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**

**Observer.java**

package ObserverPatternExample;

public interface Observer {

    void update(int price);

}

**Stock.java**

package ObserverPatternExample;

public interface Stock {

    void register(Observer o);

    void deregister(Observer o);

    void notifyObservers();

}

**StockMarket.java**

package ObserverPatternExample;

import java.util.ArrayList;

import java.util.List;

public class StockMarket implements Stock {

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

    private int stockPrice;

    @Override

    public void register(Observer o) {

        observers.add(o);

    }

    @Override

    public void deregister(Observer o) {

        observers.remove(o);

    }

    @Override

    public void notifyObservers() {

        for (Observer o : observers) {

            o.update(stockPrice);

        }

    }

    public void setPrice(int price) {

        this.stockPrice = price;

        notifyObservers();

    }

}

**MobileApp.java**

package ObserverPatternExample;

public class MobileApp implements Observer {

    @Override

    public void update(int price) {

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

    }

}

**Test.java**

package ObserverPatternExample;

public class Main {

    public static void main(String[] args) {

        StockMarket stockMarket = new StockMarket();

        Observer mobile = new MobileApp();

        Observer web = new WebApp();

        stockMarket.register(mobile);

        stockMarket.register(web);

        stockMarket.setPrice(100);

        stockMarket.setPrice(150);

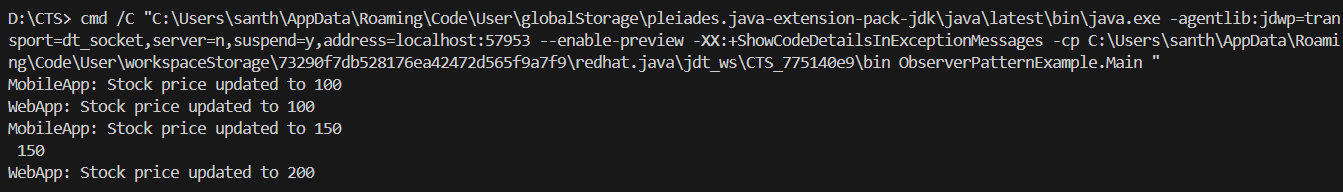
        stockMarket.deregister(mobile);

        stockMarket.setPrice(200);

    }

}

**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 StrategyPattern;

public interface PaymentStrategy {

    void pay(double amount);

}

**CreditCardPayment.java**

package StrategyPattern;

public class CreditCardPayment implements PaymentStrategy {

    public void pay(double amount) {

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

    }

}

**PayPalPayment.java**

package StrategyPattern;

public class PayPalPayment implements PaymentStrategy {

    public void pay(double amount) {

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

    }

}

**PaymentContext.java**

package StrategyPattern;

public class PaymentContext {

    private PaymentStrategy strategy;

    public void setStrategy(PaymentStrategy strategy) {

        this.strategy = strategy;

    }

    public void executePayment(double amount) {

        strategy.pay(amount);

    }

}

**Test.java**

package StrategyPattern;

public class Test {

    public static void main(String[] args) {

        PaymentContext context = new PaymentContext();

        context.setStrategy(new CreditCardPayment());

        context.executePayment(100.0);

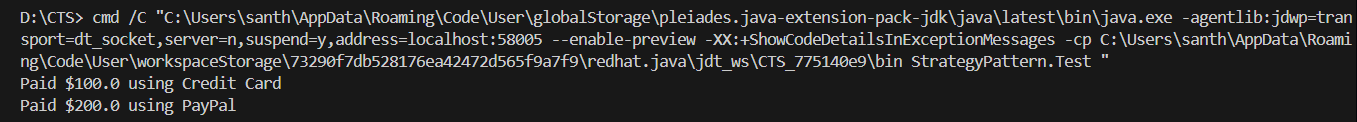
        context.setStrategy(new PayPalPayment());

        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.

**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**

public interface Command {

    void execute();

}

**Light.java**

public class Light {

    public void turnOn() {

        System.out.println("The light is on");

    }

    public void turnOff() {

        System.out.println("The light is off");

    }

}

**LightOnCommand.java**

public class LightOnCommand implements Command {

    private Light light;

    public LightOnCommand(Light light) {

        this.light = light;

    }

    @Override

    public void execute() {

        light.turnOn();

    }

}

**LightOffCommand.java**

public class LightOffCommand implements Command {

    private Light light;

    public LightOffCommand(Light light) {

        this.light = light;

    }

    @Override

    public void execute() {

        light.turnOff();

    }

}

**RemoteControl.java**

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.");

        }

    }

}

**Test.java**

public class Test {

    public static void main(String[] args) {

        Light light = new Light();

        Command lightOn = new LightOnCommand(light);

        Command lightOff = new LightOffCommand(light);

        RemoteControl remote = new RemoteControl();

        remote.setCommand(lightOn);

        remote.pressButton();

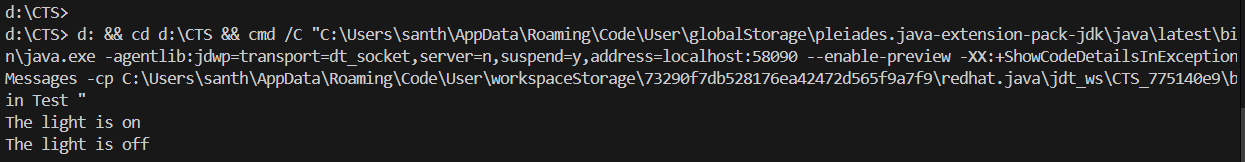
        remote.setCommand(lightOff);

        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 controller;

public 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;

    }

    public String getName() {

        return name;

    }

    public void setName(String name) {

        this.name = name;

    }

    public int getId() {

        return id;

    }

    public void setId(int id) {

        this.id = id;

    }

    public String getGrade() {

        return grade;

    }

    public void setGrade(String grade) {

        this.grade = grade;

    }

}

**StudentView.java**

package controller;

public class StudentView {

    public void displayStudentDetails(String name, int 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 controller;

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 setStudentGrade(String grade) {

        model.setGrade(grade);

    }

    public void updateView() {

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

    }

}

**Test.java**

package controller;

public class Test {

    public static void main(String[] args) {

        Student student = new Student("John", 1, "A");

        StudentView view = new StudentView();

        StudentController controller = new StudentController(student, view);

        controller.updateView();

        controller.setStudentName("Jane");

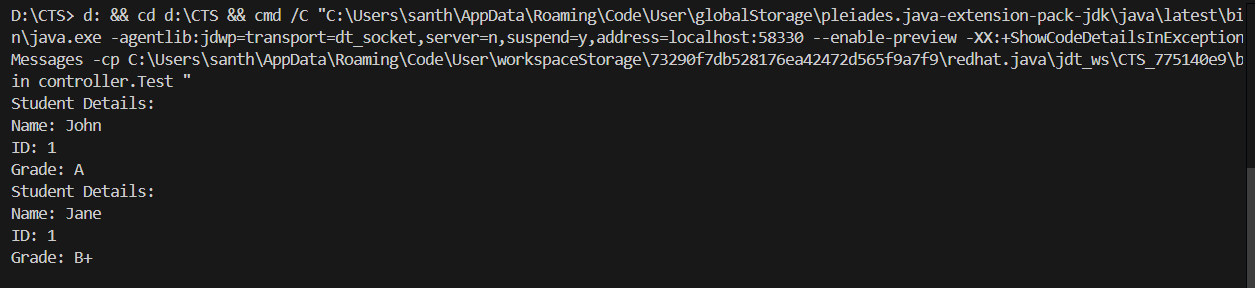
        controller.setStudentGrade("B+");

        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.

**Solution**

**CustomerRepository.java**

public interface CustomerRepository {

    String findCustomerById(int id);

}

**CustomerRepositoryImpl.java**

public class CustomerRepositoryImpl implements CustomerRepository {

    @Override

    public String findCustomerById(int id) {

        return "Customer with ID " + id + " found.";

    }

}

**CustomerService.java**

public class CustomerService {

    private CustomerRepository repo;

    public CustomerService(CustomerRepository repo) {

        this.repo = repo;

    }

    public String getCustomer(int id) {

        return repo.findCustomerById(id);

    }

}

**Test.java**

public class Test {

    public static void main(String[] args) {

        CustomerRepository repo = new CustomerRepositoryImpl();

        CustomerService service = new CustomerService(repo);

        String customerInfo = service.getCustomer(123);

        System.out.println(customerInfo);

    }

}

**Output:**

