**EXERCISE 1: SINGLETON PATTERN:**

**STEP 1. Create a New Java Project**

Create a new Java project named SingletonPatternExample in your preferred IDE.

**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.

**STEP 2. Define the Singleton Class**

**//Logger.java**

public class Logger {

private static volatile Logger instance;

private Logger() {}

public static Logger getInstance() {

if (instance == null) {

synchronized (Logger.class) {

if (instance == null) {

instance = new Logger();

}

}

}

return instance;

}

public void log(String message) {

System.out.println("[" + java.time.LocalDateTime.now() + "] Log message: " + message);

}

}

**STEP 3. Implement the Singleton Pattern**

The Logger class follows the Singleton design pattern by ensuring that there is only one instance of the class and providing a global point of access to it.

**STEP 4. Test the Singleton Implementation**

Create a test class named SingletonPatternTest to verify that only one instance of Logger is created and used across the application.

//**SingletonPatternTest.java**

public class SingletonPatternTest {

public static void main(String[] args) {

Logger logger1 = Logger.getInstance();

logger1.log("Hi, this is the first message");

Logger logger2 = Logger.getInstance();

logger2.log("Hi, this is the second message");

if (logger1 == logger2) {

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

} else {

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

}

}

}

**EXERCISE 2: 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.

**STEP 1.Create a New Java Project:**

Create a new Java project named FactoryMethodPatternExample.

**STEP 2.Define Document Classes:**

abstract class Document {

public abstract void open();

public abstract void close();

public abstract void save();

}

**STEP 3.Create Concrete Document Classes:**

class WordDocument extends Document {

@Override

public void open() {

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

}

@Override

public void close() {

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

}

@Override

public void save() {

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

}

}

// Concrete PdfDocument Class

class PdfDocument extends Document {

@Override

public void open() {

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

}

@Override

public void close() {

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

}

@Override

public void save() {

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

}

}

// Concrete ExcelDocument Class

class ExcelDocument extends Document {

@Override

public void open() {

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

}

@Override

public void close() {

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

}

@Override

public void save() {

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

}

}

**STEP 4.Implement the Factory Method:**

abstract class DocumentFactory {

public abstract Document createDocument();

}

// Concrete WordDocumentFactory Class

class WordDocumentFactory extends DocumentFactory {

@Override

public Document createDocument() {

return new WordDocument();

}

}

// Concrete PdfDocumentFactory Class

class PdfDocumentFactory extends DocumentFactory {

@Override

public Document createDocument() {

return new PdfDocument();

}

}

// Concrete ExcelDocumentFactory Class

class ExcelDocumentFactory extends DocumentFactory {

@Override

public Document createDocument() {

return new ExcelDocument();

}

}

**STEP 5.Test the Factory Method Implementation:**

public class FactoryMethodPatternExample

{

public static void main(String[] args)

{

DocumentFactory wordFactory = new WordDocumentFactory();

Document wordDoc = wordFactory.createDocument();

wordDoc.open();

wordDoc.save();

wordDoc.close();

DocumentFactory pdfFactory = new PdfDocumentFactory();

Document pdfDoc = pdfFactory.createDocument();

pdfDoc.open();

pdfDoc.save();

pdfDoc.close();

DocumentFactory excelFactory = new ExcelDocumentFactory();

Document excelDoc = excelFactory.createDocument();

excelDoc.open();

excelDoc.save();

excelDoc.close();

}

}

**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.

**STEP1: Create a New Java Project:**

Create a new Java project named **BuilderPatternExample**.

**STEP2: Define a Product Class:**

//**Computer.java**

package Builder\_example;

public class Computer {

private String cpu;

private String ram;

private String storage;

private String operatingSystem;

private Computer(Builder builder) {

this.cpu = builder.cpu;

this.ram = builder.ram;

this.storage = builder.storage;

this.operatingSystem = builder.operatingSystem;

}

public String getCPU() {

return cpu;

}

public String getRAM() {

return ram;

}

public String getStorage() {

return storage;

}

public String getOperatingSystem() {

return operatingSystem;

}

@Override

public String toString() {

return "Computer Configuration: [CPU=" + cpu + ", RAM=" + ram + ", Storage=" + storage + ", OS=" + operatingSystem + "]";

}

public static class Builder {

private String cpu;

private String ram;

private String storage;

private String operatingSystem;

public Builder() {}

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 setOperatingSystem(String operatingSystem) {

this.operatingSystem = operatingSystem;

return this;

}

public Computer build() {

return new Computer(this);

}

}

}

//**Test\_builder.java**

package Builder\_example;

import java.util.Scanner;

public class Test\_builder {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

// Get input for Gaming Computer

System.out.println("Enter details for Gaming Computer:");

System.out.print("CPU: ");

String gamingCpu = scanner.nextLine();

System.out.print("RAM: ");

String gamingRam = scanner.nextLine();

System.out.print("Storage: ");

String gamingStorage = scanner.nextLine();

System.out.print("Operating System: ");

String gamingOS = scanner.nextLine();

Computer gamingComputer = new Computer.Builder()

.setCPU(gamingCpu)

.setRAM(gamingRam)

.setStorage(gamingStorage)

.setOperatingSystem(gamingOS)

.build();

System.out.println("\nEnter details for Office Computer:");

System.out.print("CPU: ");

String officeCpu = scanner.nextLine();

System.out.print("RAM: ");

String officeRam = scanner.nextLine();

System.out.print("Storage: ");

String officeStorage = scanner.nextLine();

System.out.print("Operating System: ");

String officeOS = scanner.nextLine();

Computer officeComputer = new Computer.Builder()

.setCPU(officeCpu)

.setRAM(officeRam)

.setStorage(officeStorage)

.setOperatingSystem(officeOS)

.build();

System.out.println("\nEnter details for Budget Computer:");

System.out.print("CPU: ");

String budgetCpu = scanner.nextLine();

System.out.print("RAM: ");

String budgetRam = scanner.nextLine();

System.out.print("Storage: ");

String budgetStorage = scanner.nextLine();

System.out.print("Operating System: ");

String budgetOS = scanner.nextLine();

Computer budgetComputer = new Computer.Builder()

.setCPU(budgetCpu)

.setRAM(budgetRam)

.setStorage(budgetStorage)

.setOperatingSystem(budgetOS)

.build();

System.out.println("\nGaming Computer Configuration: " + gamingComputer);

System.out.println("Office Computer Configuration: " + officeComputer);

System.out.println("Budget Computer Configuration: " + budgetComputer);

scanner.close();

}

}

**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**

**Step 1: Create a New Java Project**

**Step 2: Define Target Interface**

**// PaymentProcessor.java**

public interface PaymentProcessor {

void processPayment(double amount);

}

**Step 3: Implement Adaptee Classes**

**// Paypal.java**

public class Paypal {

public void sendPayment(double amount) {

System.out.println("Paypal: Sending payment of Rs." + amount);

}

}

**// Stripe.java**

public class Stripe {

public void makePayment(double amount) {

System.out.println("Stripe: Making payment of Rs." + amount);

}

}

// Square.java

public class Square {

public void executePayment(double amount) {

System.out.println("Square: Making payment of Rs." + amount);

}

}

**Step 4: Implement the Adapter Classes**

// PaypalAdapter.java

public class PaypalAdapter implements PaymentProcessor {

private Paypal paypal;

public PaypalAdapter(Paypal paypal) {

this.paypal = paypal;

}

@Override

public void processPayment(double amount) {

paypal.sendPayment(amount);

}

}

// StripeAdapter.java

public class StripeAdapter implements PaymentProcessor {

private Stripe stripe;

public StripeAdapter(Stripe stripe) {

this.stripe = stripe;

}

@Override

public void processPayment(double amount) {

stripe.makePayment(amount);

}

}

// SquareAdapter.java

public class SquareAdapter implements PaymentProcessor {

private Square square;

public SquareAdapter(Square square) {

this.square = square;

}

@Override

public void processPayment(double amount) {

square.executePayment(amount);

}

}

**Step 5: Create AdapterPatternExample Class**

**// AdapterPatternExample.java**

import java.util.Scanner;

public class AdapterPatternExample {

public static void main(String[] args) {

AdapterPatternExample example = new AdapterPatternExample();

example.run();

}

public void run() {

Scanner scanner = new Scanner(System.in);

System.out.println("Choose a payment processor:");

System.out.println("1. Paypal");

System.out.println("2. Stripe");

System.out.println("3. Square");

System.out.print("Enter your choice (1/2/3): ");

int choice = scanner.nextInt();

System.out.print("Enter the amount to be processed: Rs. ");

double amount = scanner.nextDouble();

PaymentProcessor processor = null;

switch (choice) {

case 1:

Paypal paypal = new Paypal();

processor = new PaypalAdapter(paypal);

break;

case 2:

Stripe stripe = new Stripe();

processor = new StripeAdapter(stripe);

break;

case 3:

Square square = new Square();

processor = new SquareAdapter(square);

break;

default:

System.out.println("Invalid choice. Using default (Paypal).");

Paypal defaultPaypal = new Paypal();

processor = new PaypalAdapter(defaultPaypal);

break;

}

processTransaction(processor, amount);

scanner.close();

}

private void processTransaction(PaymentProcessor processor, double amount) {

processor.processPayment(amount);

}

}

**Exercise 5: 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.

**Step 1: Create a New Java Project**

Create a new Java project named `DecoratorPatternExample`.

**Step 2: Define Notifier Interface**

**//Notifier.java**

public interface Notifier {

void send(String message);

}

**//** **EmailNotifier.java**

public class EmailNotifier implements Notifier {

@Override

public void send(String message) {

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

}

}

**//** **NotifierDecorator.java**

// Abstract Decorator Class

public abstract class NotifierDecorator implements Notifier {

protected Notifier wrappedNotifier;

public NotifierDecorator(Notifier notifier) {

this.wrappedNotifier = notifier;

}

@Override

public void send(String message) {

wrappedNotifier.send(message);

}

}

**//** **SMSNotifierDecorator.java**

public class SMSNotifierDecorator extends NotifierDecorator {

public SMSNotifierDecorator(Notifier notifier) {

super(notifier);

}

@Override

public void send(String message) {

super.send(message);

sendSMS(message);

}

private void sendSMS(String message) {

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

}

}

**// SlackNotifierDecorator.java**

public class SlackNotifierDecorator extends NotifierDecorator {

public SlackNotifierDecorator(Notifier notifier) {

super(notifier);

}

@Override

public void send(String message) {

super.send(message);

sendSlack(message);

}

private void sendSlack(String message) {

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

}

}

**//Main.java**

import java.util.Scanner;

public class Main {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

System.out.println("Enter the message to send:");

String userMessage = scanner.nextLine();

Notifier notifier = new EmailNotifier();

notifier = new SMSNotifierDecorator(notifier);

notifier = new SlackNotifierDecorator(notifier);

notifier.send(userMessage);

scanner.close();

}

}

**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.
6. **Create a New Java Project:**

//**Image.java**

public interface Image {

void display();

}

**//RealImage.java**

public class RealImage implements Image {

private String fileName;

public RealImage(String fileName) {

this.fileName = fileName;

loadFromDisk(fileName);

}

private void loadFromDisk(String fileName) {

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

// Simulate a delay in loading the image from a remote server

try {

Thread.sleep(2000);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

@Override

public void display() {

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

}

}

**//ProxyImage.java**

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);

}

realImage.display();

}

**}**

**//ProxyPatternDemo.java**

import java.util.Scanner;

public class ProxyPatternDemo {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

// Define image file names

String[] fileNames = {"test\_1.jpg", "test\_2.jpg", "test\_3.jpg"};

Image[] images = new Image[fileNames.length];

// Initialize proxies

for (int i = 0; i < fileNames.length; i++) {

images[i] = new ProxyImage(fileNames[i]);

}

while (true) {

System.out.println("Choose an image to display:");

for (int i = 0; i < fileNames.length; i++) {

System.out.println((i + 1) + ". " + fileNames[i]);

}

System.out.println((fileNames.length + 1) + ". Exit");

int choice = scanner.nextInt();

if (choice == fileNames.length + 1) {

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

break;

} else if (choice > 0 && choice <= fileNames.length) {

images[choice - 1].display();

} else {

System.out.println("Invalid choice. Please enter a number between 1 and " + (fileNames.length + 1));

}

}

scanner.close();

}

}

**Exercise 7: 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:**
   1. Create a new Java project named ObserverPatternExample.
2. **Define Subject Interface:**
   1. Create an interface Stock with methods to register, deregister, and notify observers.
3. **Implement Concrete Subject:**
   1. Create a class StockMarket that implements Stock and maintains a list of observers.
4. **Define Observer Interface:**
   1. Create an interface Observer with a method update().
5. **Implement Concrete Observers:**
   1. Create classes MobileApp, WebApp that implement Observer.
6. **Test the Observer Implementation:**
   1. Create a test class to demonstrate the registration and notification of observers.

**Step 1: Create a New Java Project**

Create a new Java project named **ObserverPatternExample**.

**Step 2: Define Subject Interface**

Create an interface **Stock** with methods to register, deregister, and notify observers.

**//Stock.java**

public interface Stock {

void registerObserver(Observer observer);

void deregisterObserver(Observer observer);

void notifyObservers();

}

**//StockMarket.java**

import java.util.ArrayList;

import java.util.List;

public class StockMarket implements Stock {

private List<Observer> observers;

private String stockName;

private double stockPrice;

public StockMarket() {

observers = new ArrayList<>();

}

public void setStock(String stockName, double stockPrice) {

this.stockName = stockName;

this.stockPrice = stockPrice;

notifyObservers();

}

@Override

public void registerObserver(Observer observer) {

observers.add(observer);

}

@Override

public void deregisterObserver(Observer observer) {

observers.remove(observer);

}

@Override

public void notifyObservers() {

for (Observer observer : observers) {

observer.update(stockName, stockPrice);

}

}

}

**//Observer.java**

public interface Observer {

void update(String stockName, double stockPrice);

}

**//MobileApp.java**

public class MobileApp implements Observer {

private String appName;

public MobileApp(String appName) {

this.appName = appName;

}

@Override

public void update(String stockName, double stockPrice) {

System.out.println(appName + " received update: " + stockName + " stock price is now $" + stockPrice);

}

}

**//WebApp.java**

public class WebApp implements Observer {

private String appName;

public WebApp(String appName) {

this.appName = appName;

}

@Override

public void update(String stockName, double stockPrice) {

System.out.println(appName + " received update: " + stockName + " stock price is now $" + stockPrice);

}

}

**//ObserverPatternTest.java**

import java.util.Scanner;

public class ObserverPatternTest {

public static void main(String[] args) {

StockMarket stockMarket = new StockMarket();

Scanner scanner = new Scanner(System.in);

Observer mobileApp = new MobileApp("MobileApp");

Observer webApp = new WebApp("WebApp");

stockMarket.registerObserver(mobileApp);

stockMarket.registerObserver(webApp);

while (true) {

System.out.println("Enter stock name (or type 'exit' to quit): ");

String stockName = scanner.nextLine();

if (stockName.equalsIgnoreCase("exit")) {

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

break;

}

System.out.println("Enter stock price: ");

double stockPrice;

while (true) {

try {

stockPrice = Double.parseDouble(scanner.nextLine());

if (stockPrice < 0) {

System.out.println("Stock price cannot be negative. Please enter a valid price.");

} else {

break;

}

} catch (NumberFormatException e) {

System.out.println("Invalid input. Please enter a valid price.");

}

}

stockMarket.setStock(stockName, stockPrice);

}

scanner.close();

}

}

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

**CODE:**

**PaymentStrategy.java:**

public interface PaymentStrategy {

    void pay(double amount);

}

**PaymentContext.java**

public class PaymentContext {

    private PaymentStrategy paymentstrategy;

    public void setPaymentStrategy(PaymentStrategy paymentStrategy) {

        this.paymentstrategy = paymentStrategy;

    }

    public void pay(double amount) {

        if (paymentstrategy != null) {

            paymentstrategy.pay(amount);

        } else {

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

        }

    }

}

**PayPalPayment,java**

public class PayPalPayment implements PaymentStrategy {

    String email;

    String password;

    PayPalPayment(String email, String password) {

        this.email = email;

        this.password = password;

    }

    public void pay(double amount) {

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

    }

}

**CreditCardPayment.java**

public class CreditCardPayment implements PaymentStrategy {

    String cardnumber;

    String cardHolderNAme;

    String cvv;

    String expiryDate;

    CreditCardPayment(String cardnumber, String cardHolderName, String cvv, String expiryDate) {

        this.cardnumber = cardnumber;

        this.cardHolderNAme = cardHolderName;

        this.cvv = cvv;

        this.expiryDate = expiryDate;

    }

    public void pay(double amount) {

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

    }

}

**Main.java**

import java.util.Scanner;

public class Main {

    public static void main(String[] args) {

        Scanner scanner = **new** Scanner(System.in);

        PaymentContext paymentContext = **new** PaymentContext();

        try {

            System.out.println("Choose payment method (1: Credit Card, 2: PayPal):");

            int choice = scanner.nextInt();

            scanner.nextLine();

            System.out.println("Enter amount to pay:");

            double amount = scanner.nextDouble();

            scanner.nextLine();

            switch (choice) {

                case 1:

                    System.out.println("Enter card number:");

                    String cardNumber = scanner.nextLine();

                    System.out.println("Enter card holder name:");

                    String cardHolderName = scanner.nextLine();

                    System.out.println("Enter CVV:");

                    String cvv = scanner.nextLine();

                    System.out.println("Enter expiry date (MM/YY):");

                    String expiryDate = scanner.nextLine();

                    PaymentStrategy creditCardPayment = **new** CreditCardPayment(cardNumber, cardHolderName, cvv,

                            expiryDate);

                    paymentContext.setPaymentStrategy(creditCardPayment);

                    break;

                case 2:

                    System.out.println("Enter PayPal email:");

                    String email = scanner.nextLine();

                    System.out.println("Enter PayPal password:");

                    String password = scanner.nextLine();

                    PaymentStrategy payPalPayment = **new** PayPalPayment(email, password);

                    paymentContext.setPaymentStrategy(payPalPayment);

                    break;

                default:

                    System.out.println("Invalid choice");

                    return;

            }

            paymentContext.pay(amount);

        } finally {

            scanner.close();

        }

    }

}

**EXERCISE 9: 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.

**IMPLEMENTATION:**

// CommandPatternExample.java

package commandpatternexample;

import java.util.Scanner;

public class CommandPatternExample {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

Light livingRoomLight = new Light();

Command lightOn = new LightOnCommand(livingRoomLight);

Command lightOff = new LightOffCommand(livingRoomLight);

RemoteControl remote = new RemoteControl();

while (true) {

System.out.println("Choose an option:");

System.out.println("1. Turn the light on");

System.out.println("2. Turn the light off");

System.out.println("3. Exit");

System.out.print("Enter your choice: ");

int choice = scanner.nextInt();

if (choice == 1) {

remote.setCommand(lightOn);

remote.pressButton();

} else if (choice == 2) {

remote.setCommand(lightOff);

remote.pressButton();

} else if (choice == 3) {

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

break;

} else {

System.out.println("Invalid choice. Please enter 1, 2, or 3.");

}

}

scanner.close();

}

}

**// Command.java**

package commandpatternexample;

interface Command {

void execute();

}

**// LightOnCommand.java**

package commandpatternexample;

class LightOnCommand implements Command {

private Light light;

public LightOnCommand(Light light) {

this.light = light;

}

@Override

public void execute() {

light.turnOn();

}

}

**// LightOffCommand.java**

package commandpatternexample;

class LightOffCommand implements Command {

private Light light;

public LightOffCommand(Light light) {

this.light = light;

}

@Override

public void execute() {

light.turnOff();

}

}

**// RemoteControl.java**

package commandpatternexample;

class RemoteControl {

private Command command;

public void setCommand(Command command) {

this.command = command;

}

public void pressButton() {

command.execute();

}

}

**// Light.java**

package commandpatternexample;

class Light {

public void turnOn() {

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

}

public void turnOff() {

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

}

}

**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:**

**// Student.java**

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**

class StudentView {

public void displayStudentDetails(String studentName, int studentId, String studentGrade) {

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

System.out.println("STUDENT DETAILS");

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

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

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

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

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

}

}

**// StudentController.java**

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 String getStudentName() {

return model.getName();

}

public void setStudentId(int id) {

model.setId(id);

}

public int getStudentId() {

return model.getId();

}

public void setStudentGrade(String grade) {

model.setGrade(grade);

}

public String getStudentGrade() {

return model.getGrade();

}

public void updateView() {

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

}

}

// Main.java

import java.util.Scanner;

public class Main {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

System.out.print("Enter student name: ");

String name = scanner.nextLine();

System.out.print("Enter student ID: ");

int id = scanner.nextInt();

scanner.nextLine(); // Consume newline

System.out.print("Enter student grade: ");

String grade = scanner.nextLine();

Student student = new Student(name, id, grade);

StudentView view = new StudentView();

StudentController controller = new StudentController(student, view);

controller.updateView();

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

System.out.print("Enter new student name: ");

String newName = scanner.nextLine();

System.out.print("Enter new student ID: ");

int newId = scanner.nextInt();

scanner.nextLine(); // Consume newline

System.out.print("Enter new student grade: ");

String newGrade = scanner.nextLine();

controller.setStudentName(newName);

controller.setStudentId(newId);

controller.setStudentGrade(newGrade);

// Display updated student details

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

controller.updateView();

}

}

**Exercise 11: 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.
7. **Define Repository Interface**

**// CustomerRepository.java**

public interface CustomerRepository {

Customer findCustomerById(int id);

}

1. **Implement Concrete Repository**

**// CustomerRepositoryImpl.java**

public class CustomerRepositoryImpl implements CustomerRepository {

@Override

public Customer findCustomerById(int id) {

return new Customer(id, "John Doe");

}

}

1. **Service Class**

**// CustomerService.java**

public class CustomerService {

private final CustomerRepository customerRepository;

public CustomerService(CustomerRepository customerRepository) {

this.customerRepository = customerRepository;

}

public Customer getCustomerById(int id) {

return customerRepository.findCustomerById(id);

}

}

1. **Dependency Injection Implementation**

**// Customer.java**

public class Customer {

private int id;

private String name;

public Customer(int id, String name) {

this.id = id;

this.name = name;

}

@Override

public String toString() {

return "Customer{id=" + id + ", name='" + name + "'}";

}

}

1. **Main Class**

**// Main.java**

public class Main {

public static void main(String[] args) {

CustomerRepository customerRepository = new CustomerRepositoryImpl();

CustomerService customerService = new CustomerService(customerRepository);

Customer customer = customerService.getCustomerById(1);

System.out.println(customer);

}

}