Week1\_Design Principles and Patterns

**Exercise 1: Implementing the Singleton Pattern**

**Program:**

**Logger class:**

**package** singleton;

**public** **class** Logger {

**private** **static** Logger *instance*;

**private** Logger() {

System.***out***.println("Logger instance created.");

}

**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("LOG: " + message);

}

}

**Class SingletonTest :**

**package** singleton;

**public** **class** SingletonTest {

**public** **static** **void** main(String[] args) {

System.***out***.println("--- Testing Logger Singleton ---");

System.***out***.println("Attempting to get first logger instance...");

Logger logger1 = Logger.*getInstance*();

logger1.log("This is the first log message.");

System.***out***.println("\nAttempting to get second logger instance...");

Logger logger2 = Logger.*getInstance*();

logger2.log("This is the second log message.");

System.***out***.println("\nAttempting to get third logger instance...");

Logger logger3 = Logger.*getInstance*();

logger3.log("This is the third log message.");

System.***out***.println("\n--- Verifying Instances ---");

**boolean** areSameInstance1And2 = (logger1 == logger2);

**boolean** areSameInstance2And3 = (logger2 == logger3);

System.***out***.println("Are logger1 and logger2 the same instance? " + areSameInstance1And2);

System.***out***.println("Are logger2 and logger3 the same instance? " + areSameInstance2And3);

**if** (areSameInstance1And2 && areSameInstance2And3) {

System.***out***.println("\nSuccess: All logger instances are indeed the same.");

System.***out***.println("The Singleton pattern is correctly implemented.");

} **else** {

System.***out***.println("\nFailure: Different logger instances were created.");

System.***out***.println("The Singleton pattern might not be correctly implemented.");

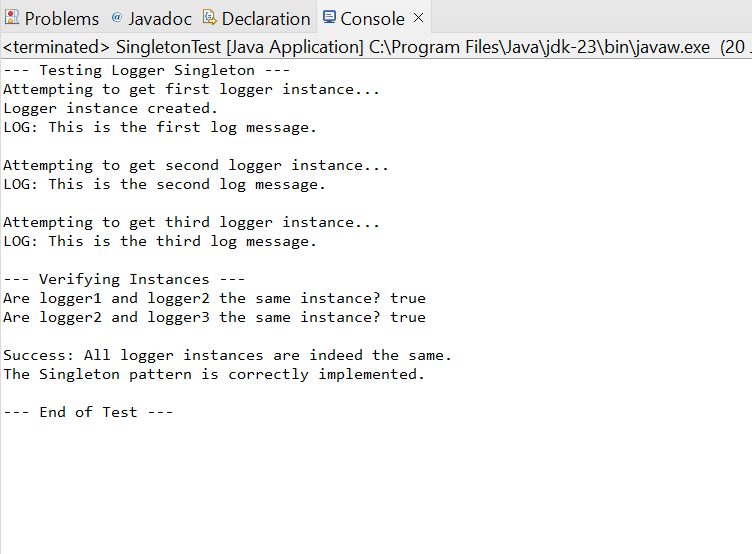
}

System.***out***.println("\n--- End of Test ---");

}

}

**OUTPUT :**



**Exercise 2: Implementing the Factory Method Pattern**

**Program:**

package factory;

/\*\*

 \* Product Interface: Defines the common interface for all document types.

 \* Concrete products will implement this interface.

 \*/

interface Document {

    void open();

    void save();

    void close();

}

/\*\*

 \* Concrete Product: Implements the Document interface for a Word document.

 \*/

class WordDocument implements Document {

    @Override

    public void open() {

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

    }

    @Override

    public void save() {

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

    }

    @Override

    public void close() {

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

    }

}

/\*\*

 \* Concrete Product: Implements the Document interface for a PDF document.

 \*/

class PdfDocument implements Document {

    @Override

    public void open() {

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

    }

    @Override

    public void save() {

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

    }

    @Override

    public void close() {

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

    }

}

/\*\*

 \* Concrete Product: Implements the Document interface for an Excel document.

 \*/

class ExcelDocument implements Document {

    @Override

    public void open() {

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

    }

    @Override

    public void save() {

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

    }

    @Override

    public void close() {

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

    }

}

/\*\*

 \* Creator Interface/Abstract Class: Declares the factory method,

 \* which returns an object of the Product type.

 \*/

abstract class DocumentCreator {

    // The factory method

    public abstract Document createDocument();

    // Other operations that might use the factory method

    public void printDocumentDetails() {

        Document doc = createDocument(); // Call the factory method

        doc.open();

        System.out.println("Document created by " + this.getClass().getSimpleName());

        doc.save();

        doc.close();

    }

}

/\*\*

 \* Concrete Creator: Overrides the factory method to return a specific

 \* concrete product (WordDocument).

 \*/

class WordDocumentCreator extends DocumentCreator {

    @Override

    public Document createDocument() {

        return new WordDocument();

    }

}

/\*\*

 \* Concrete Creator: Overrides the factory method to return a specific

 \* concrete product (PdfDocument).

 \*/

class PdfDocumentCreator extends DocumentCreator {

    @Override

    public Document createDocument() {

        return new PdfDocument();

    }

}

/\*\*

 \* Concrete Creator: Overrides the factory method to return a specific

 \* concrete product (ExcelDocument).

 \*/

class ExcelDocumentCreator extends DocumentCreator {

    @Override

    public Document createDocument() {

        return new ExcelDocument();

    }

}

/\*\*

 \* Client Code: Demonstrates the usage of the Factory Method Pattern.

 \* The client code works with the creator interface rather than concrete creators,

 \* making it flexible and easy to introduce new document types.

 \* This class now contains the main method and encapsulates all other classes.

 \*/

public class FactoryMethodProgram {

    public static void main(String[] args) {

        System.out.println("--- Demonstrating Factory Method Pattern (Single File) ---");

        // Create a Word Document using its creator

        System.out.println("\nCreating Word Document:");

        DocumentCreator wordCreator = new WordDocumentCreator();

        Document wordDoc = wordCreator.createDocument();

        wordDoc.open();

        wordDoc.save();

        wordDoc.close();

        // Demonstrate the utility method in DocumentCreator

        System.out.println("\nUsing creator utility for PDF Document:");

        DocumentCreator pdfCreator = new PdfDocumentCreator();

        pdfCreator.printDocumentDetails(); // This calls createDocument internally

        // Create an Excel Document using its creator

        System.out.println("\nCreating Excel Document:");

        DocumentCreator excelCreator = new ExcelDocumentCreator();

        Document excelDoc = excelCreator.createDocument();

        excelDoc.open();

        excelDoc.save();

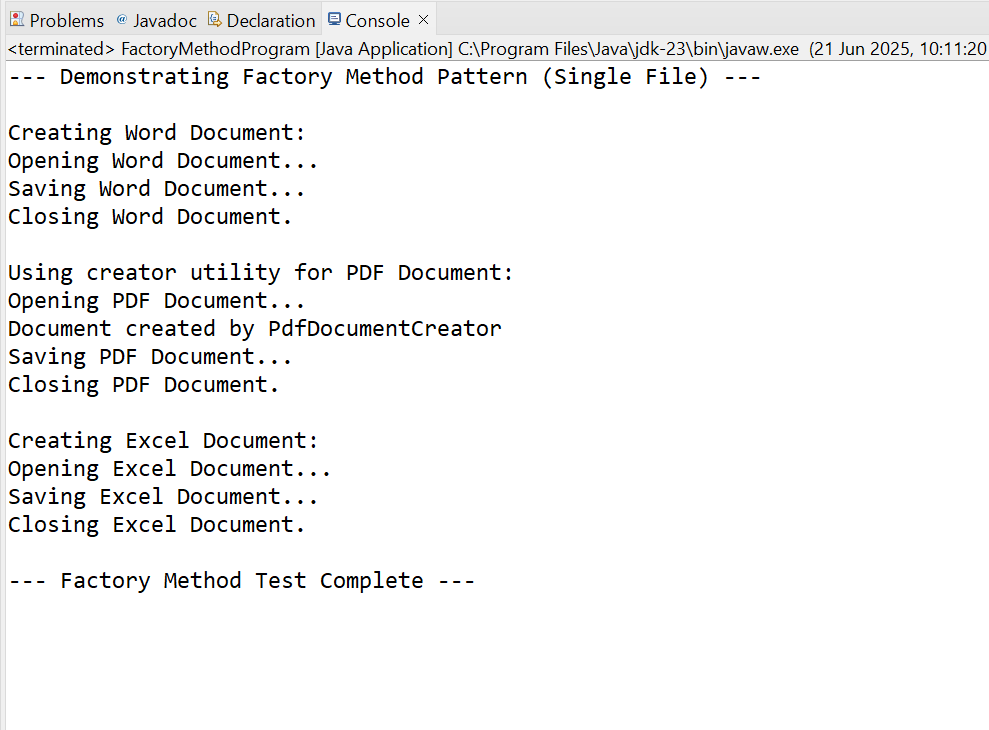
        excelDoc.close();

        System.out.println("\n--- Factory Method Test Complete ---");

    }

}

**OUTPUT:**

****

**Exercise 3: Implementing the Builder Pattern:**

**Program**:

package builderpattern;

/\*\*

 \* Product Class: Computer

 \* Represents the complex object to be built.

 \* It has various attributes, some of which may be optional.

 \*/

class Computer {

    // Required attributes

    private String cpu;

    private int ramGB;

    private int storageGB;

    // Optional attributes

    private boolean hasGraphicsCard;

    private String operatingSystem;

    private String keyboardType;

    private String mouseType;

    // Private constructor to enforce creation through the Builder.

    // It takes the Builder instance as a parameter to get all attribute values.

    private Computer(ComputerBuilder builder) {

        this.cpu = builder.cpu;

        this.ramGB = builder.ramGB;

        this.storageGB = builder.storageGB;

        this.hasGraphicsCard = builder.hasGraphicsCard;

        this.operatingSystem = builder.operatingSystem;

        this.keyboardType = builder.keyboardType;

        this.mouseType = builder.mouseType;

    }

    // Getters for all attributes

    public String getCpu() {

        return cpu;

    }

    public int getRamGB() {

        return ramGB;

    }

    public int getStorageGB() {

        return storageGB;

    }

    public boolean hasGraphicsCard() {

        return hasGraphicsCard;

    }

    public String getOperatingSystem() {

        return operatingSystem;

    }

    public String getKeyboardType() {

        return keyboardType;

    }

    public String getMouseType() {

        return mouseType;

    }

    @Override

    public String toString() {

        return "Computer Configuration:\n" +

               "  CPU: " + cpu + "\n" +

               "  RAM: " + ramGB + "GB\n" +

               "  Storage: " + storageGB + "GB\n" +

               "  Graphics Card: " + (hasGraphicsCard ? "Included" : "Not Included") + "\n" +

               "  OS: " + (operatingSystem != null ? operatingSystem : "N/A") + "\n" +

               "  Keyboard: " + (keyboardType != null ? keyboardType : "N/A") + "\n" +

               "  Mouse: " + (mouseType != null ? mouseType : "N/A");

    }

    /\*\*

     \* Builder Class: ComputerBuilder

     \* A static nested class that provides a fluent API for building Computer objects.

     \*/

    public static class ComputerBuilder {

        // Same attributes as Computer, initially set to default/null

        private String cpu;

        private int ramGB;

        private int storageGB;

        private boolean hasGraphicsCard = false; // Default to false

        private String operatingSystem = null;

        private String keyboardType = null;

        private String mouseType = null;

        // Constructor for required attributes

        public ComputerBuilder(String cpu, int ramGB, int storageGB) {

            this.cpu = cpu;

            this.ramGB = ramGB;

            this.storageGB = storageGB;

        }

        // Methods to set optional attributes, returning the builder itself for chaining

        public ComputerBuilder withGraphicsCard(boolean hasGraphicsCard) {

            this.hasGraphicsCard = hasGraphicsCard;

            return this;

        }

        public ComputerBuilder withOperatingSystem(String operatingSystem) {

            this.operatingSystem = operatingSystem;

            return this;

        }

        public ComputerBuilder withKeyboardType(String keyboardType) {

            this.keyboardType = keyboardType;

            return this;

        }

        public ComputerBuilder withMouseType(String mouseType) {

            this.mouseType = mouseType;

            return this;

        }

        /\*\*

         \* build() method: Creates and returns a Computer instance.

         \*/

        public Computer build() {

            return new Computer(this);

        }

    }

}

/\*\*

 \* Test Class: BuilderPatternProgram

 \* Demonstrates the creation of different configurations of Computer

 \* using the Builder pattern.

 \*/

public class BuilderPatternProgram {

    public static void main(String[] args) {

        System.out.println("--- Demonstrating Builder Pattern ---");

        // 1. Build a basic gaming PC

        System.out.println("\nBuilding a Basic Gaming PC:");

        Computer gamingPC = new Computer.ComputerBuilder("Intel i7", 16, 512)

                                        .withGraphicsCard(true)

                                        .withOperatingSystem("Windows 11")

                                        .withMouseType("Gaming Mouse")

                                        .build();

        System.out.println(gamingPC);

        // 2. Build a high-end workstation

        System.out.println("\nBuilding a High-End Workstation:");

        Computer workstation = new Computer.ComputerBuilder("AMD Ryzen 9", 64, 2048)

                                           .withGraphicsCard(true)

                                           .withOperatingSystem("Linux")

                                           .withKeyboardType("Mechanical Keyboard")

                                           .withMouseType("Ergonomic Mouse")

                                           .build();

        System.out.println(workstation);

        // 3. Build a simple office PC (without graphics card or specific peripherals)

        System.out.println("\nBuilding a Simple Office PC:");

        Computer officePC = new Computer.ComputerBuilder("Intel i3", 8, 256)

                                        .withOperatingSystem("Windows 10")

                                        .build();

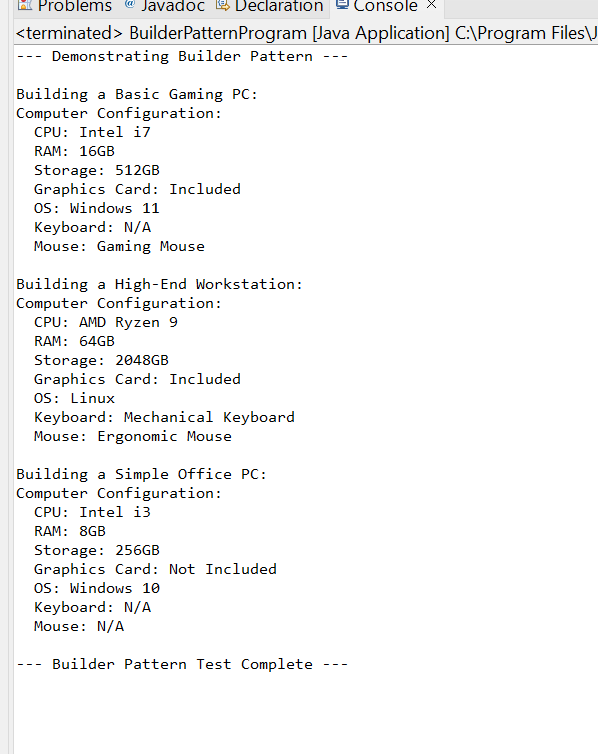
        System.out.println(officePC);

        System.out.println("\n--- Builder Pattern Test Complete ---");

    }

}

**OUTPUT:**



**Exercise 4: Implementing the Adapter Pattern**

**Program**:

package adapterpattern;

/\*\*

 \* Target Interface: PaymentProcessor

 \* This is the interface that our client code expects to work with.

 \*/

interface PaymentProcessor {

    void processPayment(double amount);

}

/\*\*

 \* Adaptee Class 1: OldPaymentGatewayA

 \* An existing third-party payment gateway with a specific interface.

 \* It has its own methods that are not compatible with PaymentProcessor directly.

 \*/

class OldPaymentGatewayA {

    public void makePaymentA(double amount) {

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

        // Simulate complex logic specific to Gateway A

    }

}

/\*\*

 \* Adaptee Class 2: LegacyPaymentGatewayB

 \* Another existing third-party payment gateway with a different interface.

 \*/

class LegacyPaymentGatewayB {

    public void executePaymentB(double value) {

        System.out.println("Initiating payment for amount " + value + " via LegacyPaymentGatewayB.");

        // Simulate complex logic specific to Gateway B

    }

    public void completeTransaction() {

        System.out.println("LegacyPaymentGatewayB: Transaction completed.");

    }

}

/\*\*

 \* Adapter Class 1: PaymentGatewayAAdapter

 \* Adapts OldPaymentGatewayA to the PaymentProcessor interface.

 \* It implements the Target interface and wraps an instance of the Adaptee.

 \*/

class PaymentGatewayAAdapter implements PaymentProcessor {

    private OldPaymentGatewayA gatewayA; // Reference to the adaptee

    public PaymentGatewayAAdapter(OldPaymentGatewayA gatewayA) {

        this.gatewayA = gatewayA;

    }

    @Override

    public void processPayment(double amount) {

        System.out.println("Adapter: Converting processPayment call to makePaymentA for Gateway A.");

        gatewayA.makePaymentA(amount); // Delegates the call to the adaptee's specific method

    }

}

/\*\*

 \* Adapter Class 2: PaymentGatewayBAdapter

 \* Adapts LegacyPaymentGatewayB to the PaymentProcessor interface.

 \*/

class PaymentGatewayBAdapter implements PaymentProcessor {

    private LegacyPaymentGatewayB gatewayB; // Reference to the adaptee

    public PaymentGatewayBAdapter(LegacyPaymentGatewayB gatewayB) {

        this.gatewayB = gatewayB;

    }

    @Override

    public void processPayment(double amount) {

        System.out.println("Adapter: Converting processPayment call to executePaymentB for Gateway B.");

        gatewayB.executePaymentB(amount); // Delegates the call to the adaptee's specific method

        gatewayB.completeTransaction();   // Can also handle additional, related calls if needed

    }

}

/\*\*

 \* Test Class: AdapterPatternProgram

 \* Demonstrates the use of different payment gateways through their respective adapters.

 \* The client code (main method) interacts solely with the common PaymentProcessor interface,

 \* making it independent of the specific gateway implementations.

 \*/

public class AdapterPatternProgram {

    public static void main(String[] args) {

        System.out.println("--- Demonstrating Adapter Pattern ---");

        // Client code expects a PaymentProcessor interface

        PaymentProcessor processor1;

        PaymentProcessor processor2;

        // Scenario 1: Integrate with OldPaymentGatewayA using its adapter

        System.out.println("\nIntegrating with OldPaymentGatewayA:");

        OldPaymentGatewayA oldGateway = new OldPaymentGatewayA();

        processor1 = new PaymentGatewayAAdapter(oldGateway);

        processor1.processPayment(100.50);

        // Scenario 2: Integrate with LegacyPaymentGatewayB using its adapter

        System.out.println("\nIntegrating with LegacyPaymentGatewayB:");

        LegacyPaymentGatewayB legacyGateway = new LegacyPaymentGatewayB();

        processor2 = new PaymentGatewayBAdapter(legacyGateway);

        processor2.processPayment(250.75);

        // You can easily add more payment gateways and their adapters

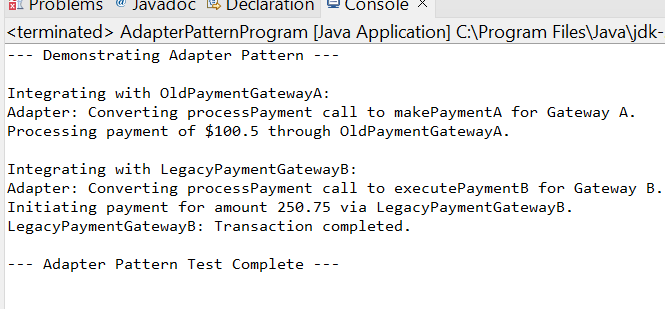
        // without modifying the client code (this main method).

        System.out.println("\n--- Adapter Pattern Test Complete ---");

    }

}

**OUTPUT:**



**Exercise 5: Implementing the Decorator Pattern**

**Program**:

package decoratorpattern;

/\*\*

 \* Component Interface: Notifier

 \* Defines the common operation that both concrete components and decorators will implement.

 \*/

interface Notifier {

    void send(String message);

}

/\*\*

 \* Concrete Component: EmailNotifier

 \* The base concrete implementation of the Notifier.

 \*/

class EmailNotifier implements Notifier {

    @Override

    public void send(String message) {

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

    }

}

/\*\*

 \* Abstract Decorator: NotifierDecorator

 \* Implements the Component interface and holds a reference to a Component object.

 \* It also implements the send method by delegating to the wrapped component.

 \*/

abstract class NotifierDecorator implements Notifier {

    protected Notifier wrappedNotifier; // The notifier being decorated

    public NotifierDecorator(Notifier notifier) {

        this.wrappedNotifier = notifier;

    }

    @Override

    public void send(String message) {

        wrappedNotifier.send(message); // Delegates to the wrapped notifier

    }

}

/\*\*

 \* Concrete Decorator 1: SMSNotifierDecorator

 \* Adds SMS sending functionality to the wrapped Notifier.

 \*/

class SMSNotifierDecorator extends NotifierDecorator {

    public SMSNotifierDecorator(Notifier notifier) {

        super(notifier);

    }

    @Override

    public void send(String message) {

        super.send(message); // Call the wrapped notifier's send method

        sendSMS(message);    // Add SMS specific functionality

    }

    private void sendSMS(String message) {

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

    }

}

/\*\*

 \* Concrete Decorator 2: SlackNotifierDecorator

 \* Adds Slack notification functionality to the wrapped Notifier.

 \*/

class SlackNotifierDecorator extends NotifierDecorator {

    public SlackNotifierDecorator(Notifier notifier) {

        super(notifier);

    }

    @Override

    public void send(String message) {

        super.send(message); // Call the wrapped notifier's send method

        sendSlackMessage(message); // Add Slack specific functionality

    }

    private void sendSlackMessage(String message) {

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

    }

}

/\*\*

 \* Test Class: DecoratorPatternProgram

 \* Demonstrates sending notifications via multiple channels using decorators.

 \*/

public class DecoratorPatternProgram {

    public static void main(String[] args) {

        System.out.println("--- Demonstrating Decorator Pattern ---");

        // 1. Send a basic email notification

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

        Notifier emailOnlyNotifier = new EmailNotifier();

        emailOnlyNotifier.send("Hello World!");

        // 2. Send notification via Email and SMS

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

        Notifier emailAndSMSNotifier = new SMSNotifierDecorator(new EmailNotifier());

        emailAndSMSNotifier.send("Your order has shipped!");

        // 3. Send notification via Email, SMS, and Slack

        System.out.println("\nSending Email, SMS, and Slack Notification:");

        Notifier fullNotifier = new SlackNotifierDecorator(new SMSNotifierDecorator(new EmailNotifier()));

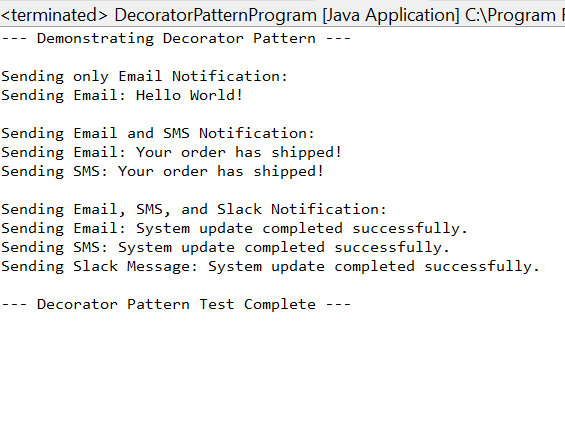
        fullNotifier.send("System update completed successfully.");

        System.out.println("\n--- Decorator Pattern Test Complete ---");

    }

}

**OUTPUT**:



**Exercise 6: Implementing the Proxy Pattern**

**Program**:

package proxypattern;

/\*\*

 \* Subject Interface: Image

 \* Defines the common interface for RealImage and ProxyImage.

 \*/

interface Image {

    void display();

}

/\*\*

 \* Real Subject Class: RealImage

 \* Represents the actual image that needs to be loaded (potentially an expensive operation).

 \*/

class RealImage implements Image {

    private String filename;

    public RealImage(String filename) {

        this.filename = filename;

        loadFromRemoteServer(filename); // Simulates expensive loading

    }

    private void loadFromRemoteServer(String filename) {

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

        try {

            Thread.sleep(2000); // Simulate network delay

        } catch (InterruptedException e) {

            Thread.currentThread().interrupt();

        }

        System.out.println("Image " + filename + " loaded.");

    }

    @Override

    public void display() {

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

    }

}

/\*\*

 \* Proxy Class: ProxyImage

 \* Controls access to RealImage, providing lazy initialization and caching.

 \*/

class ProxyImage implements Image {

    private String filename;

    private RealImage realImage; // Reference to the real subject

    private boolean isCached = false; // Simple caching mechanism

    public ProxyImage(String filename) {

        this.filename = filename;

    }

    @Override

    public void display() {

        if (!isCached) {

            System.out.println("Proxy: Image not cached. Initializing RealImage for " + filename + ".");

            if (realImage == null) {

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

            }

            isCached = true; // Mark as cached after first load

        } else {

            System.out.println("Proxy: Image " + filename + " is already cached. Displaying directly.");

        }

        realImage.display(); // Delegate the display call to the real image

    }

}

/\*\*

 \* Test Class: ProxyPatternProgram

 \* Demonstrates the use of ProxyImage to load and display images.

 \*/

public class ProxyPatternProgram {

    public static void main(String[] args) {

        System.out.println("--- Demonstrating Proxy Pattern ---");

        // Image will not be loaded immediately, only when display() is called

        System.out.println("\nFirst image creation (lazy load):");

        Image image1 = new ProxyImage("photo1.jpg");

        // No "Loading image..." message yet

        System.out.println("\nFirst call to display() for photo1.jpg:");

        image1.display(); // This will trigger the actual loading

        System.out.println("\nSecond call to display() for photo1.jpg:");

        image1.display(); // This should use the cached image

        System.out.println("\nCreating and displaying a second image:");

        Image image2 = new ProxyImage("document\_scan.png");

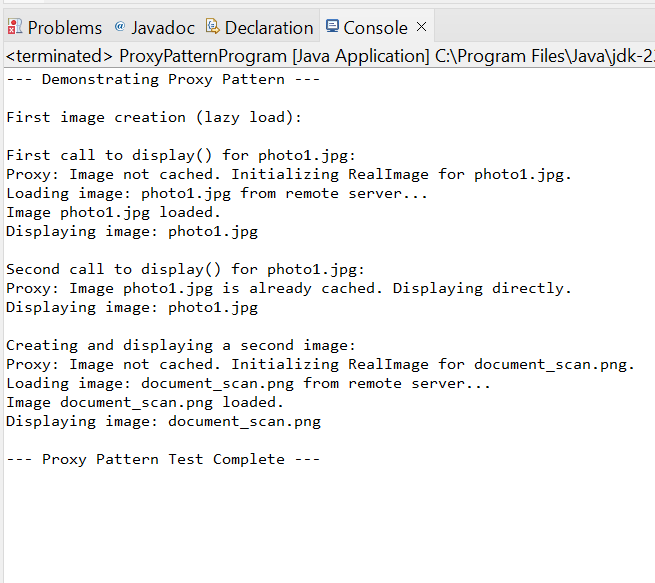
        image2.display(); // This will trigger loading for the new image

        System.out.println("\n--- Proxy Pattern Test Complete ---");

    }

}

**OUTPUT:**



**Exercise 7: Implementing the Observer Pattern**

**Program:**

package observerpattern;

import java.util.ArrayList;

import java.util.List;

/\*\*

 \* Observer Interface: StockObserver

 \* Defines the update method that concrete observers will implement

 \* to receive notifications from the subject.

 \*/

interface StockObserver {

    void update(String stockSymbol, double price);

}

/\*\*

 \* Subject Interface: StockSubject

 \* Defines methods for attaching, detaching, and notifying observers.

 \*/

interface StockSubject {

    void registerObserver(StockObserver observer);

    void deregisterObserver(StockObserver observer);

    void notifyObservers();

}

/\*\*

 \* Concrete Subject: StockMarket

 \* Maintains a list of observers and notifies them when its state (stock prices) changes.

 \*/

class StockMarket implements StockSubject {

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

    private String stockSymbol;

    private double price;

    public StockMarket(String stockSymbol, double initialPrice) {

        this.stockSymbol = stockSymbol;

        this.price = initialPrice;

        System.out.println("StockMarket created for " + stockSymbol + " with initial price $" + initialPrice);

    }

    @Override

    public void registerObserver(StockObserver observer) {

        observers.add(observer);

        System.out.println("Observer registered: " + observer.getClass().getSimpleName());

    }

    @Override

    public void deregisterObserver(StockObserver observer) {

        observers.remove(observer);

        System.out.println("Observer deregistered: " + observer.getClass().getSimpleName());

    }

    /\*\*

     \* Notifies all registered observers about the change in stock price.

     \*/

    @Override

    public void notifyObservers() {

        System.out.println("\nNotifying observers for " + stockSymbol + " price change to $" + price);

        for (StockObserver observer : observers) {

            observer.update(stockSymbol, price);

        }

    }

    /\*\*

     \* Method to change the stock price and trigger notification.

     \*/

    public void setPrice(double newPrice) {

        if (this.price != newPrice) {

            System.out.println("\nStock " + stockSymbol + " price changed from $" + this.price + " to $" + newPrice);

            this.price = newPrice;

            notifyObservers(); // Notify observers when price changes

        } else {

            System.out.println("Stock " + stockSymbol + " price remains $" + newPrice + ". No notification needed.");

        }

    }

    public String getStockSymbol() {

        return stockSymbol;

    }

    public double getPrice() {

        return price;

    }

}

/\*\*

 \* Concrete Observer 1: MobileAppDisplay

 \* Updates its display when notified of a stock price change.

 \*/

class MobileAppDisplay implements StockObserver {

    private String name;

    public MobileAppDisplay(String name) {

        this.name = name;

    }

    @Override

    public void update(String stockSymbol, double price) {

        System.out.println(name + " (Mobile App): " + stockSymbol + " is now $" + price);

    }

}

/\*\*

 \* Concrete Observer 2: WebAppDisplay

 \* Updates its display when notified of a stock price change.

 \*/

class WebAppDisplay implements StockObserver {

    private String name;

    public WebAppDisplay(String name) {

        this.name = name;

    }

    @Override

    public void update(String stockSymbol, double price) {

        System.out.println(name + " (Web App): " + stockSymbol + " price updated to $" + price);

    }

}

/\*\*

 \* Test Class: ObserverPatternProgram

 \* Demonstrates the registration and notification of observers.

 \*/

public class ObserverPatternProgram {

    public static void main(String[] args) {

        System.out.println("--- Demonstrating Observer Pattern ---");

        // Create a StockMarket subject

        StockMarket appleStock = new StockMarket("AAPL", 170.00);

        // Create observers

        StockObserver mobileApp1 = new MobileAppDisplay("User A's Phone");

        StockObserver webApp1 = new WebAppDisplay("User B's Dashboard");

        StockObserver mobileApp2 = new MobileAppDisplay("User C's Tablet");

        // Register observers to the subject

        appleStock.registerObserver(mobileApp1);

        appleStock.registerObserver(webApp1);

        appleStock.registerObserver(mobileApp2);

        // Simulate stock price changes

        System.out.println("\n--- Simulating Price Changes ---");

        appleStock.setPrice(172.50); // Price changes, observers are notified

        appleStock.setPrice(172.50); // No change, no notification

        // Deregister one observer

        System.out.println("\n--- Deregistering User A's Phone ---");

        appleStock.deregisterObserver(mobileApp1);

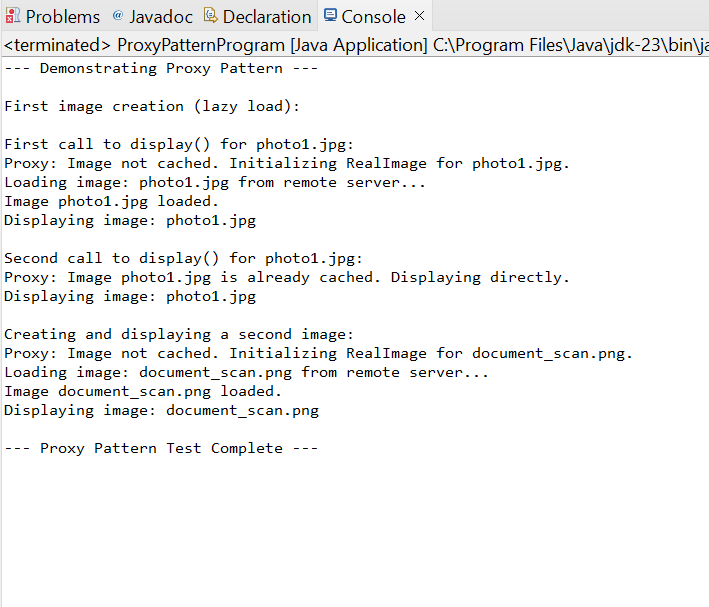
        appleStock.setPrice(175.00); // Price changes again, only remaining observers are notified

        System.out.println("\n--- Observer Pattern Test Complete ---");

    }

}

OUTPUT:



**Exercise 8: Implementing the Strategy Pattern**

**Program**:

package strategypattern;

/\*\*

 \* Strategy Interface: PaymentStrategy

 \* Declares the common interface for all supported payment algorithms.

 \*/

interface PaymentStrategy {

    void pay(double amount);

}

/\*\*

 \* Concrete Strategy 1: CreditCardPayment

 \* Implements the payment logic for credit card.

 \*/

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("Paying $" + amount + " using Credit Card (Card No: " + cardNumber.substring(0, 4) + "XXXX XXXX " + cardNumber.substring(12) + ")");

        // Simulate actual credit card processing logic

    }

}

/\*\*

 \* Concrete Strategy 2: PayPalPayment

 \* Implements the payment logic for PayPal.

 \*/

class PayPalPayment implements PaymentStrategy {

    private String email;

    public PayPalPayment(String email) {

        this.email = email;

    }

    @Override

    public void pay(double amount) {

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

        // Simulate actual PayPal processing logic

    }

}

/\*\*

 \* Concrete Strategy 3: BitcoinPayment (Example of adding a new strategy)

 \* Implements the payment logic for Bitcoin.

 \*/

class BitcoinPayment implements PaymentStrategy {

    private String walletAddress;

    public BitcoinPayment(String walletAddress) {

        this.walletAddress = walletAddress;

    }

    @Override

    public void pay(double amount) {

        System.out.println("Paying $" + amount + " using Bitcoin (Wallet: " + walletAddress + ")");

        System.out.println("  -- Please confirm transaction on your Bitcoin wallet.");

    }

}

/\*\*

 \* Context Class: PaymentContext

 \* Holds a reference to a Strategy object and delegates the payment execution to it.

 \* The client interacts with the context.

 \*/

class PaymentContext {

    private PaymentStrategy paymentStrategy;

    public void setPaymentStrategy(PaymentStrategy paymentStrategy) {

        this.paymentStrategy = paymentStrategy;

        System.out.println("Payment strategy set to: " + paymentStrategy.getClass().getSimpleName());

    }

    public void executePayment(double amount) {

        if (paymentStrategy == null) {

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

            return;

        }

        System.out.println("Attempting to process payment of $" + amount + "...");

        paymentStrategy.pay(amount); // Delegates to the selected strategy

        System.out.println("Payment processed.");

    }

}

/\*\*

 \* Test Class: StrategyPatternProgram

 \* Demonstrates selecting and using different payment strategies at runtime.

 \*/

public class StrategyPatternProgram {

    public static void main(String[] args) {

        System.out.println("--- Demonstrating Strategy Pattern ---");

        PaymentContext paymentSystem = new PaymentContext();

        // Scenario 1: Pay using Credit Card

        System.out.println("\n--- Scenario 1: Credit Card Payment ---");

        PaymentStrategy creditCard = new CreditCardPayment("1234567890123456", "John Doe");

        paymentSystem.setPaymentStrategy(creditCard);

        paymentSystem.executePayment(50.00);

        // Scenario 2: Pay using PayPal

        System.out.println("\n--- Scenario 2: PayPal Payment ---");

        PaymentStrategy payPal = new PayPalPayment("john.doe@example.com");

        paymentSystem.setPaymentStrategy(payPal);

        paymentSystem.executePayment(120.75);

        // Scenario 3: Pay using Bitcoin (new strategy)

        System.out.println("\n--- Scenario 3: Bitcoin Payment ---");

        PaymentStrategy bitcoin = new BitcoinPayment("1A1zP1eP5QGefi2DMPTfTL5SLmv7DivfNa");

        paymentSystem.setPaymentStrategy(bitcoin);

        paymentSystem.executePayment(0.0012);

        System.out.println("\n--- Strategy Pattern Test Complete ---");

    }

}

**OUTPUT:**

