**Exercise 1: Implementing the Singleton Pattern**

**Logger.java**

package mypackage;

public class Logger {

private static Logger instance;

private Logger() {

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

}

public static Logger getInstance() {

if (instance == null) {

instance = new Logger();

}

return instance;

}

public void log(String message) {

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

}

}

**TestSingleton.java**

package mypackage;

public class TestSingleton {

public static void main(String[] args) {

System.out.println("Testing Singleton Pattern...");

Logger logger1 = Logger.getInstance();

logger1.log("First log message");

Logger logger2 = Logger.getInstance();

logger2.log("Second log message");

if (logger1 == logger2) {

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

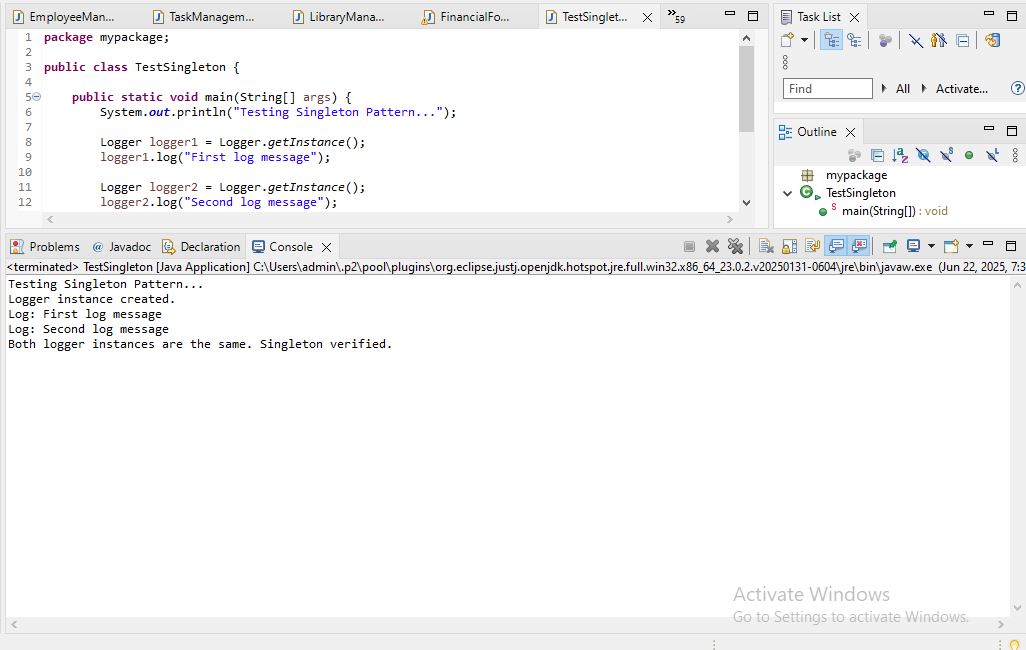
} else {

System.out.println("Different logger instances. Singleton failed.");

}

}

}



**Implementing the Singleton Pattern**  
We need to ensure that the Logger utility class has only one instance throughout the application to maintain consistent logging.

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

A new Java project is created with the name SingletonPatternExample.

**2. Define a Singleton Class:**

* The class Logger is defined with:
  + A private static variable instance to hold the single instance of Logger.
  + A private constructor to prevent external instantiation.
  + A public static method getInstance() to provide global access to the single instance.

**3. Implement the Singleton Pattern:**

* When getInstance() is called for the first time, a new instance of Logger is created.
* On subsequent calls, the existing instance is returned.
* This ensures that only one instance exists throughout the application lifecycle.

**4. Test the Singleton Implementation:**

* The TestSingleton class tests the Logger class.
* It retrieves the Logger instance multiple times.
* It compares the instances to verify that both references point to the same object, confirming the Singleton behavior.

**Advantages of Singleton Pattern:**

* Controls object creation and ensures only one instance.
* Saves memory by avoiding multiple instances.
* Useful for shared resources like logging, configuration, or database connections.

**Time Complexity:**

* The time complexity of getInstance() is O(1), as it performs a simple check and returns the instance.

**Possible Optimizations:**

* Use synchronization for thread-safety in multi-threaded environments.
* Use eager initialization or double-checked locking for advanced implementations.

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

**Document.java**

package mypackage;

public interface Document {

void open();

}

**WordDocument.java**

package mypackage;

public class WordDocument implements Document {

public void open() {

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

}

}

**PdfDocument.java**

package mypackage;

public class PdfDocument implements Document {

public void open() {

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

}

}

**ExcelDocument.java**

package mypackage;

public class ExcelDocument implements Document {

public void open() {

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

}

}

**DocumentFactory.java**

package mypackage;

public abstract class DocumentFactory {

public abstract Document createDocument();

}

**WordDocumentFactory.java**

package mypackage;

public class WordDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new WordDocument();

}

}

**PdfDocumentFactory.java**

package mypackage;

public class PdfDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new PdfDocument();

}

}

**ExcelDocumentFactory.java**

package mypackage;

public class ExcelDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new ExcelDocument();

}

}

**TestFactoryMethod.java**

package mypackage;

import java.util.Scanner;

public class TestFactoryMethod {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

System.out.println("Select Document Type:");

System.out.println("1. Word Document");

System.out.println("2. PDF Document");

System.out.println("3. Excel Document");

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

int choice = sc.nextInt();

DocumentFactory factory = null;

switch (choice) {

case 1:

factory = new WordDocumentFactory();

break;

case 2:

factory = new PdfDocumentFactory();

break;

case 3:

factory = new ExcelDocumentFactory();

break;

default:

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

System.exit(0);

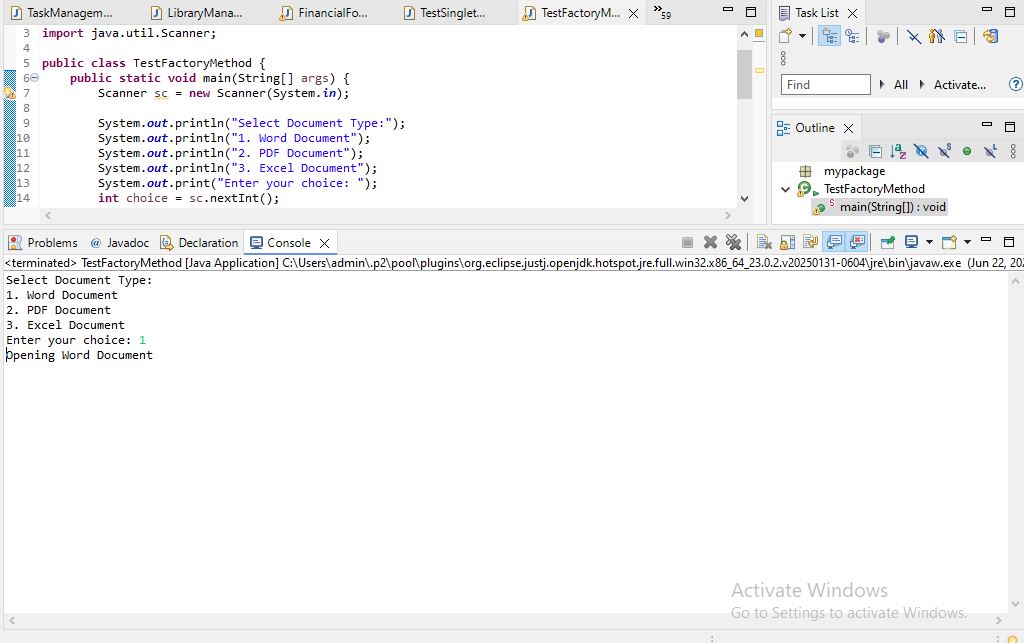
}

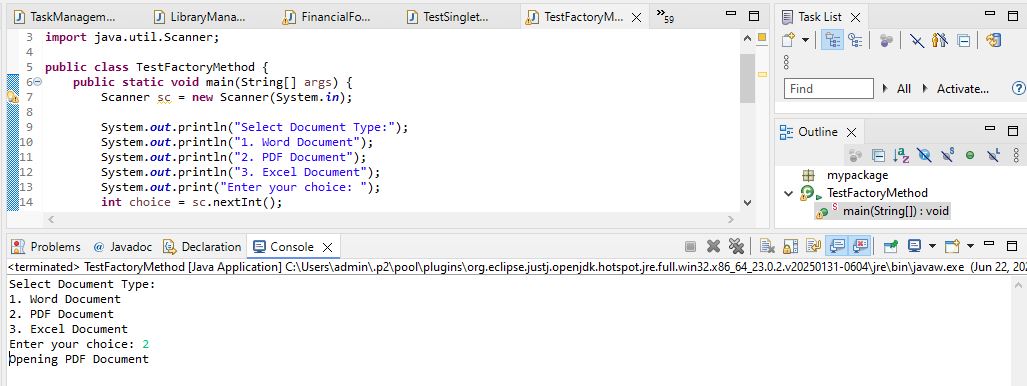
Document doc = factory.createDocument();

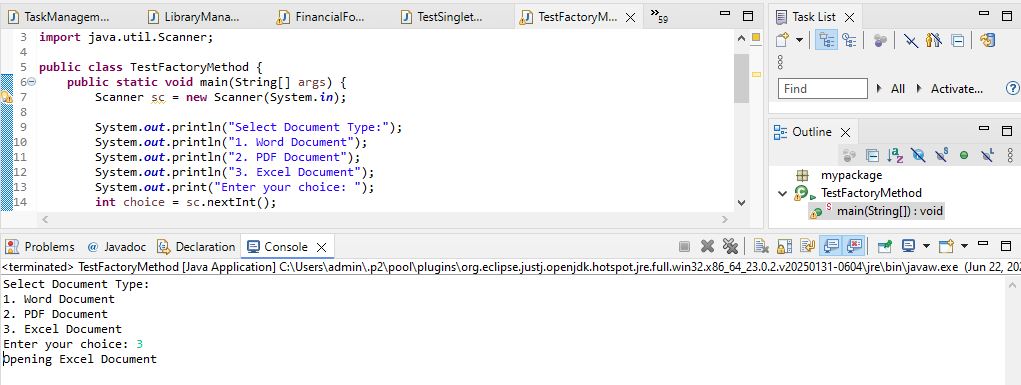
doc.open();

}

}







We are developing a document management system that creates different types of documents (Word, PDF, Excel). To achieve loose coupling and scalability, we use the Factory Method Pattern.

**1. Create Java Project:**

* A new Java project named FactoryMethodPatternExample is created.

**2. Define Document Interface:**

* A common interface Document is created with a method open() that all document types implement.

**3. Create Concrete Document Classes:**

* WordDocument, PdfDocument, and ExcelDocument classes implement the Document interface and provide their own version of the open() method.

**4. Implement Factory Method:**

* An abstract class DocumentFactory is created with an abstract method createDocument().
* Concrete factory classes WordDocumentFactory, PdfDocumentFactory, and ExcelDocumentFactory extend DocumentFactory and implement the createDocument() method to return the corresponding document object.

**5. Test Factory Method Implementation:**

* In the TestFactoryMethod class, the user selects the document type.
* Based on user choice, the corresponding factory is created.
* The factory creates the document object and calls open() method to simulate opening the document.

**Advantages of Factory Method Pattern:**

* Provides loose coupling between client code and object creation logic.
* Adding new document types requires creating new factory classes without changing existing code.
* Promotes open/closed principle: open for extension, closed for modification.

**Time Complexity:**

* The time complexity for object creation is O(1), as only one object is created at a time.

**Real-life Usage:**

* Document editors (Microsoft Office, Google Docs)
* File readers
* GUI toolkits

**Exercise 3: Implementing the Builder Pattern**

**Computer.java**

**package** mypackage;

**public** **class** Computer {

// Required parameters

**private** String CPU;

**private** String RAM;

// Optional parameters

**private** String storage;

**private** String graphicsCard;

**private** String operatingSystem;

// Private constructor

**private** Computer(Builder builder) {

**this**.CPU = builder.CPU;

**this**.RAM = builder.RAM;

**this**.storage = builder.storage;

**this**.graphicsCard = builder.graphicsCard;

**this**.operatingSystem = builder.operatingSystem;

}

// Static nested Builder class

**public** **static** **class** Builder {

**private** String CPU;

**private** String RAM;

**private** String storage;

**private** String graphicsCard;

**private** String operatingSystem;

**public** Builder(String CPU, String RAM) {

**this**.CPU = CPU;

**this**.RAM = RAM;

}

**public** Builder storage(String storage) {

**this**.storage = storage;

**return** **this**;

}

**public** Builder graphicsCard(String graphicsCard) {

**this**.graphicsCard = graphicsCard;

**return** **this**;

}

**public** Builder operatingSystem(String operatingSystem) {

**this**.operatingSystem = operatingSystem;

**return** **this**;

}

**public** Computer build() {

**return** **new** Computer(**this**);

}

}

@Override

**public** String toString() {

**return** "Computer Configuration:\n" +

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

"RAM: " + RAM + "\n" +

"Storage: " + (storage != **null** ? storage : "Not specified") + "\n" +

"Graphics Card: " + (graphicsCard != **null** ? graphicsCard : "Not specified") + "\n" +

"Operating System: " + (operatingSystem != **null** ? operatingSystem : "Not specified");

}

}

**TestBuilderPattern.java**

**package** mypackage;

**public** **class** TestBuilderPattern {

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

// Creating computer with minimal configuration

Computer basicComputer = **new** Computer.Builder("Intel i5", "8GB").build();

System.***out***.println(basicComputer);

System.***out***.println();

// Creating computer with full configuration

Computer gamingComputer = **new** Computer.Builder("AMD Ryzen 9", "32GB")

.storage("1TB SSD")

.graphicsCard("NVIDIA RTX 4090")

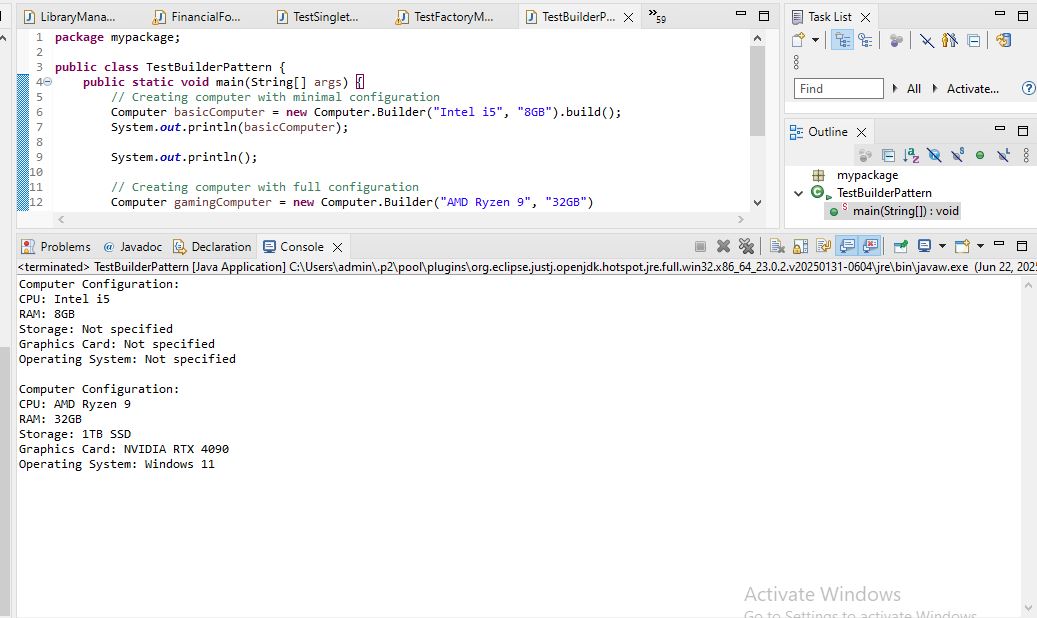
.operatingSystem("Windows 11")

.build();

System.***out***.println(gamingComputer);

}

}

  
We are developing a system to create complex objects such as a Computer with multiple optional parts.

**1. Product Class:**  
The Computer class represents the product with attributes: CPU, RAM, Storage, Graphics Card, and Operating System.  
Some attributes are mandatory (CPU, RAM), while others are optional.

**2. Builder Class:**  
A static nested class Builder is created inside the Computer class to allow step-by-step construction.  
The Builder class has methods to set optional attributes.  
The build() method returns the fully constructed Computer object.

**3. Advantages of Builder Pattern:**

* Handles construction of complex objects.
* Supports optional parameters flexibly.
* Improves code readability and maintainability.
* Avoids multiple constructors with many parameters (constructor telescoping problem).
* Produces immutable final object after building.

**4. Time Complexity:**  
Building object: O(1) (constant time for object creation).

**5. Real-life Examples:**

* GUI builders
* Query builders
* Configuration builders in frameworks

**Exercise 4: Implementing the Adapter Pattern**

**PaymentProcessor.java**

package mypackage;

public interface PaymentProcessor {

void processPayment(double amount);

}

**PayPal.java**

package mypackage;

public class PayPal {

public void sendPayment(double amount) {

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

}

}

**Stripe.java**

package mypackage;

public class Stripe {

public void makePayment(double amount) {

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

}

}

**Razorpay.java**

package mypackage;

public class Razorpay {

public void executePayment(double amount) {

System.out.println("Processing Razorpay payment of ₹" + amount);

}

}

### Implement Adapter Classes

**PayPalAdapter.java**

package mypackage;

public class PayPalAdapter implements PaymentProcessor {

private PayPal paypal;

public PayPalAdapter(PayPal paypal) {

this.paypal = paypal;

}

public void processPayment(double amount) {

paypal.sendPayment(amount);

}

}

**StripeAdapter.java**

package mypackage;

public class StripeAdapter implements PaymentProcessor {

private Stripe stripe;

public StripeAdapter(Stripe stripe) {

this.stripe = stripe;

}

public void processPayment(double amount) {

stripe.makePayment(amount);

}

}

**RazorpayAdapter.java**

package mypackage;

public class RazorpayAdapter implements PaymentProcessor {

private Razorpay razorpay;

public RazorpayAdapter(Razorpay razorpay) {

this.razorpay = razorpay;

}

public void processPayment(double amount) {

razorpay.executePayment(amount);

}

}

### 5. Test the Adapter Implementation

**TestAdapterPattern.java**

package mypackage;

import java.util.Scanner;

public class TestAdapterPattern {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

System.out.println("Select Payment Gateway:");

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

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

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

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

int choice = sc.nextInt();

System.out.print("Enter amount to pay (in ₹): ");

double amount = sc.nextDouble();

PaymentProcessor processor = null;

switch (choice) {

case 1:

processor = new PayPalAdapter(new PayPal());

break;

case 2:

processor = new StripeAdapter(new Stripe());

break;

case 3:

processor = new RazorpayAdapter(new Razorpay());

break;

default:

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

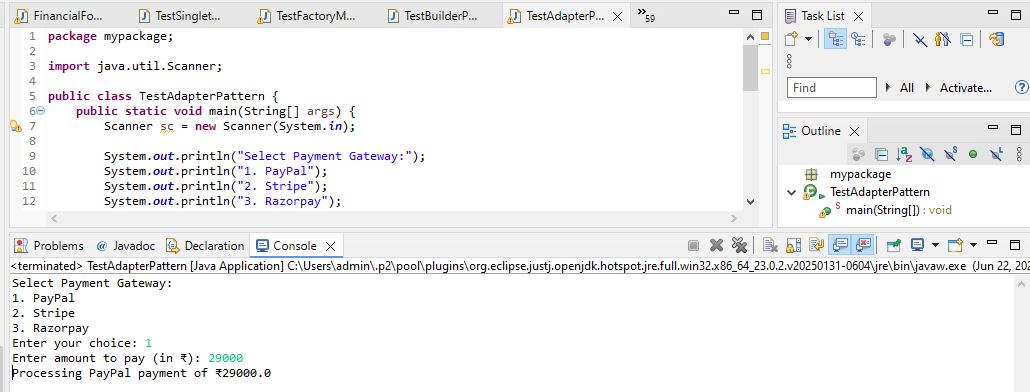
System.exit(0);

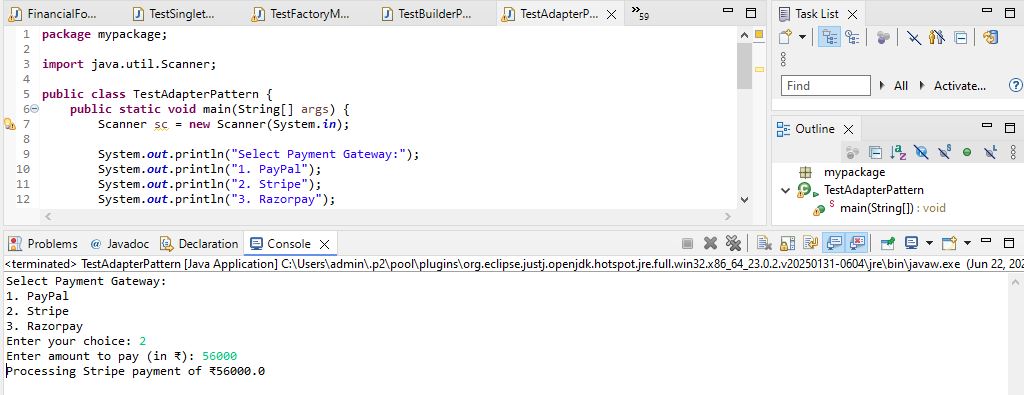
}

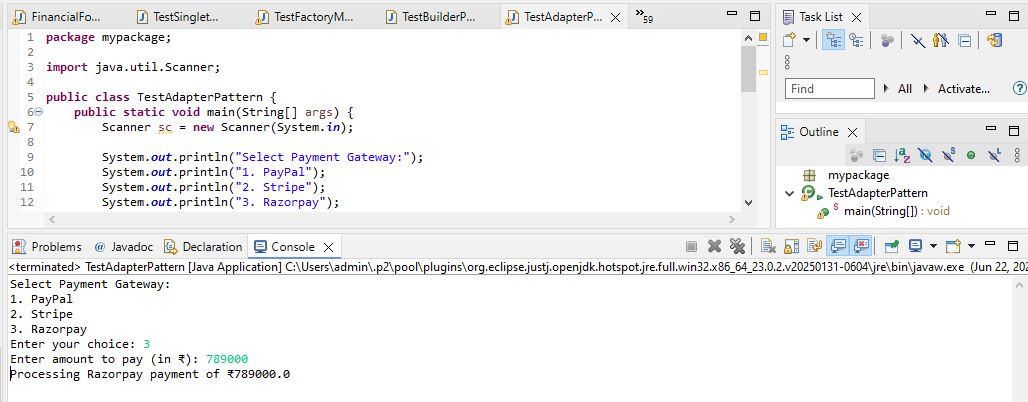
processor.processPayment(amount);

}

}







We are developing a payment processing system that needs to integrate with multiple third-party payment gateways (such as PayPal, Stripe, Razorpay), each having its own interface. To standardize interaction with these gateways, we apply the Adapter Pattern.

**1. Target Interface**

We define a common interface PaymentProcessor with the method processPayment(). This interface is used by the client code to process payments without knowing the internal implementation of each gateway.

**2. Adaptee Classes**

The existing payment gateways have their own interfaces and methods:

* PayPal class has the method sendPayment().
* Stripe class has the method makePayment().
* Razorpay class has the method executePayment().

These classes are called Adaptees because their interfaces do not match the standard PaymentProcessor interface.

**3. Adapter Classes**

For each Adaptee class, we create an Adapter class that implements the PaymentProcessor interface. These adapters internally translate the processPayment() call into the corresponding Adaptee method:

* PayPalAdapter calls sendPayment().
* StripeAdapter calls makePayment().
* RazorpayAdapter calls executePayment().

This allows the client code to use a unified interface while hiding the differences in the underlying payment gateway implementations.

**4. Advantages of Adapter Pattern**

* Allows integration of multiple incompatible systems through a common interface.
* Promotes loose coupling between the client and external libraries or APIs.
* Simplifies future extensions if new payment gateways need to be added.
* Ensures that client code remains consistent and does not require changes when integrating additional payment systems.

**5. Time Complexity**

The Adapter methods simply delegate the call to the Adaptee methods. Therefore, each call operates in constant time: O(1)

**6. Real-life Applications**

* Payment gateway integrations in e-commerce platforms.
* Legacy system integration where old and new interfaces must work together.
* Wrapping third-party libraries to conform to an internal standard interface.

**Exercise 5: Implementing the Decorator Pattern**

**Notifier.java**

**package** mypackage;

**public** **interface** Notifier {

**void** send(String message);

}

### Implement Concrete Component

**EmailNotifier.java**

**package** mypackage;

**public** **class** EmailNotifier **implements** Notifier {

@Override

**public** **void** send(String message) {

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

}

}

### Implement Decorator Classes

#### Abstract Decorator Class

**NotifierDecorator.java**

**package** mypackage;

**public** **abstract** **class** NotifierDecorator **implements** Notifier {

**protected** Notifier notifier;

**public** NotifierDecorator(Notifier notifier) {

**this**.notifier = notifier;

}

**public** **void** send(String message) {

notifier.send(message);

}

}

#### Concrete Decorator: SMS

**SMSNotifierDecorator.java**

**package** mypackage;

**public** **class** SMSNotifierDecorator **extends** NotifierDecorator {

**public** SMSNotifierDecorator(Notifier notifier) {

**super**(notifier);

}

@Override

**public** **void** send(String message) {

**super**.send(message);

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

}

}

#### Concrete Decorator: Slack

**SlackNotifierDecorator.java**

**package** mypackage;

**public** **class** SlackNotifierDecorator **extends** NotifierDecorator {

**public** SlackNotifierDecorator(Notifier notifier) {

**super**(notifier);

}

@Override

**public** **void** send(String message) {

**super**.send(message);

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

}

}

### Test the Decorator Implementation

**TestDecoratorPattern.java**

**package** mypackage;

**public** **class** TestDecoratorPattern {

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

// Base notifier - Email only

Notifier emailNotifier = **new** EmailNotifier();

// Decorate with SMS

Notifier smsNotifier = **new** SMSNotifierDecorator(emailNotifier);

// Decorate with Slack on top of SMS

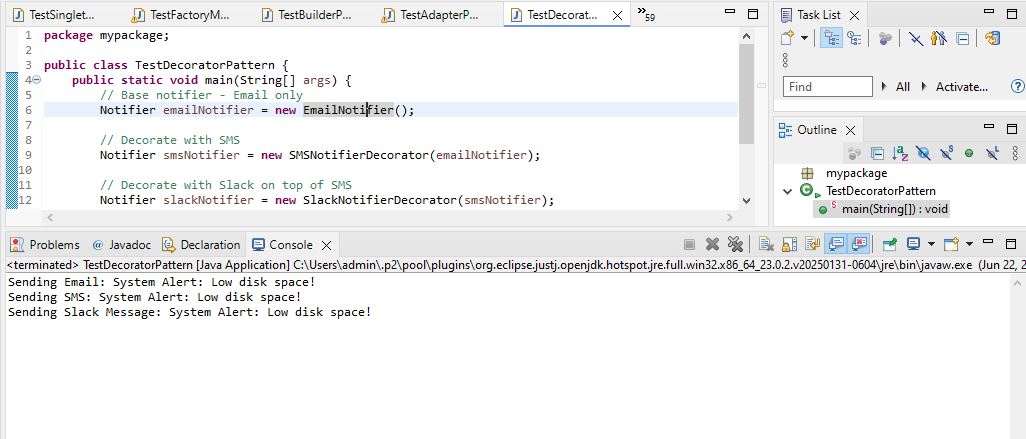
Notifier slackNotifier = **new** SlackNotifierDecorator(smsNotifier);

// Final notification will go via Email, SMS, and Slack

slackNotifier.send("System Alert: Low disk space!");

}

}



We are developing a notification system where notifications can be sent via multiple channels like Email, SMS, and Slack. The Decorator Pattern allows us to dynamically add new notification channels without modifying existing code.

**1. Component Interface**  
We define the interface Notifier with the method send(). This is the base interface that all notifiers implement.

**2. Concrete Component**  
EmailNotifier implements Notifier and provides basic email notification functionality.

**3. Decorator Classes**

* NotifierDecorator is an abstract class that implements Notifier and wraps another Notifier object.
* Concrete decorators like SMSNotifierDecorator and SlackNotifierDecorator extend NotifierDecorator and add extra behavior.

**4. Advantages of Decorator Pattern**

* Allows dynamic addition of new behaviors to existing objects at runtime.
* Promotes code reusability and flexibility.
* Follows the open-closed principle (open for extension, closed for modification).
* Avoids subclass explosion compared to inheritance-based extension.

**5. Time Complexity**

* Each decorator adds O(1) time as it simply calls the next layer of notifier.
* Total time complexity depends on the number of decorators chained.

**6. Real-life Applications**

* Notification systems (Email, SMS, Push, Slack, etc.).
* I/O Streams in Java (BufferedReader, BufferedInputStream are decorators).
* UI components where multiple decorations (border, color, effects) are added.

**Exercise 6: Implementing the Proxy Pattern**

**Image.java**

**package** mypackage;

**public** **interface** Image {

**void** display();

}

**Implement Real Subject Class**

**RealImage.java**

**package** mypackage;

**public** **class** RealImage **implements** Image {

**private** String fileName;

**public** RealImage(String fileName) {

**this**.fileName = fileName;

loadFromRemoteServer();

}

**private** **void** loadFromRemoteServer() {

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

}

**public** **void** display() {

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

}

}

**Implement Proxy Class**

**ProxyImage.java**

**package** mypackage;

**public** **class** ProxyImage **implements** Image {

**private** RealImage realImage;

**private** String fileName;

**public** ProxyImage(String fileName) {

**this**.fileName = fileName;

}

**public** **void** display() {

**if** (realImage == **null**) {

realImage = **new** RealImage(fileName); // Lazy initialization

} **else** {

System.***out***.println("Image already loaded from cache.");

}

realImage.display();

}

}

**Test the Proxy Implementation**

**TestProxyPattern.java**

**package** mypackage;

**public** **class** TestProxyPattern {

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

Image image = **new** ProxyImage("nature\_photo.jpg");

// Image will be loaded from remote server only once

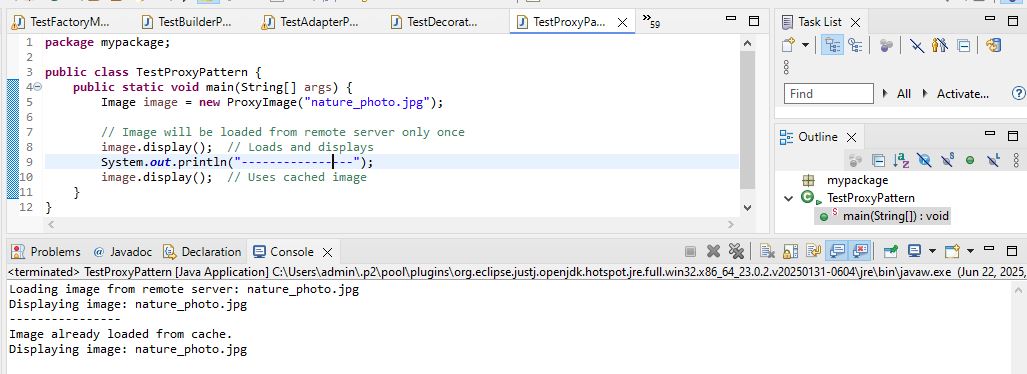
image.display(); // Loads and displays

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

image.display(); // Uses cached image

}

}

  
We are developing an image viewer application that loads images from a remote server. Loading large images takes time and resources. To optimize performance, we apply the Proxy Pattern.

**1. Subject Interface**

The Image interface defines a common method display() which both the real image and proxy implement.

**2. Real Subject Class**

RealImage implements Image and loads the image from the remote server when it is instantiated.

* Actual loading happens inside the constructor.
* Displays image when display() is called.

**3. Proxy Class**

ProxyImage also implements Image but delays creation of RealImage until display() is first called (lazy initialization).

* Caches the loaded image to avoid repeated loading.

**4. Advantages of Proxy Pattern**

* Provides lazy loading to avoid unnecessary resource usage.
* Reduces memory usage and improves performance.
* Controls access to the real object.
* Can add additional functionality like access control, logging, and caching.

**5. Time Complexity**

* First call: O(1) for proxy call + time for real image loading.
* Subsequent calls: O(1), as image is loaded from cache.

**6. Real-life Applications**

* Virtual Proxies for large resource loading (images, videos, files).
* Remote Proxies to control access to remote objects (e.g., RPC, web services).
* Protection Proxies for access control and security.

**Exercise 7: Implementing the Observer Pattern**

**Define Subject Interface**

**Stock.java**

**package** mypackage;

**public** **interface** Stock {

**void** registerObserver(Observer observer);

**void** deregisterObserver(Observer observer);

**void** notifyObservers();

**void** setPrice(**double** price);

}

**Implement Concrete Subject**

**StockMarket.java**

**package** mypackage;

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** StockMarket **implements** Stock {

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

**private** **double** price;

**public** **void** registerObserver(Observer observer) {

observers.add(observer);

}

**public** **void** deregisterObserver(Observer observer) {

observers.remove(observer);

}

**public** **void** notifyObservers() {

**for** (Observer observer : observers) {

observer.update(price);

}

}

**public** **void** setPrice(**double** price) {

**this**.price = price;

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

notifyObservers();

}

}

**Define Observer Interface**

**Observer.java**

**package** mypackage;

**public** **interface** Observer {

**void** update(**double** price);

}

**Implement Concrete Observers**

**package** mypackage;

**public** **class** MobileApp **implements** Observer {

**private** String appName;

**public** MobileApp(String appName) {

**this**.appName = appName;

}

**public** **void** update(**double** price) {

System.***out***.println(appName + " (Mobile App) received stock price update: " + price);

}

}

**WebApp.java**

**package** mypackage;

**public** **class** WebApp **implements** Observer {

**private** String appName;

**public** WebApp(String appName) {

**this**.appName = appName;

}

**public** **void** update(**double** price) {

System.***out***.println(appName + " (Web App) received stock price update: " + price);

}

}

**Test the Observer Implementation**

**TestObserverPattern.java**

**package** mypackage;

**public** **class** TestObserverPattern {

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

StockMarket stockMarket = **new** StockMarket();

Observer mobileObserver = **new** MobileApp("StockMonitor");

Observer webObserver = **new** WebApp("StockWebPortal");

stockMarket.registerObserver(mobileObserver);

stockMarket.registerObserver(webObserver);

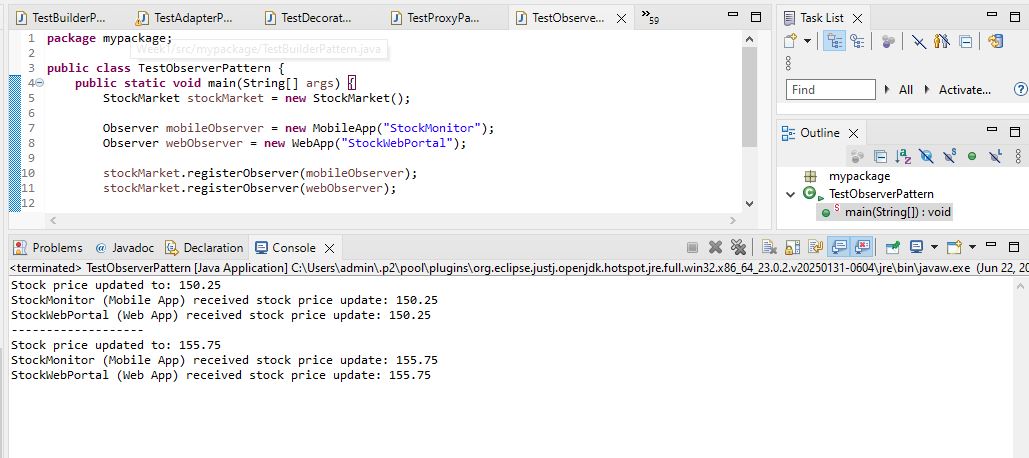
stockMarket.setPrice(150.25);

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

stockMarket.setPrice(155.75);

}

}

****

We are developing a stock market monitoring application where multiple clients need to receive stock price updates automatically. The Observer Pattern allows efficient notification to all registered observers when the stock price changes.

**1. Subject Interface**

The Stock interface defines methods for:

* registerObserver(): to add observers.
* deregisterObserver(): to remove observers.
* notifyObservers(): to notify all observers.
* setPrice(): to update the stock price.

**2. Concrete Subject**

The StockMarket class implements Stock and:

* Maintains a list of registered observers.
* Calls notifyObservers() whenever stock price changes.

**3. Observer Interface**

The Observer interface declares the method update() which is called by the subject.

**4. Concrete Observers**

* MobileApp and WebApp implement Observer.
* Each displays stock price updates when notified.

**5. Advantages of Observer Pattern**

* Allows dynamic addition/removal of observers.
* Promotes loose coupling between subject and observers.
* Automatic update of multiple components on state change.
* Scales easily as the number of observers grows.

**6. Time Complexity**

* Register/deregister: O(1)
* Notify observers: O(n), where n is the number of observers.

**7. Real-life Applications**

* Stock market monitoring systems.
* Event handling systems.
* GUI frameworks (event listeners).
* Social media notifications.

**Exercise 8: Implementing the Strategy Pattern**

**PaymentStrategy.java**

**package** mypackage;

**public** **interface** PaymentStrategy {

**void** pay(**double** amount);

}

**Implement Concrete Strategies**

**CreditCardPayment.java**

**package** mypackage;

**public** **class** CreditCardPayment **implements** PaymentStrategy {

**private** String cardNumber;

**private** String cardHolderName;

**public** CreditCardPayment(String cardNumber, String cardHolderName) {

**this**.cardNumber = cardNumber;

**this**.cardHolderName = cardHolderName;

}

**public** **void** pay(**double** amount) {

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

}

}

**PayPalPayment.java**

**package** mypackage;

**public** **class** PayPalPayment **implements** PaymentStrategy {

**private** String email;

**public** PayPalPayment(String email) {

**this**.email = email;

}

**public** **void** pay(**double** amount) {

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

}

}

**Implement Context Class**

**PaymentContext.java**

**package** mypackage;

**public** **class** PaymentContext {

**private** PaymentStrategy paymentStrategy;

**public** **void** setPaymentStrategy(PaymentStrategy paymentStrategy) {

**this**.paymentStrategy = paymentStrategy;

}

**public** **void** payAmount(**double** amount) {

**if** (paymentStrategy == **null**) {

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

**return**;

}

paymentStrategy.pay(amount);

}

}

**Test the Strategy Implementation with User Input**

**TestStrategyPattern.java**

**package** mypackage;

**import** java.util.Scanner;

**public** **class** TestStrategyPattern {

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

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

PaymentContext context = **new** PaymentContext();

System.***out***.println("Enter payment amount:");

**double** amount = sc.nextDouble();

sc.nextLine(); // Consume newline

System.***out***.println("Select payment method:");

System.***out***.println("1. Credit Card");

System.***out***.println("2. PayPal");

**int** choice = sc.nextInt();

sc.nextLine(); // Consume newline

**switch** (choice) {

**case** 1:

System.***out***.println("Enter Card Number:");

String cardNumber = sc.nextLine();

System.***out***.println("Enter Card Holder Name:");

String cardHolder = sc.nextLine();

context.setPaymentStrategy(**new** CreditCardPayment(cardNumber, cardHolder));

**break**;

**case** 2:

System.***out***.println("Enter PayPal Email:");

String email = sc.nextLine();

context.setPaymentStrategy(**new** PayPalPayment(email));

**break**;

**default**:

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

sc.close();

**return**;

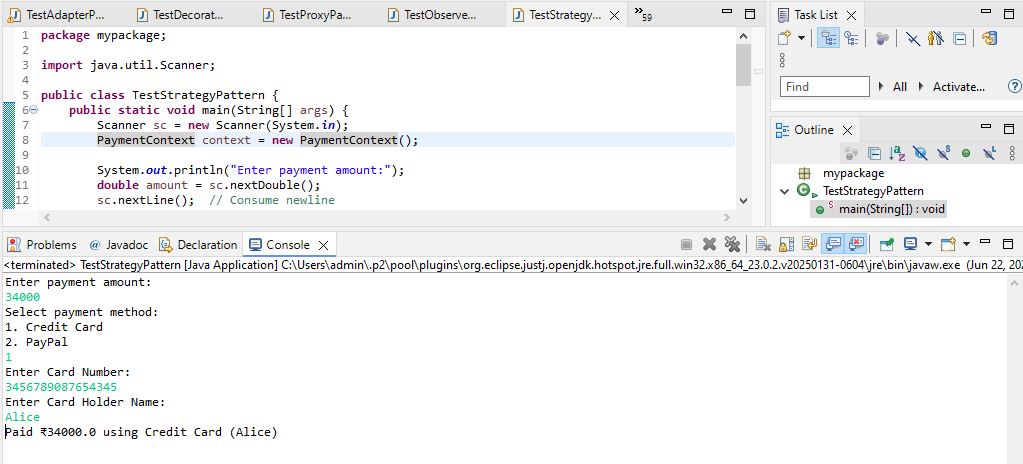
}

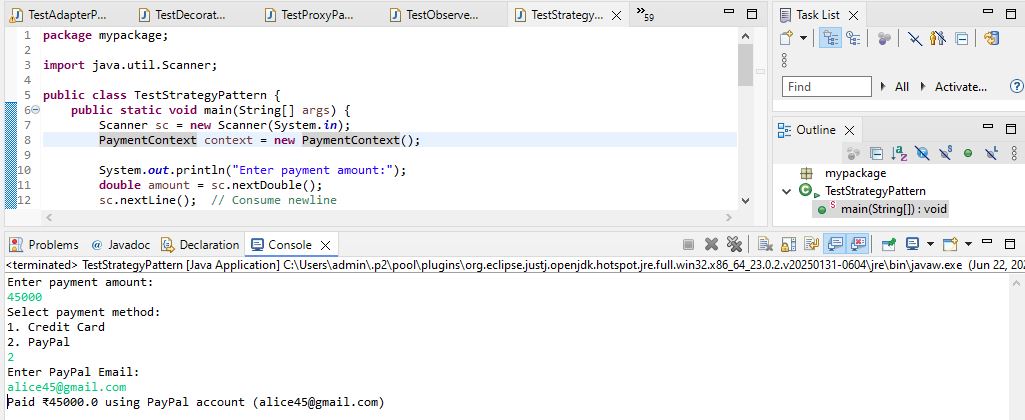
context.payAmount(amount);

sc.close();

}

}

****

****

We are developing a payment system that supports multiple payment methods like Credit Card and PayPal. The payment method is selected at runtime. We apply the Strategy Pattern to solve this.

**1. Strategy Interface**

* PaymentStrategy defines a common method pay().

**2. Concrete Strategies**

* CreditCardPayment implements PaymentStrategy and handles credit card payments.
* PayPalPayment implements PaymentStrategy and handles PayPal payments.

**3. Context Class**

* PaymentContext holds a reference to PaymentStrategy and delegates the payAmount() call to the selected strategy.

**4. Advantages of Strategy Pattern**

* Allows dynamic selection of algorithms at runtime.
* Promotes flexibility and easy extension of new payment methods.
* Removes conditional logic from the client code.
* Promotes open-closed principle (easy to add new strategies without modifying existing code).

**5. Time Complexity**

* O(1) for selecting and executing any strategy.

**6. Real-life Applications**

* Payment systems.
* Compression algorithms.
* Sorting algorithms selection.
* File format converters.

**Exercise 9: Implementing the Command Pattern**

**Command.java**

**package** mypackage;

**public** **interface** Command {

**void** execute();

}

**Implement Receiver Class**

**Light.java**

**package** mypackage;

**public** **class** Light {

**public** **void** turnOn() {

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

}

**public** **void** turnOff() {

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

}

}

**Implement Concrete Commands**

**LightOnCommand.java**

**package** mypackage;

**public** **class** LightOnCommand **implements** Command {

**private** Light light;

**public** LightOnCommand(Light light) {

**this**.light = light;

}

**public** **void** execute() {

light.turnOn();

}

}

**LightOffCommand.java**

**package** mypackage;

**public** **class** LightOffCommand **implements** Command {

**private** Light light;

**public** LightOffCommand(Light light) {

**this**.light = light;

}

**public** **void** execute() {

light.turnOff();

}

}

**Implement Invoker Class**

**RemoteControl.java**

**package** mypackage;

**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 assigned.");

}

}

}

**Test the Command Implementation**

**TestCommandPattern.java**

**package** mypackage;

**import** java.util.Scanner;

**public** **class** TestCommandPattern {

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

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

Light light = **new** Light();

RemoteControl remote = **new** RemoteControl();

System.***out***.println("Home Automation - Command Pattern");

System.***out***.println("1. Turn ON Light");

System.***out***.println("2. Turn OFF Light");

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

**int** choice = sc.nextInt();

**if** (choice == 1) {

remote.setCommand(**new** LightOnCommand(light));

} **else** **if** (choice == 2) {

remote.setCommand(**new** LightOffCommand(light));

} **else** {

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

sc.close();

**return**;

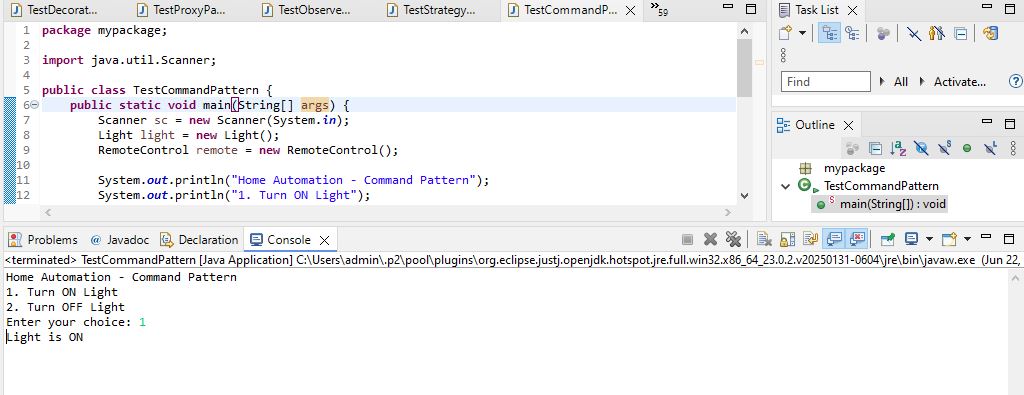
}

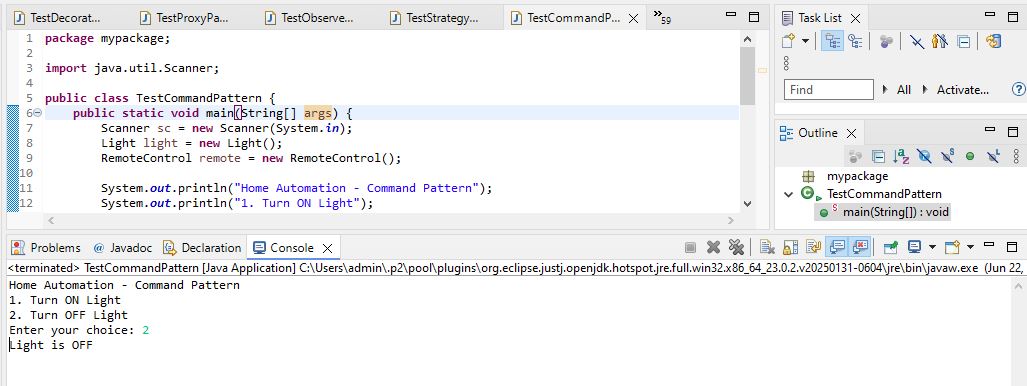
remote.pressButton();

sc.close();

}

}





We are developing a home automation system where commands can be issued to turn devices on or off. We use the Command Pattern to encapsulate requests as objects.

**1. Components**

* **Command Interface:** Declares execute() method.
* **Concrete Commands:** LightOnCommand and LightOffCommand implement Command and call the corresponding methods on Light.
* **Receiver:** The Light class performs actual operations (turnOn() and turnOff()).
* **Invoker:** RemoteControl stores and executes commands.
* **Client:** The main method creates commands and assigns them to the invoker.

**2. Advantages of Command Pattern**

* Decouples sender (Invoker) from receiver.
* Supports undo/redo functionality.
* Commands can be queued, logged, or scheduled.
* Easy to add new commands without modifying existing code (Open-Closed Principle).

**3. Time Complexity**

* Setting command: O(1)
* Executing command: O(1)

**4. Real-Life Applications**

* Home automation systems.
* Transaction management.
* Task scheduling systems.
* GUI button actions (like Undo, Redo).

**Exercise 10: Implementing the MVC Pattern**

**Student.java**

**package** mypackage;

**public** **class** Student {

**private** String name;

**private** String id;

**private** String grade;

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

**this**.name = name;

**this**.id = id;

**this**.grade = grade;

}

// Getters

**public** String getName() { **return** name; }

**public** String getId() { **return** id; }

**public** String getGrade() { **return** grade; }

// Setters

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

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

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

}

**Define View Class**

**StudentView.java**

**package** mypackage;

**public** **class** StudentView {

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

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

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

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

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

}

}

**Define Controller Class**

**StudentController.java**

**package** mypackage;

**public** **class** StudentController {

**private** Student model;

**private** StudentView view;

**public** StudentController(Student model, StudentView view) {

**this**.model = model;

**this**.view = view;

}

// Setters for updating model

**public** **void** setStudentName(String name) { model.setName(name); }

**public** **void** setStudentId(String id) { model.setId(id); }

**public** **void** setStudentGrade(String grade) { model.setGrade(grade); }

// Getters for retrieving data

**public** String getStudentName() { **return** model.getName(); }

**public** String getStudentId() { **return** model.getId(); }

**public** String getStudentGrade() { **return** model.getGrade(); }

// Update View

**public** **void** updateView() {

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

}

}

**Test the MVC Implementation**

**TestMVCPattern.java**

**package** mypackage;

**import** java.util.Scanner;

**public** **class** TestMVCPattern {

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

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

// Create initial model

Student student = **new** Student("Default Name", "0000", "A");

StudentView view = **new** StudentView();

StudentController controller = **new** StudentController(student, view);

controller.updateView(); // Display default details

// User input to update student details

System.***out***.println("\nEnter new student details:");

System.***out***.print("Name: ");

String name = sc.nextLine();

System.***out***.print("ID: ");

String id = sc.nextLine();

System.***out***.print("Grade: ");

String grade = sc.nextLine();

// Update model via controller

controller.setStudentName(name);

controller.setStudentId(id);

controller.setStudentGrade(grade);

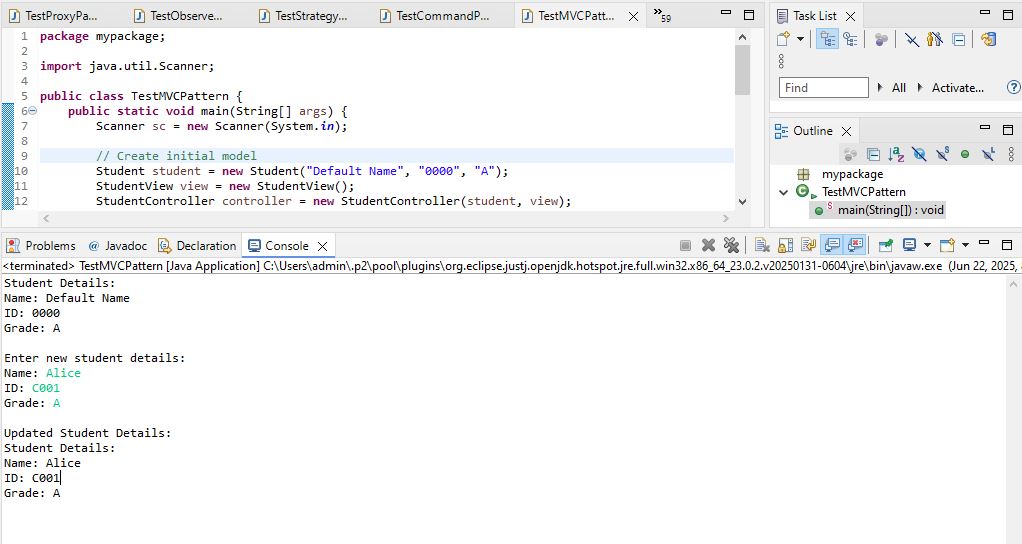
System.***out***.println("\nUpdated Student Details:");

controller.updateView();

sc.close();

}

}



We are developing a student management system that follows the MVC design pattern to separate concerns between data, UI, and logic.

**1. Components**

* **Model (Student):**  
  Holds the data of the student (name, id, grade).
* **View (StudentView):**  
  Responsible for displaying student details.
* **Controller (StudentController):**  
  Handles user input, updates the model, and refreshes the view.

**2. Advantages of MVC Pattern**

* Separates business logic from UI.
* Makes the system easier to maintain and scale.
* Allows independent development of model, view, and controller.
* Improves testability.

**3. Time Complexity**

* Updating model: O(1)
* Displaying view: O(1)

**4. Real-Life Applications**

* Web applications (Spring MVC, ASP.NET MVC)
* GUI applications (Java Swing MVC)
* Mobile applications (Android MVC)

**Exercise 11: Implementing Dependency Injection**

**CustomerRepository.java**

**package** mypackage;

**public** **interface** CustomerRepository {

String findCustomerById(String id);

}

**Implement Concrete Repository**

**CustomerRepositoryImpl.java**

**package** mypackage;

**import** java.util.HashMap;

**import** java.util.Map;

**public** **class** CustomerRepositorylmpl **implements** CustomerRepository {

**private** Map<String, String> customerData;

**public** CustomerRepositorylmpl() {

customerData = **new** HashMap<>();

customerData.put("C001", "Alice");

customerData.put("C002", "Bob");

customerData.put("C003", "Charlie");

}

@Override

**public** String findCustomerById(String id) {

**return** customerData.getOrDefault(id, "Customer Not Found");

}

}

**Define Service Class**

**CustomerService.java**

**package** mypackage;

**public** **class** CustomerService {

**private** CustomerRepository customerRepository;

// Constructor Injection

**public** CustomerService(CustomerRepository customerRepository) {

**this**.customerRepository = customerRepository;

}

**public** **void** getCustomerDetails(String id) {

String customerName = customerRepository.findCustomerById(id);

System.***out***.println("Customer ID: " + id + ", Name: " + customerName);

}

}

**Test Dependency Injection Implementation**

**TestDependencyInjection.java**

**package** mypackage;

**import** java.util.Scanner;

**public** **class** TestDependencyInjection {

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

// Create repository implementation

CustomerRepository repository = **new** CustomerRepositorylmpl();

// Inject repository into service using constructor injection

CustomerService service = **new** CustomerService(repository);

// Get user input

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

System.***out***.print("Enter Customer ID to search: ");

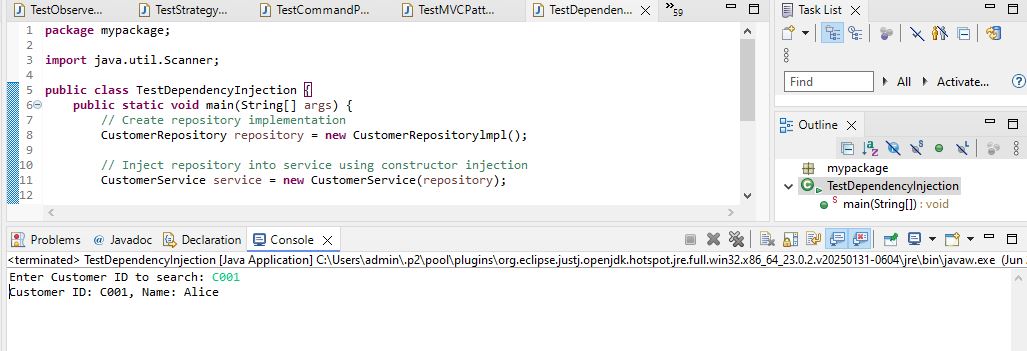
String id = sc.nextLine();

service.getCustomerDetails(id);

sc.close();

}

}

****

We are developing a customer management application where CustomerService depends on CustomerRepository to retrieve customer data. We use Dependency Injection to manage dependencies.

**1. What is Dependency Injection?**

Dependency Injection (DI) is a design pattern where the dependencies of a class are provided from the outside rather than created inside the class. This promotes loose coupling and improves flexibility and testability.

**2. Types of Dependency Injection**

* **Constructor Injection (used in this code):** Dependencies are injected via class constructor.
* **Setter Injection:** Dependencies are injected via setter methods.
* **Field Injection:** Dependencies are injected directly into fields (common in frameworks like Spring).

**3. Advantages of Dependency Injection**

* Reduces coupling between classes.
* Easier to test (mock dependencies during testing).
* Easier to maintain and extend.
* Improves code reusability.

**4. Time Complexity**

* Fetching customer data: O(1) (using HashMap)

**5. Real-Life Applications**

* Used extensively in frameworks like Spring, Angular.
* Web services, enterprise applications, microservices.