**Exercise 1: Implementing the Singleton Pattern**

**Scenario:**

You need to ensure that a logging utility class in your application has only one instance throughout the application lifecycle to ensure consistent logging.

**Code**

*Logger.java*

package singleton;

public class Logger {

    private static Logger instance;

    private Logger() {

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

    }

    // Public method to get the single instance

    public static Logger getInstance() {

        if (instance == null) {

            instance = new Logger();

        }

        return instance;

    }

    public void log(String message) {

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

    }

}

*SingletonTest.java*

package singleton;

public class SingletonTest {

    public static void main(String[] args) {

        Logger logger1 = Logger.getInstance();

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

        Logger logger2 = Logger.getInstance();

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

  if (logger1 == logger2) {

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

        } else {

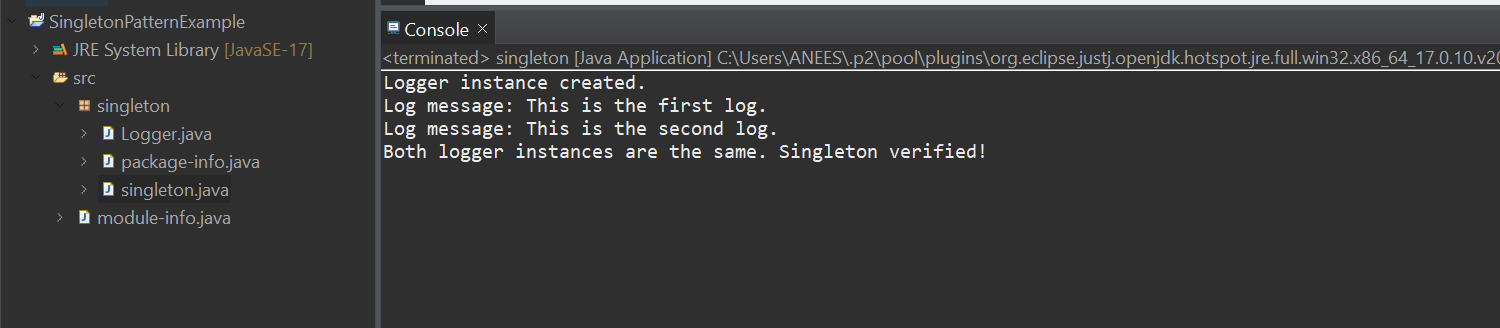
            System.out.println("Logger instances are different. Singleton failed!");

        }

    }

}

**Output**



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

**Scenario:**

You are developing a document management system that needs to create different types of documents (e.g., Word, PDF, Excel). Use the Factory Method Pattern to achieve this.

**Code**

*Main.java:*

package documentfactory;

public class Main {

public static void main(String[] args) {

DocumentFactory wordFactory = new WordDocumentFactory();

Document wordDoc = wordFactory.createDocument();

wordDoc.open();

DocumentFactory pdfFactory = new PdfDocumentFactory();

Document pdfDoc = pdfFactory.createDocument();

pdfDoc.open();

DocumentFactory excelFactory = new ExcelDocumentFactory();

Document excelDoc = excelFactory.createDocument();

excelDoc.open();

}

}

*Document.java*

package documentfactory;  
public interface Document {  
 void open();  
}

*WordDocument.java*

package documentfactory;  
public class WordDocument implements Document {  
 public void open() {  
 System.out.println("Opening Word document...");  
 }  
}

*PdfDocument.java*

package documentfactory;  
public class PdfDocument implements Document {  
 public void open() {  
 System.out.println("Opening PDF document...");  
 }  
}

*ExcelDocument.java*

package documentfactory;  
public class ExcelDocument implements Document {  
 public void open() {  
 System.out.println("Opening Excel document...");  
 }  
}

*DocumentFactory.java*

package documentfactory;  
public abstract class DocumentFactory {  
 public abstract Document createDocument();  
}

*WordDocumentFactory.java*

package documentfactory;  
public class WordDocumentFactory extends DocumentFactory {  
 public Document createDocument() {  
 return new WordDocument();  
 }  
}

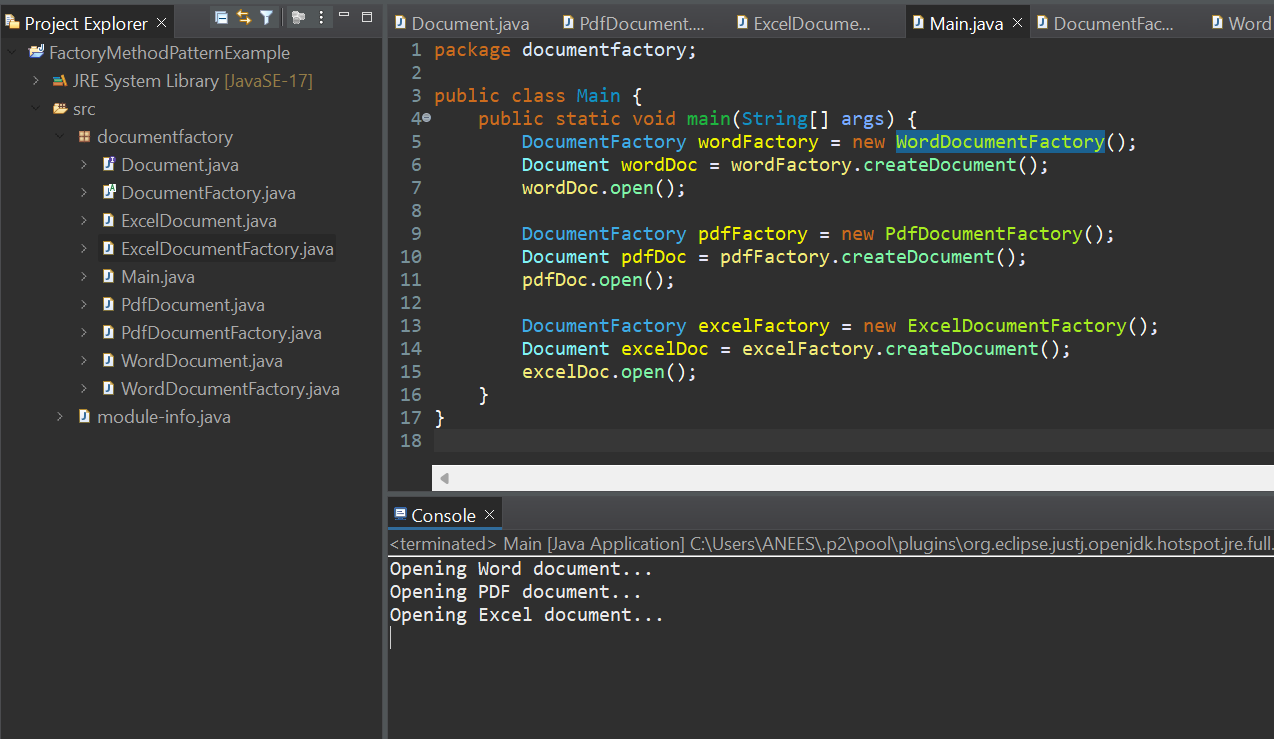
*PdfDocumentFactory.java*

package documentfactory;  
public class PdfDocumentFactory extends DocumentFactory {  
 public Document createDocument() {  
 return new PdfDocument();  
 }  
}

*ExcelDocumentFactory.java*

package documentfactory;  
public class ExcelDocumentFactory extends DocumentFactory {  
 public Document createDocument() {  
 return new ExcelDocument();  
 }  
}

**Output**

****

**Exercise 3: Implementing the Builder Pattern**

**Scenario:**

You are developing a system to create complex objects such as a Computer with multiple optional parts. Use the Builder Pattern to manage the construction process.

**Code**

*Computer.java*

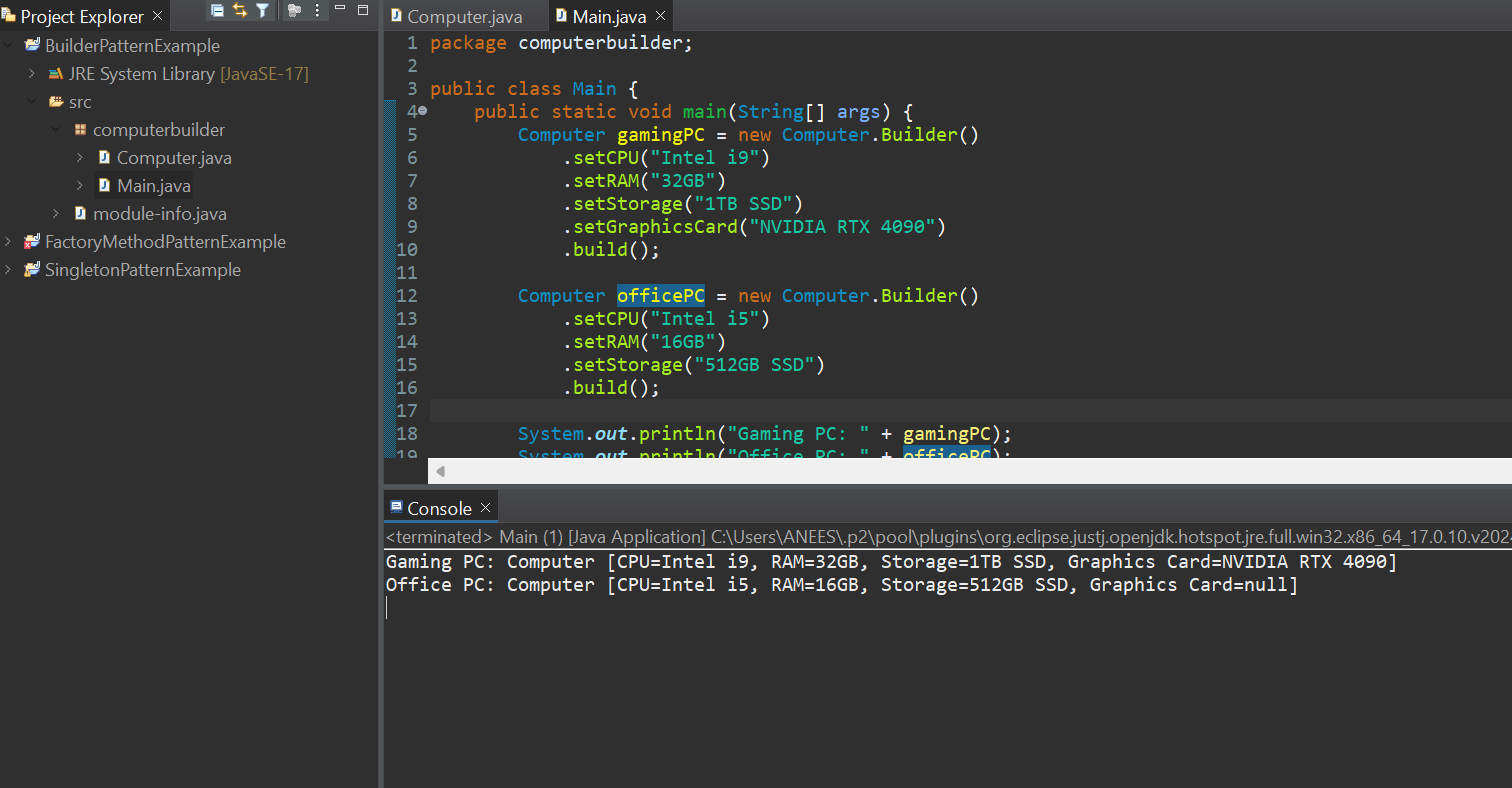
package computerbuilder;  
public class Computer {  
 private String CPU;  
 private String RAM;  
 private String storage;  
 private String graphicsCard;  
 private Computer(Builder builder) {  
 this.CPU = builder.CPU;  
 this.RAM = builder.RAM;  
 this.storage = builder.storage;  
 this.graphicsCard = builder.graphicsCard;  
 }  
 public static class Builder {  
 private String CPU;  
 private String RAM;

private String storage;  
 private String graphicsCard;  
 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 setGraphicsCard(String graphicsCard) {  
 this.graphicsCard = graphicsCard;  
 return this;  
 }  
 public Computer build() {  
 return new Computer(this);  
 }  
 }  
 @Override  
 public String toString() {  
 return "Computer [CPU=" + CPU + ", RAM=" + RAM + ", Storage=" + storage + ", Graphics Card=" + graphicsCard + "]";  
 }  
}

*Main.java*

package computerbuilder;  
public class Main {  
 public static void main(String[] args) {  
 Computer gamingPC = new Computer.Builder()  
 .setCPU("Intel i9")  
 .setRAM("32GB")  
 .setStorage("1TB SSD")  
 .setGraphicsCard("NVIDIA RTX 4090")  
 .build();  
 Computer officePC = new Computer.Builder()  
 .setCPU("Intel i5")  
 .setRAM("16GB")  
 .setStorage("512GB SSD")  
 .build();  
 System.out.println("Gaming PC: " + gamingPC);  
 System.out.println("Office PC: " + officePC);  
 }  
}

**Output**

****

**Exercise 4: Implementing the Adapter Pattern**

**Scenario:**

You are developing a payment processing system that needs to integrate with multiple third-party payment gateways with different interfaces. Use the Adapter Pattern to achieve this.

**Code**

*Main.java*

package paymentadapter;  
public class Main {  
 public static void main(String[] args) {  
 PaymentProcessor paypalProcessor = new PayPalAdapter(new PayPalGateway());  
 paypalProcessor.processPayment(1500);  
 PaymentProcessor stripeProcessor = new StripeAdapter(new StripeGateway());  
 stripeProcessor.processPayment(2500);  
 }  
}

*PaymentProcessor.java*

package paymentadapter;  
public interface PaymentProcessor {  
 void processPayment(double amount);  
}

*PayPalGateway.java*

package paymentadapter;  
public class PayPalGateway {  
 public void sendPayment(double amount) {  
 System.out.println("Processing payment via PayPal: Rs." + amount);  
 }  
}

*StripeGateway.java*

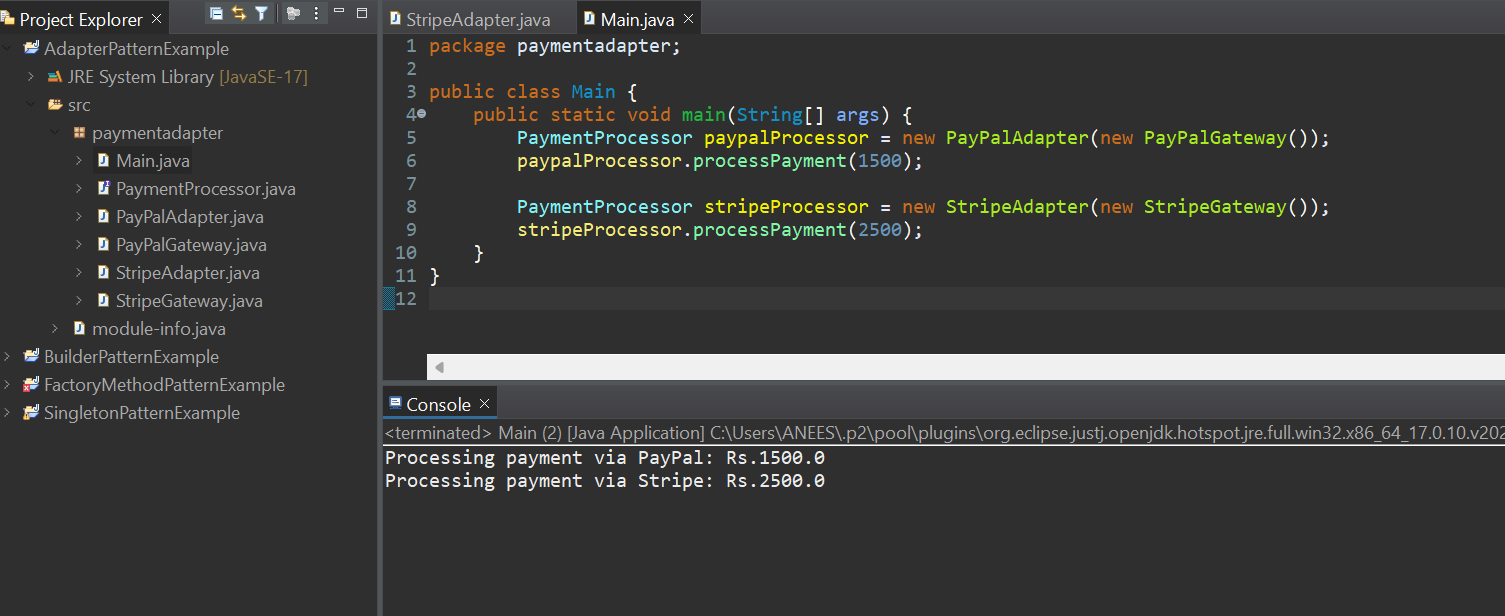
package paymentadapter;  
public class StripeGateway {  
 public void makePayment(double amount) {  
 System.out.println("Processing payment via Stripe: Rs." + amount);  
 }  
}

*PayPalAdapter.java*

package paymentadapter;  
public class PayPalAdapter implements PaymentProcessor {  
 private PayPalGateway paypal;  
 public PayPalAdapter(PayPalGateway paypal) {  
 this.paypal = paypal;  
 }  
 @Override  
 public void processPayment(double amount) {  
 paypal.sendPayment(amount);  
 }} *StripeAdapter.java*

package paymentadapter;  
public class StripeAdapter implements PaymentProcessor {  
 private StripeGateway stripe;  
  
 public StripeAdapter(StripeGateway stripe) {  
 this.stripe = stripe;  
 }  
 @Override  
 public void processPayment(double amount) {  
 stripe.makePayment(amount);  
 }}

**Output**

****

**Exercise 5: Implementing the Decorator Pattern**

**Scenario:**

You are developing a notification system where notifications can be sent via multiple channels (e.g., Email, SMS). Use the Decorator Pattern to add functionalities dynamically.

**Code**

*Main.java*

package notificationdecorator;  
public class Main {

public static void main(String[] args) {  
 Notifier baseNotifier = new EmailNotifier();  
 Notifier smsNotifier = new SMSNotifierDecorator(baseNotifier);  
 Notifier slackNotifier = new SlackNotifierDecorator(smsNotifier);  
 slackNotifier.send("Server is down!");  
 }  
}

*Notifier.java*

package notificationdecorator;

public interface Notifier {  
 void send(String message);  
}

*EmailNotifier.java*

package notificationdecorator;  
public class EmailNotifier implements Notifier {  
 public void send(String message) {  
 System.out.println("Sending Email: " + message);  
 }  
}

*NotifierDecorator.java*

package notificationdecorator;  
public abstract class NotifierDecorator implements Notifier {  
 protected Notifier notifier;  
 public NotifierDecorator(Notifier notifier) {  
 this.notifier = notifier;  
 }  
 public void send(String message) {  
 notifier.send(message);  
 }  
}

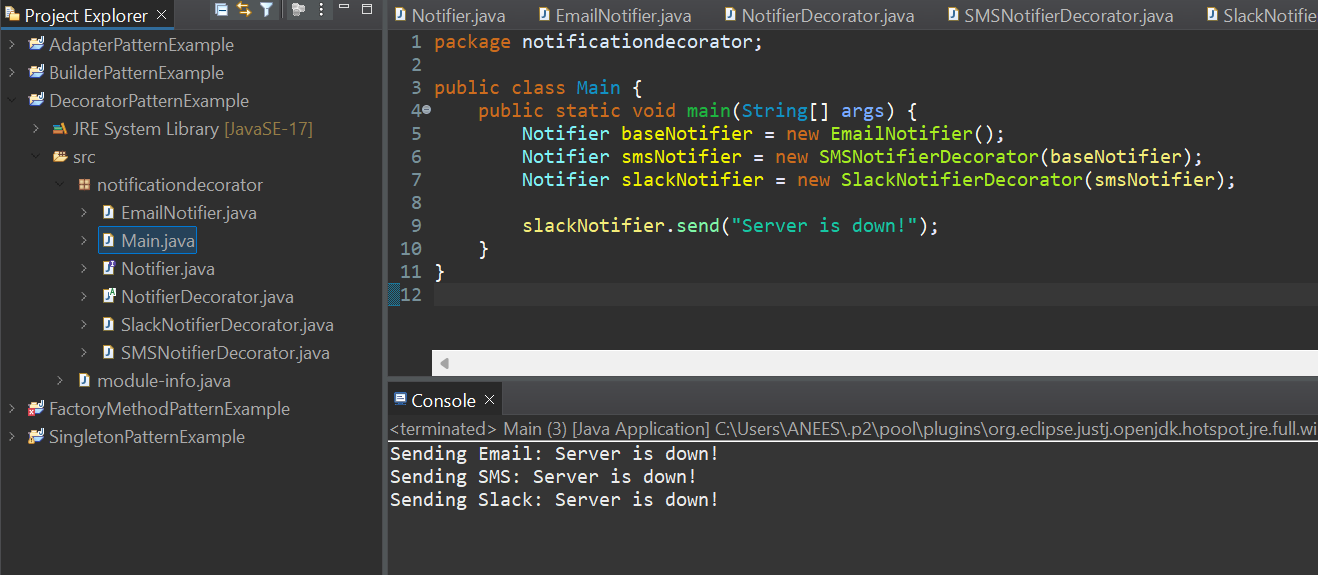
*SMSNotifierDecorator.java*

package notificationdecorator;  
public class SMSNotifierDecorator extends NotifierDecorator {  
 public SMSNotifierDecorator(Notifier notifier) {  
 super(notifier);  
 }  
 public void send(String message) {  
 super.send(message);  
 System.out.println("Sending SMS: " + message);  
 }  
}

*SlackNotifierDecorator.java*

package notificationdecorator;  
public class SlackNotifierDecorator extends NotifierDecorator {  
 public SlackNotifierDecorator(Notifier notifier) {  
 super(notifier);  
 }  
 public void send(String message) {  
 super.send(message);  
   
 System.out.println("Sending Slack: " + message);  
 }  
}

**Output**



**Exercise 6: Implementing the Proxy Pattern**

**Scenario:**

You are developing an image viewer application that loads images from a remote server. Use the Proxy Pattern to add lazy initialization and caching.

**Code**

*Image.java*

package com.viewer.proxy;

public interface Image {

void display();

}

*RealImage.java*

package com.viewer.proxy;

public class RealImage implements Image {

private String fileName;

public RealImage(String fileName) {

this.fileName = fileName;

loadFromServer(fileName);

}

private void loadFromServer(String file) {

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

}

*@Override*

public void display() {

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

}}

*ProxyImage.java*

package com.viewer.proxy;

public class ProxyImage implements Image {

private RealImage realImage;

private String fileName;

public ProxyImage(String fileName) {

this.fileName = fileName;

}

*@Override*

public void display() {

// Lazy initialization

if (realImage == null) {

realImage = new RealImage(fileName);

} else {

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

}

realImage.display();

}

}

*Main.java*

package com.viewer.proxy;

public class Main {

public static void main(String[] args) {

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

Image image2 = new ProxyImage("photo2.png");

// First time: loads from server

image1.display();

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

// Second time: uses cache

image1.display();

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

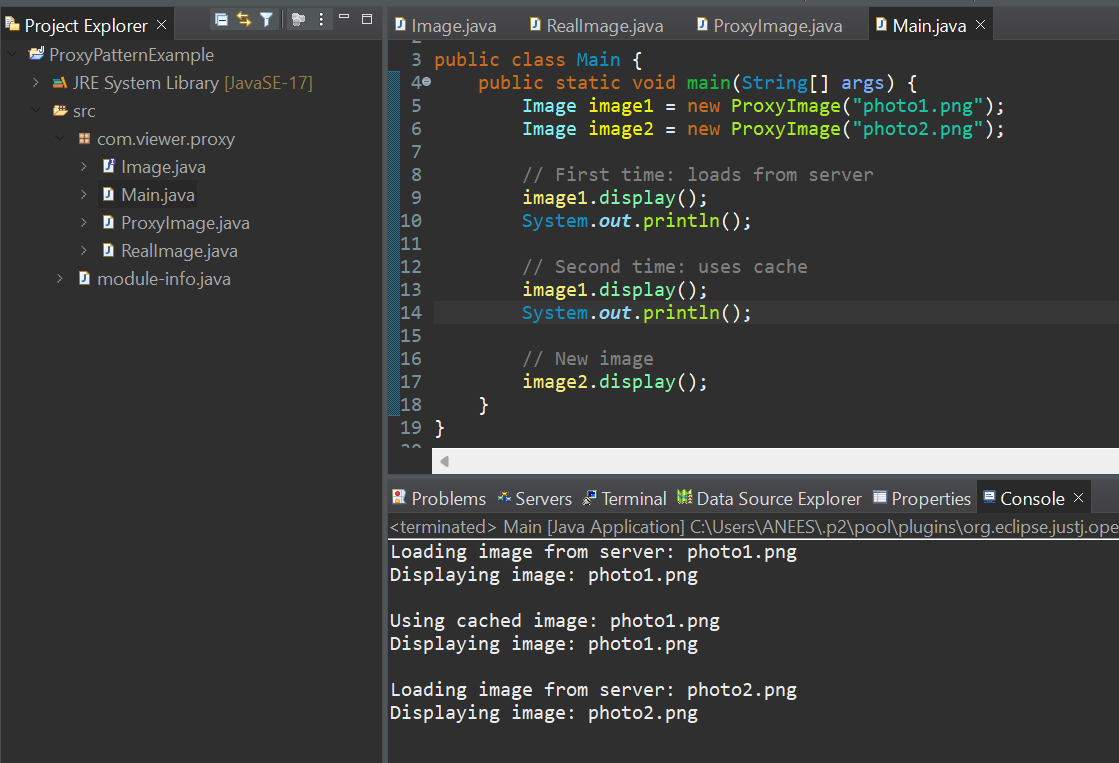
// New image

image2.display();

}

}

**Output**

****

**Exercise 7: Implementing the Observer Pattern**

**Scenario:**

You are developing a stock market monitoring application where multiple clients need to be notified whenever stock prices change. Use the Observer Pattern to achieve this.

**Code**

*Observer.java*

package com.observerpattern;

public interface Observer {

void update(String stockName, double newPrice);}

*Stock.java*

package com.observerpattern;

public interface Stock {

void registerObserver(Observer o);

void removeObserver(Observer o);

void notifyObservers();

}

*StockMarket.java*

package com.observerpattern;

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(String stockName, double initialPrice) {

this.stockName = stockName;

this.stockPrice = initialPrice;

this.observers = new ArrayList<>();

}

*@Override*

public void registerObserver(Observer o) {

observers.add(o);

}

*@Override*

public void removeObserver(Observer o) {

observers.remove(o);

}

*@Override*

public void notifyObservers() {

for (Observer o : observers) {

o.update(stockName, stockPrice);

}

}

// Trigger price change and notify all observers

public void setStockPrice(double newPrice) {

this.stockPrice = newPrice;

notifyObservers();

}

}

*MobileApp.java*

package com.observerpattern;

public class MobileApp implements Observer {

private String appId;

public MobileApp(String appId) {

this.appId = appId;

}

*@Override*

public void update(String stockName, double newPrice) {

System.***out***.println("MobileApp [" + appId + "]: " + stockName + " is now ₹" + newPrice);

}}

*WebApp.java*

package com.observerpattern;

public class WebApp implements Observer {

private String siteName;

public WebApp(String siteName) {

this.siteName = siteName;

}

*@Override*

public void update(String stockName, double newPrice) {

System.***out***.println("WebApp [" + siteName + "]: Price updated - " + stockName + " → ₹" + newPrice);}}

*Main.java*

package com.observerpattern;

public class Main {

public static void main(String[] args) {

StockMarket cts = new StockMarket("CTS", 3500.00);

Observer mob1 = new MobileApp("User101");

Observer web1 = new WebApp("StockWeb");

cts.registerObserver(mob1);

cts.registerObserver(web1);

System.***out***.println("Initial stock update:");

cts.setStockPrice(3550.50);

System.***out***.println("\nPrice updated again:");

cts.setStockPrice(3599.75);

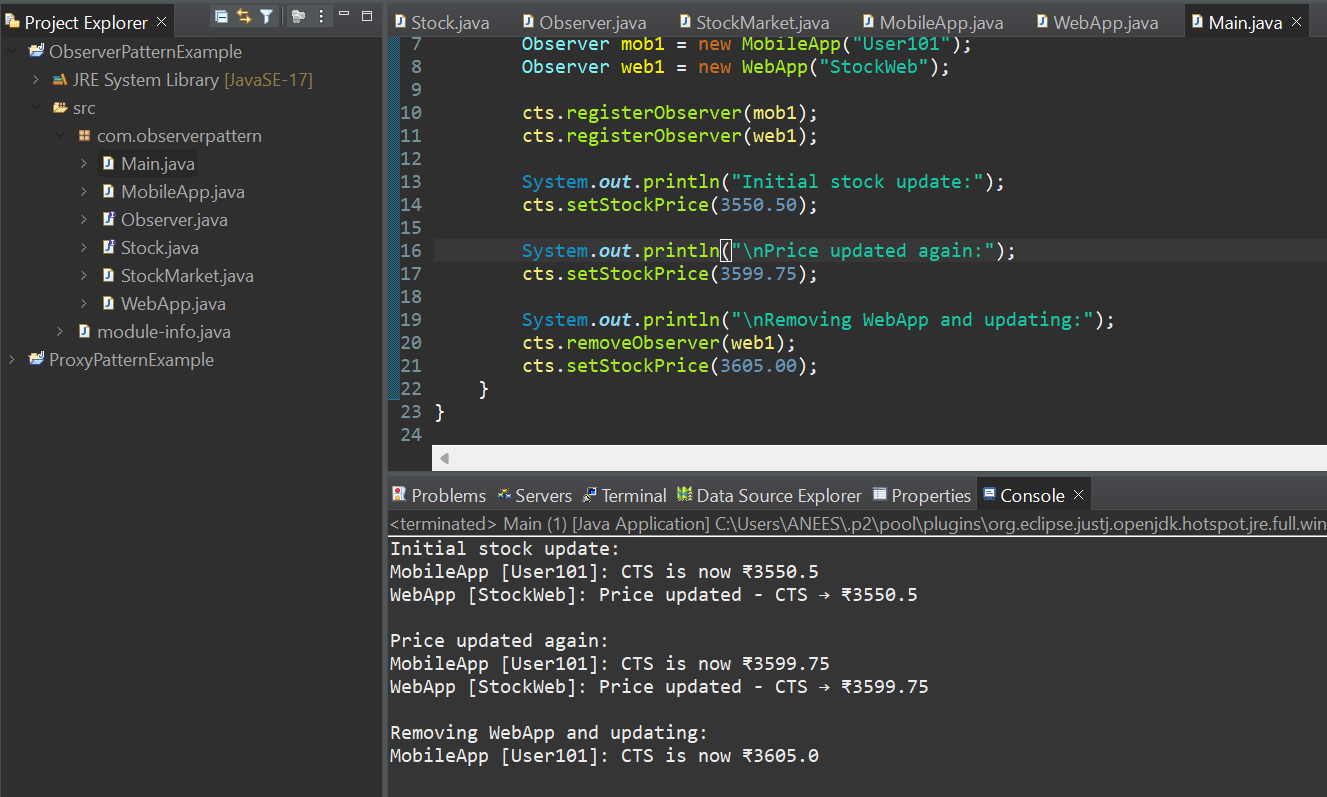
System.***out***.println("\nRemoving WebApp and updating:");

cts.removeObserver(web1);

cts.setStockPrice(3605.00);

}}

**Output**

****

**Exercise 8: Implementing the Strategy Pattern**

**Scenario:**

You are developing a payment system where different payment methods (e.g., Credit Card, PayPal) can be selected at runtime. Use the Strategy Pattern to achieve this

**Code**

*PaymentStrategy.java*

package com.strategy;

public interface PaymentStrategy {

void pay(double amount);

}

*CreditCardPayment.java*

package com.strategy;

public class CreditCardPayment implements PaymentStrategy {

private String cardNumber;

private String cardHolder;

public CreditCardPayment(String cardNumber, String cardHolder) {

this.cardNumber = cardNumber;

this.cardHolder = cardHolder;

}

@Override

public void pay(double amount) {

System.out.println("Paid ₹" + amount + " using Credit Card [" + cardNumber + "] - Holder: " + cardHolder);

}}

*PayPalPayment.java*

package com.strategy;

public class PayPalPayment implements PaymentStrategy {

private String email;

public PayPalPayment(String email) {

this.email = email;

}

@Override

public vioid pay(double amount) {

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

}

}

*PaymentContext.java*

package com.strategy;

public class PaymentContext {

private PaymentStrategy strategy;

public void setPaymentStrategy(PaymentStrategy strategy) {

this.strategy = strategy;

}

public void makePayment(double amount) {

if (strategy == null) {

System.out.println("Please select a payment method first.");

} else {

strategy.pay(amount);

}}}

*Main.java*

package com.strategy;

public class Main {

public static void main(String[] args) {

PaymentContext context = new PaymentContext();

// Using Credit Card

context.setPaymentStrategy(new CreditCardPayment("1234-5678-9012-3456", "Anees Fathima"));

context.makePayment(1500.00);

System.out.println();

// Using PayPal

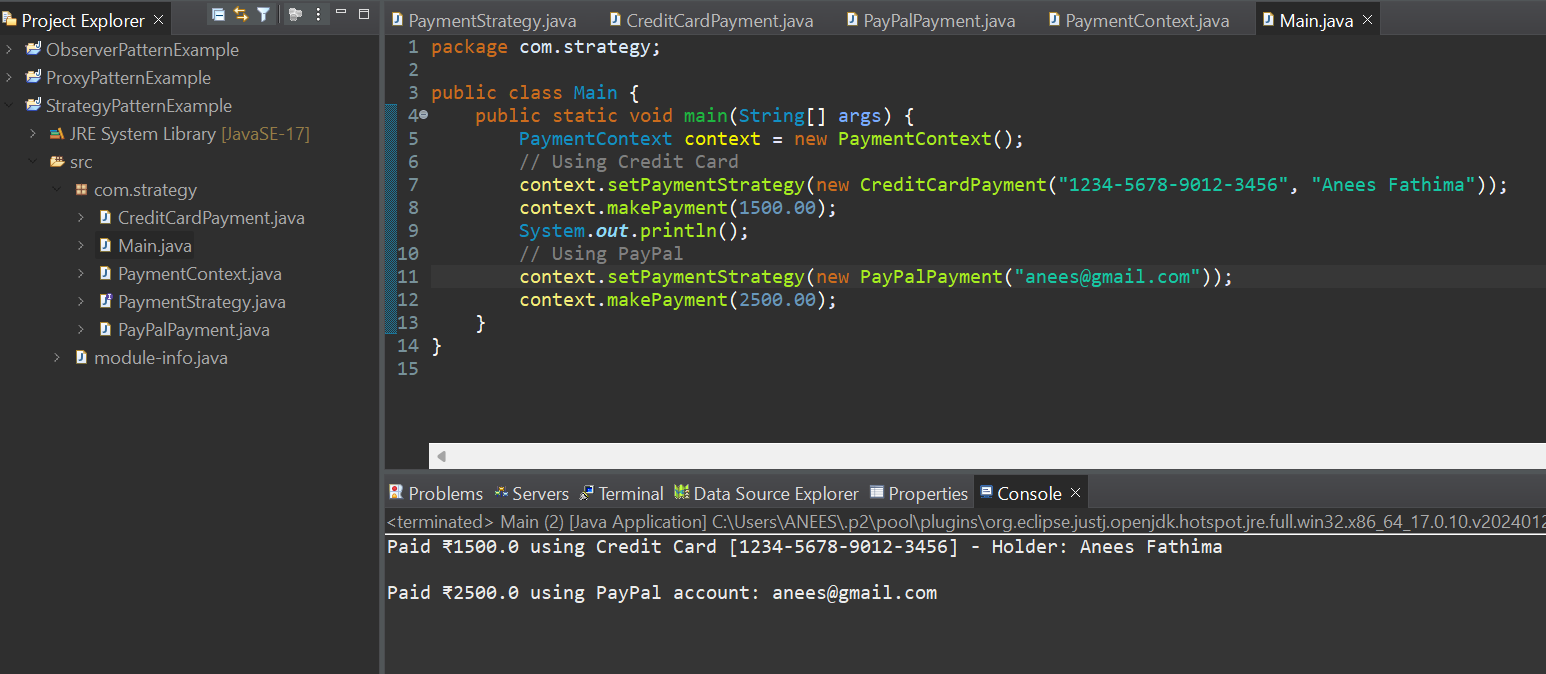
context.setPaymentStrategy(new PayPalPayment("anees@gmail.com"));

context.makePayment(2500.00);

}

}

**Output**



**Exercise 9: Implementing the Command Pattern**

**Scenario:** You are developing a home automation system where commands can be issued to turn devices on or off. Use the Command Pattern to achieve this.

**Code**

*Command.java*

package com.automation;

interface Command {

void execute();

}

*Light.java*

package com.automation;

public class Light {

private String location;

public Light(String location) {

this.location = location;

}

public void turnOn() {

System.***out***.println(location + " light is ON.");

}

public void turnOff() {

System.***out***.println(location + " light is OFF.");

}

}

*LightOnCommand.java*

package com.automation;

public class LightOnCommand implements Command {

private Light light;

public LightOnCommand(Light light) {

this.light = light;

}

*@Override*

public void execute() {

light.turnOn();

}

}

*LightOffCommand.java*

package com.automation;

public class LightOffCommand implements Command {

private Light light;

public LightOffCommand(Light light) {

this.light = light;

}

*@Override*

public void execute() {

light.turnOff();

}}

*RemoteControl.java*

package com.automation;

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 is set.");

}}}

*Main.java*

package com.automation;

import java.util.Scanner;

public class Main {

public static void main(String[] args) {

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

Light bedroomLight = new Light("Bedroom");

Command lightOn = new LightOnCommand(bedroomLight);

Command lightOff = new LightOffCommand(bedroomLight);

RemoteControl remote = new RemoteControl();

System.***out***.println("Welcome to the Home Automation System.");

System.***out***.println("Enter command (1 = Turn ON, 2 = Turn OFF, 0 = Exit):");

while (true) {

System.***out***.print("Your choice: ");

int choice = scanner.nextInt();

switch (choice) {

case 1:

remote.setCommand(lightOn);

remote.pressButton();

break;

case 2:

remote.setCommand(lightOff);

remote.pressButton();

break;

case 0:

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

scanner.close();

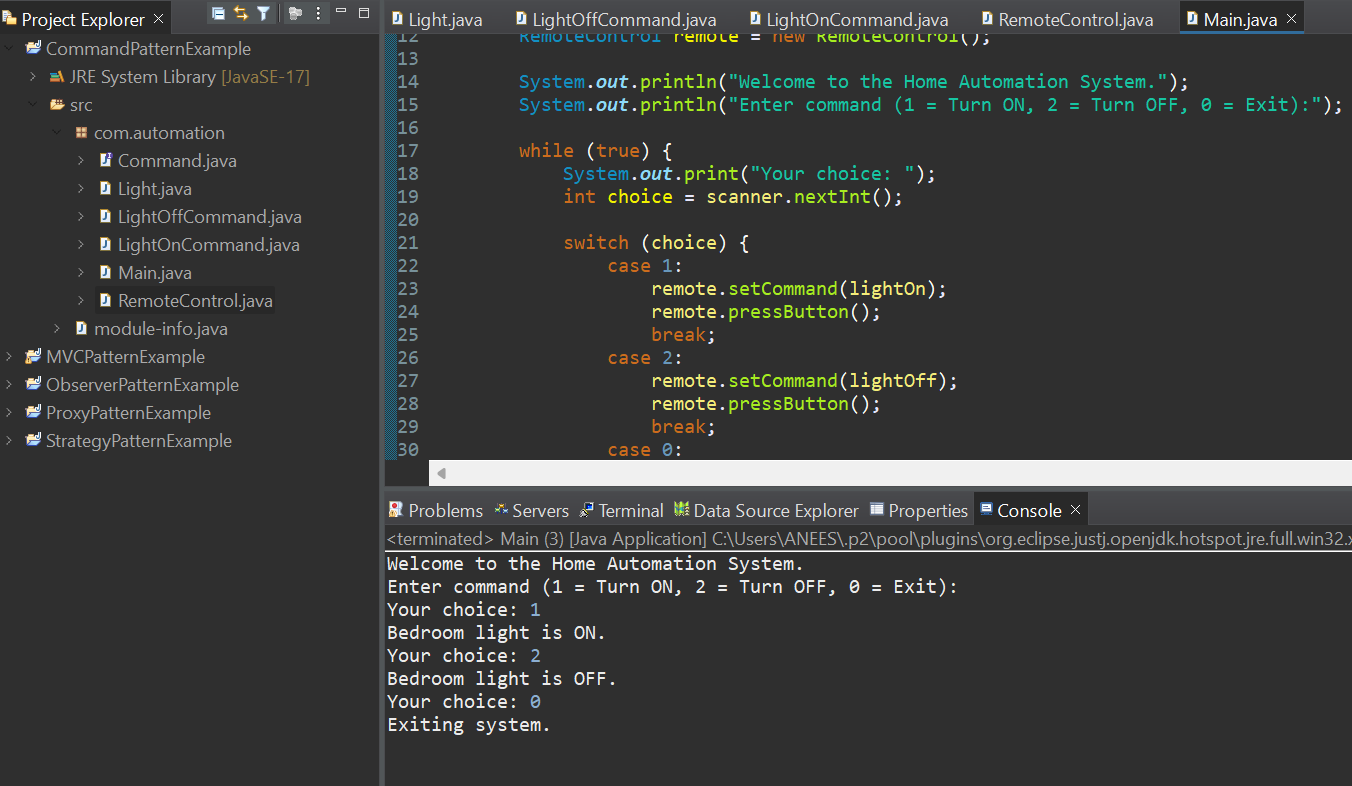
return;

default:

System.***out***.println("Invalid option. Try again.");

}}}}

**Output**

****

**Exercise 10: Implementing the MVC Pattern**

**Scenario:**

You are developing a simple web application for managing student records using the MVC pattern.

**Code**

*Student.java*

package com.mvc;

public class Student {

private String name;

private String id;

private String grade;

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

this.name = name;

this.id = id;

this.grade = grade;

}

// Getters

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; }

}

*StudentView.java*

package com.mvc;

public class StudentView {

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

System.out.println("\n--- Student Details ---");

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

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

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

}

}

*StudentController.java*

package com.mvc;

public class StudentController {

private Student model;

private StudentView view;

public StudentController(Student model, StudentView view) {

this.model = model;

this.view = view;

}

// Update 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); }

// Display using view

public void updateView() {

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

}

}

*Main.java*

package com.mvc;

import java.util.Scanner;

public class Main {

public static void main(String[] args) {

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

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

String name = sc.nextLine();

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

String id = sc.nextLine();

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

String grade = sc.nextLine();

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

StudentView view = new StudentView();

StudentController controller = new StudentController(model, view);

// Show initial details

controller.updateView();

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

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

String newGrade = sc.nextLine();

controller.setStudentGrade(newGrade);

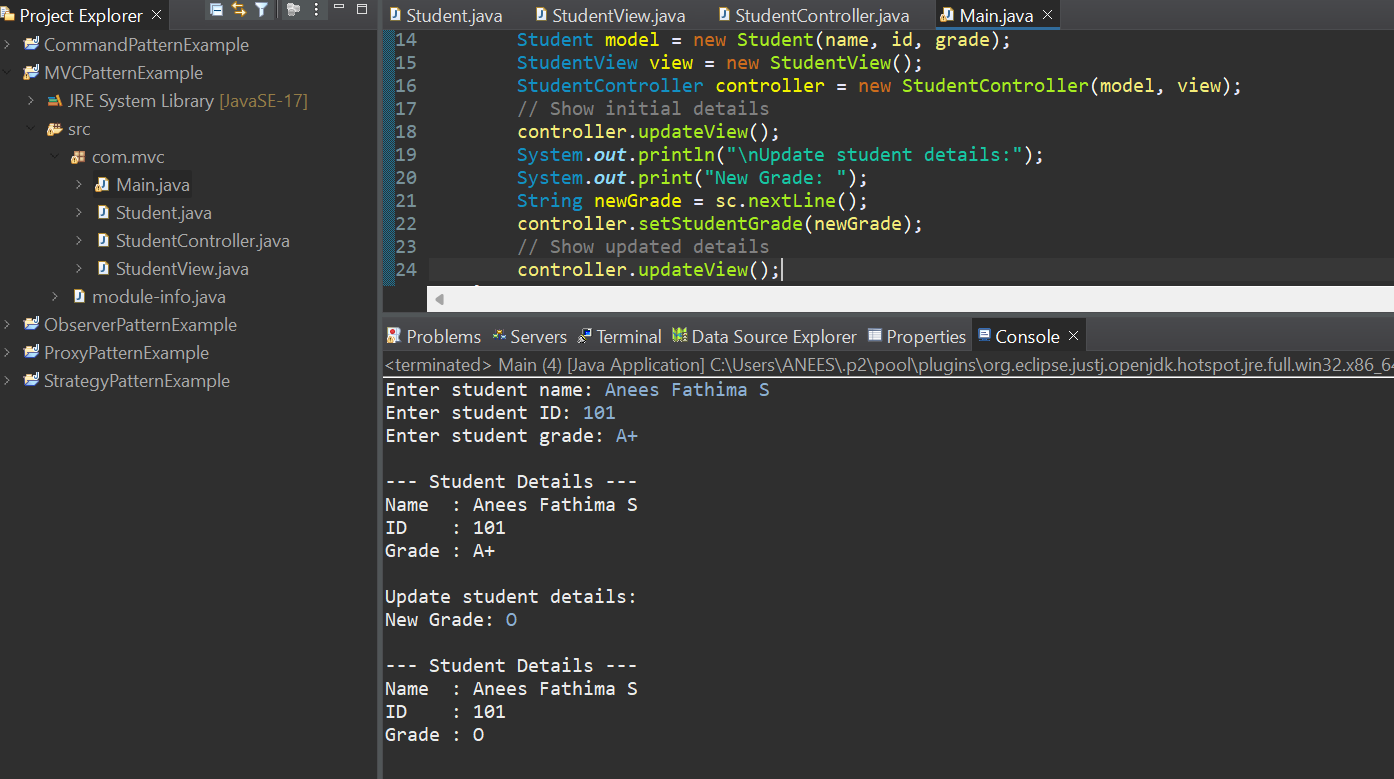
// Show updated details

controller.updateView();

}

}

**Output**



**Exercise 11: Implementing Dependency Injection**

**Scenario:**

You are developing a customer management application where the service class depends on a repository class. Use Dependency Injection to manage these dependencies.

**Code**

*Customer.java*

package com.dependency;

public class Customer {

private String id;

private String name;

private String email;

public Customer(String id, String name, String email) {

this.id = id;

this.name = name;

this.email = email;

}

public String getId() { return id; }

public String getName() { return name; }

public String getEmail() { return email; }}

*CustomerRepository.java*

package com.dependency;

public interface CustomerRepository {

Customer findCustomerById(String id);

}

*CustomerRepositoryImpl.java*

package com.dependency;

import java.util.HashMap;

import java.util.Map;

public class CustomerRepositoryImpl implements CustomerRepository {

private Map<String, Customer> customerDB;

public CustomerRepositoryImpl() {

customerDB = new HashMap<>();

customerDB.put("C101", new Customer("C101", "Anees Fathima", "anees@gmail.com"));

customerDB.put("C102", new Customer("C102", "Shalini", "shalini@gmail.com"));

customerDB.put("C103", new Customer("C103", "Ananthi", "Ananthi@gmail.com"));

}

*@Override*

public Customer findCustomerById(String id) {

return customerDB.get(id);

}

}

*CustomerService .java*

package com.dependency;

public class CustomerService {

private CustomerRepository repository;

public CustomerService(CustomerRepository repository) {

this.repository = repository;

}

public void displayCustomerById(String id) {

Customer customer = repository.findCustomerById(id);

if (customer != null) {

System.***out***.println("\nCustomer Found:");

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

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

System.***out***.println("Email : " + customer.getEmail());

} else {

System.***out***.println("Customer with ID " + id + " not found.");

}}}

*Main.java*

package com.dependency;

import java.util.\*;

public class Main {

public static void main(String[] args) {

// Dependency Injection

CustomerRepository repository = new CustomerRepositoryImpl();

CustomerService service = new CustomerService(repository);

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

System.***out***.println("Welcome to the Customer Management System");

while (true) {

System.***out***.print("\nEnter Customer ID to search (or type 'exit' to quit): ");

String input = scanner.nextLine();

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

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

break;

}

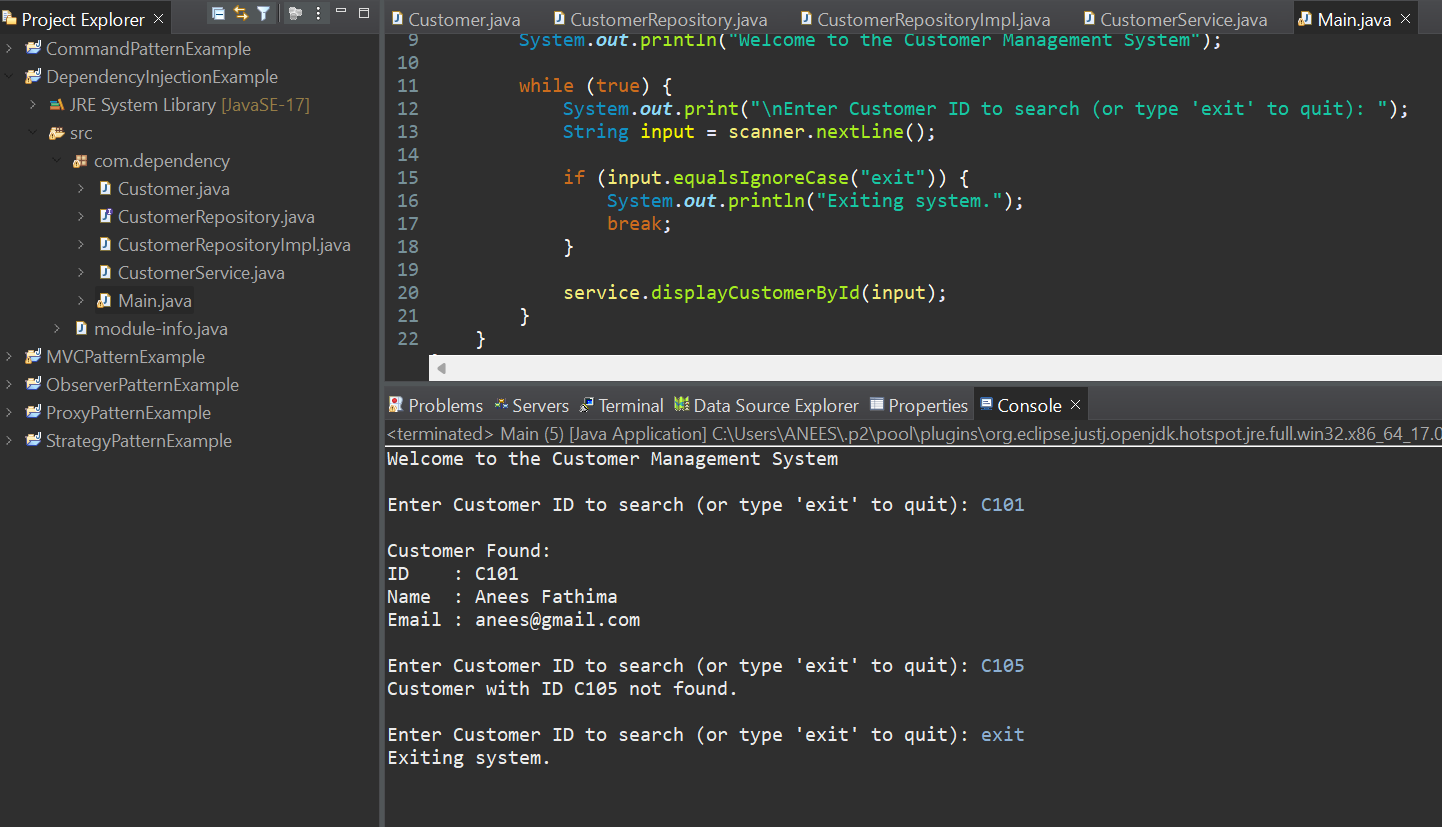
service.displayCustomerById(input);

}

}

}

**Output**

****