|  |  |
| --- | --- |
| A picture of a winding road and trees  Hands-On Exercises Submission  Module 1 – Design Patterns and Principles | Name: Rohini Vaiyathilingam  **Skill:** Design Principles & Patterns |

WEEK 1 : HANDS ON EXERCISES

Exercise 1: Implementing the Singleton Pattern

//CODE

import java.text.SimpleDateFormat;

import java.util.Date;

public class Logger {

private static volatile Logger instance = null;

private Logger() {

System.out.println("[System] Logger initialized at " + getCurrentTimestamp());

}

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("[" + getCurrentTimestamp() + "] " + message);

}

private String getCurrentTimestamp() {

return new SimpleDateFormat("HH:mm:ss.SSS").format(new Date());

}

public static void main(String[] args) {

Logger logger1 = Logger.getInstance();

logger1.log("Singleton pattern is awesome!");

logger1.log("Logging a second message just to test.");

Logger logger2 = Logger.getInstance();

logger2.log("Logger still works with the same instance.");

logger2.log("Same instance? " + (logger1 == logger2));

Runnable task = () -> {

Logger logger = Logger.getInstance();

logger.log("Logging from thread " + Thread.currentThread().getName());

};

Thread t1 = new Thread(task, "Worker-1");

Thread t2 = new Thread(task, "Worker-2");

Thread t3 = new Thread(task, "Worker-3");

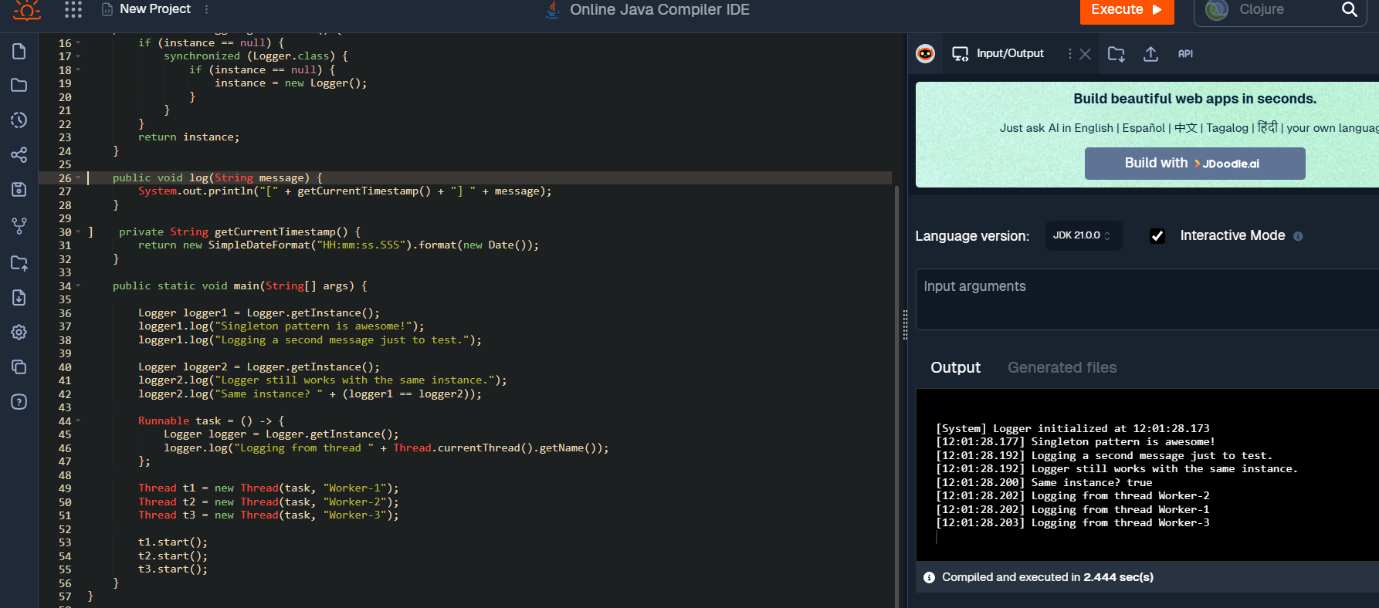
t1.start();

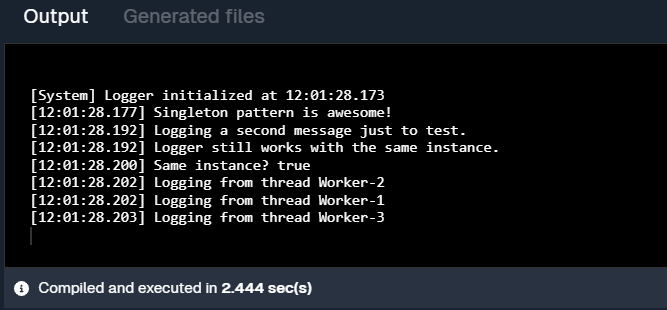
t2.start();

t3.start();

}

}





Exercise 2: Implementing the Factory Method Pattern

//Code

interface Document {

void open();

}

class WordDocument implements Document {

public void open() {

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

}

}

class PdfDocument implements Document {

public void open() {

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

}

}

class ExcelDocument implements Document {

public void open() {

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

}

}

abstract class DocumentFactory {

public abstract Document createDocument();

}

class WordDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new WordDocument();

}

}

class PdfDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new PdfDocument();

}

}

class ExcelDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new ExcelDocument();

}

}

public class DocumentApp {

public static void main(String[] args) {

System.out.println("[System] Document Manager Started ✨");

DocumentFactory wordFactory = new WordDocumentFactory();

Document wordDoc = wordFactory.createDocument();

wordDoc.open();

DocumentFactory pdfFactory = new PdfDocumentFactory();

Document pdfDoc = pdfFactory.createDocument();

pdfDoc.open();

DocumentFactory excelFactory = new ExcelDocumentFactory();

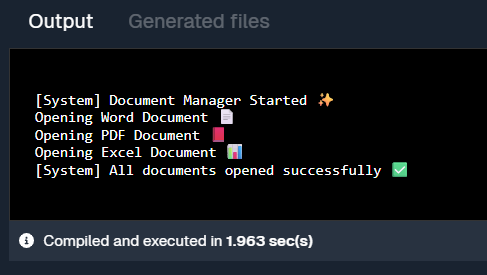
Document excelDoc = excelFactory.createDocument();

excelDoc.open();

System.out.println("[System] All documents opened successfully ✅");

}

}



Exercise 3: Implementing the Builder Pattern

class Computer {

// Required attributes

private final String cpu;

private final String ram;

private final String storage;

private final String graphicsCard;

private final String operatingSystem;

private Computer(Builder builder) {

this.cpu = builder.cpu;

this.ram = builder.ram;

this.storage = builder.storage;

this.graphicsCard = builder.graphicsCard;

this.operatingSystem = builder.operatingSystem;

}

public void showSpecs() {

System.out.println("🖥️ Computer Configuration:");

System.out.println("CPU: " + cpu);

System.out.println("RAM: " + ram);

System.out.println("Storage: " + (storage != null ? storage : "Not included"));

System.out.println("Graphics Card: " + (graphicsCard != null ? graphicsCard : "Not included"));

System.out.println("Operating System: " + (operatingSystem != null ? operatingSystem : "Not included"));

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

}

// Static nested Builder Class

public static class Builder {

private final String cpu;

private final String ram;

private String storage;

private String graphicsCard;

private String operatingSystem;

// Constructor for required attributes

public Builder(String cpu, String ram) {

this.cpu = cpu;

this.ram = ram;

}

public Builder setStorage(String storage) {

this.storage = storage;

return this;

}

public Builder setGraphicsCard(String graphicsCard) {

this.graphicsCard = graphicsCard;

return this;

}

public Builder setOperatingSystem(String os) {

this.operatingSystem = os;

return this;

}

public Computer build() {

return new Computer(this);

}

}

}

public class ComputerBuilderDemo {

public static void main(String[] args) {

System.out.println("[System] Builder Pattern Demo Started 🧱");

Computer basicComputer = new Computer.Builder("Intel i3", "4GB").build();

basicComputer.showSpecs();

Computer gamingComputer = new Computer.Builder("Intel i9", "32GB")

.setGraphicsCard("NVIDIA RTX 4080")

.setStorage("2TB SSD")

.setOperatingSystem("Windows 11")

.build();

gamingComputer.showSpecs();

Computer officeComputer = new Computer.Builder("AMD Ryzen 5", "16GB")

.setStorage("1TB HDD")

.setOperatingSystem("Ubuntu Linux")

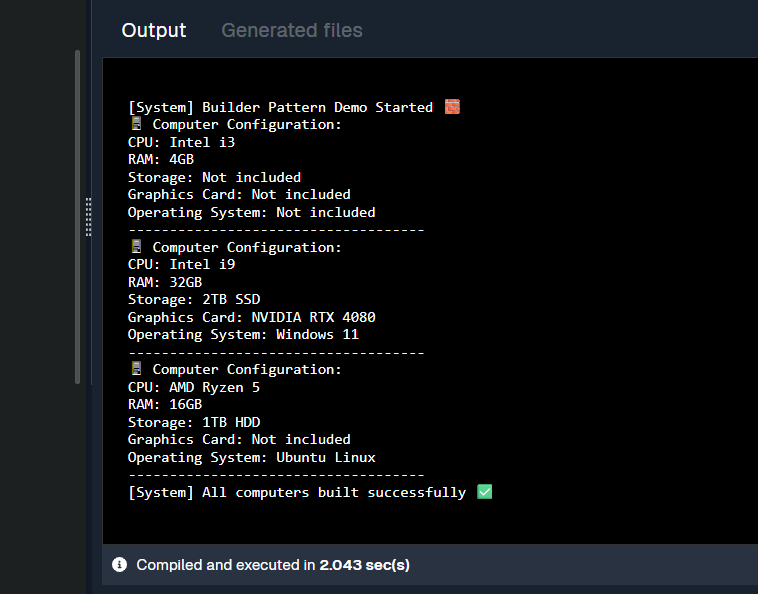
.build();

officeComputer.showSpecs();

System.out.println("[System] All computers built successfully ✅");

}

}



Exercise 4: Implementing the Adapter Pattern

interface PaymentProcessor {

void processPayment(double amount);

}

class PayPalGateway {

public void makePayment(double amountInDollars) {

System.out.println("Processing payment via PayPal: $" + amountInDollars);

}

}

class StripeGateway {

public void payAmount(double amount) {

System.out.println("Payment processed using Stripe: $" + amount);

}

}

class RazorpayGateway {

public void doTransaction(double amt) {

System.out.println("Razorpay processed the transaction: ₹" + amt);

}

}

class PayPalAdapter implements PaymentProcessor {

private PayPalGateway paypal;

public PayPalAdapter() {

this.paypal = new PayPalGateway();

}

public void processPayment(double amount) {

paypal.makePayment(amount);

}

}

class StripeAdapter implements PaymentProcessor {

private StripeGateway stripe;

public StripeAdapter() {

this.stripe = new StripeGateway();

}

public void processPayment(double amount) {

stripe.payAmount(amount);

}

}

class RazorpayAdapter implements PaymentProcessor {

private RazorpayGateway razorpay;

public RazorpayAdapter() {

this.razorpay = new RazorpayGateway();

}

public void processPayment(double amount) {

razorpay.doTransaction(amount);

}

}

public class PaymentSystemDemo {

public static void main(String[] args) {

System.out.println("[System] Payment Processing Started 💳");

PaymentProcessor paypalProcessor = new PayPalAdapter();

paypalProcessor.processPayment(2500.00);

PaymentProcessor stripeProcessor = new StripeAdapter();

stripeProcessor.processPayment(3999.99);

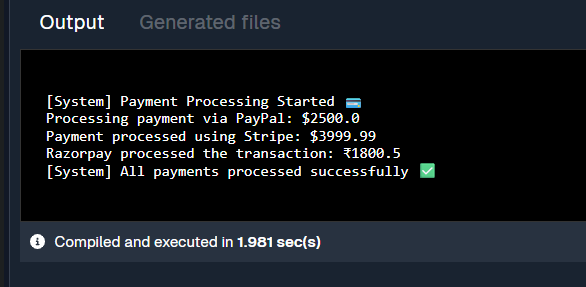
PaymentProcessor razorpayProcessor = new RazorpayAdapter();

razorpayProcessor.processPayment(1800.50);

System.out.println("[System] All payments processed successfully ✅");

}

}



Exercise 5: Implementing the Decorator Pattern

interface Notifier {

void send(String message);

}

class EmailNotifier implements Notifier {

public void send(String message) {

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

}

}

abstract class NotifierDecorator implements Notifier {

protected Notifier wrappee;

public NotifierDecorator(Notifier notifier) {

this.wrappee = notifier;

}

public void send(String message) {

wrappee.send(message); // delegate to wrapped notifier

}

}

class SMSNotifierDecorator extends NotifierDecorator {

public SMSNotifierDecorator(Notifier notifier) {

super(notifier);

}

public void send(String message) {

super.send(message);

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

}

}

class SlackNotifierDecorator extends NotifierDecorator {

public SlackNotifierDecorator(Notifier notifier) {

super(notifier);

}

public void send(String message) {

super.send(message);

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

}

}

public class NotifierDemo {

public static void main(String[] args) {

System.out.println("[System] Notification Dispatch Started 🚀");

Notifier baseNotifier = new EmailNotifier();

Notifier smsNotifier = new SMSNotifierDecorator(baseNotifier);

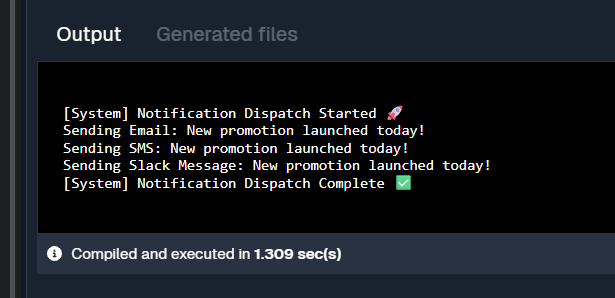
Notifier fullNotifier = new SlackNotifierDecorator(smsNotifier);

fullNotifier.send("New promotion launched today!");

System.out.println("[System] Notification Dispatch Complete ✅");

}

}



Exercise 6: Implementing the Proxy Pattern

interface Image {

void display();

}

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);

try {

Thread.sleep(1000); // simulate network delay

} catch (InterruptedException e) {

System.out.println("Loading interrupted");

}

}

public void display() {

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

}

}

class ProxyImage implements Image {

private String filename;

private RealImage realImage;

public ProxyImage(String filename) {

this.filename = filename;

}

public void display() {

if (realImage == null) {

System.out.println("🔄 First-time access. Creating RealImage...");

realImage = new RealImage(filename); // lazy loading

} else {

System.out.println("✅ Using cached image...");

}

realImage.display();

}

}

public class ImageViewerDemo {

public static void main(String[] args) {

System.out.println("[System] Image Viewer Started 🖥️");

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

image1.display();

System.out.println();

image1.display();

System.out.println();

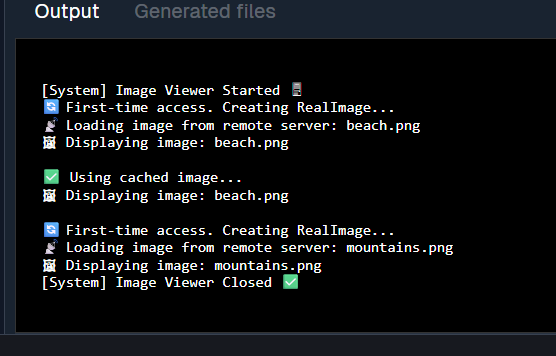
Image image2 = new ProxyImage("mountains.png");

image2.display(); // Loads new image from server

System.out.println("[System] Image Viewer Closed ✅");

}

}



Exercise 7: Implementing the Observer Pattern

import java.util.ArrayList;

import java.util.List;

interface Stock {

void register(Observer observer);

void deregister(Observer observer);

void notifyObservers();

void setPrice(double newPrice);

}

interface Observer {

void update(double updatedPrice);

}

class StockMarket implements Stock {

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

private double stockPrice;

public void register(Observer observer) {

observers.add(observer);

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

}

public void deregister(Observer observer) {

observers.remove(observer);

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

}

public void notifyObservers() {

for (Observer observer : observers) {

observer.update(stockPrice);

}

}

public void setPrice(double newPrice) {

System.out.println("💹 Stock price updated to ₹" + newPrice);

this.stockPrice = newPrice;

notifyObservers();

}

}

class MobileApp implements Observer {

public void update(double updatedPrice) {

System.out.println("📱 MobileApp: Stock price is now ₹" + updatedPrice);

}

}

class WebApp implements Observer {

public void update(double updatedPrice) {

System.out.println("💻 WebApp: Stock price is now ₹" + updatedPrice);

}

}

public class StockMarketApp {

public static void main(String[] args) {

System.out.println("[System] Stock Market Notification System Started 🧠");

StockMarket stockMarket = new StockMarket();

Observer mobileObserver = new MobileApp();

Observer webObserver = new WebApp();

stockMarket.register(mobileObserver);

stockMarket.register(webObserver);

stockMarket.setPrice(1587.45);

System.out.println();

stockMarket.setPrice(1603.20);

System.out.println();

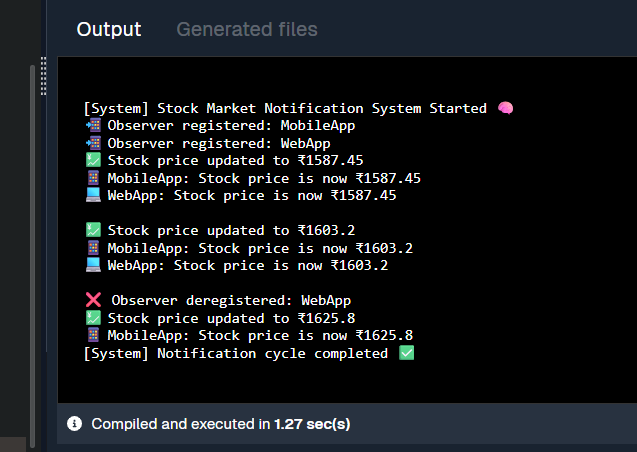
stockMarket.deregister(webObserver);

stockMarket.setPrice(1625.80);

System.out.println("[System] Notification cycle completed ✅");

}

}



Exercise 8: Implementing the Strategy Pattern

interface PaymentStrategy {

void pay(double amount);

}

class CreditCardPayment implements PaymentStrategy {

private String cardNumber;

public CreditCardPayment(String cardNumber) {

this.cardNumber = cardNumber;

}

public void pay(double amount) {

System.out.println("💳 Paid ₹" + amount + " using Credit Card ending with " + cardNumber.substring(cardNumber.length() - 4));

}

}

class PayPalPayment implements PaymentStrategy {

private String email;

public PayPalPayment(String email) {

this.email = email;

}

public void pay(double amount) {

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

}

}

class PaymentContext {

private PaymentStrategy strategy;

public void setPaymentStrategy(PaymentStrategy strategy) {

this.strategy = strategy;

}

public void makePayment(double amount) {

if (strategy == null) {

System.out.println("⚠️ No payment method selected!");

} else {

strategy.pay(amount);

}

}

}

public class PaymentStrategyDemo {

public static void main(String[] args) {

System.out.println("[System] Payment Strategy Demo Started 🔁");

PaymentContext context = new PaymentContext();

context.setPaymentStrategy(new CreditCardPayment("1234567890123456"));

context.makePayment(2999.99);

System.out.println();

context.setPaymentStrategy(new PayPalPayment("user@example.com"));

context.makePayment(1599.50);

System.out.println();

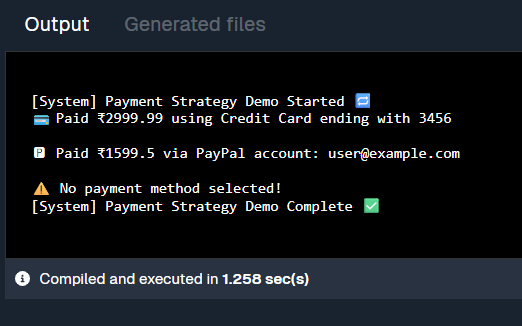
context.setPaymentStrategy(null);

context.makePayment(800.00);

System.out.println("[System] Payment Strategy Demo Complete ✅");

}

}



Exercise 9: Implementing the Command Pattern

interface Command {

void execute();

}

class Light {

public void turnOn() {

System.out.println("💡 Light is ON");

}

public void turnOff() {

System.out.println("🔌 Light is OFF");

}

}

class LightOnCommand implements Command {

private Light light;

public LightOnCommand(Light light) {

this.light = light;

}

public void execute() {

light.turnOn();

}

}

class LightOffCommand implements Command {

private Light light;

public LightOffCommand(Light light) {

this.light = light;

}

public void execute() {

light.turnOff();

}

}

class RemoteControl {

private Command command;

public void setCommand(Command command) {

this.command = command;

}

public void pressButton() {

if (command != null) {

command.execute();

} else {

System.out.println("⚠️ No command set.");

}

}

}

public class HomeAutomationDemo {

public static void main(String[] args) {

System.out.println("[System] Home Automation System Started 🏠");

Light livingRoomLight = new Light();

Command lightsOn = new LightOnCommand(livingRoomLight);

Command lightsOff = new LightOffCommand(livingRoomLight);

RemoteControl remote = new RemoteControl();

remote.setCommand(lightsOn);

remote.pressButton();

remote.setCommand(lightsOff);

remote.pressButton();

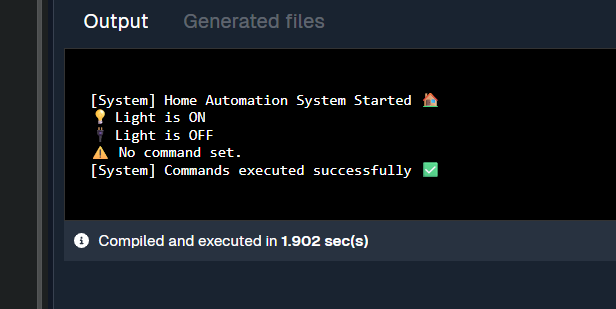
remote.setCommand(null);

remote.pressButton();

System.out.println("[System] Commands executed successfully ✅");

}

}



Exercise 10: Implementing the MVC Pattern

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;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public String getId() {

return id;

}

public void setId(String id) {

this.id = id;

}

public String getGrade() {

return grade;

}

public void setGrade(String grade) {

this.grade = grade;

}

}

class StudentView {

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

System.out.println("🎓 Student Record:");

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

System.out.println("🆔 ID : " + id);

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

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

}

}

class StudentController {

private Student model;

private StudentView view;

public StudentController(Student model, StudentView view) {

this.model = model;

this.view = view;

}

public void setStudentName(String name) {

model.setName(name);

}

public void setStudentGrade(String grade) {

model.setGrade(grade);

}

public void updateView() {

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

}

}

public class MVCDemo {

public static void main(String[] args) {

System.out.println("[System] MVC Pattern Demo Started ✅");

Student model = new Student("Rohit Sharma", "SKCT2301", "A");

StudentView view = new StudentView();

StudentController controller = new StudentController(model, view);

controller.updateView();

controller.setStudentName("Rohini V");

controller.setStudentGrade("A+");

controller.updateView();

System.out.println("[System] MVC Demo Completed Successfully 🎯");

}

}

