**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.

**Solution:**

package SingletonPattern;

public class Main {

public static void main(String[] args) {

Logger l1=Logger.*getInstance*();

l1.log("The Log l1 is created");

Logger l2=Logger.*getInstance*();

l2.log("The Log l2 is created");

if(l1==l2)

{

System.***out***.println("Both are same instance");

}

else {

System.***out***.println("Both are differnt instance");

}

}

}

package SingletonPattern;

public class Logger {

private static Logger *singleInstance*;

private Logger()

{

System.***out***.println("Log is created....");

}

public static Logger getInstance() {

if(*singleInstance* == null)

{

*singleInstance* = new Logger();

}

return *singleInstance*;

}

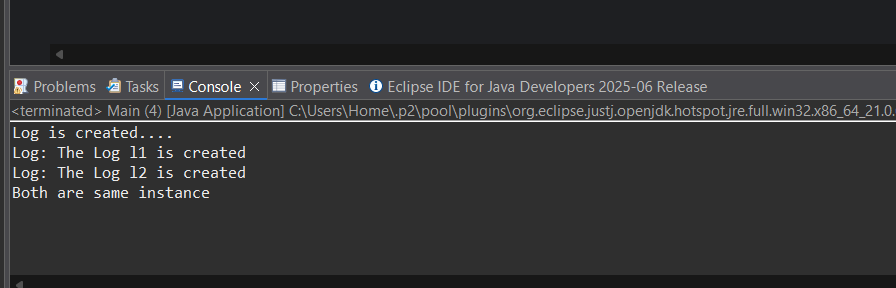
public void log(String msg)

{

System.***out***.println("Log: "+msg);

}

}

**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.

**Solution:**

package FactoryMethodPattern;

public interface Document {

void open();

}

package FactoryMethodPattern;

public abstract class DocumentFactory {

public abstract Document createDocument();

}

package FactoryMethodPattern;

public class ExcelDocument implements Document{

public void open() {

System.***out***.println("Open Excel Sheet...");

}

}

package FactoryMethodPattern;

public class ExcelDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new ExcelDocument();

}

}

package FactoryMethodPattern;

public class PdfDocument implements Document{

public void open() {

System.***out***.println("Opening Pdf Document...");

}

}

package FactoryMethodPattern;

public class PdfDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new PdfDocument();

}

}

package FactoryMethodPattern;

public class WordDocument implements Document {

public void open() {

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

}

}

package FactoryMethodPattern;

public class WordDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new WordDocument();

}

}

package FactoryMethodPattern;

public class Main {

public static void main(String[] args) {

DocumentFactory wf = new WordDocumentFactory();

Document wordDoc = wf.createDocument();

wordDoc.open();

DocumentFactory pf = new PdfDocumentFactory();

Document pdfDoc = pf.createDocument();

pdfDoc.open();

DocumentFactory ef = new ExcelDocumentFactory();

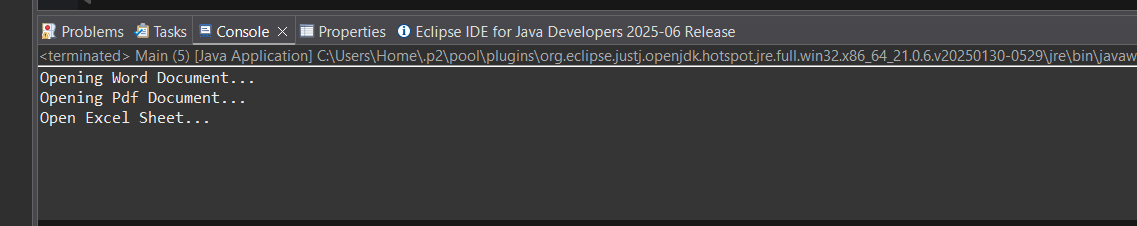
Document excelDoc = ef.createDocument();

excelDoc.open();

}

}

**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.

**Solution:**

package BuilderPattern;

public class Computer {

private String CPU;

private String RAM;

private String storage;

private String OwnerName;

private String GPU;

private boolean WiFi;

private boolean Bluetooth;

private Computer(Builder b) {

this.CPU = b.CPU;

this.RAM = b.RAM;

this.storage = b.storage;

this.OwnerName=b.OwnerName;

this.GPU = b.GPU;

this.WiFi = b.WiFi;

this.Bluetooth = b.Bluetooth;

}

public String toString() {

return "Computer [ Owner= "+OwnerName+", CPU= " + CPU + ", RAM= " + RAM + ", Storage= " + storage +

", GPU= " + GPU + ", WiFi= " + WiFi + ", Bluetooth= " + Bluetooth + "]";

}

public static class Builder{

private String CPU;

private String RAM;

private String storage;

private String OwnerName;

private String GPU;

private boolean WiFi;

private boolean Bluetooth;

public void setOwnerName(String ownerName) {

OwnerName = ownerName;

}

public void setCPU(String CPU) {

this.CPU = CPU;

}

public void setRAM(String RAM) {

this.RAM = RAM;

}

public void setStorage(String storage) {

this.storage = storage;

}

public void setGPU(String GPU) {

this.GPU = GPU;

}

public void setWiFi(boolean WiFi) {

this.WiFi = WiFi;

}

public void setBluetooth(boolean Bluetooth) {

this.Bluetooth = Bluetooth;

}

public Computer build() {

return new Computer(this);

}

}

}

package BuilderPattern;

public class Main {

public static void main(String[] args) {

Computer.Builder mypc=new Computer.Builder();

mypc.setOwnerName("Tarun Karrthick");

mypc.setCPU("Intel i7");

mypc.setRAM("16 GB");

mypc.setStorage("1 Tb ssd");

mypc.setGPU("RTX 360");

mypc.setBluetooth(true);

mypc.setWiFi(true);

Computer c= mypc.build();

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

Computer.Builder pc=new Computer.Builder();

pc.setOwnerName("King");

pc.setCPU("Intel i5");

pc.setRAM("16 GB");

pc.setStorage("1 Tb ssd");

pc.setGPU("RTX 360");

pc.setBluetooth(true);

pc.setWiFi(true);

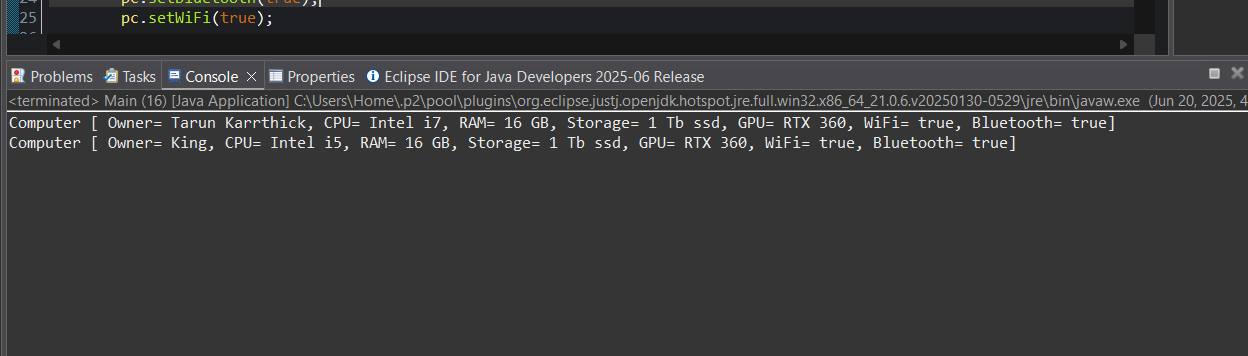
Computer p= pc.build();

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

}

}

**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.

**Solution:**

package AdapterPattern;

public class Gpay {

public void processPayment(int amount)

{

System.*out*.println("Payment done via Google Pay Rs : "+ amount );

}

}

package AdapterPattern;

public class GpayAdapter implements PaymentProcessor {

private Gpay ga;

public GpayAdapter(Gpay ga) {

this.ga=ga;

}

public void processPayment(int amount)

{

ga.processPayment(amount);

}

}

package AdapterPattern;

public interface PaymentProcessor {

void processPayment(int amount);

}

package AdapterPattern;

public class Paytm {

public void processPayment(int amount) {

System.***out***.println("Payment done via Paytm Rs: "+amount);

}

}

package AdapterPattern;

public class PaytmAdapter implements PaymentProcessor{

private Paytm pa;

public PaytmAdapter(Paytm pa) {

this.pa=pa;

}

public void processPayment(int amount)

{

pa.processPayment(amount);

}

}

package AdapterPattern;

public class Main {

public static void main(String[] args) {

Gpay gpay= new Gpay();

PaymentProcessor gpayprocess = new GpayAdapter(gpay);

gpayprocess.processPayment(25000);

Paytm paytm=new Paytm();

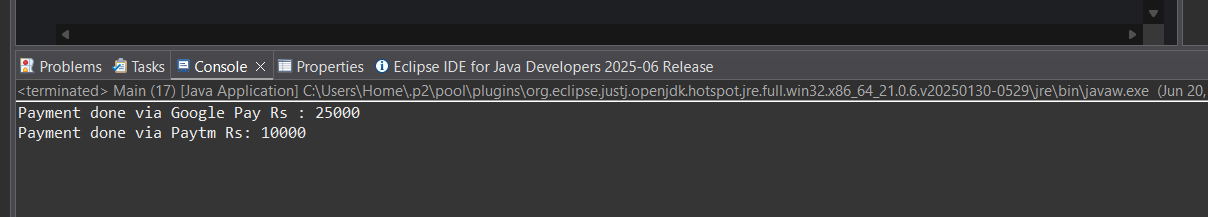
PaymentProcessor paytmprocess = new PaytmAdapter(paytm);

paytmprocess.processPayment(10000);

}

}

**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.

**Solution:**

package DecoratorPattern;

public class EmailNotifier implements Notifier {

public void send(String msg)

{

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

}

}

package DecoratorPattern;

public interface Notifier {

void send(String msg);

}

package DecoratorPattern;

public abstract class NotifierDecorator implements Notifier{

public Notifier refNotifier;

public NotifierDecorator(Notifier refNotifier) {

this.refNotifier = refNotifier;

}

public void send(String msg) {

refNotifier.send(msg);

}

}

package DecoratorPattern;

public class SlackNotifierDecorator extends NotifierDecorator {

public SlackNotifierDecorator(Notifier ref) {

super(ref);

}

public void send(String msg) {

super.send(msg);

sendslack(msg);

}

private void sendslack(String msg) {

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

}

}

package DecoratorPattern;

public class SMSNotifierDecorator extends NotifierDecorator{

public SMSNotifierDecorator(Notifier ref){

super(ref);

}

public void send(String msg) {

super.send(msg);

sendsms(msg);

}

private void sendsms(String msg) {

System.***out***.println("Sending sms: "+msg);

}

}

package DecoratorPattern;

public class Main {

public static void main(String[] args) {

Notifier ref=new EmailNotifier();

SMSNotifierDecorator sms=new SMSNotifierDecorator(ref);

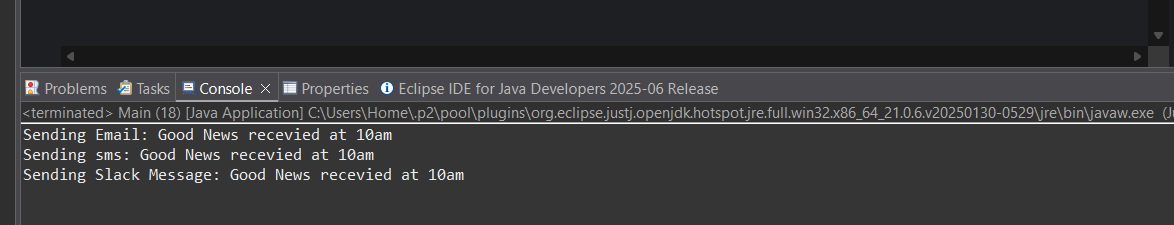
SlackNotifierDecorator slack = new SlackNotifierDecorator(sms);

slack.send("Good News recevied at 10am");

}

}

**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.

**Solution:**

package ProxyPattern;

public interface Image {

void display();

}

package ProxyPattern;

import java.util.\*;

public class ProxyImage implements Image{

private String filename;

private static Map<String, RealImage> *mp* = new HashMap<>();

public ProxyImage(String filename){

this.filename = filename;

}

public void display() {

RealImage real = *mp*.get(filename);

if(real==null) {

real =new RealImage(filename);

*mp*.put(filename,real);

}

else {

System.***out***.println("Loading from memory: "+filename);

}

real.display();

}

}

package ProxyPattern;

public class RealImage implements Image{

private String filename;

public RealImage(String filename)

{

this.filename=filename;

loadfromserver();

}

public void loadfromserver() {

System.***out***.println("Loading "+filename+" from the server...");

}

public void display() {

System.***out***.println("Displaying "+filename);

}

}

package ProxyPattern;

public class Main {

public static void main(String[] args) {

Image im1=new ProxyImage("Picture 1.jpg");

Image im2=new ProxyImage("Picture 2.jpg");

Image im3=new ProxyImage("Picture 3.jpg");

System.***out***.println("First call to picture 1.jpg");

im1.display();

System.***out***.println("First call to picture 2.jpg");

im2.display();

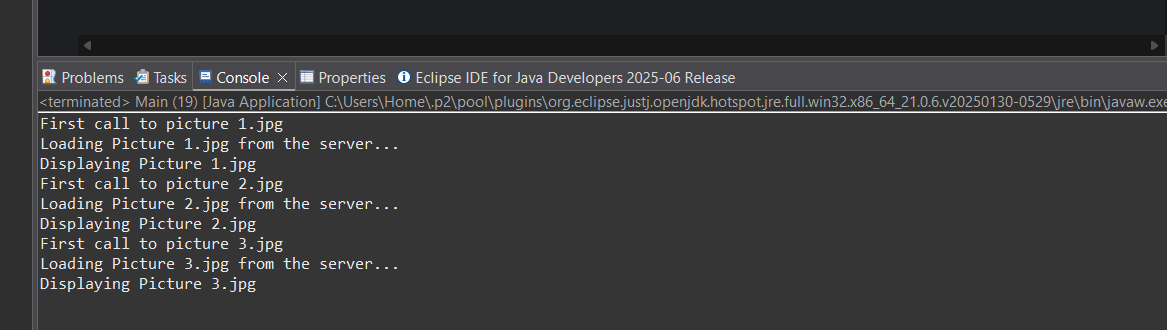
System.***out***.println("First call to picture 3.jpg");

im3.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.

**Solution:**

package ObserverPattern;

public class MobileApp implements Observer{

private String name;

public MobileApp(String name) {

this.name=name;

}

public void update(double price) {

System.***out***.println(name+" MobileApp - New Price ₹ "+price+"...");

}

}

package ObserverPattern;

public interface Observer {

void update(double price);

}

package ObserverPattern;

public interface Stock {

void register(Observer o);

void deregister(Observer o);

void notifyObservers();

}

package ObserverPattern;

import java.util.\*;

public class StockMarket implements Stock{

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

private double price;

public void setPrice(double price) {

this.price = price;

notifyObservers();

}

public void register(Observer o) {

ob.add(o);

}

public void deregister(Observer o) {

ob.remove(o);

}

public void notifyObservers() {

for(Observer o : ob) {

o.update(price);

}

}

}

package ObserverPattern;

public class WebApp implements Observer{

private String name;

public WebApp(String name) {

this.name=name;

}

public void update(double price) {

System.***out***.println(name+" WebApp - New Price ₹ "+price+"...");

}

}

package ObserverPattern;

public class Main {

public static void main(String[] args) {

StockMarket market = new StockMarket();

Observer mobileUser = new MobileApp("Investor A");

Observer webUser = new WebApp("Investor B");

market.register(mobileUser);

market.register(webUser);

System.***out***.println("Updating stock price to ₹120.50...");

market.setPrice(120.50);

System.***out***.println("Removing Web App observer...");

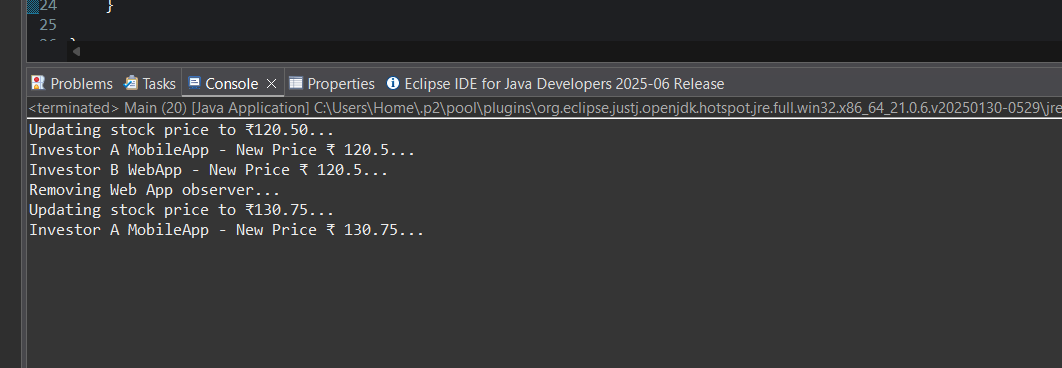
market.deregister(webUser);

System.***out***.println("Updating stock price to ₹130.75...");

market.setPrice(130.75);

}

}

**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.

**Solution:**

package StrategyPattern;

public class CreditCardPayment implements PaymentStrategy {

public void pay(double amount) {

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

}

}

package StrategyPattern;

public class PaymentContext {

private PaymentStrategy strategy;

public void setStrategy(PaymentStrategy strategy) {

this.strategy = strategy;

}

public void makepayment(double amount) {

if(strategy == null) {

System.***out***.println("No Payment method is selected");

}

else {

strategy.pay(amount);

}

}

}

package StrategyPattern;

public interface PaymentStrategy {

void pay(double amount);

}

package StrategyPattern;

public class PayPalPayment implements PaymentStrategy{

public void pay(double amount) {

System.***out***.println("Paid ₹"+amount+" using PayPalPayment...");

}

}

package StrategyPattern;

public class Main {

public static void main(String[] args) {

PaymentContext pc= new PaymentContext();

pc.setStrategy(new CreditCardPayment());

pc.makepayment(15000);

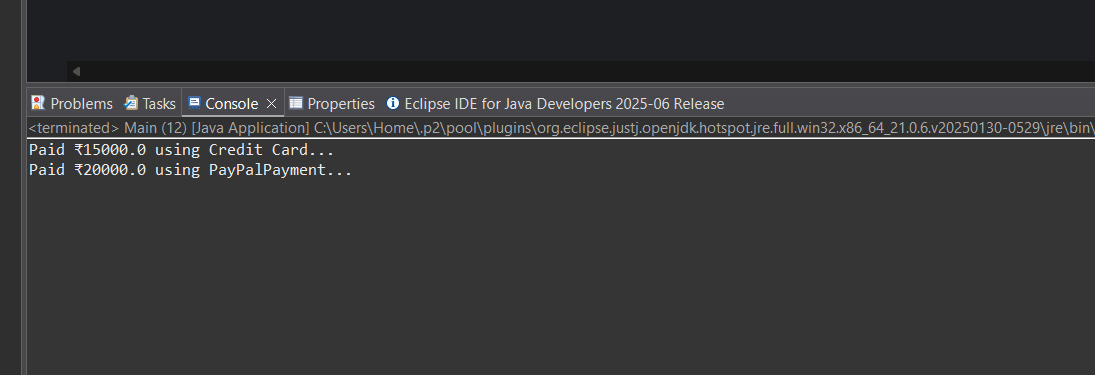
pc.setStrategy(new PayPalPayment());

pc.makepayment(20000);

}

}

**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.

**Solution:**

package CommandPattern;

public interface Command {

void execute();

}

package CommandPattern;

public class Light {

public void turnOn() {

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

}

public void turnOff() {

System.***out***.print("Light is Off...");

}

}

package CommandPattern;

public class LightOffCommand implements Command{

private Light light;

public LightOffCommand(Light light) {

this.light=light;

}

public void execute() {

light.turnOff();

}

}

package CommandPattern;

public class LightOnCommand implements Command{

private Light light;

public LightOnCommand(Light light) {

this.light=light;

}

public void execute() {

light.turnOn();

}

}

package CommandPattern;

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("Give command first!!!");

}

}

}

package CommandPattern;

public class Main {

public static void main(String[] args) {

Light l = new Light();

Command LightOn= new LightOnCommand(l);

Command lightOff= new LightOffCommand(l);

RemoteControl remote = new RemoteControl();

remote.setCommand(LightOn);

remote.pressButton();

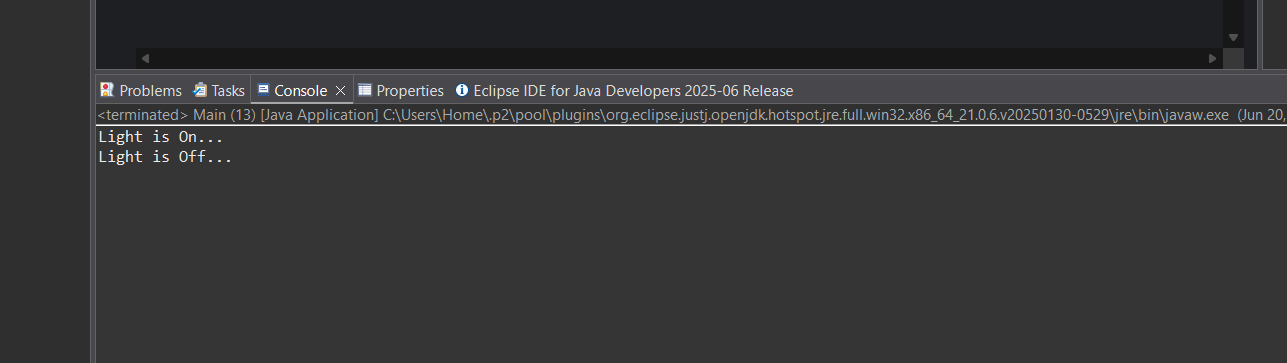
remote.setCommand(lightOff);

remote.pressButton();

}

}

**Output**:



**Exercise 10: Implementing the MVC Pattern**

**Scenario:**

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

**Solution:**

package MVCPattern;

public class Student {

private String name;

private int id;

private String grade;

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public int getId() {

return id;

}

public void setId(int id) {

this.id = id;

}

public String getGrade() {

return grade;

}

public void setGrade(String grade) {

this.grade = grade;

}

}

package MVCPattern;

public class StudentController {

private Student model;

private StudentView view;

public StudentController(Student model,StudentView view) {

this.model=model;

this.view=view;

}

public void setStudentName(String name) {

model.setName(name);

}

public String getStudentName() {

return model.getName();

}

public void setStudentId(int id) {

model.setId(id);

}

public int getStudentId() {

return model.getId();

}

public void setStudentGrade(String grade) {

model.setGrade(grade);

}

public String getStudentGrade() {

return model.getGrade();

}

public void updateView() {

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

}

}

package MVCPattern;

public class StudentView {

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

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

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

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

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

}

}

package MVCPattern;

public class Main {

public static void main(String[] args) {

Student s=new Student();

s.setName("Tarun");

s.setId(191);

s.setGrade("A+");

StudentView sv=new StudentView();

StudentController control=new StudentController(s,sv);

control.updateView();

control.setStudentName("Tarun Karrthick");

control.setStudentGrade("O");

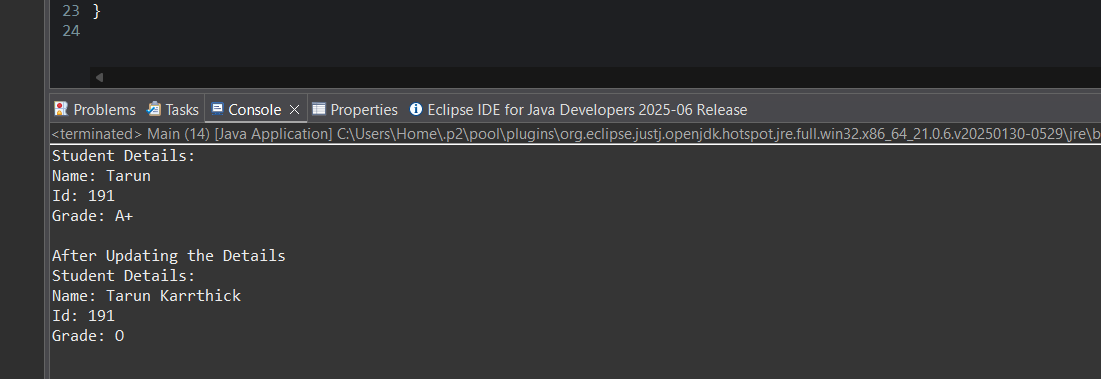
System.***out***.println("\nAfter Updating the Details");

control.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.

**Solution:**

package DependencyInjection;

public interface CustomerRepository {

String findCustomerById(String Id);

}

package DependencyInjection;

public class CustomerRepositoryImpl implements CustomerRepository{

public String findCustomerById(String Id) {

return "Customer [ ID = "+Id+", Reference\_Name = Virat]";

}

}

package DependencyInjection;

public class CustomerService {

private final CustomerRepository repo;

public CustomerService(CustomerRepository repo) {

this.repo=repo;

}

public void getCustomerDetails(String id) {

String customer = repo.findCustomerById(id);

System.***out***.println("Customer Details: " + customer);

}

}

package DependencyInjection;

public class Main {

public static void main(String[] args) {

CustomerRepository repo = new CustomerRepositoryImpl();

CustomerService ser= new CustomerService(repo);

ser.getCustomerDetails("191");

}

}

**Output:**

