Week 1 - Hands-On: Design Patterns

# Exercise 1: Singleton Pattern - Logger Utility

In this exercise, the Singleton Design Pattern was used to implement a centralized Logger class. This ensures that only one instance of Logger exists in the system.

* Key Concepts:
* - Private constructor and static instance.
* - Ensures consistent logging with one shared object.
* - Demonstrated using getInstance() method.

**CODE:**

**//Logger.java:**

public class Logger {

   private static Logger instance;

   private Logger() {

      System.out.println("Logger Initialized");

   }

   public static Logger getInstance() {

      if (instance == null) {

         instance = new Logger();

      }

      return instance;

   }

   public void log(String var1) {

      System.out.println("LOG: " + var1);

   }

}

//Main.java:

public class Main {

   public Main() {

   }

   public static void main(String[] var0) {

      Logger var1 = Logger.getInstance();

      var1.log("Starting Application");

      Logger var2 = Logger.getInstance();

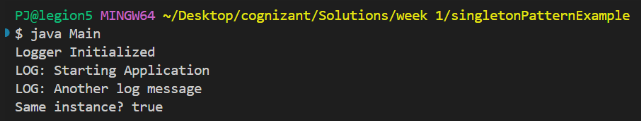
      var2.log("Another log message");

      System.out.println("Same instance? " + (var1 == var2));

   }

}

OUTPUT:



# Exercise 2: Factory Method Pattern - Document Creator

This exercise demonstrates the Factory Method Pattern by creating a document management system supporting Word, PDF, and Excel document creation using individual factory classes.

* Key Concepts:
* - Interface: Document, Concrete classes: WordDocument, PdfDocument, ExcelDocument.
* - Abstract factory class with createDocument().
* - Concrete factories implement the factory method for different types.

**CODE:**

**Document.java:**

public interface Document {

   void open();

}

**DocumentFactory.java:**

public abstract class DocumentFactory {

    public abstract Document createDocument();

}

**ExcelDocument.java:**

public class ExcelDocument implements Document {

    @Override

    public void open() {

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

    }

}

**ExcelDocumentFactory.java:**

public class ExcelDocumentFactory extends DocumentFactory {

    @Override

    public Document createDocument() {

        return new ExcelDocument();

    }

}

**//Main.java:**

public class Main {

    public static void main(String[] args) {

        DocumentFactory wordFactory = new WordDocumentFactory();

        Document word = wordFactory.createDocument();

        word.open();

        DocumentFactory pdfFactory = new PdfDocumentFactory();

        Document pdf = pdfFactory.createDocument();

        pdf.open();

        DocumentFactory excelFactory = new ExcelDocumentFactory();

        Document excel = excelFactory.createDocument();

        excel.open();

    }

}

**//PdfDocument.java:**

public class PdfDocument implements Document {

    @Override

    public void open() {

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

    }

}

**//PdfDocumentFactory.java:**

public class PdfDocumentFactory extends DocumentFactory {

    @Override

    public Document createDocument() {

        return new PdfDocument();

    }

}

**//WordDocument.java:**

public class WordDocument implements Document {

    @Override

    public void open() {

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

    }

}

**//WordDocumentFactory.java:**

public class WordDocumentFactory extends DocumentFactory {

    @Override

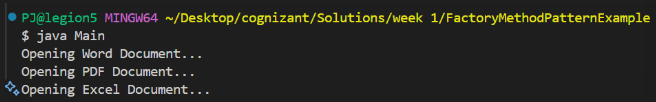
    public Document createDocument() {

        return new WordDocument();

    }

}

CODE:



# Exercise 3: Builder Pattern - Computer Configuration

This task uses the Builder Pattern to construct Computer objects with required and optional fields. The builder is a static nested class inside the Computer class.

* Key Concepts:
* - Static nested Builder class.
* - Required and optional fields handled separately.
* - Fluent API style with method chaining.

CODE:

// Computer.java

public class Computer {

// required parameters

private String HDD;

private String RAM;

// optional parameters

private boolean isGraphicsCardEnabled;

private boolean isBluetoothEnabled;

public String getHDD() {

return HDD;

}

public String getRAM() {

return RAM;

}

public boolean isGraphicsCardEnabled() {

return isGraphicsCardEnabled;

}

public boolean isBluetoothEnabled() {

return isBluetoothEnabled;

}

private Computer(ComputerBuilder builder) {

this.HDD = builder.HDD;

this.RAM = builder.RAM;

this.isGraphicsCardEnabled = builder.isGraphicsCardEnabled;

this.isBluetoothEnabled = builder.isBluetoothEnabled;

}

public static class ComputerBuilder {

// required parameters

private String HDD;

private String RAM;

// optional parameters

private boolean isGraphicsCardEnabled;

private boolean isBluetoothEnabled;

public ComputerBuilder(String hdd, String ram) {

this.HDD = hdd;

this.RAM = ram;

}

public ComputerBuilder setGraphicsCardEnabled(boolean isGraphicsCardEnabled) {

this.isGraphicsCardEnabled = isGraphicsCardEnabled;

return this;

}

public ComputerBuilder setBluetoothEnabled(boolean isBluetoothEnabled) {

this.isBluetoothEnabled = isBluetoothEnabled;

return this;

}

public Computer build() {

return new Computer(this);

}

}

}

// Main.java:

public class Main {

public static void main(String[] args) {

Computer comp = new Computer.ComputerBuilder("500 GB", "2 GB")

.setBluetoothEnabled(true)

.setGraphicsCardEnabled(true)

.build();

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

System.out.println("HDD: " + comp.getHDD());

System.out.println("RAM: " + comp.getRAM());

System.out.println("Bluetooth Enabled: " + comp.isBluetoothEnabled());

System.out.println("Graphics Card Enabled: " + comp.isGraphicsCardEnabled());

}

}

//Computer$Builder.java:

public class Computer$Builder {

   private String CPU;

   private String RAM;

   private String storage;

   private String graphicsCard;

   private String operatingSystem;

   public Computer$Builder(String var1, String var2) {

      this.CPU = var1;

      this.RAM = var2;

   }

   public Computer$Builder setStorage(String var1) {

      this.storage = var1;

      return this;

   }

   public Computer$Builder setGraphicsCard(String var1) {

      this.graphicsCard = var1;

      return this;

   }

   public Computer$Builder setOperatingSystem(String var1) {

      this.operatingSystem = var1;

      return this;

   }

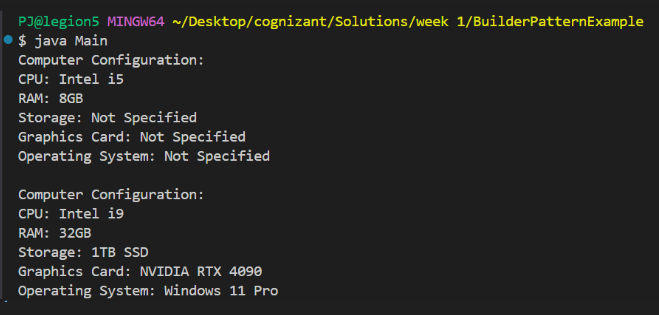
   public Computer build() {

      return new Computer(this);

   }

}

OUTPUT:



# Exercise 4: Adapter Pattern - Payment Gateway Integration

We used the Adapter Pattern to integrate multiple third-party payment systems like PayPal and Stripe that have different interfaces, under a common interface PaymentProcessor.

* Key Concepts:
* - Target interface: PaymentProcessor.
* - Adaptee classes: PayPalGateway, StripeGateway.
* - Adapter classes translate calls to compatible forms.

CODE:

**CODE:**

**//Main.java:**

public class Main {

   public Main() {

   }

   public static void main(String[] var0) {

      PayPalAdapter var1 = new PayPalAdapter(new PayPalGateway());

      var1.processPayment(150.75);

      StripeAdapter var2 = new StripeAdapter(new StripeGateway());

      var2.processPayment(300.0);

   }

}

**//PaymentProcessor.java:**

public interface PaymentProcessor {

   void processPayment(double var1);

}

public class PayPalAdapter implements PaymentProcessor {

   private PayPalGateway paypal;

   public PayPalAdapter(PayPalGateway var1) {

      this.paypal = var1;

   }

   public void processPayment(double var1) {

      this.paypal.sendPayment(var1);

   }

}

**// PayPalGateway.java:**

public class PayPalGateway {

   public PayPalGateway() {

   }

   public void sendPayment(double var1) {

      System.out.println("Processing PayPal payment of $" + var1);

   }

}

//StripeAdapter.java:

public class StripeAdapter implements PaymentProcessor {

   private StripeGateway stripe;

   public StripeAdapter(StripeGateway var1) {

      this.stripe = var1;

   }

   public void processPayment(double var1) {

      this.stripe.makePayment(var1);

   }

}

**// StripeGateway.java:**

public class StripeGateway {

   public StripeGateway() {

   }

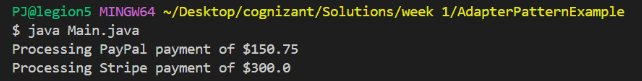
   public void makePayment(double var1) {

      System.out.println("Processing Stripe payment of $" + var1);

   }

}

OUTPUT:



# Exercise 5: Decorator Pattern - Notification System

This exercise used the Decorator Pattern to send notifications through multiple channels (Email, SMS, Slack) by wrapping functionalities around a base notifier dynamically.

* Key Concepts:
* - Component interface: Notifier.
* - Concrete component: EmailNotifier.
* - Abstract decorator class and dynamic functionality extension.

//Image.java:

public interface Image {

    void display();

}

//Main.java:

public class Main {

    public static void main(String[] args) {

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

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

        // Image not loaded yet

        System.out.println("First time display of image1:");

        image1.display();  // Loads and displays

        System.out.println("\nSecond time display of image1:");

        image1.display();  // Only displays, no loading

        System.out.println("\nDisplay image2:");

        image2.display();  // Loads and displays

    }

}

//ProxyImage.java:

public class ProxyImage implements Image {

    private RealImage realImage;

    private String filename;

    public ProxyImage(String filename) {

        this.filename = filename;

    }

    @Override

    public void display() {

        if (realImage == null) {

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

        }

        realImage.display(); // Uses cached object

    } }

//RealImage.java:

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);

    }

    @Override

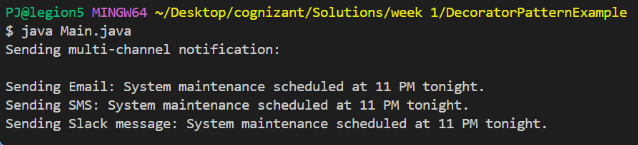
    public void display() {

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

    }

}

OUTPUT:



# Exercise 6: Proxy Pattern – Image Viewer with Lazy Initialization

Objective:  
To implement a proxy that loads images only when needed and reuses cached instances to avoid repeated loading.

Scenario:  
You are building an image viewer app where images are loaded from a remote server. Loading can be expensive, so you use a proxy to delay it (lazy loading) and avoid reloading.

Key Concepts:

* Image interface defines display()
* RealImage loads the image from a "remote server" and displays it
* ProxyImage wraps RealImage, delays its creation, and caches it
* The proxy avoids unnecessary re-loading.

Benefits:

* Saves memory
* Improves performance
* Adds security/control around object creation

CODE:

//Image.java:

public interface Image {

    void display();

}

//Main.java:

public class Main {

    public static void main(String[] args) {

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

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

        // Image not loaded yet

        System.out.println("First time display of image1:");

        image1.display();  // Loads and displays

        System.out.println("\nSecond time display of image1:");

        image1.display();  // Only displays, no loading

        System.out.println("\nDisplay image2:");

        image2.display();  // Loads and displays

    }

}

//ProxyImage.java:

public class ProxyImage implements Image {

    private RealImage realImage;

    private String filename;

    public ProxyImage(String filename) {

        this.filename = filename;

    }

    @Override

    public void display() {

        if (realImage == null) {

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

        }

        realImage.display(); // Uses cached object

    }

}

//RelImage.java:

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);

    }

    @Override

    public void display() {

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

    }

}

OUTPUT:



# Exercise 7: Observer Pattern – Stock Market Monitoring System

Objective:  
Notify multiple clients (observers) when a stock price is updated in real-time.

Scenario:  
In a stock market app, various platforms (like mobile and web) subscribe to receive updates when stock prices change.

Key Concepts:

* Stock interface defines register/remove/notify methods
* StockMarket is the subject holding a list of observers
* Observer interface defines update()
* MobileApp and WebApp are observers
* When a stock changes, all observers are notified

Benefits:

* Loosely coupled design
* Real-time update support
* Easily extensible with new observer types

CODE:

//Main.java:

public class Main {

    public static void main(String[] args) {

        StockMarket stockMarket = new StockMarket();

        Observer mobileUser = new MobileApp("Alice");

        Observer webUser = new WebApp("Bob");

        stockMarket.registerObserver(mobileUser);

        stockMarket.registerObserver(webUser);

        stockMarket.setStock("TCS", 3500.00);

        System.out.println();

        stockMarket.setStock("INFY", 1450.50);

}

}

//MobileApp.java;

public class MobileApp implements Observer {

    private String user;

    public MobileApp(String user) {

        this.user = user;

    }

    @Override

    public void update(String stockName, double newPrice) {

        System.out.println("MobileApp [" + user + "]: " + stockName + " updated to $" + newPrice);

}

}

//Observer.java:

public interface Observer {

    void update(String stockName, double stockPrice);

}

//Stock.java:

public interface Stock {

    void registerObserver(Observer observer);

    void removeObserver(Observer observer);

    void notifyObservers();

}

//Stock.java:

public interface Stock {

    void registerObserver(Observer observer);

    void removeObserver(Observer observer);

    void notifyObservers();

}

//StockMarket.java:

import java.util.ArrayList;

import java.util.List;

public class StockMarket implements Stock {

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

    private String stockName;

    private double stockPrice;

    public void setStock(String stockName, double newPrice) {

        this.stockName = stockName;

        this.stockPrice = newPrice;

        notifyObservers();

    }

    @Override

    public void registerObserver(Observer observer) {

        observers.add(observer);

    }

    @Override

    public void removeObserver(Observer observer) {

        observers.remove(observer);

    }

    @Override

    public void notifyObservers() {

        for (Observer o : observers) {

            o.update(stockName, stockPrice);

        }

    }

}

//WebApp.java:

public class WebApp implements Observer {

    private String user;

    public WebApp(String user) {

        this.user = user;

    }

    @Override

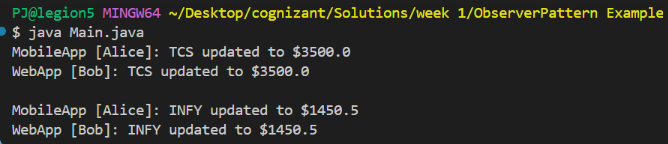
    public void update(String stockName, double newPrice) {

        System.out.println("WebApp [" + user + "]: " + stockName + " updated to $" + newPrice);

    }

}

OUTPUT:



# Exercise 8: Strategy Pattern – Dynamic Payment System

Objective:  
To implement multiple payment methods (like Credit Card, PayPal) and choose between them at runtime using the Strategy Pattern.

Scenario:  
A payment platform allows users to select their preferred payment method. The system uses a strategy interface to switch between strategies without modifying business logic.

Key Concepts:

* PaymentStrategy interface defines the pay() method.
* CreditCardPayment and PayPalPayment implement the interface.
* PaymentContext holds a reference to the strategy and executes it.
* The strategy can be swapped at runtime.

Benefits:

* Allows dynamic switching of algorithms (strategies)
* Promotes open/closed principle
* Makes payment methods extensible

**CODE:**

// code name: PaymentStrategy.java

public interface PaymentStrategy {

void pay(double amount);

}

// code name: CreditCardPayment.java

public class CreditCardPayment implements PaymentStrategy {

private String cardNumber;

private String name;

private String cvv;

public CreditCardPayment(String cardNumber, String name, String cvv) {

this.cardNumber = cardNumber;

this.name = name;

this.cvv = cvv;

}

@Override

public void pay(double amount) {

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

}

}

// code name: PayPalPayment.java

public class PayPalPayment implements PaymentStrategy {

private String email;

private String password;

public PayPalPayment(String email, String password) {

this.email = email;

this.password = password;

}

@Override

public void pay(double amount) {

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

}

}

// code name: PaymentContext.java

public class PaymentContext {

private PaymentStrategy strategy;

public void setPaymentStrategy(PaymentStrategy strategy) {

this.strategy = strategy;

}

public void pay(double amount) {

if (strategy == null) {

throw new IllegalStateException("Payment strategy not set.");

}

strategy.pay(amount);

}

}

// code name: Main.java

import java.util.Scanner;

public class Main {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

PaymentContext context = new PaymentContext();

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

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

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

int choice = scanner.nextInt();

scanner.nextLine(); // consume newline

switch (choice) {

case 1:

System.out.print("Enter Card Number: ");

String cardNumber = scanner.nextLine();

System.out.print("Enter Name on Card: ");

String name = scanner.nextLine();

System.out.print("Enter CVV: ");

String cvv = scanner.nextLine();

context.setPaymentStrategy(new CreditCardPayment(cardNumber, name, cvv));

break;

case 2:

System.out.print("Enter PayPal Email: ");

String email = scanner.nextLine();

System.out.print("Enter PayPal Password: ");

String password = scanner.nextLine();

context.setPaymentStrategy(new PayPalPayment(email, password));

break;

default:

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

return;

}

System.out.print("Enter Amount to Pay: ");

double amount = scanner.nextDouble();

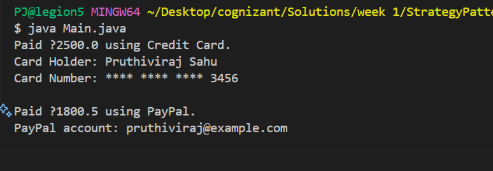
context.pay(amount);

scanner.close();

}

}

CODE:



# Exercise 9: Command Pattern – Device Control via Remote

Objective:  
To implement a command pattern for turning devices like lights on/off via remote control buttons.

Scenario:  
In a home automation system, pressing a remote button sends a command to the light. The command encapsulates the request, allowing the remote to be decoupled from the actual action logic.

Key Concepts:

* Command interface defines execute()
* Light is the receiver with methods turnOn() and turnOff()
* LightOnCommand and LightOffCommand implement Command
* RemoteControl invokes commands

Benefits:

* Decouples invoker from receiver
* Supports undo/redo functionality
* Commands can be stored, logged, or queued

CODE:

**// Command.java:**

public interface Command {

void execute();

}

**// Light.java:**

public class Light {

public void turnOn() {

System.out.println("The light is on");

}

public void turnOff() {

System.out.println("The light is off");

}

}

**// LightOnCommand.java:**

public class LightOnCommand implements Command {

private Light light;

public LightOnCommand(Light light) {

this.light = light;

}

@Override

public void execute() {

light.turnOn();

}

}

**// LightOffCommand.java:**

public class LightOffCommand implements Command {

private Light light;

public LightOffCommand(Light light) {

this.light = light;

}

@Override

public void execute() {

light.turnOff();

}

}

**// RemoteControl.java:**

public class RemoteControl {

private Command command;

public void setCommand(Command command) {

this.command = command;

}

public void pressButton() {

command.execute();

}

public static void main(String[] args) {

Light light = new Light();

Command lightsOn = new LightOnCommand(light);

Command lightsOff = new LightOffCommand(light);

RemoteControl remote = new RemoteControl();

remote.setCommand(lightsOn);

remote.pressButton(); // Output: The light is on

remote.setCommand(lightsOff);

remote.pressButton(); // Output: The light is off

}

}

**//Main.java:**

package command;

public class Main {

    public static void main(String[] args) {

        Light livingRoomLight = new Light();

        Command lightOn = new LightOnCommand(livingRoomLight);

        Command lightOff = new LightOffCommand(livingRoomLight);

        RemoteControl remote = new RemoteControl();

        System.out.println("Turning the light ON:");

        remote.setCommand(lightOn);

        remote.pressButton();

        System.out.println("\nTurning the light OFF:");

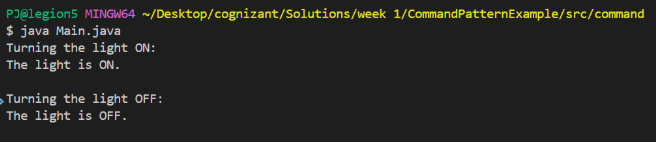
        remote.setCommand(lightOff);

        remote.pressButton();

    }

}

OUTPUT:



# Exercise 10: MVC Pattern – Student Record Web Application

Objective:  
Separate concerns by applying the Model-View-Controller pattern to manage and display student data.

Scenario:  
In a student management system, you need to separate data (model), display (view), and logic (controller) so the UI and backend logic can evolve independently.

Key Concepts:

* Student is the model with name, ID, grade
* StudentView displays student info
* StudentController controls how data is passed and updated
* Main simulates student updates and view refreshes

Benefits:

* Clear separation of responsibilities
* UI changes don’t affect business logic
* Promotes maintainable and testable code

**CODE:**

Main.java:

package mvc;

public class Main {

    public static void main(String[] args) {

        // Create model and view

        Student student = new Student("Pruthiviraj", "S123", "A");

        StudentView view = new StudentView();

        // Create controller

        StudentController controller = new StudentController(student, view);

        // Display initial data

        controller.updateView();

        System.out.println("\nUpdating student details...\n");

        // Update data via controller

        controller.setStudentName("Raj Sahu");

        controller.setStudentGrade("A+");

        // Display updated data

        controller.updateView();

    }

}

//Student.java:

package mvc;

public class Student {

    private String name;

    private String id;

    private String grade;

    // Constructor

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

        this.name = name;

        this.id = id;

        this.grade = grade;

    }

    // Getters and Setters

    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;

    }}

//StudentController.java:

package mvc;

public class StudentController {

    private Student model;

    private StudentView view;

    public StudentController(Student model, StudentView view) {

        this.model = model;

        this.view = view;

    }

    // Update model data

    public void setStudentName(String name) {

        model.setName(name);

    }

    public void setStudentId(String id) {

        model.setId(id);

    }

    public void setStudentGrade(String grade) {

        model.setGrade(grade);

    }

    // Retrieve model data

    public String getStudentName() {

        return model.getName();

    }

    public String getStudentId() {

        return model.getId();

    }

    public String getStudentGrade() {

        return model.getGrade();

    }

    // Display via view

    public void updateView() {

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

    }

}

//StudentView.java:

package mvc;

public class StudentView {

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

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

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

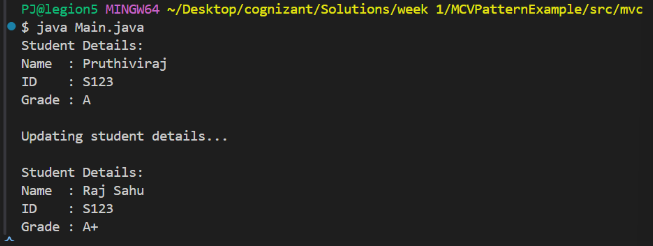
        System.out.println("ID    : " + id);

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

    }

}

OUTPUT:



# Exercise 11: Dependency Injection – Customer Management

**Objective:**  
Decouple the service class from the repository class using constructor-based Dependency Injection.

**Scenario:**  
In a customer management app, the service depends on a repository to fetch customer data. Instead of tightly coupling them, you inject the repository dependency into the service.

**Key Concepts:**

* CustomerRepository is the interface
* CustomerRepositoryImpl provides in-memory data
* CustomerService uses DI to access repository
* Main demonstrates the wiring and logic

**Benefits:**

* Loose coupling and flexibility
* Easy to substitute mock data for testing
* Scalable for real-world apps

**CODE:**

**Main.java:**

package di;

public class Main {

    public static void main(String[] args) {

        // Manual Dependency Injection

        CustomerRepository repository = new CustomerRepositoryImpl();

        CustomerService service = new CustomerService(repository);

        // Use the service

        System.out.println("Fetching Customer C101:");

        service.printCustomerDetails("C101");

        System.out.println("\nFetching Customer C999:");

        service.printCustomerDetails("C999");

    }

}

**//CustomerService.java:**

package di;

public class CustomerService {

    private final CustomerRepository customerRepository;

    // Constructor Injection

    public CustomerService(CustomerRepository customerRepository) {

        this.customerRepository = customerRepository;

    }

    public void printCustomerDetails(String id) {

        Customer customer = customerRepository.findCustomerById(id);

        if (customer != null) {

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

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

        } else {

            System.out.println("Customer not found for ID: " + id);

        }

    }

}

**//StudentView.java:**

package di;

public class Customer {

    private String id;

    private String name;

    public Customer(String id, String name) {

        this.id = id;

        this.name = name;

    }

    public String getId() {

        return id;

    }

    public String getName() {

        return name;

    }

}

CODE:

