MANDATORY HANDS-ON

WEEK-1

**DESIGN PRINCIPLES AND PATTERNS**

**Exercise 1: Implementing the Singleton Pattern**

**Java Project Name: SingletonPatternExample**

**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 message) {

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

    }

}

**Main.java**

public class Main {

    public static void main(String[] args) {

        Logger logger1 = Logger.getInstance();

        logger1.log("Logging message from logger1.");

        Logger logger2 = Logger.getInstance();

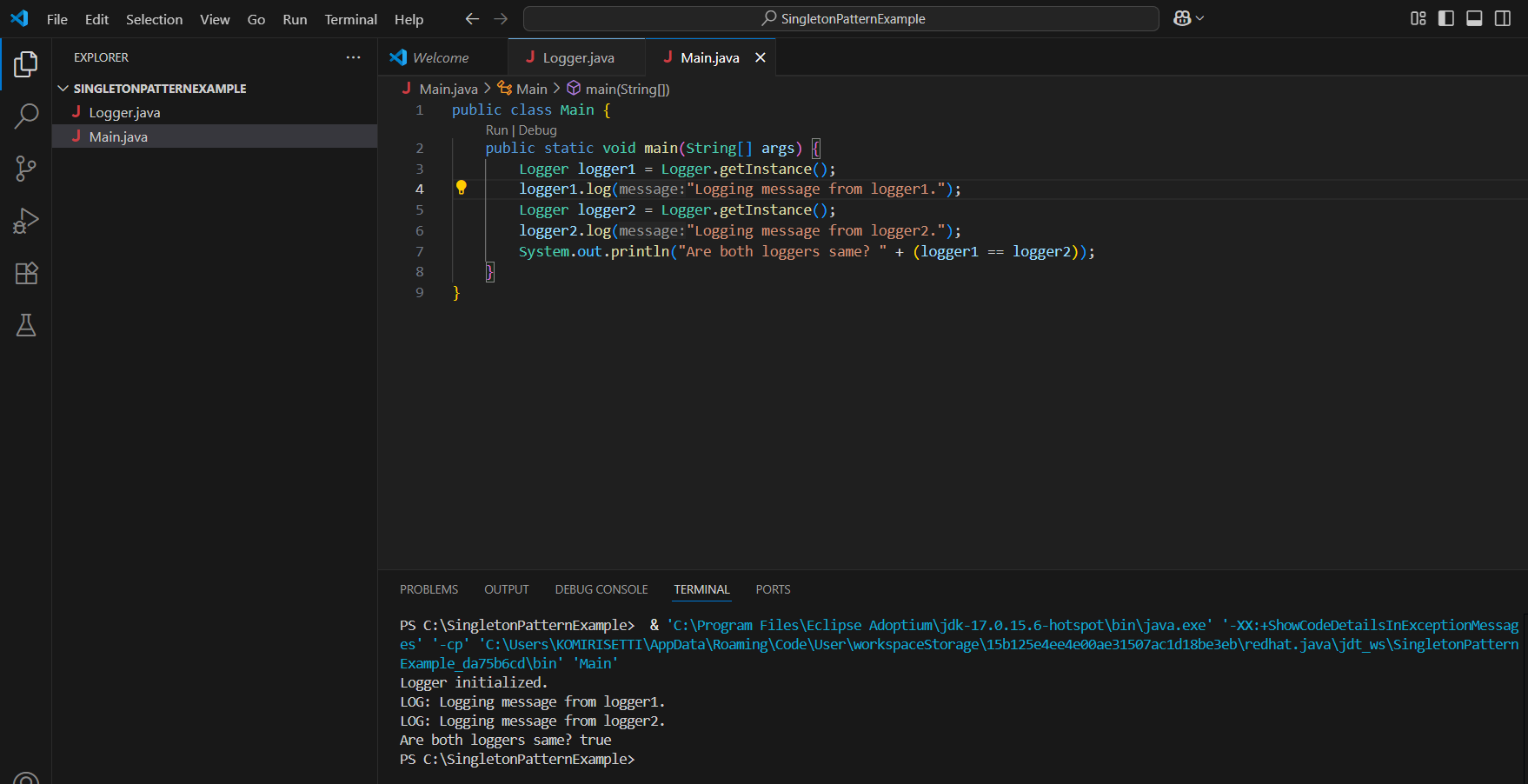
        logger2.log("Logging message from logger2.");

        System.out.println("Are both loggers same? " + (logger1 == logger2));

    }

}

**Output:**

****

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

**Java Project Name: FactoryMethodPatternExample**

**Document.java**

interface Document {

    void create();

}

**WordDocument.java**

class WordDocument implements Document {

    public void create() {

        System.out.println("Creating a Word document...");

    }

}

**PdfDocument.java**

class PdfDocument implements Document {

    public void create() {

        System.out.println("Creating a PDF document...");

    }

}

**ExcelDocument.java**

class ExcelDocument implements Document {

    public void create() {

        System.out.println("Creating an Excel document...");

    }

}

**DocumentFactory.java**

abstract class DocumentFactory {

    abstract Document createDocument();

}

**WordDocumentFactory.java**

class WordDocumentFactory extends DocumentFactory {

    Document createDocument() {

        return new WordDocument();

    }

}

**PdfDocumentFactory.java**

class PdfDocumentFactory extends DocumentFactory {

    Document createDocument() {

        return new PdfDocument();

    }

}

**ExcelDocumentFactory.java**

class ExcelDocumentFactory extends DocumentFactory {

    Document createDocument() {

        return new ExcelDocument();

    }

}

**Main.java**

class FactoryMain {

    public static void main(String[] args) {

        DocumentFactory wordFactory = new WordDocumentFactory();

        Document wordDoc = wordFactory.createDocument();

        wordDoc.create();

        DocumentFactory pdfFactory = new PdfDocumentFactory();

        Document pdfDoc = pdfFactory.createDocument();

        pdfDoc.create();

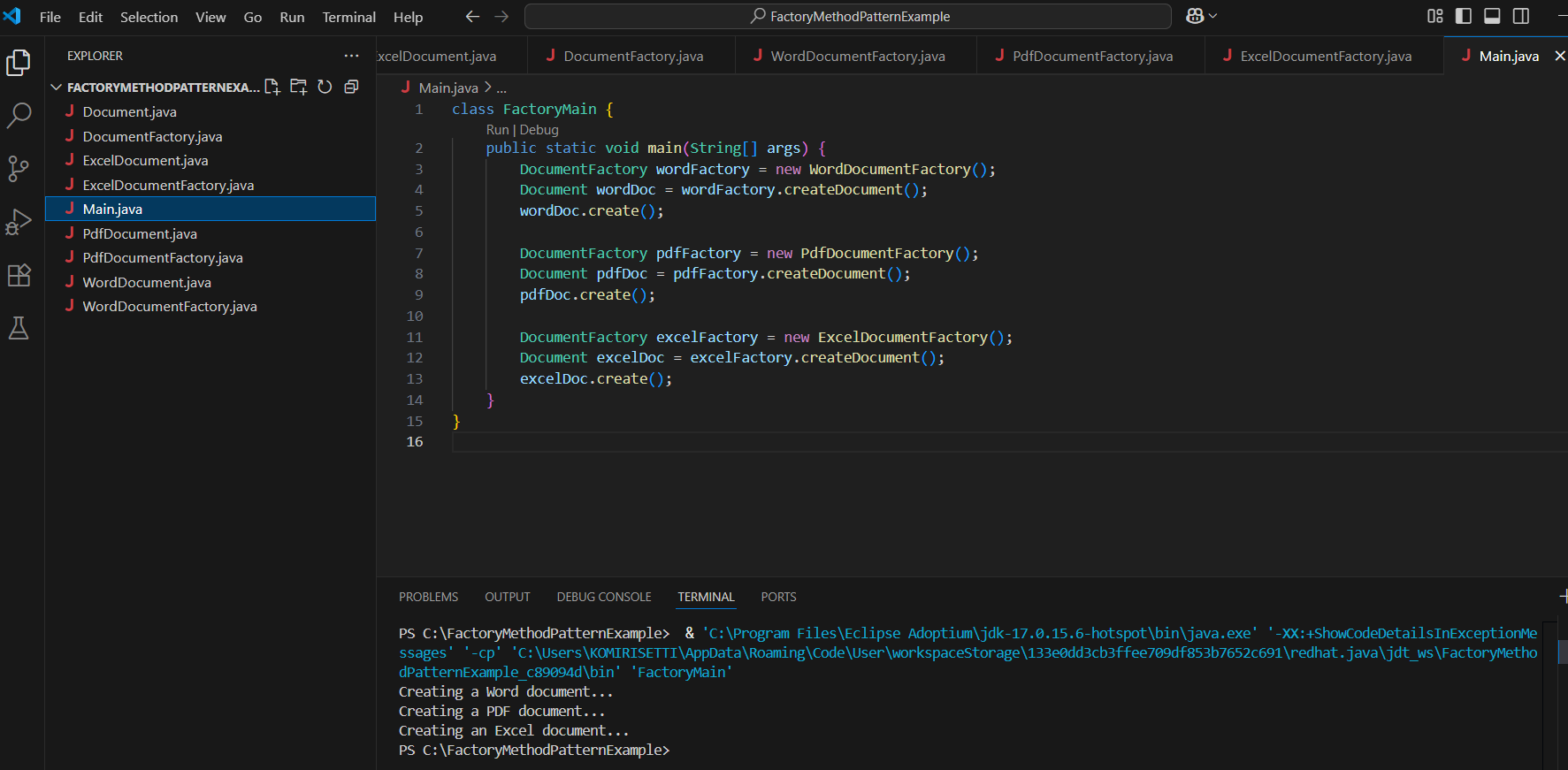
        DocumentFactory excelFactory = new ExcelDocumentFactory();

        Document excelDoc = excelFactory.createDocument();

        excelDoc.create();

    }

}

**Output:**

**DESIGN PRINCIPLES AND PATTERNS**

**(Other Hands-on exercises of design principles and patterns)**

**Exercise 3: Implementing the Builder Pattern**

**Java Project Name : BuilderPatternExample**

**Computer.java**

class Computer {

    private String cpu;

    private String ram;

    private String storage;

    private Computer(Builder builder) {

        this.cpu = builder.cpu;

        this.ram = builder.ram;

        this.storage = builder.storage;

    }

    public void display() {

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

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

        System.out.println("Storage: " + storage);

    }

    public static class Builder {

        private String cpu;

        private String ram;

        private String storage;

        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 Computer build() {

            return new Computer(this);

        }

    }

}

**Main.java**

class BuilderMain {

    public static void main(String[] args) {

        Computer highEndPC = new Computer.Builder()

            .setCPU("Intel i7")

            .setRAM("16GB")

            .setStorage("512GB SSD")

            .build();

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

        highEndPC.display();

        Computer basicPC = new Computer.Builder()

            .setCPU("AMD Ryzen 5")

            .setRAM("8GB")

            .setStorage("256GB SSD")

            .build();

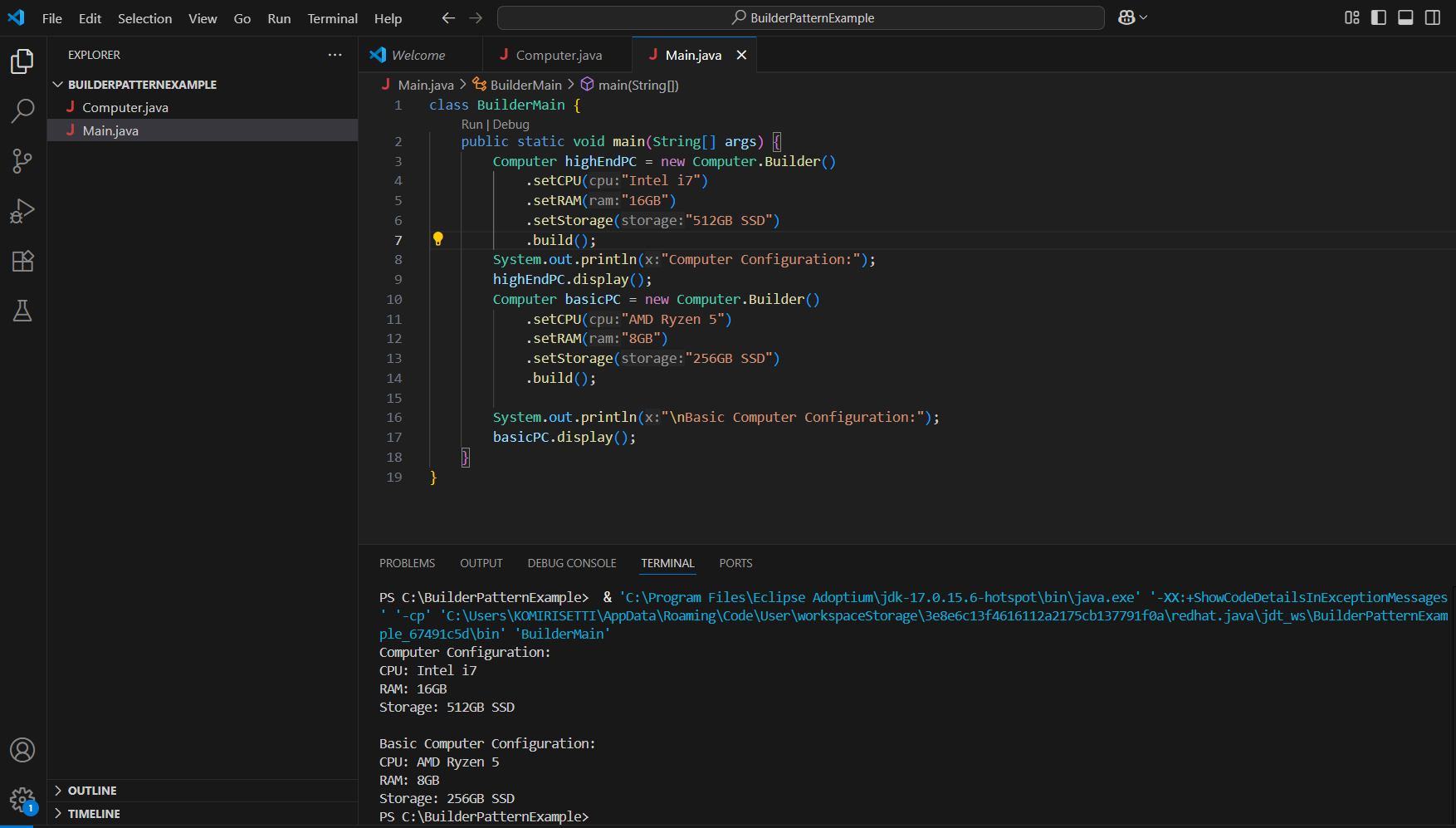
        System.out.println("\nBasic Computer Configuration:");

        basicPC.display();

    }

}

**Output:**

****

**Exercise 4: Implementing the Adapter Pattern**

**Java Project Name : AdapterPatternExample**

**PaymentProcessor.java**

public interface PaymentProcessor {

    void processPayment(double amount);

}

**PayPalGateway.java**

public class PayPalGateway {

    public void sendPayment(double amount) {

        System.out.println("Paid " + amount + " via PayPal.");

    }

}

**StripeGateway.java**

public class StripeGateway {

    public void makePayment(double amount) {

        System.out.println("Paid " + amount + " via Stripe.");

    }

}

**PayPalAdapter.java**

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**

public class StripeAdapter implements PaymentProcessor {

    private StripeGateway stripe;

    public StripeAdapter(StripeGateway stripe) {

        this.stripe = stripe;

    }

    @Override

    public void processPayment(double amount) {

        stripe.makePayment(amount);

    }

}

**Main.java**

public class Main {

    public static void main(String[] args) {

        PaymentProcessor paypalProcessor = new PayPalAdapter(new PayPalGateway());

        PaymentProcessor stripeProcessor = new StripeAdapter(new StripeGateway());

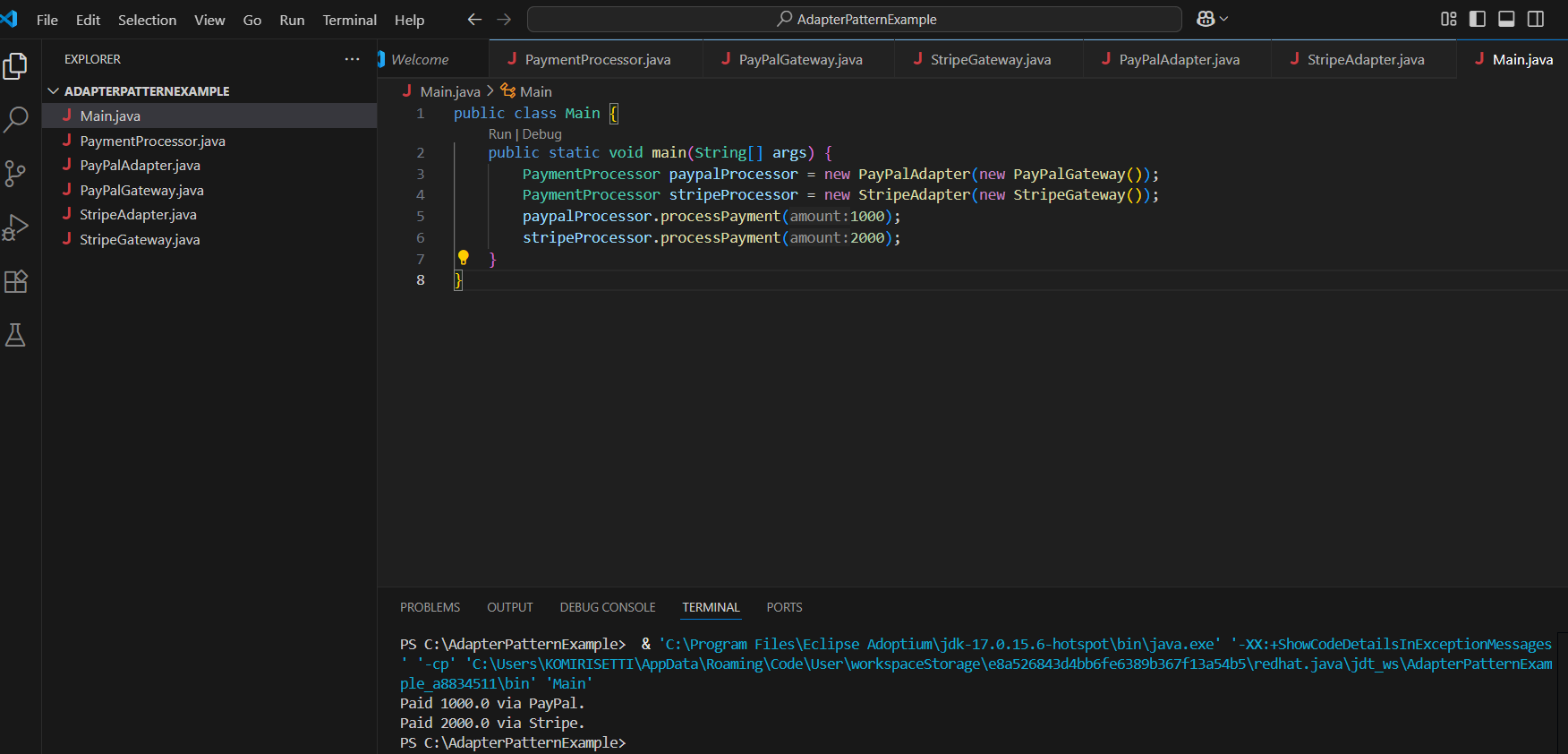
        paypalProcessor.processPayment(1000);

        stripeProcessor.processPayment(2000);

    }

}

**Output:**

****

**Exercise 5: Implementing the Decorator Pattern**

**Java Project Name : DecoratorPatternExample**

**Notifier.java**

public interface Notifier {

    void send(String message);

}

**EmailNotifier.java**

public class EmailNotifier implements Notifier {

    @Override

    public void send(String message) {

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

    }

}

**NotifierDecorator.java**

public abstract class NotifierDecorator implements Notifier {

    protected Notifier wrappedNotifier;

    public NotifierDecorator(Notifier notifier) {

        this.wrappedNotifier = notifier;

    }

    public void send(String message) {

        wrappedNotifier.send(message);

    }

}

**SMSNotifierDecorator.java**

public class SMSNotifierDecorator extends NotifierDecorator {

    public SMSNotifierDecorator(Notifier notifier) {

        super(notifier);

    }

    @Override

    public void send(String message) {

        super.send(message);

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

    }

}

**SlackNotifierDecorator.java**

public class SlackNotifierDecorator extends NotifierDecorator {

    public SlackNotifierDecorator(Notifier notifier) {

        super(notifier);

    }

    @Override

    public void send(String message) {

        super.send(message);

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

    }

}

**Main.java**

public class Main {

    public static void main(String[] args) {

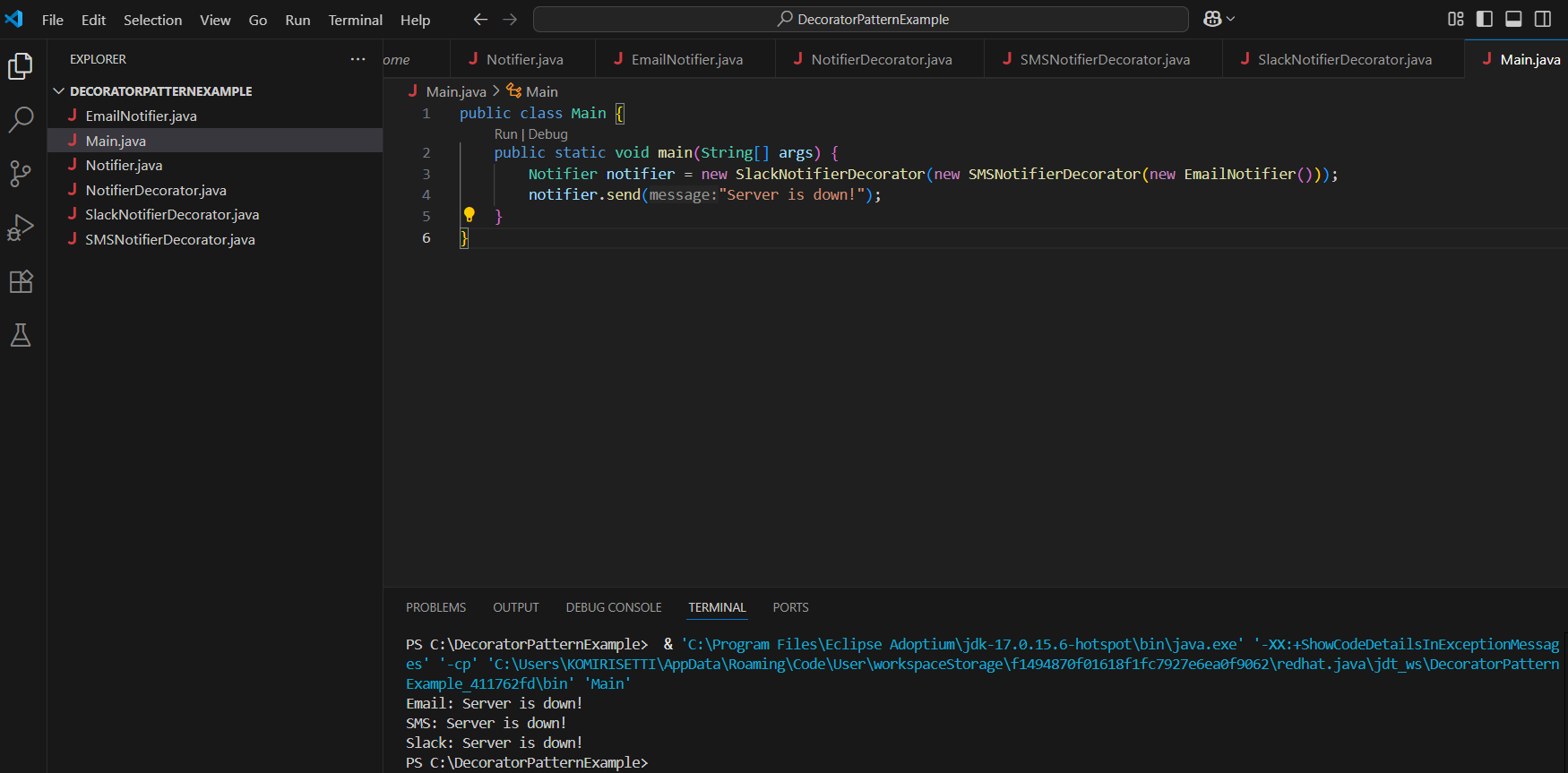
        Notifier notifier = new SlackNotifierDecorator(new SMSNotifierDecorator(new EmailNotifier()));

        notifier.send("Server is down!");

    }

}

**Output:**

****

**Exercise 6: Implementing the Proxy Pattern**

**Java Project Name : ProxyPatternExample**

**Image.java**

public interface Image {

    void display();

}

**RealImage.java**

public class RealImage implements Image {

    private String fileName;

    public RealImage(String fileName) {

        this.fileName = fileName;

        loadFromDisk();

    }

    private void loadFromDisk() {

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

    }

    @Override

    public void display() {

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

    }

}

**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);

        }

        realImage.display();

    }

}

**Main.java**

public class Main {

    public static void main(String[] args) {

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

        System.out.println("First call:");

        image.display();

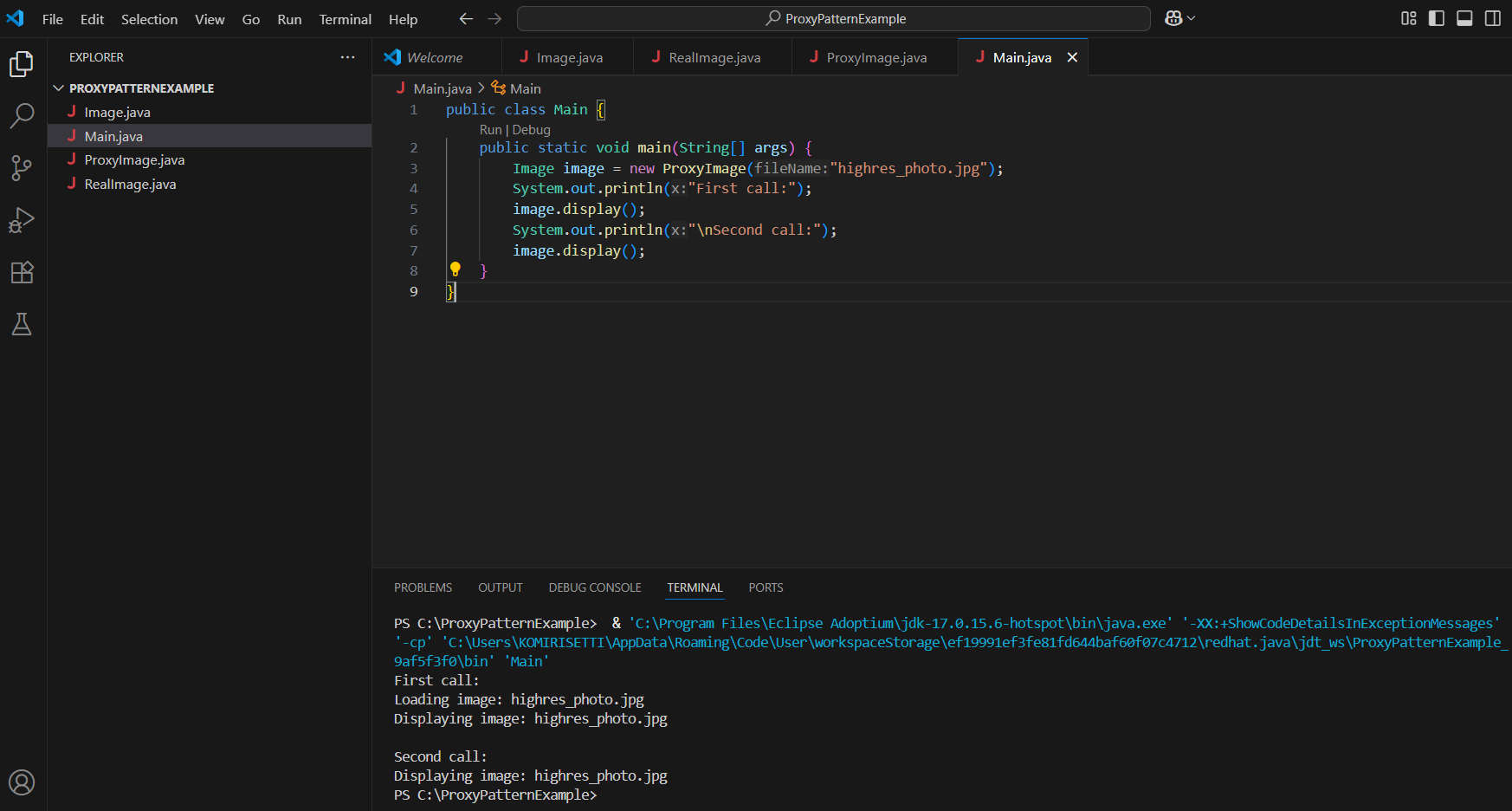
        System.out.println("\nSecond call:");

        image.display();

    }

}

**Output:**

****

**Exercise 7: Implementing the Observer Pattern**

**Java Project Name : ObserverPatternExample**

**Observer.java**

public interface Observer {

    void update(String stockName, double stockPrice);

}

**Stock.java**

public interface Stock {

    void registerObserver(Observer o);

    void removeObserver(Observer o);

    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 setStockPrice(String stockName, double stockPrice) {

        this.stockName = stockName;

        this.stockPrice = stockPrice;

        notifyObservers();

    }

    @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);

        }

    }

}

**MobileApp.java**

public class MobileApp implements Observer {

    private String appName;

    public MobileApp(String appName) {

        this.appName = appName;

    }

    @Override

    public void update(String stockName, double stockPrice) {

        System.out.println("[" + appName + "] Stock Update -> " + stockName + ": Rs." + stockPrice);

    }

}

**WebApp.java**

public class WebApp implements Observer {

    private String webName;

    public WebApp(String webName) {

        this.webName = webName;

    }

    @Override

    public void update(String stockName, double stockPrice) {

        System.out.println("[" + webName + "] Stock Update -> " + stockName + ": Rs." + stockPrice);

    }

}

**Main.java**

public class Main {

    public static void main(String[] args) {

        StockMarket stockMarket = new StockMarket();

        Observer mobile = new MobileApp("Phone Tracker");

        Observer web = new WebApp("Browser Monitor");

        stockMarket.registerObserver(mobile);

        stockMarket.registerObserver(web);

        stockMarket.setStockPrice("TCS", 3645.50);

        stockMarket.setStockPrice("Infosys", 1452.75);

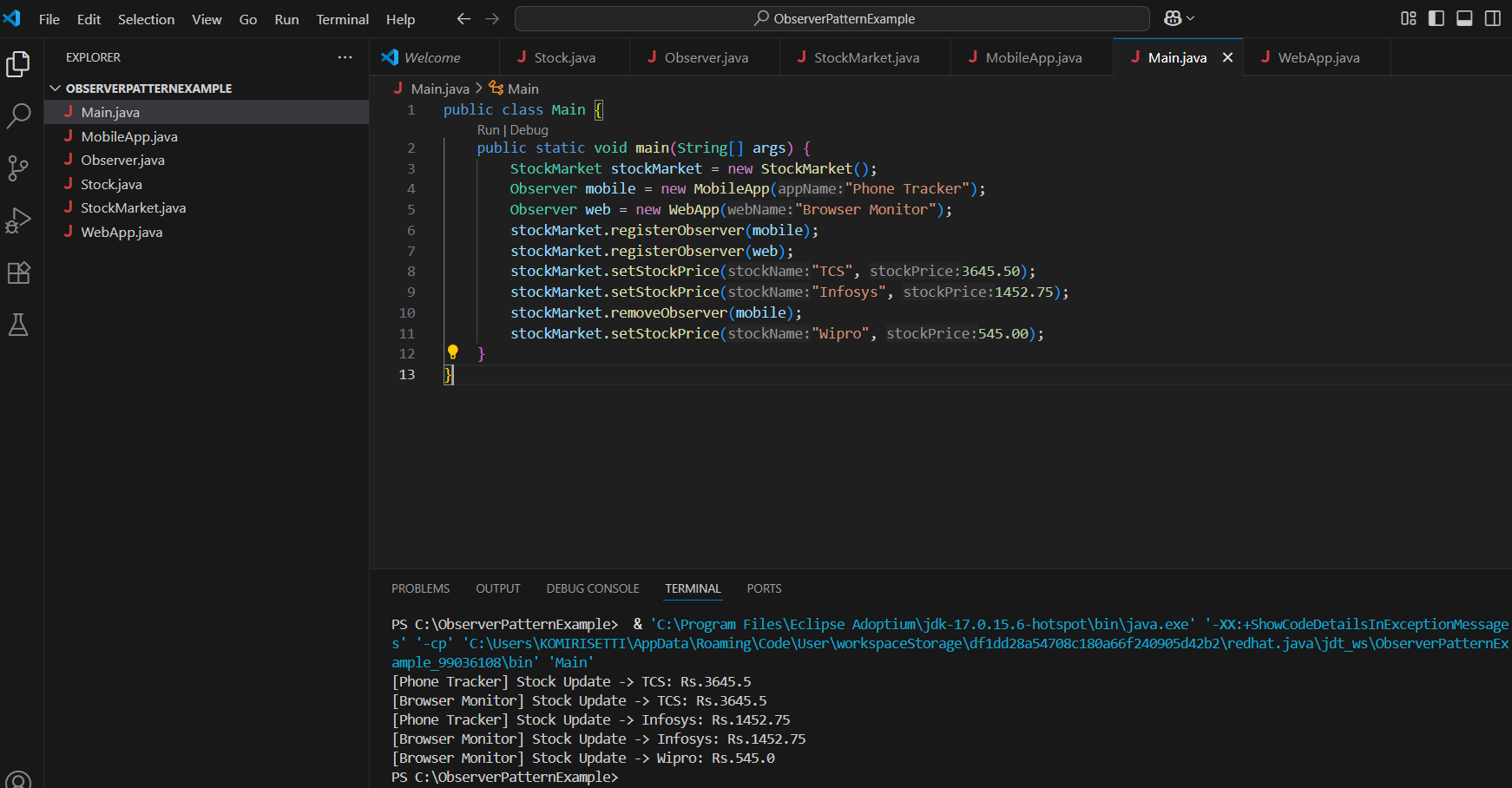
        stockMarket.removeObserver(mobile);

        stockMarket.setStockPrice("Wipro", 545.00);

    }

}

**Output:**

****

**Exercise 8: Implementing the Strategy Pattern**

**Java Project Name : StrategyPatternExample**

**PaymentStrategy.java**

public interface PaymentStrategy {

    void pay(double amount);

}

**CreditCardPayment.java**

public class CreditCardPayment implements PaymentStrategy {

    private String cardHolderName;

    public CreditCardPayment(String cardHolderName) {

        this.cardHolderName = cardHolderName;

    }

    @Override

    public void pay(double amount) {

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

    }

}

**PayPalPayment.java**

public class PayPalPayment implements PaymentStrategy {

    private String email;

    public PayPalPayment(String email) {

        this.email = email;

    }

    @Override

    public void pay(double amount) {

        System.out.println("Paid Rs." + amount + " using PayPal (" + email + ").");

    }

}

**PaymentContext.java**

public class PaymentContext {

    private PaymentStrategy paymentStrategy;

    public void setPaymentStrategy(PaymentStrategy paymentStrategy) {

        this.paymentStrategy = paymentStrategy;

    }

    public void makePayment(double amount) {

        if (paymentStrategy == null) {

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

            return;

        }

        paymentStrategy.pay(amount);

    }

}

**Main.java**

public class Main {

    public static void main(String[] args) {

        PaymentContext context = new PaymentContext();

        PaymentStrategy creditCard = new CreditCardPayment("J K");

        context.setPaymentStrategy(creditCard);

        context.makePayment(1000.0);

        PaymentStrategy paypal = new PayPalPayment("jk@example.com");

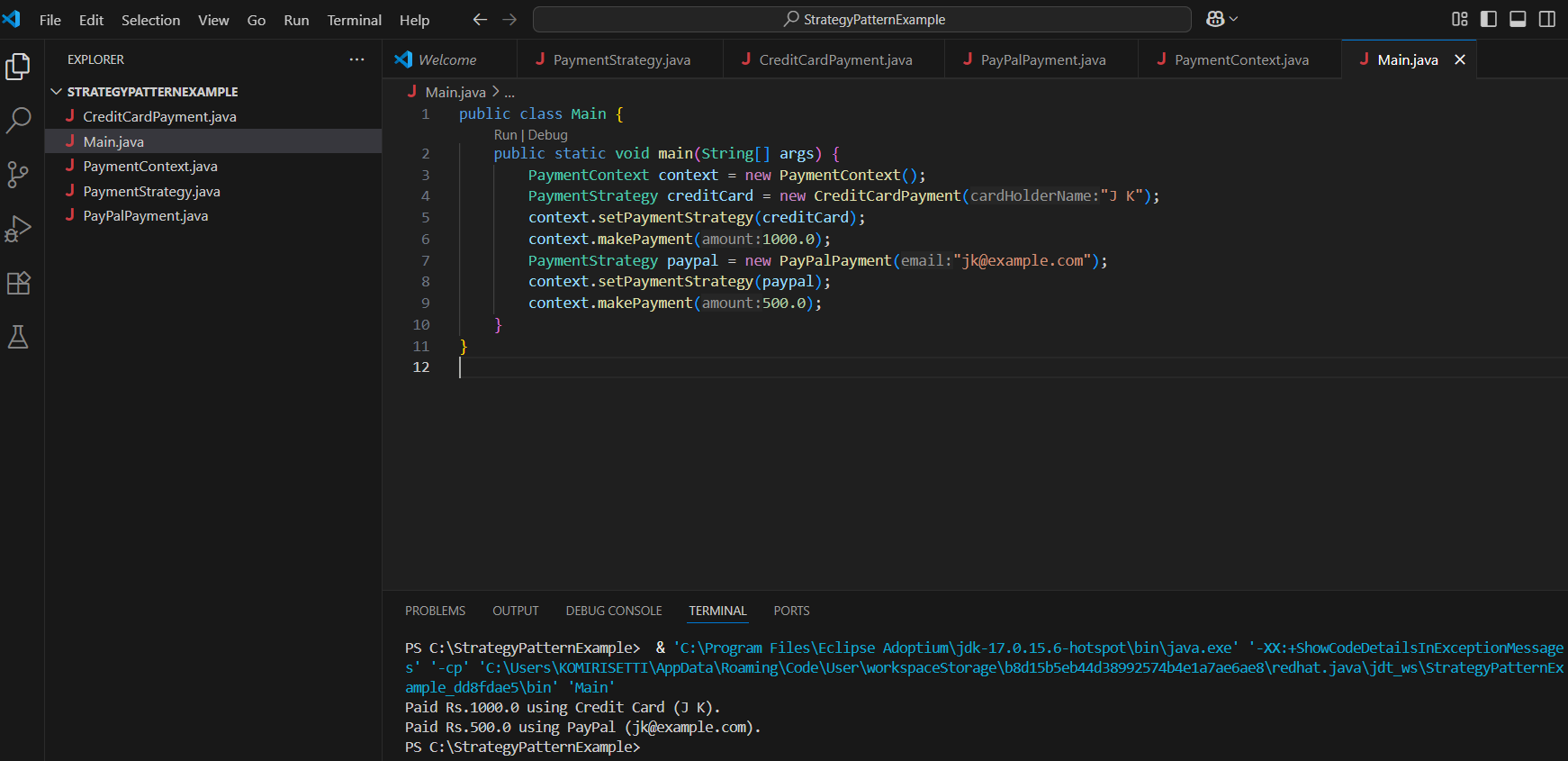
        context.setPaymentStrategy(paypal);

        context.makePayment(500.0);

    }

}

**Output:**

****

**Exercise 9: Implementing the Command Pattern**

**Java Project Name : CommandPatternExample**

**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() {

        if (command != null) {

            command.execute();

        } else {

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

        }

    }

}

**Main.java**

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();

        remote.setCommand(lightOn);

        remote.pressButton();

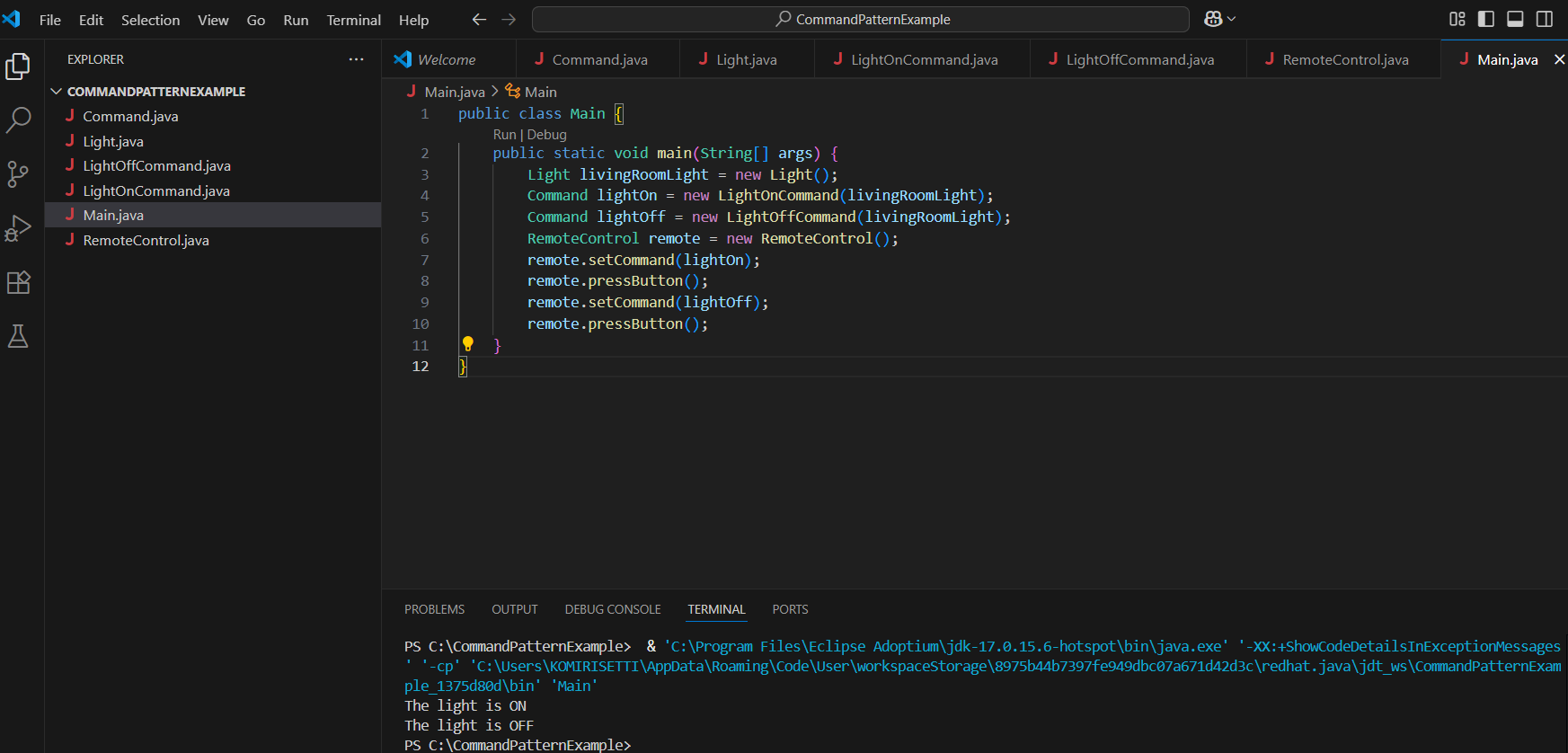
        remote.setCommand(lightOff);

        remote.pressButton();

    }

}

**Output:**

****

**Exercise 10: Implementing the MVC Pattern**

**Java Project Name : MVCPatternExample**

**Student.java**

public class Student {

    private String name;

    private int id;

    private String grade;

    public Student(String name, int 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 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;

    }

}

**StudentView.java**

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);

    }

}

**StudentController.java**

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());

    }

}

**Main.java**

public class Main {

    public static void main(String[] args) {

        Student student = new Student("Aarav", 101, "A");

        StudentView view = new StudentView();

        StudentController controller = new StudentController(student, view);

        controller.updateView();

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

        controller.setStudentName("Anika");

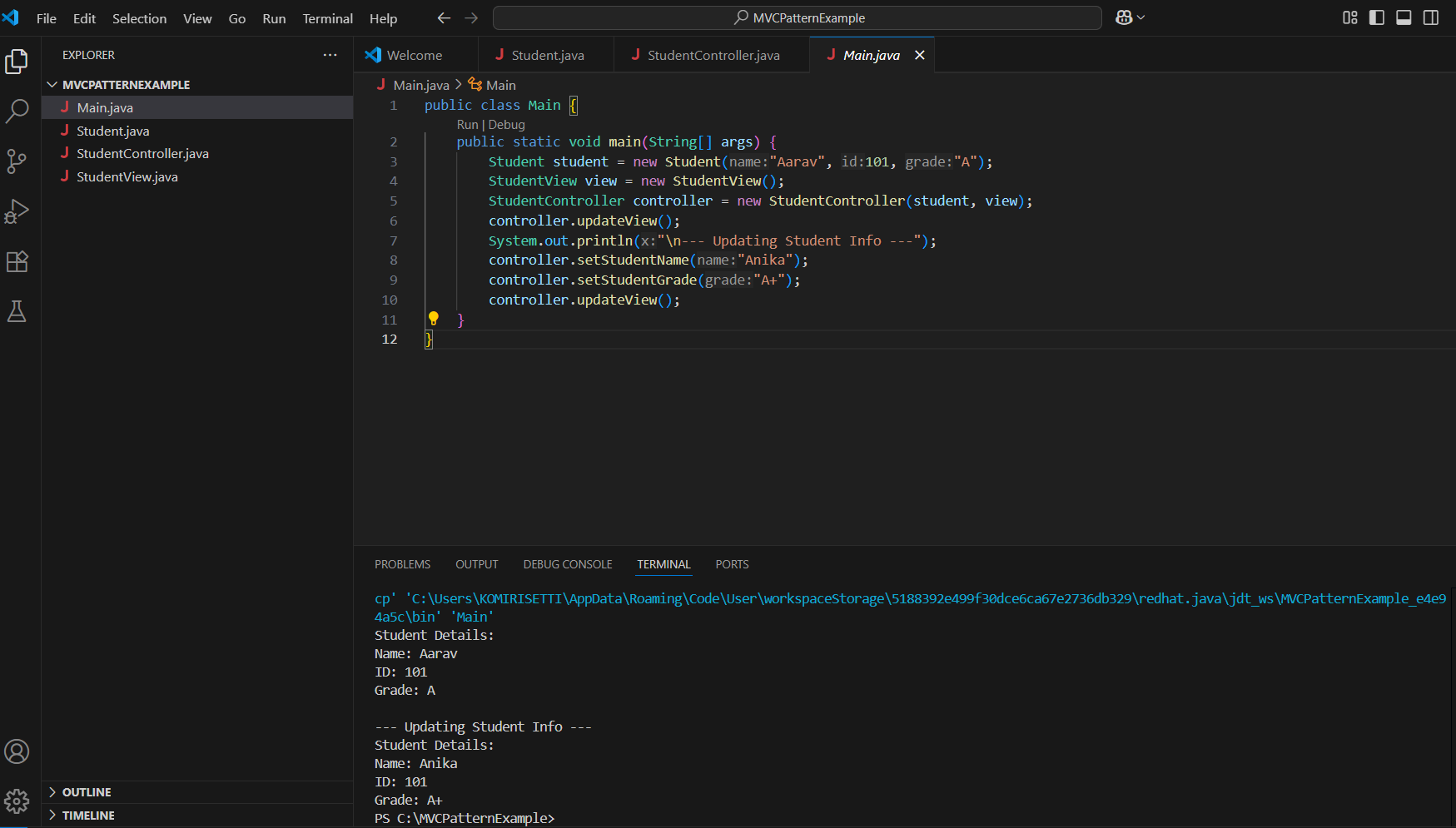
        controller.setStudentGrade("A+");

        controller.updateView();

    }

}

**Output:**

****

**Exercise 11: Implementing Dependency Injection**

**Java Project Name : DependencyInjectionExample**

**CustomerRepository.java**

public interface CustomerRepository {

    String findCustomerById(int id);

}

**CustomerRepositoryImpl.java**

public class CustomerRepositoryImpl implements CustomerRepository {

    @Override

    public String findCustomerById(int id) {

        return "Customer ID: " + id + ", Name: Anjali Verma";

    }

}

**CustomerService.java**

public class CustomerService {

    private final CustomerRepository customerRepository;

    public CustomerService(CustomerRepository customerRepository) {

        this.customerRepository = customerRepository;

    }

    public void showCustomerDetails(int id) {

        String customer = customerRepository.findCustomerById(id);

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

    }

}

**Main.java**

public class Main {

    public static void main(String[] args) {

        CustomerRepository repository = new CustomerRepositoryImpl();

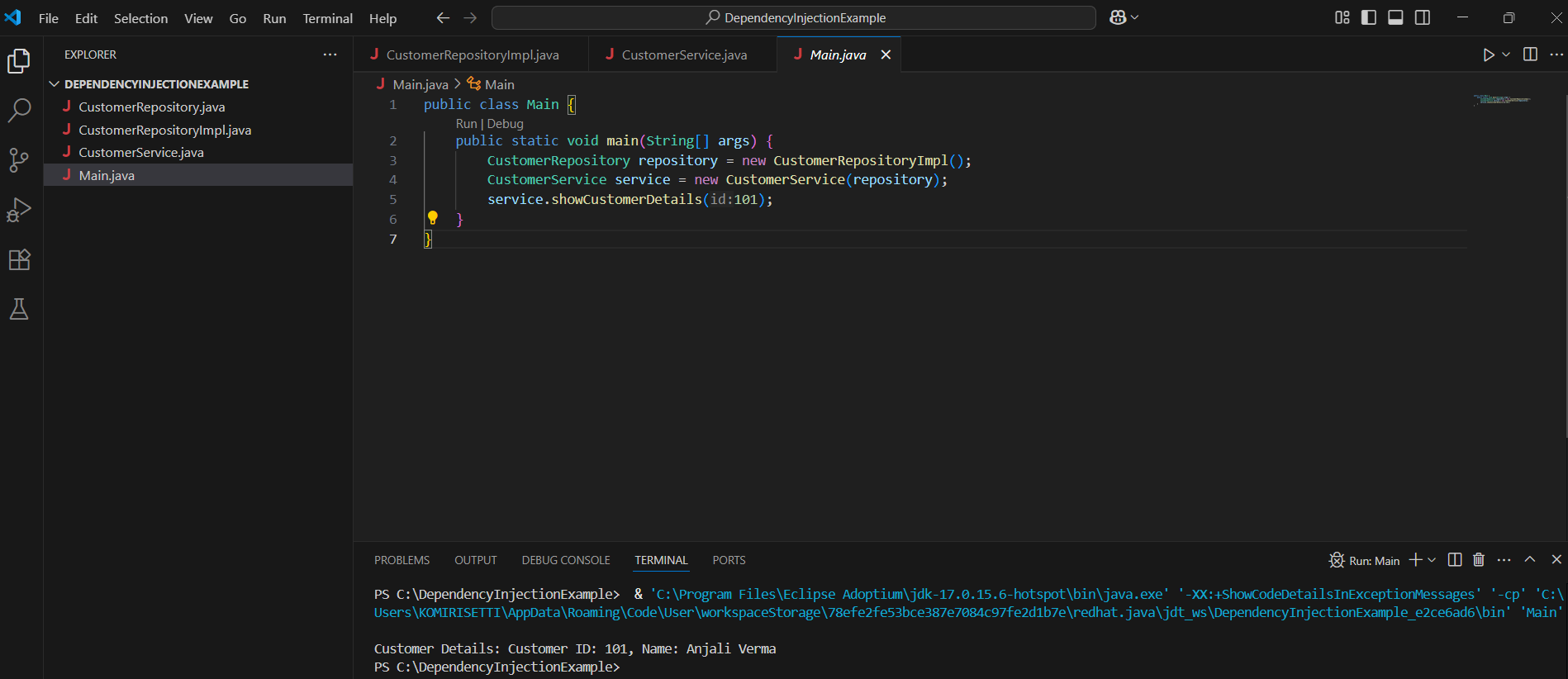
        CustomerService service = new CustomerService(repository);

        service.showCustomerDetails(101);

    }

}

**Output:**



**DATA STRUCTURES AND ALGORITHMS**

**Exercise 2: E-commerce Platform Search Function**

**Java Project Name : E-commerce Platform Search Function**

**Step 1:**

Big O Notation is a mathematical notation used to describe the efficiency of an algorithm in terms of time and space complexity. It gives us an upper bound on the running time as the input size grows.

Time Complexities for Search:

| Algorithm | Best Case | Average Case | Worst Case |
| --- | --- | --- | --- |
| Linear Search | O(1) | O(n) | O(n) |
| Binary Search | O(1) | O(log n) | O(log n) |

* Best case: When the item is found in the first comparison.
* Average case: When the item is found somewhere in the middle.
* Worst case: When the item is not found or is at the last position (linear) / the search space is reduced completely (binary).

**Step 2:**

**Product.java**

public class Product {

    int productId;

    String productName;

    String category;

    public Product(int productId, String productName, String category) {

        this.productId = productId;

        this.productName = productName;

        this.category = category;

    }

    public String toString() {

        return productId + " | " + productName + " | " + category;

    }

}

**Step 3:**

**SearchEngine.java**

public class SearchEngine {

    public static int linearSearch(Product[] products, String name) {

        for (int i = 0; i < products.length; i++) {

            if (products[i].productName.equalsIgnoreCase(name)) {

                return i;

            }

        }

        return -1;

    }

    public static int binarySearch(Product[] products, String name) {

        int low = 0, high = products.length - 1;

        while (low <= high) {

            int mid = (low + high) / 2;

            int cmp = products[mid].productName.compareToIgnoreCase(name);

            if (cmp == 0) return mid;

            else if (cmp < 0) low = mid + 1;

            else high = mid - 1;

        }

        return -1;

    }

}

**Main.java**

import java.util.Arrays;

import java.util.Comparator;

public class Main {

    public static void main(String[] args) {

        Product[] products = {

            new Product(201, "Laptop", "Electronics"),

            new Product(202, "Chair", "Furniture"),

            new Product(203, "Book", "Education")

        };

        Arrays.sort(products, Comparator.comparing(p -> p.productName));

        int index1 = SearchEngine.linearSearch(products, "Chair");

        int index2 = SearchEngine.binarySearch(products, "Chair");

        System.out.println("Linear Search Index: " + index1);

        System.out.println("Binary Search Index: " + index2);

    }

}

**Step 4:**

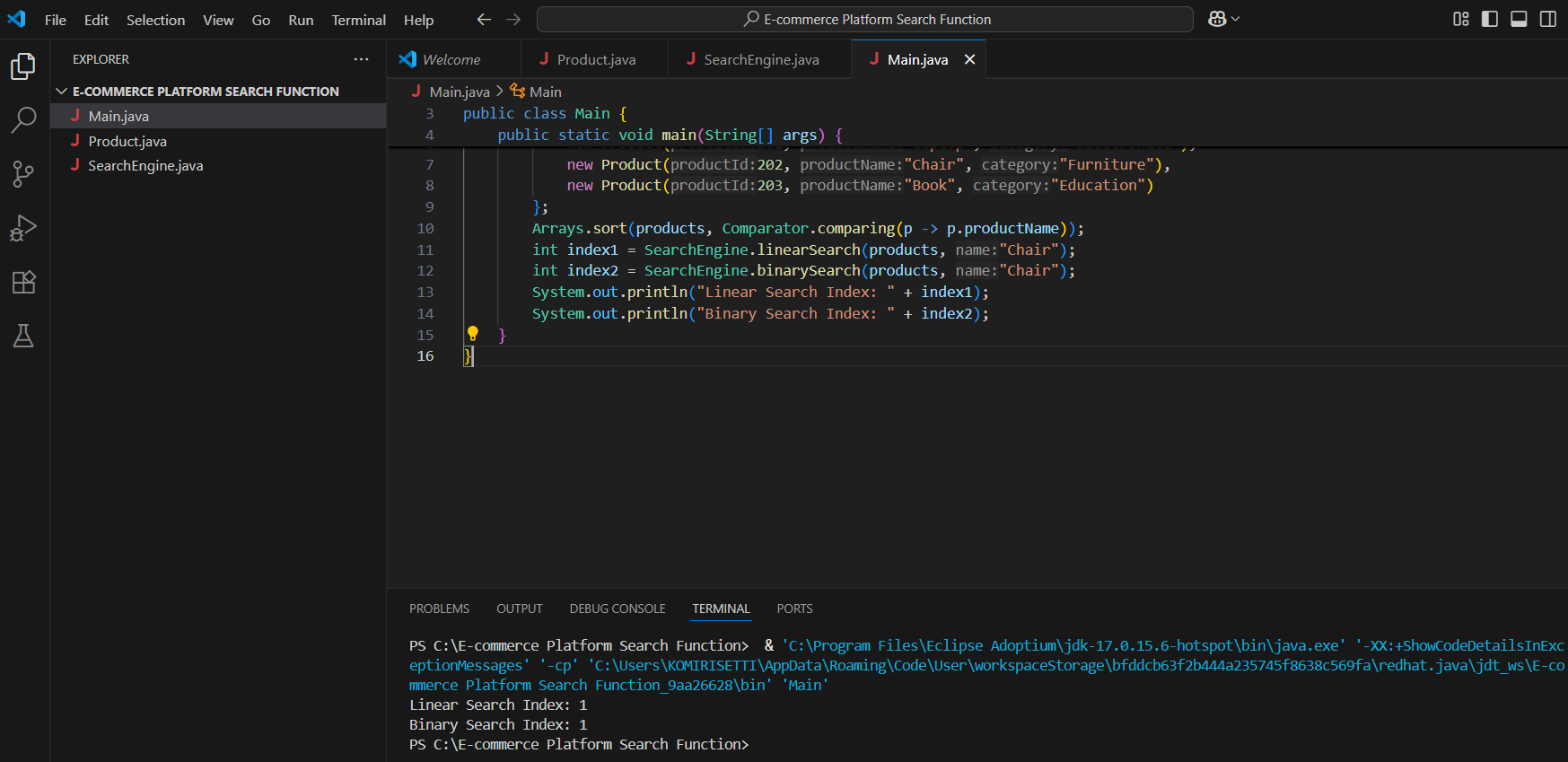
**Time Complexity Comparison:**

| Algorithm | Time Complexity |
| --- | --- |
| Linear Search | O(n) |
| Binary Search | O(log n), requires sorted data |

Suitability:

* Linear Search is simple and works on unsorted data, but is slow for large datasets.
* Binary Search is much faster, but the data must be sorted.
* For an e-commerce platform with large datasets, Binary Search is more suitable because it performs much faster due to logarithmic complexity.

**Output:**

****

**Exercise 7: Financial Forecasting**

**Java Project Name : Financial Forecasting**

**Step 1:**

Recursion is a programming technique where a function calls itself to solve smaller instances of a problem until it reaches a base condition. It is particularly useful when a problem can be broken down into similar subproblems.

In the context of financial forecasting, recursion simplifies the process of applying the compound interest formula repeatedly over a number of years. Instead of using loops, recursion allows us to express the logic more naturally and elegantly.

For example, to forecast the future value with a constant growth rate, we can define:

FutureValue(years) = FutureValue(years - 1) \* (1 + rate)

This approach continues until the number of years reaches zero, at which point the present value is returned. Thus, recursion offers a clean and structured way to calculate the result over multiple periods.

**Step 2:**

* I created a Java class named FinancialForecast.
* Inside this class, I defined a recursive method called forecast() that takes three parameters:
* presentValue (initial investment or value),
* rate (growth rate per year),
* years (number of years to forecast).

**Step 3:**

**FinancialForecast.java**

public class FinancialForecast {

    public static double forecast(double presentValue, double rate, int years) {

        if (years == 0) return presentValue;

        return forecast(presentValue \* (1 + rate), rate, years - 1);

    }

    public static void main(String[] args) {

        double presentValue = 10000;

        double rate = 0.05;

        int years = 5;

        double futureValue = forecast(presentValue, rate, years);

        System.out.println("Future Value (Recursive): " + futureValue);

    }

}

**Step 4:**

**Time Complexity of the Recursive Algorithm:**

* The recursive method calls itself once for every year, decreasing the years by 1 each time until it reaches 0.
* Therefore, the **time complexity is O(n)**, where n is the number of years.

**Space Complexity:**

* Since each recursive call is added to the call stack, the **space complexity is also O(n)**.

**Issues with Recursion:**

* For **large values of years**, recursion may lead to a **stack overflow** due to deep call stacks.
* Each call holds memory until the base case is reached, which increases **memory usage**.

**Optimization Techniques:**

To avoid excessive computation and memory usage, we can optimize the recursive solution by:

1. **Using Iteration Instead**:

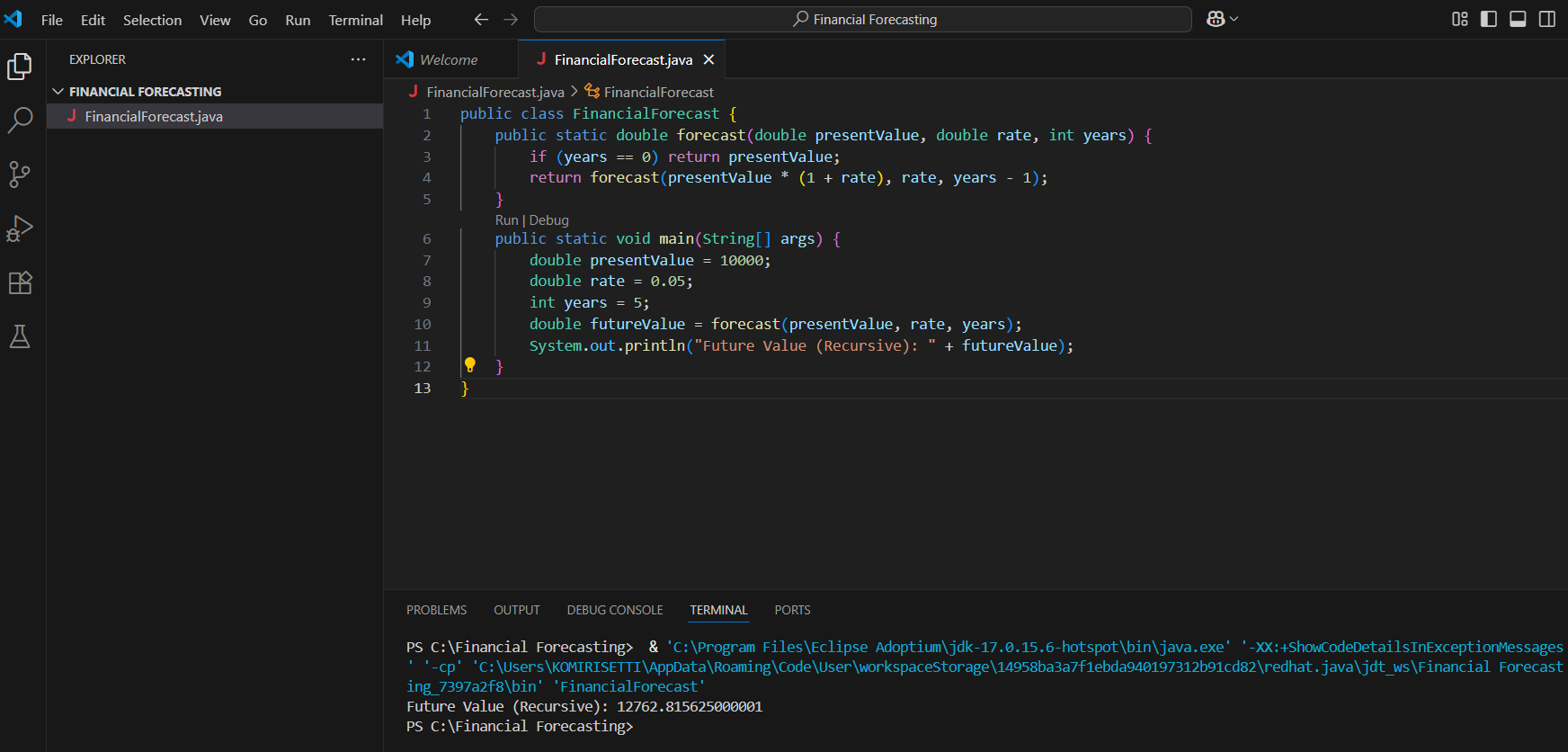
* A loop-based approach avoids call stack buildup.
* Time complexity remains **O(n)** but **space complexity reduces to O(1)**.

1. **Using the Compound Interest Formula** (if allowed):

Future Value=Present Value×(1+rate)years

* Time complexity: **O(1)**
* Space complexity: **O(1)**
* This is the most efficient method but replaces recursion.

**Output:**

****

**DATA STRUCTURES AND ALGORITHMS**

**(Other Hands-on exercises of data structures and algorithms)**

**Exercise 1: Inventory Management System**

**Java Project Name : Inventory Management System**

**Step 1:**

Why are Data Structures & Algorithms Essential?

* Warehouses may deal with thousands of products.
* Efficient data storage and retrieval help with:
  + Fast lookups (e.g., checking if a product exists)
  + Quick updates (e.g., stock quantity or price)
  + Deletions (e.g., discontinued products)
* Poor structure leads to slower operations, especially as inventory size grows.

Suitable Data Structures:

| Data Structure | Use Case | Pros | Cons |
| --- | --- | --- | --- |
| ArrayList | Useful if order matters or small datasets | Easy to use | Slow search/update (O(n)) |
| HashMap (productId as key) | Best for fast lookup/update/delete | O(1) average time | No ordering of elements |

* HashMap<Integer, Product> – stores products with fast access via productId. (Best Choice)

**Step 2:**

* Create a new Java project called InventoryManagementSystem.

Project Structure:

InventoryManagementSystem

* Product.java
* Inventory.java
* Main.java

**Step 3:**

**Product.java**

public class Product {

    int productId;

    String productName;

    int quantity;

    double price;

    public Product(int productId, String productName, int quantity, double price) {

        this.productId = productId;

        this.productName = productName;

        this.quantity = quantity;

        this.price = price;

    }

    public String toString() {

        return productId + " | " + productName + " | Qty: " + quantity + " | Price: " + price;

    }

}

**InventoryManager.java**

import java.util.HashMap;

public class InventoryManager {

    private HashMap<Integer, Product> inventory = new HashMap<>();

    public void addProduct(Product p) {

        inventory.put(p.productId, p);

    }

    public void updateProduct(int productId, int newQty, double newPrice) {

        if (inventory.containsKey(productId)) {

            Product p = inventory.get(productId);

            p.quantity = newQty;

            p.price = newPrice;

        }

    }

    public void deleteProduct(int productId) {

        inventory.remove(productId);

    }

    public void displayInventory() {

        for (Product p : inventory.values()) {

            System.out.println(p);

        }

    }

}

**Main.java**

public class Main {

    public static void main(String[] args) {

        InventoryManager im = new InventoryManager();

        im.addProduct(new Product(101, "Mouse", 50, 299.99));

        im.addProduct(new Product(102, "Keyboard", 30, 599.99));

        System.out.println("Inventory:");

        im.displayInventory();

        im.updateProduct(101, 60, 279.99);

        im.deleteProduct(102);

        System.out.println("Updated Inventory:");

        im.displayInventory();

    }

}

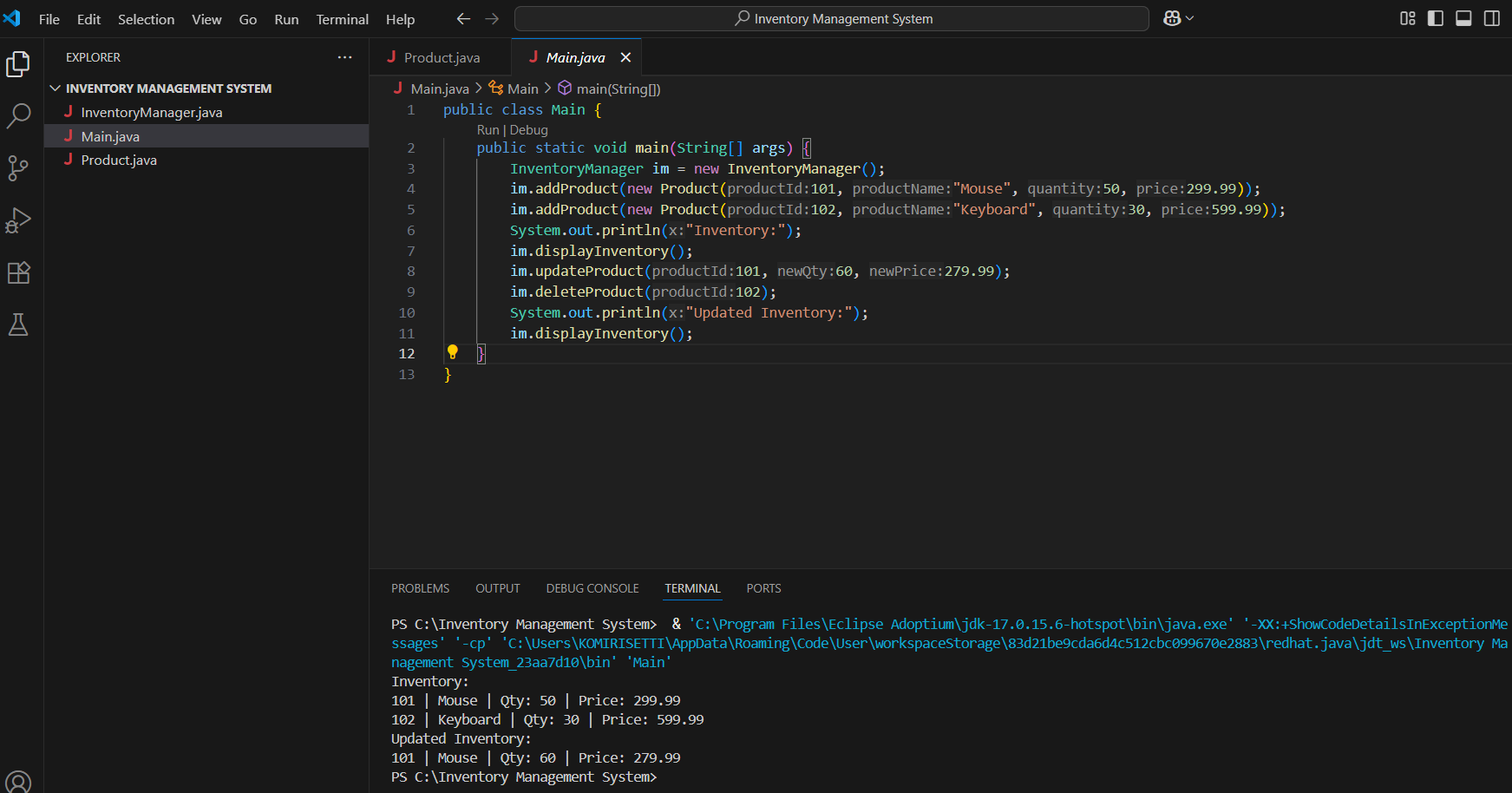
**Step 4:**

| **Operation** | **Time Complexity** | **Explanation** |
| --- | --- | --- |
| **addProduct()** | **O(1) avg** | **HashMap insertion is constant time** |
| **updateProduct()** | **O(1) avg** | **HashMap lookup & field update is constant** |
| **deleteProduct()** | **O(1) avg** | **HashMap removal is constant** |
| **displayAllProducts()** | **O(n)** | **Must iterate through all entries** |

**Optimizations:**

* **Use TreeMap instead of HashMap if you want sorted products by productId (O(log n) operations).**
* **Add search by product name using additional structures (e.g., another Map<String, List<Product>>).**
* **Use database like MySQL for persistent and scalable storage for real-world use.**

**Output:**

****

**Exercise 3: Sorting Customer Orders**

**Java Project Name : Sorting Customer Orders**

**Step 1:**

1. Bubble Sort

* Compares adjacent elements and swaps them if they are in the wrong order.
* Time Complexity:
  + - Best: O(n) (if already sorted)
    - Worst: O(n²)
    - Stable sort

2. Insertion Sort

* Builds the sorted list one item at a time.
* Time Complexity:
  + - Best: O(n)
    - Worst: O(n²)
    - Stable

3. Quick Sort

* Uses divide and conquer. Picks a pivot and partitions the array around it.
* Time Complexity:
  + - Best/Average: O(n log n)
    - Worst: O(n²) (rare, if pivot is poorly chosen)
    - Not stable

4. Merge Sort

* Recursively divides the array into halves, sorts them, and merges.
* Time Complexity:
  + - Always O(n log n)
    - Stable

**Step 2:**

**Order.java**

public class Order {

    int orderId;

    String customerName;

    double totalPrice;

    public Order(int orderId, String customerName, double totalPrice) {

        this.orderId = orderId;

        this.customerName = customerName;

        this.totalPrice = totalPrice;

    }

    public String toString() {

        return orderId + " | " + customerName + " | ₹" + totalPrice;

    }

}

**Step 3:**

**SortOrders.java**

public class SortOrders {

    public static void bubbleSort(Order[] orders) {

        for (int i = 0; i < orders.length - 1; i++) {

            for (int j = 0; j < orders.length - i - 1; j++) {

                if (orders[j].totalPrice > orders[j + 1].totalPrice) {

                    Order temp = orders[j];

                    orders[j] = orders[j + 1];

                    orders[j + 1] = temp;

                }

            }

        }

    }

    public static void quickSort(Order[] orders, int low, int high) {

        if (low < high) {

            int pi = partition(orders, low, high);

            quickSort(orders, low, pi - 1);

            quickSort(orders, pi + 1, high);

        }

    }

    private static int partition(Order[] orders, int low, int high) {

        double pivot = orders[high].totalPrice;

        int i = low - 1;

        for (int j = low; j < high; j++) {

            if (orders[j].totalPrice < pivot) {

                i++;

                Order temp = orders[i];

                orders[i] = orders[j];

                orders[j] = temp;

            }

        }

        Order temp = orders[i + 1];

        orders[i + 1] = orders[high];

        orders[high] = temp;

        return i + 1;

    }

}

**Main.java**

public class Main {

    public static void main(String[] args) {

        Order[] orders = {

            new Order(301, "Amit", 4500.00),

            new Order(302, "Divya", 950.00),

            new Order(303, "Rahul", 12500.00)

        };

        System.out.println("Before Sorting:");

        for (Order o : orders) System.out.println(o);

        SortOrders.quickSort(orders, 0, orders.length - 1);

        System.out.println("\nAfter Quick Sort:");

        for (Order o : orders) System.out.println(o);

    }

}

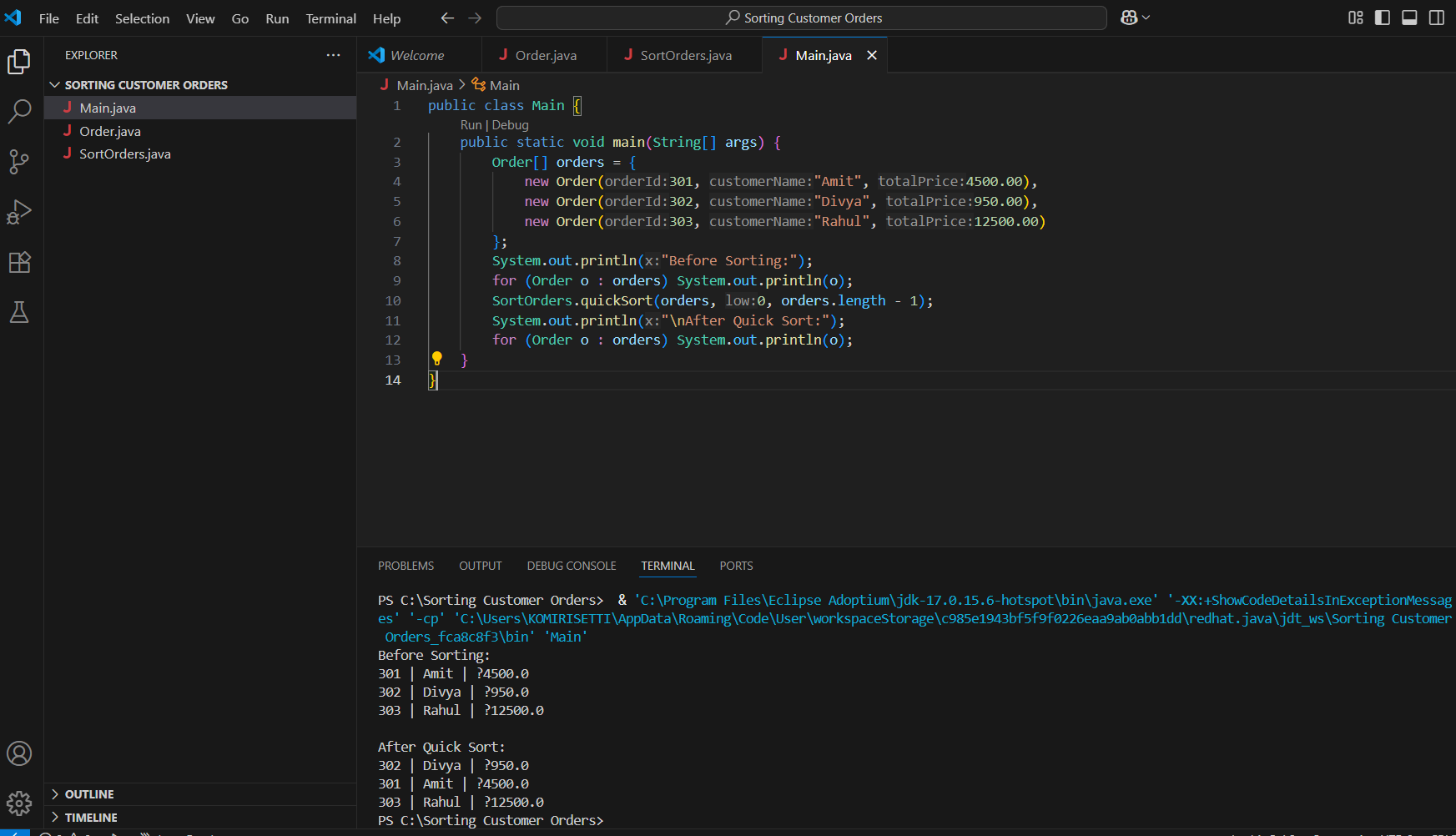
**Step 4:**

| Algorithm | Best Case | Average Case | Worst Case | Stable? | Suitable for Large Data? |
| --- | --- | --- | --- | --- | --- |
| Bubble Sort | O(n) | O(n²) | O(n²) | Yes | ❌ No |
| Quick Sort | O(n log n) | O(n log n) | O(n²) | No | ✅ Yes (with good pivot) |

Why is Quick Sort Preferred?

* Much faster for large datasets.
* Bubble Sort compares every element multiple times.
* Quick Sort reduces unnecessary comparisons using divide & conquer.

**Output:**

****

**Exercise 4: Employee Management System**

**Java Project Name : Employee Management System**

**Step 1:**

How Arrays Are Represented in Memory:

* Arrays are stored in contiguous memory locations.
* Each element can be accessed directly using its index.
* The index is calculated using the formula:  
  Address of element = Base address + (index × size of data type)

Advantages of Arrays:

* Fast access using index: O(1)
* Easy to implement and use for fixed-size collections.

**Step 2:**

**Employee.java**

public class Employee {

    int employeeId;

    String name;

    String position;

    double salary;

    public Employee(int employeeId, String name, String position, double salary) {

        this.employeeId = employeeId;

        this.name = name;

        this.position = position;

        this.salary = salary;

    }

    public String toString() {

        return employeeId + " | " + name + " | " + position + " | ₹" + salary;

    }

}

**Step 3:**

**EmployeeManager.java**

public class EmployeeManager {

    Employee[] employees = new Employee[100];

    int count = 0;

    public void addEmployee(Employee e) {

        employees[count++] = e;

    }

    public void searchEmployee(int id) {

        for (int i = 0; i < count; i++) {

            if (employees[i].employeeId == id) {

                System.out.println("Found: " + employees[i]);

                return;

            }

        }

        System.out.println("Employee not found.");

    }

    public void deleteEmployee(int id) {

        for (int i = 0; i < count; i++) {

            if (employees[i].employeeId == id) {

                for (int j = i; j < count - 1; j++) {

                    employees[j] = employees[j + 1];

                }

                count--;

                return;

            }

        }

    }

    public void listEmployees() {

        for (int i = 0; i < count; i++) {

            System.out.println(employees[i]);

        }

    }

}

**Main.java**

public class Main {

    public static void main(String[] args) {

        EmployeeManager em = new EmployeeManager();

        em.addEmployee(new Employee(401, "Ravi", "Manager", 75000));

        em.addEmployee(new Employee(402, "Sneha", "Developer", 50000));

        System.out.println("All Employees:");

        em.listEmployees();

        em.searchEmployee(402);

        em.deleteEmployee(401);

        System.out.println("After Deletion:");

        em.listEmployees();

    }

}

**Step 4:**

| **Operation** | **Time Complexity** | **Explanation** |
| --- | --- | --- |
| Add | O(1) | Adds at the end (if space is available) |
| Search | O(n) | Linear search by ID |
| Traverse | O(n) | Goes through each employee |
| Delete | O(n) | Needs to shift elements |

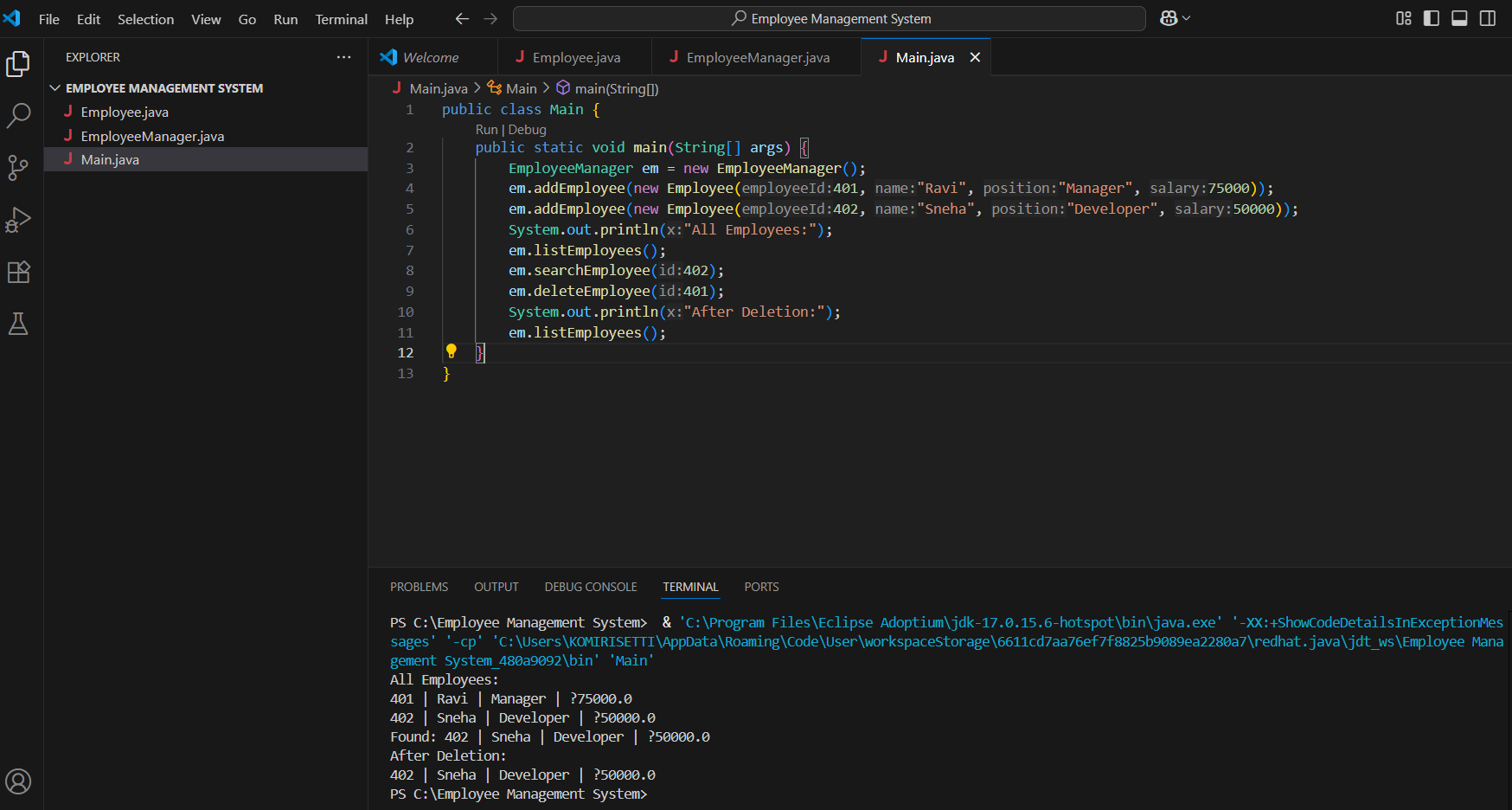
Limitations of Arrays:

* Fixed size: Can't grow dynamically.
* Insertion/deletion at middle requires shifting (inefficient).
* Wasted space if not fully used.

When to Use Arrays:

* When the number of records is known and fixed.
* When you need fast random access using index.

**Output:**

****

**Exercise 5: Task Management System**

**Java Project Name : Task Management System**

**Step 1:**

Types of Linked Lists:

* Singly Linked List: Nodes point only to the next node.
* Doubly Linked List: Nodes point to both next and previous nodes.

Singly Linked List is chosen for this simple task manager.

**Step 2 and 3:**

class Task {

    int taskId;

    String taskName;

    String status;

    Task next;

    public Task(int taskId, String taskName, String status) {

        this.taskId = taskId;

        this.taskName = taskName;

        this.status = status;

        this.next = null;

    }

    public String toString() {

        return taskId + " | " + taskName + " | " + status;

    }

}

class TaskManager {

    Task head = null;

    public void addTask(Task newTask) {

        if (head == null) {

            head = newTask;

        } else {

            Task temp = head;

            while (temp.next != null) {

                temp = temp.next;

            }

            temp.next = newTask;

        }

    }

    public Task searchTask(int id) {

        Task current = head;

        while (current != null) {

            if (current.taskId == id) return current;

            current = current.next;

        }

        return null;

    }

    public void traverseTasks() {

        Task current = head;

        while (current != null) {

            System.out.println(current);

            current = current.next;

        }

    }

    public void deleteTask(int id) {

        if (head == null) return;

        if (head.taskId == id) {

            head = head.next;

            return;

        }

        Task prev = head;

        Task current = head.next;

        while (current != null) {

            if (current.taskId == id) {

                prev.next = current.next;

                return;

            }

            prev = current;

            current = current.next;

        }

    }

    public static void main(String[] args) {

        TaskManager tm = new TaskManager();

        tm.addTask(new Task(1, "Design UI", "Pending"));

        tm.addTask(new Task(2, "Code Backend", "In Progress"));

        tm.traverseTasks();

        System.out.println("Searching Task 2: " + tm.searchTask(2));

        tm.deleteTask(1);

        System.out.println("After deletion:");

        tm.traverseTasks();

    }

}

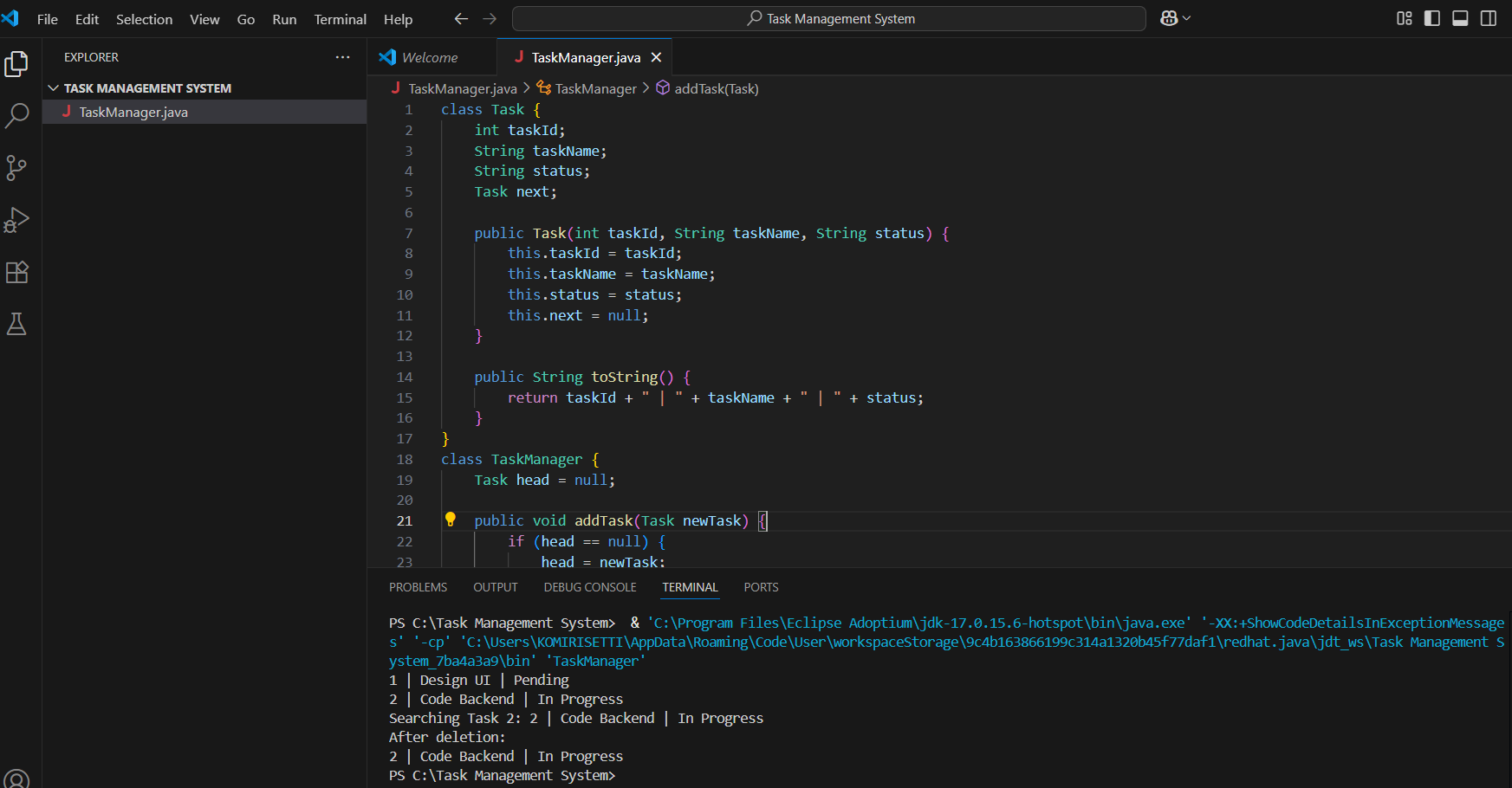
**Step 4:**

| **Operation** | **Time Complexity** |
| --- | --- |
| Add | O(n) |
| Search | O(n) |
| Traverse | O(n) |
| Delete | O(n) |

**Advantages of Linked Lists over Arrays:**

* Dynamic size (no need to predefine size)
* Easier insertion/deletion (no shifting like arrays)

**Output:**

****

**Exercise 6: Library Management System**

**Java Project Name : Library Management System**

**Step 1:**

* Linear Search: Check each item one by one.
* Binary Search: Search in a sorted array by dividing it repeatedly (requires sorted data).

**Step 2 and 3:**

import java.util.Arrays;

class Book {

    int bookId;

    String title;

    String author;

    public Book(int bookId, String title, String author) {

        this.bookId = bookId;

        this.title = title;

        this.author = author;

    }

    public String toString() {

        return bookId + " | " + title + " | " + author;

    }

}

public class LibrarySearch {

    public static Book linearSearch(Book[] books, String title) {

        for (Book b : books) {

            if (b.title.equalsIgnoreCase(title)) return b;

        }

        return null;

    }

    public static Book binarySearch(Book[] books, String title) {

        int low = 0, high = books.length - 1;

        while (low <= high) {

            int mid = (low + high) / 2;

            int cmp = books[mid].title.compareToIgnoreCase(title);

            if (cmp == 0) return books[mid];

            else if (cmp < 0) low = mid + 1;

            else high = mid - 1;

        }

        return null;

    }

    public static void main(String[] args) {

        Book[] books = {

            new Book(101, "C Programming", "Dennis Ritchie"),

            new Book(102, "Java Basics", "James Gosling"),

            new Book(103, "Python 101", "Guido van Rossum")

        };

        Arrays.sort(books, (a, b) -> a.title.compareToIgnoreCase(b.title));

        System.out.println("Linear Search:");

        System.out.println(linearSearch(books, "Java Basics"));

        System.out.println("Binary Search:");

        System.out.println(binarySearch(books, "Python 101"));

    }

}

**Step 4:**

| **Algorithm** | **Time Complexity** |
| --- | --- |
| Linear Search | O(n) |
| Binary Search | O(log n) |

**When to use:**

* **Linear Search**: For **unsorted** or small datasets.
* **Binary Search**: For **sorted** and large datasets.

**Output:**

