**Exercise 1: Inventory Management System**

* **Why Data Structures & Algorithms?**  
  Efficient handling of large inventories needs fast lookup, insertion, deletion, and update. Without optimized structures, operations could be too slow to scale.
* **Suitable Data Structures:**

ArrayList – Good for ordered storage, but search and delete are O(n).

HashMap – Best for fast access using keys (like productId); insertion, search, and deletion are O(1) on average.

**InventoryManagementSystem.java**

import java.util.HashMap;

import java.util.Map;

import java.util.Scanner;

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;

    }

    @Override

    public String toString() {

        return "ProductID: " + productId + ", Name: " + productName + ", Qty: " + quantity + ", Price: " + price;

    }

}

class Inventory {

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

    public void addProduct(Product product) {

        if (productMap.containsKey(product.productId)) {

            System.out.println("Product with ID " + product.productId + " already exists.");

        } else {

            productMap.put(product.productId, product);

            System.out.println("Product added.");

        }

    }

    public void updateProduct(int productId, int quantity, double price) {

        Product p = productMap.get(productId);

        if (p != null) {

            p.quantity = quantity;

            p.price = price;

            System.out.println("Product updated.");

        } else {

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

        }

    }

    public void deleteProduct(int productId) {

        if (productMap.remove(productId) != null) {

            System.out.println("Product deleted.");

        } else {

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

        }

    }

    public void displayInventory() {

        if (productMap.isEmpty()) {

            System.out.println("Inventory is empty.");

        } else {

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

                System.out.println(p);

            }

        }

    }

}

public class InventoryManagementSystem {

    public static void main(String[] args) {

        Inventory inventory = new Inventory();

        Scanner sc = new Scanner(System.in);

        int choice;

        do {

            System.out.println("\n--- Inventory Menu ---");

            System.out.println("1. Add Product");

            System.out.println("2. Update Product");

            System.out.println("3. Delete Product");

            System.out.println("4. Display Inventory");

            System.out.println("5. Exit");

            System.out.print("Enter your choice: ");

            choice = sc.nextInt();

            switch (choice) {

                case 1:

                    System.out.print("Enter Product ID: ");

                    int id = sc.nextInt();

                    sc.nextLine(); // consume newline

                    System.out.print("Enter Product Name: ");

                    String name = sc.nextLine();

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

                    int qty = sc.nextInt();

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

                    double price = sc.nextDouble();

                    inventory.addProduct(new Product(id, name, qty, price));

                    break;

                case 2:

                    System.out.print("Enter Product ID to update: ");

                    int uid = sc.nextInt();

                    System.out.print("Enter new Quantity: ");

                    int newQty = sc.nextInt();

                    System.out.print("Enter new Price: ");

                    double newPrice = sc.nextDouble();

                    inventory.updateProduct(uid, newQty, newPrice);

                    break;

                case 3:

                    System.out.print("Enter Product ID to delete: ");

                    int did = sc.nextInt();

                    inventory.deleteProduct(did);

                    break;

                case 4:

                    inventory.displayInventory();

                    break;

                case 5:

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

                    break;

                default:

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

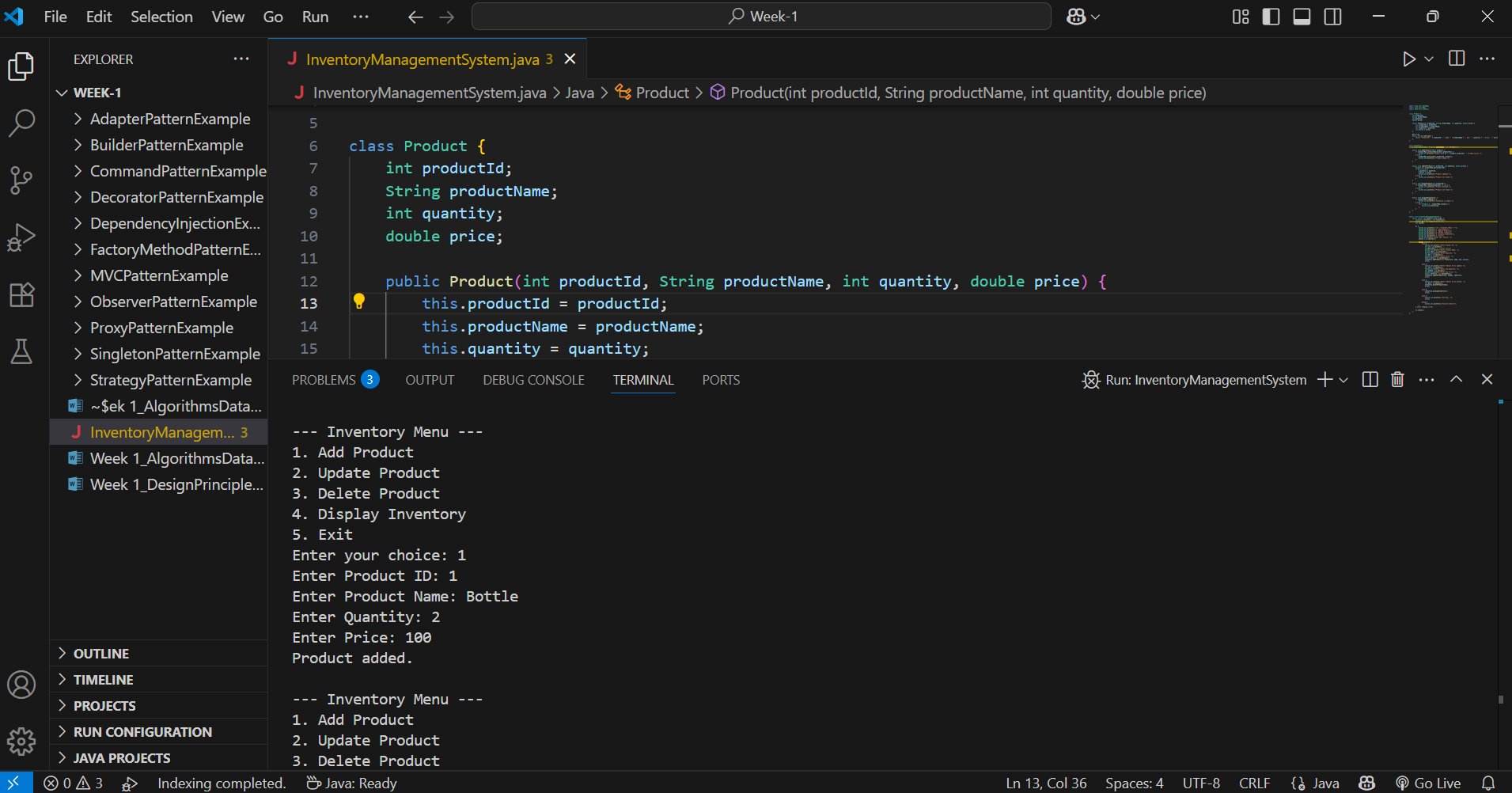
            }

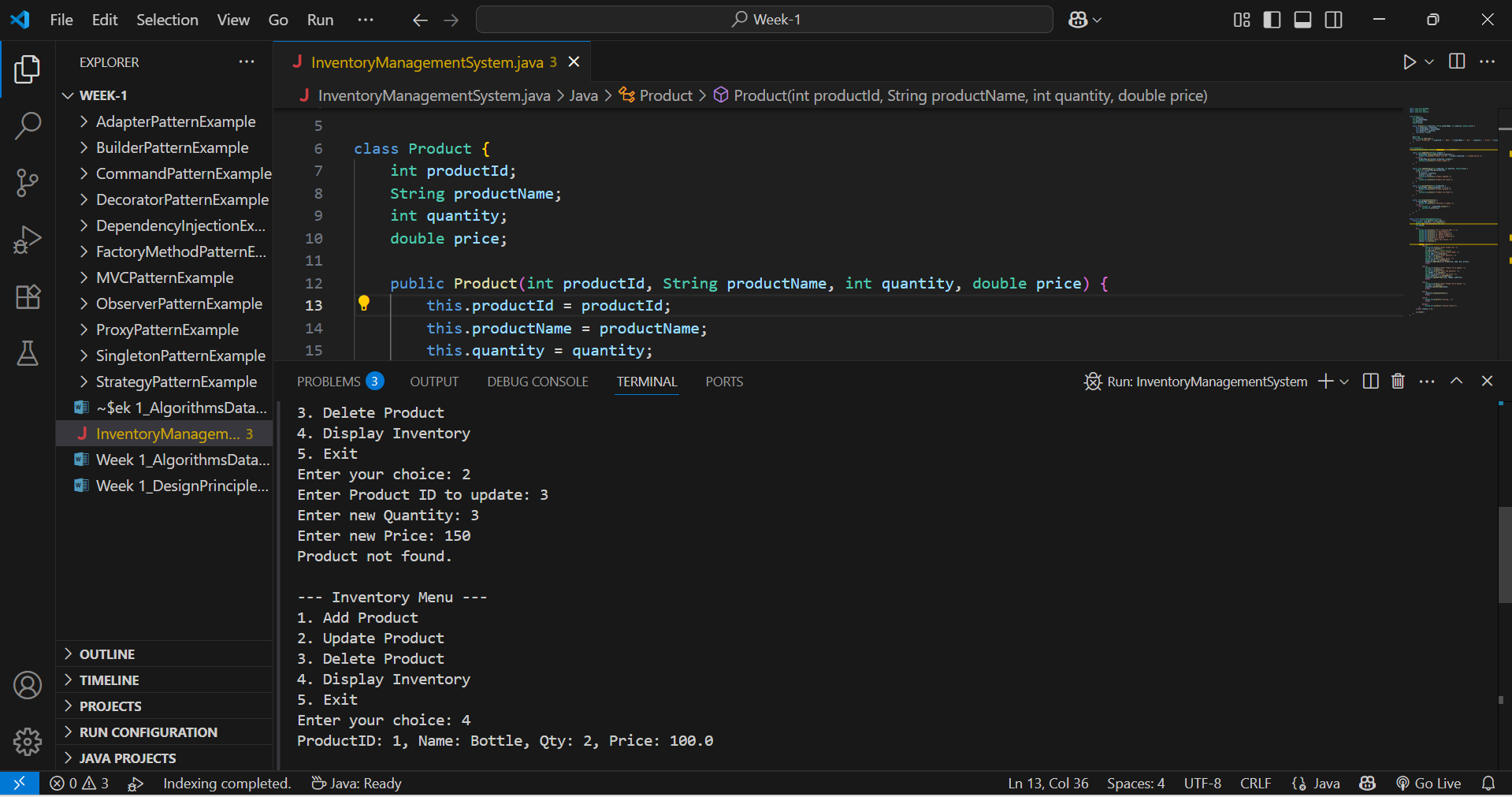
        } while (choice != 5);

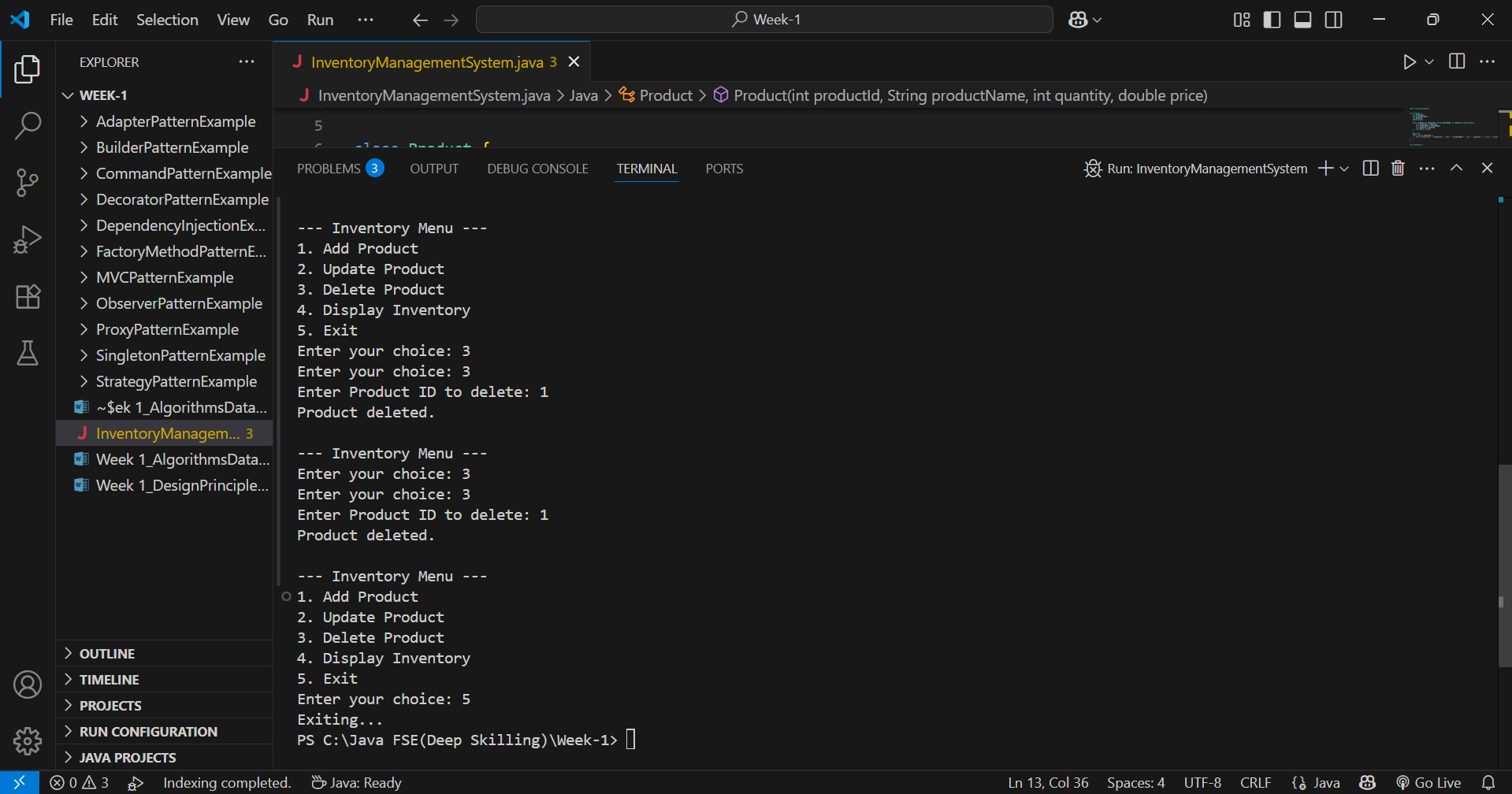
        sc.close();

    }

}







**Operations & Time Complexity**

1. **Add Product**
   * Time: **O(1)** average
   * Reason: Insert by key in HashMap
2. **Update Product**
   * Time: **O(1)** average
   * Reason: Access directly using productId
3. **Delete Product**
   * Time: **O(1)** average
   * Reason: Remove by key
4. **Display All Products**
   * Time: **O(n)**
   * Reason: Loop through all entries

**Optimization Tips:**

* Use ConcurrentHashMap for thread safety.
* Use database indexing for scalable inventory storage.
* Cache frequently accessed products if needed.

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

**Understanding Asymptotic Notation**

* **Big O Notation**: Describes the upper bound of an algorithm's running time as input size grows.
  + Example: O(n), O(log n), O(n²)
* Search Scenarios:
  + Best Case: Item found in first comparison (O(1) for linear and binary).
  + Average Case: Item found somewhere in the middle.
  + Worst Case: Item not found or last checked (O(n) for linear, O(log n) for binary).

**EcommerceSearchFunction.java**

import java.util.Arrays;

import java.util.Comparator;

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;

    }

    @Override

    public String toString() {

        return "ID: " + productId + ", Name: " + productName + ", Category: " + category;

    }

}

public class EcommerceSearchFunction {

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

        for (Product p : products) {

            if (p.productName.equalsIgnoreCase(name)) {

                return p;

            }

        }

        return null;

    }

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

        int low = 0;

        int high = products.length - 1;

        while (low <= high) {

            int mid = (low + high) / 2;

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

            if (cmp == 0) {

                return products[mid];

            } else if (cmp < 0) {

                low = mid + 1;

            } else {

                high = mid - 1;

            }

        }

        return null;

    }

    public static void main(String[] args) {

        Product[] products = {

            new Product(101, "Laptop", "Electronics"),

            new Product(102, "Shampoo", "Personal Care"),

            new Product(103, "Mobile", "Electronics"),

            new Product(104, "Notebook", "Stationery"),

            new Product(105, "T-Shirt", "Clothing")

        };

        String searchItem = "Mobile";

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

        Product result1 = linearSearch(products, searchItem);

        if (result1 != null)

            System.out.println("Found: " + result1);

        else

            System.out.println("Product not found!");

        System.out.println("\nBinary Search:");

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

        Product result2 = binarySearch(products, searchItem);

        if (result2 != null)

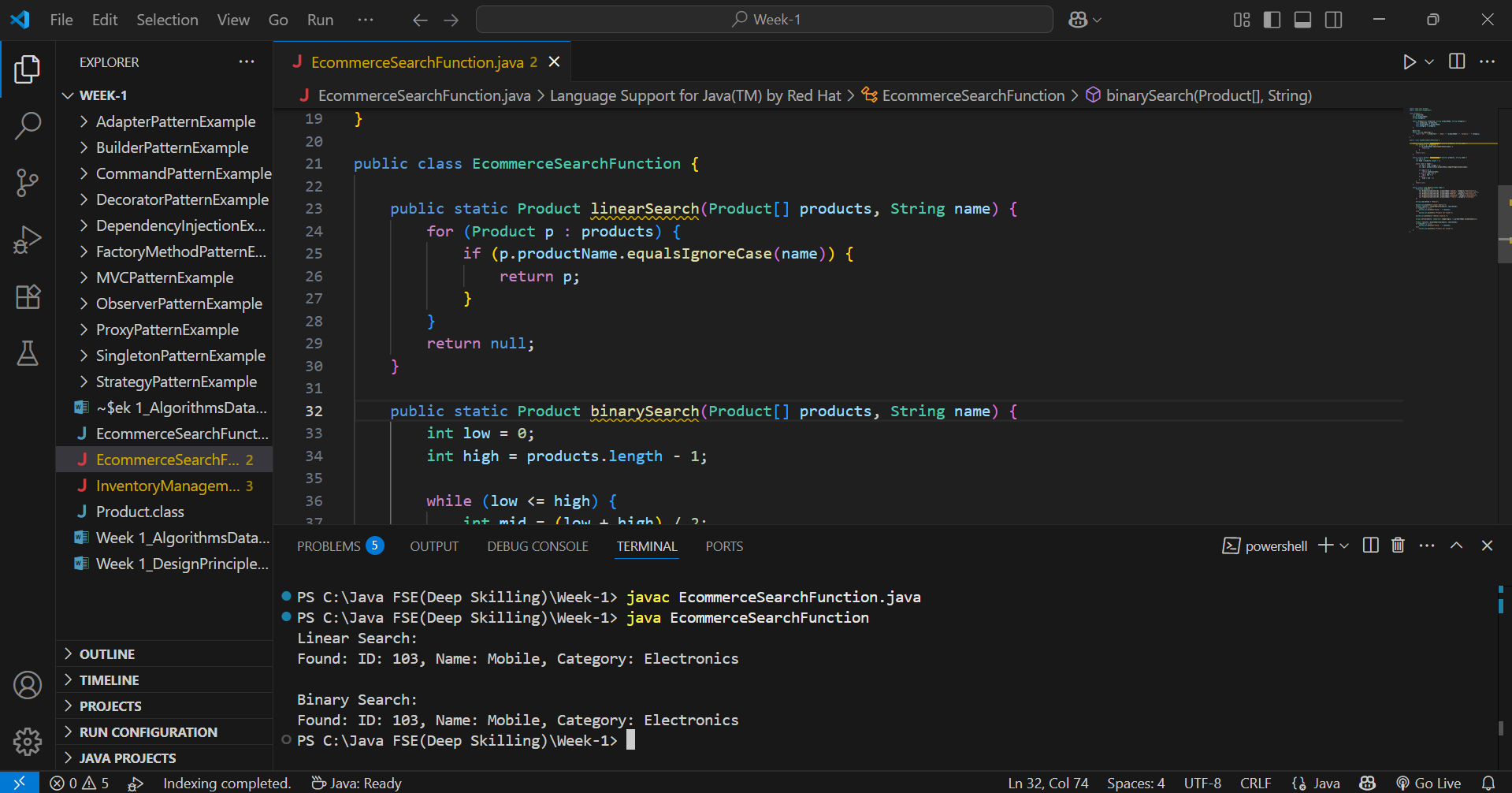
            System.out.println("Found: " + result2);

        else

            System.out.println("Product not found!");

    }

}



**Linear Search (Unsorted Array)**

* **Best:** O(1)
* **Worst:** O(n)
* **Use for:** Small or unsorted data

**Binary Search (Sorted Array)**

* **Best:** O(1)
* **Worst:** O(log n)
* **Use for:** Large, sorted data
* Binary Search is more suitable for large-scale e-commerce platforms where performance matters.
* Use binary search with sorted arrays or, even better, switch to hash maps or search trees for optimal results in real-world applications.

**Exercise 3: Sorting Customer Orders**

**Time Complexity Analysis**

**🔹 Bubble Sort**

1. **Best Case:**
   * **O(n)**
   * Occurs when the array is already sorted. Only one pass is needed (with no swaps).
2. **Average Case:**
   * **O(n²)**
   * Comparisons and swaps occur for roughly half the elements per pass.
3. **Worst Case:**
   * **O(n²)**
   * Happens when the array is sorted in reverse order. Maximum comparisons and swaps.

**🔹 Quick Sort**

1. **Best Case:**
   * **O(n log n)**
   * Occurs when the pivot divides the array into two equal halves each time.
2. **Average Case:**
   * **O(n log n)**
   * Expected in most random datasets. Efficient in practice.
3. **Worst Case:**
   * **O(n²)**
   * Happens when pivot is always the smallest or largest element (e.g., already sorted array without optimization).

**OrderSorting.java**

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;

    }

    @Override

    public String toString() {

        return "OrderID: " + orderId + ", Customer: " + customerName + ", Total Price: " + totalPrice;

    }

}

public class OrderSorting {

    public static void bubbleSort(Order[] orders) {

        int n = orders.length;

        for (int i = 0; i < n - 1; i++) {

            for (int j = 0; j < n - 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;

    }

    public static void printOrders(Order[] orders) {

        for (Order order : orders) {

            System.out.println(order);

        }

    }

    public static void main(String[] args) {

        Order[] orders = {

            new Order(101, "Alice", 2500.50),

            new Order(102, "Bob", 1500.00),

            new Order(103, "Charlie", 3200.75),

            new Order(104, "Diana", 800.00)

        };

        System.out.println("Original Orders:");

        printOrders(orders);

        System.out.println("\nOrders Sorted by Bubble Sort:");

        bubbleSort(orders);

        printOrders(orders);

        orders = new Order[] {

            new Order(101, "Alice", 2500.50),

            new Order(102, "Bob", 1500.00),

            new Order(103, "Charlie", 3200.75),

            new Order(104, "Diana", 800.00)

        };

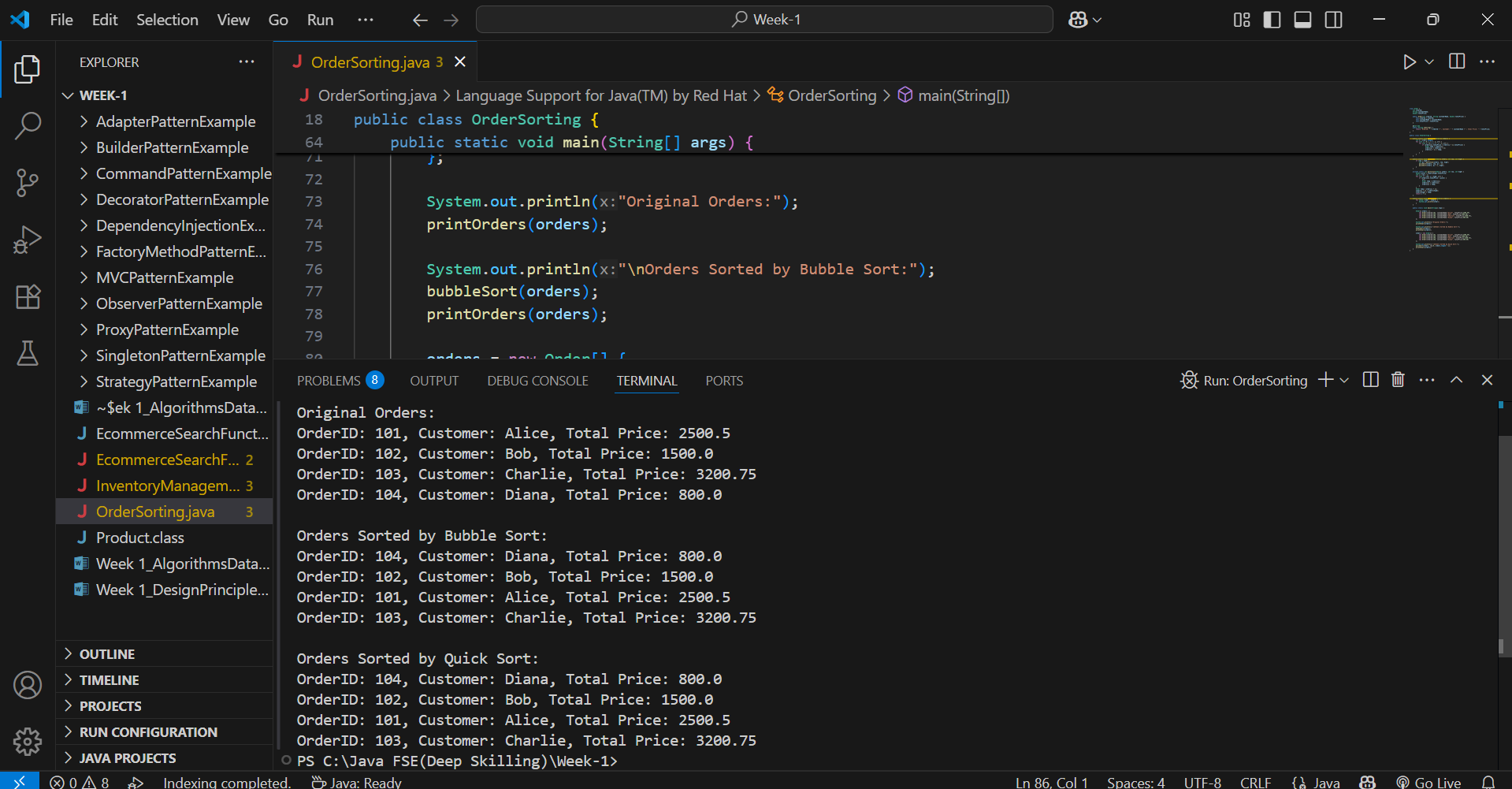
        System.out.println("\nOrders Sorted by Quick Sort:");

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

        printOrders(orders);

    }

}



**Time Complexity**

**🔹 Bubble Sort**

* Best: O(n)
* Average: O(n²)
* Worst: O(n²)

**🔹 Quick Sort**

* Best: O(n log n)
* Average: O(n log n)
* Worst: O(n²)

**Why Quick Sort is Preferred**

* **Much faster** for large datasets.
* **Efficient average case**: O(n log n).
* **Fewer swaps and comparisons** than Bubble Sort.
* **Used in real-world libraries** (e.g., Java’s Arrays.sort).

**Exercise 4: Employee Management System**

**Array Representation in Memory**

* Arrays are stored in **contiguous memory locations**.
* Each element can be accessed in **O(1)** time using the index.
* **Advantages:** Fast access, simple to implement.
* **Limitation:** Fixed size, costly insert/delete operations.

**EmployeeManagementSystem.java**

import java.util.Scanner;

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;

    }

    @Override

    public String toString() {

        return "ID: " + employeeId + ", Name: " + name + ", Position: " + position + ", Salary: " + salary;

    }

}

public class EmployeeManagementSystem {

    static final int MAX\_EMPLOYEES = 100;

    static Employee[] employees = new Employee[MAX\_EMPLOYEES];

    static int count = 0;

    public static void addEmployee(Employee emp) {

        if (count < MAX\_EMPLOYEES) {

            employees[count++] = emp;

            System.out.println("Employee added.");

        } else {

            System.out.println("Employee list is full.");

        }

    }

    public static 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 static void listEmployees() {

        if (count == 0) {

            System.out.println("No employees.");

        } else {

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

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

            }

        }

    }

    public static 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];

                }

                employees[--count] = null;

                System.out.println("Employee deleted.");

                return;

            }

        }

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

    }

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        int choice;

        do {

            System.out.println("\n--- Employee Management Menu ---");

            System.out.println("1. Add Employee");

            System.out.println("2. Search Employee by ID");

            System.out.println("3. List All Employees");

            System.out.println("4. Delete Employee by ID");

            System.out.println("5. Exit");

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

            choice = sc.nextInt();

            switch (choice) {

                case 1:

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

                    int id = sc.nextInt();

                    sc.nextLine();

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

                    String name = sc.nextLine();

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

                    String position = sc.nextLine();

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

                    double salary = sc.nextDouble();

                    addEmployee(new Employee(id, name, position, salary));

                    break;

                case 2:

                    System.out.print("Enter ID to search: ");

                    int searchId = sc.nextInt();

                    searchEmployee(searchId);

                    break;

                case 3:

                    listEmployees();

                    break;

                case 4:

                    System.out.print("Enter ID to delete: ");

                    int deleteId = sc.nextInt();

                    deleteEmployee(deleteId);

                    break;

                case 5:

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

                    break;

                default:

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

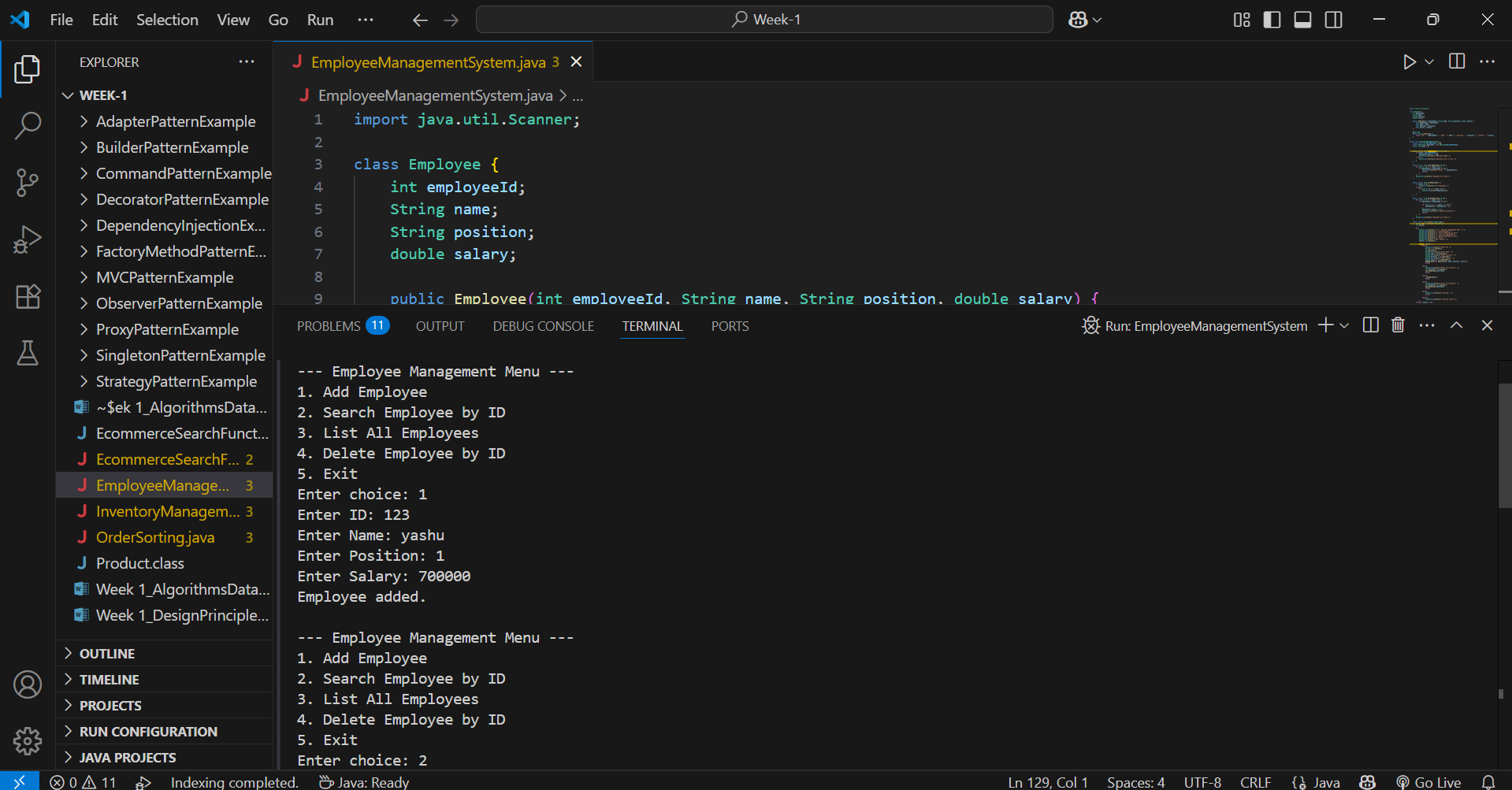
            }

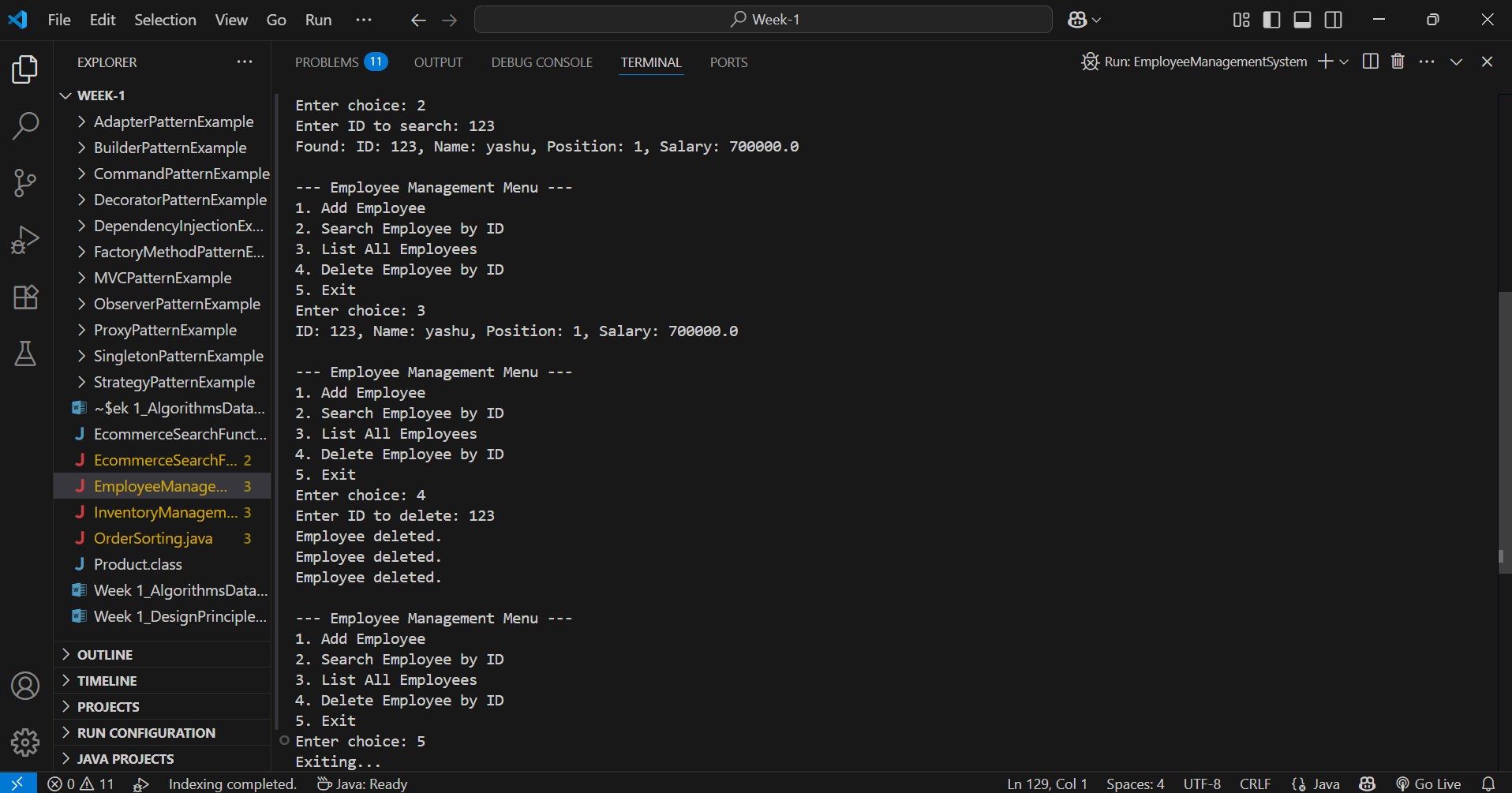
        } while (choice != 5);

        sc.close();

    }

}





**Time Complexity of Operations (Using Array)**

1. **Add Employee**
   * **Time Complexity:**
     + **O(1)** → if adding at the end
     + **O(n)** → if shifting elements is required (e.g., insert at specific position)
2. **Search Employee by ID**
   * **Time Complexity:** **O(n)**
   * Reason: Linear search required as array is unsorted.
3. **Traverse Employees**
   * **Time Complexity:** **O(n)**
   * Reason: Visit each element one by one.
4. **Delete Employee by ID**
   * **Time Complexity:** **O(n)**
   * Reason: Find index + shift elements after deletion.

**Limitations of Arrays**

1. **Fixed Size:**
   * Size must be declared upfront; can't grow dynamically.
2. **Inefficient Deletion/Insertion:**
   * Requires shifting elements — costly in large arrays.
3. **Linear Search Only (if unsorted):**
   * No direct indexing by ID; must loop through.
4. **No Built-in Flexibility:**
   * No easy resizing, sorting, or filtering like in ArrayList.

**When to Use Arrays**

* When the number of records is small and fixed.
* When you need fast access by index.
* When memory usage must be tightly controlled.

**Exercise 5: Task Management System**

**Types of Linked Lists**

1. **Singly Linked List**
   * Each node has: data + next pointer
   * Traversal: One direction (head to tail)
2. **Doubly Linked List**
   * Each node has: data + next + prev
   * Traversal: Both directions (forward & backward)
   * More flexible but uses more memory

**TaskManagementSystem.java**

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;

    }

    @Override

    public String toString() {

        return "ID: " + taskId + ", Name: " + taskName + ", Status: " + status;

    }

}

public class TaskManagementSystem {

    static Task head = null;

    public static void addTask(int id, String name, String status) {

        Task newTask = new Task(id, name, status);

        if (head == null) {

            head = newTask;

        } else {

            Task temp = head;

            while (temp.next != null) {

                temp = temp.next;

            }

            temp.next = newTask;

        }

        System.out.println("Task added.");

    }

    public static void searchTask(int id) {

        Task temp = head;

        while (temp != null) {

            if (temp.taskId == id) {

                System.out.println("Found: " + temp);

                return;

            }

            temp = temp.next;

        }

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

    }

    public static void listTasks() {

        Task temp = head;

        if (temp == null) {

            System.out.println("No tasks found.");

        } else {

            while (temp != null) {

                System.out.println(temp);

                temp = temp.next;

            }

        }

    }

    public static void deleteTask(int id) {

        if (head == null) {

            System.out.println("No tasks to delete.");

            return;

        }

        if (head.taskId == id) {

            head = head.next;

            System.out.println("Task deleted.");

            return;

        }

        Task temp = head;

        while (temp.next != null && temp.next.taskId != id) {

            temp = temp.next;

        }

        if (temp.next == null) {

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

        } else {

            temp.next = temp.next.next;

            System.out.println("Task deleted.");

        }

    }

    public static void main(String[] args) {

        addTask(1, "Design UI", "Pending");

        addTask(2, "Write Backend", "In Progress");

        addTask(3, "Testing", "Not Started");

        System.out.println("\nAll Tasks:");

        listTasks();

        System.out.println("\nSearch Task with ID 2:");

        searchTask(2);

        System.out.println("\nDelete Task with ID 1:");

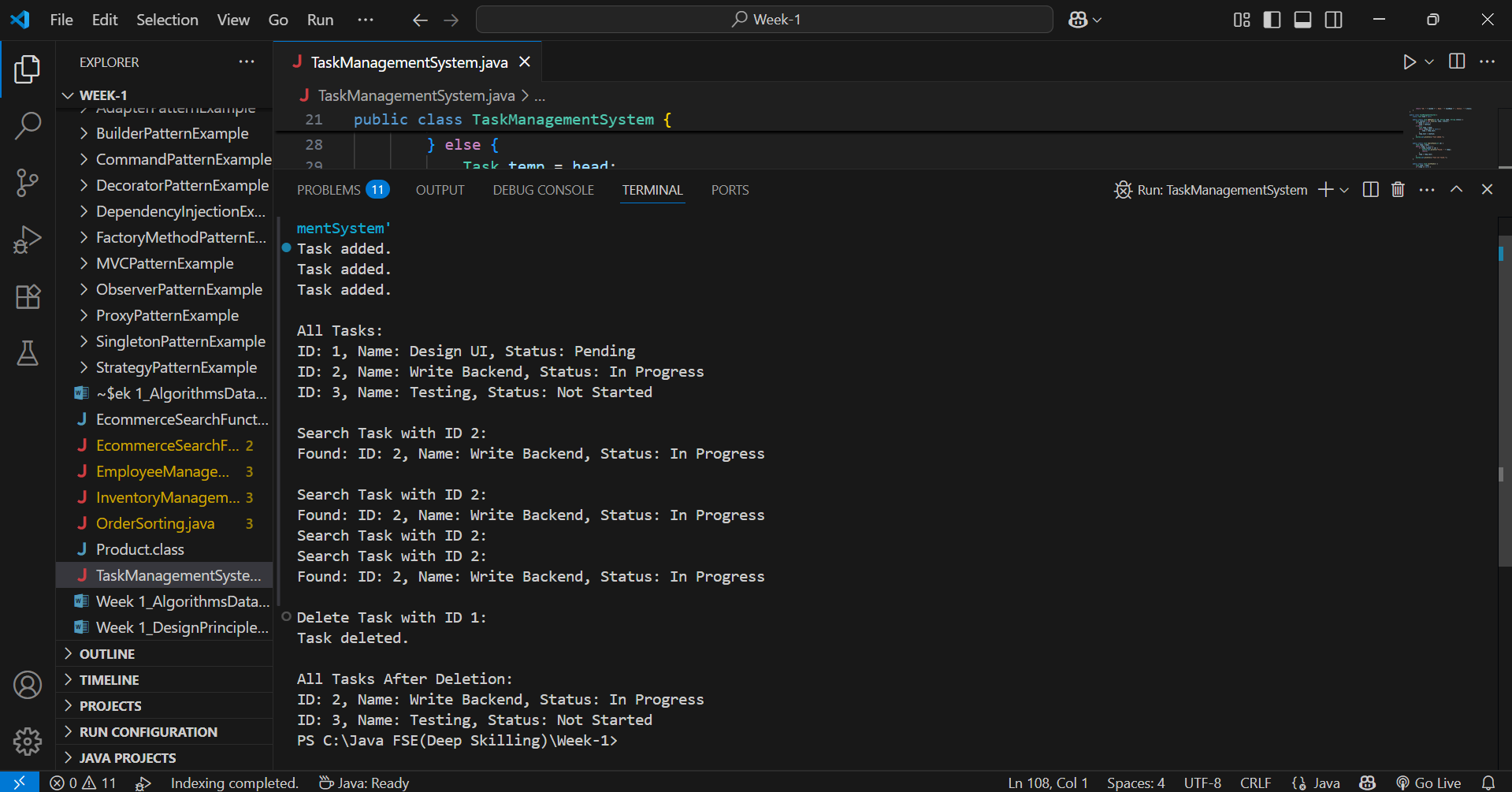
        deleteTask(1);

        System.out.println("\nAll Tasks After Deletion:");

        listTasks();

    }

}



**Time Complexity of Operations (Singly Linked List)**

1. **Add Task**
   * **Time Complexity:** O(n)
   * Reason: Must traverse to the end of the list to add a task.
2. **Search Task**
   * **Time Complexity:** O(n)
   * Reason: Need to check each node one by one.
3. **Traverse Tasks**
   * **Time Complexity:** O(n)
   * Reason: Visit each node to display tasks.
4. **Delete Task**
   * **Time Complexity:** O(n)
   * Reason: Must search for the node before deleting.

**Advantages of Linked Lists Over Arrays**

1. **Dynamic Size:**
   * No need to define size in advance; grows as needed.
2. **Efficient Insert/Delete:**
   * Insertion and deletion don’t require shifting elements like in arrays.
3. **Memory Utilization:**
   * Uses only as much memory as needed for the current elements.
4. **No Wasted Space:**
   * Unlike arrays, no pre-allocated unused space.

**Exercise 6: Library Management System**

**LibraryManagementSystem.java**

import java.util.Arrays;

import java.util.Comparator;

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;

    }

    @Override

    public String toString() {

        return "ID: " + bookId + ", Title: " + title + ", Author: " + author;

    }

}

public class LibraryManagementSystem {

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

        for (Book book : books) {

            if (book.title.equalsIgnoreCase(title)) {

                System.out.println("Found (Linear): " + book);

                return;

            }

        }

        System.out.println("Book not found (Linear).");

    }

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

        int left = 0, right = books.length - 1;

        while (left <= right) {

            int mid = (left + right) / 2;

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

            if (cmp == 0) {

                System.out.println("Found (Binary): " + books[mid]);

                return;

            } else if (cmp < 0) {

                left = mid + 1;

            } else {

                right = mid - 1;

            }

        }

        System.out.println("Book not found (Binary).");

    }

    public static void main(String[] args) {

        Book[] books = {

            new Book(101, "Java Basics", "James"),

            new Book(102, "Data Structures", "Mark"),

            new Book(103, "Algorithms", "Alice"),

            new Book(104, "Operating Systems", "Bob")

        };

        Arrays.sort(books, Comparator.comparing(b -> b.title.toLowerCase()));

        linearSearch(books, "Data Structures");

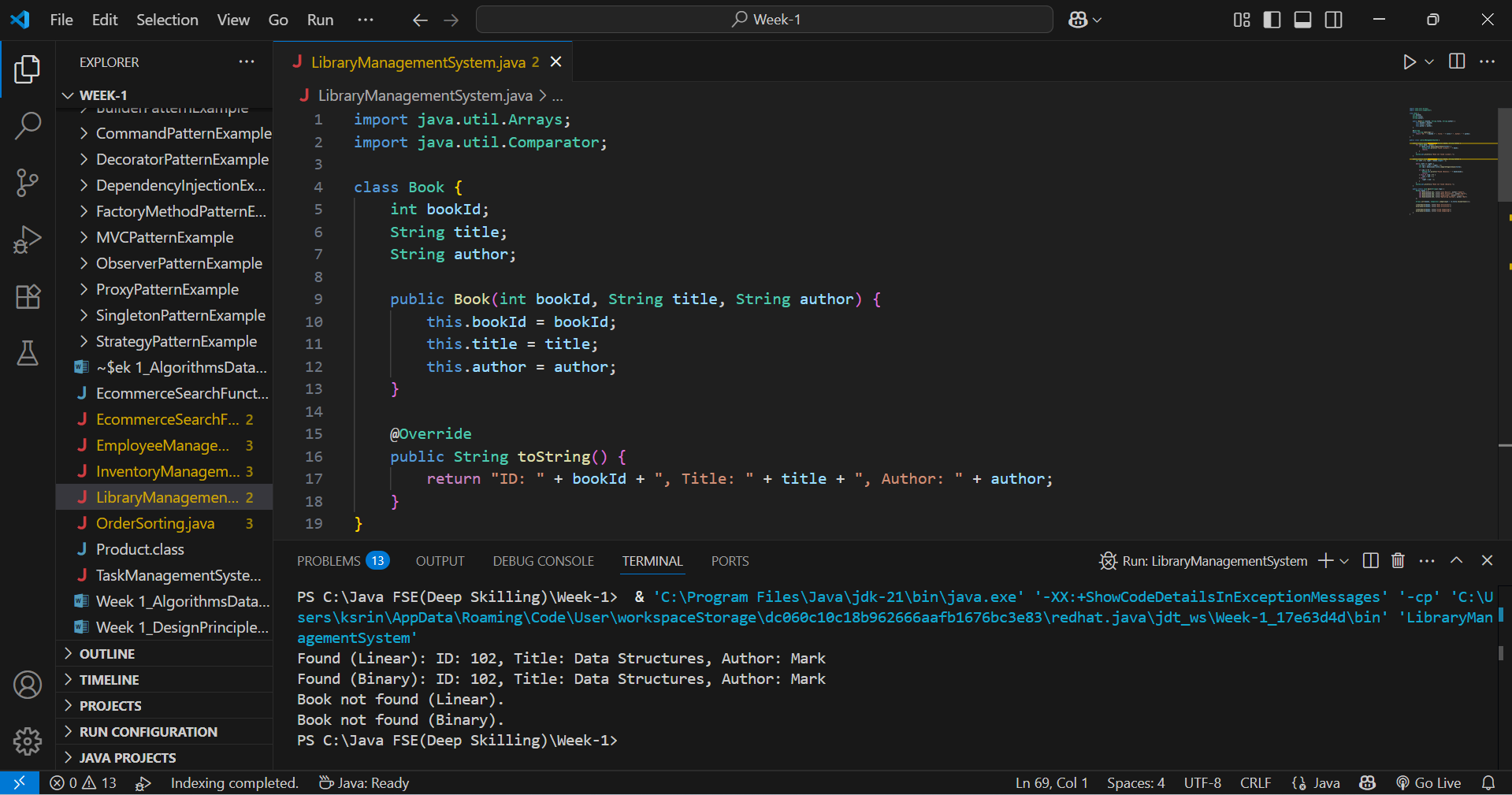
        binarySearch(books, "Data Structures");

        linearSearch(books, "Cloud Computing");

        binarySearch(books, "Cloud Computing");

    }

}

****

**Search Algorithms**

**🔹 Linear Search**

* Goes through each element one by one.
* **Works on**: Unsorted or sorted data.
* **Time Complexity:**
  + Best: O(1)
  + Average/Worst: O(n)

**🔹 Binary Search**

* Repeatedly divides the sorted array to search.
* **Works only on:** Sorted data.
* **Time Complexity:**
  + Best: O(1)
  + Average/Worst: O(log n)

**Time Complexity**

**Linear Search**

* Best: O(1) — if match is at start
* Worst: O(n) — if match is last or not found
* Use when data is small or unsorted

**Binary Search**

* Best: O(1)
* Worst: O(log n)
* Use when data is sorted and large

**When to Use Which Search**

1. **Use Linear Search:**
   * When the dataset is **small**
   * When the dataset is **unsorted**
   * When quick implementation is needed
2. **Use Binary Search:**
   * When the dataset is **large and sorted**
   * For **faster performance** on repeated lookups

**Exercise 7: Financial Forecasting**

* **Recursion** is a method where the solution to a problem depends on smaller instances of the same problem.
* A recursive method **calls itself** with a reduced input.
* It simplifies problems like:
* Factorial, Fibonacci, Tree traversals, Forecasting, etc.

**FinancialForecasting.java**

public class FinancialForecasting {

    public static double forecastValue(double presentValue, double growthRate, int years) {

        if (years == 0) {

            return presentValue;

        }

        return forecastValue(presentValue \* (1 + growthRate), growthRate, years - 1);

    }

    public static void main(String[] args) {

        double presentValue = 10000;

        double growthRate = 0.10;

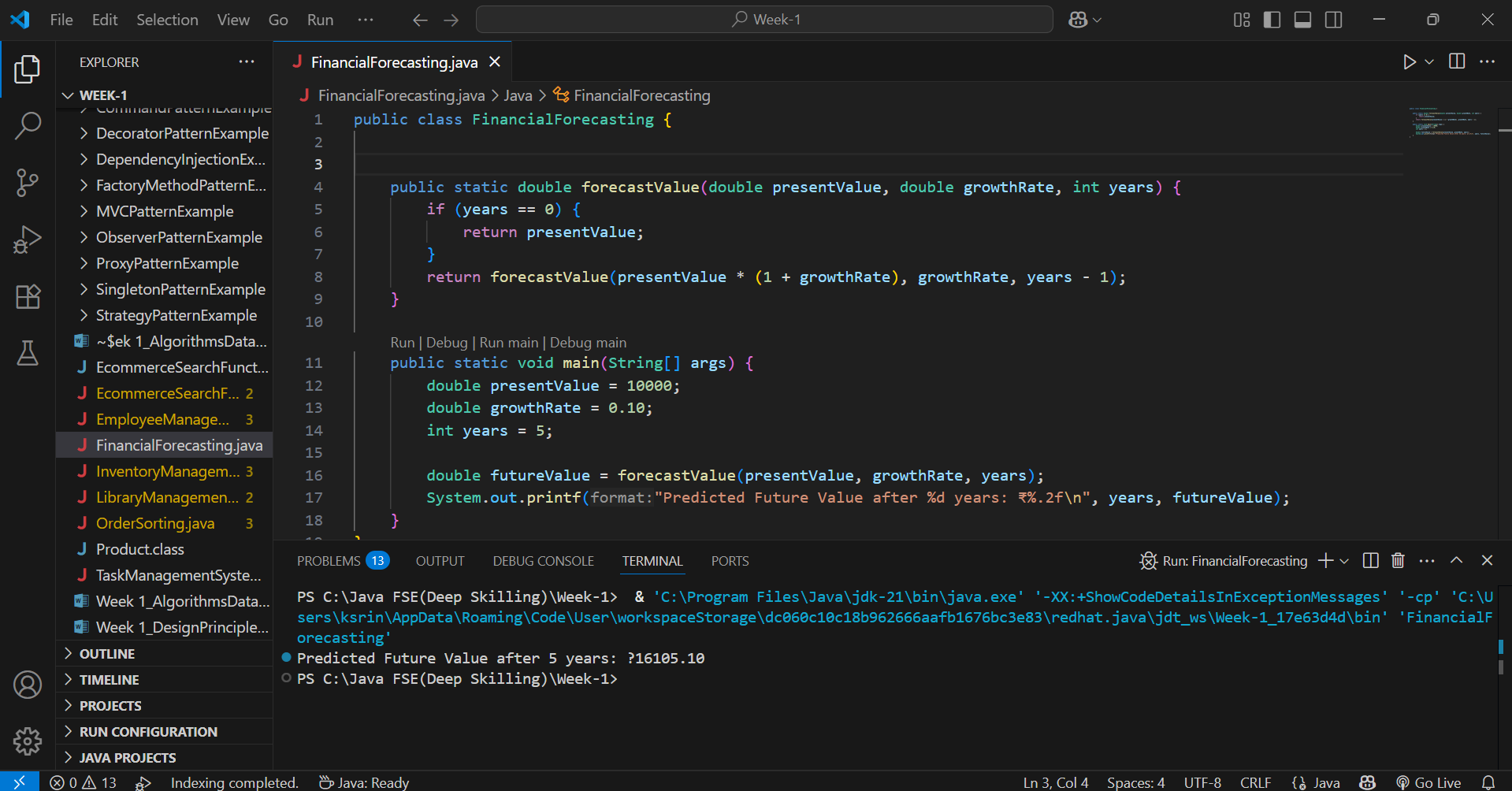
        int years = 5;

        double futureValue = forecastValue(presentValue, growthRate, years);

        System.out.printf("Predicted Future Value after %d years: ₹%.2f\n", years, futureValue);

    }

}



**Analysis**

**🔹 Time Complexity**

* Recursive calls = **O(n)** (n = number of years)
* Each call does one multiplication → linear recursion

**🔹 Drawback**

* Stack overhead for each recursive call
* Inefficient for very large n

**Optimization**

1. **Use Iteration Instead of Recursion**
   * Loop from 1 to n, multiply each year
   * Saves stack memory
2. **Use Exponentiation by Squaring**
   * Reduces time from **O(n)** to **O(log n)**
   * Efficient for large exponents