**WEEK-1**

**Data Structures and Algorithms HandsOn**

**Exercise 1: Inventory Management System:**

When you're dealing with a large number of products in an inventory system, it's not just about storing the data — it's about being able to find, update, or remove items as needed quickly. If the system isn’t built with the right data structures, even basic tasks like searching for a product or changing its quantity can become slow as the inventory grows. Using efficient data structures and algorithms helps keep things fast and smooth, even when the data gets big. It’s the key to making sure the system works well under pressure.

**Suitable Data Structures:**

* **HashMap** is ideal when we want fast access to products using productId as a key. It provides **O(1)** average time for insert, search, and delete.
* **ArrayList** can be used for simple lists, but it takes **O(n)** for search, which is inefficient for large datasets.

Hence, for a warehouse inventory system, **HashMap** is a better fit.

**Code:**

**Inventory.java:**

package week1.topic2.ex1;

import java.util.\*;

class Product {

    private String productId;

    private String productName;

    private int quantity;

    private double price;

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

        this.productId = productId;

        this.productName = productName;

        this.quantity = quantity;

        this.price = price;

    }

    public void setProductName(String name) {

        this.productName = name;

    }

    public void setQuantity(int qty) {

        this.quantity = qty;

    }

    public void setPrice(double price) {

        this.price = price;

    }

    public String getProductId() {

        return productId;

    }

    public void display() {

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

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

        System.out.println("Quantity: " + quantity);

        System.out.println("Price: " + price);

        System.out.println();

    }

}

class Inventory {

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

    public void addProduct(Product product) {

        productMap.put(product.getProductId(), product);

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

    }

    public void updateProduct(String id, String name, int qty, double price) {

        Product product = productMap.get(id);

        if (product != null) {

            product.setProductName(name);

            product.setQuantity(qty);

            product.setPrice(price);

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

        }

    }

    public void deleteProduct(String id) {

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

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

        }

    }

    public void showInventory() {

        System.out.println();

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

            product.display();

        }

    }

}

**Main.java:**

package week1.topic2.ex1;

public class Main {

    public static void main(String[] args) {

        Inventory inventory = new Inventory();

        Product p1 = new Product("P101", "Laptop", 10, 75000);

        Product p2 = new Product("P102", "Mouse", 50, 450);

        Product p3 = new Product("P103", "Keyboard", 30, 900);

        inventory.addProduct(p1);

        inventory.addProduct(p2);

        inventory.addProduct(p3);

        inventory.updateProduct("P102", "Wireless Mouse", 45, 550);

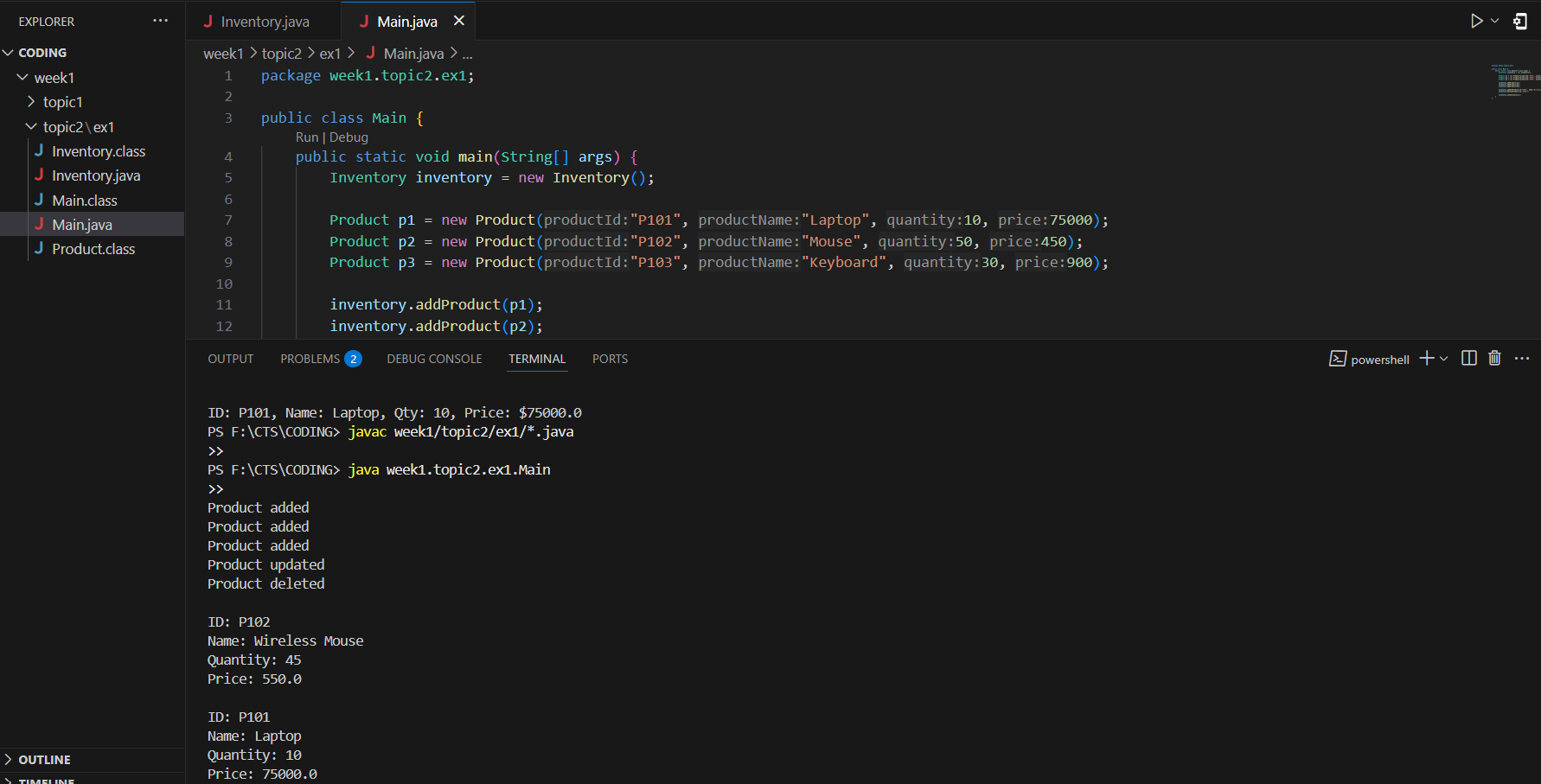
        inventory.deleteProduct("P103");

        inventory.showInventory();

    }

}

**Output:**

****

**Time Complexity Analysis:**

* Add Product - O(1) average
* Update Product - O(1) average
* Delete Product - O(1) average
* Show All Products - O(n) where n is the number of products

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

**Big O Notation** is used to measure the time or space complexity of an algorithm. It helps us understand how an algorithm performs as the input size grows, ignoring constant factors.

**Search Operation Cases:**

* **Best Case:** The item is found in the first few checks.
* **Average Case:** The item is somewhere in the middle.
* **Worst Case:** The item is either not present or at the end (takes the most time).

For example:

* In **linear search**, the best case is O(1), average is O(n/2) ~ O(n), worst is O(n).
* In **binary search**, the best case is O(1), average and worst are O(log n)

**Code:**

**Product.java:**

package week1.topic2.ex2;

public class Product {

    String productId;

    String productName;

    String category;

    public Product(String id, String name, String category) {

        this.productId = id;

        this.productName = name;

        this.category = category;

    }

    public void display() {

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

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

        System.out.println("Category: " + category);

        System.out.println();

    }

}

**Main.java:**

package week1.topic2.ex2;

import java.util.Arrays;

import java.util.Comparator;

public class Main {

    public static void main(String[] args) {

        Product[] products = {

            new Product("P001", "Laptop", "Electronics"),

            new Product("P002", "T-shirt", "Clothing"),

            new Product("P003", "Keyboard", "Electronics"),

            new Product("P004", "Sneakers", "Footwear"),

            new Product("P005", "Watch", "Accessories")

        };

        System.out.println("Linear Search for 'Keyboard':");

        linearSearch(products, "Keyboard");

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

        System.out.println("Binary Search for 'Sneakers':");

        binarySearch(products, "Sneakers");

    }

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

        for (Product p : products) {

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

                p.display();

                return;

            }

        }

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

    }

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

        int low = 0;

        int high = products.length - 1;

        while (low <= high) {

            int mid = (low + high) / 2;

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

            if (cmp == 0) {

                products[mid].display();

                return;

            } else if (cmp < 0) {

                high = mid - 1;

            } else {

                low = mid + 1;

            }

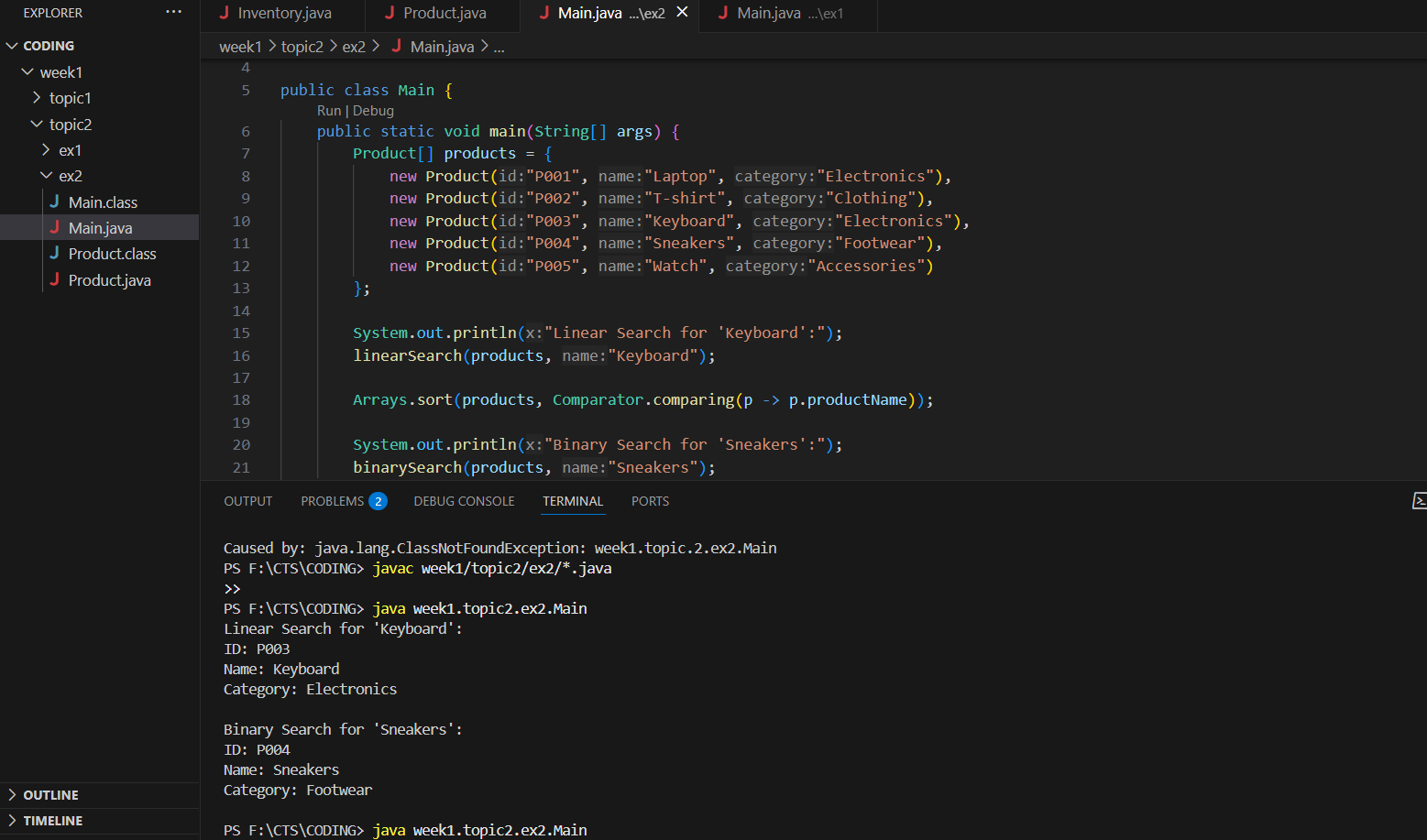
        }

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

    }

}

**Output:**

****

**Time Complexity Analysis:**

| **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) |

**Algorithm is more suitable:**

* Linear search works on unsorted data and is easy to implement but becomes slower as the data grows.
* Binary search is faster but requires sorted data.

For an e-commerce platform, binary search is more suitable because:

* Product listings are usually pre-sorted (by name, ID, price, etc.).
* Performance matters when users search frequently.
* O(log n) scaling is excellent for large inventories.

**Exercise 3: Sorting Customer Orders**

**Bubble Sort:** Compares adjacent elements and swaps them if they’re in the wrong order.  
Repeats until the array is completely sorted.

* Time Complexity: Best: O(n), Average/Worst: O(n²)
* Space Complexity: O(1)

**Insertion Sort:** Builds the final sorted array one item at a time.  
Inserts each element into its correct position.

* Time Complexity: Best: O(n), Average/Worst: O(n²)
* Space Complexity: O(1)

**Quick Sort:** Uses a pivot to divide and conquer by partitioning the array.  
Sorts elements on each side of the pivot recursively.

* Time Complexity: Best/Average: O(n log n), Worst: O(n²)
* Space Complexity: O(log n)

**Merge Sort**: Divides the array into halves, sorts each half, and merges them.  
It’s a stable and consistent divide-and-conquer algorithm.

* Time Complexity: Best/Average/Worst: O(n log n)
* Space Complexity: O(n)

**Code:**

**Order.java:**

package week1.topic2.ex3;

public class Order {

    String orderId;

    String customerName;

    double totalPrice;

    public Order(String id, String name, double price) {

        this.orderId = id;

        this.customerName = name;

        this.totalPrice = price;

    }

    public void display() {

        System.out.println("Order ID: " + orderId);

        System.out.println("Customer: " + customerName);

        System.out.println("Total Price: $" + totalPrice);

        System.out.println();

    }

}

**Main.java:**

package week1.topic2.ex3;

public class Main {

    public static void main(String[] args) {

        Order[] orders = {

            new Order("O101", "Amit", 7500),

            new Order("O102", "Priya", 2450),

            new Order("O103", "Ravi", 9800),

            new Order("O104", "Sneha", 3200),

            new Order("O105", "Kiran", 4500)

        };

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

        for (Order o : orders) o.display();

        System.out.println("Bubble Sort:");

        bubbleSort(orders.clone());

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

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

        for (Order o : orders) o.display();

    }

    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;

                }

            }

        }

        for (Order o : orders) o.display();

    }

    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);

        }

    }

    public 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];

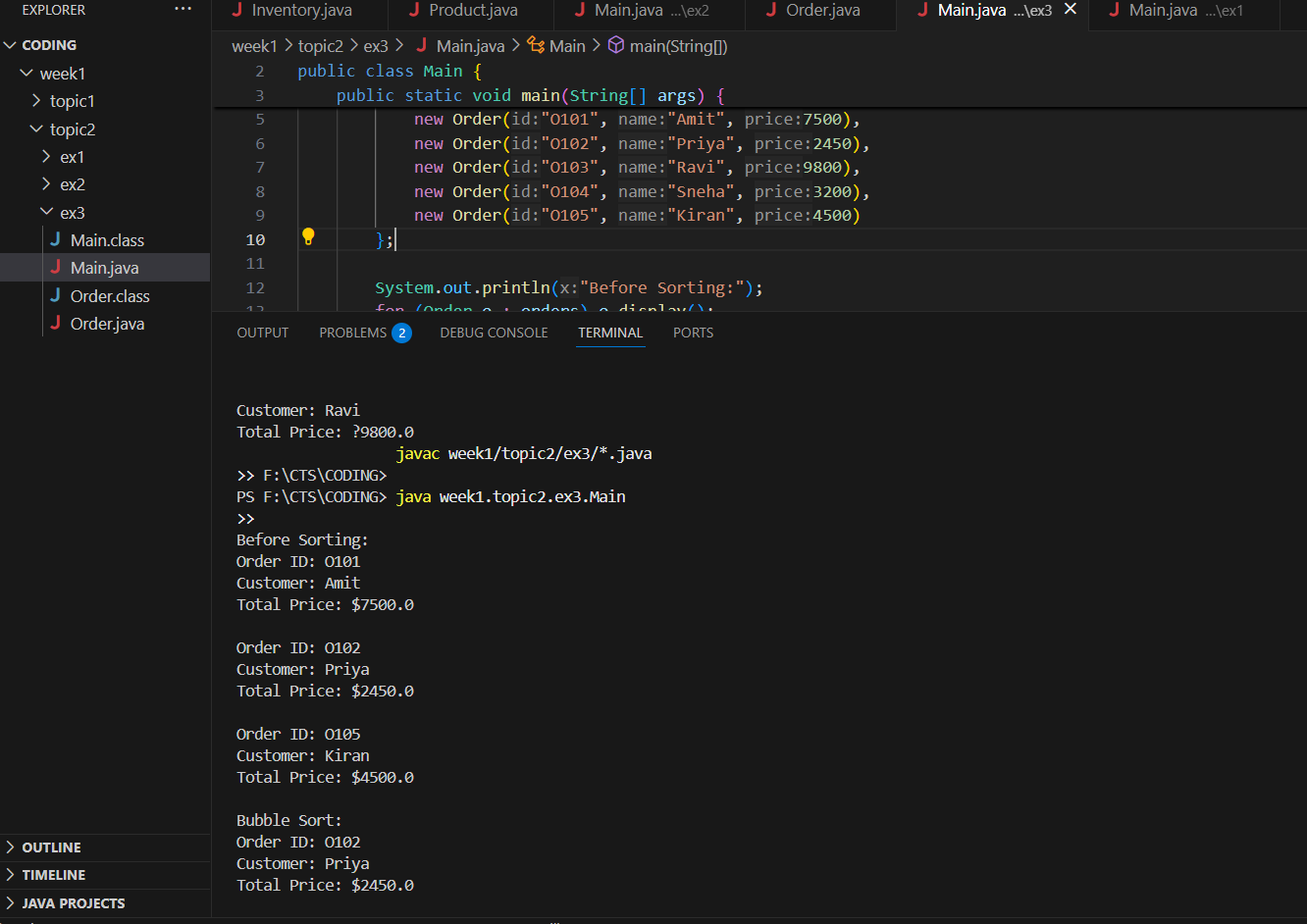
        orders[high] = temp;

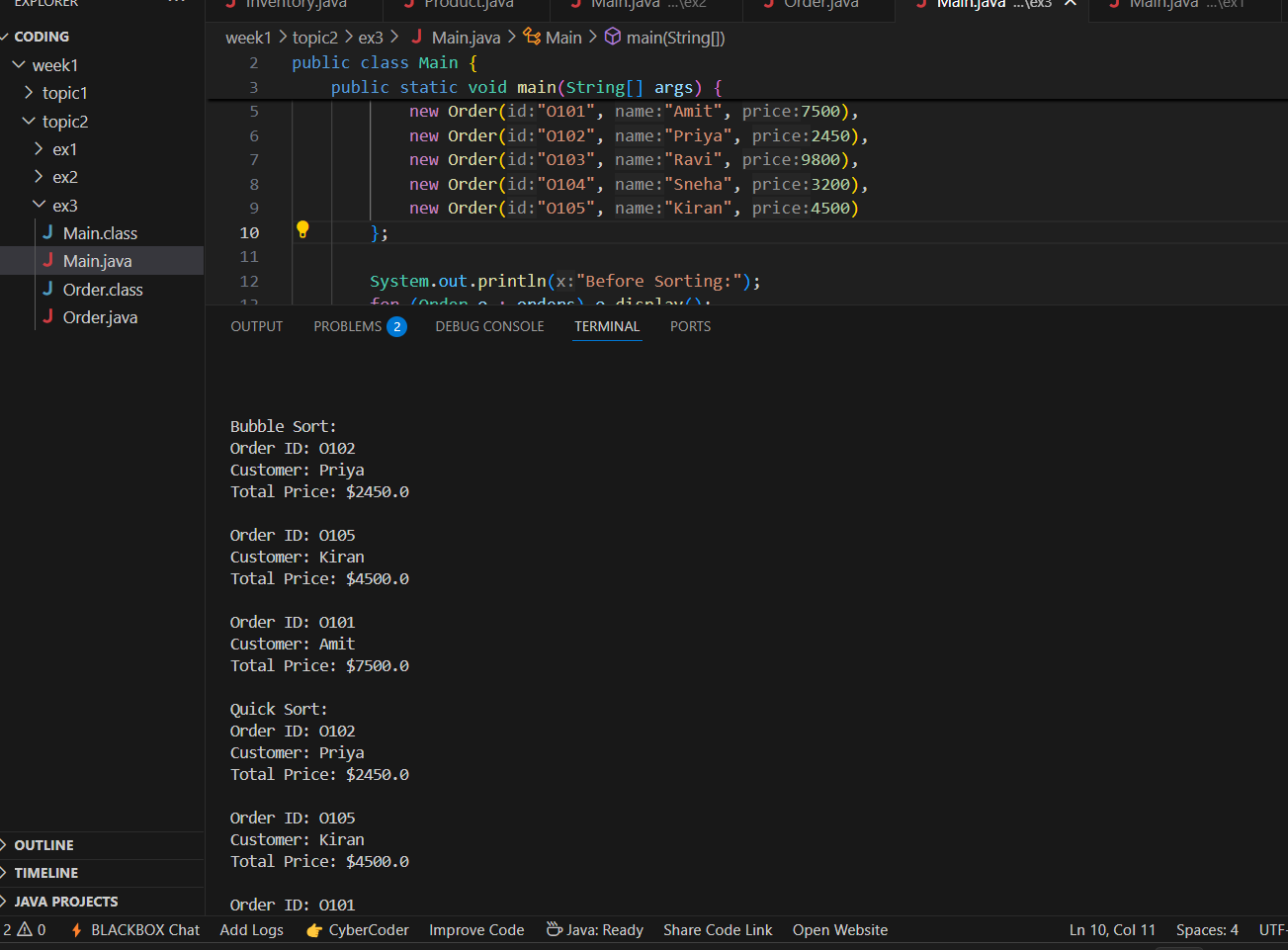
        return i + 1;

    }

}

**Output:**

****

****

**Time Complexity and Analysis:**

| Metric | Bubble Sort | Quick Sort |
| --- | --- | --- |
| Time Complexity (Best) | O(n) | O(n log n) |
| Time Complexity (Average) | O(n²) | O(n log n) |
| Time Complexity (Worst) | O(n²) | O(n²) |
| Space Complexity | O(1) | O(log n) |
| Real-World Use | Rarely used | Very common |

**Why Quick Sort is Preferred:**

* Much faster for large lists
* Works well in memory
* Uses divide-and-conquer efficiently
* Bubble sort becomes very slow as data grows

**Exercise 4: Employee Management System**

Arrays in memory are stored in contiguous memory blocks. Each element is placed right next to the other, and you can directly access any element using its index.

**Advantages:**

* Fast access: Accessing an element by index is O(1) time.
* Memory efficient: Minimal overhead compared to dynamic data structures.

**Code:**

**Employee.java:**

package week1.topic2.ex4;

public class Employee {

    String employeeId;

    String name;

    String position;

    double salary;

    public Employee(String id, String name, String position, double salary) {

        this.employeeId = id;

        this.name = name;

        this.position = position;

        this.salary = salary;

    }

    public void display() {

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

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

        System.out.println("Position: " + position);

        System.out.println("Salary: ₹" + salary);

        System.out.println();

    }

}

**EmployeeSystem.java:**

package week1.topic2.ex4;

public class EmployeeSystem {

    Employee[] employees = new Employee[100];

    int size = 0;

    public void addEmployee(Employee emp) {

        if (size < employees.length) {

            employees[size++] = emp;

        }

    }

    public void searchEmployee(String id) {

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

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

                employees[i].display();

                return;

            }

        }

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

    }

    public void traverseEmployees() {

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

            employees[i].display();

        }

    }

    public void deleteEmployee(String id) {

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

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

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

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

                }

                employees[size - 1] = null;

                size--;

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

                return;

            }

        }

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

    }

}

**Main.java:**

package week1.topic2.ex4;

public class Main {

    public static void main(String[] args) {

        EmployeeSystem system = new EmployeeSystem();

        Employee e1 = new Employee("E001", "Aarav", "Developer", 70000);

        Employee e2 = new Employee("E002", "Meera", "Manager", 95000);

        Employee e3 = new Employee("E003", "Rohan", "Designer", 65000);

        system.addEmployee(e1);

        system.addEmployee(e2);

        system.addEmployee(e3);

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

        system.traverseEmployees();

        System.out.println("Searching for E002:");

        system.searchEmployee("E002");

        System.out.println("Deleting E001:");

        system.deleteEmployee("E001");

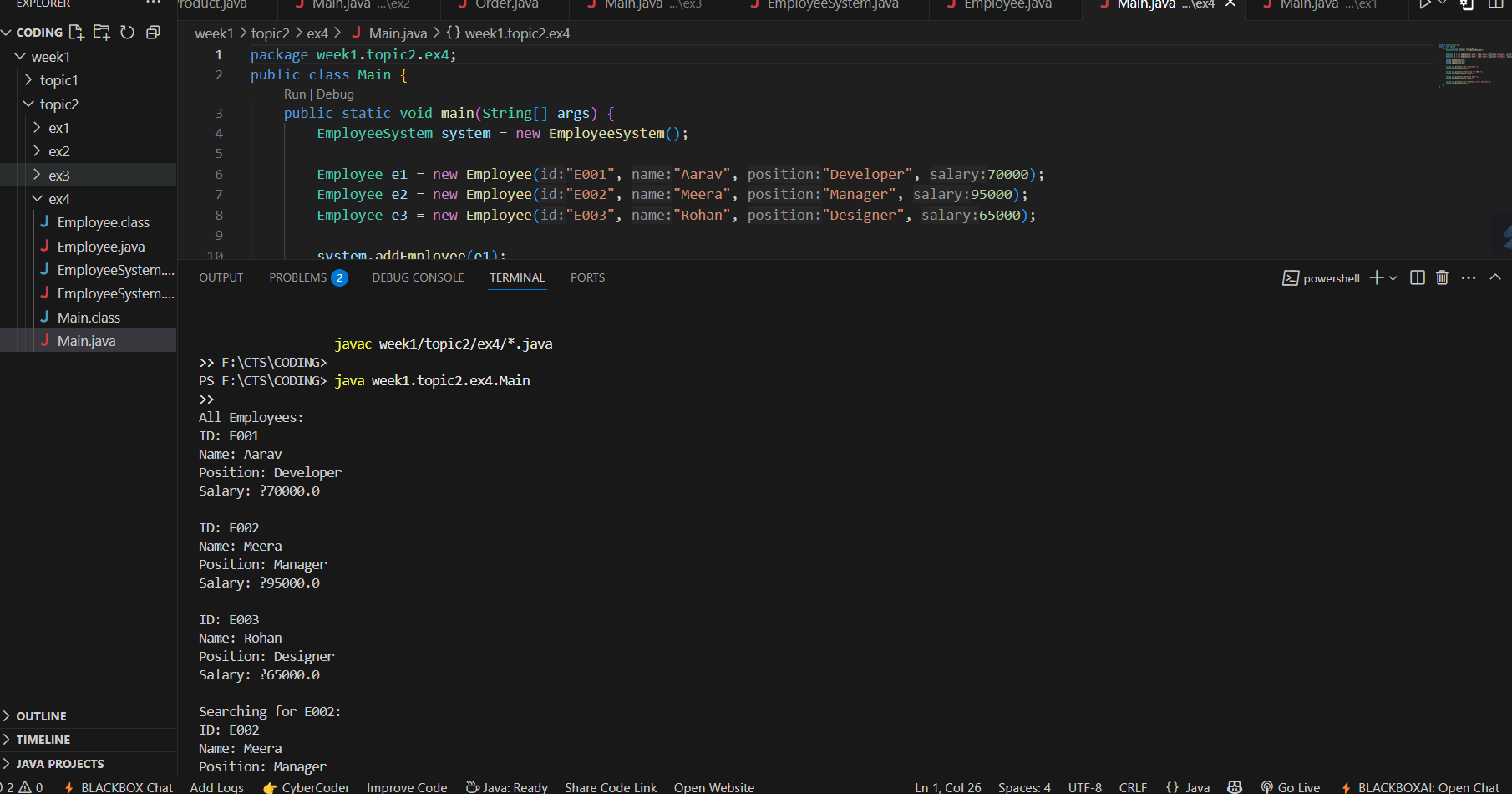
        System.out.println("All Employees after deletion:");

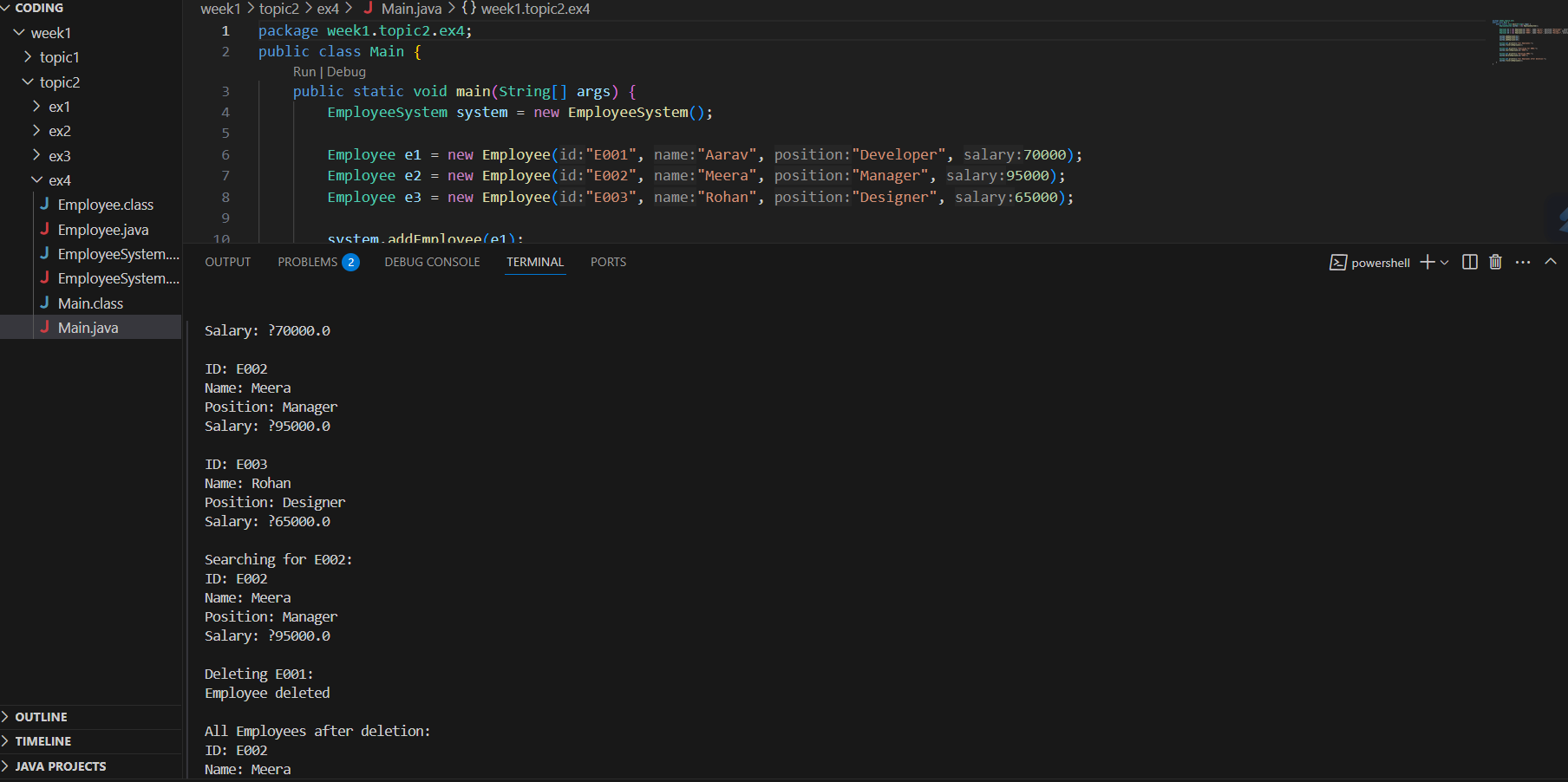
        system.traverseEmployees();

    }

}

**Output:**

****

****

**Time Complexity and Analysis:**

* **Add:** O(1) - Insert at next index
* **Search:** O(n) - Linear search through the array
* **Traverse:** O(n) - Go through all employees
* **Delete:** O(n) - Shift elements after deletion

**Limitations of Arrays**

* **Fixed size**: Can’t grow once defined
* **Costly deletion/insertion**: Elements need shifting
* **Use when**: The number of employees is roughly fixed and fast access is important

**Exercise 5: Task Management System**

**Singly Linked List:**Each node stores data and a reference to the next node. It can only be traversed in one direction.

**Doubly Linked List:**Each node stores data, a reference to the next node, and a reference to the previous node. It supports two-way traversal.

**Code:**

**TaskList.java:**

package week1.topic2.ex5;

class Node {

    Task task;

    Node next;

    public Node(Task task) {

        this.task = task;

        this.next = null;

    }

}

public class TaskList {

    Node head = null;

    public void addTask(Task task) {

        Node newNode = new Node(task);

        if (head == null) {

            head = newNode;

        } else {

            Node temp = head;

            while (temp.next != null) temp = temp.next;

            temp.next = newNode;

        }

    }

    public void searchTask(String id) {

        Node temp = head;

        while (temp != null) {

            if (temp.task.taskId.equalsIgnoreCase(id)) {

                temp.task.display();

                return;

            }

            temp = temp.next;

        }

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

    }

    public void deleteTask(String id) {

        if (head == null) return;

        if (head.task.taskId.equalsIgnoreCase(id)) {

            head = head.next;

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

            return;

        }

        Node prev = head;

        Node curr = head.next;

        while (curr != null) {

            if (curr.task.taskId.equalsIgnoreCase(id)) {

                prev.next = curr.next;

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

                return;

            }

            prev = curr;

            curr = curr.next;

        }

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

    }

    public void traverseTasks() {

        Node temp = head;

        while (temp != null) {

            temp.task.display();

            temp = temp.next;

        }

    }

}

**Task.java:**

package week1.topic2.ex5;

public class Task {

    String taskId;

    String taskName;

    String status;

    public Task(String id, String name, String status) {

        this.taskId = id;

        this.taskName = name;

        this.status = status;

    }

    public void display() {

        System.out.println("Task ID: " + taskId);

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

        System.out.println("Status: " + status);

        System.out.println();

    }

}

**Main.java:**

package week1.topic2.ex5;

public class Main {

    public static void main(String[] args) {

        TaskList taskList = new TaskList();

        Task t1 = new Task("T101", "Design Homepage", "Pending");

        Task t2 = new Task("T102", "Write API", "In Progress");

        Task t3 = new Task("T103", "Test Features", "Completed");

        taskList.addTask(t1);

        taskList.addTask(t2);

        taskList.addTask(t3);

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

        taskList.traverseTasks();

        System.out.println("Searching for T102:");

        taskList.searchTask("T102");

        System.out.println("Deleting T101:");

        taskList.deleteTask("T101");

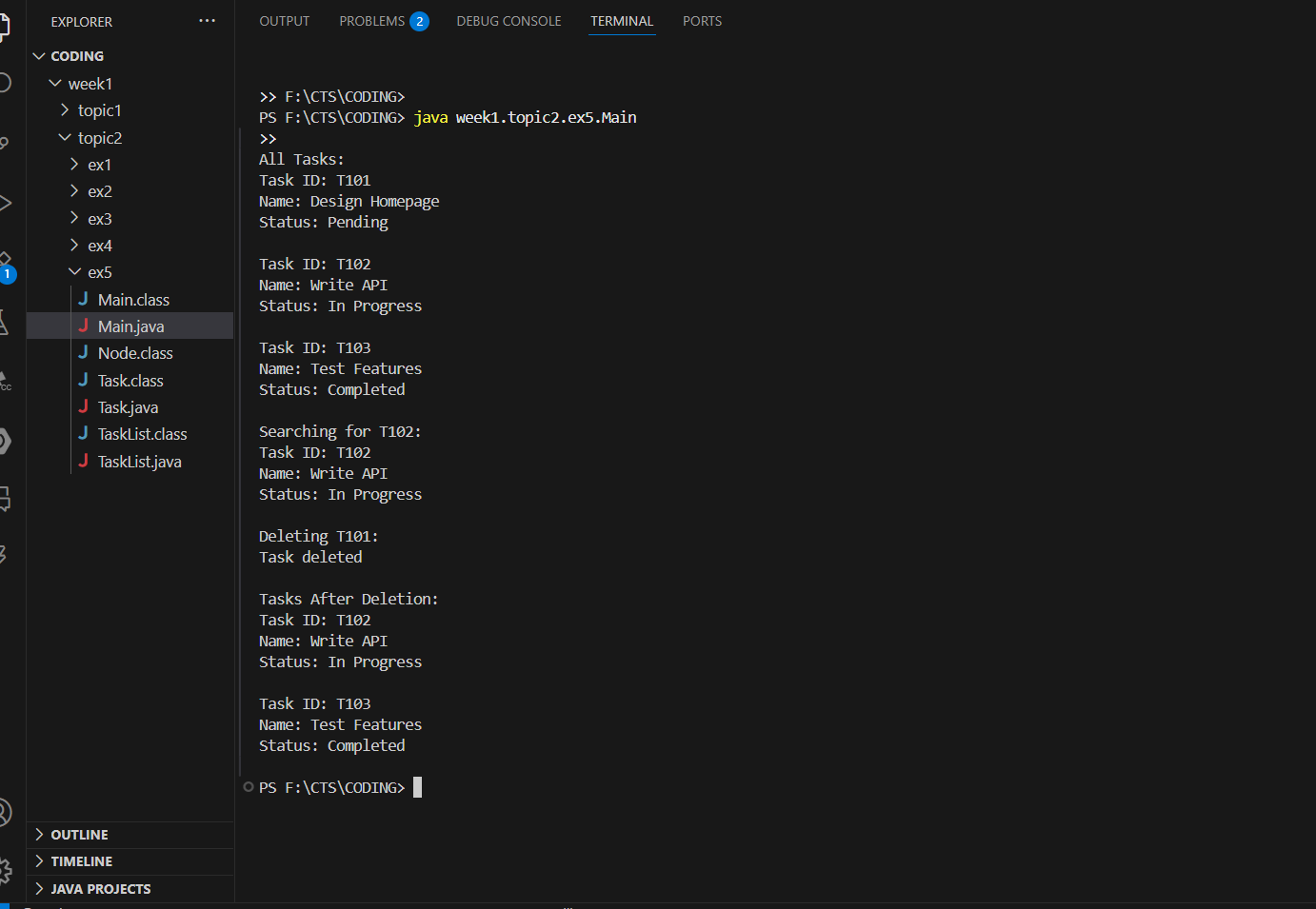
        System.out.println("Tasks After Deletion:");

        taskList.traverseTasks();

    }

}

**Output:**

****

**Time Complexity and Analysis:**

* **Add:** O(n) – Traverse to end and add
* **Search:** O(n) – Check each node
* **Traverse:** O(n) – Visit each node
* **Delete:** O(n) – Search and relink nodes

**Advantages of Linked Lists over Arrays**

* Can grow or shrink dynamically (no fixed size needed)
* Insertions and deletions are easier (no shifting required)
* Memory is allocated only when needed

**Exercise 6: Library Management System**

**Linear Search:**  
Goes through each element one by one to find the match. Useful for small or unsorted lists.  
Time complexity: **O(n)**

**Binary Search:**  
Works only on sorted data. Repeatedly divides the search space in half.  
Time complexity: **O(log n)**

**Code:**

**Book.java:**

package week1.topic2.ex6;

public class Book {

    String bookId;

    String title;

    String author;

    public Book(String id, String title, String author) {

        this.bookId = id;

        this.title = title;

        this.author = author;

    }

    public void display() {

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

        System.out.println("Title: " + title);

        System.out.println("Author: " + author);

        System.out.println();

    }

}

**Main.java:**

package week1.topic2.ex6;

import java.util.Arrays;

import java.util.Comparator;

public class Main {

    public static void main(String[] args) {

        Book[] books = {

            new Book("B101", "The Alchemist", "Paulo Coelho"),

            new Book("B102", "Atomic Habits", "James Clear"),

            new Book("B103", "Clean Code", "Robert C. Martin"),

            new Book("B104", "Java Basics", "John Doe"),

            new Book("B105", "Zebra Tales", "Jane Smith")

        };

        System.out.println("Linear Search for 'Clean Code':");

        linearSearch(books, "Clean Code");

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

        System.out.println("Binary Search for 'Java Basics':");

        binarySearch(books, "Java Basics");

    }

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

        for (Book book : books) {

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

                book.display();

                return;

            }

        }

        System.out.println("Book not found\n");

    }

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

        int low = 0;

        int high = books.length - 1;

        while (low <= high) {

            int mid = (low + high) / 2;

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

            if (cmp == 0) {

                books[mid].display();

                return;

            } else if (cmp < 0) {

                high = mid - 1;

            } else {

                low = mid + 1;

            }

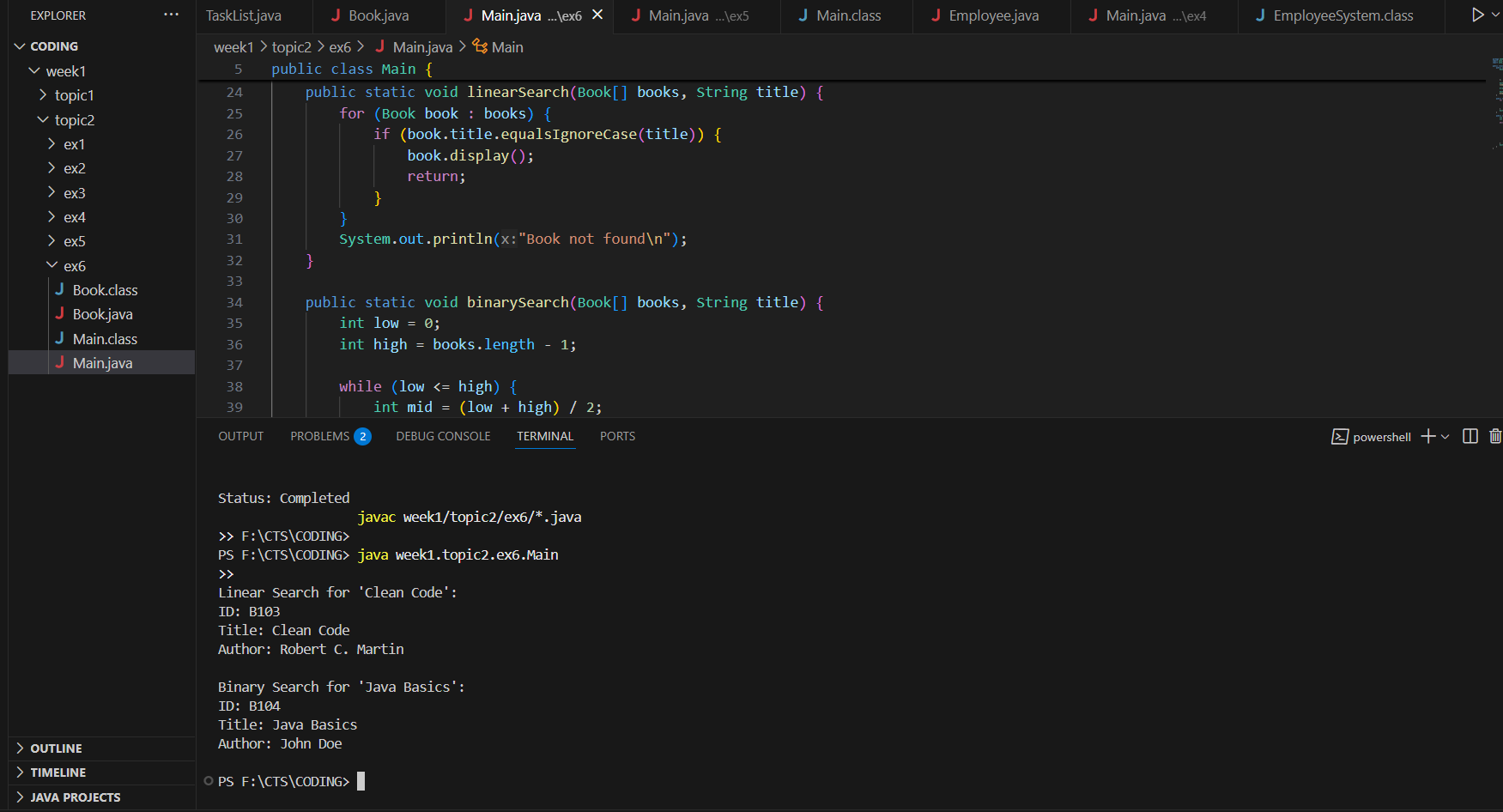
        }

        System.out.println("Book not found\n");

    }

}

**Output:**

****

**Time Complexity and Analysis:**

**Linear Search:**

* Use when data is **unsorted** or for **small datasets**.
* Simple to implement and doesn’t require pre-sorting.

**Binary Search:**

* Use for **sorted** data and **large datasets**.
* Much faster than linear search on large arrays due to its **O(log n)** performance.

**Exercise 7: Financial Forecasting**

**Recursion:**

Recursion is a technique where a method calls itself to solve a smaller part of the problem.It’s especially helpful when a problem can be broken into repeating sub-problems, like calculating compound growth or Fibonacci numbers

**Code:**

**Main.java:**

package week1.topic2.ex7;

public class Main {

    public static void main(String[] args) {

        double presentValue = 10000;

        double growthRate = 0.08;

        int years = 5;

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

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

    }

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

        if (years == 0) {

            return value;

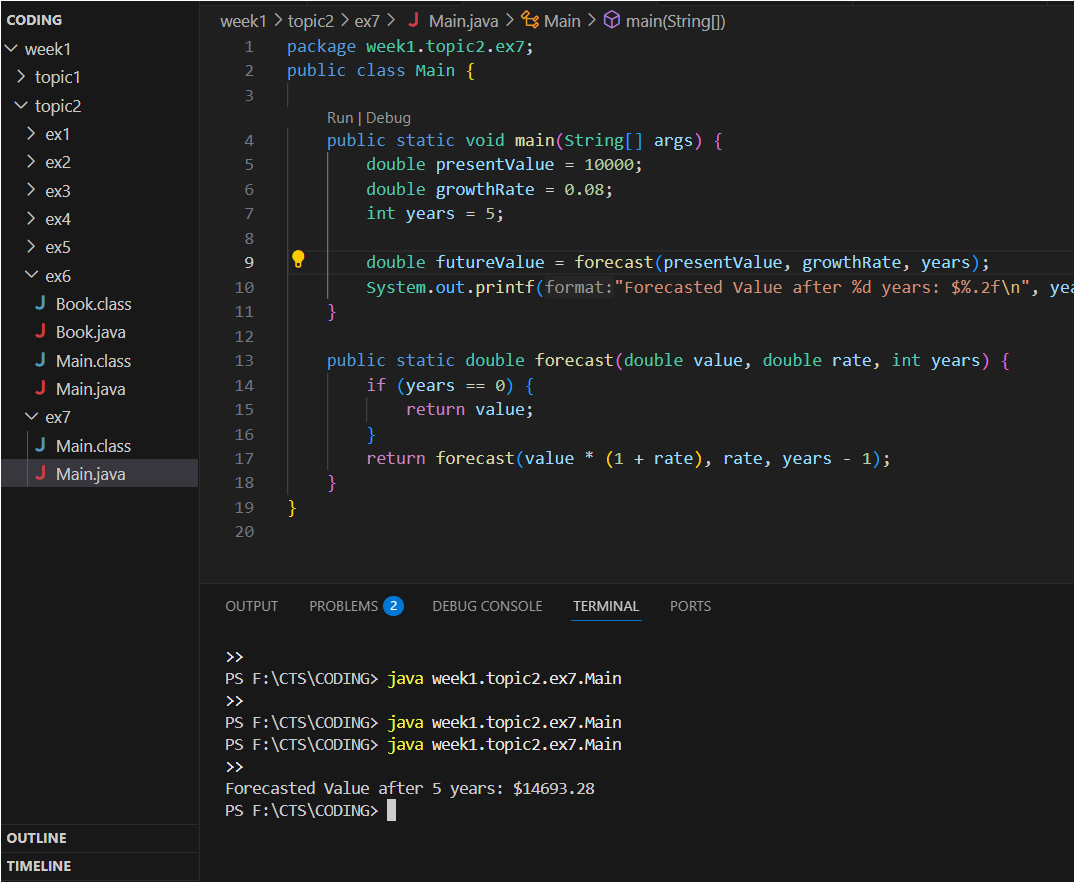
        }

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

    }

}

**Output:**

****

**Time Complexity and Analysis:**The recursive function runs once per year, so the time complexity is O(n) where n is the number of years.

**Optimization:**  
This recursion is already linear and doesn't have overlapping subproblems, so it doesn't need memoization.  
However, for more complex forecasting models (like Fibonacci-style predictions), you can optimize recursion by using memoization or converting to iteration to avoid stack overflow.