**Exercise 1: Inventory Management System**

**Scenario:**

You are developing an inventory management system for a warehouse. Efficient data storage and retrieval are crucial.

**Explain why data structures and algorithms are essential in handling large inventories.**

Large inventories contain thousands of products which require efficient methods to create, retrieve, update, and delete. So, it is crucial to use appropriate data structures that allow to perform operations efficiently.

**Discuss the types of data structures suitable for this problem.**

HashMap can be used to store the product with their unique Product ID. The get() method in hashmaps retrieves the Product in O(1) time complexity. Adding Products using the put() method is also done in O(1) time complexity. So HashMap can be considered suitable for this problem.

To store different products in the inventory, ArrayList can also be used. But there are a few limitations when using arraylist for inventory management. Adding products to the arraylist can be done in O(1) time complexity. But to retrieve and update or delete the product, the retrieval process can take O(n) time complexity. This is not generally considered good in terms of efficiency of the program.

Other data structures such as LinkedList can also be used to store products in the inventory, but the time complexity for adding, retrieving and deleting the products would be high.

**Project: InventoryManagementSystem**

**Product.java**

public class Product {  
 private int productId;  
 private String productName;  
 private int quantity;  
 private 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 int getProductId() {  
 return productId;  
 }  
  
 public void setProductId(int productId) {  
 this.productId = productId;  
 }  
  
 public String getProductName() {  
 return productName;  
 }  
  
 public void setProductName(String productName) {  
 this.productName = productName;  
 }  
  
 public int getQuantity() {  
 return quantity;  
 }  
  
 public void setQuantity(int quantity) {  
 this.quantity = quantity;  
 }  
  
 public double getPrice() {  
 return price;  
 }  
  
 public void setPrice(double price) {  
 this.price = price;  
 }  
  
 public void printProduct(){  
 System.*out*.println("Product Id: " + productId + "\nProduct Name: " + productName + "\nProduct Quantity: " + quantity + "\nProduct Price: " + price + "\n");  
 }  
}

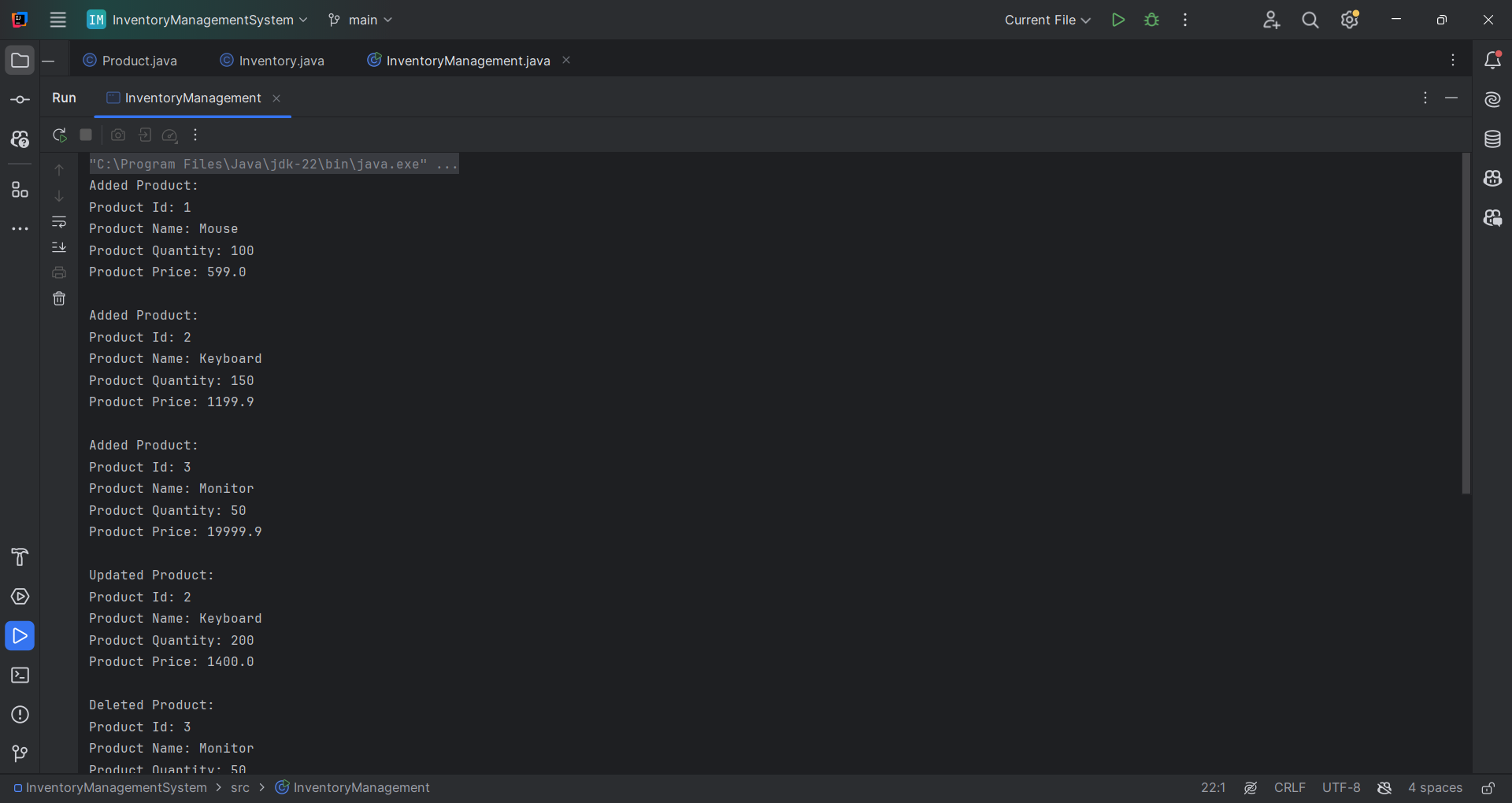
**Inventory.java**

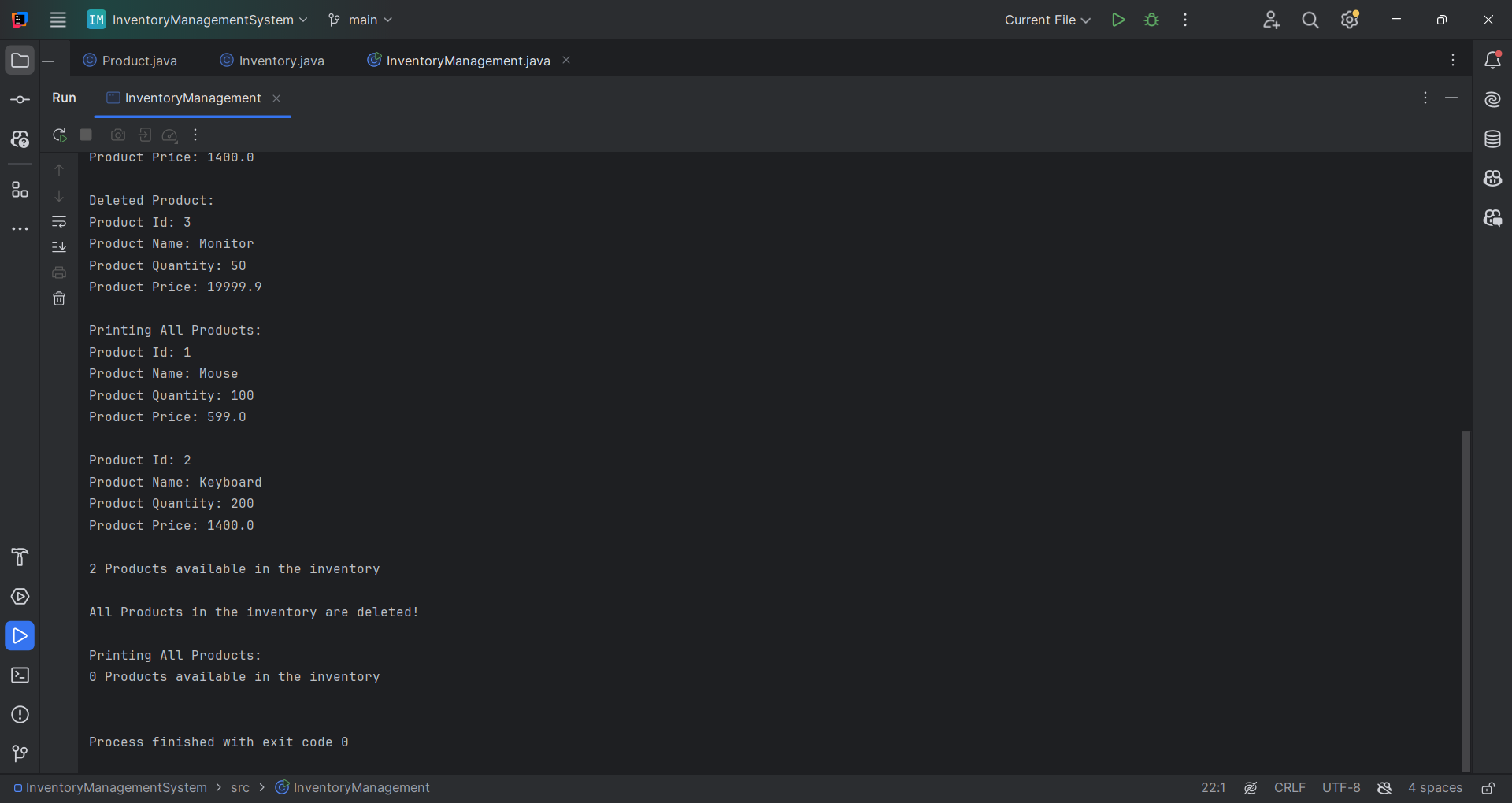
import java.util.\*;  
  
public class Inventory {  
 private HashMap<Integer, Product> inventory;  
  
 public Inventory(){  
 inventory = new HashMap<>();  
 }  
  
 public void addProduct(Product product){  
 inventory.put(product.getProductId(), product);  
 System.*out*.println("Added Product:");  
 product.printProduct();  
 }  
  
 public void updateProduct(int productId, int newQuantity, double newPrice){  
 if(inventory.containsKey((productId))){  
 Product product = inventory.get(productId);  
 product.setQuantity(newQuantity);  
 product.setPrice(newPrice);  
 System.*out*.println("Updated Product:");  
 product.printProduct();  
 }else{  
 System.*out*.println("Product ID not found! Please enter valid Product ID");  
 }  
 }  
  
 public void deleteProduct(int productId){  
 if(inventory.containsKey(productId)){  
 Product deleted = inventory.remove(productId);  
 System.*out*.println("Deleted Product:");  
 deleted.printProduct();  
 }else{  
 System.*out*.println("Product ID not found! Please enter valid Product ID");  
 }  
 }  
  
 public void deleteAllProducts(){  
 inventory.clear();  
 System.*out*.println("All Products in the inventory are deleted!\n");  
 }  
  
 public void printAllProducts(){  
 int count = 0;  
 System.*out*.println("Printing All Products:");  
 for(Product product : inventory.values()){  
 product.printProduct();  
 count++;  
 }  
 System.*out*.println(count + " Products available in the inventory\n");  
 }  
}

**InventoryManagement.java**

public class InventoryManagement {  
 public static void main(String[] args){  
 Inventory inventory = new Inventory();  
 Product product1 = new Product(1, "Mouse", 100, 599.0);  
 Product product2 = new Product(2, "Keyboard", 150, 1199.9);  
 Product product3 = new Product(3, "Monitor", 50, 19999.9);  
  
 inventory.addProduct(product1);  
 inventory.addProduct(product2);  
 inventory.addProduct(product3);  
  
 inventory.updateProduct(2, 200, 1400.00);  
  
 inventory.deleteProduct(3);  
  
 inventory.printAllProducts();  
  
 inventory.deleteAllProducts();  
 inventory.printAllProducts();  
 }  
}

**Output:**





**Analysis:**

**Analyze the time complexity of each operation (add, update, delete) in your chosen data structure.**

1. Add: As we are using a hashmap to store products, adding elements to the hashmap can be done in O(1) time complexity.
2. Update: Retrieving products from the hashmap and updating them is done in O(1) time complexity.
3. Delete: Deleting elements from the hashmap also takes O(1) time complexity.

**Discuss how you can optimize these operations.**

Using a hashmap to store the products by their product id will help to easily add and retrieve products from the hashmap.

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

**Scenario:**

You are working on the search functionality of an e-commerce platform. The search needs to be optimized for fast performance.

**Explain Big O notation and how it helps in analyzing algorithms.**

Big O Notation is a mathematical notation used to describe the performance or complexity of an algorithm. It calculates the performance of the algorithm in terms of Time Complexity and Space complexity. It helps developers know the worst possible time and space that the algorithm can use. For example, if we perform a linear search in an array, the worst case is when the element is at the end of the array. So, the time complexity would be O(n) for n elements.

**Describe the best, average, and worst-case scenarios for search operations.**

Best-case scenario: Best-case scenario is when the desired element is found immediately. For example, when performing linear search on an array and the desired element is the first element. This is called best case scenario.

Average-case scenario: Average-case scenario is when the desired element is located somewhere in the middle. For example, when performing linear search on an array and the desired element is the middle element. This is called average-case scenario.

Worst-case scenario: Worst-case scenario is when the desired element is not found or it is the last element. For example, when performing linear search on an array, and the desired element is at the end of the array.

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

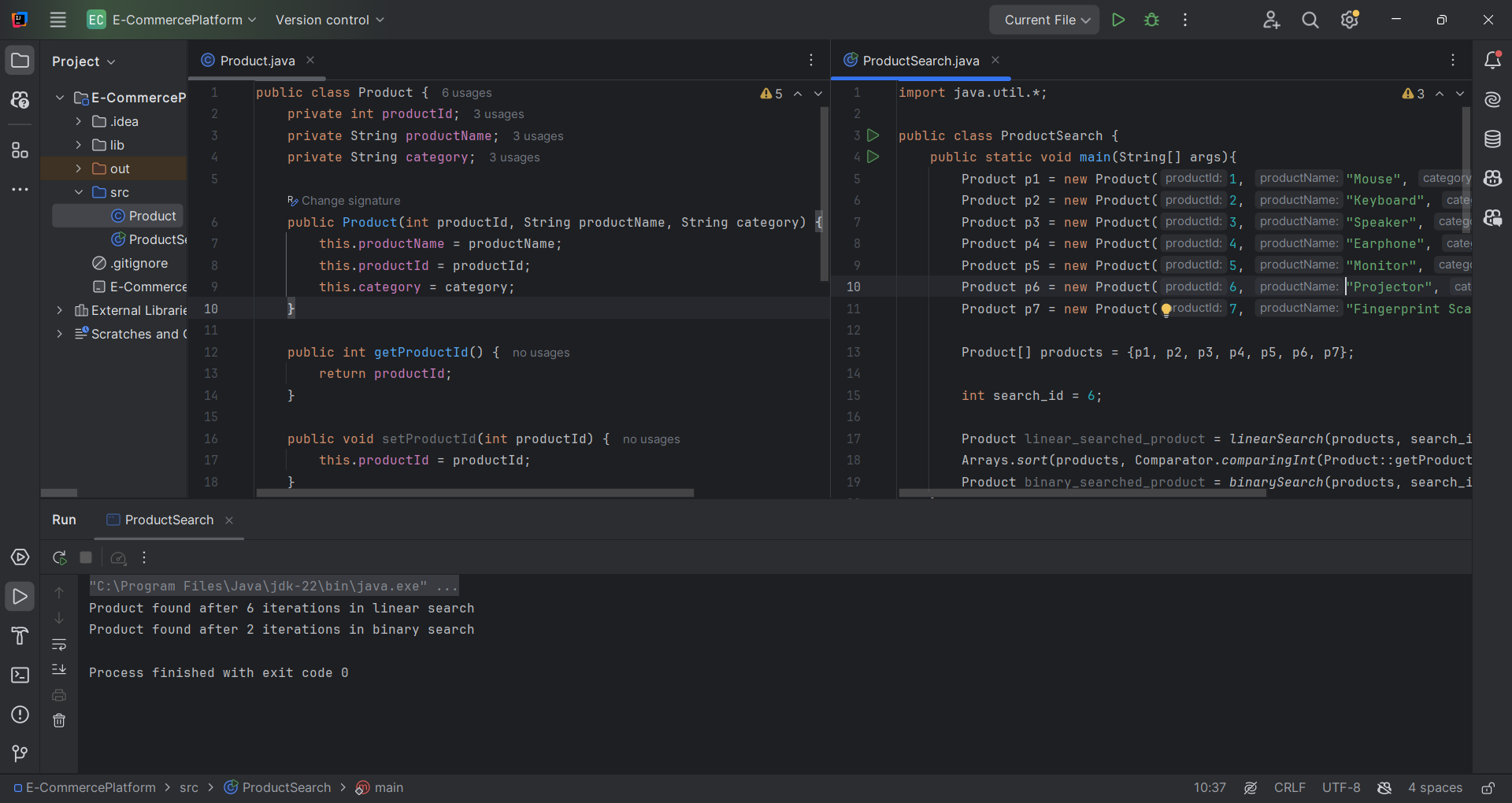
**Product.java**

public class Product {  
 private int productId;  
 private String productName;  
 private String category;  
  
 public Product(int productId, String productName, String category) {  
 this.productName = productName;  
 this.productId = productId;  
 this.category = category;  
 }  
  
 public int getProductId() {  
 return productId;  
 }  
  
 public void setProductId(int productId) {  
 this.productId = productId;  
 }  
  
 public String getProductName() {  
 return productName;  
 }  
  
 public void setProductName(String productName) {  
 this.productName = productName;  
 }  
  
 public String getCategory() {  
 return category;  
 }  
  
 public void setCategory(String category) {  
 this.category = category;  
 }  
}

**ProductSearch.java**

import java.util.\*;  
  
public class ProductSearch {  
 public static void main(String[] args){  
 Product p1 = new Product(1, "Mouse", "Input devices");  
 Product p2 = new Product(2, "Keyboard", "Input devices");  
 Product p3 = new Product(3, "Speaker", "Output devices");  
 Product p4 = new Product(4, "Earphone", "Output devices");  
 Product p5 = new Product(5, "Monitor", "Output devices");  
 Product p6 = new Product(6, "Projector", "Output device");  
 Product p7 = new Product(7, "Fingerprint Scanner", "Input device");  
  
 Product[] products = {p1, p2, p3, p4, p5, p6, p7};  
  
 int search\_id = 6;  
  
 Product linear\_searched\_product = *linearSearch*(products, search\_id);  
 Arrays.*sort*(products, Comparator.*comparingInt*(Product::getProductId));  
 Product binary\_searched\_product = *binarySearch*(products, search\_id);  
 }  
 public static Product linearSearch(Product[] products, int search\_id){  
 for(int i = 0; i < products.length; i++){  
 if(products[i].getProductId() == search\_id){  
 System.*out*.println("Product found after " + (i + 1) + " iterations in linear search");  
 return products[i];  
 }  
 }  
 System.*out*.println("Product not found");  
 return null;  
 }  
  
 public static Product binarySearch(Product[] products, int search\_id){  
 int i = 0;  
 int low = 0;  
 int high = products.length - 1;  
 int mid = (low + high)/2;  
 while(low <= high){  
 i++;  
 mid = (low + high)/2;  
 if(products[mid].getProductId() == search\_id){  
 System.*out*.println("Product found after " + i + " iterations in binary search");  
 return products[mid];  
 }else if(products[mid].getProductId() < search\_id){  
 low = mid + 1;  
 }else{  
 high = mid - 1;  
 }  
 }  
 System.*out*.println("Product not found!!!");  
 return null;  
 }  
}

**Output:**



**Analysis:**

1. **Compare the time complexity of linear and binary search algorithms.**

**Linear Search:** Time Complexity - O(n)

**Binary Search:** Time Complexity – O(log n)

Linear search checks from the beginning to the end until the desired element is found. So the worst case time complexity would be O(n). In our program, we have searched for the element with the productId. Linear search takes 6 iterations to find the element.

Binary search checks whether the middle element is the desired element, else it checks whether the element is after the middle element or before the middle element as the array is sorted. This process is repeated until the desired element is found. Binary Search takes O(log n) time complexity. The same product with the productid 6 is found in just 2 iterations while using binary search.

1. **Discuss which algorithm is more suitable for your platform and why.**

Both search algorithms have their own advantages and disadvantages. In terms of search time complexity, Binary search is much faster. But whenever a new product is added to the inventory, the inventory must be sorted.

In conclusion for large number of products, binary search is more suitable when sorted by the productid.

**Exercise 3: Sorting Customer Orders**

**Scenario:**

You are tasked with sorting customer orders by their total price on an e-commerce platform. This helps in prioritizing high-value orders.

**Explain different sorting algorithms (Bubble Sort, Insertion Sort, Quick Sort, Merge Sort).**

1. Bubble Sort:

Bubble sort works by iteratively comparing the adjacent items and swaps them if they are not in the right order.

Start from the first element. Compare first and second element. If the first is greater than the second element, swap them. Move to the next pair and repeat the process. At the end of the first traversal, the largest element moves to the last position. Repeat the process for remaining elements.

1. Insertion Sort:

Insertion Sort is a simple sorting algorithm that builds the final sorted array one item at a time.

Start from the second element. Compare it to the elements before it. Insert it into its correct position by shifting larger elements one position to the right.

1. Quick Sort:

It works by selecting a pivot element and partitioning the array such that all elements less than the pivot come before it, and all elements greater than the pivot come after it.

This process is repeated for the left partition and the right partition recursively until the array is sorted.

1. Merge Sort:

Merge sort algorithm is based on the divide and conquer approach.

Divide the array. Recursively sort both the halves. Finally, join both the halves.

**Project: SortingCustomerOrders**

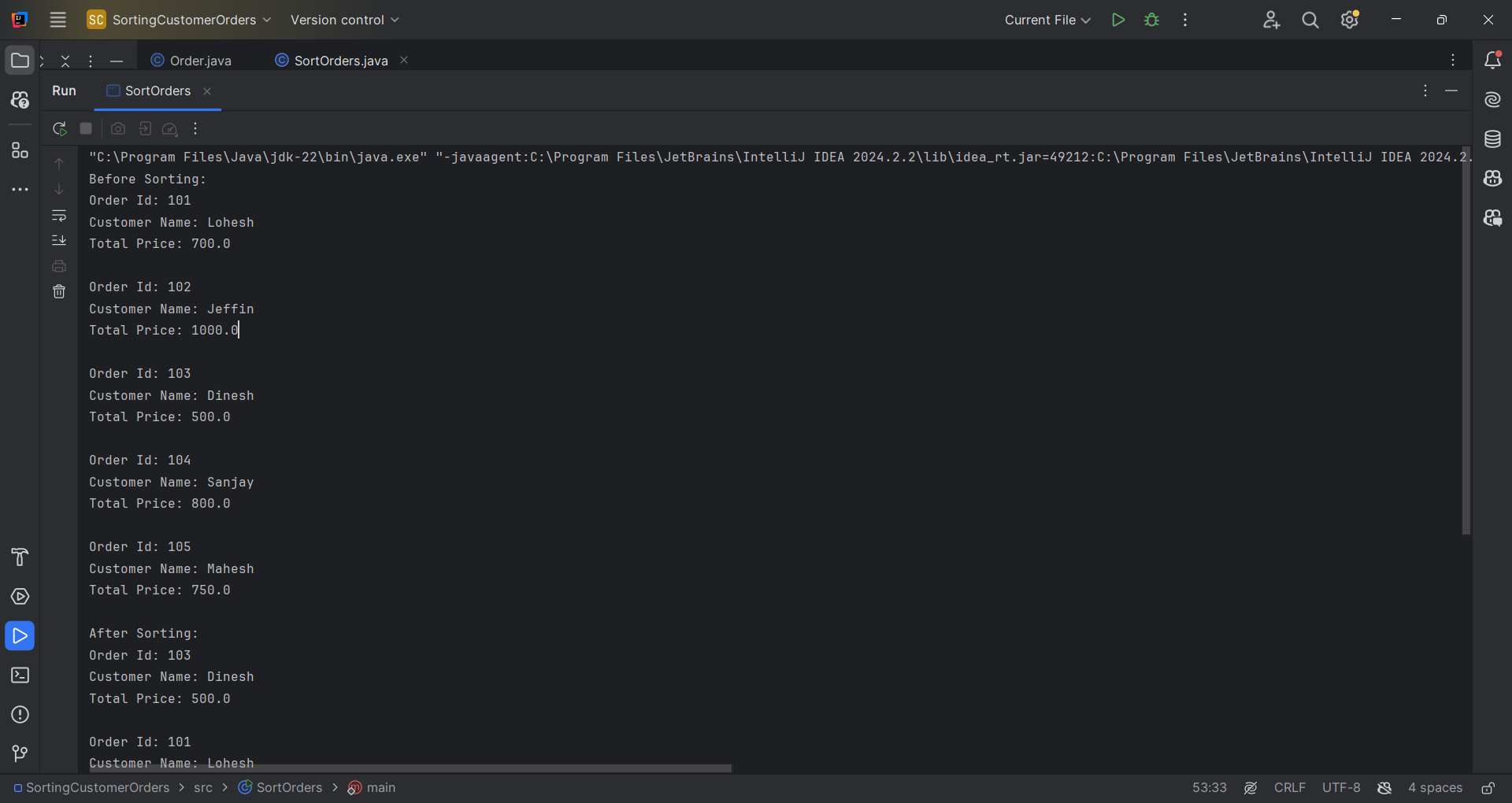
**Order.java**

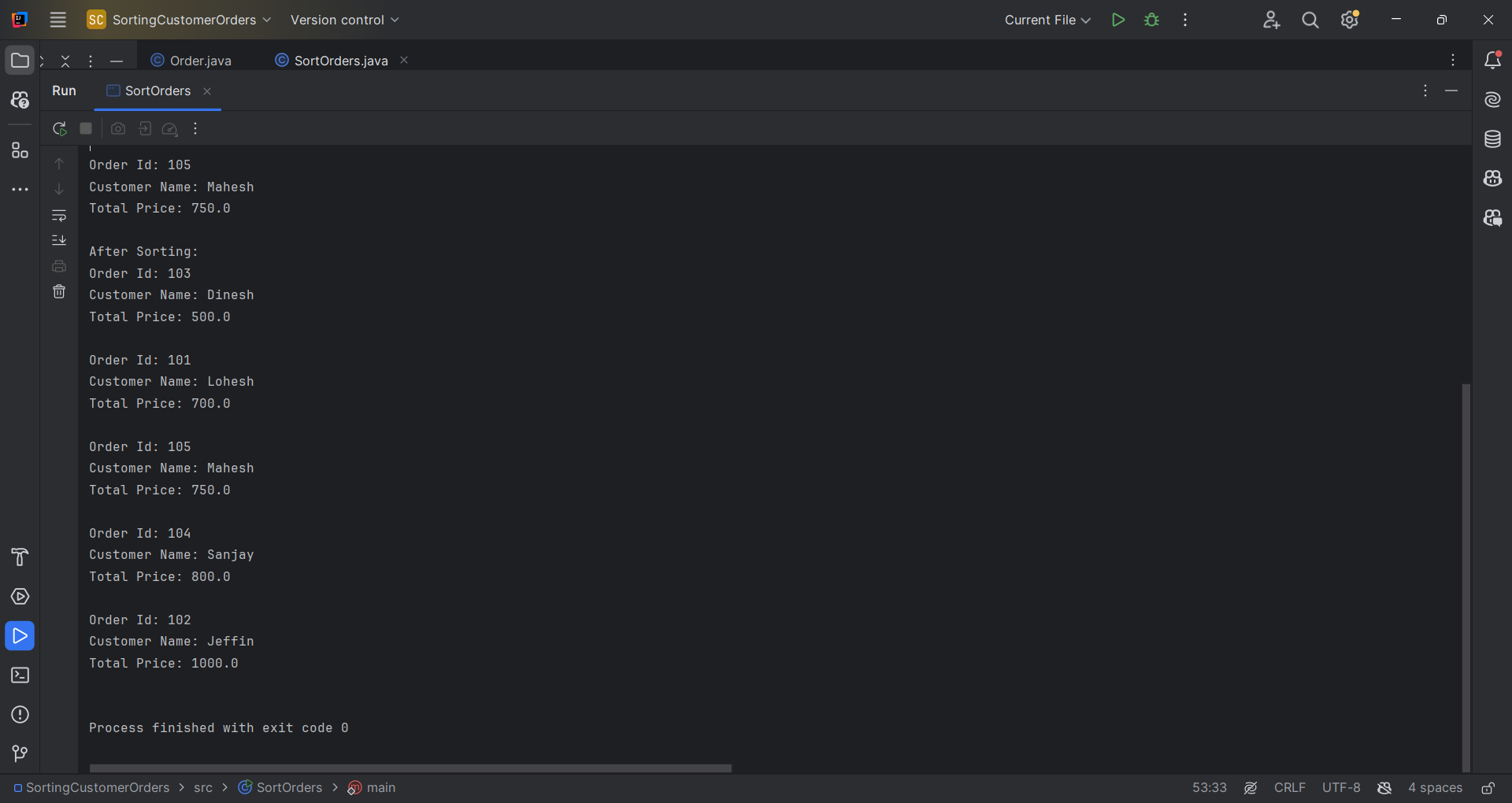
public class Order {  
 private int orderId;  
 private String customerName;  
 private double totalPrice;  
  
 public Order(int orderId, String customerName, double totalPrice) {  
 this.orderId = orderId;  
 this.customerName = customerName;  
 this.totalPrice = totalPrice;  
 }  
  
 public int getOrderId() {  
 return orderId;  
 }  
  
 public void setOrderId(int orderId) {  
 this.orderId = orderId;  
 }  
  
 public String getCustomerName() {  
 return customerName;  
 }  
  
 public void setCustomerName(String customerName) {  
 this.customerName = customerName;  
 }  
  
 public double getTotalPrice() {  
 return totalPrice;  
 }  
  
 public void setTotalPrice(double totalPrice) {  
 this.totalPrice = totalPrice;  
 }  
  
 public void printOrder(){  
 System.*out*.println("Order Id: " + orderId + "\nCustomer Name: " + customerName + "\nTotal Price: " + totalPrice + "\n");  
 }  
}

**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].getTotalPrice() > orders[j + 1].getTotalPrice()){  
 Order temp = orders[j];  
 orders[j] = orders[j + 1];  
 orders[j + 1] = temp;  
 }  
 }  
 }  
 }  
 public static void swap(Order a, Order b){  
 Order temp = a;  
 a = b;  
 b = temp;  
 }  
 public static int partition(Order[] orders, int low, int high){  
 Order pivot = orders[0];  
 int i = low;  
 int j = high;  
 while(i < j) {  
 while (orders[i].getTotalPrice() <= pivot.getTotalPrice() && i <= high) {  
 i++;  
 }  
 while (orders[j].getTotalPrice() >= pivot.getTotalPrice() && j >= low) {  
 j--;  
 }  
 if(i < j){  
 *swap*(orders[i], orders[j]);  
 }  
 }  
 *swap*(orders[low], orders[j]);  
 return j;  
 }  
 public static void quickSort(Order[] orders, int low, int high){  
 if(low < high){  
 int pIndex = *partition*(orders, low, high);  
 *quickSort*(orders, low, pIndex - 1);  
 *quickSort*(orders, pIndex + 1, high);  
 }  
 }  
  
 public static void main(String[] args){  
 Order[] orders = {  
 new Order(101, "Lohesh", 700),  
 new Order(102, "Jeffin", 1000),  
 new Order(103, "Dinesh", 500),  
 new Order(104, "Sanjay", 800),  
 new Order(105, "Mahesh", 750)  
 };  
 System.*out*.println("Before Sorting:");  
 for (Order o : orders) {  
 o.printOrder();  
 }  
 *bubbleSort*(orders);  
 System.*out*.println("After Sorting:");  
  
 for(Order o: orders){  
 o.printOrder();  
 }  
  
 }  
}

**Output:**

****

****

**Analysis:**

**Compare the performance (time complexity) of Bubble Sort and Quick Sort.**

Bubbe sort: Time complexity: Best Case: O(n log n), Average Case: O(n log n), Worst Case: O(n^2)

Quick sort: Time complexity: Best Case: O(n^2), Average Case: O(n^2), Worst Case: O(n^2)

**Discuss why Quick Sort is generally preferred over Bubble Sort.**

Quick Sort performs fewer swaps and passes through the array, making it more efficient in practice. Quick Sort also breaks the problem into smaller subproblems and solves them recursively. This makes it easier to optimize and parallelize in modern systems.

**Exercise 4: Employee Management System**

**Scenario:**

You are developing an employee management system for a company. Efficiently managing employee records is crucial.

**Explain how arrays are represented in memory and their advantages.**

An Array is stored as a contiguous block of memory. When the array is created, a single chunk of memory is allocated to hold all its elements sequentially.

Easy to declare, use and interate over arrays.

Random access in arrays allows direct retrieval of any element in O(1) Time complexity using its index.

**Project: EmployeeManagementSystem**

Employee.java

public class Employee {  
 private int employeeId;  
 private String name;  
 private String position;  
 private int salary;  
  
 public Employee(int employeeId, String name, String position, int Salary) {  
 this.employeeId = employeeId;  
 this.salary = salary;  
 this.position = position;  
 this.name = name;  
 }  
  
 public int getEmployeeId() {  
 return employeeId;  
 }  
  
 public void setEmployeeId(int employeeId) {  
 this.employeeId = employeeId;  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 public String getPosition() {  
 return position;  
 }  
  
 public void setPosition(String position) {  
 this.position = position;  
 }  
  
 public int getSalary() {  
 return salary;  
 }  
  
 public void setSalary(int salary) {  
 this.salary = salary;  
 }  
  
 public void printEmployee(){  
 System.*out*.println("Employee Id: " + employeeId + "\nName: " + name + "\nPosition: " + position + "\nSalary: " + salary + "\n");  
 }  
}

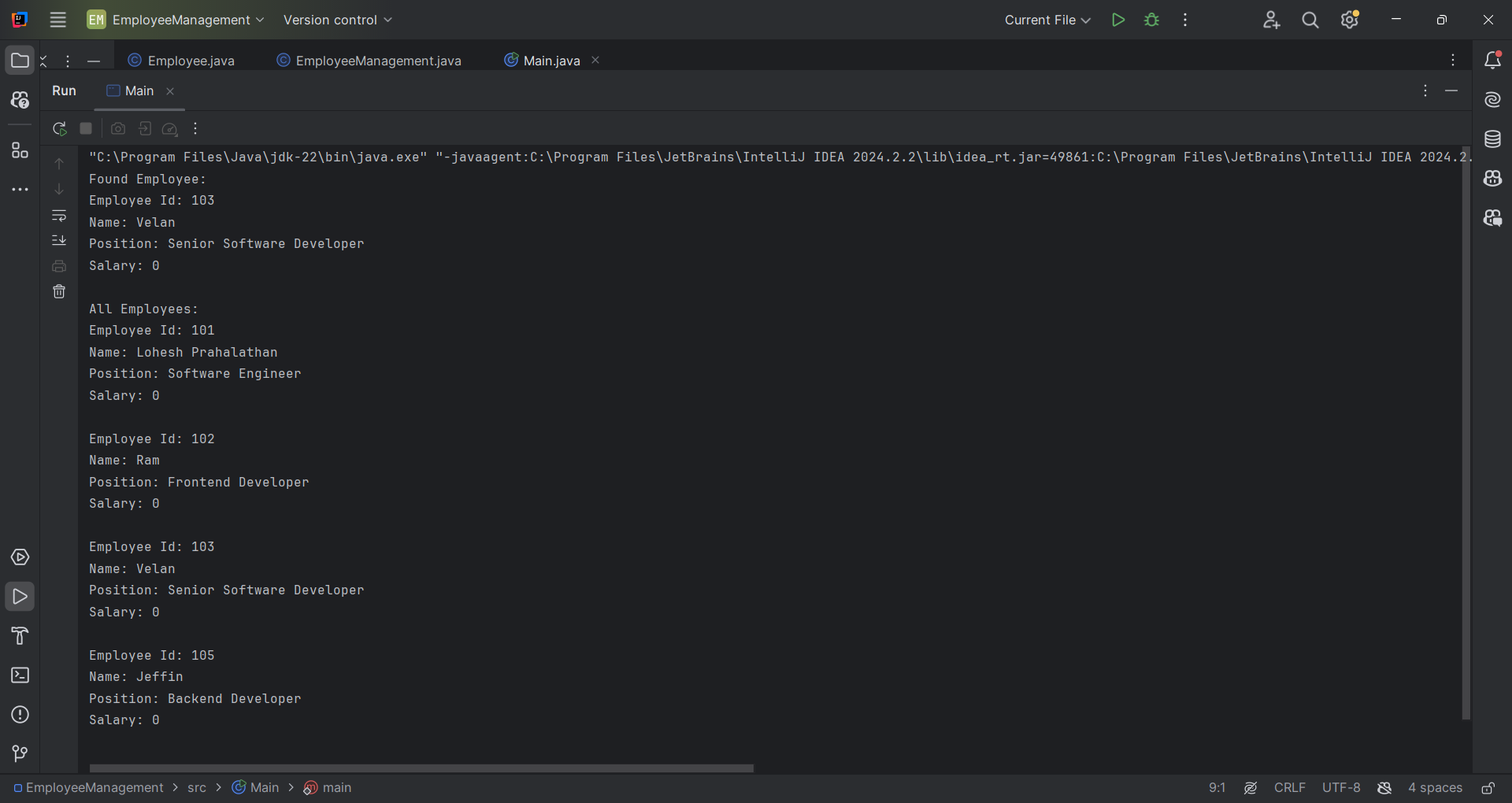
**EmployeeManagement.java**

public class EmployeeManagement {  
 private int count = 0;  
 private Employee[] employees;  
  
 public EmployeeManagement(int size) {  
 this.employees = new Employee[size];  
 }  
  
 public void addEmployee(Employee e){  
 if(count < employees.length){  
 employees[count] = e;  
 count++;  
 }else{  
 System.*out*.println("Employees array is full");  
 }  
 }  
  
 public void searchEmployee(int id){  
 for(int i = 0; i < count; i++){  
 if(employees[i].getEmployeeId() == id){  
 System.*out*.println("Found Employee: ");  
 employees[i].printEmployee();  
 }  
 }  
 }  
  
 public void displayAllEmployees(){  
 for(int i = 0; i < count; i++){  
 employees[i].printEmployee();  
 }  
 }  
  
 public void deleteEmployee(int id){  
 int index = -1;  
 for(int i = 0; i < count; i++){  
 if(employees[i].getEmployeeId() == id) {  
 index = i;  
 break;  
 }  
 }  
 if(index != -1){  
 for (int j = index; j < count - 1; j++) {  
 employees[j] = employees[j + 1];  
 }  
 count--;  
 employees[count] = null;  
 }  
 }  
}

**Main.java**

public class Main {  
 public static void main(String[] args){  
 EmployeeManagement employeeManagement = new EmployeeManagement(5);  
 employeeManagement.addEmployee(new Employee(101, "Lohesh Prahalathan", "Software Engineer", 675000));  
 employeeManagement.addEmployee(new Employee(102, "Ram", "Frontend Developer", 400000));  
 employeeManagement.addEmployee(new Employee(103, "Velan", "Senior Software Developer", 1000000));  
 employeeManagement.addEmployee(new Employee(104, "Sanjay", "Backend Developer", 500000));  
 employeeManagement.addEmployee(new Employee(105, "Jeffin", "Backend Developer", 500000));  
  
 employeeManagement.searchEmployee(103);  
  
 employeeManagement.deleteEmployee(104);  
  
 System.*out*.println("All Employees:");  
 employeeManagement.displayAllEmployees();  
 }  
}

**Output:**



**Analysis:**

**Analyze the time complexity of each operation (add, search, traverse, delete).**

1. Add: Adding elements can take O(n) time complexity, but using a variable to store the last element can help adding elements to the array in O(1) time complexity.
2. Update: Retrieving products from the array and updating them is done in O(n) time complexity.
3. Delete: Deleting elements from the array also takes O(n) time complexity.

**Discuss the limitations of arrays and when to use them.**

When using arrays, deletion of elements in between the array can take O(n) time complexity to shift all the element after the deleted element to its left.

Searching elements can take O(n) time complexity, but if the arrays is sorted, binary search can be used to optimize the search operation and it takes O(log n) time complexity.

Size of the array cannot be increased dynamically during runtime which is difficult when the size of the array is not known.

**Exercise 5: Task Management System**

**Scenario:**

You are developing a task management system where tasks need to be added, deleted, and traversed efficiently.

**Explain the different types of linked lists (Singly Linked List, Doubly Linked List).**

1. Singly Linked List:

A singly linked list is a linear data structure where each node contains its data and the pointer to the next node.

1. Doubly Linked List:

A doubly linked list is a linear data structure where each node contains its data, pointer to the next node and a pointer to the previous node.

**Project: TaskManagementSystem**

**Task.java**

public class Task {  
 private int taskId;  
 private String taskName;  
 private String status;  
  
 public Task(int taskId, String taskName, String status) {  
 this.taskId = taskId;  
 this.taskName = taskName;  
 this.status = status;  
 }  
  
 public int getTaskId() {  
 return taskId;  
 }  
  
 public void setTaskId(int taskId) {  
 this.taskId = taskId;  
 }  
  
 public String getTaskName() {  
 return taskName;  
 }  
  
 public void setTaskName(String taskName) {  
 this.taskName = taskName;  
 }  
  
 public String getStatus() {  
 return status;  
 }  
  
 public void setStatus(String status) {  
 this.status = status;  
 }  
  
 public void printTask(){  
 System.*out*.println("Task Id: " + taskId + "\nName: " + taskName + "\nStatus: " + status + "\n");  
 }  
}

**TaskManager.java**

class TaskNode{  
 Task task;  
 TaskNode next;  
 public TaskNode(Task task){  
 this.task = task;  
 this.next = null;  
 }  
  
}  
  
public class TaskManager {  
 private TaskNode head;  
  
 public void addTask(Task task){  
 TaskNode temp = new TaskNode(task);  
 if(head == null){  
 head = temp;  
 }else{  
 TaskNode curr = head;  
 while(curr.next != null){  
 curr = curr.next;  
 }  
 curr.next = temp;  
 }  
 }  
  
 public void searchTask(int id){  
 TaskNode temp = head;  
 while(temp != null){  
 if(temp.task.getTaskId() == id){  
 temp.task.printTask();  
 return;  
 }  
 temp = temp.next;  
 }  
 System.*out*.println("Task not found");  
 }  
  
 public void displayAllTasks(){  
 TaskNode temp = head;  
 System.*out*.println("All Tasks:");  
 while(temp != null){  
 temp.task.printTask();  
 temp = temp.next;  
 }  
 }  
  
 public void deleteTask(int id){  
 if(head == null){  
 System.*out*.println("Task not found\n");  
 return;  
 }  
 if(head.task.getTaskId() == id){  
 System.*out*.println("Deleted Task Successfully\n");  
 head = head.next;  
 return;  
 }  
 TaskNode temp = head;  
 while(temp.next != null){  
 if(temp.next.task.getTaskId() == id){  
 System.*out*.println("Deleted Task Successfully\n");  
 temp.next = temp.next.next;  
 return;  
 }  
 temp = temp.next;  
 }  
 System.*out*.println("Task not found\n");  
 }  
}

**Main.java**

public class Main {

public static void main(String[] args){

TaskManager manager = new TaskManager();

manager.addTask(new Task(1, "Design module", "Pending"));

manager.addTask(new Task(2, "Implement API", "In Progress"));

manager.addTask(new Task(3, "Write tests", "Pending"));

manager.displayAllTasks();

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

manager.searchTask(2);

manager.deleteTask(1);

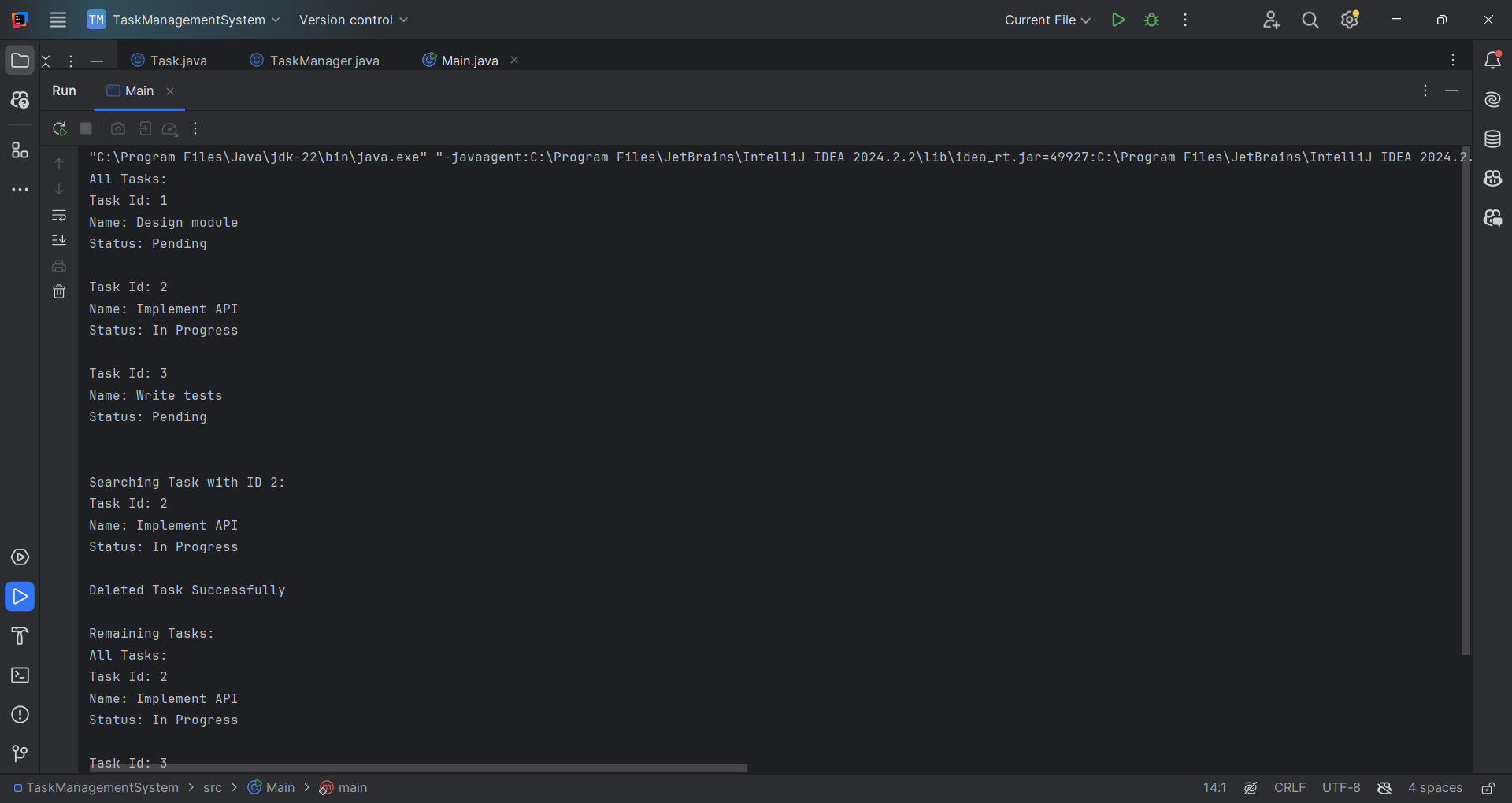
System.out.println("Remaining Tasks:");

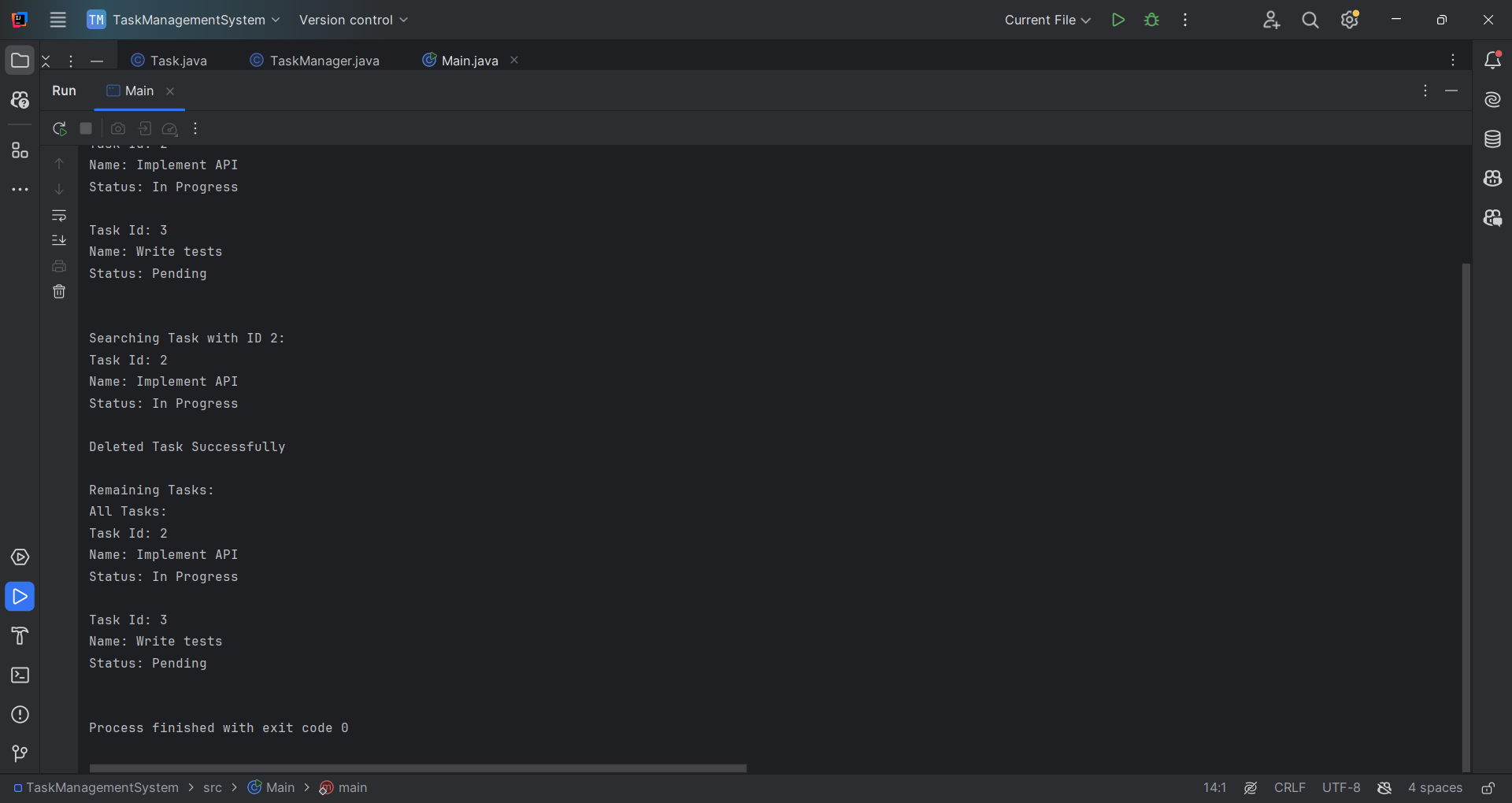
manager.displayAllTasks();

}

}

**Output:**





**Analysis:**

**Analyze the time complexity of each operation.**

1. Add: As we are using a linked list to store products, adding elements to the linked list can be done in O(n) time complexity.
2. Update: Retrieving products from the linked list and updating them is done in O(n) time complexity.
3. Delete: Deleting elements from the linked list also takes O(n) time complexity.

**Exercise 6: Library Management System**

**Scenario:**

You are developing a library management system where users can search for books by title or author.

**Explain linear search and binary search algorithms.**

1. Linear Search:

Linear search scans each element in the list one by one until the target is found or the list ends. It takes O(n) time complexity to search elements.

Start from the first element, compare each element in the list by one until the target is found or the list ends.

1. Binary Search:

Binary Search repeatedly divides a sorted array in half to find the target.

Find the middle element, if it is equal to the target return the index. If target less than middle, search left half or else search right half. Repeat until found or right less than left.

**Project: LibraryManagementSystem**

**Book.java**

public class Book {  
 private int bookId;  
 private String title;  
 private String author;  
  
 public Book(int bookId, String title, String author) {  
 this.bookId = bookId;  
 this.title = title;  
 this.author = author;  
 }  
 public String getTitle() {  
 return title.toLowerCase();  
 }  
  
 public String getAuthor() {  
 return author;  
 }  
 public int getBookId() {  
 return bookId;  
 }  
 public void setBookId(int bookId) {  
 this.bookId = bookId;  
 }  
 public void setTitle(String title) {  
 this.title = title;  
 }  
 public void setAuthor(String author) {  
 this.author = author;  
 }  
  
 public void printBook() {  
 System.*out*.println("ID: " + bookId + "\nTitle: " + title + "\nAuthor: " + author);  
 }  
}

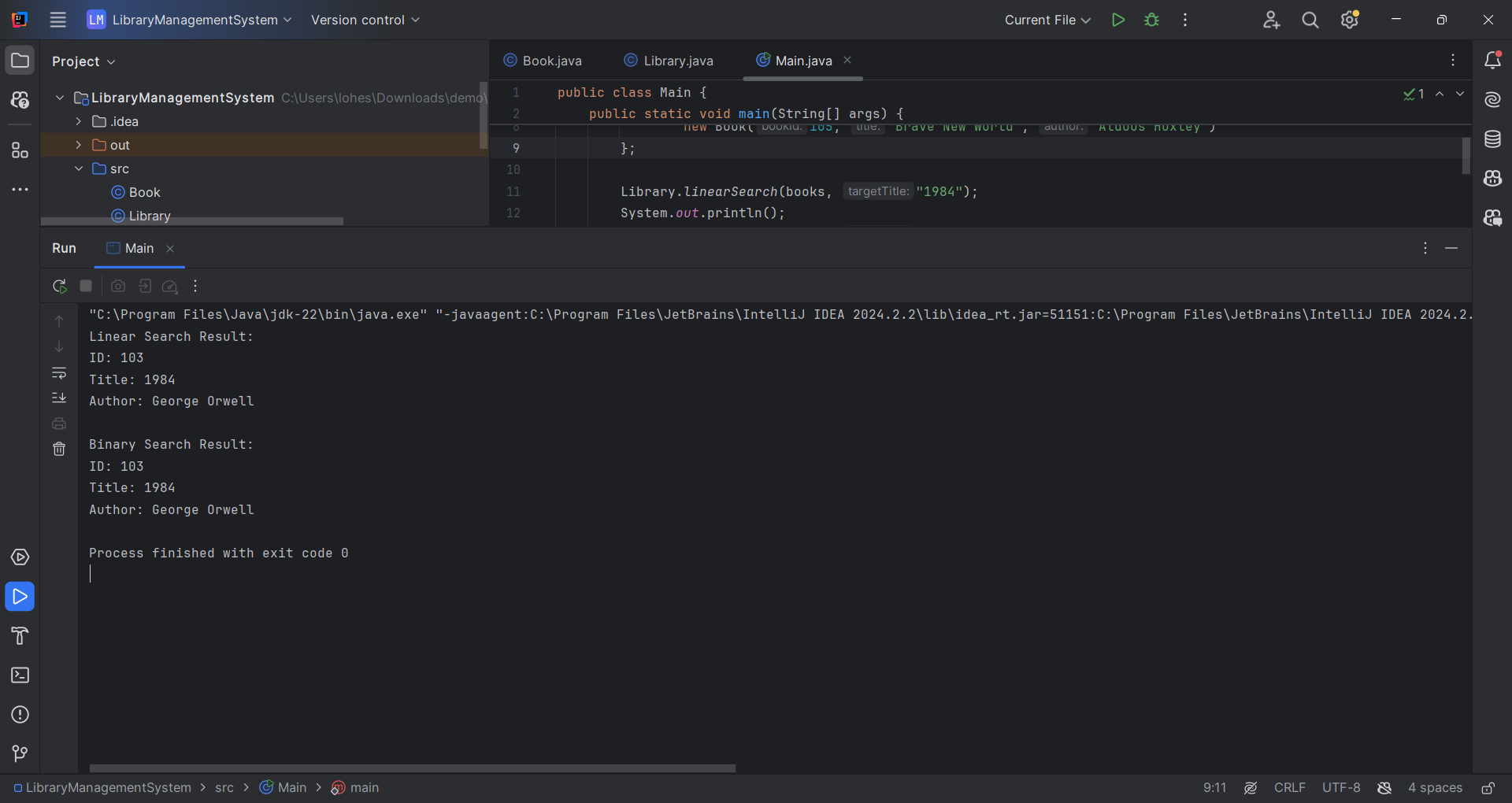
**Library.java**

import java.util.Arrays;  
import java.util.Comparator;  
  
public class Library {  
  
 public static void linearSearch(Book[] books, String targetTitle) {  
 System.*out*.println("Linear Search Result:");  
 boolean found = false;  
 for (Book book : books) {  
 if (book.getTitle().equalsIgnoreCase(targetTitle)) {  
 book.printBook();  
 found = true;  
 }  
 }  
 if (!found) System.*out*.println("Book not found");  
 }  
 public static void binarySearch(Book[] books, String targetTitle) {  
 System.*out*.println("Binary Search Result:");  
  
 Arrays.*sort*(books, Comparator.*comparing*(Book::getTitle));  
  
 int low = 0, high = books.length - 1;  
 while (low <= high) {  
 int mid = (low + high) / 2;  
 int compare = targetTitle.compareToIgnoreCase(books[mid].getTitle());  
 if (compare == 0) {  
 books[mid].printBook();  
 return;  
 } else if (compare < 0) {  
 high = mid - 1;  
 } else {  
 low = mid + 1;  
 }  
 }  
 System.*out*.println("Book not found");  
 }  
}

**Main.java**

public class Main {  
 public static void main(String[] args) {  
 Book[] books = {  
 new Book(101, "The Alchemist", "Paulo Coelho"),  
 new Book(102, "Clean Code", "Robert C. Martin"),  
 new Book(103, "1984", "George Orwell"),  
 new Book(104, "To Kill a Mockingbird", "Harper Lee"),  
 new Book(105, "Brave New World", "Aldous Huxley")  
 };  
  
 Library.*linearSearch*(books, "1984");  
 System.*out*.println();  
 Library.*binarySearch*(books, "1984");  
 }  
}

Output:



**Analysis:**

**Compare the time complexity of linear and binary search.**

1. Linear Search: Best Case – O(1), Worst Case – O(n)
2. Binary Search: Best Case – O(1), Worst Case – O(log n)

**Discuss when to use each algorithm based on the data set size and order.**

Use Linear search when the list is small or not sorted.

Use Binary search when working with large and sorted data sets.

**Exercise 7: Financial Forecasting**

**Scenario:**

You are developing a financial forecasting tool that predicts future values based on past data.

**Explain the concept of recursion and how it can simplify certain problems.**

Recursion is a programming technique where a method calls itself to solve a problem.

For example, to print n natural numbers using recursion:

public static void printNumbers(int n) {

if (n == 0) return;

printNumbers(n - 1);

System.out.println(n);

}

The method keeps calling itself with a smaller value of n until n becomes 0. This repeated self-calling process is known as recursion.

**Project: FinancialForecasting**

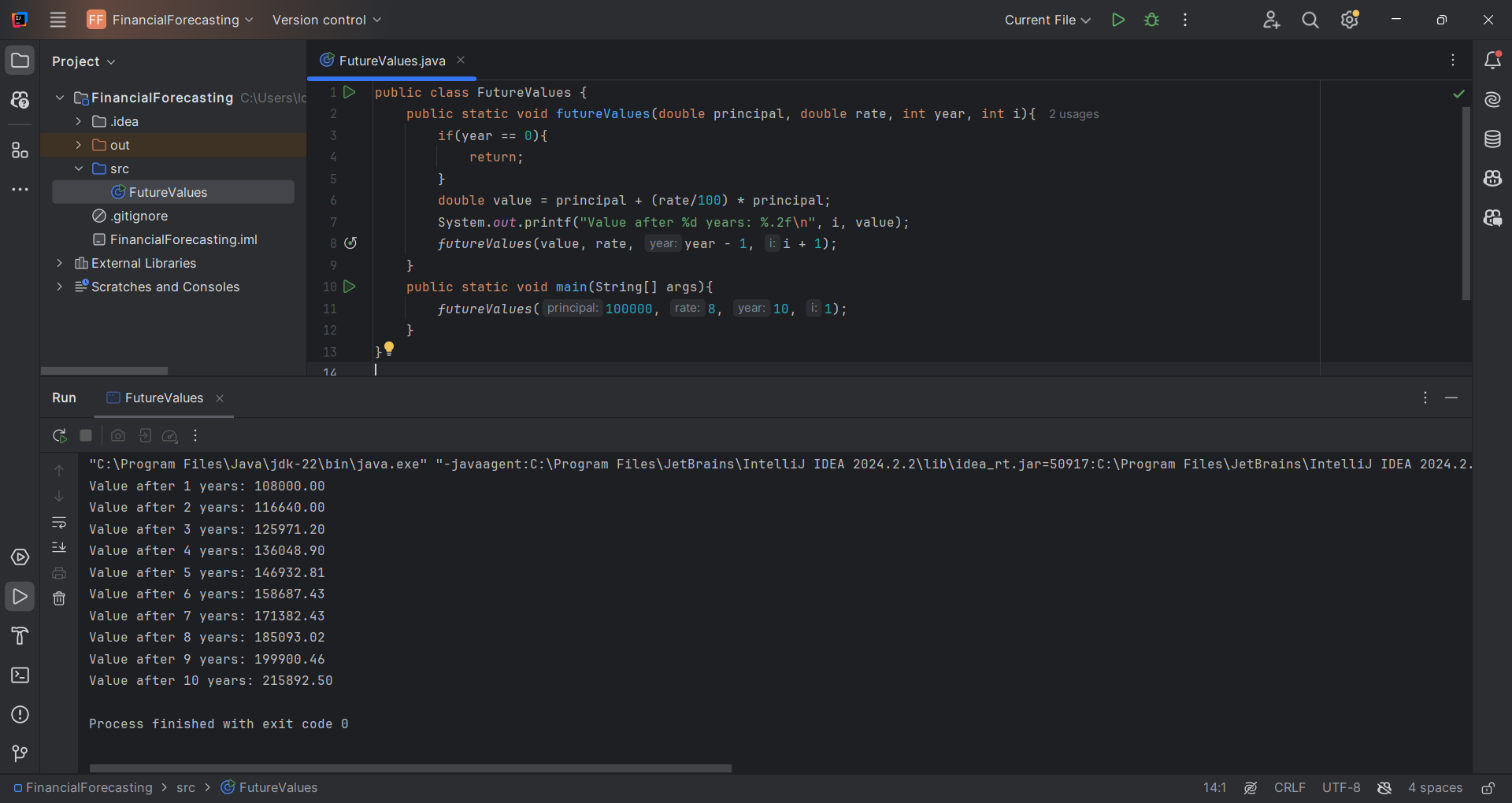
**FutureValues.java**

public class FutureValues {  
 public static void futureValues(double principal, double rate, int year, int i){  
 if(year == 0){  
 return;  
 }  
 double value = principal + (rate/100) \* principal;  
 System.*out*.printf("Value after %d years: %.2f\n", i, value);  
 *futureValues*(value, rate, year - 1, i + 1);  
 }  
 public static void main(String[] args){  
 *futureValues*(100000, 8, 10, 1);  
 }  
}

**Explanation:**

The futureValues() method is executed recursively until the remaining years become 0.  
Each time the method is called recursively, it prints the value of the principal amount after the corresponding year..

Output:



**Analysis:**

**Discuss the time complexity of your recursive algorithm.**

Time Complexity: O(n), where n is the number of years.

**Explain how to optimize the recursive solution to avoid excessive computation.**

An iterative loop does the same work without the overhead of function calls or stack usage

Optimize recursion using memoization by storing results of already computed subproblems in a cache. This problem doesn’t require memoization as it doesn’t require already computed values.