**Exercise 1: Inventory Management System**

**Scenario:**

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

**Steps:**

1. **Understand the Problem:**
   * Explain why data structures and algorithms are essential in handling large inventories.
   * Discuss the types of data structures suitable for this problem.
2. **Setup:**
   * Create a new project for the inventory management system.
3. **Implementation:**
   * Define a class Product with attributes like **productId**, **productName**, **quantity**, and **price**.
   * Choose an appropriate data structure to store the products (e.g., ArrayList, HashMap).
   * Implement methods to add, update, and delete products from the inventory.
4. **Analysis:**
   * Analyze the time complexity of each operation (add, update, delete) in your chosen data structure.
   * Discuss how you can optimize these operations.

In an inventory system with potentially thousands of products, efficient searching—such as finding a product by its ID—is critical to ensure quick access to information. Additionally, real-time performance is essential for operations like adding, removing, or modifying products to keep the system responsive and reliable. Optimal memory usage also plays a vital role, allowing the system to handle large volumes of data efficiently without compromising speed or performance under heavy load.

Using HashMap:

Product.java

**package** InventoryManagement;

**public** **class** Product {

**private** String productId;

**private** String productName;

**private** **int** quality;

**private** **double** price;

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

**this**.productId=productId;

**this**.productName=productName;

**this**.quality=quality;

**this**.price=price;

}

**public** String getProductId() {

**return** productId;

}

**public** String getProductName() {

**return** productName;

}

**public** **int** getQuality() {

**return** quality;

}

**public** **double** getPrice() {

**return** price;

}

**public** **void** setQuality(**int** quality) {

**this**.quality=quality;

}

**public** **void** setPrice(**double** price) {

**this**.price=price;

}

**public** String toString() {

**return** "Product id: "+productId+

", Name: "+productName+

",Quality: "+quality+

",Price: Rs"+price;

}

}

InventoryManagement.java

**package** InventoryManagement;

**import** java.util.HashMap;

**public** **class** InventoryManger {

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

**public** **void** addProduct(Product product) {

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

System.***out***.println("Product added: "+product.getProductId());

}

**public** **void** updateProduct(String productId,**int** quality,**double** price) {

**if**(inventory.containsKey(productId)) {

Product product=inventory.get(productId);

product.setQuality(quality);

product.setPrice(price);

System.***out***.println("Product updates: "+productId);

}

**else** {

System.***out***.println("Product not found: "+productId);

}

}

**public** **void** deleteProduct(String productId) {

**if**(inventory.remove(productId)!=**null**) {

System.***out***.println("Product deleted: "+productId);

}

**else** {

System.***out***.println("Product not found: "+productId);

}

}

**public** **void** showInventory() {

System.***out***.println("\n Current Inventory:");

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

System.***out***.println(p);

}

}

}

Test.java

**package** InventoryManagement;

**public** **class** Test {

**public** **static** **void** main(String []args) {

InventoryManger manager = **new** InventoryManger();

Product p1 = **new** Product("P001", "Keyboard", 100, 750.00);

Product p2 = **new** Product("P002", "Mouse", 150, 350.00);

Product p3 = **new** Product("P003", "Monitor", 50, 8500.00);

manager.addProduct(p1);

manager.addProduct(p2);

manager.addProduct(p3);

manager.showInventory();

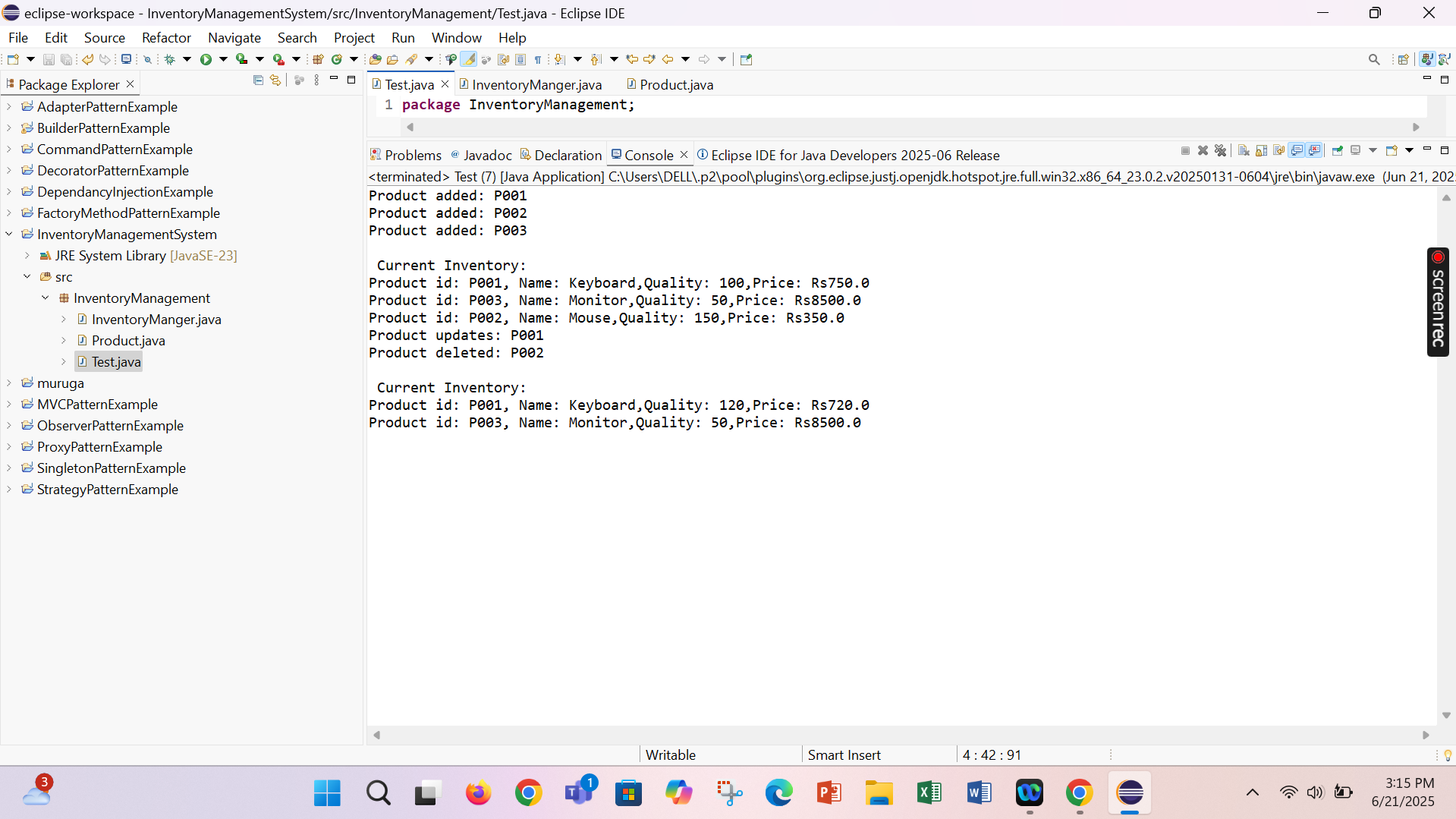
manager.updateProduct("P001", 120, 720.00);

manager.deleteProduct("P002");

manager.showInventory();

}

}



**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.

**Steps:**

1. **Understand Asymptotic Notation:**
   * Explain Big O notation and how it helps in analyzing algorithms.
   * Describe the best, average, and worst-case scenarios for search operations.
2. **Setup:**
   * Create a class **Product** with attributes for searching, such as **productId, productName**, and **category**.
3. **Implementation:**
   * Implement linear search and binary search algorithms.
   * Store products in an array for linear search and a sorted array for binary search.
4. **Analysis:**
   * Compare the time complexity of linear and binary search algorithms.
   * Discuss which algorithm is more suitable for your platform and why.

Big O notation describes the **upper bound** of an algorithm's running time in terms of input size n. It helps analyze **scalability** and **efficiency** by ignoring constant factors and focusing on growth rates.

Product.java

**public** **class** Product {

**int** ProductId;

String ProductName;

String ProductCategory;

**public** Product(**int** ProductId,String ProductName,String ProductCategory) {

**this**.ProductId=ProductId;

**this**.ProductName=ProductName;

**this**.ProductCategory=ProductCategory;

}

**public** String toString() {

**return** ProductId+": "+ProductName+"( "+ProductCategory+")";

}

}

LinearSearch.java(O(n)):

**public** **class** LinearSearch {

**public** **static** Product linear(Product[] products,**int** targetId) {

**for**(Product p:products) {

**if**(p.ProductId==targetId) {

**return** p;

}

}

**return** **null**;

}

}

BinarySearch.java(O(log n)):

**import** java.util.Arrays;

**import** java.util.Comparator;

**public** **class** binarysearch {

**public** **static** Product binary(Product[] products,**int** targetId) {

**int** left=0,right=products.length-1;

**while**(left<=right) {

**int** mid=left+(right-left)/2;

**if**(products[mid].ProductId==targetId) {

**return** products[mid];

}

**else** **if**(products[mid].ProductId<targetId) {

left=mid+1;

}

**else** {

right=mid-1;

}

}

**return** **null**;

}

**public** **static** **void** sort(Product[] products) {

Arrays.*sort*(products,Comparator.*comparingInt*(p->p.ProductId));

}

}

Main.java

**public** **class** Main {

**public** **static** **void** main(String[] args) {

Product[] products = {

**new** Product(105, "Shoes", "Footwear"),

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

**new** Product(110, "Watch", "Accessories"),

**new** Product(103, "Smartphone", "Electronics")

};

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

Product found = LinearSearch.*linear*(products, 103);

System.***out***.println(found != **null** ? found : "Product not found");

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

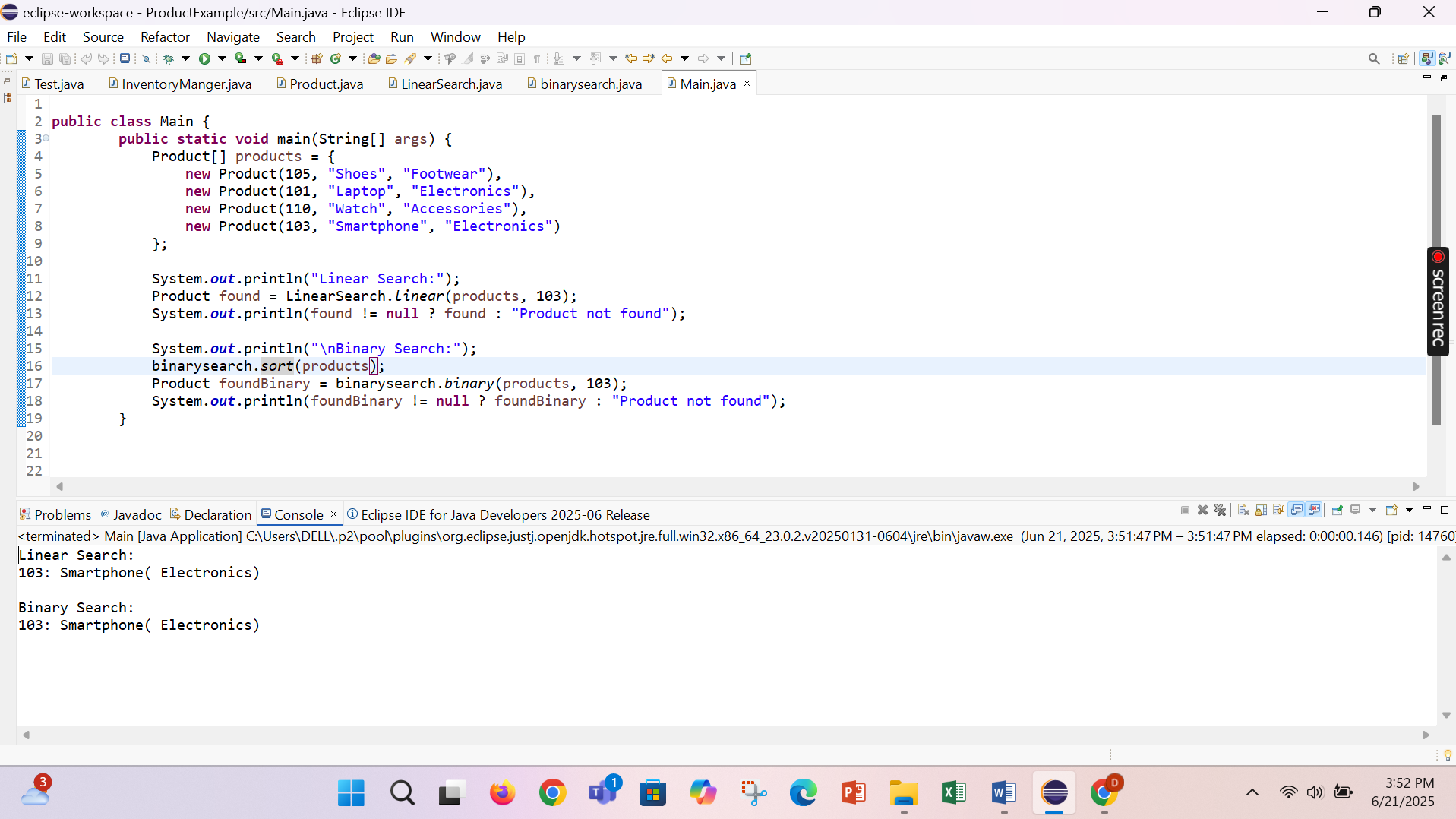
binarysearch.*sort*(products);

Product foundBinary = binarysearch.*binary*(products, 103);

System.***out***.println(foundBinary != **null** ? foundBinary : "Product not found");

}

}



For an **e-commerce platform**, where products may number in **thousands or millions**, **binary search** is more suitable because:

* It significantly reduces lookup time.
* With sorted product lists or binary search trees, it scales efficiently.
* Search performance is critical for user experience.

**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.

**Steps:**

1. **Understand Sorting Algorithms:**
   * Explain different sorting algorithms (Bubble Sort, Insertion Sort, Quick Sort, Merge Sort).
2. **Setup:**
   * Create a class **Order** with attributes like **orderId**, **customerName**, and **totalPrice**.
3. **Implementation:**
   * Implement **Bubble Sort** to sort orders by **totalPrice**.
   * Implement **Quick Sort** to sort orders by **totalPrice**.
4. **Analysis:**
   * Compare the performance (time complexity) of Bubble Sort and Quick Sort.
   * Discuss why Quick Sort is generally preferred over Bubble Sort.
5. **Bubble Sort**

* Compares adjacent elements and swaps them if they’re in the wrong order.
* Repeats until the array is sorted.
* **Time Complexity:**
  + Best: O(n) (if already sorted with an optimized check)
  + Average/Worst: O(n²)

1. **Insertion Sort**

* Builds the sorted array one element at a time by inserting the next element into its correct position.
* **Time Complexity:**
  + Best: O(n) (if nearly sorted)
  + Average/Worst: O(n²)

1. **Quick Sort**

* Picks a pivot, partitions the array so elements < pivot are on the left, > pivot on the right, then sorts sub-arrays recursively.
* **Time Complexity:**
  + Best/Average: O(n log n)
  + Worst: O(n²) (rare, if pivot is always min/max element)

1. **Merge Sort**

* Divides array into halves, sorts each half, then merges sorted halves.
* **Time Complexity:** O(n log n) in all cases
* Needs extra space for merging.

Order.java

**package** order;

**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+" - Rs. "+totalPrice;

}

}

Bubblesort.java

**package** order;

**public** **class** Bubblesort {

**public** **static** **void** bubblesort(Order[] orders) {

**int** n=orders.length;

**boolean** swapped;

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

swapped=**false**;

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

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

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

swapped=**true**;

}

}

**if**(!swapped)**break**;

}

}

}

Quicksort.java

**package** order;

**public** **class** quicksort {

**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;

}

}

Mains.java

**package** order;

**public** **class** Mains {

**public** **static** **void** main(String[] args) {

Order[] orders = {

**new** Order(201, "Alice", 250.75),

**new** Order(202, "Bob", 99.50),

**new** Order(203, "Charlie", 450.00),

**new** Order(204, "Daisy", 300.25)

};

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

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

System.***out***.println("\nBubble Sort (Descending by Price):");

Bubblesort.*bubblesort*(orders);

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

orders = **new** Order[]{

**new** Order(201, "Alice", 250.75),

**new** Order(202, "Bob", 99.50),

**new** Order(203, "Charlie", 450.00),

**new** Order(204, "Daisy", 300.25)

};

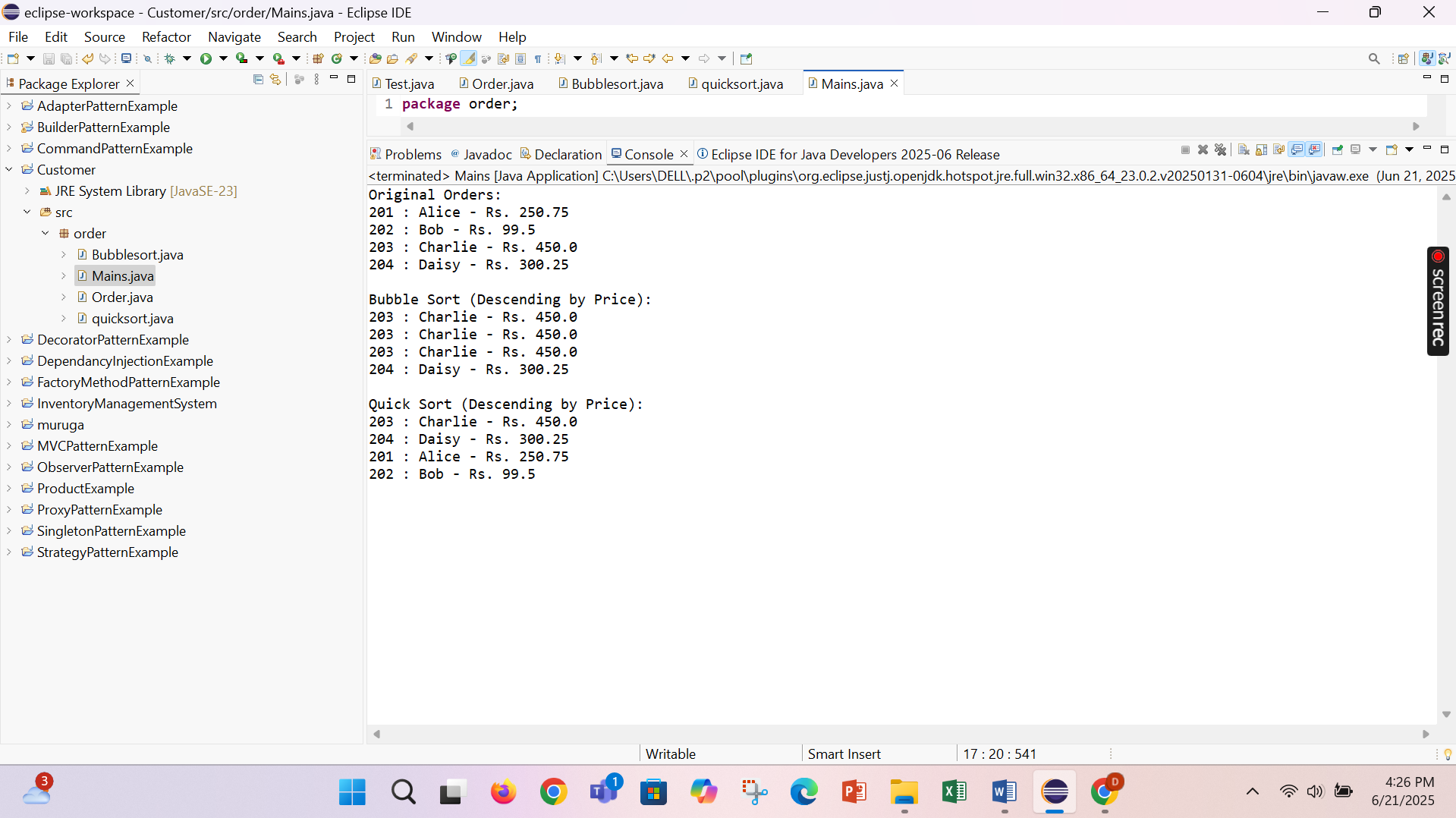
System.***out***.println("\nQuick Sort (Descending by Price):");

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

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

}

}



Quick sort is preferred because,

* Quick Sort is much faster on average because it eliminates unnecessary comparisons through partitioning.
* Bubble Sort repeatedly compares every adjacent pair, leading to excessive swaps and inefficiency on large datasets.
* Quick Sort is the standard for high-performance sorting in real-world systems.

**Exercise 4: Employee Management System**

**Scenario:**

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

**Steps:**

1. **Understand Array Representation:**
   * Explain how arrays are represented in memory and their advantages.
2. **Setup:**
   * Create a class Employee with attributes like **employeeId**, **name**, **position**, and **salary**.
3. **Implementation:**
   * Use an array to store employee records.
   * Implement methods to **add**, **search**, **traverse**, and **delete** employees in the array.
4. **Analysis:**
   * Analyze the time complexity of each operation (add, search, traverse, delete).
   * Discuss the limitations of arrays and when to use them.

 Arrays in Java are **contiguous blocks** of memory.

 Each element is stored at a fixed offset, allowing **direct access** via index (O(1) time).

 Data is stored in a **sequential** manner.

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 :"+EmployeeId+" ,name: "+name+" ,position: "+position+" with salary: "+salary;

}

}

EmployeeImplements.java

**public** **class** EmployeeImplements {

**private** Employee[] employees;

**private** **int** size;

**public** EmployeeImplements(**int** capacity) {

employees = **new** Employee[capacity];

size = 0;

}

**public** **void** addEmployee(Employee emp) {

**if** (size < employees.length) {

employees[size++] = emp;

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

} **else** {

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

}

}

**public** Employee Search(**int** id) {

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

**if** (employees[i].EmployeeId == id) {

**return** employees[i];

}

}

**return** **null**;

}

**public** **boolean** Delete(**int** id) {

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

**if** (employees[i].EmployeeId == id) {

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

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

}

employees[--size] = **null**;

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

**return** **true**;

}

}

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

**return** **false**;

}

**public** **void** display() {

**if** (size == 0) {

System.***out***.println("No employees to display.");

} **else** {

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

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

}

}

}

}

Mains.java

**public** **class** mains {

**public** **static** **void** main(String []args) {

EmployeeImplements ems=**new** EmployeeImplements(5);

ems.addEmployee(**new** Employee(1, "Alice", "Manager", 70000));

ems.addEmployee(**new** Employee(2, "Bob", "Developer", 60000));

ems.addEmployee(**new** Employee(3, "Charlie", "Designer", 50000));

System.***out***.println("\nAll Employees:");

ems.display();

System.***out***.println("\nSearching for Employee with ID 2:");

Employee e = ems.Search(2);

System.***out***.println(e != **null** ? e : "Not found");

System.***out***.println("\nDeleting Employee with ID 1:");

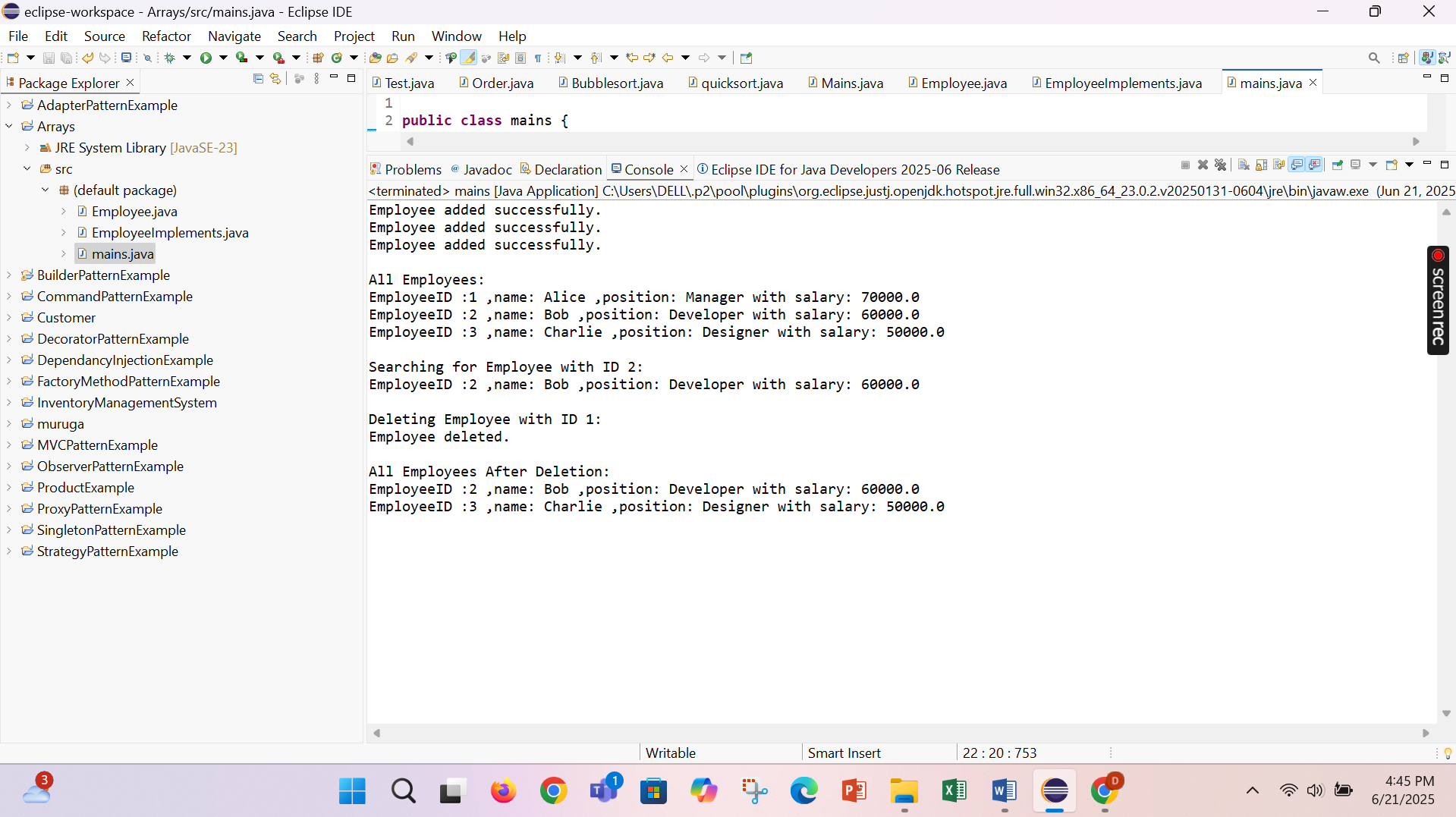
ems.Delete(1);

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

ems.display();

}

}



**Exercise 5: Task Management System**

**Scenario:**

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

**Steps:**

1. **Understand Linked Lists:**
   * Explain the different types of linked lists (Singly Linked List, Doubly Linked List).
2. **Setup:**
   * Create a class **Task** with attributes like **taskId**, **taskName**, and **status**.
3. **Implementation:**
   * Implement a singly linked list to manage tasks.
   * Implement methods to **add**, **search**, **traverse**, and **delete** tasks in the linked list.
4. **Analysis:**
   * Analyze the time complexity of each operation.
   * Discuss the advantages of linked lists over arrays for dynamic data.

|  |  |  |
| --- | --- | --- |
| Type | Description | Characteristics |
| Singly Linked List | Each node points to the next | One-directional, simple structure |
| Doubly Linked List | Each node points to both next and previous | Allows bidirectional traversal |

Task.java

**package** task;

**public** **class** Task {

**int** taskId;

String taskName;

String status;

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

**this**.taskId = taskId;

**this**.taskName = taskName;

**this**.status = status;

}

@Override

**public** String toString() {

**return** taskId + ": " + taskName + " [" + status + "]";

}

}

Node.java

**package** task;

**class** Node {

Task task;

Node next;

**public** Node(Task task) {

**this**.task = task;

**this**.next = **null**;

}

}

**class** TaskManager {

**private** Node head;

**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;

}

System.***out***.println("Task added: " + task.taskName);

}

**public** Task searchTask(**int** taskId) {

Node temp = head;

**while** (temp != **null**) {

**if** (temp.task.taskId == taskId) {

**return** temp.task;

}

temp = temp.next;

}

**return** **null**;

}

**public** **void** displayTasks() {

Node temp = head;

**if** (temp == **null**) {

System.***out***.println("No tasks available.");

**return**;

}

**while** (temp != **null**) {

System.***out***.println(temp.task);

temp = temp.next;

}

}

**public** **boolean** deleteTask(**int** taskId) {

**if** (head == **null**) **return** **false**;

**if** (head.task.taskId == taskId) {

head = head.next;

**return** **true**;

}

Node prev = head;

Node current = head.next;

**while** (current != **null**) {

**if** (current.task.taskId == taskId) {

prev.next = current.next;

**return** **true**;

}

prev = current;

current = current.next;

}

**return** **false**;

}

}

Main.java

**package** task;

**public** **class** Main {

**public** **static** **void** main(String[] args) {

TaskManager tm = **new** TaskManager();

tm.addTask(**new** Task(101, "Design Database", "Pending"));

tm.addTask(**new** Task(102, "Develop UI", "In Progress"));

tm.addTask(**new** Task(103, "Write Documentation", "Pending"));

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

tm.displayTasks();

System.***out***.println("\nSearching for Task ID 102:");

Task t = tm.searchTask(102);

System.***out***.println(t != **null** ? t : "Task not found");

System.***out***.println("\nDeleting Task ID 101:");

**boolean** deleted = tm.deleteTask(101);

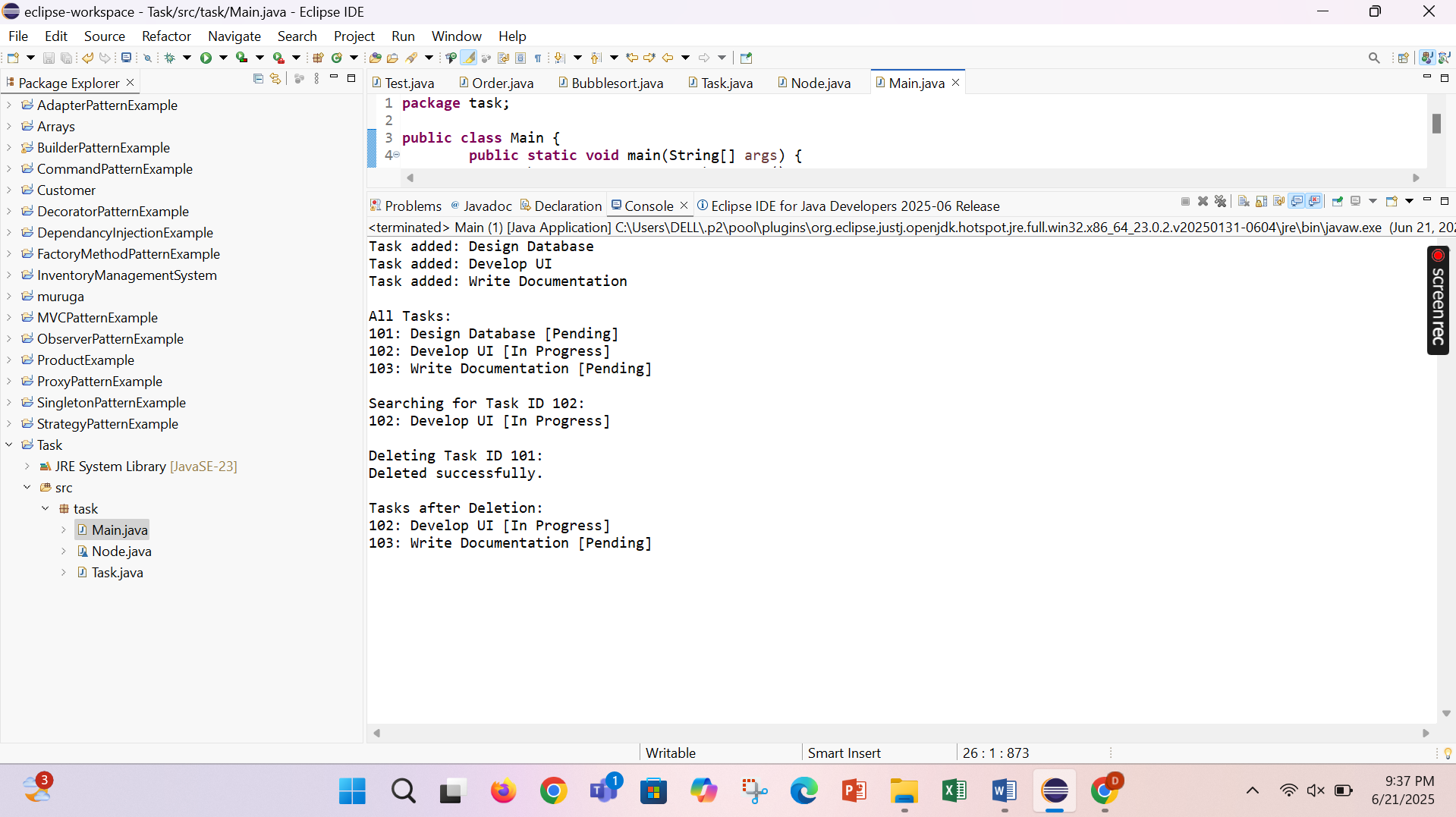
System.***out***.println(deleted ? "Deleted successfully." : "Task not found.");

System.***out***.println("\nTasks after Deletion:");

tm.displayTasks();

}

}



**Exercise 6: Library Management System**

**Scenario:**

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

**Steps:**

1. **Understand Search Algorithms:**
   * Explain linear search and binary search algorithms.
2. **Setup:**
   * Create a class **Book** with attributes like **bookId**, **title**, and **author**.
3. **Implementation:**
   * Implement linear search to find books by title.
   * Implement binary search to find books by title (assuming the list is sorted).
4. **Analysis:**
   * Compare the time complexity of linear and binary search.
   * Discuss when to use each algorithm based on the data set size and order.

Book.java

**public** **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** bookId + ": \"" + title + "\" by " + author;

}

}

LibrarySearch.java

**import** java.util.Arrays;

**import** java.util.Comparator;

**public** **class** LibrarySearch {

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

**for** (Book b : books) {

**if** (b.title.equalsIgnoreCase(title)) {

**return** b;

}

}

**return** **null**;

}

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

**int** left = 0;

**int** right = books.length - 1;

**while** (left <= right) {

**int** mid = left + (right - left) / 2;

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

**if** (comparison == 0) {

**return** books[mid];

} **else** **if** (comparison < 0) {

left = mid + 1;

} **else** {

right = mid - 1;

}

}

**return** **null**;

}

**public** **static** **void** sortBooksByTitle(Book[] books) {

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

}

}

Main.java

**public** **class** Main {

**public** **static** **void** main(String[] args) {

Book[] books = {

**new** Book(101, "Data Structures", "Mark Allen Weiss"),

**new** Book(102, "Operating Systems", "Silberschatz"),

**new** Book(103, "Clean Code", "Robert C. Martin"),

**new** Book(104, "Introduction to Algorithms", "CLRS")

};

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

Book result1 = LibrarySearch.*linearSearchByTitle*(books, "Clean Code");

System.***out***.println(result1 != **null** ? result1 : "Book not found");

System.***out***.println("\n Binary Search for 'Operating Systems':");

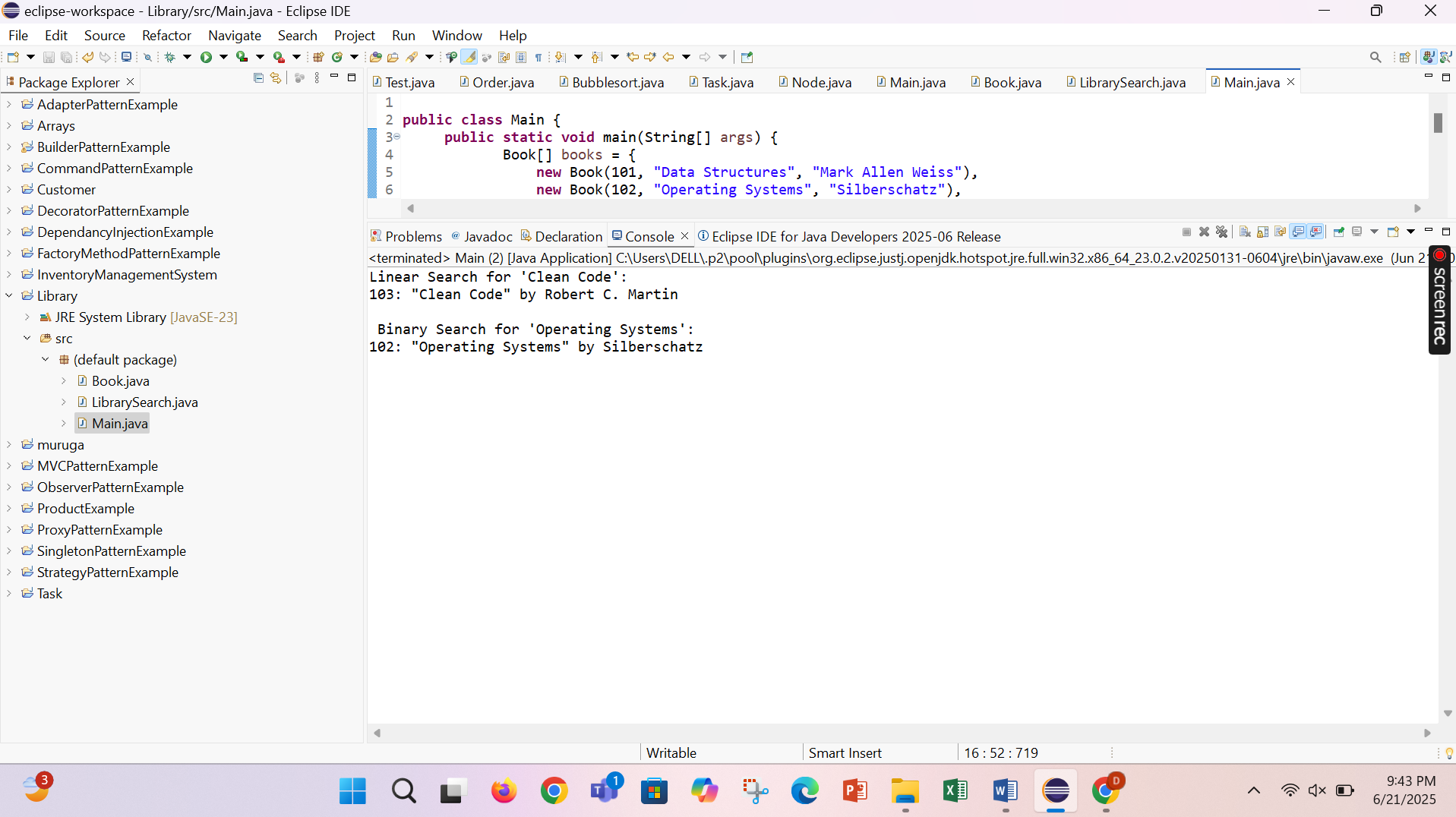
LibrarySearch.*sortBooksByTitle*(books);

Book result2 = LibrarySearch.*binarySearchByTitle*(books, "Operating Systems");

System.***out***.println(result2 != **null** ? result2 : "Book not found");

}

}



**Exercise 7: Financial Forecasting**

**Scenario:**

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

**Steps:**

1. **Understand Recursive Algorithms:**
   * Explain the concept of recursion and how it can simplify certain problems.
2. **Setup:**
   * Create a method to calculate the future value using a recursive approach.
3. **Implementation:**
   * Implement a recursive algorithm to predict future values based on past growth rates.
4. **Analysis:**
   * Discuss the time complexity of your recursive algorithm.
   * Explain how to optimize the recursive solution to avoid excessive computation.

**package** recursion;

**public** **class** FinalForecast {

**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 = 0.08;

**int** years = 5;

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

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

}

}

