**Exercise 1: Inventory Management System**

**1.Importance of DSA**

* **Warehouses deal with thousands of items. You need to store, search, update, and delete items quickly.**
* **Choosing the right data structure ensures:**
  + **Fast product search**
  + **Easy updates of quantity/price**
  + **Efficient deletion and addition of products**

**2.Suitable Data Structures:**

* **ArrayList: Good for small, ordered lists but slow when searching or deleting.**
* **HashMap: Best for quick access using keys (e.g., productId as key).**
* **TreeMap: Maintains order and allows fast retrieval based on sorted keys.**

**3.This is my code:**

**package com.example.inventory;**

**public class Product {**

**int productId;**

**String productname;**

**int quantity;**

**double price;**

**public Product(int productId,String productname,int quantity,double price)**

**{**

**this.productId=productId;**

**this.productname=productname;**

**this.quantity=quantity;**

**this.price=price;**

**}**

**@Override**

**public String toString()**

**{**

**return "Product ID: " + productId + ", Name: " + productname +", Quantity: " + quantity + ", Price: " + price;**

**}**

**}**

**package com.example.inventory;**

**import java.util.HashMap;**

**public class Inventory {**

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

**public void addp(Product p)**

**{**

**inventory.put(p.productId,p);**

**}**

**public void update(int productId,String name,int quantity,double price)**

**{**

**Product p=inventory.get(productId);**

**if(p!=null)**

**{**

**p.productname=name;**

**p.quantity=quantity;**

**p.price=price;**

**}else {**

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

**}**

**}**

**public void delete(int productId)**

**{**

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

**{**

**inventory.remove(productId);**

**}else {**

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

**}**

**}**

**public void display() {op**

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

**{**

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

**}**

**}**

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

**Inventory inv = new Inventory();**

**inv.addp(new Product(1, "Laptop", 10, 55000));**

**inv.addp(new Product(2, "Mouse", 50, 500));**

**inv.addp(new Product(3, "Keyboard", 30, 800));**

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

**inv.display();**

**System.*out*.println("\nUpdating Product 2:");**

**inv.update(2, "Wireless Mouse", 40, 750);**

**inv.display();**

**System.*out*.println("\nDeleting Product 1:");**

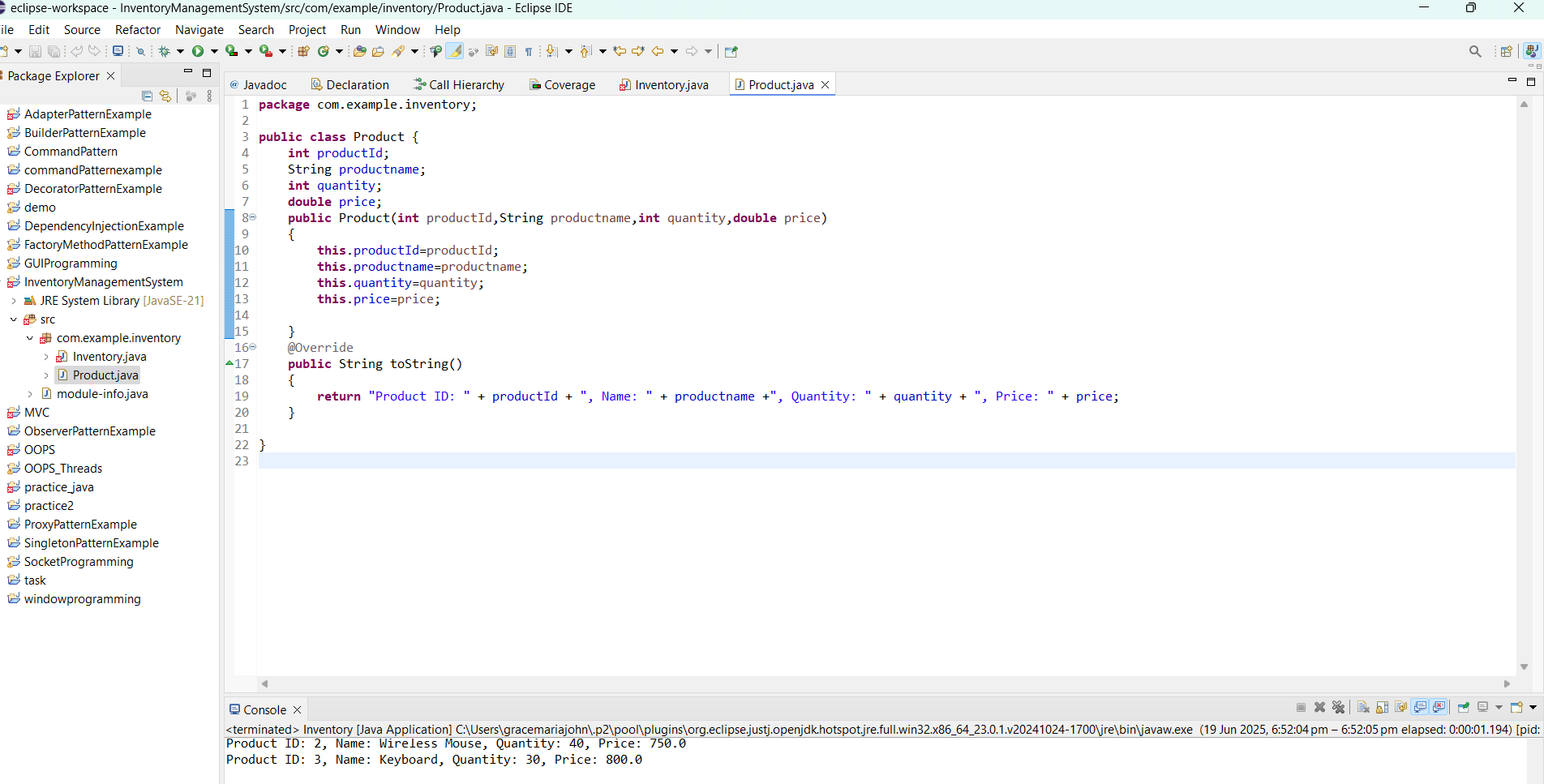
**inv.delete(1);**

**inv.display();**

**}**

**}**

**This is my Output:**

****

**4.Analysis**

**Operation:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Add Product** | | **-** | **O(1)** |  |
| **Update Product** | **-** | | **O(1)** |  |
| **Delete Product** | **-** | | **O(1)** |  |
| **Display** | **-** | | **O(n)** |  |
| **Optimization Tips:**   * **Use TreeMap if you need products sorted by ID.** * **Use ConcurrentHashMap in multi-threaded environments.** * **If you often search by name or price, consider using indexes or secondary maps.** |  | |  |  |

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

1. **Big O Notation**

**Big O notation tells us how the runtime of an algorithm increases with input size. It helps you compare algorithms and choose the most efficient one.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Constant** | | **O(1)** | |  |
| **Linear** | | **O(n)** | |  |
| **Logarithmic** | | **O(log n)** | |  |
| **Quadratic** | | **O(n²)** | |  |
| **Search Type** | **Best Case** | | **Average Case** | | | **Worst Case** |
| **Linear Search** | **O(1) – First item** | | **O(n/2) ≈ O(n)** | | | **O(n) – Last/missing item** |
| **Binary Search** | **O(1) – Middle item** | | **O(log n)** | | | **O(log n) – Few steps** |

**2.This is my code:**

**package com.example.inventory;**

**public class Product {**

**int productId;**

**String productname;**

**String category;**

**public Product(int productId,String productname,String category)**

**{**

**this.productId=productId;**

**this.productname=productname;**

**this.category=category;**

**}**

**@Override**

**public String toString() {**

**return "ID: " + productId + ", Name: " + productname + ", Category: " + category;**

**}**

**}**

**package com.example.inventory;**

**import java.util.Arrays;**

**public class search {**

**public static int linearSearch(Product[] products,String target)**

**{**

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

**{**

**if(products[i].productname.equalsIgnoreCase(target))**

**{**

**return i;**

**}**

**}**

**return -1;**

**}**

**public static int binarySearch(Product[] products,String target)**

**{**

**int l=0,h=products.length-1;**

**while(l<=h)**

**{**

**int m=(l+h)/2;**

**int c=products[m].productname.compareToIgnoreCase(target);**

**if(c==0)**

**{**

**return m;**

**}**

**else if(c<0)**

**{**

**l=m+1;**

**}else {**

**h=m-1;**

**}**

**}**

**return -1;**

**}**

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

**{**

**Product[] products={**

**new Product(1, "Laptop", "Electronics"),**

**new Product(2, "Shoes", "Fashion"),**

**new Product(3, "Phone", "Electronics"),**

**new Product(4, "Watch", "Accessories"),**

**new Product(5, "Tablet", "Electronics")**

**};**

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

**int r1=*linearSearch*(products,"Phone");**

**System.*out*.println(r1 != -1 ? products[r1] : "Product not found");**

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

**Arrays.*sort*(products, (a, b) -> a.productname.compareToIgnoreCase(b.productname));**

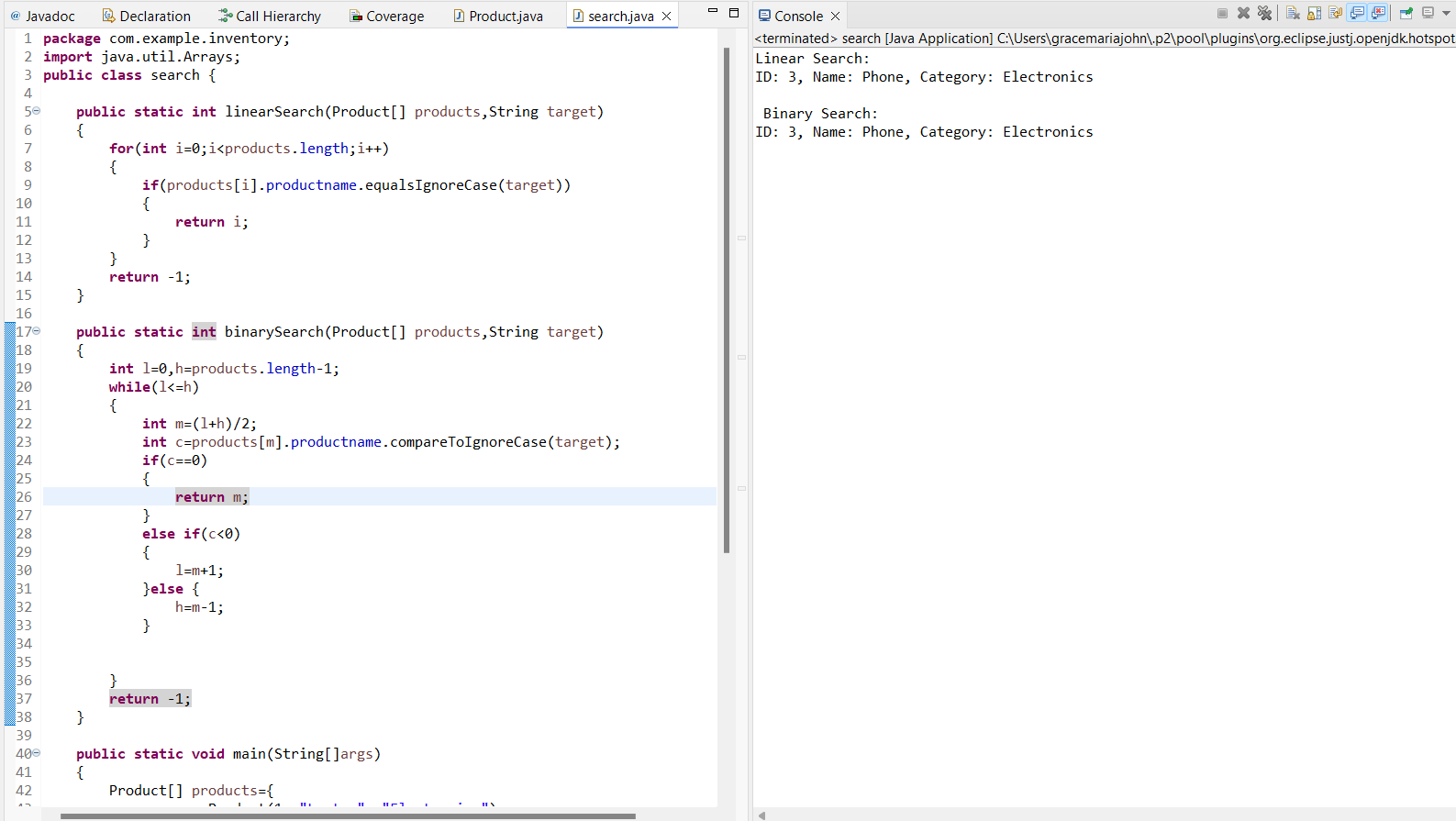
**int result2 = *binarySearch*(products, "Phone");**

**System.*out*.println(result2 != -1 ? products[result2] : "Product not found");**

**}**

**}**

**3.This is my Output:**

****

**4.Analysis**

* Binary Search is better if the array is sorted.
* Use Linear Search for small or unsorted data sets.

**Exercise 3: Sorting Customer Orders**

**1.**

| **Algorithm** | **Time Complexity** | **Best Case** | **Worst Case** |  |
| --- | --- | --- | --- | --- |
| **Bubble Sort** | **O(n²)** | **O(n)** | **O(n²)** |  |
| **Insertion Sort** | **O(n²)** | **O(n)** | **O(n²)** |  |
| **Quick Sort** | **O(n log n)** | **O(n log n)** | **O(n²)** |  |
| **Merge Sort** | **O(n log n)** | **O(n log n)** | **O(n log n)** |  |

**2.This is my code:**

**package com.example.inventory;**

**public class Product {**

**int productId;**

**String productname;**

**String category;**

**public Product(int productId,String productname,String category)**

**{**

**this.productId=productId;**

**this.productname=productname;**

**this.category=category;**

**}**

**@Override**

**public String toString() {**

**return "ID: " + productId + ", Name: " + productname + ", Category: " + category;**

**}**

**}**

**package com.example.orders;**

**public class SortOrders {**

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

**{**

**int n=orders.length;**

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

**{**

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

**{**

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

**{**

**Order t= orders[j];**

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

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

**}**

**}**

**}**

**}**

**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].tprice;**

**int i=low-1;**

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

**{**

**if(orders[j].tprice>=pivot)**

**{**

**i++;**

**Order t=orders[i];**

**orders[i]=orders[j];**

**orders[j]=t;**

**}**

**}**

**Order t=orders[i+1];**

**orders[i+1]=orders[high];**

**orders[high]=t;**

**return i+1;**

**}**

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

**for (Order o : orders) {**

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

**}**

**}**

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

**Order[] orders = {**

**new Order(101, "Alice", 4000),**

**new Order(102, "Bob", 9000),**

**new Order(103, "Charlie", 2000),**

**new Order(104, "Daisy", 7000)**

**};**

**System.*out*.println("Bubble Sort (Descending Total Price):");**

***bubblesort*(orders);**

***printOrders*(orders);**

**orders = new Order[] {**

**new Order(101, "Alice", 4000),**

**new Order(102, "Bob", 9000),**

**new Order(103, "Charlie", 2000),**

**new Order(104, "Daisy", 7000)**

**};**

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

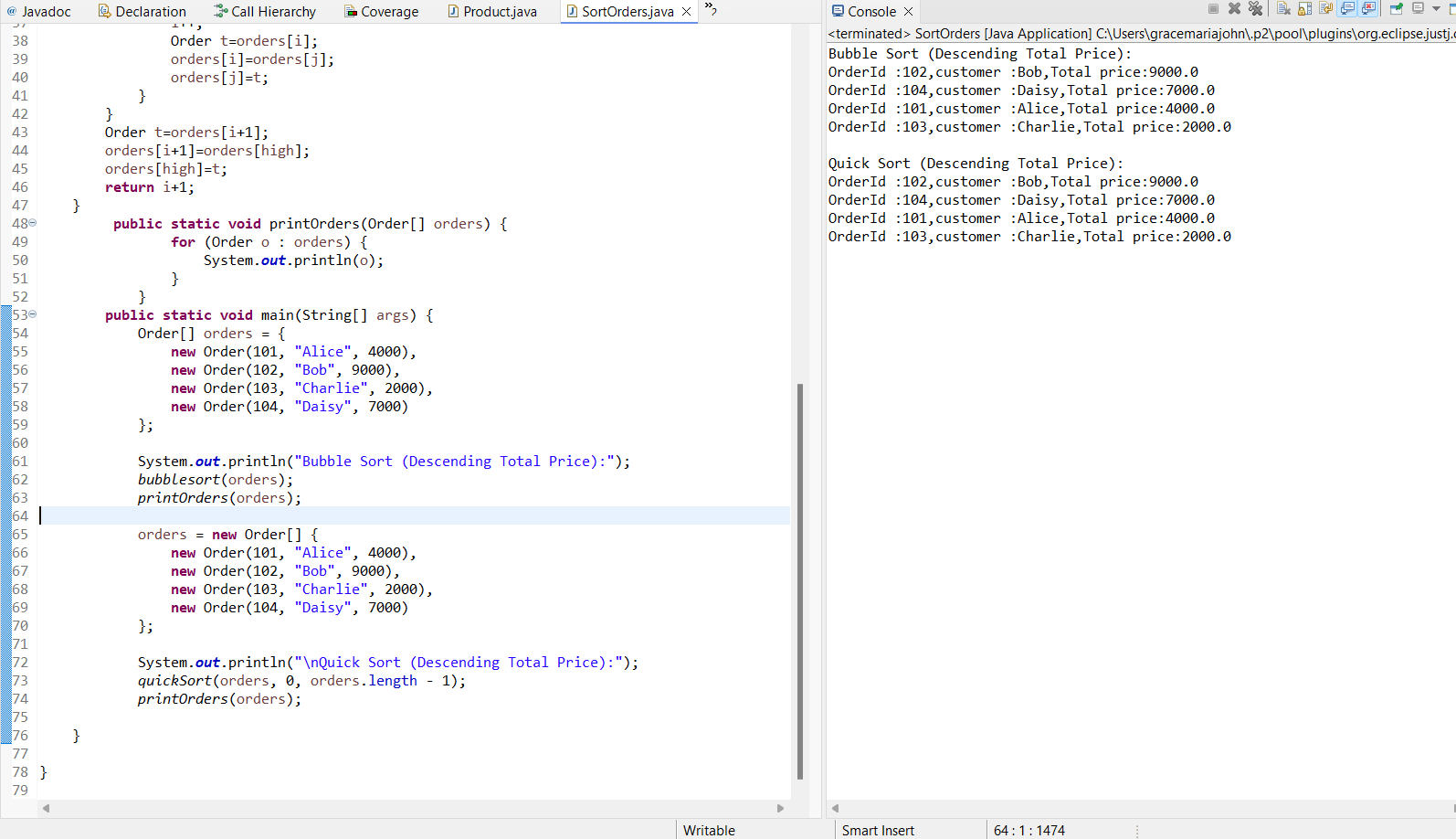
***quickSort*(orders, 0, orders.length - 1);**

***printOrders*(orders);**

**}**

**}**

**3.This is my Output:**

****

**4.Analysis**

| **Sort Method** | **Best Case** | **Average Case** | **Worst Case** | **Suitable When...** |
| --- | --- | --- | --- | --- |
| **Bubble Sort** | **O(n)** | **O(n²)** | **O(n²)** | **Very small datasets or beginners learning sorting** |
| **Quick Sort** | **O(n log n)** | **O(n log n)** | **O(n²)** | **Fastest in most real-world scenarios** |

**Exercise 4: Employee Management System**

* **1.** **Arrays are contiguous memory blocks.**
* **Each element is accessed using its index (starts from 0).**
* **Fast access using index: O(1)**
* **Fixed size: you must define the size when you create the array.**

**2.This is my code:**

**package com.example.orders;**

**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;**

**}**

**@Override**

**public String toString() {**

**return "ID: " + employeeId + ", Name: " + name +**

**", Position: " + position + ", Salary: ₹" + salary;**

**}**

**}**

**package com.example.orders;**

**public class EmployeeSystem {**

**private Employee[] employees;**

**private int size;**

**public EmployeeSystem(int capacity) {**

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

**size = 0;**

**}**

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

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

**employees[size++] = e;**

**} else {**

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

**}**

**}**

**public int searchEmployee(int id) {**

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

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

**return i;**

**}**

**}**

**return -1;**

**}**

**public void deleteEmployee(int id) {**

**int index = searchEmployee(id);**

**if (index != -1) {**

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

**employees[i] = employees[i + 1];**

**}**

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

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

**} else {**

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

**}**

**}**

**public void displayAll() {**

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

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

**} else {**

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

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

**}**

**}**

**}**

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

**EmployeeSystem system = new EmployeeSystem(5);**

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

**system.addEmployee(new Employee(2, "Bob", "Developer", 50000));**

**system.addEmployee(new Employee(3, "Charlie", "Designer", 40000));**

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

**system.displayAll();**

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

**int index = system.searchEmployee(2);**

**System.*out*.println(index != -1 ? system.employees[index] : "Not found");**

**System.*out*.println("\nDeleting Employee ID 2:");**

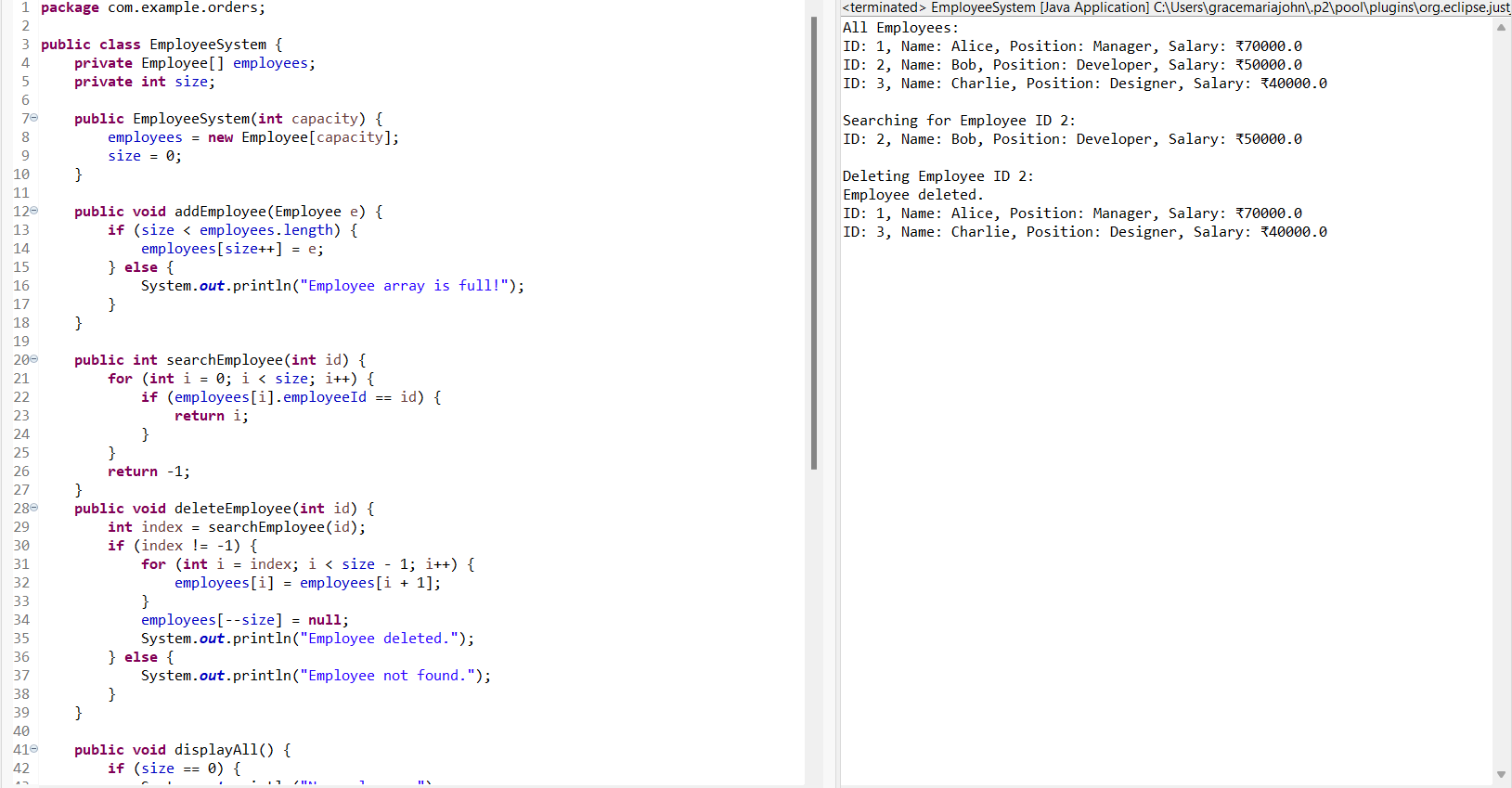
**system.deleteEmployee(2);**

**system.displayAll();**

**}**

**}**

**3.This is my Output:**

****

**4.Analysis**

* **Fixed Size: You can’t add more than the initial capacity.**
* **Deletion is expensive: You need to shift elements.**
* **No dynamic growth: Unlike ArrayList or LinkedList.**

**It is used by when fixed size and fast accesing of elements.**

**Exercise 5: Task Management System**

**1.**

|  |  |
| --- | --- |
| * **Singly Linked List** | **-Each node points to the next node. One direction only.** |
| * **Doubly Linked List** | **-Each node points to the next and previous node. Bidirectional.** |

**2.This is my code:**

**package com.example.orders;**

**public 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;**

**}**

**}**

**package com.example.orders;**

**public class TaskManager {**

**Task head = null;**

**public 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;**

**}**

**}**

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

**Task temp = head;**

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

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

**return temp;**

**}**

**temp = temp.next;**

**}**

**return null;**

**}**

**public 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 prev = head;**

**Task curr = head.next;**

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

**if (curr.taskId == id) {**

**prev.next = curr.next;**

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

**return;**

**}**

**prev = curr;**

**curr = curr.next;**

**}**

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

**}**

**public void displayTasks() {**

**if (head == null) {**

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

**return;**

**}**

**Task temp = head;**

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

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

**temp = temp.next;**

**}**

**}**

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

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

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

**tm.addTask(2, "Write Backend", "Pending");**

**tm.addTask(3, "Test System", "Completed");**

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

**tm.displayTasks();**

**System.*out*.println("\n Searching Task ID 2:");**

**Task t = tm.searchTask(2);**

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

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

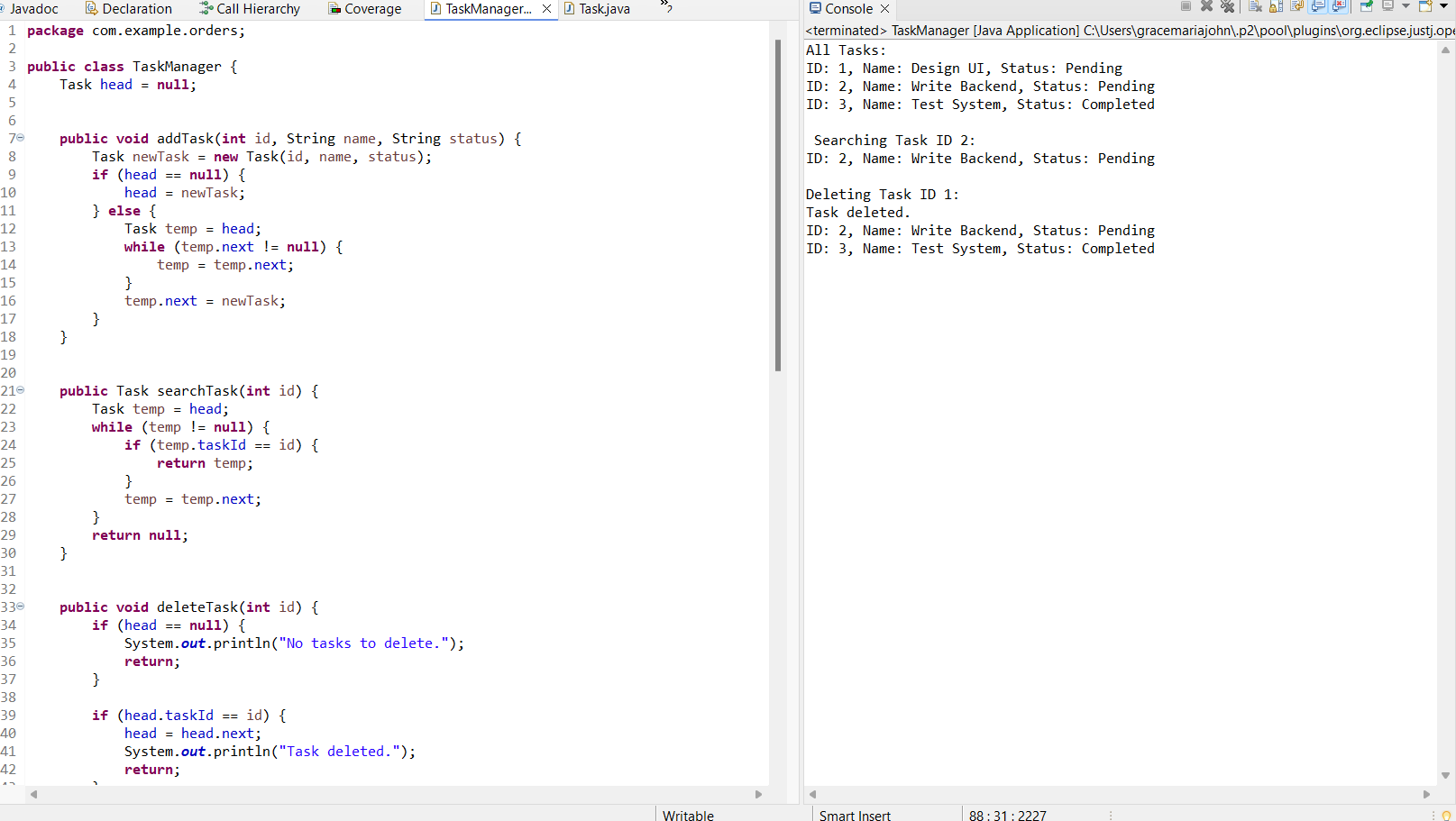
**tm.deleteTask(1);**

**tm.displayTasks();**

**}**

**}**

**3.This is my Output:**

****

**4.Analysis**

| **Operation** | **Time Complexity** | **Reason** |
| --- | --- | --- |
| **Add** | **O(n)** | **Traverse to the end** |
| **Search** | **O(n)** | **Traverse each node** |
| **Delete** | **O(n)** | **Search + reconnect nodes** |
| **Traverse** | **O(n)** | **Visit all nodes** |

**Exercise 6: Library Management System**

**1.Linear Search- check element by element**

**Binary Search -works only on sorted data**

**2.This is my code:**

**package com.example.orders;**

**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 "Book ID: " + bookId + ", Title: " + title + ", Author: " + author;**

**}**

**}**

**package com.example.orders;**

**import java.util.Arrays;**

**public class LibrarySearch {**

**public static int linearSearch(Book[] books, String targetTitle) {**

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

**if (books[i].title.equalsIgnoreCase(targetTitle)) {**

**return i;**

**}**

**}**

**return -1;**

**}**

**public static int binarySearch(Book[] books, String targetTitle) {**

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

**while (low <= high) {**

**int mid = (low + high) / 2;**

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

**if (cmp == 0) return mid;**

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

**else high = mid - 1;**

**}**

**return -1;**

**}**

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

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

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

**}**

**}**

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

**Book[] books = {**

**new Book(1, "Java Programming", "James Gosling"),**

**new Book(2, "Data Structures", "Narasimha Karumanchi"),**

**new Book(3, "C Programming", "Dennis Ritchie"),**

**new Book(4, "Python Basics", "Guido van Rossum"),**

**new Book(5, "Algorithms", "Robert Sedgewick")**

**};**

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

**int idx1 = *linearSearch*(books, "Python Basics");**

**System.*out*.println(idx1 != -1 ? books[idx1] : "Book not found");**

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

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

**int idx2 = *binarySearch*(books, "Python Basics");**

**System.*out*.println(idx2 != -1 ? books[idx2] : "Book not found");**

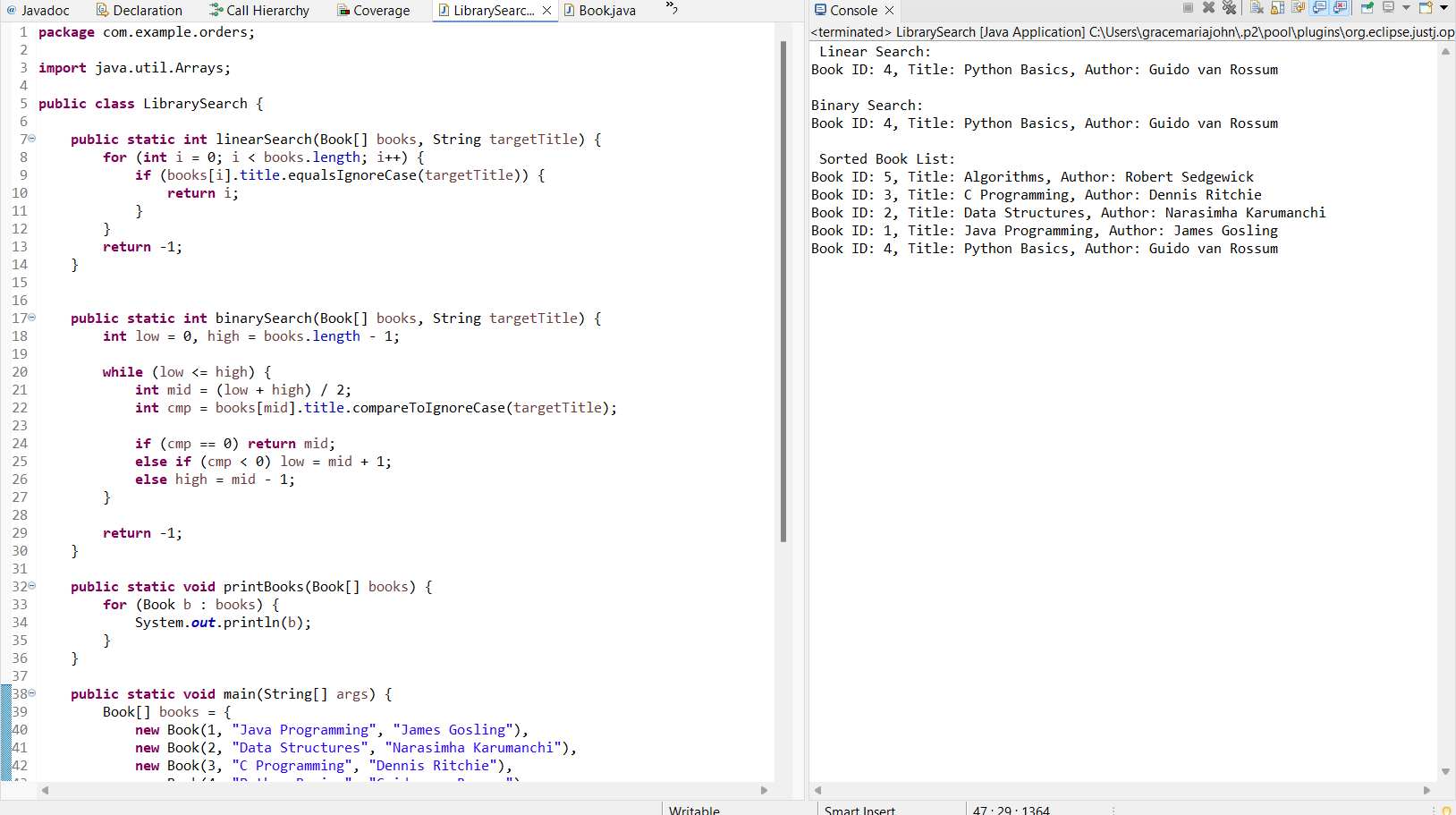
**System.*out*.println("\n Sorted Book List:");**

***printBooks*(books);**

**}**

**}**

**3.This is my Output:**

****

**4.Analysis**

| **Algorithm** | **Time Complexity** | **When to Use** |
| --- | --- | --- |
| **Linear Search** | **O(n)** | **For small or unsorted data** |
| **Binary Search** | **O(log n)** | **For sorted data and frequent searches** |

**Exercise 7: Financial Forecasting**

**1.** **Recursion is when a function calls itself to solve smaller instances of a problem.eg.,Factorial Number**

**2.This is my code:**

**package com.example.orders;**

**public class FinancialForecast {**

**public static double forecast(double currentValue, double growthRate, int years) {**

**if (years == 0) {**

**return currentValue;**

**}**

**return *forecast*(currentValue \* (1 + growthRate), growthRate, years - 1);**

**}**

**public static double forecastOptimized(double currentValue, double growthRate, int years) {**

**return currentValue \* Math.*pow*(1 + growthRate, years);**

**}**

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

**double presentValue = 10000;**

**double growthRate = 0.10;**

**int years = 5;**

**System.*out*.println("Forecast using Recursion:");**

**double future1 = *forecast*(presentValue, growthRate, years);**

**System.*out*.printf("Future Value (Recursive): ₹%.2f\n", future1);**

**System.*out*.println("\n Optimized Forecast using Math.pow:");**

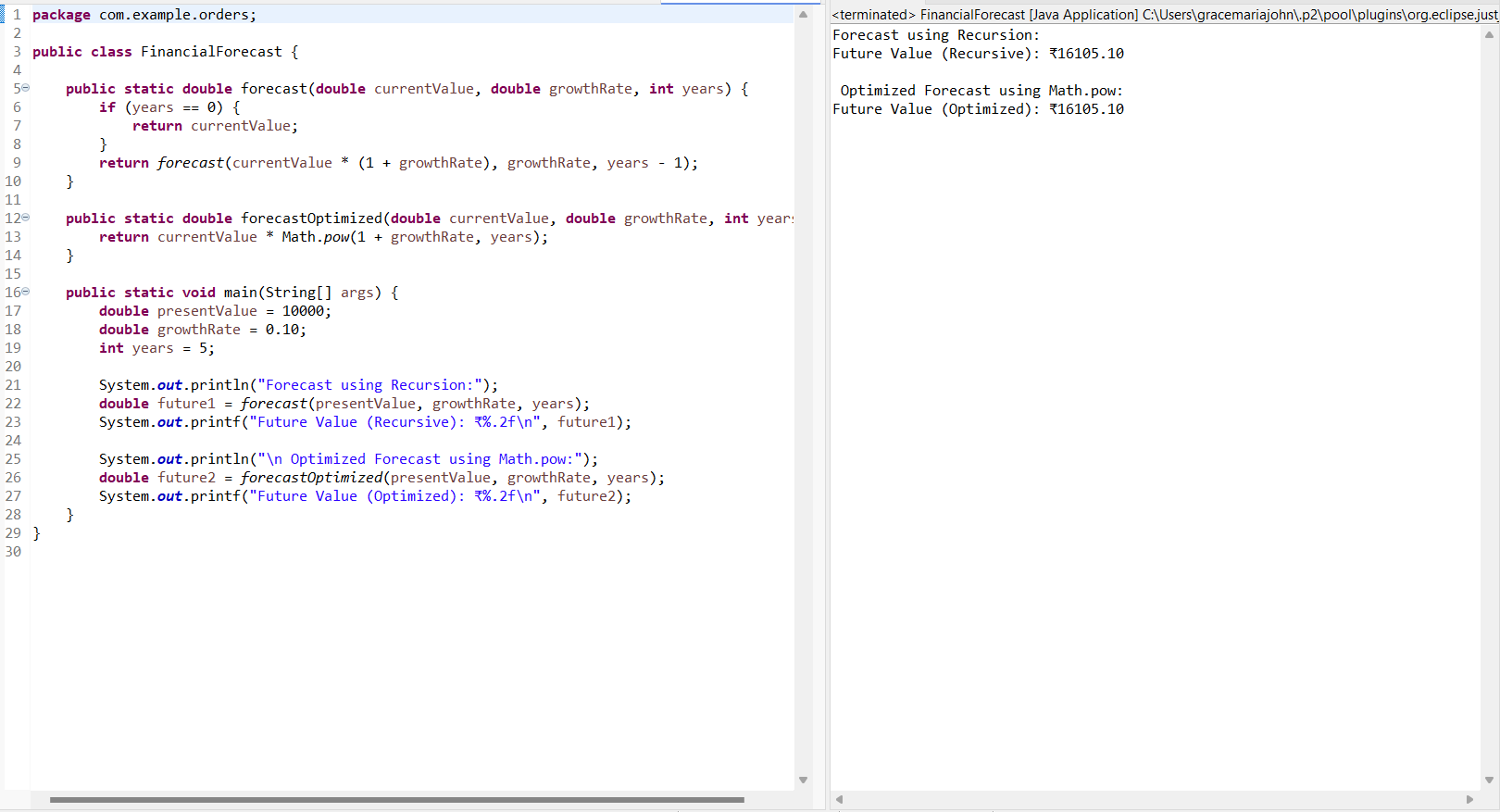
**double future2 = *forecastOptimized*(presentValue, growthRate, years);**

**System.*out*.printf("Future Value (Optimized): ₹%.2f\n", future2);**

**}**

**}**

**3.This is my Output:**

****

4.Analysis

| **Function** | **Time Complexity** | **Explanation** |
| --- | --- | --- |
| Recursive Forecast | O(n) | One call per year (linear growth) |
| Optimized (pow) | O(1) | Uses built-in exponential function |

* Recursive functions can use **extra memory** due to function call stack.
* For large n (e.g., 1000 years), recursion may cause a **stack overflow**.