**Exercise 1: Inventory Management System**

**InventoryManagementSystem.java**

**package** mypackage;

**import** java.util.\*;

**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** String getProductName() { **return** productName; }

**public** **int** getQuantity() { **return** quantity; }

**public** **double** getPrice() { **return** price; }

**public** **void** setProductName(String productName) { **this**.productName = productName; }

**public** **void** setQuantity(**int** quantity) { **this**.quantity = quantity; }

**public** **void** setPrice(**double** price) { **this**.price = price; }

@Override

**public** String toString() {

**return** "Product ID: " + productId + ", Name: " + productName +

", Quantity: " + quantity + ", Price: $" + price;

}

}

**public** **class** InventoryManagementSystem {

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

**private** **static** Scanner *sc* = **new** Scanner(System.***in***);

**public** **static** **void** addProduct() {

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

**int** id = *sc*.nextInt();

*sc*.nextLine();

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

System.***out***.println("Product ID already exists!");

**return**;

}

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

String name = *sc*.nextLine();

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

**int** qty = *sc*.nextInt();

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

**double** price = *sc*.nextDouble();

Product product = **new** Product(id, name, qty, price);

*inventory*.put(id, product);

System.***out***.println("Product added successfully!");

}

**public** **static** **void** updateProduct() {

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

**int** id = *sc*.nextInt();

*sc*.nextLine();

**if** (!*inventory*.containsKey(id)) {

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

**return**;

}

Product product = *inventory*.get(id);

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

String name = *sc*.nextLine();

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

**int** qty = *sc*.nextInt();

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

**double** price = *sc*.nextDouble();

product.setProductName(name);

product.setQuantity(qty);

product.setPrice(price);

System.***out***.println("Product updated successfully!");

}

**public** **static** **void** deleteProduct() {

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

**int** id = *sc*.nextInt();

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

System.***out***.println("Product deleted successfully!");

} **else** {

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

}

}

**public** **static** **void** displayInventory() {

**if** (*inventory*.isEmpty()) {

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

**return**;

}

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

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

}

}

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

**int** choice;

**do** {

System.***out***.println("\nInventory Management System");

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

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

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

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

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

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

choice = *sc*.nextInt();

**switch** (choice) {

**case** 1: *addProduct*(); **break**;

**case** 2: *updateProduct*(); **break**;

**case** 3: *deleteProduct*(); **break**;

**case** 4: *displayInventory*(); **break**;

**case** 5: System.***out***.println("Exiting..."); **break**;

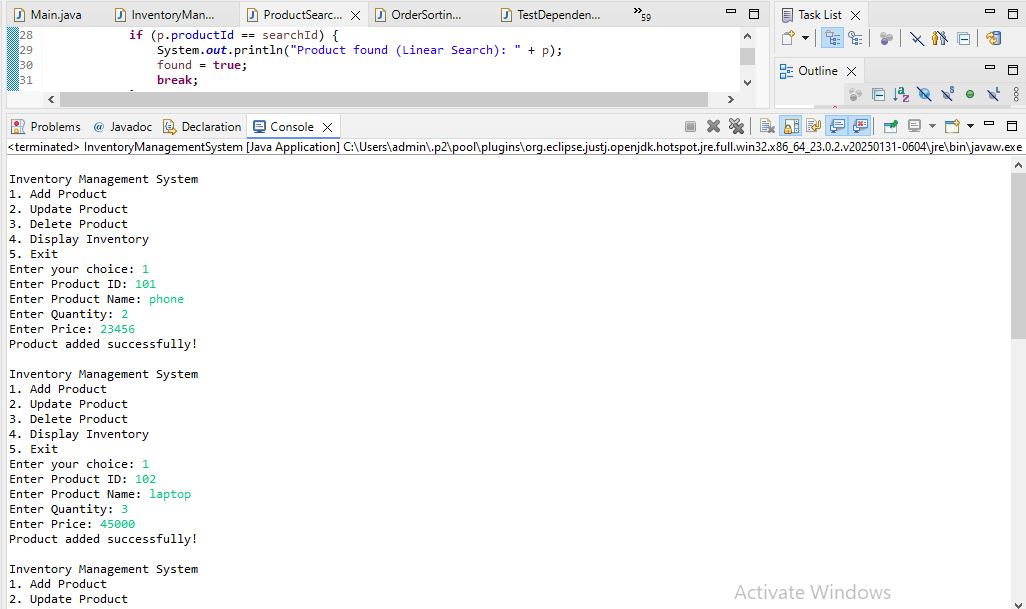
**default**: System.***out***.println("Invalid choice!");

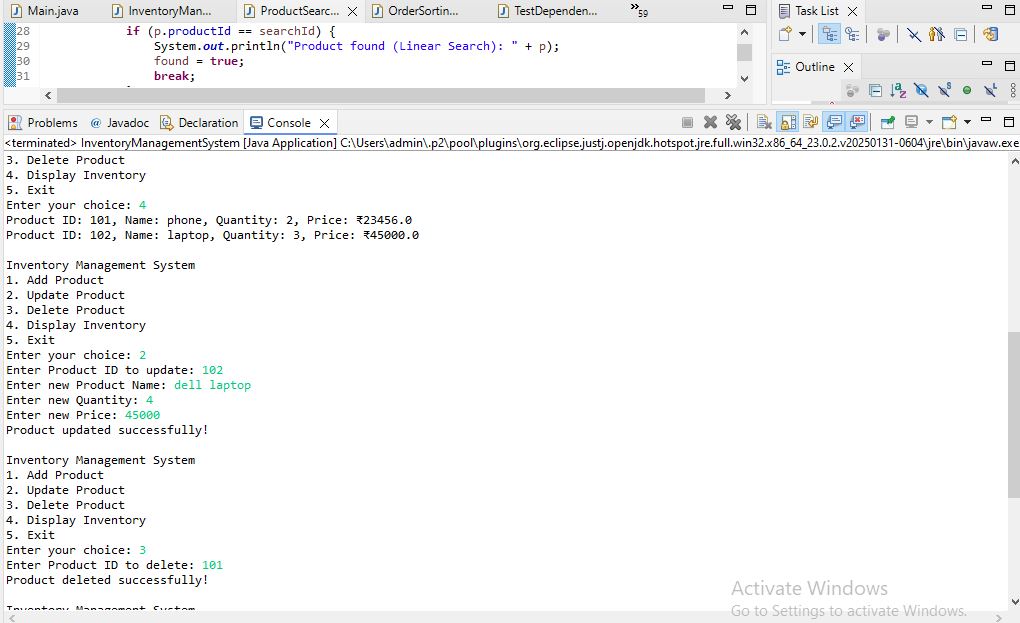
}

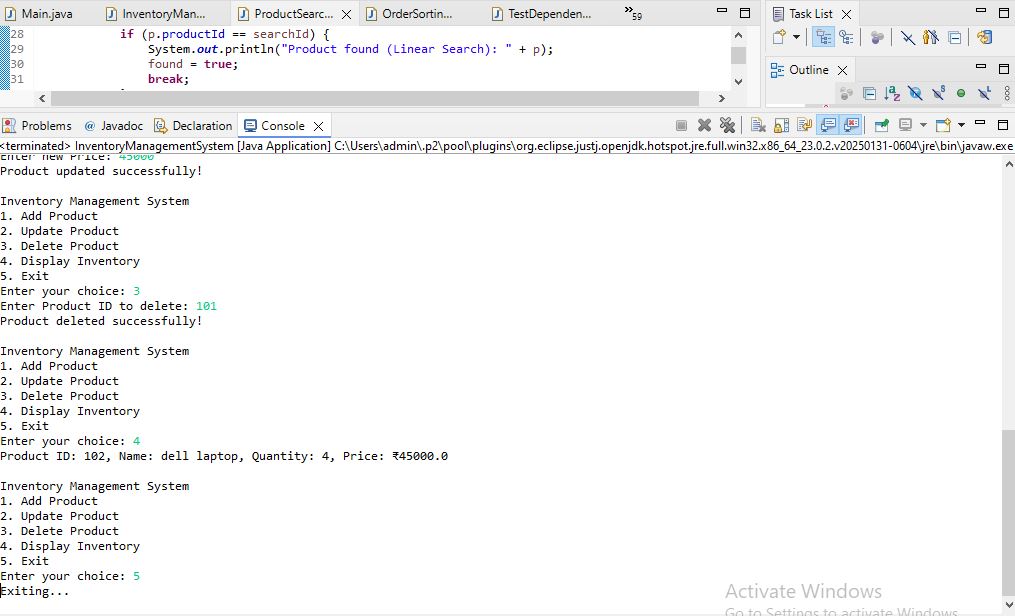
} **while** (choice != 5);

}

}







**➔ Why are data structures and algorithms essential in handling large**

**inventories?**

* In a warehouse, there can be thousands or even millions of products.
* Efficient storage and retrieval of product information is critical for:
  + Fast searching
  + Quick updates
  + Real-time stock management
* **Data structures** help organize data for faster access.
* **Algorithms** help perform operations (like search, insert, update, delete) efficiently.

Without proper data structures, operations may take longer time, causing system slowdowns and operational delays.

**➔ Suitable data structures for this problem:**

| **Data Structure** | **Use Case** | **Advantage** |
| --- | --- | --- |
| **ArrayList** | Simple, ordered list of products | Easy to implement, good for small data |
| **HashMap** | Key-value pair storage | Fast lookup, insertion, deletion using productId as key |
| **TreeMap** | Sorted key-value storage | Allows fast sorted operations |

In this system, we use **HashMap** because:

* Product ID is unique.
* Fast access by key (productId).
* Insertion, update, delete, and search are very efficient.

**2. Setup**

* A new project is created for the Inventory Management System.
* The main class is named InventoryManagementSystem.
* A Product class is defined to store product details.

**3. Implementation**

**➔ Product class**

Attributes:

* productId (int): Unique ID for each product.
* productName (String): Name of product.
* quantity (int): Stock quantity available.
* price (double): Price of product.

**➔ Data Structure**

* We use **HashMap<Integer, Product> inventory**:
  + Key: productId
  + Value: Product object
* HashMap allows quick access based on productId.

**➔ Operations Implemented**

| **Operation** | **Description** |
| --- | --- |
| **Add Product** | Adds new product into inventory |
| **Update Product** | Updates details of existing product |
| **Delete Product** | Removes product from inventory |
| **Display Inventory** | Displays all products currently in inventory |

**4. Analysis**

**➔ Time Complexity of Operations using HashMap**

| **Operation** | **Time Complexity** | **Explanation** |
| --- | --- | --- |
| **Add Product** | O(1) | Inserting into HashMap |
| **Update Product** | O(1) | Access by key and update fields |
| **Delete Product** | O(1) | Remove by key |
| **Display Inventory** | O(n) | Traverses all products |

HashMap provides **constant time (O(1))** for add, update, and delete operations.  
Only display requires linear time (O(n)) since we need to go through all entries.

**➔ How to optimize further?**

* In real-world systems, as the data grows larger:
  + Use **database indexing** for very large datasets.
  + Use **concurrent data structures** for multi-user systems.
  + Use **persistent storage** (like SQL or NoSQL databases) for durability.
  + Implement **caching** for frequent queries.

For simple warehouse systems with moderate data size, HashMap provides excellent performance.

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

**ProductSearchSystem.java**

**package** mypackage;

**import** java.util.\*;

**public** **class** ProductSearchSystem {

// Step 2: Product class

**static** **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;

}

**public** String toString() {

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

}

}

// Linear Search (unsorted array)

**public** **static** **void** linearSearch(Product[] products, **int** searchId) {

**boolean** found = **false**;

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

**if** (p.productId == searchId) {

System.***out***.println("Product found (Linear Search): " + p);

found = **true**;

**break**;

}

}

**if** (!found) {

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

}

}

// Binary Search (sorted array)

**public** **static** **void** binarySearch(Product[] products, **int** searchId) {

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

**boolean** found = **false**;

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

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

**if** (products[mid].productId == searchId) {

System.***out***.println("Product found (Binary Search): " + products[mid]);

found = **true**;

**break**;

} **else** **if** (products[mid].productId < searchId) {

left = mid + 1;

} **else** {

right = mid - 1;

}

}

**if** (!found) {

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

}

}

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

Scanner sc = **new** Scanner(System.***in***);

// Step 2: Setup

System.***out***.print("Enter number of products: ");

**int** n = sc.nextInt();

sc.nextLine();

Product[] products = **new** Product[n];

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

System.***out***.println("Enter details for Product " + (i+1));

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

**int** id = sc.nextInt();

sc.nextLine();

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

String name = sc.nextLine();

System.***out***.print("Category: ");

String category = sc.nextLine();

products[i] = **new** Product(id, name, category);

}

// Sorting the array for binary search

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

System.***out***.print("\nEnter Product ID to search: ");

**int** searchId = sc.nextInt();

// Step 3: Implementation

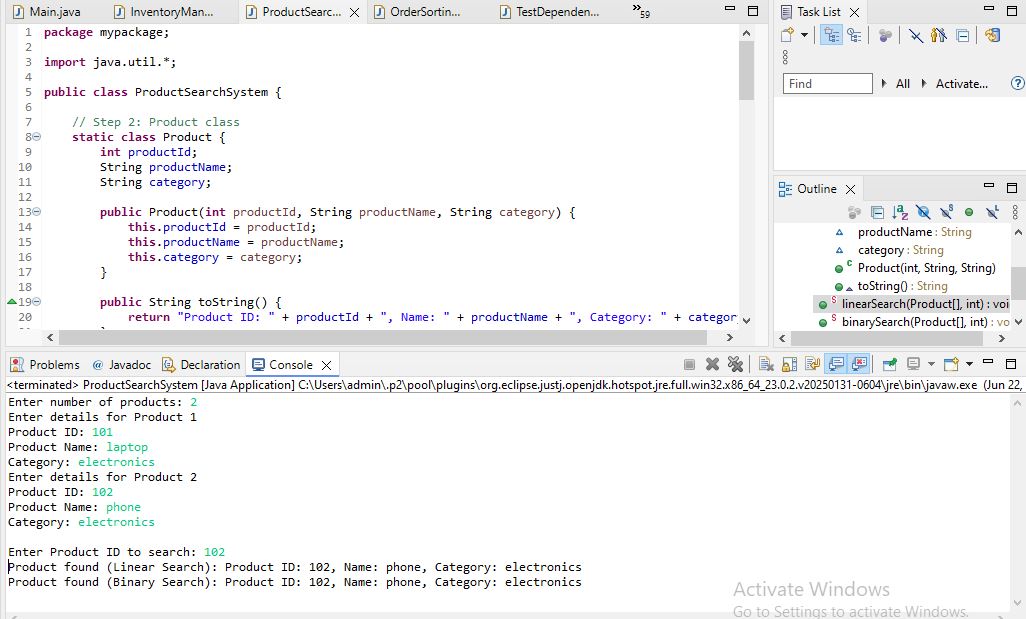
*linearSearch*(products, searchId);

*binarySearch*(products, searchId);

sc.close();

}

}



**1. Understand Asymptotic Notation**

**➔ What is Big O Notation?**

Big O notation is a mathematical notation that describes how an algorithm's **time or space complexity grows** as the input size increases. It focuses on:

* **Scalability**: How efficiently an algorithm handles large data.
* **Machine-independent**: Ignores hardware, language, or compiler-specific details.
* **Worst-case behavior**: Usually used to analyze the maximum time an algorithm can take.

**Example:**

* O(1) — constant time (very fast)
* O(n) — linear time (slower as data increases)
* O(log n) — logarithmic time (efficient even for large data)
* O(n²) — quadratic time (very slow for large data)

**➔ Why is it useful?**

* Helps us **compare algorithms** objectively.
* Helps select the most **efficient algorithm** for large datasets.
* Especially useful in search problems, sorting, and optimization tasks.

**➔ Best, Average, Worst-case for search operations**

| **Algorithm** | **Best Case** | **Average Case** | **Worst Case** |
| --- | --- | --- | --- |
| **Linear Search** | O(1) (first element match) | O(n/2) ≈ O(n) | O(n) (last element or not found) |
| **Binary Search** | O(1) (middle element match) | O(log n) | O(log n) (worst case: many divisions) |

**2. Setup**

We define a class Product with the following attributes:

* productId (integer): unique identifier for each product.
* productName (String): name of the product.
* category (String): category to which the product belongs.

This structure allows us to search products based on their ID.

**3. Implementation**

**➔ Linear Search**

* The linear search algorithm scans the array **from start to end**.
* Compares each product's productId with the target productId.
* If found, returns the product; if not found, returns not found after checking all items.
* **Does not require sorting**.

**➔ Binary Search**

* Binary search requires the array to be **sorted by productId**.
* It divides the search space into halves:
  + If the middle element matches, we return it.
  + If the target is smaller, search in the left half.
  + If the target is larger, search in the right half.
* Much faster for large datasets compared to linear search.

**4. Analysis**

**➔ Time Complexity Comparison**

| **Algorithm** | **Time Complexity** |
| --- | --- |
| **Linear Search** | O(n) |
| **Binary Search** | O(log n) |

* Linear search checks each element one by one, making it inefficient for large data.
* Binary search significantly reduces search time by dividing the problem size repeatedly.

**➔ Which algorithm is more suitable for your platform?**

* In **small datasets**, both algorithms perform well.
* In **large-scale e-commerce platforms**, where millions of products exist:
  + Binary search is much more efficient (O(log n)).
  + However, binary search needs the data to be **sorted**.
  + In real-world e-commerce platforms, search is even further optimized using:
    - **Indexing** in databases.
    - **Search engines** like Elasticsearch.
    - **Hashing** or **B-trees** for even faster search.

**Exercise 3: Sorting Customer Orders**

**OrderSortingSystem.java**

**package** mypackage;

**import** java.util.\*;

**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** String getCustomerName() { **return** customerName; }

**public** **double** getTotalPrice() { **return** totalPrice; }

@Override

**public** String toString() {

**return** String.*format*("Order ID: %d, Customer: %s, Total Price: ₹%,.2f",

orderId, customerName, totalPrice);

}

}

**public** **class** OrderSortingSystem {

// Bubble Sort implementation

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

// Swap

Order temp = orders[j];

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

orders[j + 1] = temp;

}

}

}

}

// Quick Sort implementation

**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].getTotalPrice();

**int** i = low - 1;

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

**if** (orders[j].getTotalPrice() >= pivot) { // Descending order

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;

}

// Display orders

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

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

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

}

}

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

Scanner sc = **new** Scanner(System.***in***);

System.***out***.print("Enter number of orders: ");

**int** n = sc.nextInt();

sc.nextLine();

Order[] orders = **new** Order[n];

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

System.***out***.println("Enter details for Order " + (i + 1));

System.***out***.print("Order ID: ");

**int** orderId = sc.nextInt();

sc.nextLine();

System.***out***.print("Customer Name: ");

String customerName = sc.nextLine();

System.***out***.print("Total Price: ");

**double** totalPrice = sc.nextDouble();

sc.nextLine();

orders[i] = **new** Order(orderId, customerName, totalPrice);

}

System.***out***.println("\nOrders sorted using Bubble Sort (High to Low Total Price):");

*bubbleSort*(orders);

*displayOrders*(orders);

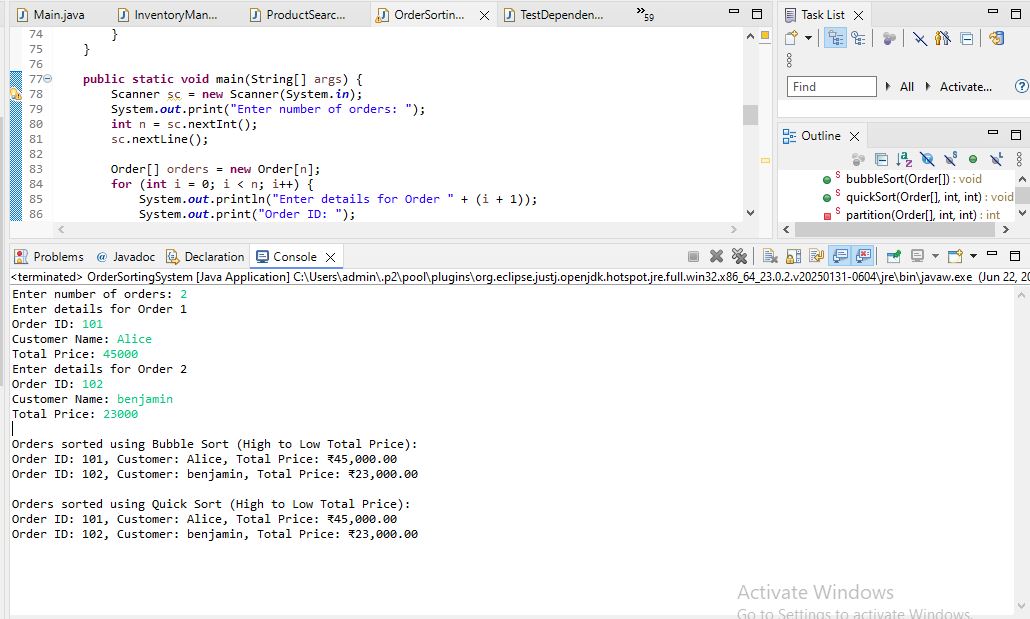
System.***out***.println("\nOrders sorted using Quick Sort (High to Low Total Price):");

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

*displayOrders*(orders);

}

}



**1. Understand Sorting Algorithms**

**➔ Bubble Sort:**

* Compares adjacent elements and swaps them if out of order.
* Simple but inefficient for large datasets.
* Time Complexity:
  + Best Case: O(n) (if already sorted)
  + Average Case: O(n²)
  + Worst Case: O(n²)

**➔ Insertion Sort:**

* Builds the sorted array one element at a time.
* Good for small datasets.
* Time Complexity:
  + Best Case: O(n)
  + Average & Worst Case: O(n²)

**➔ Quick Sort:**

* Divide and conquer algorithm.
* Picks a pivot, partitions the array, recursively sorts partitions.
* Very efficient for large datasets.
* Time Complexity:
  + Best & Average Case: O(n log n)
  + Worst Case: O(n²) (if poorly chosen pivot)

**➔ Merge Sort:**

* Also divide and conquer.
* Always divides into halves, merges back sorted.
* Stable, guaranteed O(n log n) even in worst case.
* Requires extra space.

**2. Setup**

* Class Order is created with:
  + orderId
  + customerName
  + totalPrice

**3. Implementation**

* **Bubble Sort** and **Quick Sort** are implemented to sort orders by totalPrice (descending).
* Input is taken from user for orders.
* Both sorting methods are applied and displayed.

**4. Analysis**

**➔ Performance Comparison**

| **Algorithm** | **Best Case** | **Average Case** | **Worst Case** |
| --- | --- | --- | --- |
| Bubble Sort | O(n) | O(n²) | O(n²) |
| Quick Sort | O(n log n) | O(n log n) | O(n²) |

* Bubble Sort performs poorly for large data.
* Quick Sort is much faster for most cases.

**➔ Why is Quick Sort preferred?**

* Much faster for large data.
* Average time complexity O(n log n).
* Bubble Sort’s quadratic time makes it unsuitable for real-world use.

**Exercise 4: Employee Management System**

**EmployeeManagementSystem.java**

**package** mypackage;

**import** java.util.Scanner;

**class** Employee {

**private** **int** employeeId;

**private** String name;

**private** String position;

**private** **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** **int** getEmployeeId() { **return** employeeId; }

**public** String getName() { **return** name; }

**public** String getPosition() { **return** position; }

**public** **double** getSalary() { **return** salary; }

@Override

**public** String toString() {

**return** String.*format*("Employee ID: %d, Name: %s, Position: %s, Salary: ₹%,.2f",

employeeId, name, position, salary);

}

}

**public** **class** EmployeeManagementSystem {

**private** **static** **final** **int** ***MAX\_EMPLOYEES*** = 100;

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

**private** **static** **int** *count* = 0;

**private** **static** Scanner *sc* = **new** Scanner(System.***in***);

**public** **static** **void** addEmployee() {

**if** (*count* >= ***MAX\_EMPLOYEES***) {

System.***out***.println("Employee limit reached!");

**return**;

}

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

**int** id = *sc*.nextInt();

*sc*.nextLine();

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

String name = *sc*.nextLine();

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

String position = *sc*.nextLine();

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

**double** salary = *sc*.nextDouble();

*sc*.nextLine();

*employees*[*count*++] = **new** Employee(id, name, position, salary);

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

}

**public** **static** **void** searchEmployee() {

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

**int** id = *sc*.nextInt();

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

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

System.***out***.println("Employee found: " + *employees*[i]);

**return**;

}

}

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

}

**public** **static** **void** traverseEmployees() {

**if** (*count* == 0) {

System.***out***.println("No employees in the system.");

**return**;

}

System.***out***.println("Employee List:");

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

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

}

}

**public** **static** **void** deleteEmployee() {

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

**int** id = *sc*.nextInt();

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

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

// Shift all elements after i to the left

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

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

}

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

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

**return**;

}

}

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

}

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

**int** choice;

**do** {

System.***out***.println("\nEmployee Management System");

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

System.***out***.println("2. Search Employee");

System.***out***.println("3. Display All Employees");

System.***out***.println("4. Delete Employee");

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

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

choice = *sc*.nextInt();

**switch** (choice) {

**case** 1: *addEmployee*(); **break**;

**case** 2: *searchEmployee*(); **break**;

**case** 3: *traverseEmployees*(); **break**;

**case** 4: *deleteEmployee*(); **break**;

**case** 5: System.***out***.println("Exiting..."); **break**;

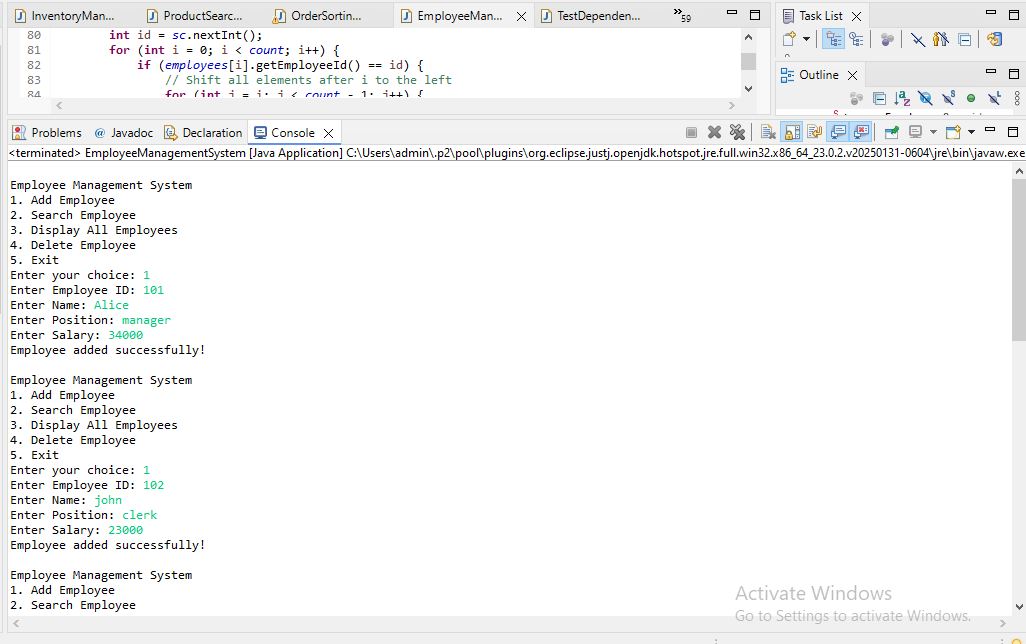
**default**: System.***out***.println("Invalid choice!");

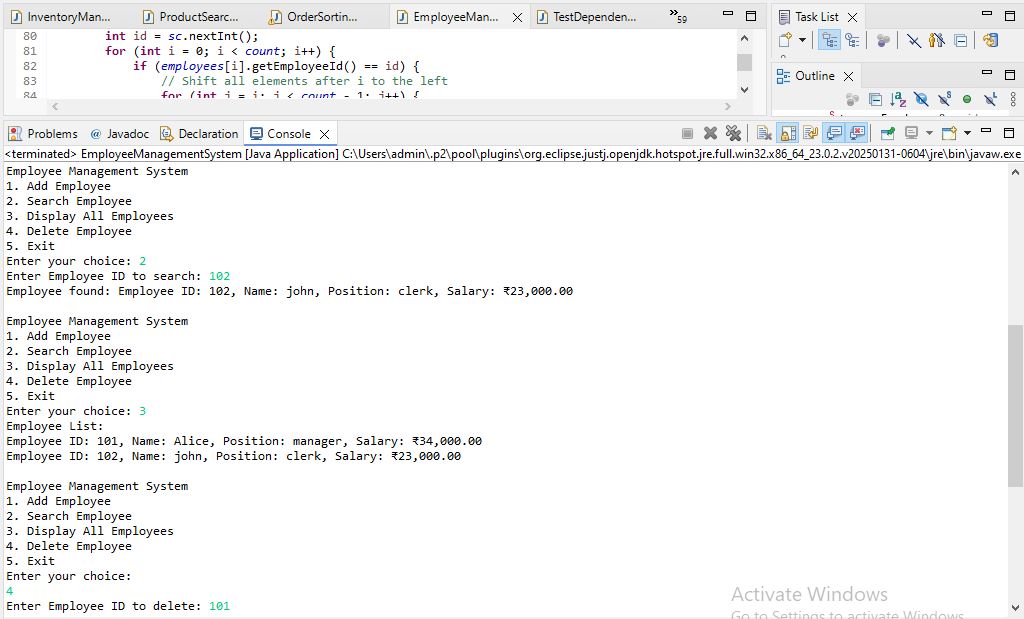
}

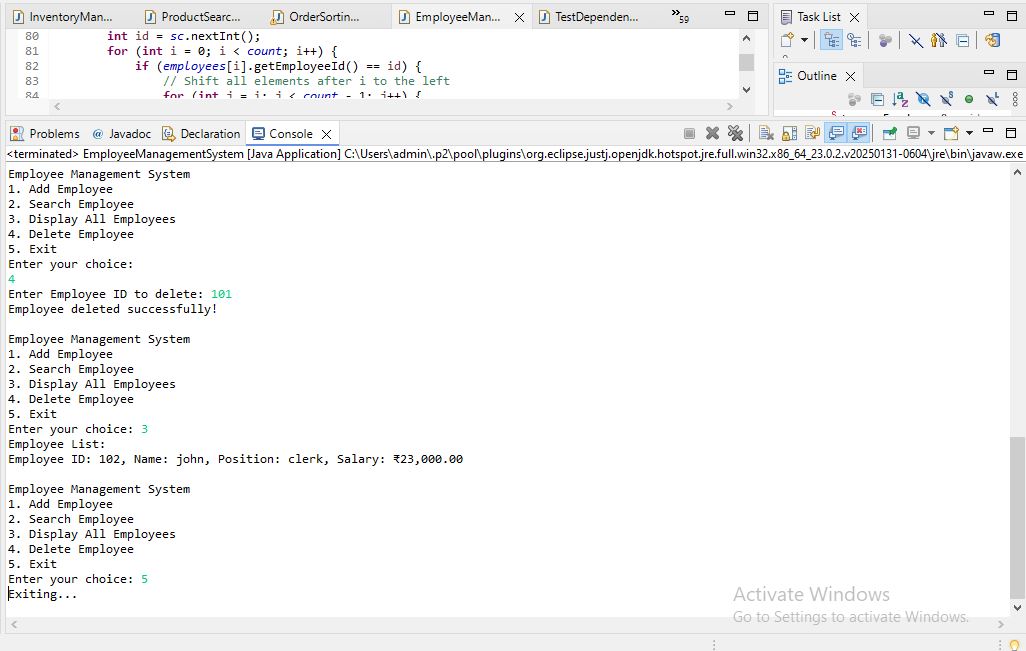
} **while** (choice != 5);

}

}







**1. Understand Array Representation**

**➔ How arrays are represented in memory:**

* Arrays store elements in **contiguous memory locations**.
* Each element can be directly accessed using its index.
* Memory address for element i = base address + (i \* size\_of\_element)
* Efficient for random access.

**➔ Advantages of arrays:**

* Fast access (O(1)) using index.
* Simple structure, easy to use.
* Efficient use of memory for fixed-size data.

**2. Setup**

* We create a class Employee with:
  + employeeId (int)
  + name (String)
  + position (String)
  + salary (double)
* An array Employee[] is used to store employee records.

**3. Implementation**

* Use a fixed-size array Employee[] employees = new Employee[100].
* Operations implemented:
  + **Add Employee** → adds a new record.
  + **Search Employee** → searches by employeeId.
  + **Traverse Employees** → lists all employees.
  + **Delete Employee** → removes an employee by shifting elements.

**4. Analysis**

**➔ Time Complexity**

| **Operation** | **Time Complexity** | **Explanation** |
| --- | --- | --- |
| **Add** | O(1) | Inserting at end (using count index) |
| **Search** | O(n) | Linear search |
| **Traverse** | O(n) | Loop through all employees |
| **Delete** | O(n) | Search + shift elements |

**➔ Limitations of arrays:**

* Fixed size — difficult to resize dynamically.
* Insertion and deletion (except at end) require shifting elements (O(n)).
* Waste of memory if many empty slots.
* No built-in dynamic growth like ArrayList or LinkedList.

**➔ When to use arrays:**

* When maximum size is known beforehand.
* When frequent random access (using index) is needed.
* For small to moderate datasets where dynamic resizing is not critical.

**Exercise 5: Task Management System**

**Task Management System.java**

**package** mypackage;

**import** java.util.Scanner;

// Task Node for Singly Linked List

**class** SinglyTask {

**int** taskId;

String taskName;

String status;

SinglyTask next;

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

**this**.taskId = taskId;

**this**.taskName = taskName;

**this**.status = status;

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

}

}

// Task Node for Doubly Linked List

**class** DoublyTask {

**int** taskId;

String taskName;

String status;

DoublyTask next, prev;

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

**this**.taskId = taskId;

**this**.taskName = taskName;

**this**.status = status;

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

**this**.prev = **null**;

}

}

// Task Node for Circular Linked List

**class** CircularTask {

**int** taskId;

String taskName;

String status;

CircularTask next;

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

**this**.taskId = taskId;

**this**.taskName = taskName;

**this**.status = status;

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

}

}

**public** **class** TaskManagementSystem {

**static** Scanner *sc* = **new** Scanner(System.***in***);

**static** SinglyTask *singlyHead* = **null**;

**static** DoublyTask *doublyHead* = **null**;

**static** CircularTask *circularHead* = **null**;

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

**while** (**true**) {

System.***out***.println("\nSelect Linked List Type:");

System.***out***.println("1. Singly Linked List");

System.***out***.println("2. Doubly Linked List");

System.***out***.println("3. Circular Linked List");

System.***out***.println("4. Exit");

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

**int** type = *sc*.nextInt();

**if** (type == 4) **break**;

**while** (**true**) {

System.***out***.println("\n1. Add Task");

System.***out***.println("2. Search Task");

System.***out***.println("3. Display Tasks");

System.***out***.println("4. Delete Task");

System.***out***.println("5. Change Linked List Type");

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

**int** choice = *sc*.nextInt();

**if** (choice == 5) **break**;

**switch** (type) {

**case** 1: *handleSingly*(choice); **break**;

**case** 2: *handleDoubly*(choice); **break**;

**case** 3: *handleCircular*(choice); **break**;

}

}

}

}

// Singly Linked List Operations

**public** **static** **void** handleSingly(**int** choice) {

**switch** (choice) {

**case** 1:

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

**int** id1 = *sc*.nextInt(); *sc*.nextLine();

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

String name1 = *sc*.nextLine();

System.***out***.print("Enter Status: ");

String status1 = *sc*.nextLine();

SinglyTask newNode = **new** SinglyTask(id1, name1, status1);

**if** (*singlyHead* == **null**) *singlyHead* = newNode;

**else** {

SinglyTask curr = *singlyHead*;

**while** (curr.next != **null**) curr = curr.next;

curr.next = newNode;

}

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

**break**;

**case** 2:

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

**int** searchId1 = *sc*.nextInt();

SinglyTask curr1 = *singlyHead*;

**while** (curr1 != **null**) {

**if** (curr1.taskId == searchId1) {

System.***out***.println("Task found: " + curr1.taskId + " " + curr1.taskName + " " + curr1.status);

**return**;

}

curr1 = curr1.next;

}

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

**break**;

**case** 3:

SinglyTask temp1 = *singlyHead*;

**if** (temp1 == **null**) {

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

**return**;

}

**while** (temp1 != **null**) {

System.***out***.println(temp1.taskId + " " + temp1.taskName + " " + temp1.status);

temp1 = temp1.next;

}

**break**;

**case** 4:

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

**int** delId1 = *sc*.nextInt();

**if** (*singlyHead* == **null**) {

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

**return**;

}

**if** (*singlyHead*.taskId == delId1) {

*singlyHead* = *singlyHead*.next;

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

**return**;

}

SinglyTask prev1 = *singlyHead*;

SinglyTask del1 = *singlyHead*.next;

**while** (del1 != **null**) {

**if** (del1.taskId == delId1) {

prev1.next = del1.next;

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

**return**;

}

prev1 = del1;

del1 = del1.next;

}

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

**break**;

}

}

// Doubly Linked List Operations

**public** **static** **void** handleDoubly(**int** choice) {

**switch** (choice) {

**case** 1:

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

**int** id2 = *sc*.nextInt(); *sc*.nextLine();

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

String name2 = *sc*.nextLine();

System.***out***.print("Enter Status: ");

String status2 = *sc*.nextLine();

DoublyTask newNode = **new** DoublyTask(id2, name2, status2);

**if** (*doublyHead* == **null**) *doublyHead* = newNode;

**else** {

DoublyTask curr = *doublyHead*;

**while** (curr.next != **null**) curr = curr.next;

curr.next = newNode;

newNode.prev = curr;

}

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

**break**;

**case** 2:

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

**int** searchId2 = *sc*.nextInt();

DoublyTask curr2 = *doublyHead*;

**while** (curr2 != **null**) {

**if** (curr2.taskId == searchId2) {

System.***out***.println("Task found: " + curr2.taskId + " " + curr2.taskName + " " + curr2.status);

**return**;

}

curr2 = curr2.next;

}

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

**break**;

**case** 3:

DoublyTask temp2 = *doublyHead*;

**if** (temp2 == **null**) {

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

**return**;

}

**while** (temp2 != **null**) {

System.***out***.println(temp2.taskId + " " + temp2.taskName + " " + temp2.status);

temp2 = temp2.next;

}

**break**;

**case** 4:

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

**int** delId2 = *sc*.nextInt();

**if** (*doublyHead* == **null**) {

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

**return**;

}

**if** (*doublyHead*.taskId == delId2) {

*doublyHead* = *doublyHead*.next;

**if** (*doublyHead* != **null**) *doublyHead*.prev = **null**;

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

**return**;

}

DoublyTask currDel = *doublyHead*.next;

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

**if** (currDel.taskId == delId2) {

currDel.prev.next = currDel.next;

**if** (currDel.next != **null**)

currDel.next.prev = currDel.prev;

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

**return**;

}

currDel = currDel.next;

}

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

**break**;

}

}

// Circular Linked List Operations

**public** **static** **void** handleCircular(**int** choice) {

**switch** (choice) {

**case** 1:

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

**int** id3 = *sc*.nextInt(); *sc*.nextLine();

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

String name3 = *sc*.nextLine();

System.***out***.print("Enter Status: ");

String status3 = *sc*.nextLine();

CircularTask newNode = **new** CircularTask(id3, name3, status3);

**if** (*circularHead* == **null**) {

*circularHead* = newNode;

*circularHead*.next = *circularHead*; // make circular

} **else** {

CircularTask temp = *circularHead*;

**while** (temp.next != *circularHead*) temp = temp.next;

temp.next = newNode;

newNode.next = *circularHead*;

}

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

**break**;

**case** 2:

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

**int** searchId3 = *sc*.nextInt();

**if** (*circularHead* == **null**) {

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

**return**;

}

CircularTask tempSearch = *circularHead*;

**do** {

**if** (tempSearch.taskId == searchId3) {

System.***out***.println("Task found: " + tempSearch.taskId + " " + tempSearch.taskName + " " + tempSearch.status);

**return**;

}

tempSearch = tempSearch.next;

} **while** (tempSearch != *circularHead*);

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

**break**;

**case** 3:

**if** (*circularHead* == **null**) {

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

**return**;

}

CircularTask temp3 = *circularHead*;

**do** {

System.***out***.println(temp3.taskId + " " + temp3.taskName + " " + temp3.status);

temp3 = temp3.next;

} **while** (temp3 != *circularHead*);

**break**;

**case** 4:

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

**int** delId3 = *sc*.nextInt();

**if** (*circularHead* == **null**) {

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

**return**;

}

// Only one node case

**if** (*circularHead*.taskId == delId3 && *circularHead*.next == *circularHead*) {

*circularHead* = **null**;

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

**return**;

}

CircularTask prev = *circularHead*;

CircularTask curr = *circularHead*.next;

**boolean** found = **false**;

**do** {

**if** (curr.taskId == delId3) {

prev.next = curr.next;

**if** (curr == *circularHead*) *circularHead* = curr.next;

found = **true**;

**break**;

}

prev = curr;

curr = curr.next;

} **while** (prev.next != *circularHead*);

**if** (found)

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

**else**

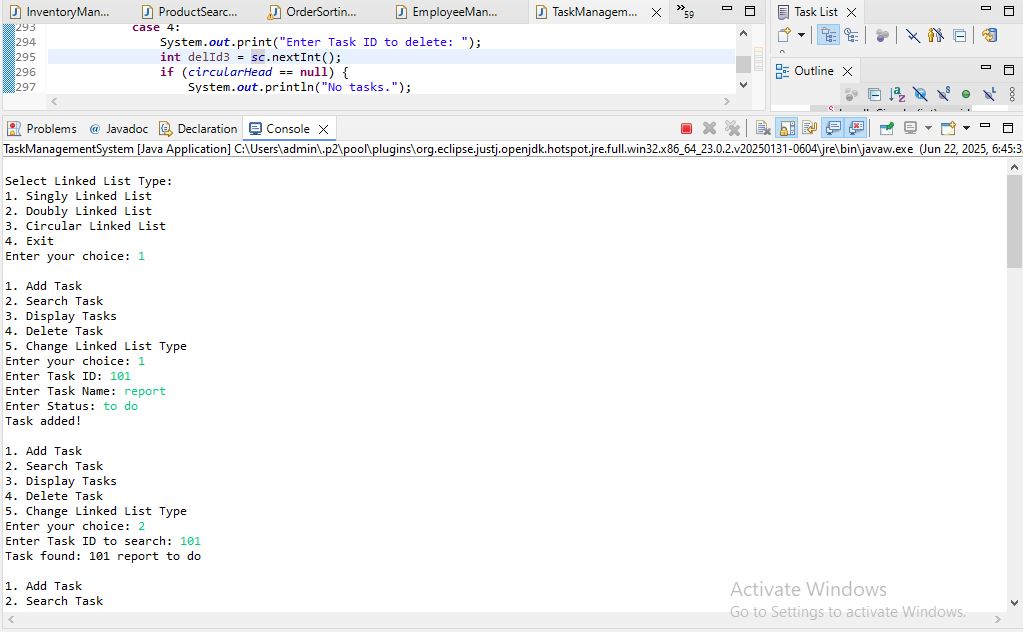
System.***out***.println("Task not found!");

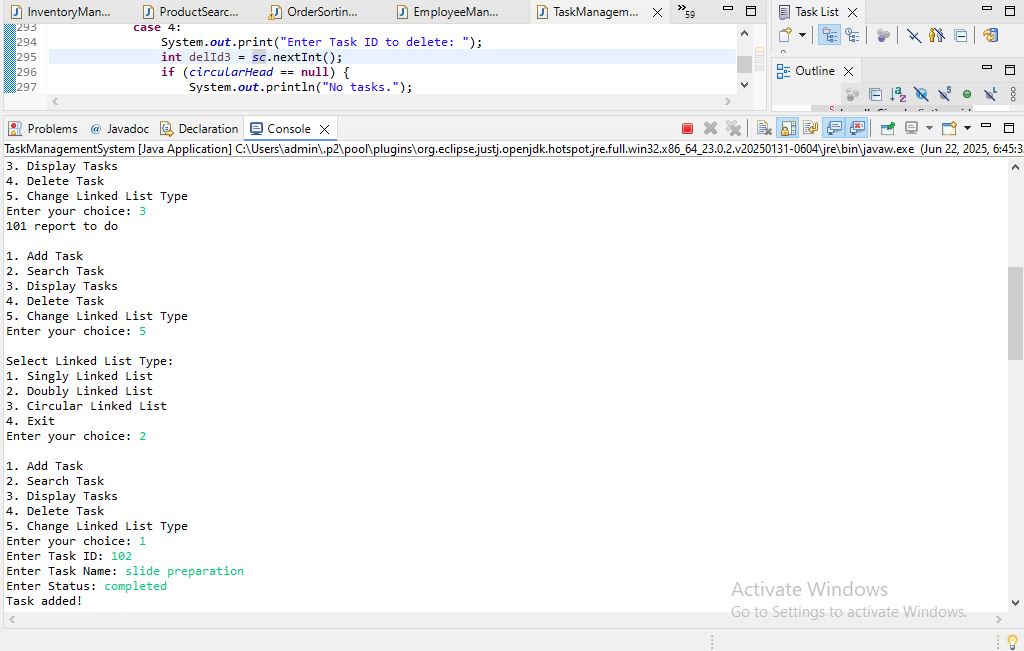
**break**;

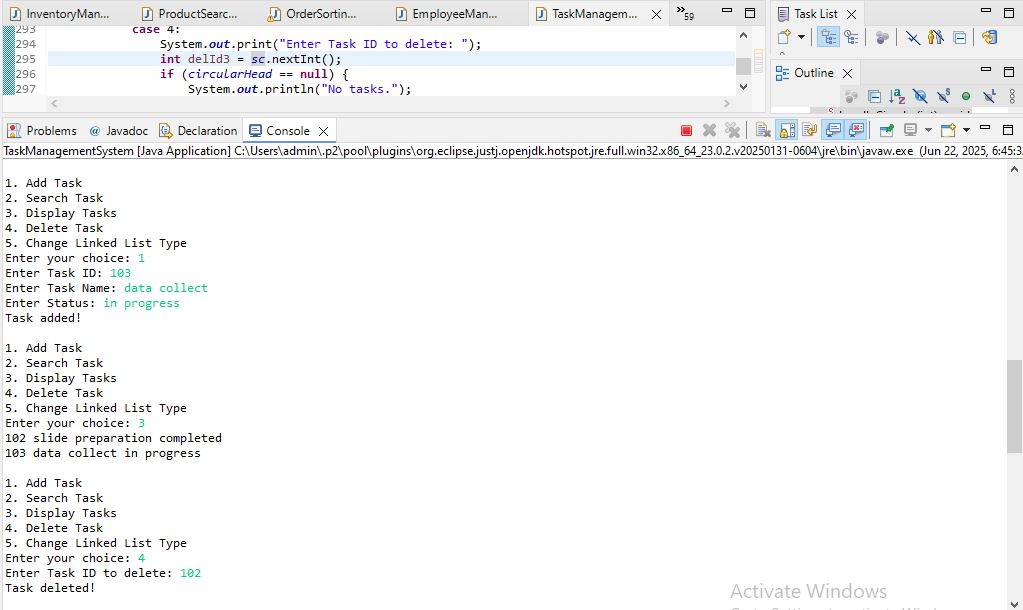
}

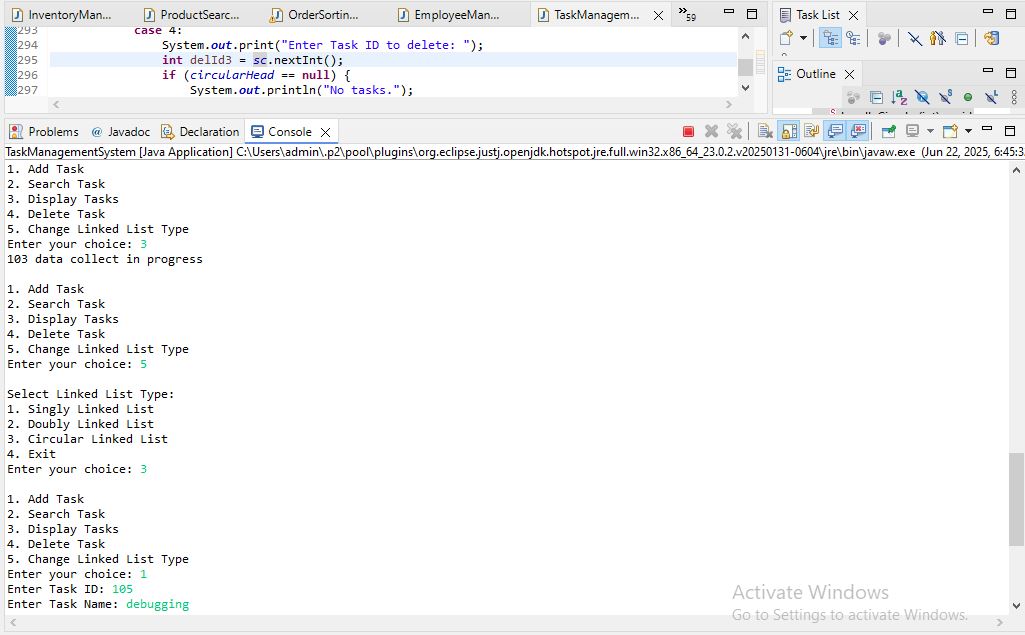
}

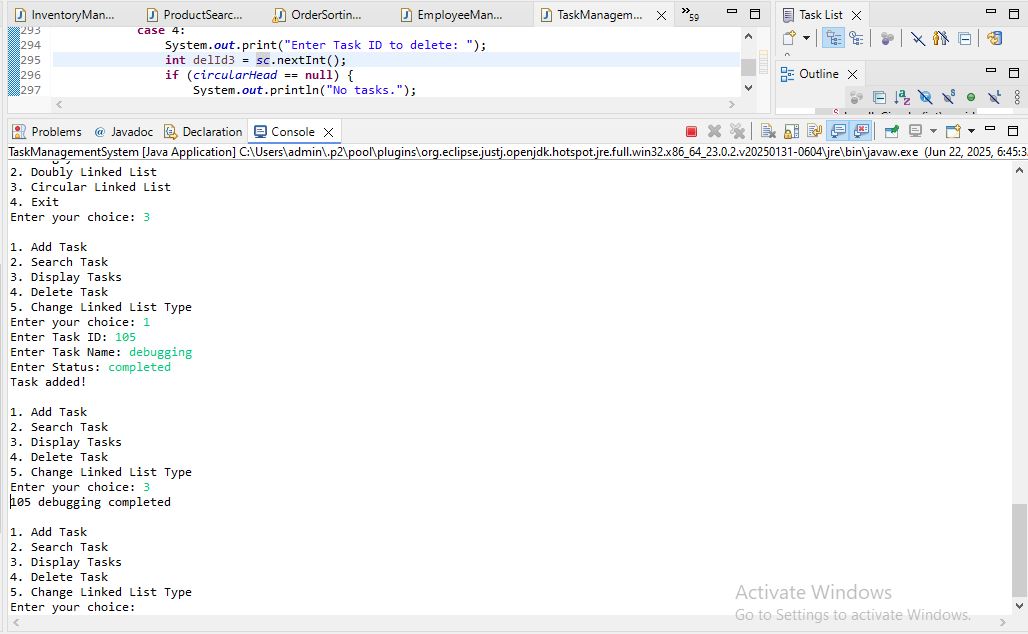
}











We are developing a **Task Management System** where the user can choose to store tasks using:

* **Singly Linked List**
* **Doubly Linked List**
* **Circular Linked List**

This system allows users to:

* Add tasks
* Search tasks
* Display all tasks
* Delete tasks

## ****Linked List Concepts****

### A. ****Singly Linked List****

* Every node has:
  + Data: taskId, taskName, status
  + Pointer to the next node
* The last node points to null.

#### ****Advantages:****

* Dynamic memory allocation
* Easy insertion and deletion at beginning

#### ****Disadvantages:****

* Cannot traverse backwards
* Deleting specific node requires traversal

### B. ****Doubly Linked List****

* Every node has:
  + Data: taskId, taskName, status
  + Pointer to the next node (next)
  + Pointer to the previous node (prev)

#### ****Advantages:****

* Can traverse forward and backward
* Deletion is easier since we have reference to previous node

#### ****Disadvantages:****

* Extra memory needed for prev pointer
* Slightly more complex insertion/deletion

### C. ****Circular Linked List****

* Similar to singly linked list but:
  + Last node’s next points back to head.
* No node has null pointer.

#### ****Advantages:****

* Continuous traversal possible
* Useful for round-robin scheduling, buffering systems

#### ****Disadvantages:****

* Special care needed in deletion & insertion
* Termination condition must be carefully handled (since no null)

## ****Setup****

We created 3 classes for node representation:

| **Linked List Type** | **Class Name** |
| --- | --- |
| Singly Linked List | SinglyTask |
| Doubly Linked List | DoublyTask |
| Circular Linked List | CircularTask |

Each class has:

* taskId (int)
* taskName (String)
* status (String)

The main class TaskManagementSystem handles all the operations.

## ****Implementation****

The program works as follows:

1 User first selects Linked List type  
2Then user performs operations:

* Add Task
* Search Task
* Display Tasks
* Delete Task

### ****Operations Implemented for All 3 Types:****

| **Operation** | **Description** |
| --- | --- |
| **Add Task** | Insert new task at end of list |
| **Search Task** | Search for taskId |
| **Traverse Tasks** | Display all tasks |
| **Delete Task** | Delete task by taskId |

## ****Time Complexity Analysis****

| **Operation** | **Time Complexity** | **Explanation** |
| --- | --- | --- |
| **Add** | O(n) | Traverse to end |
| **Search** | O(n) | Linear search |
| **Traverse** | O(n) | Visit each node |
| **Delete** | O(n) | Search + remove |

Complexity is same for all 3 types because of linear structure.

## ****Why Linked List instead of Array?****

| **Array** | **Linked List** |
| --- | --- |
| Fixed size | Dynamic size |
| Insertion/Deletion costly (shift needed) | Easy insertion/deletion |
| Random access possible (O(1)) | Only sequential access (O(n)) |
| Memory allocation contiguous | Non-contiguous allocation |

* **Linked Lists** are more suitable when:
  + The number of tasks is not fixed.
  + Frequent insertions and deletions are expected.

## ****When to use each Linked List?****

| **Linked List Type** | **Use Case** |
| --- | --- |
| **Singly Linked List** | Simple dynamic lists |
| **Doubly Linked List** | When backward traversal is needed |
| **Circular Linked List** | When round-robin style operations or endless cycles are needed |

**Exercise 6: Library Management System**

**Library Management System.java**

**package** mypackage;

**import** java.util.\*;

**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;

}

}

**public** **class** LibraryManagementSystem {

**static** Scanner *sc* = **new** Scanner(System.***in***);

**static** ArrayList<Book> *books* = **new** ArrayList<>();

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

**while** (**true**) {

System.***out***.println("\nLibrary Management System");

System.***out***.println("1. Add Book");

System.***out***.println("2. Linear Search by Title");

System.***out***.println("3. Binary Search by Title");

System.***out***.println("4. Display All Books");

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

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

**int** choice = *sc*.nextInt();

*sc*.nextLine(); // consume newline

**switch** (choice) {

**case** 1:

*addBook*();

**break**;

**case** 2:

*linearSearch*();

**break**;

**case** 3:

*binarySearch*();

**break**;

**case** 4:

*displayBooks*();

**break**;

**case** 5:

**return**;

**default**:

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

}

}

}

**public** **static** **void** addBook() {

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

**int** id = *sc*.nextInt();

*sc*.nextLine();

System.***out***.print("Enter Book Title: ");

String title = *sc*.nextLine();

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

String author = *sc*.nextLine();

*books*.add(**new** Book(id, title, author));

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

}

**public** **static** **void** linearSearch() {

**if** (*books*.isEmpty()) {

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

**return**;

}

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

String searchTitle = *sc*.nextLine();

**boolean** found = **false**;

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

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

System.***out***.println("Book found: " + b.bookId + ", " + b.title + ", " + b.author);

found = **true**;

}

}

**if** (!found) {

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

}

}

**public** **static** **void** binarySearch() {

**if** (*books*.isEmpty()) {

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

**return**;

}

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

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

String searchTitle = *sc*.nextLine();

**int** left = 0;

**int** right = *books*.size() - 1;

**boolean** found = **false**;

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

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

String midTitle = *books*.get(mid).title.toLowerCase();

**int** cmp = searchTitle.toLowerCase().compareTo(midTitle);

**if** (cmp == 0) {

System.***out***.println("Book found: " + *books*.get(mid).bookId + ", " + *books*.get(mid).title + ", " + *books*.get(mid).author);

found = **true**;

**break**;

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

right = mid - 1;

} **else** {

left = mid + 1;

}

}

**if** (!found) {

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

}

}

**public** **static** **void** displayBooks() {

**if** (*books*.isEmpty()) {

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

**return**;

}

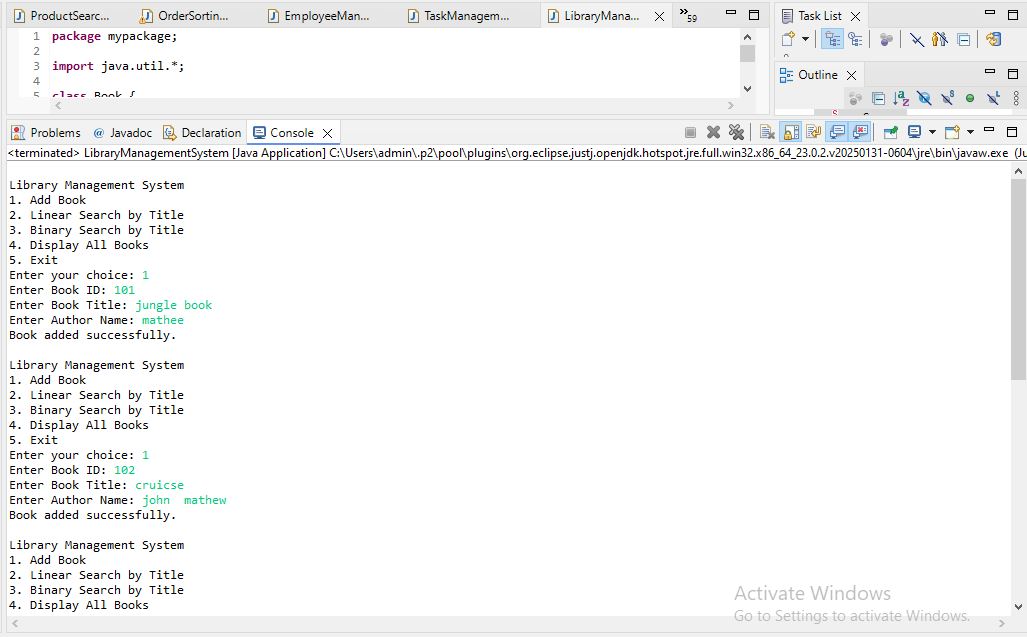
**for** (Book b : *books*) {

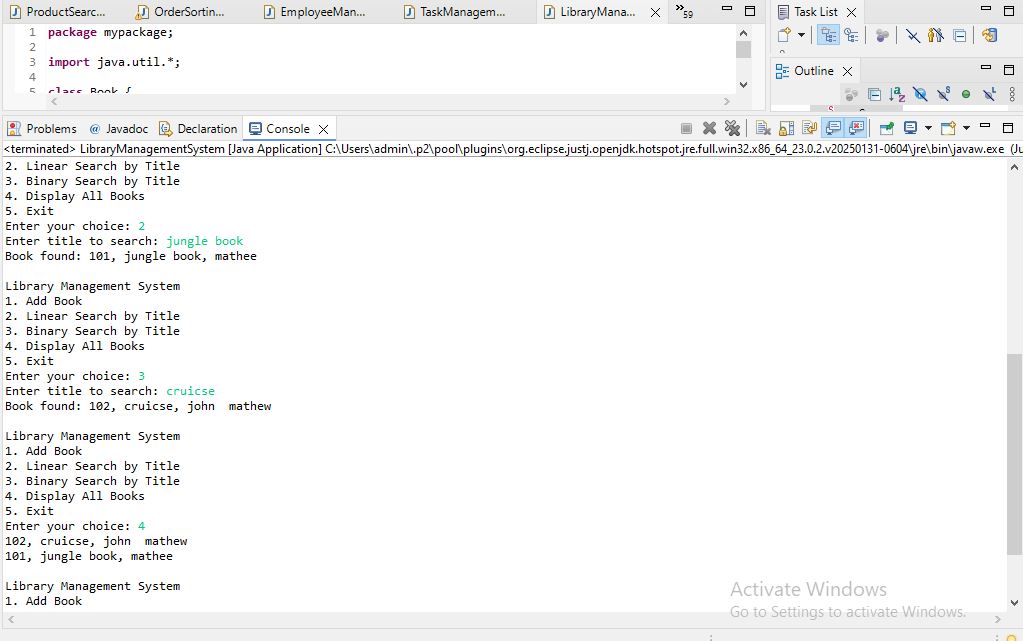
System.***out***.println(b.bookId + ", " + b.title + ", " + b.author);

}

}

}





**Linear Search:**

Linear search is a simple search algorithm that sequentially checks each element of the list until a match is found or the whole list has been searched. It does not require the list to be sorted.

**Binary Search:**

Binary search is a more efficient search algorithm that works only on sorted lists. It repeatedly divides the search interval in half. If the search key is less than the middle element, it continues the search in the left half, otherwise in the right half.

**Setup:**

We create a class Book with the following attributes:

* bookId (integer)
* title (string)
* author (string)

The system allows adding books to a list. The list can be searched using both linear and binary search.

1. Implementation:

**Linear Search:**

* Iterate through each book in the list.
* Compare the title of each book with the search query.
* If a match is found, return the book details.
* If no match is found after traversing the entire list, return "Book not found."

**Binary Search:**

* First, sort the list of books based on title.
* Set left and right pointers at the start and end of the list.
* Calculate the mid index.
* Compare the mid element's title with the search query.
* If it matches, return the book.
* If the search query is smaller, search in the left half.
* If the search query is larger, search in the right half.
* Repeat until the book is found or the search space is empty.

1. Analysis:

**Time Complexity:**

**Linear Search:**

* Best case: O(1) (first element matches)
* Average case: O(n)
* Worst case: O(n)

**Binary Search:**

* Best case: O(1) (middle element matches)
* Average case: O(log n)
* Worst case: O(log n)

**Comparison:**

Linear search works on both sorted and unsorted data but is inefficient for large datasets. Binary search is much faster but requires the data to be sorted beforehand.

When to use:

* Use linear search when the dataset is small or unsorted.
* Use binary search when the dataset is large and sorted.

**Exercise 7: Financial Forecasting**

**Financial Forecasting.java**

**package** mypackage;

**import** java.util.Scanner;

**public** **class** FinancialForecasting {

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

// Base case

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

**return** currentValue;

}

// Recursive call

**return** *predictFutureValue*(currentValue \* (1 + growthRate / 100), growthRate, years - 1);

}

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

Scanner sc = **new** Scanner(System.***in***);

System.***out***.println("Financial Forecasting Tool");

System.***out***.print("Enter current value (in Indian Rupees): ");

**double** currentValue = sc.nextDouble();

System.***out***.print("Enter annual growth rate (percentage): ");

**double** growthRate = sc.nextDouble();

System.***out***.print("Enter number of years: ");

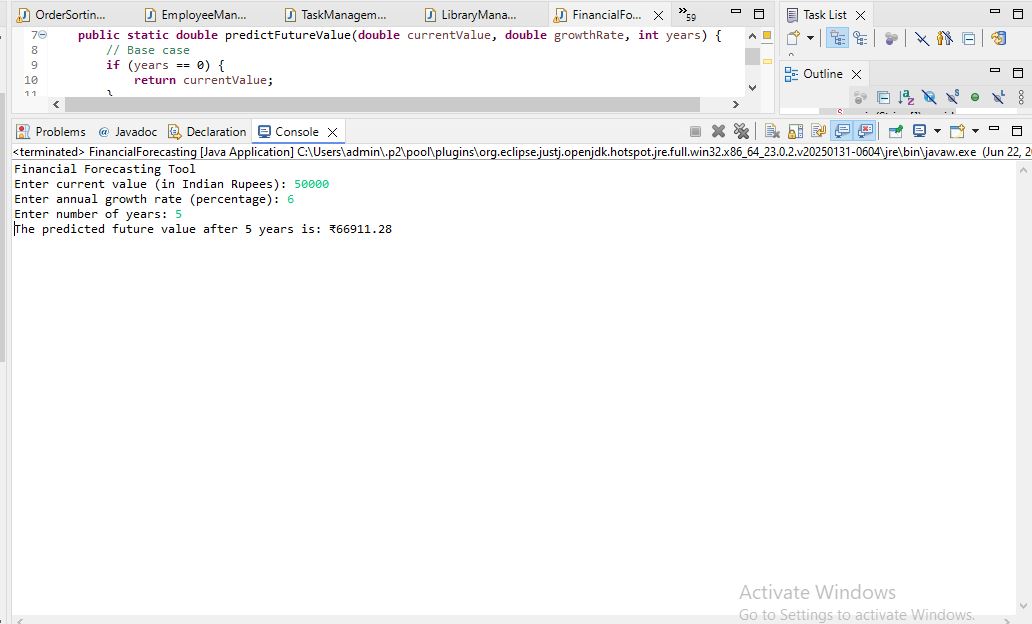
**int** years = sc.nextInt();

**double** futureValue = *predictFutureValue*(currentValue, growthRate, years);

System.***out***.printf("The predicted future value after %d years is: ₹%.2f\n", years, futureValue);

}

}



**Financial Forecasting using Recursion**

We are developing a financial forecasting tool that predicts future financial values based on past data and growth rates. The problem will be solved using recursion.

**Understand Recursive Algorithms:**

Recursion is a programming technique where a method calls itself to solve smaller instances of the same problem. Each recursive call works with a smaller input until it reaches a base case that stops the recursion.

Recursion simplifies problems that have repetitive subproblems, such as financial forecasting, where the same calculation is applied year after year.

In financial forecasting, each year’s value depends on the previous year’s value. This makes recursion a natural approach for calculating future values.

**Setup:**

We create a method called predictFutureValue that accepts:

* currentValue: The present financial amount.
* growthRate: The annual growth rate in percentage.
* years: The number of years to predict into the future.

1. Implementation:

**The recursive method works as follows:**

* If the number of years is zero, return the current value (base case).
* Otherwise, calculate the value for the next year by multiplying the current value by (1 + growthRate / 100), and then call the function recursively for (years - 1).

This continues until all years are processed.

1. Analysis:

**Time Complexity:**

* Each recursive call reduces the problem size by one year.
* The total number of calls equals the number of years.
* Therefore, time complexity is O(n), where n is the number of years.

**Optimization:**

* For very large values of years, recursion may cause stack overflow due to deep recursion.
* This can be optimized using iterative solutions or dynamic programming.
* Alternatively, the compound interest formula can directly compute the result without recursion:

Future Value **= Current Value × (1 + growthRate / 100) ^ years**

* This direct formula has constant time complexity O(1), which is highly efficient.

**Example:**

Financial Forecasting Tool

Enter current value (in Indian Rupees): 100000

Enter annual growth rate (percentage): 8

Enter number of years: 5

**The predicted future value after 5 years is: ₹146933.28**

Year 1: 100000 \* (1 + 8/100) = 108000.0

Year 2: 108000.0 \* (1 + 8/100) = 116640.0

Year 3: 116640.0 \* (1 + 8/100) = 125971.20

Year 4: 125971.20 \* (1 + 8/100) = 136048.90

Year 5: 136048.90 \* (1 + 8/100) = 146933.28