**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.

**Product Class:  
  
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 productId + " - " + productName + " - Qty: " + quantity + " - Price: $" + price;**

**}**

**}  
  
Inventory Class Using HashMap  
  
import java.util.HashMap;**

**public class Inventory {**

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

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

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

**}**

**public void updateProduct(int id, Product updatedProduct) {**

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

**inventoryMap.put(id, updatedProduct);**

**}**

**}**

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

**inventoryMap.remove(id);**

**}**

**public void displayInventory() {**

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

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

**}**

**}**

**}  
  
Main Program  
  
public class Main {**

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

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

**Product p1 = new Product(101, "Laptop", 10, 75000);**

**Product p2 = new Product(102, "Mouse", 50, 500);**

**inv.addProduct(p1);**

**inv.addProduct(p2);**

**inv.displayInventory();**

**inv.updateProduct(102, new Product(102, "Mouse", 60, 500));**

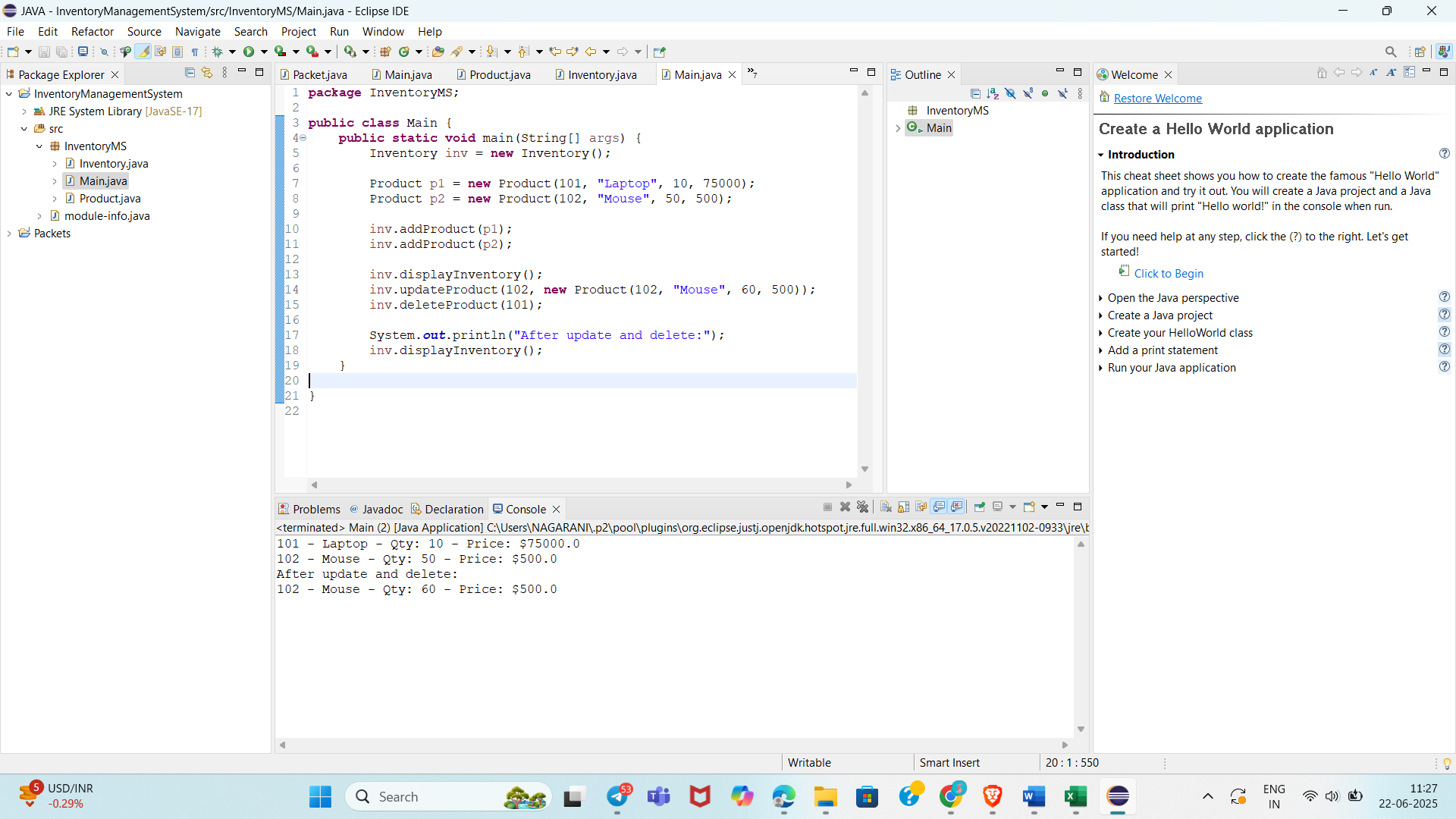
**inv.deleteProduct(101);**

**System.out.println("After update and delete:");**

**inv.displayInventory();**

**}**

**}**



**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.

**Product Class:  
  
public class Product {**

**int productId;**

**String productName;**

**String category;**

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

**this.productId = id;**

**this.productName = name;**

**this.category = category;**

**}**

**@Override**

**public String toString() {**

**return productId + " - " + productName + " (" + category + ")";**

**}**

**}**

**Linear Search:  
  
public class SearchUtil {**

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

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

**if (p.productName.equalsIgnoreCase(target)) {**

**return p;**

**}**

**}**

**return null;**

**}**

**}  
  
Binary Search  
  
import java.util.Arrays;**

**import java.util.Comparator;**

**public class SearchUtil {**

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

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

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

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

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

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

**if (cmp == 0) return products[mid];**

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

**else right = mid - 1;**

**}**

**return null;**

**}**

**}  
  
Example Usage:  
public class Main {**

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

**Product[] catalog = {**

**new Product(201, "Laptop", "Electronics"),**

**new Product(202, "Shoes", "Fashion"),**

**new Product(203, "Phone", "Electronics"),**

**new Product(204, "Book", "Stationery")**

**};**

**Product result1 = SearchUtil.linearSearch(catalog, "Shoes");**

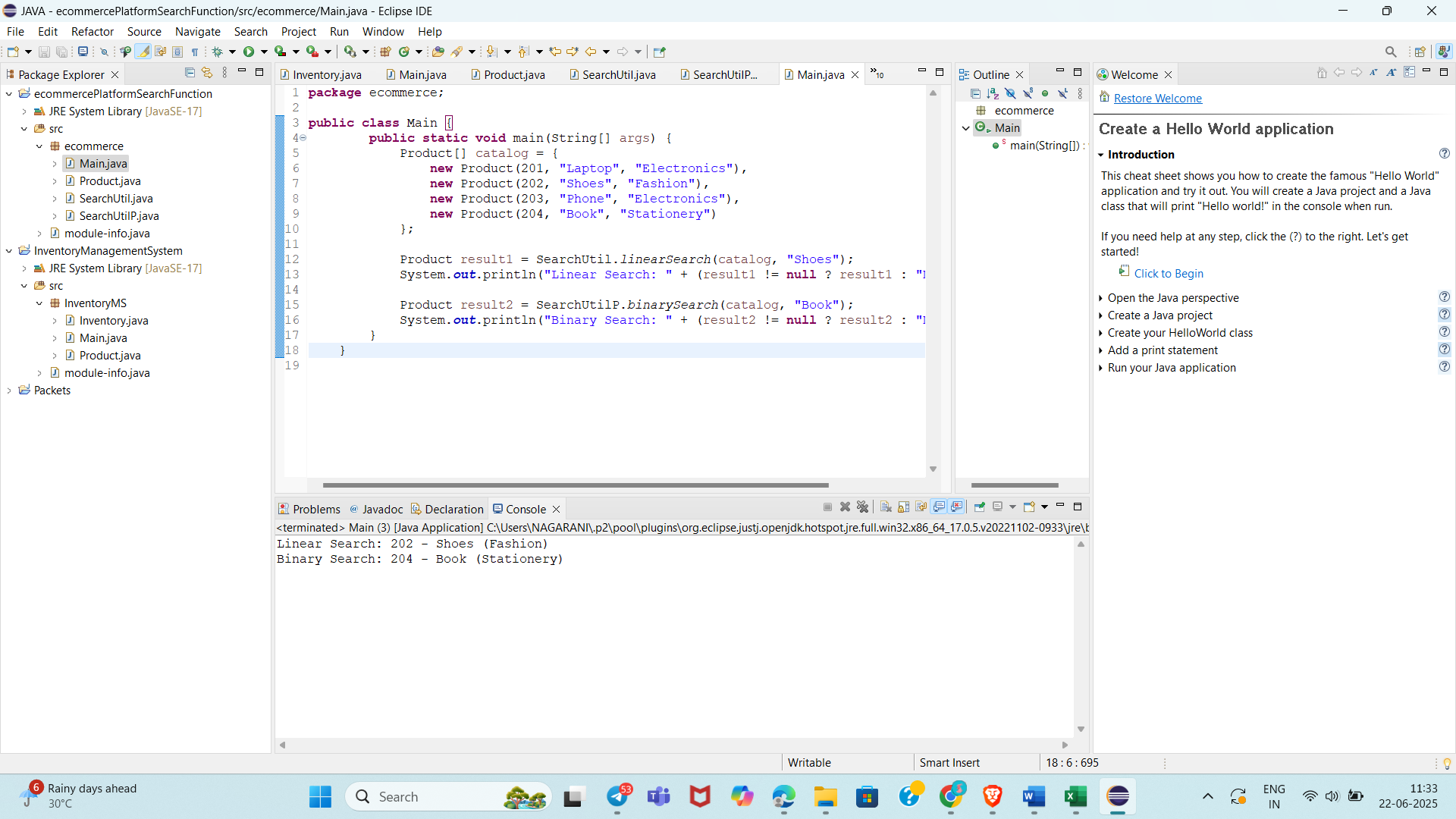
**System.out.println("Linear Search: " + (result1 != null ? result1 : "Not Found"));**

**Product result2 = SearchUtil.binarySearch(catalog, "Book");**

**System.out.println("Binary Search: " + (result2 != null ? result2 : "Not Found"));**

**}**

**}**



**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.

**Order Class**

**public class Order {**

**int orderId;**

**String customerName;**

**double totalPrice;**

**public Order(int orderId, String name, double totalPrice) {**

**this.orderId = orderId;**

**this.customerName = name;**

**this.totalPrice = totalPrice;**

**}**

**@Override**

**public String toString() {**

**return orderId + " - " + customerName + " - $" + totalPrice;**

**}**

**}  
  
Bubble Sort  
public class SortUtil {**

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

**int n = orders.length;**

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

**boolean swapped = false;**

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

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

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

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

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

**swapped = true;**

**}**

**}**

**if (!swapped) break;**

**}**

**}**

**}  
Quick Sort  
public class SortUtil {**

**public static void quickSort(Order[] orders, int low, int high) {**

**if (low < high) {**

**int pivotIndex = partition(orders, low, high);**

**quickSort(orders, low, pivotIndex - 1);**

**quickSort(orders, pivotIndex + 1, high);**

**}**

**}**

**private 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;**

**}**

**}  
  
Sample Test  
public class Main {**

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

**Order[] orders = {**

**new Order(301, "Alice", 500.0),**

**new Order(302, "Bob", 1200.0),**

**new Order(303, "Charlie", 300.0)**

**};**

**// Bubble Sort**

**SortUtil.bubbleSort(orders);**

**System.out.println("Bubble Sorted:");**

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

**// Quick Sort (reset data)**

**orders = new Order[] {**

**new Order(301, "Alice", 500.0),**

**new Order(302, "Bob", 1200.0),**

**new Order(303, "Charlie", 300.0)**

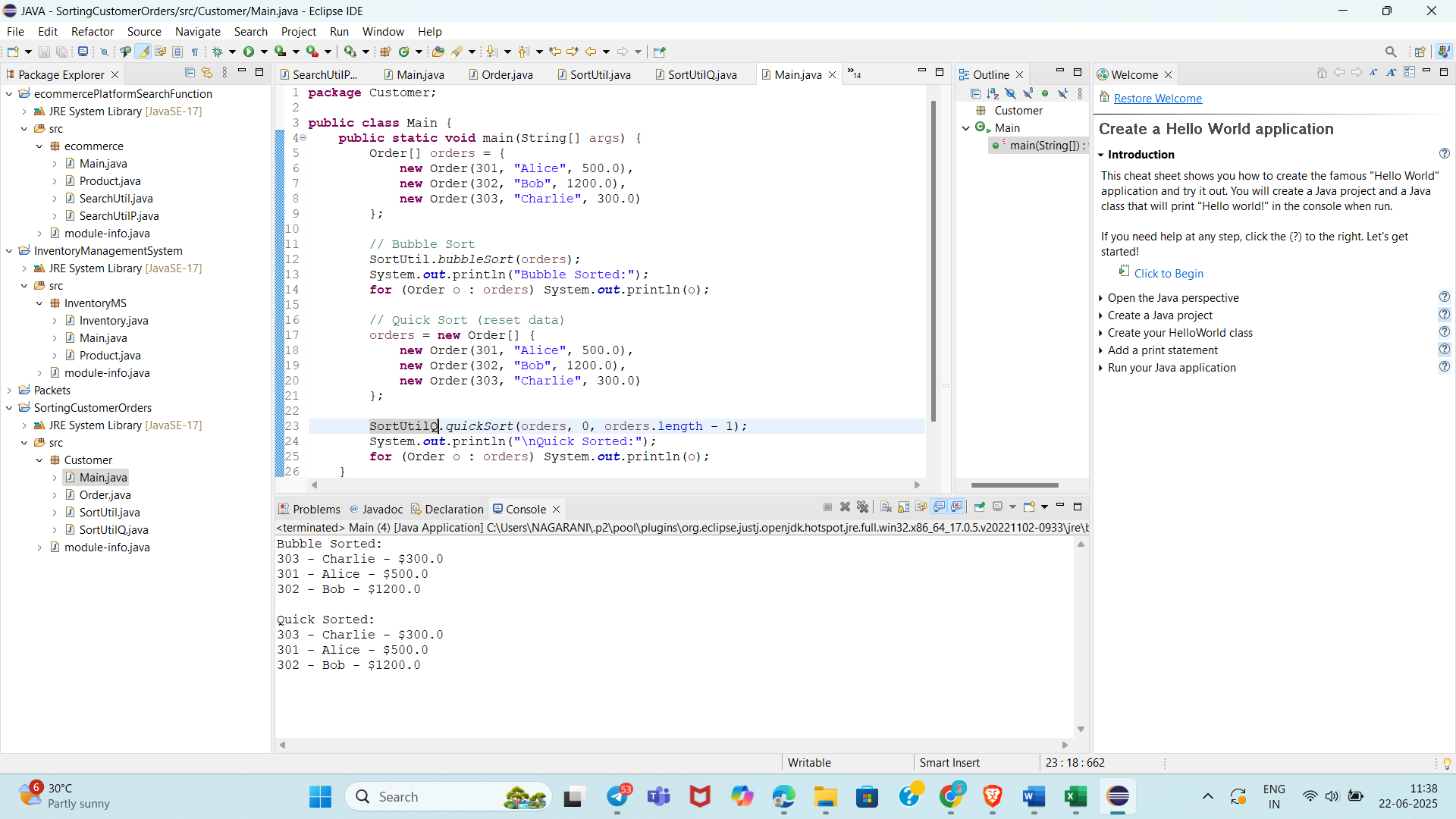
**};**

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

**System.out.println("\nQuick Sorted:");**

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

**}**

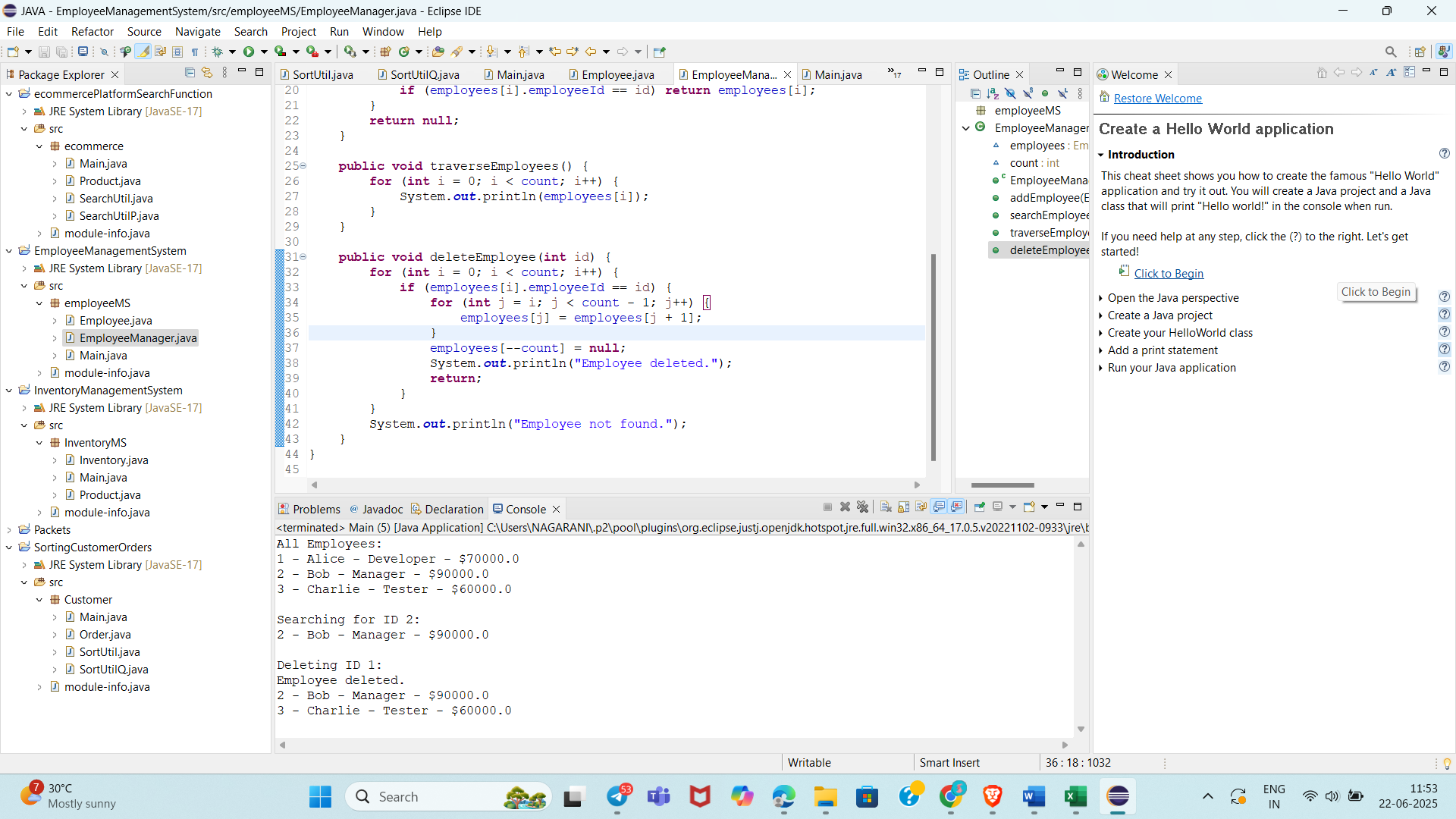
**}****Exercise 4: Employee Management System**

**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.



**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.

Employee Class

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 employeeId + " - " + name + " - " + position + " - $" + salary;

}

}

Employee Management Using Array

public class EmployeeManager {

Employee[] employees;

int count = 0;

public EmployeeManager(int capacity) {

employees = new Employee[capacity];

}

public void addEmployee(Employee emp) {

if (count < employees.length) {

employees[count++] = emp;

} else {

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

}

}

public Employee searchEmployee(int id) {

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

if (employees[i].employeeId == id) return employees[i];

}

return null;

}

public void traverseEmployees() {

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

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

}

}

public void deleteEmployee(int id) {

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

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

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

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

}

employees[--count] = null;

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

return;

}

}

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

}

}

Main class  
public class Main {

public static void main(String[] args) {

EmployeeManager manager = new EmployeeManager(5);

manager.addEmployee(new Employee(1, "Alice", "Developer", 70000));

manager.addEmployee(new Employee(2, "Bob", "Manager", 90000));

manager.addEmployee(new Employee(3, "Charlie", "Tester", 60000));

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

manager.traverseEmployees();

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

Employee emp = manager.searchEmployee(2);

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

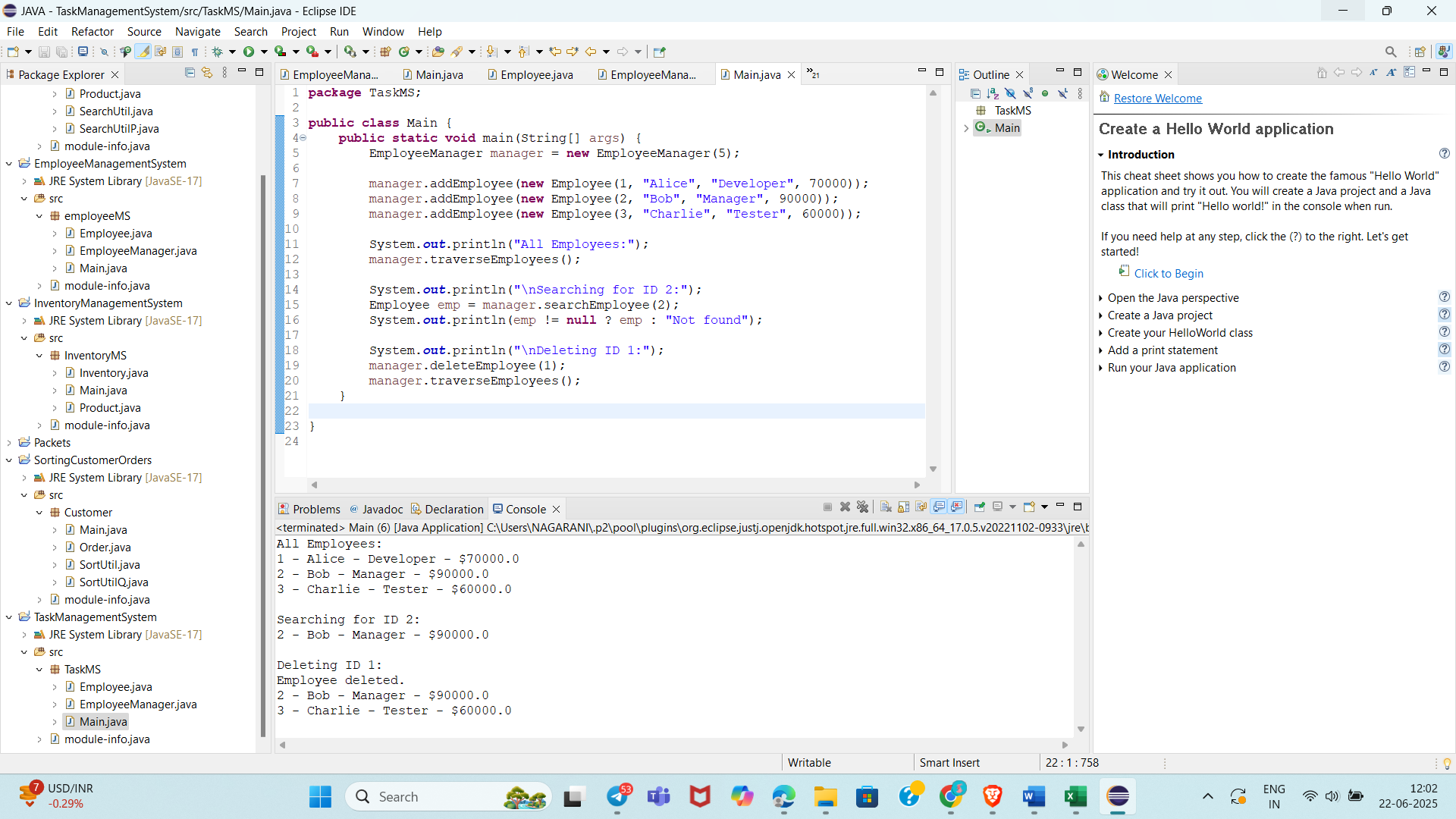
System.out.println("\nDeleting ID 1:");

manager.deleteEmployee(1);

manager.traverseEmployees();

}

}



**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.

**Task Class**

**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 taskId + " - " + taskName + " [" + status + "]";**

**}**

**}**

**Implementation: Singly Linked List**

**public class TaskManager {**

**Task head;**

**// Add task at the end**

**public void addTask(Task newTask) {**

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

**head = newTask;**

**} else {**

**Task current = head;**

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

**current = current.next;**

**}**

**current.next = newTask;**

**}**

**}**

**// Search task by ID**

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

**Task current = head;**

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

**if (current.taskId == id) return current;**

**current = current.next;**

**}**

**return null;**

**}**

**// Traverse tasks**

**public void traverseTasks() {**

**Task current = head;**

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

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

**current = current.next;**

**}**

**}**

**// Delete task by ID**

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

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

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

**head = head.next;**

**return;**

**}**

**Task current = head;**

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

**if (current.next.taskId == id) {**

**current.next = current.next.next;**

**return;**

**}**

**current = current.next;**

**}**

**}**

**}**

**Main class  
public class Main {**

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

**TaskManager manager = new TaskManager();**

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

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

**manager.addTask(new Task(3, "Code Review", "Pending"));**

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

**manager.traverseTasks();**

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

**Task found = manager.searchTask(2);**

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

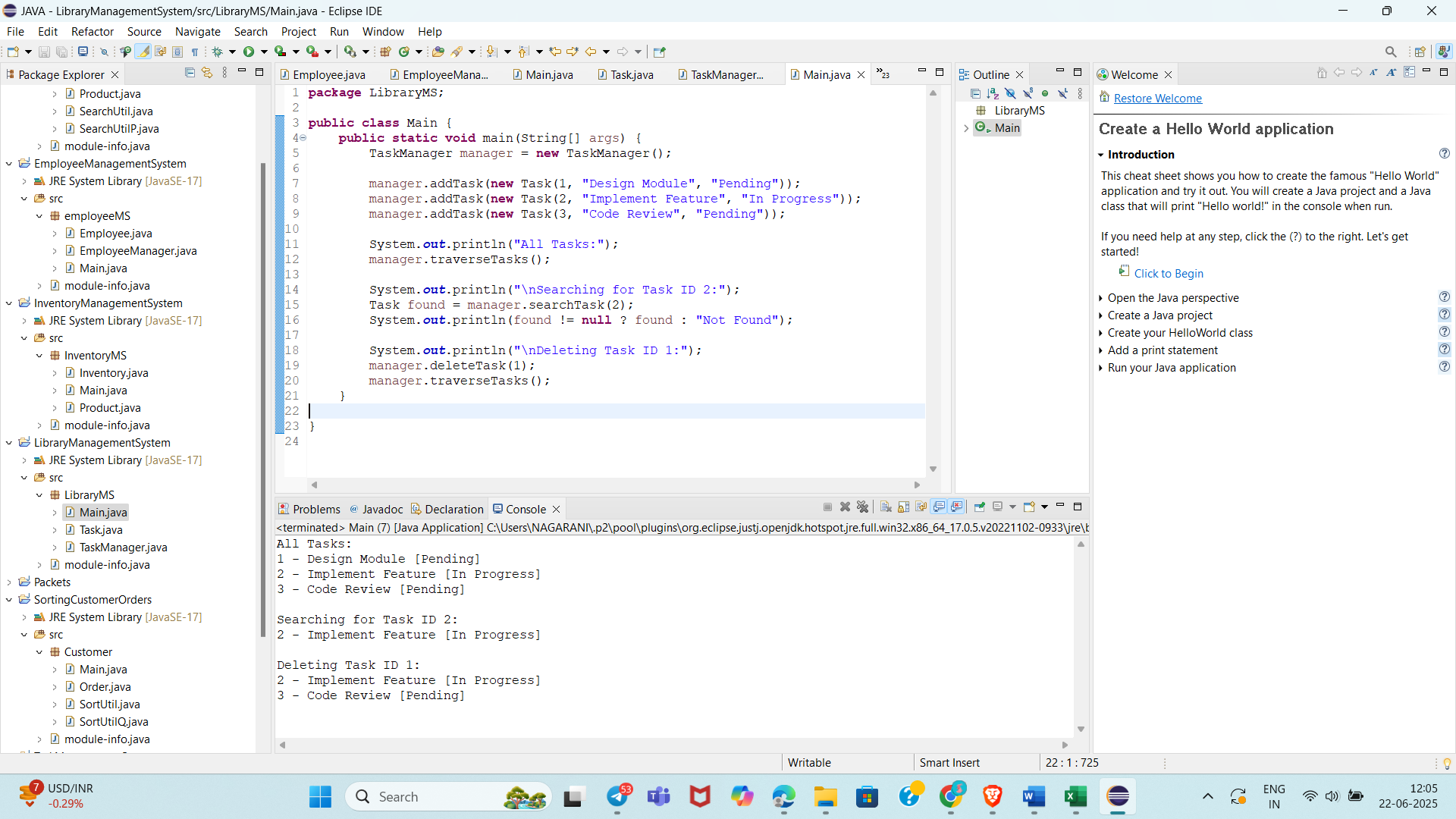
**System.out.println("\nDeleting Task ID 1:");**

**manager.deleteTask(1);**

**manager.traverseTasks();**

**}**

**}**



**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.

public class Forecast {

// Recursive method to predict future value

public static double predictFutureValue(double presentValue, double rate, int years) {

if (years == 0) return presentValue;

return predictFutureValue(presentValue \* (1 + rate), rate, years - 1);

}

public static void main(String[] args) {

double currentValue = 10000;

double annualGrowthRate = 0.08; // 8%

int futureYears = 5;

double futureValue = predictFutureValue(currentValue, annualGrowthRate, futureYears);

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

}

}

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

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

value \*= (1 + rate);

}

return value;

}

