# Exercise 1: Inventory Management System

## Implementation:

• Define a Product class with attributes: productId, productName, quantity, and price.  
• Use HashMap<String, Product> to store products.  
• Implement methods to add, update, delete, and display products.

### Product.java

package com.inventory.system;  
  
public class Product {  
 private String productId;  
 private String productName;  
 private int quantity;  
 private double price;  
  
 public Product(String productId, String productName, int quantity, double price) {  
 this.productId = productId;  
 this.productName = productName;  
 this.quantity = quantity;  
 this.price = price;  
 }  
  
 public String getProductId() { return productId; }  
 public String getProductName() { return productName; }  
 public int getQuantity() { return quantity; }  
 public double getPrice() { return price; }  
  
 public void setProductName(String name) { this.productName = name; }  
 public void setQuantity(int qty) { this.quantity = qty; }  
 public void setPrice(double price) { this.price = price; }  
  
 public String toString() {  
 return "[" + productId + "] " + productName + " - Qty: " + quantity + ", Price: $" + price;  
 }  
}

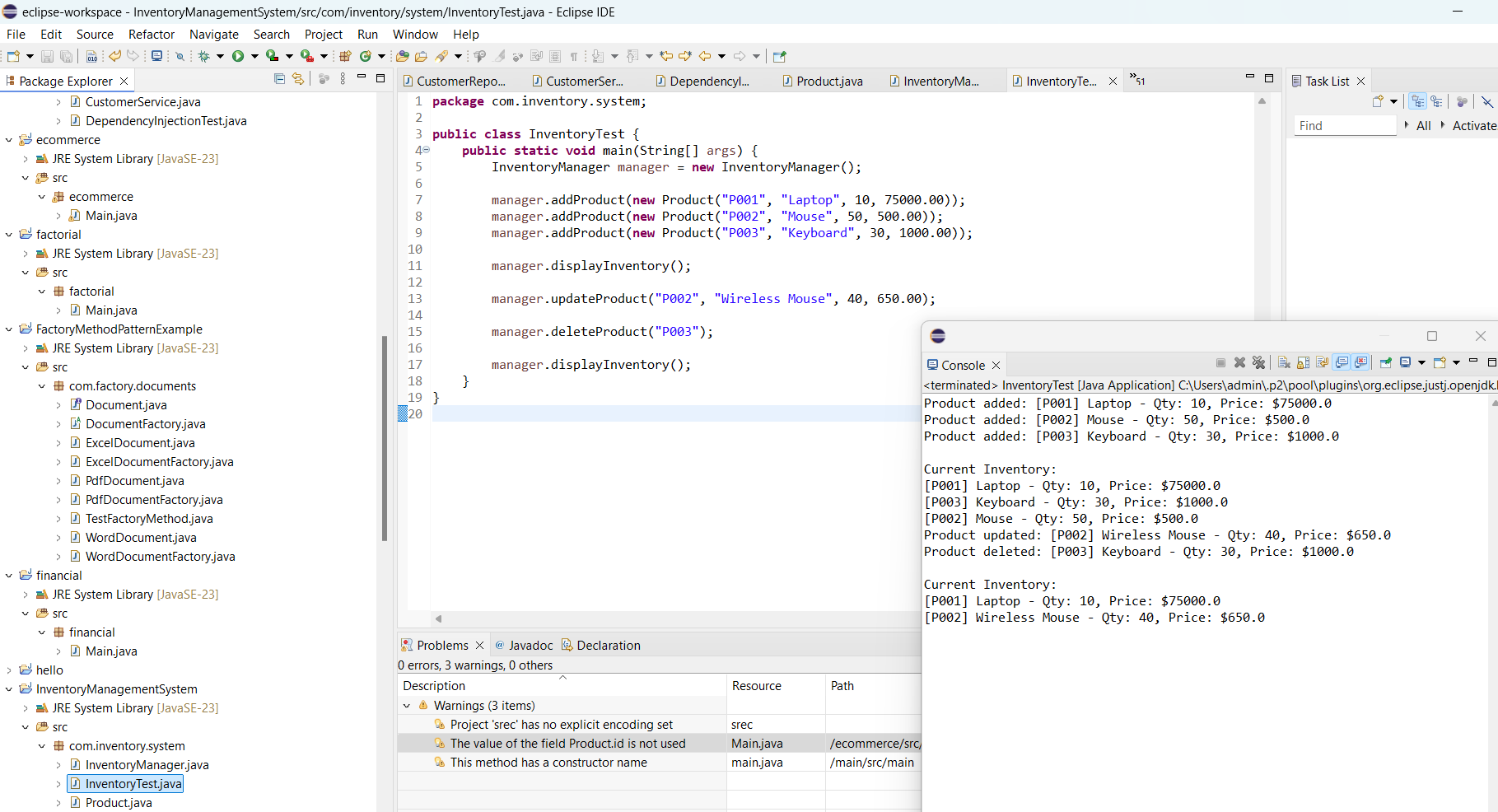
### InventoryManager.java

package com.inventory.system;  
  
import java.util.HashMap;  
import java.util.Map;  
  
public class InventoryManager {  
 private Map<String, Product> inventory;  
  
 public InventoryManager() {  
 inventory = new HashMap<>();  
 }  
  
 public void addProduct(Product product) {  
 inventory.put(product.getProductId(), product);  
 System.out.println("Product added: " + product);  
 }  
  
 public void updateProduct(String productId, String newName, int newQty, double newPrice) {  
 Product product = inventory.get(productId);  
 if (product != null) {  
 product.setProductName(newName);  
 product.setQuantity(newQty);  
 product.setPrice(newPrice);  
 System.out.println("Product updated: " + product);  
 } else {  
 System.out.println("Product ID not found.");  
 }  
 }  
  
 public void deleteProduct(String productId) {  
 Product removed = inventory.remove(productId);  
 if (removed != null) {  
 System.out.println("Product deleted: " + removed);  
 } else {  
 System.out.println("Product ID not found.");  
 }  
 }  
  
 public void displayInventory() {  
 System.out.println("\nCurrent Inventory:");  
 for (Product product : inventory.values()) {  
 System.out.println(product);  
 }  
 }  
}

### InventoryTest.java

package com.inventory.system;  
  
public class InventoryTest {  
 public static void main(String[] args) {  
 InventoryManager manager = new InventoryManager();  
  
 manager.addProduct(new Product("P001", "Laptop", 10, 75000.00));  
 manager.addProduct(new Product("P002", "Mouse", 50, 500.00));  
 manager.addProduct(new Product("P003", "Keyboard", 30, 1000.00));  
  
 manager.displayInventory();  
  
 manager.updateProduct("P002", "Wireless Mouse", 40, 650.00);  
 manager.deleteProduct("P003");  
  
 manager.displayInventory();  
 }  
}

**OUTPUT:**

****

## Analysis:

• Time Complexity:  
 - Add: O(1)  
 - Update: O(1)  
 - Delete: O(1)  
• Optimization:  
 - For faster operations, HashMap is ideal.  
 - For sorted data or range queries, TreeMap is better (O(log n)).  
 - For persistent storage, integrate a database system like MySQL.

# Exercise 2: E-commerce Platform Search Function

## Implementation:

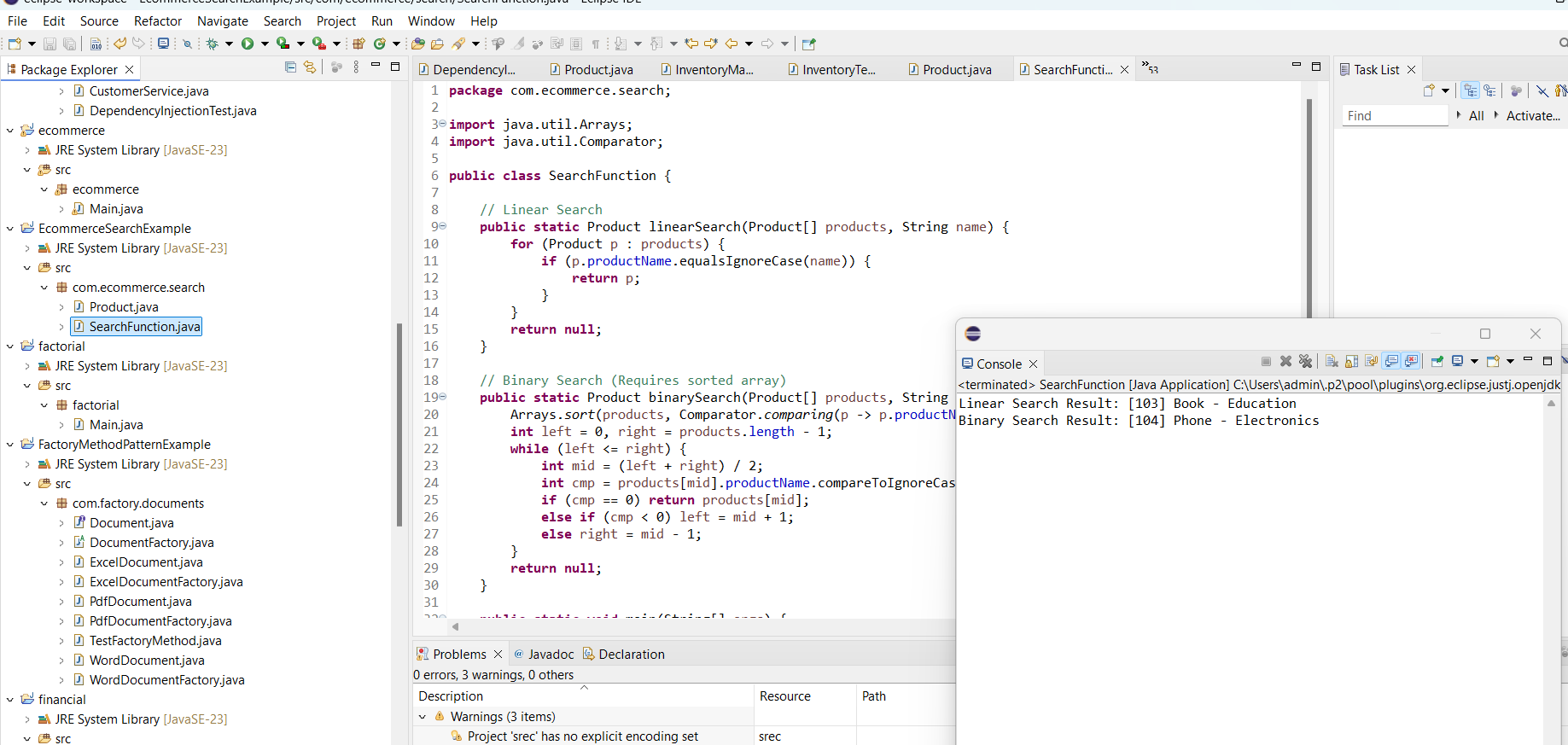
### Product.java

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 "[" + productId + "] " + productName + " - " + category;  
 }  
}

### SearchFunction.java

import java.util.Arrays;  
import java.util.Comparator;  
  
public class SearchFunction {  
  
 // Linear Search  
 public static Product linearSearch(Product[] products, String name) {  
 for (Product p : products) {  
 if (p.productName.equalsIgnoreCase(name)) {  
 return p;  
 }  
 }  
 return null;  
 }  
  
 // Binary Search (Requires sorted array)  
 public static Product binarySearch(Product[] products, String name) {  
 Arrays.sort(products, Comparator.comparing(p -> p.productName.toLowerCase()));  
 int left = 0, right = products.length - 1;  
 while (left <= right) {  
 int mid = (left + right) / 2;  
 int cmp = products[mid].productName.compareToIgnoreCase(name);  
 if (cmp == 0) return products[mid];  
 else if (cmp < 0) left = mid + 1;  
 else right = mid - 1;  
 }  
 return null;  
 }  
 public static void main(String[] args) {  
 Product[] products = {  
 new Product(101, "Laptop", "Electronics"),  
 new Product(102, "Shirt", "Clothing"),  
 new Product(103, "Book", "Education"),  
 new Product(104, "Phone", "Electronics")  
 };  
 // Linear Search  
 Product result1 = linearSearch(products, "Book");  
 System.out.println("Linear Search Result: " + (result1 != null ? result1 : "Not found"));  
  
 // Binary Search  
 Product result2 = binarySearch(products, "Phone");  
 System.out.println("Binary Search Result: " + (result2 != null ? result2 : "Not found"));  
 }  
}

**OUTPUT:**



## 4. Analysis:

• Time Complexity:  
 - Linear Search: O(n) – Best: O(1), Worst: O(n)  
 - Binary Search: O(log n) – Only for sorted data  
• Recommendation:  
 - Use Binary Search if the product list is sorted or small updates are required.  
 - Use HashMap or Search Trees for large-scale e-commerce systems for faster lookups.

# Exercise 3: Sorting Customer Orders

## Implementation:

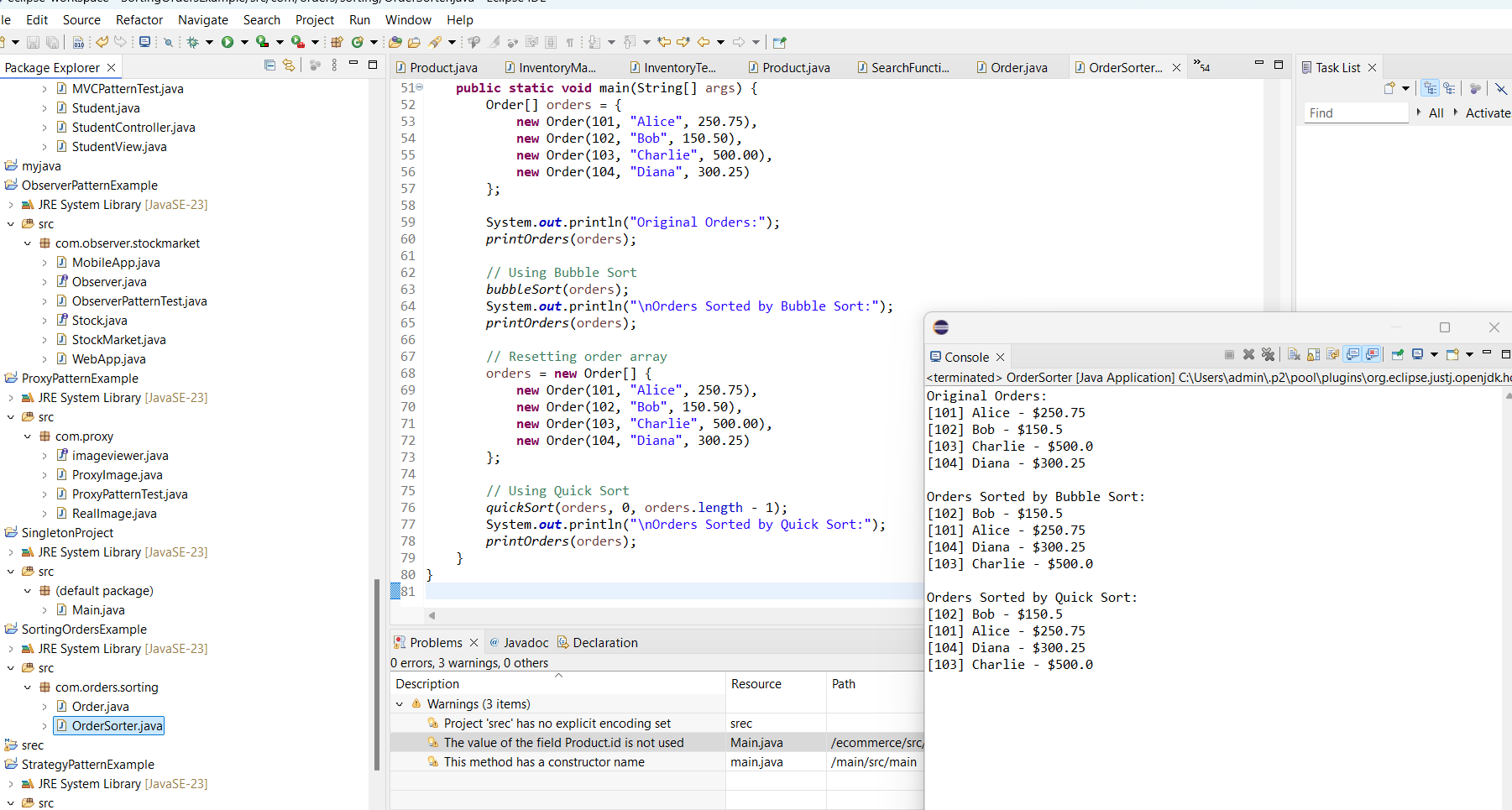
### Order.java

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 + " - $" + totalPrice;  
 }  
}

### OrderSorter.java

public class OrderSorter {  
  
 // Bubble Sort  
 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].totalPrice > orders[j + 1].totalPrice) {  
 Order temp = orders[j];  
 orders[j] = orders[j + 1];  
 orders[j + 1] = temp;  
 }  
 }  
 }  
 }  
  
 // Quick Sort  
 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].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;  
 }  
  
 public static void printOrders(Order[] orders) {  
 for (Order order : orders) {  
 System.out.println(order);  
 }  
 }  
  
 public static void main(String[] args) {  
 Order[] orders = {  
 new Order(101, "Alice", 250.75),  
 new Order(102, "Bob", 150.50),  
 new Order(103, "Charlie", 500.00),  
 new Order(104, "Diana", 300.25)  
 };  
  
 System.out.println("Original Orders:");  
 printOrders(orders);  
  
 // Using Bubble Sort  
 bubbleSort(orders);  
 System.out.println("\nOrders Sorted by Bubble Sort:");  
 printOrders(orders);  
  
 // Resetting order array  
 orders = new Order[] {  
 new Order(101, "Alice", 250.75),  
 new Order(102, "Bob", 150.50),  
 new Order(103, "Charlie", 500.00),  
 new Order(104, "Diana", 300.25)  
 };  
  
 // Using Quick Sort  
 quickSort(orders, 0, orders.length - 1);  
 System.out.println("\nOrders Sorted by Quick Sort:");  
 printOrders(orders);  
 }  
}

**OUTPUT:**

****

## Analysis:

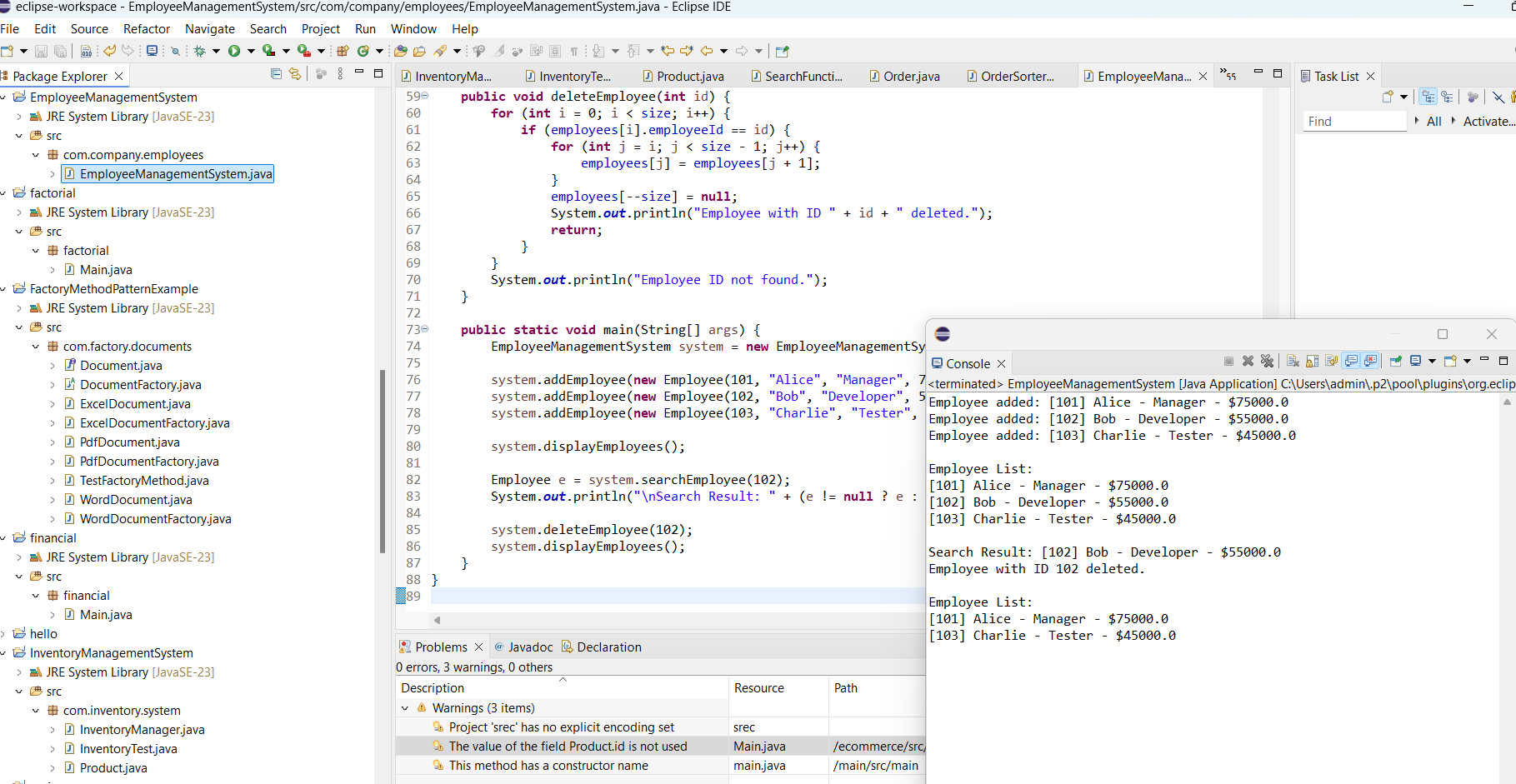
• Time Complexity Comparison:  
 - Bubble Sort: O(n^2) – inefficient for large datasets.  
 - Quick Sort: O(n log n) average case – efficient and widely used.  
• Conclusion:  
 - Quick Sort is generally preferred due to better performance on larger datasets.  
 - Bubble Sort is useful for educational purposes or small datasets.

# Exercise 4: Employee Management System

## Implementation:

public class EmployeeManagementSystem {  
 static 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 + "] " + name + " - " + position + " - $" + salary;  
 }  
 }  
  
 private Employee[] employees;  
 private int size;  
  
 public EmployeeManagementSystem(int capacity) {  
 employees = new Employee[capacity];  
 size = 0;  
 }  
  
 // Add an employee  
 public void addEmployee(Employee emp) {  
 if (size < employees.length) {  
 employees[size++] = emp;  
 System.out.println("Employee added: " + emp);  
 } else {  
 System.out.println("Employee list is full.");  
 }  
 }  
  
 // Search employee by ID  
 public Employee searchEmployee(int id) {  
 for (int i = 0; i < size; i++) {  
 if (employees[i].employeeId == id) {  
 return employees[i];  
 }  
 }  
 return null;  
 }  
  
 // Traverse and display all employees  
 public void displayEmployees() {  
 System.out.println("\nEmployee List:");  
 for (int i = 0; i < size; i++) {  
 System.out.println(employees[i]);  
 }  
 }  
  
 // Delete employee by ID  
 public void deleteEmployee(int id) {  
 for (int i = 0; i < size; i++) {  
 if (employees[i].employeeId == id) {  
 // Shift elements left to fill the gap  
 for (int j = i; j < size - 1; j++) {  
 employees[j] = employees[j + 1];  
 }  
 employees[--size] = null;  
 System.out.println("Employee with ID " + id + " deleted.");  
 return;  
 }  
 }  
 System.out.println("Employee ID not found.");  
 }  
  
 public static void main(String[] args) {  
 EmployeeManagementSystem system = new EmployeeManagementSystem(5);  
  
 system.addEmployee(new Employee(101, "Alice", "Manager", 75000));  
 system.addEmployee(new Employee(102, "Bob", "Developer", 55000));  
 system.addEmployee(new Employee(103, "Charlie", "Tester", 45000));  
  
 system.displayEmployees();  
  
 Employee e = system.searchEmployee(102);  
 System.out.println("\nSearch Result: " + (e != null ? e : "Not found"));  
  
 system.deleteEmployee(102);  
 system.displayEmployees();  
 }  
}

**OUTPUT:**



## Analysis:

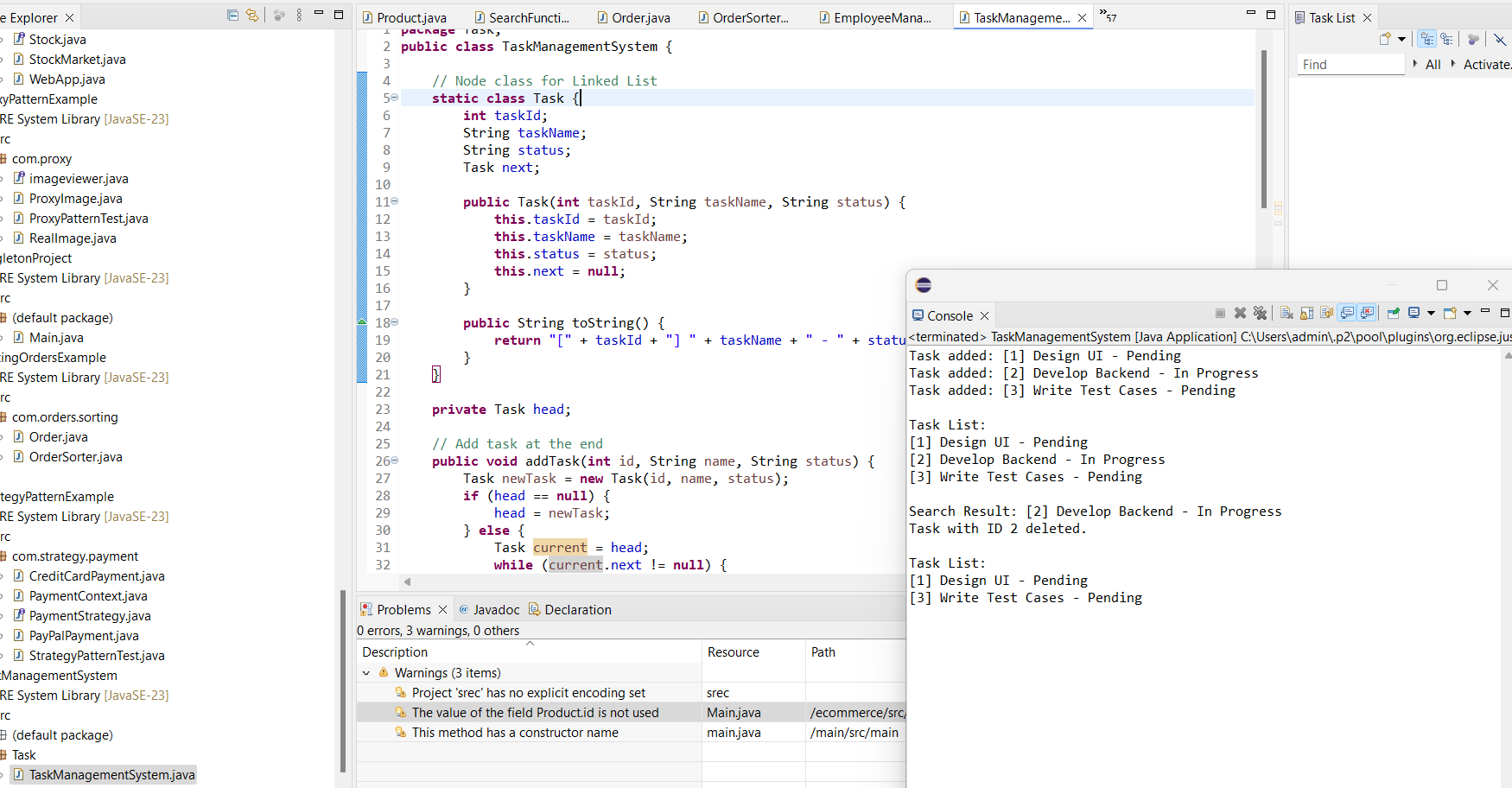
Time Complexity of Operations:  
- Add: O(1) – Adding to the end of the array is constant time.  
- Search: O(n) – Requires scanning each element linearly.  
- Traverse: O(n) – Visiting each employee one by one.  
- Delete: O(n) – Deleting an item requires shifting subsequent elements.  
  
Limitations of Arrays:  
- Fixed Size: Cannot resize dynamically, leads to wasted or insufficient space.  
- Inefficient Insert/Delete: Requires shifting elements.  
- Linear Search: No built-in optimization for faster lookups.  
- Better Options: Use ArrayList or HashMap in large or dynamic systems.

# Exercise 5: Task Management System

## Implementation:

public class TaskManagementSystem {  
  
 // Node class for Linked List  
 static 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;  
 }  
  
 public String toString() {  
 return "[" + taskId + "] " + taskName + " - " + status;  
 }  
 }  
  
 private Task head;  
  
 // Add task at the end  
 public void addTask(int id, String name, String status) {  
 Task newTask = new Task(id, name, status);  
 if (head == null) {  
 head = newTask;  
 } else {  
 Task current = head;  
 while (current.next != null) {  
 current = current.next;  
 }  
 current.next = newTask;  
 }  
 System.out.println("Task added: " + newTask);  
 }  
  
 // Search for a 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 and print all tasks  
 public void displayTasks() {  
 System.out.println("\nTask List:");  
 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;  
 System.out.println("Task with ID " + id + " deleted.");  
 return;  
 }  
  
 Task current = head;  
 while (current.next != null) {  
 if (current.next.taskId == id) {  
 current.next = current.next.next;  
 System.out.println("Task with ID " + id + " deleted.");  
 return;  
 }  
 current = current.next;  
 }  
  
 System.out.println("Task with ID " + id + " not found.");  
 }  
  
 public static void main(String[] args) {  
 TaskManagementSystem system = new TaskManagementSystem();  
  
 system.addTask(1, "Design UI", "Pending");  
 system.addTask(2, "Develop Backend", "In Progress");  
 system.addTask(3, "Write Test Cases", "Pending");  
  
 system.displayTasks();  
  
 Task t = system.searchTask(2);  
 System.out.println("\nSearch Result: " + (t != null ? t : "Not found"));  
  
 system.deleteTask(2);  
 system.displayTasks();  
 }  
}

**OUTPUT:**

****

## Analysis:

Time Complexity of Operations:  
- Add Task: O(n) – Requires traversal to the end of the list.  
- Search Task: O(n) – Linear search through the nodes.  
- Display Tasks: O(n) – Traverses and prints each task.  
- Delete Task: O(n) – Searches for task and updates links.  
  
Advantages of Linked Lists over Arrays:  
- Dynamic Size: No need for a predefined size.  
- Efficient Insert/Delete: No element shifting required.  
- No Memory Wastage: Memory allocated as needed.  
- Ideal for Dynamic Data: Useful when frequent additions or deletions are required.

## Exercise 6: Library Management System

### Implementation

java

Copy code

import java.util.Arrays;

import java.util.Comparator;

public class LibraryManagementSystem {

static 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 String toString() {

return "[" + bookId + "] " + title + " by " + author;

}

}

// Linear Search

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

for (Book book : books) {

if (book.title.equalsIgnoreCase(targetTitle)) {

return book;

}

}

return null;

}

// Binary Search (Array must be sorted by title)

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

int low = 0;

int high = books.length - 1;

while (low <= high) {

int mid = (low + high) / 2;

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

if (comparison == 0) {

return books[mid];

} else if (comparison < 0) {

low = mid + 1;

} else {

high = mid - 1;

}

}

return null;

}

public static void main(String[] args) {

Book[] books = {

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

new Book(2, "Python Basics", "Guido van Rossum"),

new Book(3, "Data Structures", "Robert Lafore"),

new Book(4, "Algorithms", "Thomas Cormen")

};

// Linear Search

String searchTitle = "Python Basics";

Book result = linearSearch(books, searchTitle);

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

// Sort books by title for Binary Search

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

// Binary Search

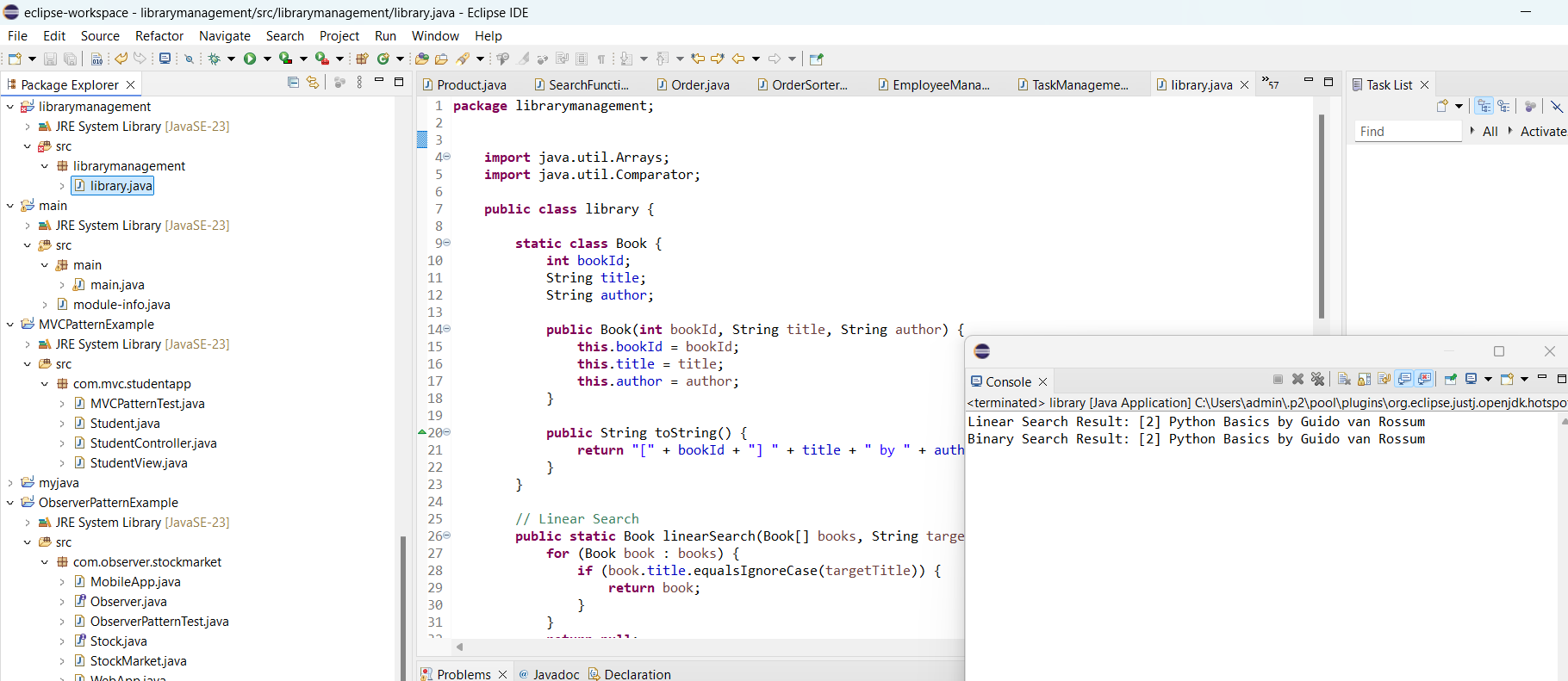
result = binarySearch(books, searchTitle);

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

}

}

**OUTPUT:**

****

### Analysis

Linear search has a time complexity of O(n). It is straightforward and works well for small or unsorted datasets. However, it becomes inefficient as the number of items increases.

Binary search is much faster, with a time complexity of O(log n). But it requires the dataset to be sorted before performing the search. It is ideal for large, sorted lists and is commonly used in search engines and sorted databases.

## Exercise 7: Financial Forecasting

### ✅ Implementation

java

Copy code

public class FinancialForecasting {

// Recursive method to calculate future value

public static double forecastValue(double currentValue, double growthRate, int years) {

if (years == 0) {

return currentValue;

}

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

}

public static void main(String[] args) {

double initialValue = 10000.0; // Starting amount

double growthRate = 0.08; // 8% growth rate

int forecastYears = 5;

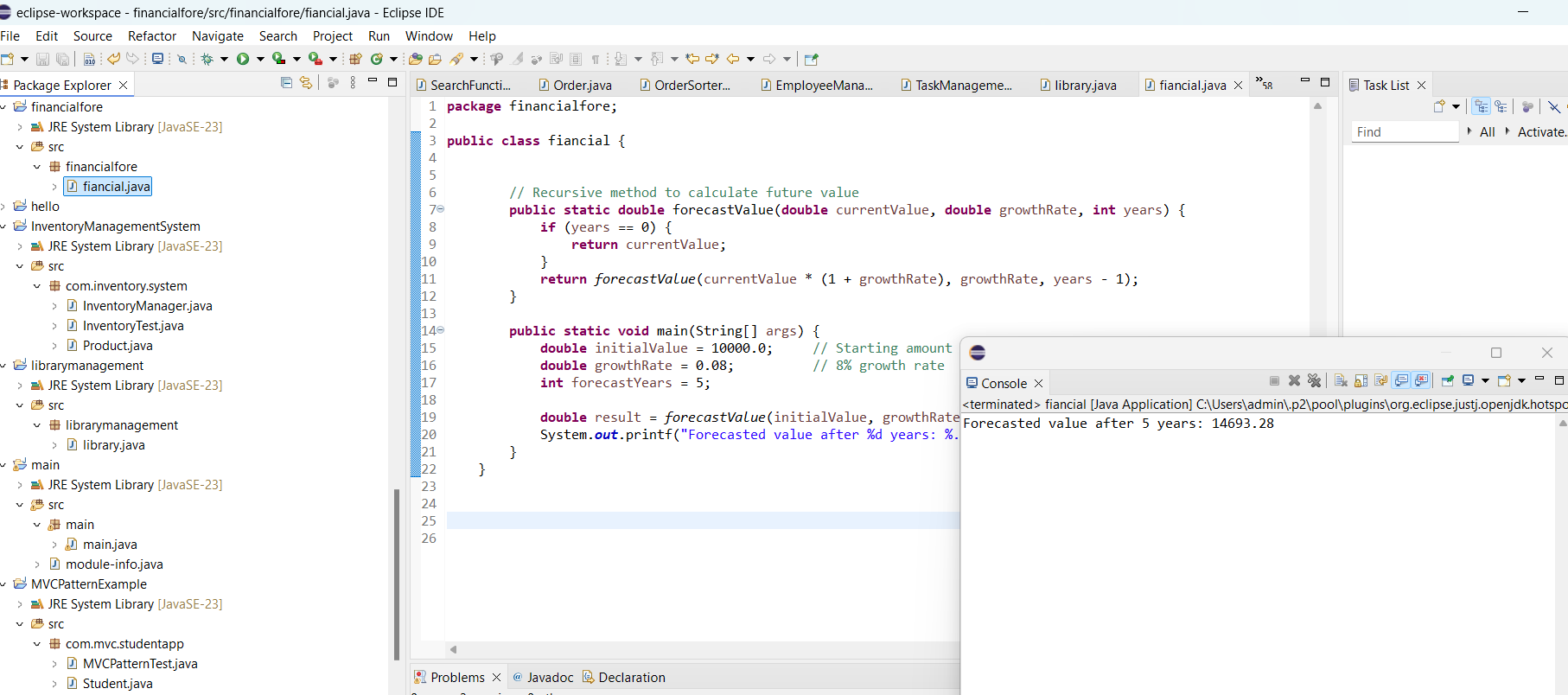
double result = forecastValue(initialValue, growthRate, forecastYears);

System.out.printf("Forecasted value after %d years: %.2f\n", forecastYears, result);

}

}

**OUTPUT:**

****

### Analysis

The recursive forecasting algorithm has a linear time complexity of O(n), where n is the number of years. Each recursive call computes the future value for the next year until the base case is reached (year 0).This approach is efficient for small to moderate timeframes, and the logic is easy to understand and implement. However, for very large values of n, it can lead to deep recursion and potential stack overflow.To optimize the solution, you can convert the recursion to an iterative loop or use memoization, although in this specific use case recursion is already efficient because each year's result only depends on the previous year.Recursion works well here because the problem is naturally sequential and builds on prior values.