# **Week 1**

**Week 1 - Design Principles & Patterns**

**Exercise 1: Implementing the Singleton Pattern**

**Objective:**

To ensure that a logging utility class (Logger) has only one instance throughout the application lifecycle to ensure consistent logging.

**Logger.java**

package hepsiba;

public class Logger {

private static Logger *instance*;

private Logger() {

System.*out*.println("Logger initialized");

}

public static Logger getInstance() {

if (*instance* == null) {

*instance* = new Logger();

}

return *instance*;

}

public void log(String message) {

System.*out*.println("LOG: " + message);

}

}

**Tester.java**

package hepsiba;

public class Tester {

public static void main(String[] args) {

Logger logger1 = Logger.*getInstance*();

logger1.log("First log message")

Logger logger2 = Logger.*getInstance*();

logger2.log("Second log message");

if (logger1 == logger2) {

System.*out*.println("Same Logger instance used.");

} else {

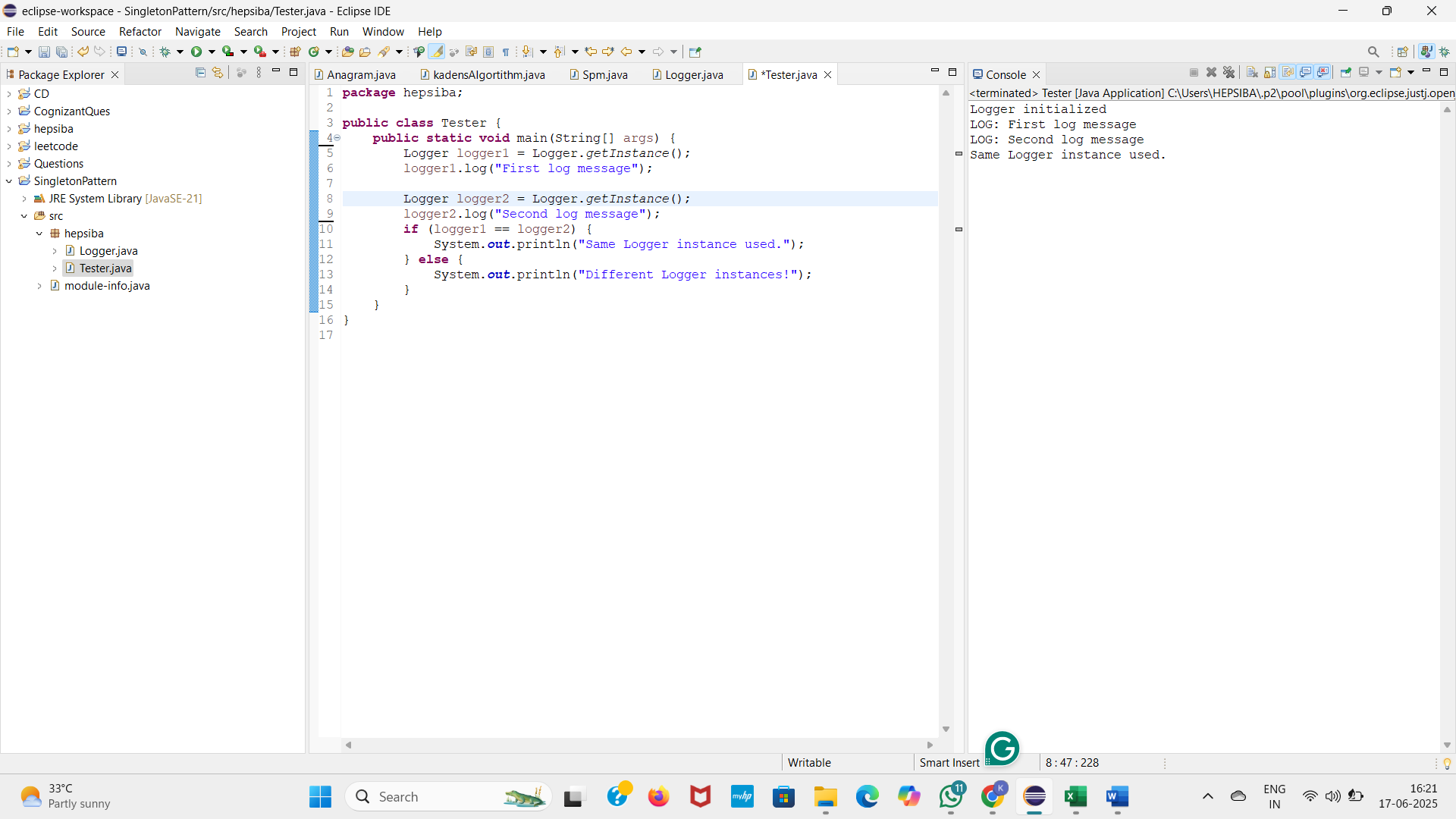
System.*out*.println("Different Logger instances!");

}

}

}

**OUTPUT**



**Week 1 - Design Principles & Patterns Exercise**

**Exercise 2 - Implementing the Factory Method Pattern**

**Objective**:

To implement the Factory Method Pattern and design a notification system that can create different types of notifications: Email, SMS, and Push. This setup makes the system easy to extend and supports loose coupling.

**1. Interface**

java

public interface Notification {

void notifyUser();

}

**2. Concrete Notification Classes**

**EmailNotification.java**

java

public class EmailNotification implements Notification {

@Override

public void notifyUser() {

System.out.println("Sending an Email Notification.");

}

}

**SMSNotification.java**

java

public class SMSNotification implements Notification {

@Override

public void notifyUser() {

System.out.println("Sending an SMS Notification.");

}

}

**PushNotification.java**

java

public class PushNotification implements Notification {

@Override

public void notifyUser() {

System.out.println("Sending a Push Notification.");

}

}

**3. Abstract Factory**

java

public abstract class NotificationFactory {

public abstract Notification createNotification();

}

**4. Concrete Factories**

**EmailFactory.java**

java

public class EmailFactory extends NotificationFactory {

@Override

public Notification createNotification() {

return new EmailNotification();

}

}

**SMSFactory.java**

java

public class SMSFactory extends NotificationFactory {

@Override

public Notification createNotification() {

return new SMSNotification();

}

}

**PushFactory.java**

java

public class PushFactory extends NotificationFactory {

public Notification createNotification() {

return new PushNotification();

}

}

**5. Test Class**

java

public class NotificationTest {

public static void main(String[] args) {

NotificationFactory emailFactory = new EmailFactory();

Notification email = emailFactory.createNotification();

email.notifyUser();

NotificationFactory smsFactory = new SMSFactory();

Notification sms = smsFactory.createNotification();

sms.notifyUser();

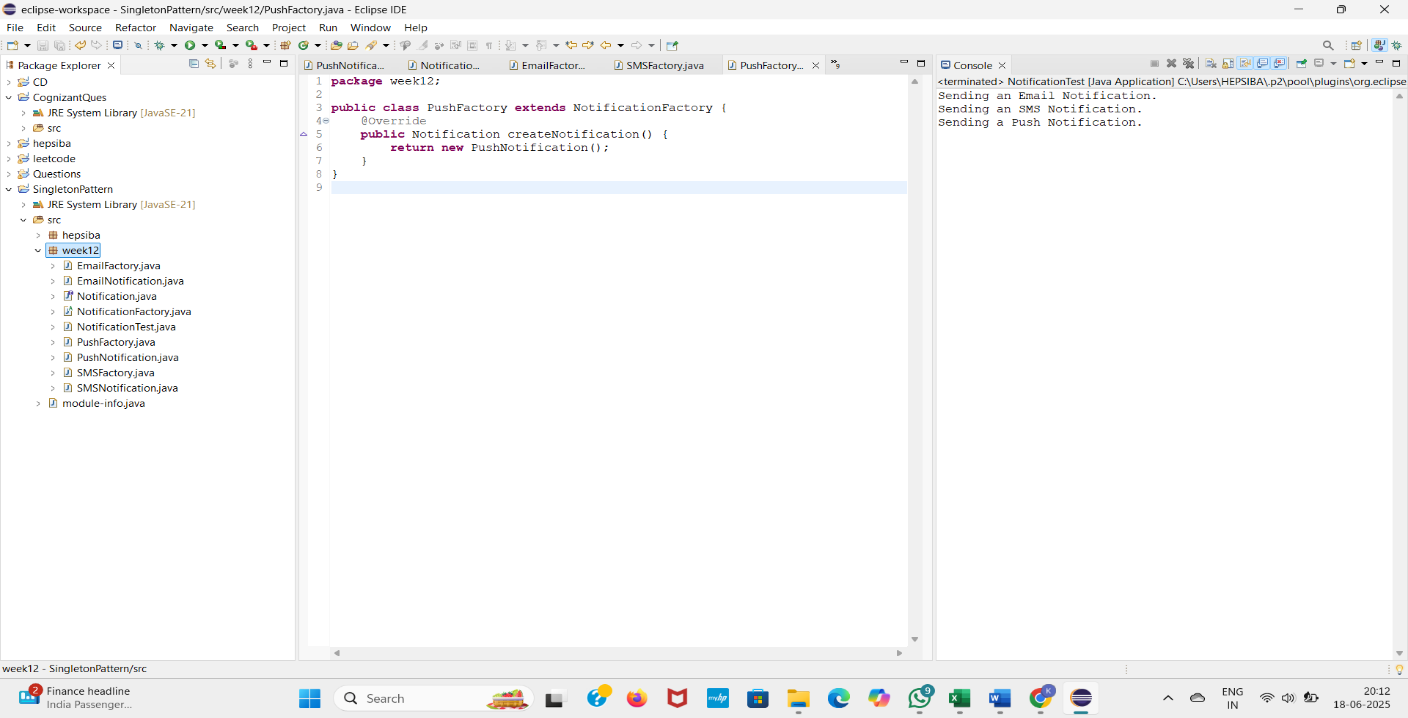
NotificationFactory pushFactory = new PushFactory();

Notification push = pushFactory.createNotification();

push.notifyUser();

}

}

**OUTPUT**

**Week 1:Data Structures and Algorithms**

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

**Objective:**

To implement and compare linear and binary search algorithms for locating products in an e-commerce platform's product catalog, with a focus on analyzing their performance using asymptotic notation**.**

**Steps**

**1. Understand Asymptotic Notation**

* **Big O Notation** helps us describe the time and space complexity of an algorithm in terms of input size n.

| **Complexity** | **Meaning** |
| --- | --- |
| O(1) | Constant time |
| O(n) | Linear time |
| O(log n) | Logarithmic time |
| O(n log n) | Linearithmic time |

* **Search Operation Scenarios**:

| **Search Type** | **Best Case** | **Average Case** | **Worst Case** |
| --- | --- | --- | --- |
| Linear Search | O(1) | O(n) | O(n) |
| Binary Search | O(1) | O(log n) | O(log n) |

**2. Setup**

We define a Product class with three attributes:

* productId
* productName
* category

**3. Implementation**

* Products are stored in a **normal array** for linear search.
* Products are **sorted** for binary search.

**Java Code**

java

package week12;

import java.util.\*;

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 "Product{" + "ID=" + productId + ", Name='" + productName + "', Category='" + category + "'}";

}

}

public class EcommerceSearch {

public static boolean isProductAvailable(Product[] products, String targetName) {

targetName = targetName.trim().toLowerCase();

for (Product product : products) {

if (product.productName.trim().toLowerCase().equals(targetName)) {

return true;

}

}

return false;

}

**Linear Search**

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

targetName = targetName.trim().toLowerCase();

for (Product product : products) {

if (product.productName.trim().toLowerCase().equals(targetName)) {

return product;

}

}

return null;

}

**Binary Search**

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

targetName = targetName.trim().toLowerCase();

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

int left = 0;

int right = products.length - 1;

while (left <= right) {

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

String midName = products[mid].productName.trim().toLowerCase();

int comparison = midName.compareTo(targetName);

if (comparison == 0) {

return products[mid];

} else if (comparison < 0) {

left = mid + 1;

} else {

right = mid - 1;

}

}

return null;

}

**Main method**

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

Product[] products = {

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

new Product(102, "Mobile", "Electronics"),

new Product(103, "T-shirt", "Clothing"),

new Product(104, "Shoes", "Footwear"),

new Product(105, "Watch", "Accessories"),

new Product(106, "Bag", "Travel"),

new Product(107, "Headphones", "Electronics")

};

System.out.print("\n Enter product name to search: ");

String targetName = sc.nextLine().trim();

if (!isProductAvailable(products, targetName)) {

System.out.println(" Product not found in the list.");

} else {

Product result1 = linearSearch(products, targetName);

System.out.println("\n Linear Search Result: " + result1);

Product result2 = binarySearch(products, targetName);

System.out.println(" Binary Search Result: " + result2);

}

sc.close();

}

}

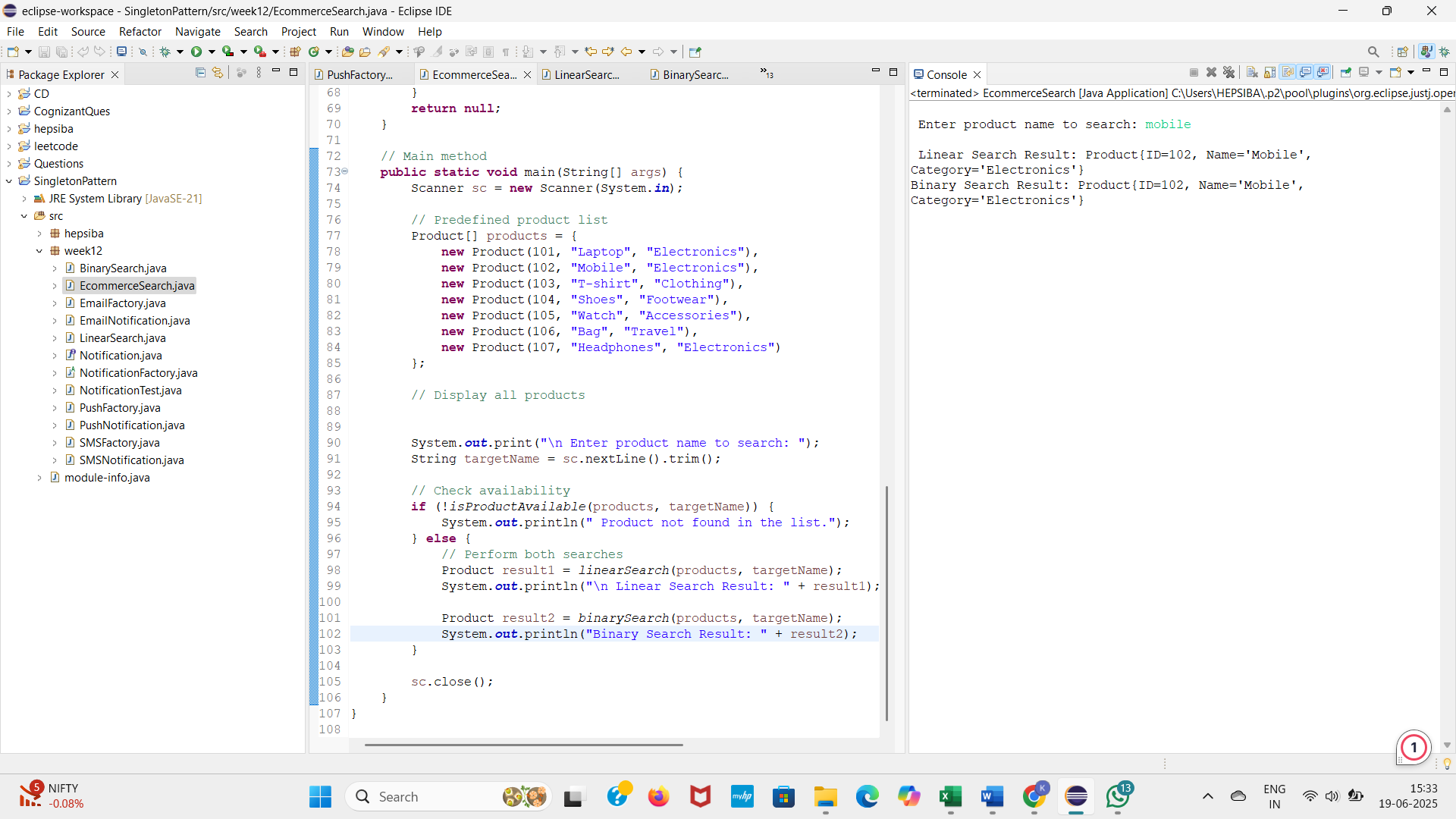
**4. Analysis**

* **Linear Search**
  + **Time Complexity:** O(n)
  + **Use Case:** Small or unsorted datasets
* **Binary Search**
  + **Time Complexity:** O(log n)
  + **Use Case:** Large sorted datasets
  + **Limitation:** Requires sorted data

**Conclusion**

For an e-commerce platform with large product catalogs, binary search is more efficient due to its log-time complexity. However, it requires the data to be sorted. For smaller datasets, linear search is simpler and acceptable.

**OUTPUT**



**Week 1 – Data Structures and Algorithms**

**Exercise 7: Financial Forecasting**

**Objective:**

To develop a recursive algorithm that predicts future financial values based on an initial amount and growth rate, and to analyze the efficiency and optimization strategies for recursion in forecasting applications.

**Analysis**

* **Time Complexity:** O(n), where n is the number of years. Each recursive call processes one year.
* **Optimization Tip:**
  + Use **memoization** or **iteration** if you need to compute values for very large n to avoid stack overflow or redundant computation.

**java**

package week12;

import java.util.Scanner;

public class FinancialForecast {

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

if (years == 0) {

return currentValue;

}

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

}

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

double currentValue = 10000; // Starting amount

double annualGrowthRate = 0.07;

System.out.println("Enter after how many years you want to predict:");

int years = sc.nextInt();

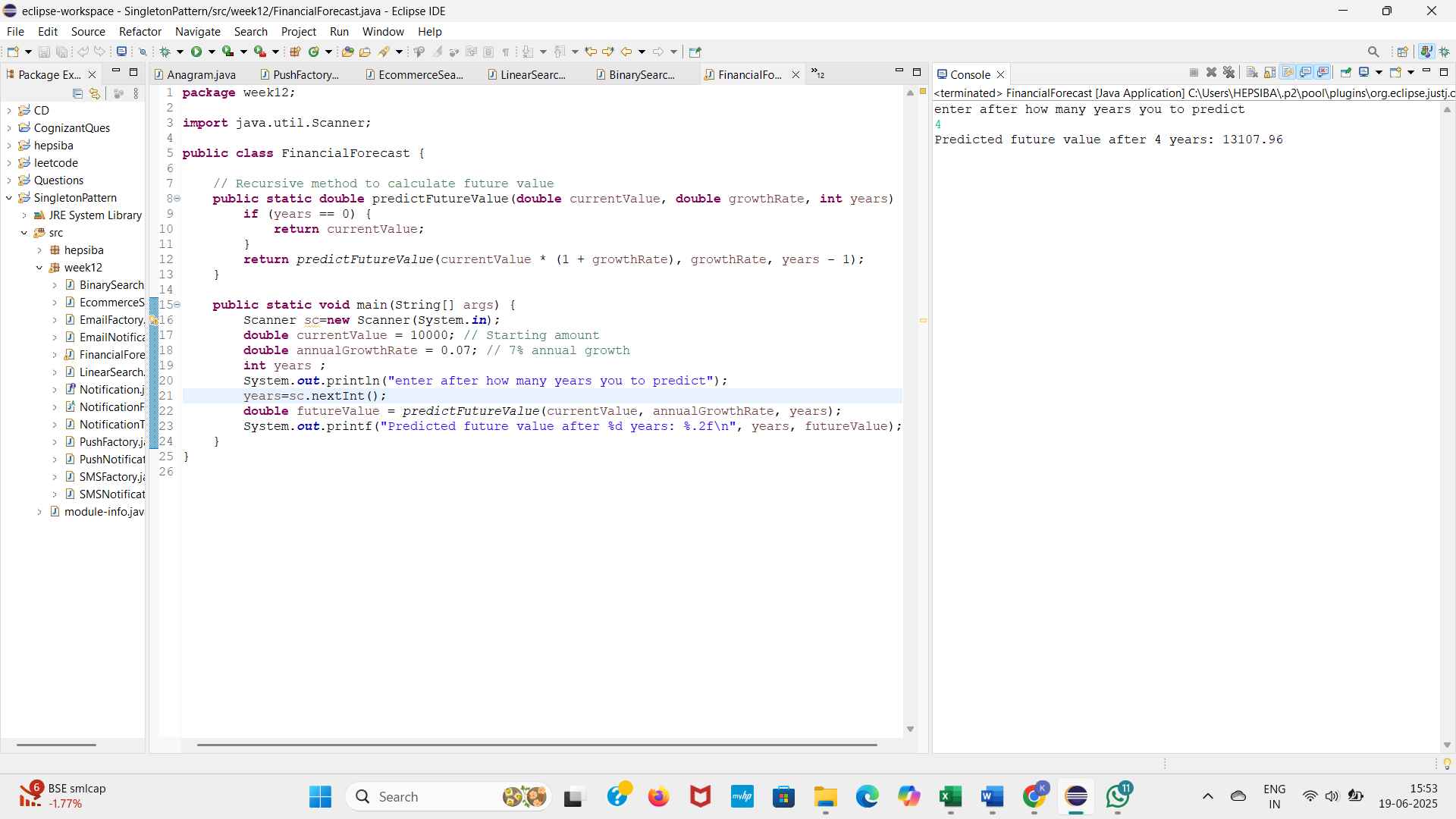
double futureValue = predictFutureValue(currentValue, annualGrowthRate, years);

System.out.printf("Predicted future value after %d years: %.2f\n", years, futureValue);

}}

**Conclusion:**

Recursion simplifies the logic for compound growth calculation but should be optimized or replaced with loops for large datasets.

**OUTPUT**

**Week 1-Data Structures and Algorithms**

**Exercise 3 - Sorting Customer Orders**

**Objective:**

To sort customer orders by their totalPrice using different sorting algorithms and analyze their efficiency.

**CODE**

package week12;

import java.util.Scanner;

public class CustomerOrderSorter {

static 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;

}

}

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;

}

}

}

}

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 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("\nOriginal Orders:");

*printOrders*(orders);

Order[] bubbleSorted = orders.clone();

*bubbleSort*(bubbleSorted);

System.*out*.println("\nOrders Sorted by Bubble Sort:");

*printOrders*(bubbleSorted);

Order[] quickSorted = orders.clone();

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

System.*out*.println("\nOrders Sorted by Quick Sort:");

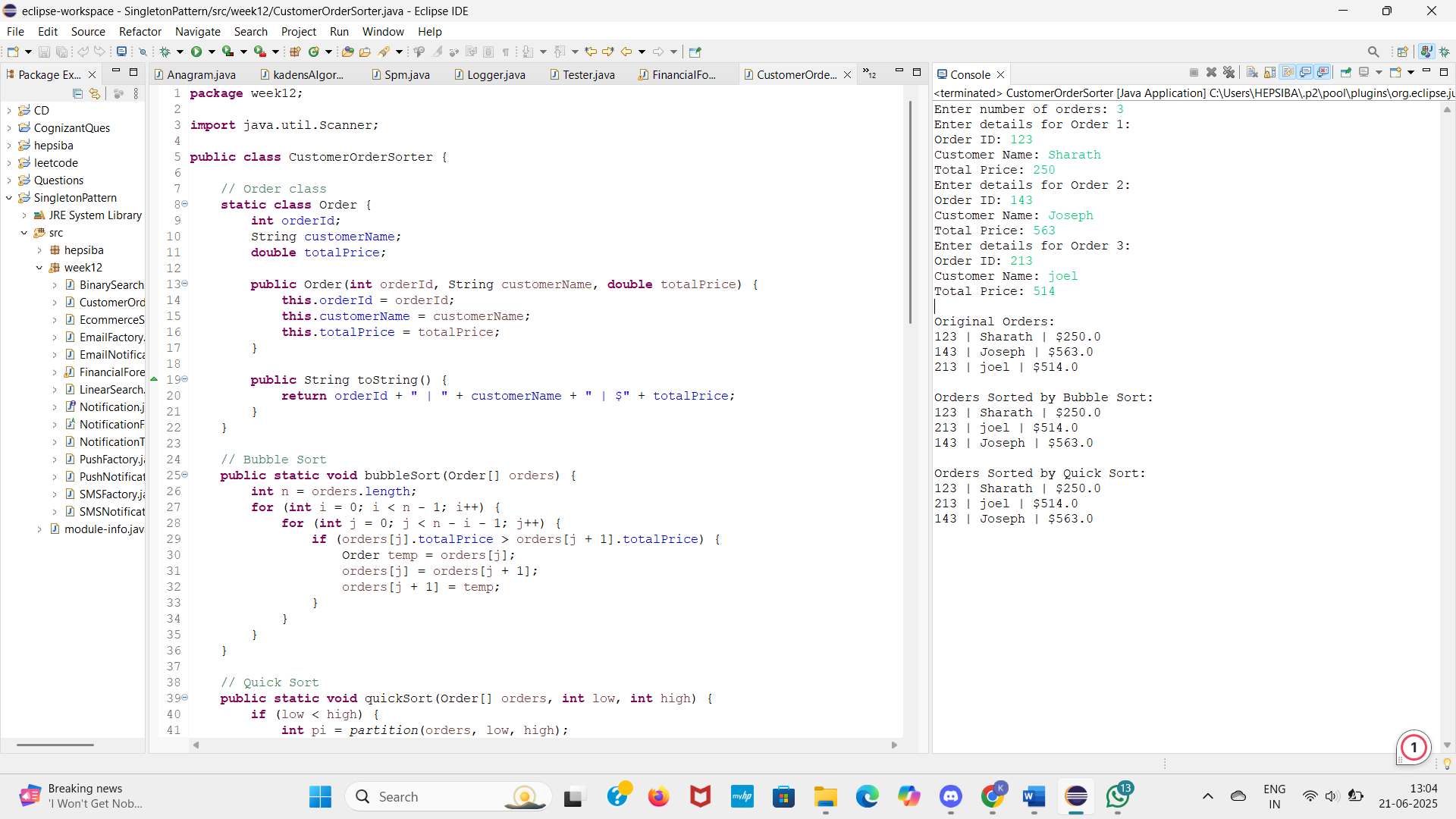
*printOrders*(quickSorted);

sc.close();

}

}

**OUTPUT**

****