**Atchaya V - 6418561**

# **Week 1**

**Design principles & Patterns**

**Exercise 1: Implementing the Singleton Pattern**

**Project: SingletonPatternExample**

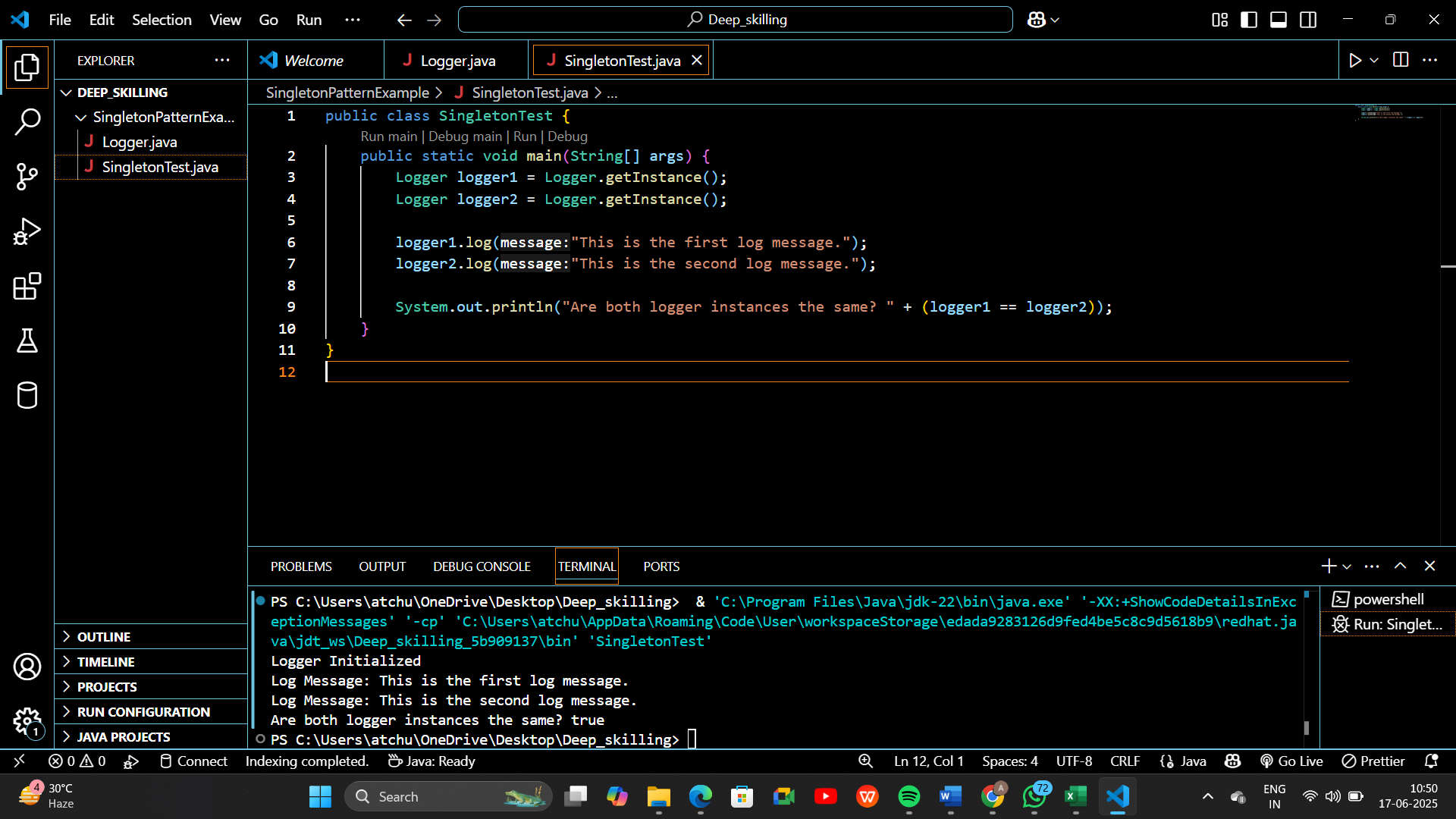
**Logger.java**

1. public class Logger {
2. private static Logger instance;
3. private Logger() {
4. System.out.println("Logger Initialized");
5. }
6. public static Logger getInstance() {
7. if (instance == null) {
8. instance = new Logger();
9. }
10. return instance;
11. }
12. public void log(String message) {
13. System.out.println("Log Message: " + message);
14. }
15. }

**SingletonTest.java**

1. public class SingletonTest {
2. public static void main(String[] args) {
3. Logger logger1 = Logger.getInstance();
4. Logger logger2 = Logger.getInstance();
5. logger1.log("This is the first log message.");
6. logger2.log("This is the second log message.");
7. System.out.println("Are both logger instances the same? " + (logger1 == logger2));
8. }
9. }

**OUTPUT:**



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

**Project: FactoryMethodPatternExample**

**Document.java**

1. package FactoryMethodPatternExample;
2. public interface Document {
3. void open();
4. }

**WordDocument.java**

1. package FactoryMethodPatternExample;
2. public class WordDocument implements Document {
3. public void open() {
4. System.out.println("Opening Word Document");
5. }
6. }

**PdfDocument.java**

1. package FactoryMethodPatternExample;
2. public class PdfDocument implements Document {
3. public void open() {
4. System.out.println("Opening PDF Document");
5. }
6. }

**ExcelDocument.java**

1. package FactoryMethodPatternExample;
2. public class ExcelDocument implements Document {
3. public void open() {
4. System.out.println("Opening Excel Document");
5. }
6. }

**DocumentFactory.java**

1. package FactoryMethodPatternExample;
2. public abstract class DocumentFactory {
3. public abstract Document createDocument();
4. }

**WordDocumentFactory.java**

1. package FactoryMethodPatternExample;
2. public class WordDocumentFactory extends DocumentFactory {
3. public Document createDocument() {
4. return new WordDocument();
5. }
6. }

**PdfDocumentFactory.java**

1. package FactoryMethodPatternExample;
2. public class PdfDocumentFactory extends DocumentFactory {
3. public Document createDocument() {
4. return new PdfDocument();
5. }
6. }

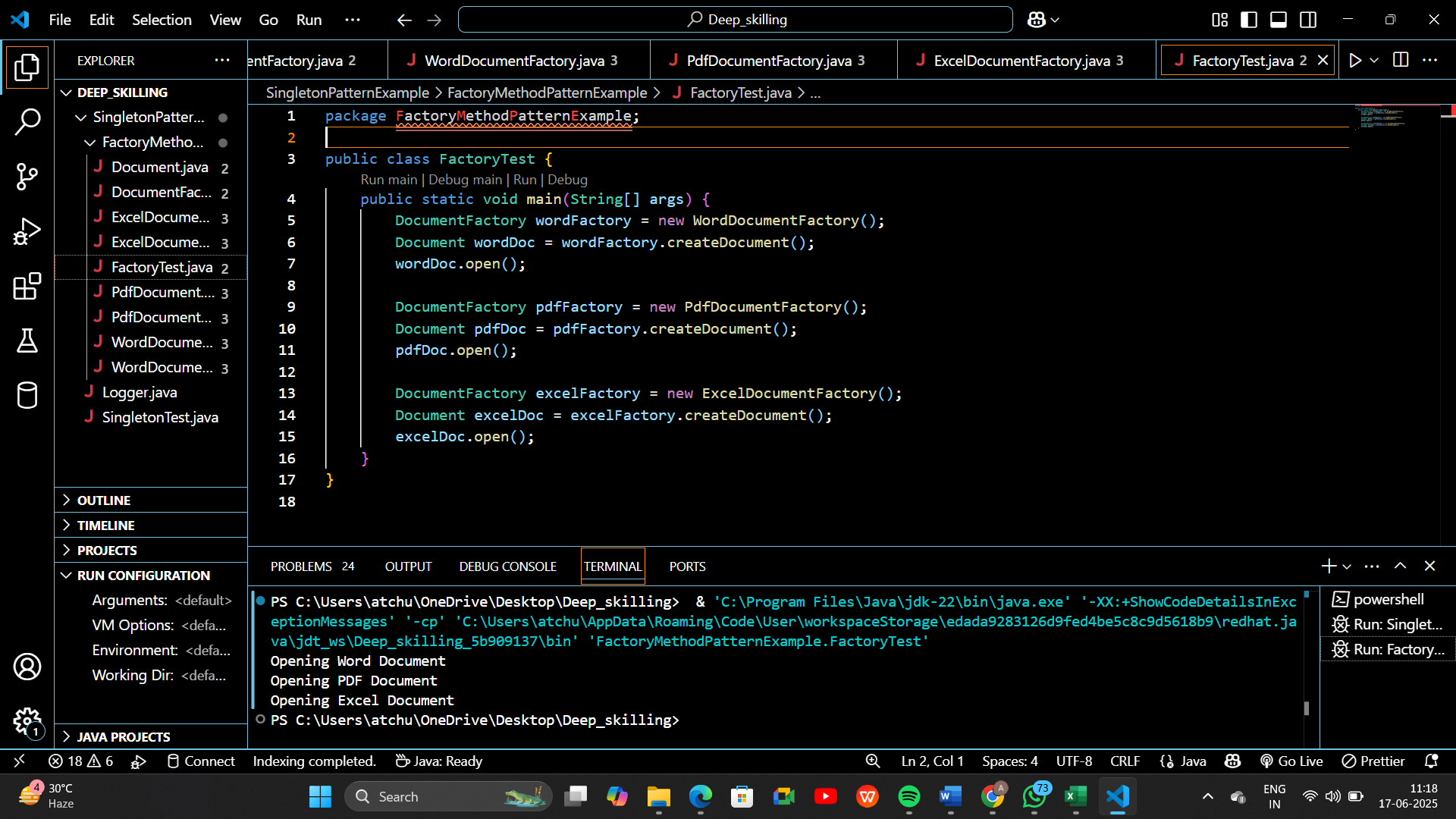
**ExcelDocumentFactory.java**

1. package FactoryMethodPatternExample;
2. public class ExcelDocumentFactory extends DocumentFactory {
3. public Document createDocument() {
4. return new ExcelDocument();
5. }
6. }

**FactoryTest.java**

1. package FactoryMethodPatternExample;
2. public class FactoryTest {
3. public static void main(String[] args) {
4. DocumentFactory wordFactory = new WordDocumentFactory();
5. Document wordDoc = wordFactory.createDocument();
6. wordDoc.open();
7. DocumentFactory pdfFactory = new PdfDocumentFactory();
8. Document pdfDoc = pdfFactory.createDocument();
9. pdfDoc.open();
10. DocumentFactory excelFactory = new ExcelDocumentFactory();
11. Document excelDoc = excelFactory.createDocument();
12. excelDoc.open();
13. }
14. }

**OUTPUT:**



**Data structures and Algorithms**

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

**Step 1: Understanding Asymptotic Notation**

Big O Notation:

Describes the upper limit on the time complexity of an algorithm. It helps to understand how an algorithm performs as the input size increases.

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

**Step 2: Setup**

**Product.java**

1. package EcommercePlatform;
2. public class Product {
3. int productId;
4. String productName;
5. String category;
6. public Product(int productId, String productName, String category) {
7. this.productId = productId;
8. this.productName = productName;
9. this.category = category;
10. }
11. public String toString() {
12. return "ID: " + productId + ", Name: " + productName + ", Category: " + category;
13. }
14. }

**Step 3: Implementation**

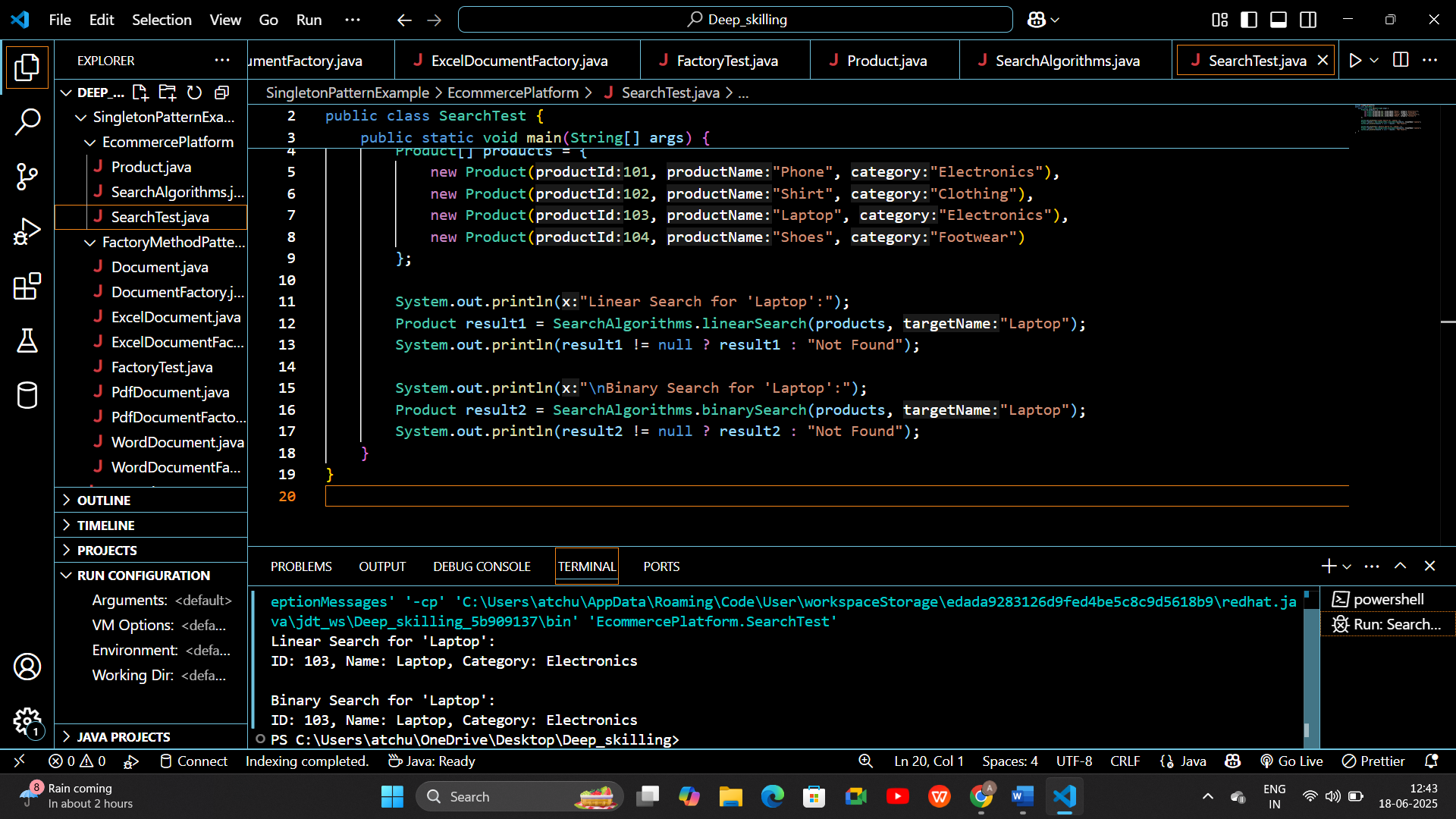
**SearchAlgorithms.java**

1. package EcommercePlatform;
2. import java.util.Arrays;
3. import java.util.Comparator;
4. public class SearchAlgorithms {
5. // Linear Search by Product Name
6. public static Product linearSearch(Product[] products, String targetName) {
7. for (Product product : products) {
8. if (product.productName.equalsIgnoreCase(targetName)) {
9. return product;
10. }
11. }
12. return null;
13. }
14. // Binary Search by Product Name
15. public static Product binarySearch(Product[] products, String targetName) {
16. // Make sure array is sorted by productName
17. Arrays.sort(products, Comparator.comparing(p -> p.productName));
18. int left = 0;
19. int right = products.length - 1;
20. while (left <= right) {
21. int mid = (left + right) / 2;
22. int comparison = targetName.compareToIgnoreCase(products[mid].productName);
23. if (comparison == 0) {
24. return products[mid];
25. } else if (comparison < 0) {
26. right = mid - 1;
27. } else {
28. left = mid + 1;
29. }
30. }
31. return null;
32. }
33. }

**Step 4:** **Analysis**

1. **SearchTest.java**
2. package EcommercePlatform;
3. public class SearchTest {
4. public static void main(String[] args) {
5. Product[] products = {
6. new Product(101, "Phone", "Electronics"),
7. new Product(102, "Shirt", "Clothing"),
8. new Product(103, "Laptop", "Electronics"),
9. new Product(104, "Shoes", "Footwear")
10. };
11. System.out.println("Linear Search for 'Laptop':");
12. Product result1 = SearchAlgorithms.linearSearch(products, "Laptop");
13. System.out.println(result1 != null ? result1 : "Not Found");
14. System.out.println("\nBinary Search for 'Laptop':");
15. Product result2 = SearchAlgorithms.binarySearch(products, "Laptop");
16. System.out.println(result2 != null ? result2 : "Not Found");
17. }
18. }

**Output:**

****

1. **Linear Search Output**

**How it works:**

* Scans each element one by one.
* Matches "Laptop" at index 1.
* No sorting needed.

**Time Complexity:**

* Best case: O(1) (first item matches)
* Worst case: O(n) (last item or not found)

1. **Binary Search Output**

**How it works:**

* Sorts the array by productName (internally).
* Performs divide-and-conquer strategy.
* Finds "Laptop" efficiently.

**Time Complexity:**

* Always: O(log n) (faster for large sorted lists)

**Binary Search is more suitable for e-commerce platforms due to better performance with sorted product data.**

**Exercise 7: Financial Forecasting**

**Step 1: Recursion Basics**

Recursion is a programming technique where a function calls itself to solve smaller instances of the same problem.

**Recursion makes issues simpler by:**

* Dividing difficult tasks into smaller, identical subproblems.
* Lowering the complexity of the code to address issues with nested structures or repeating patterns.
* Avoiding backtracking, divide-and-conquer tactics, and manual loops in tree or graph traversal.

**Step 2: Setup**

**FV = PV × (1 + r)^n**

**Step 3: Implementation**

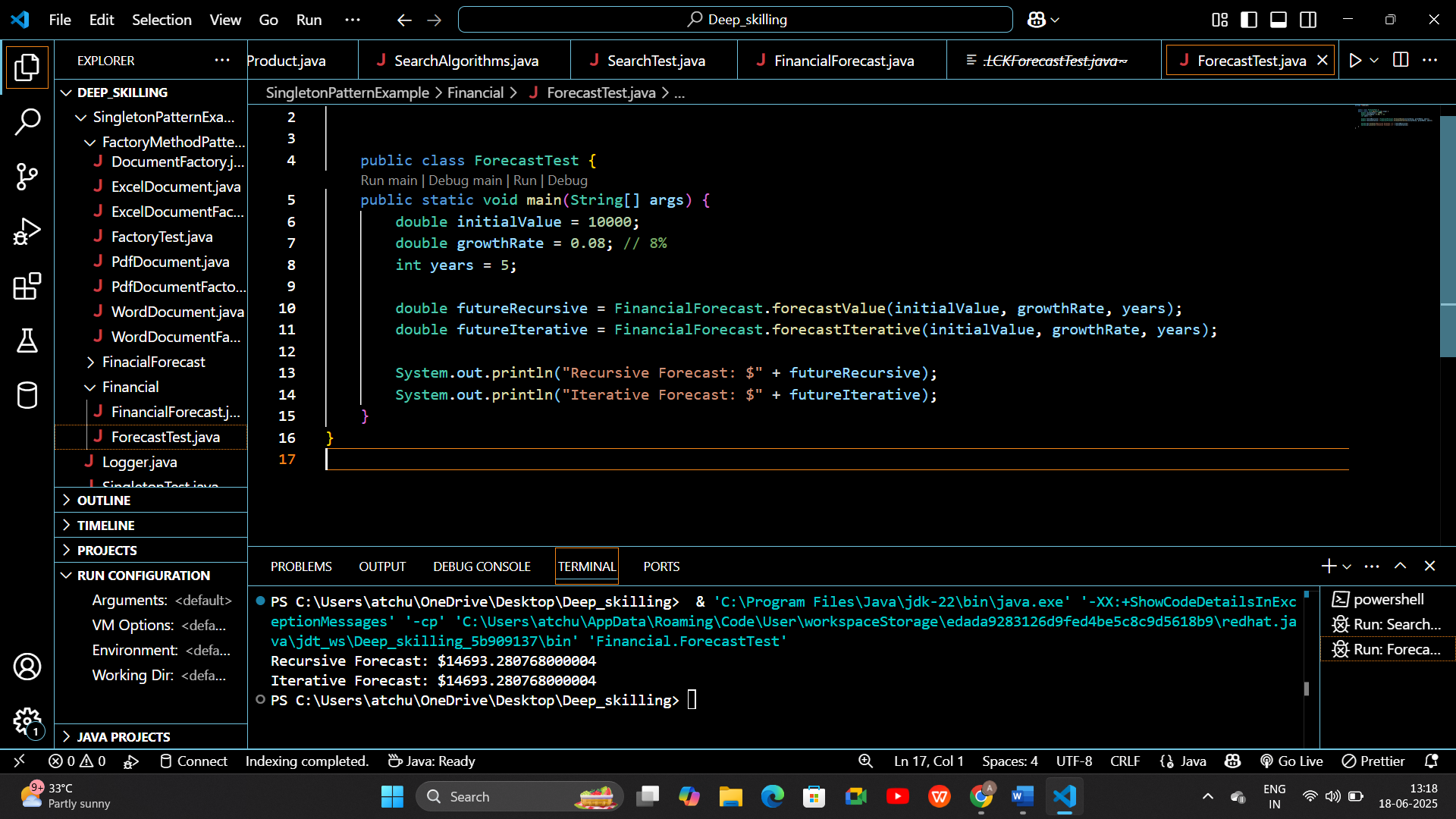
**FinancialForecast.java**

1. package Financial;
2. public class FinancialForecast {
3. // Recursive method to forecast future value
4. public static double forecastValue(double currentValue, double growthRate, int years) {
5. if (years == 0) {
6. return currentValue;
7. }
8. return forecastValue(currentValue \* (1 + growthRate), growthRate, years - 1);
9. }
10. // Optimized using tail recursion (optionally can use iteration)
11. public static double forecastIterative(double currentValue, double growthRate, int years) {
12. for (int i = 0; i < years; i++) {
13. currentValue \*= (1 + growthRate);
14. }
15. return currentValue;
16. }
17. }

**ForecastTest.java**

1. package Financial;
2. public class ForecastTest {
3. public static void main(String[] args) {
4. double initialValue = 10000;
5. double growthRate = 0.08; // 8%
6. int years = 5;
7. double futureRecursive = FinancialForecast.forecastValue(initialValue, growthRate, years);
8. double futureIterative = FinancialForecast.forecastIterative(initialValue, growthRate, years);
9. System.out.println("Recursive Forecast: $" + futureRecursive);
10. System.out.println("Iterative Forecast: $" + futureIterative);
11. }
12. }

**Output:**



**Step 4: Analysis**

Recursive Method Time Complexity: O(n) — one recursive call annually.  
  
Iterative Method: O(n) — the same as recursive, but without the overhead of function calls.  
   
**To prevent stack overflow and enhance performance for large n, use memoization or iteration.**