**Cognizant FSE – Java Assignment (WEEK -1)**

**Submitted Exercises**

**Module 1**: Design Patterns and Principles

1. **Exercise 1: Implementing the Singleton Pattern**

*Implemented a logging utility using the Singleton Design Pattern to ensure only one instance of the logger is used across the application.*

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

*Developed a document management system using the Factory Method Design Pattern for creating different types of document objects like Word, PDF, and Excel.*

**Module 2:** Data Structures and Algorithms

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

*Created a fast and efficient product search functionality using algorithmic techniques and search optimization.*

1. **Exercise 7: Financial Forecasting**

*Used recursive algorithms to build a basic financial forecasting model based on historical growth rates.*

**Submitted by**

**Name :**  ELAKKIYA S (Reg No – 111622102036)

**Email :** [elekcs030@rmkcet.ac.in](mailto:elekcs030@rmkcet.ac.in)

**College :** RMKCET

**Batch :** Java FSE – 2025

**SuperSet Id :** 6400874

1 .Project Title:

**Singleton Pattern Example**

**Objective :**

To implement the **Singleton Design Pattern** in Java to ensure that a **Logger utility class** maintain s only one instance throughout the application lifecycle, ensuring consistent and centralized logging.

**Design Pattern Used :**

**Singleton Pattern**

The Singleton pattern restricts the instantiation of a class to a single object. It is used where only one object is needed to coordinate actions across the system.

**Project Structure :**

SingletonPatternExample

│

├── Logger.java // Singleton class

└── Main.java // Driver class to test singleton

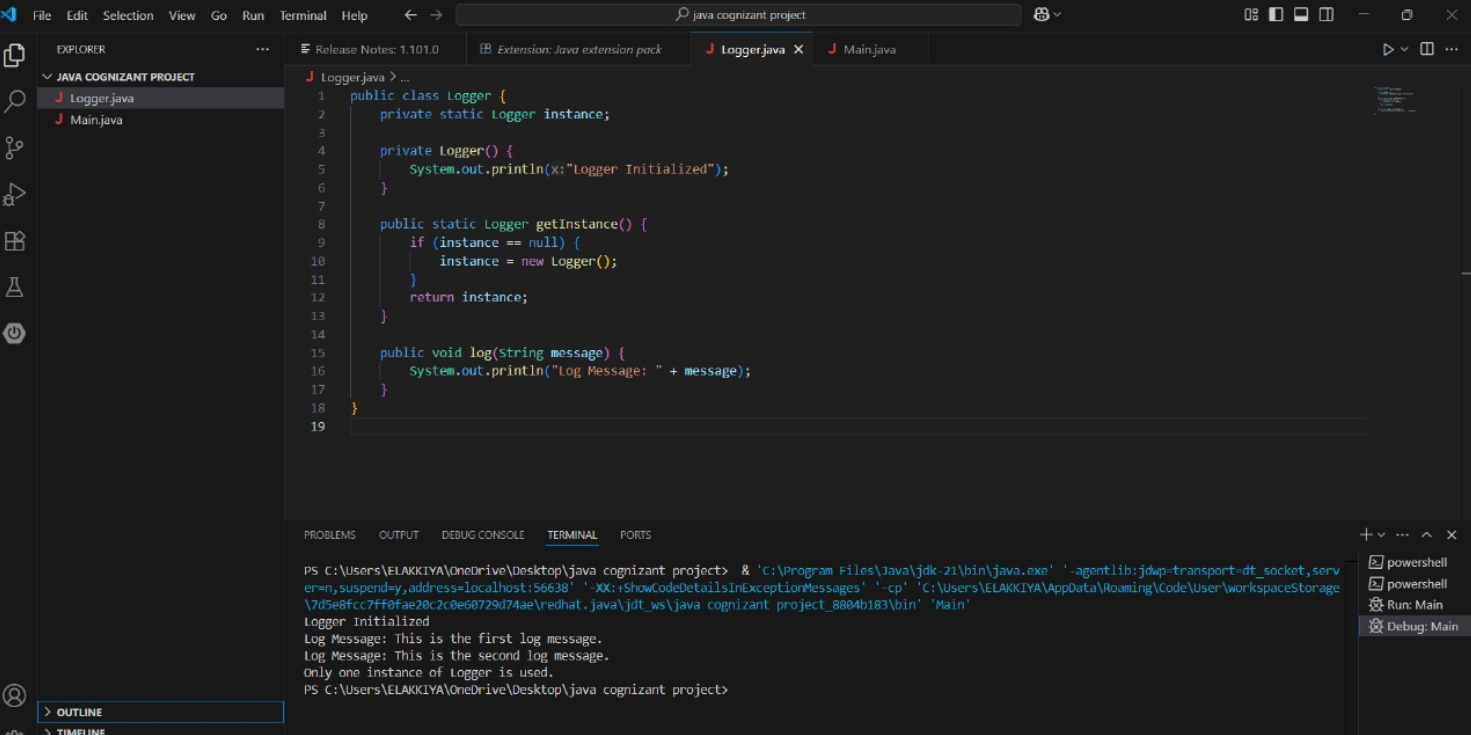
**Class Details :**

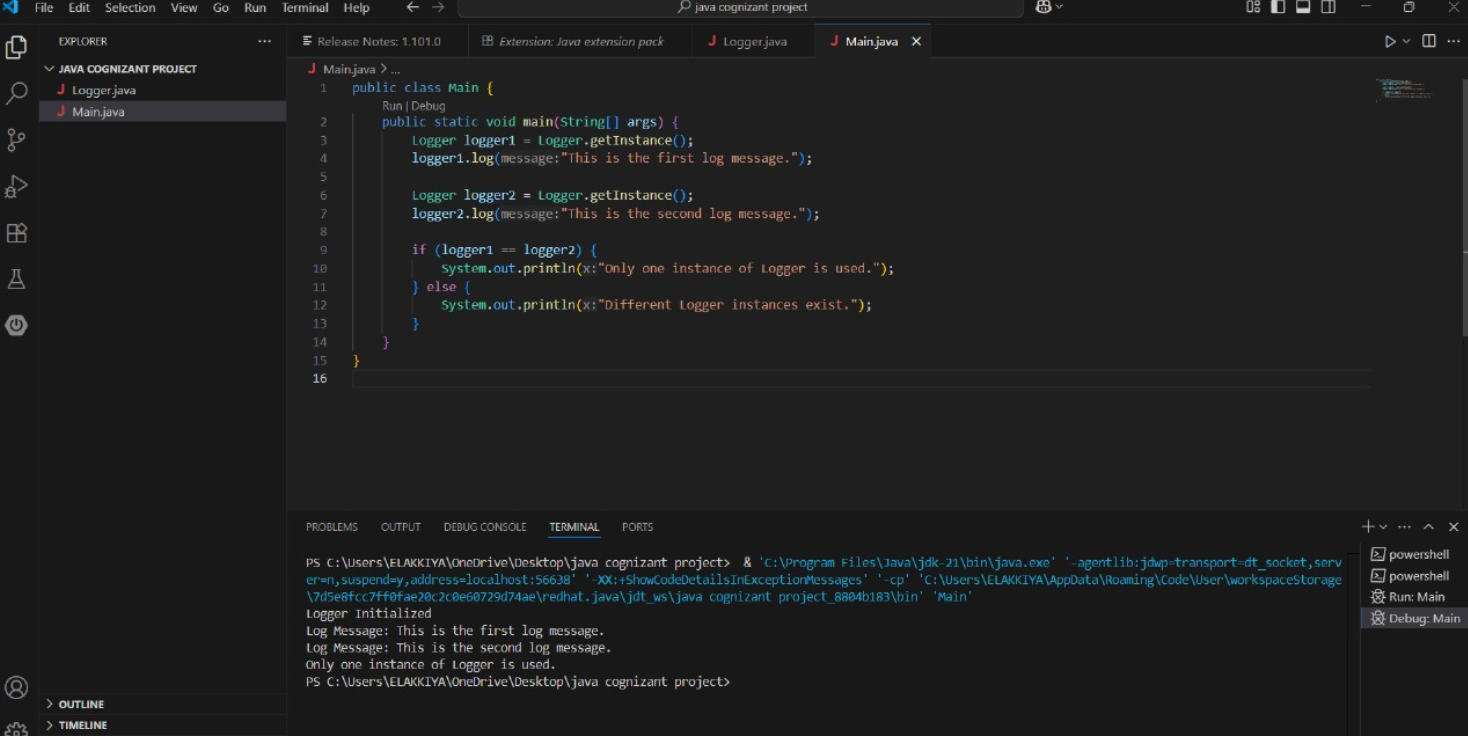
**Logger.java**

* Implements the Singleton pattern.
* Contains a private constructor.
* Holds a private static instance of itself.
* Provides a public static method getInstance() to access the instance.
* Includes a method log(String message) to print log messages.

**Main.java**

* Demonstrates the use of the Singleton Logger.
* Verifies that multiple calls to Logger.getInstance() return the same instance.

****

****

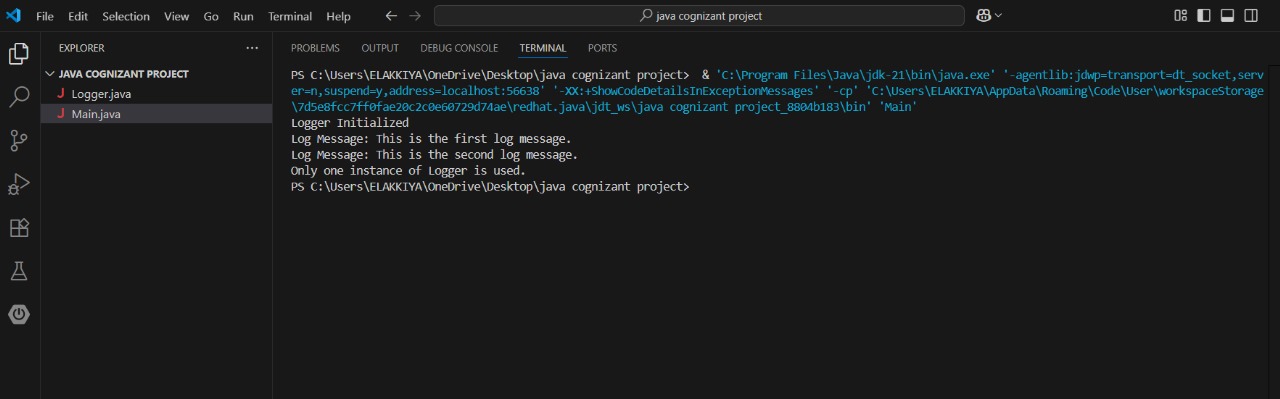
**Output Sample:**

**Logger Initialized**

**Log Message: This is the first log message.**

**Log Message: This is the second log message.**

**Only one instance of Logger is used.**



**Conclusion :**

In this exercise, we used the Singleton Pattern to ensure that only one instance of the **Logger class** is created.

This helps maintain consistent logging throughout the application and saves system resources. It is a useful pattern when a single shared object is needed.

2 .Project Title:

**Title: Factory Method Pattern – Document Management System**

**Objective :**

To implement the Factory Method Design Pattern in Java by creating a document management system that can handle different types of documents (Word, PDF, and Excel) using a modular and scalable design.

**Exercise Description:**

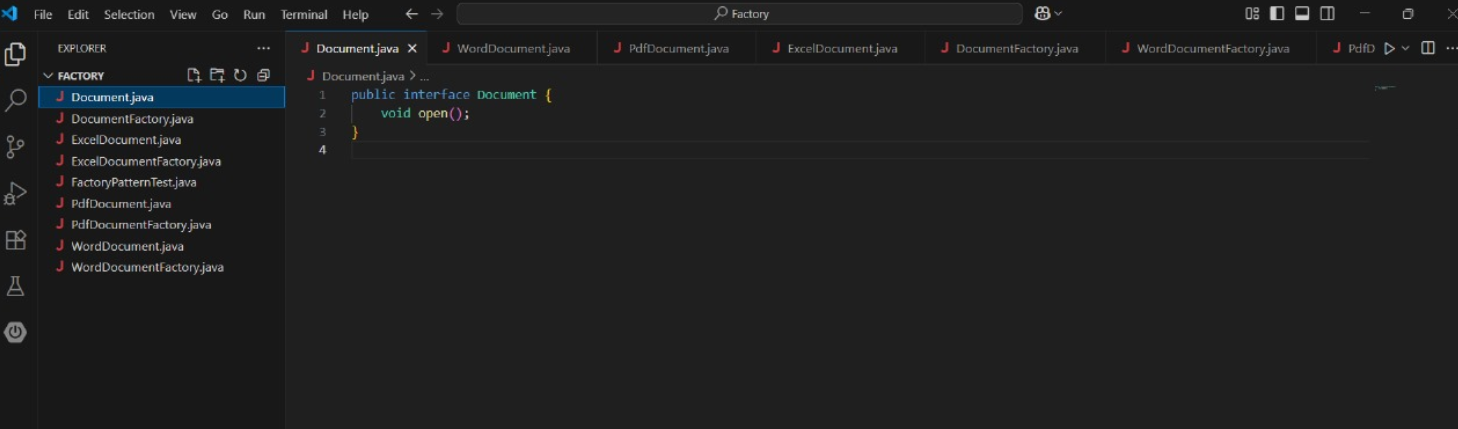
You are developing a document management system that needs to create different types of documents (e.g., Word, PDF, Excel). The Factory Method Pattern is applied to separate the instantiation logic from the client code, making the system flexible and easy to extend.

**Project Setup :**

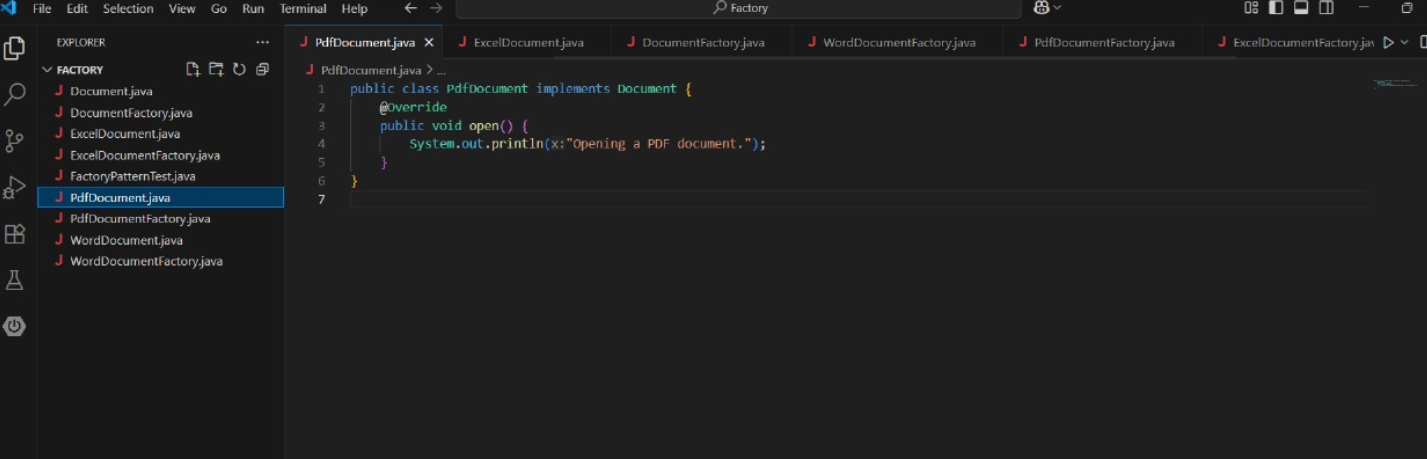
* **Project Name:** FactoryMethodPatternExample
* **Language:** Java
* **IDE Used:** Visual Studio Code
* **JDK Version:** 21

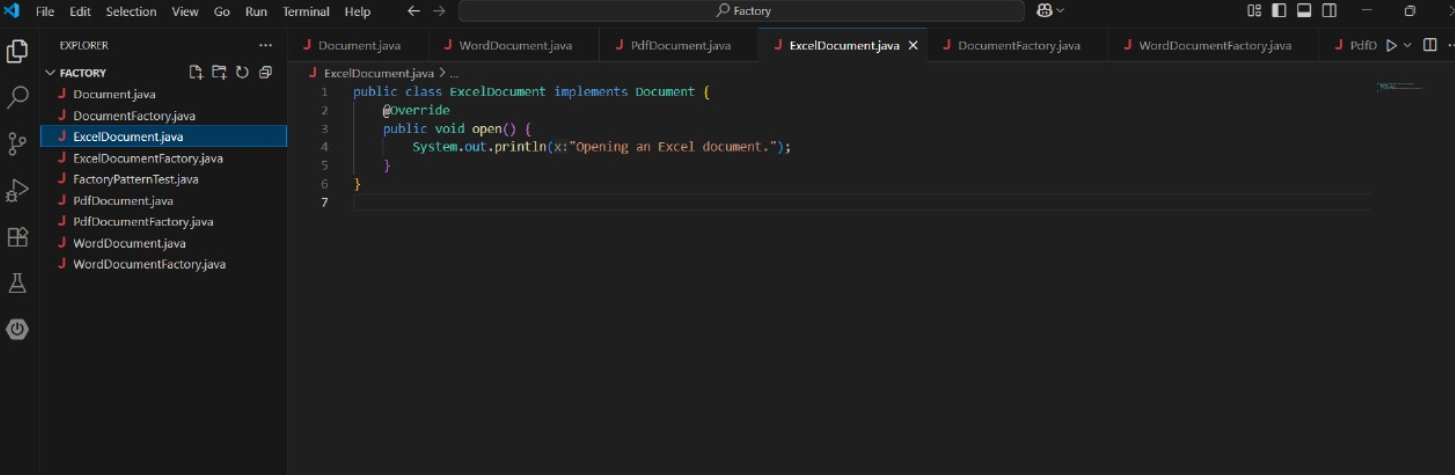
**Class Overview:**

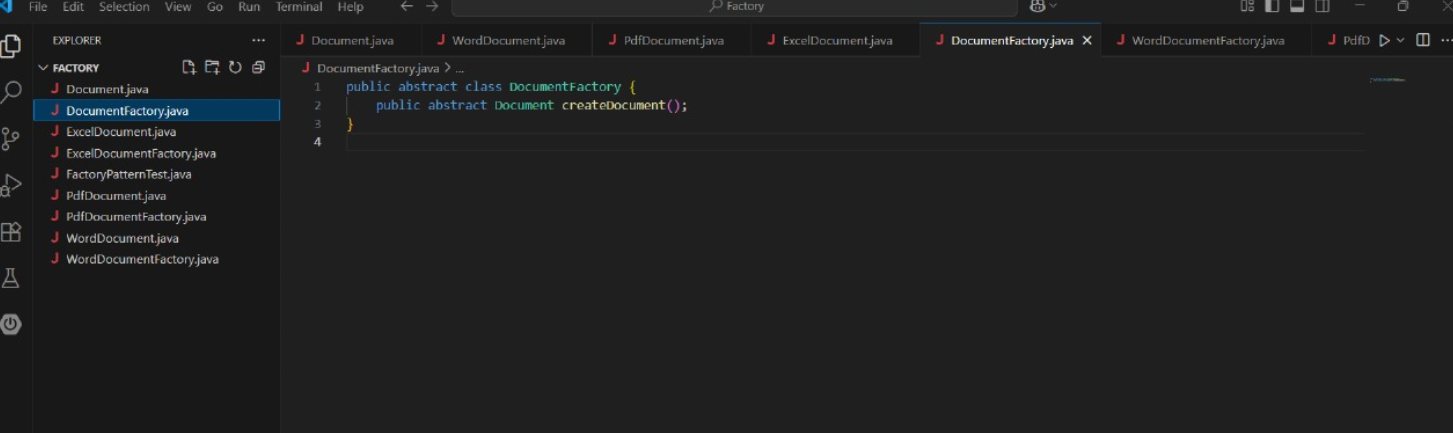
| **Class Name** | **Description** |
| --- | --- |
| Document | Interface for all document types with an open() method. |
| WordDocument | Implements Document, opens Word documents. |
| PdfDocument | Implements Document, opens PDF documents. |
| ExcelDocument | Implements Document, opens Excel documents. |
| DocumentFactory | Abstract class with createDocument() method. |
| WordDocumentFactory | Concrete factory to create WordDocument. |
| PdfDocumentFactory | Concrete factory to create PdfDocument. |
| ExcelDocumentFactory | Concrete factory to create ExcelDocument. |
|  |  |
| FactoryPatternTest | Main test class to demonstrate the Factory Method usage. |

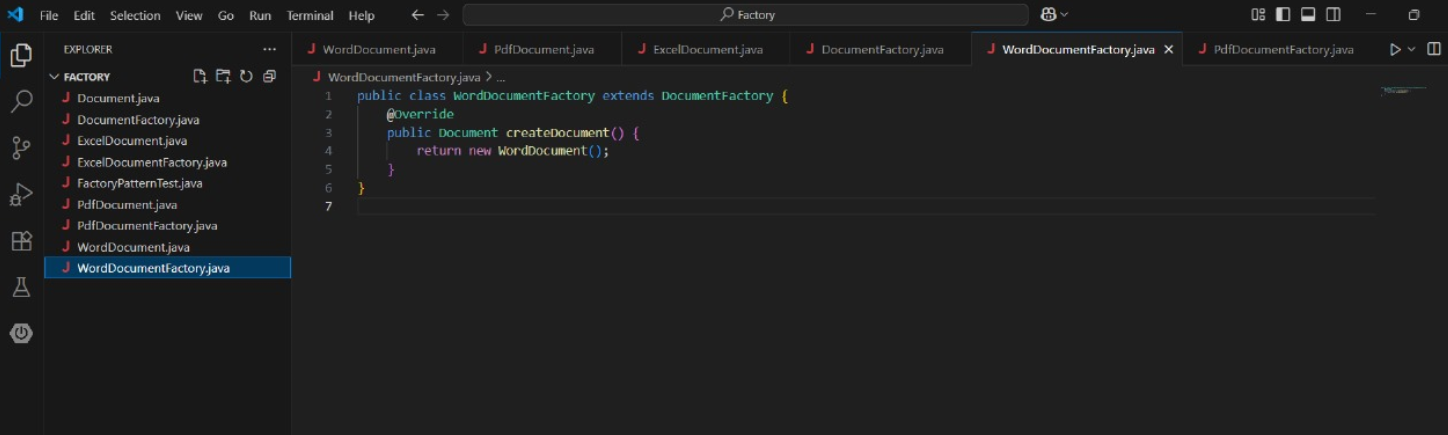


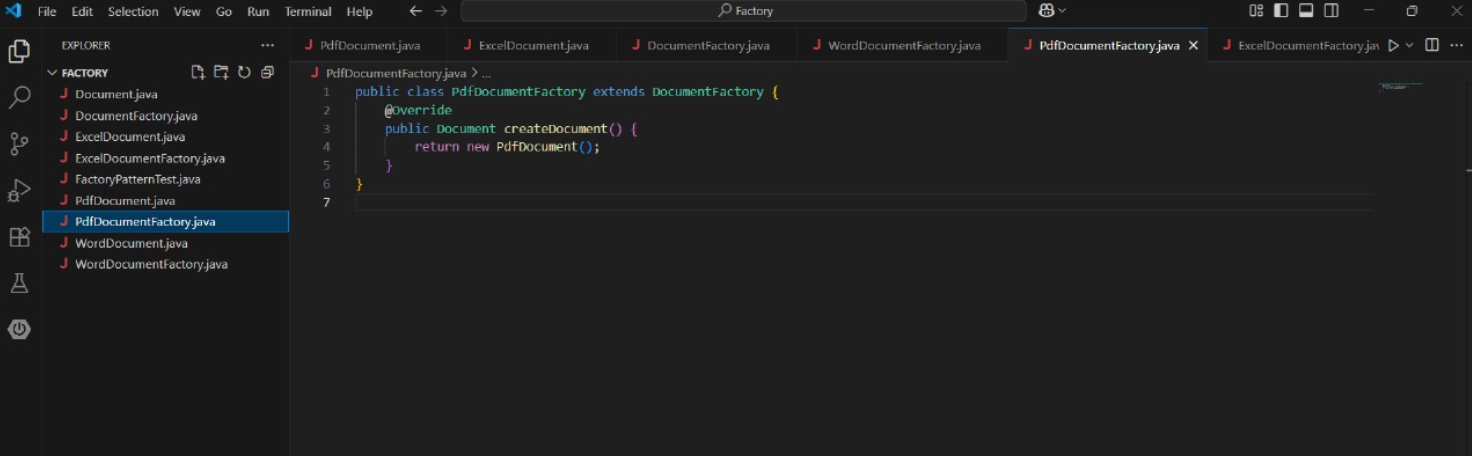


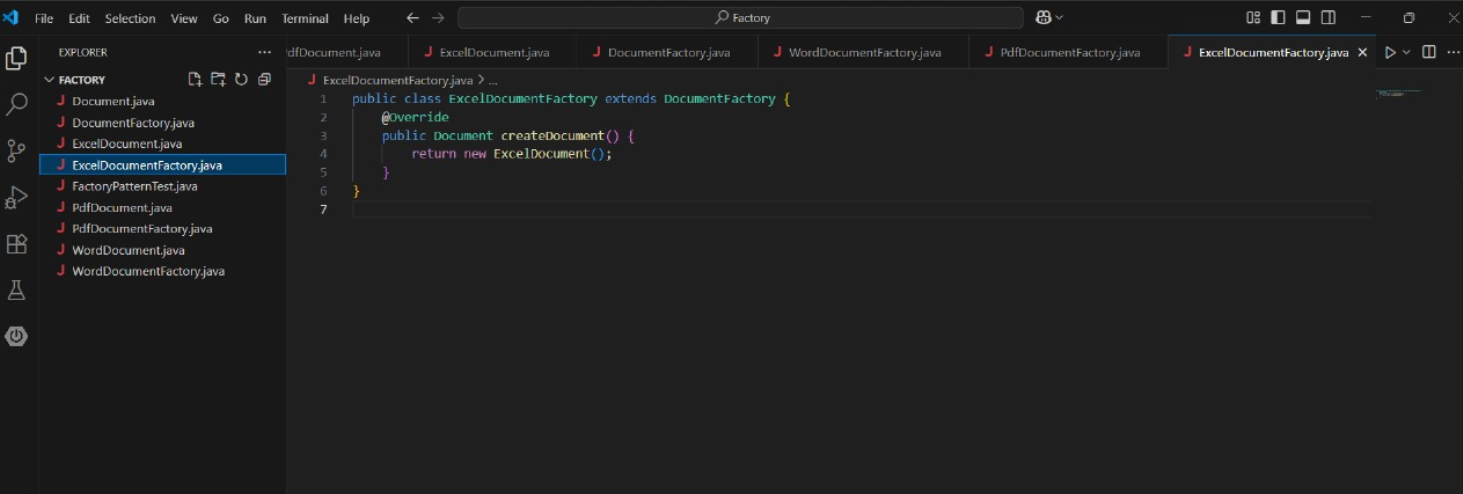


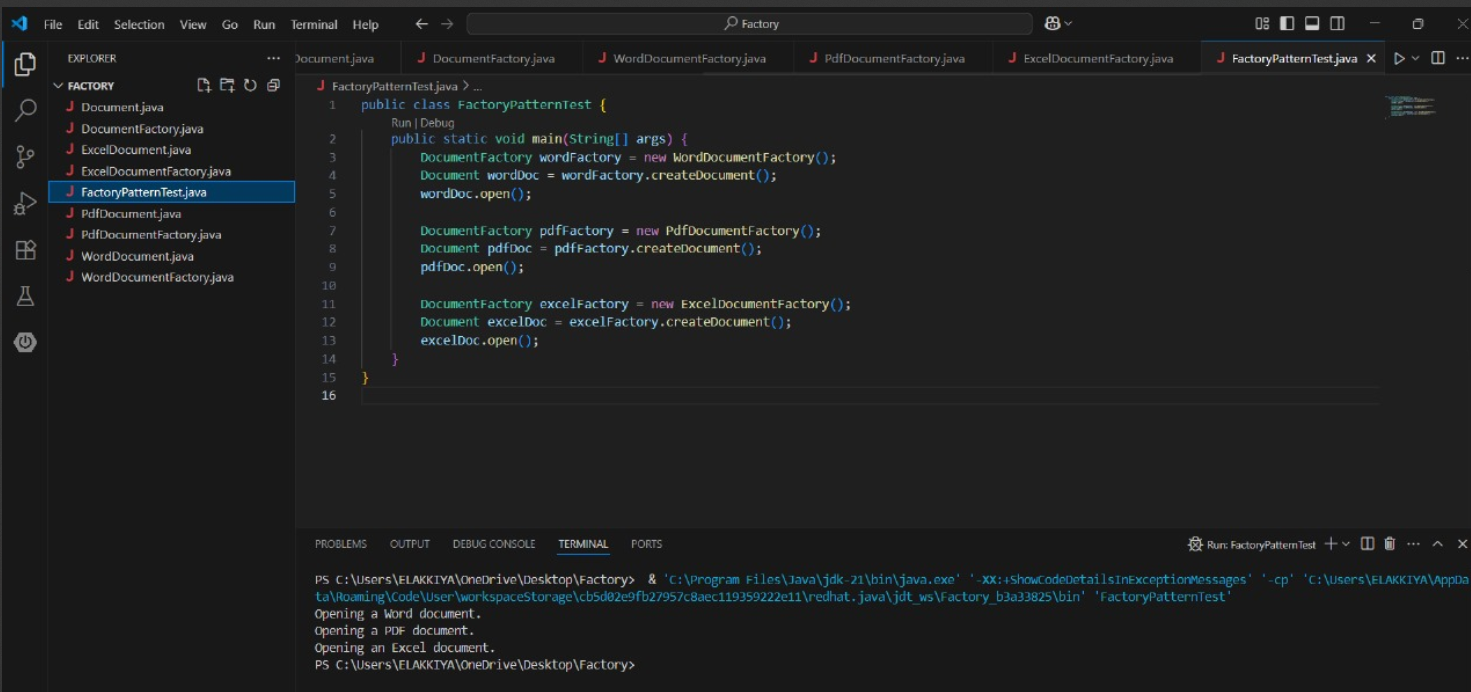












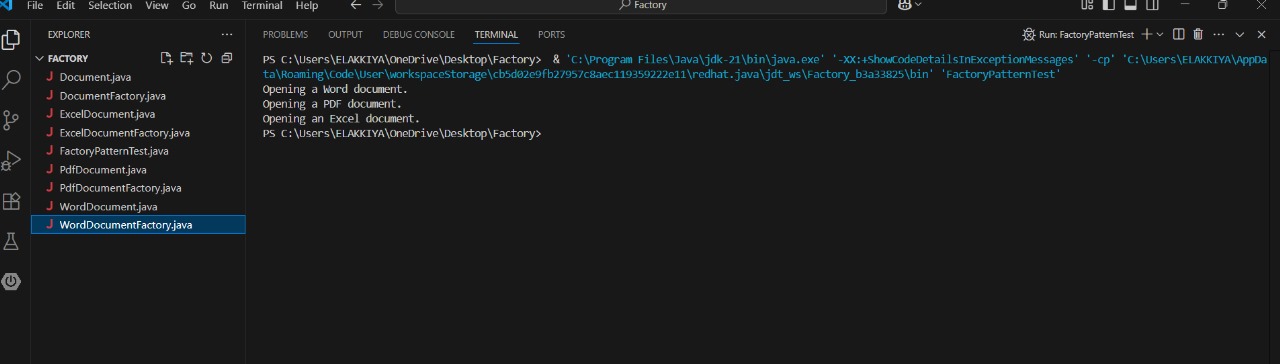
**Output of the Program:**

When FactoryPatternTest.java is executed, the output is:

Opening a Word document.

Opening a PDF document.

Opening an Excel document.



**Steps Followed :**

1. **Created Interface:**  
   Document interface with the open() method.
2. **Implemented Document Types:**  
   Classes WordDocument, PdfDocument, and ExcelDocument implemented the interface.
3. **Created Abstract Factory:**  
   DocumentFactory defines a factory method createDocument().
4. **Built Concrete Factories:**  
   WordDocumentFactory, PdfDocumentFactory, and ExcelDocumentFactory implemented createDocument().
5. **Demonstrated Functionality:**  
   A test class FactoryPatternTest used each factory to instantiate and use document objects.

**Conclusion :**

The project successfully demonstrates the **Factory Method Pattern** in a practical scenario of managing multiple document types. It helps to encapsulate the object creation logic and allows easy extension and maintenance in real-world software systems.

3.Project Title:

**E-commerce Platform Search Optimization Using Java**

**Exercise Overview**

This project implements and analyzes two search algorithms—**Linear Search** and **Binary Search**—within the context of an **E-commerce platform**. It focuses on the search functionality used to locate products quickly and efficiently.

**Objective :**

To develop a search module that demonstrates the difference between **linear** and **binary search** methods and their impact on **search performance** in real-world applications such as product lookups in an e-commerce site.

**Project Structure** :

EcommerceSearch/

│

├── Product.java

├── LinearSearch.java

├── BinarySearch.java

├── SearchDemo.java

**Implementation Summary :**

**1. Product.java**

Defines the Product class with attributes:

* productId
* productName
* category

**2. LinearSearch.java**

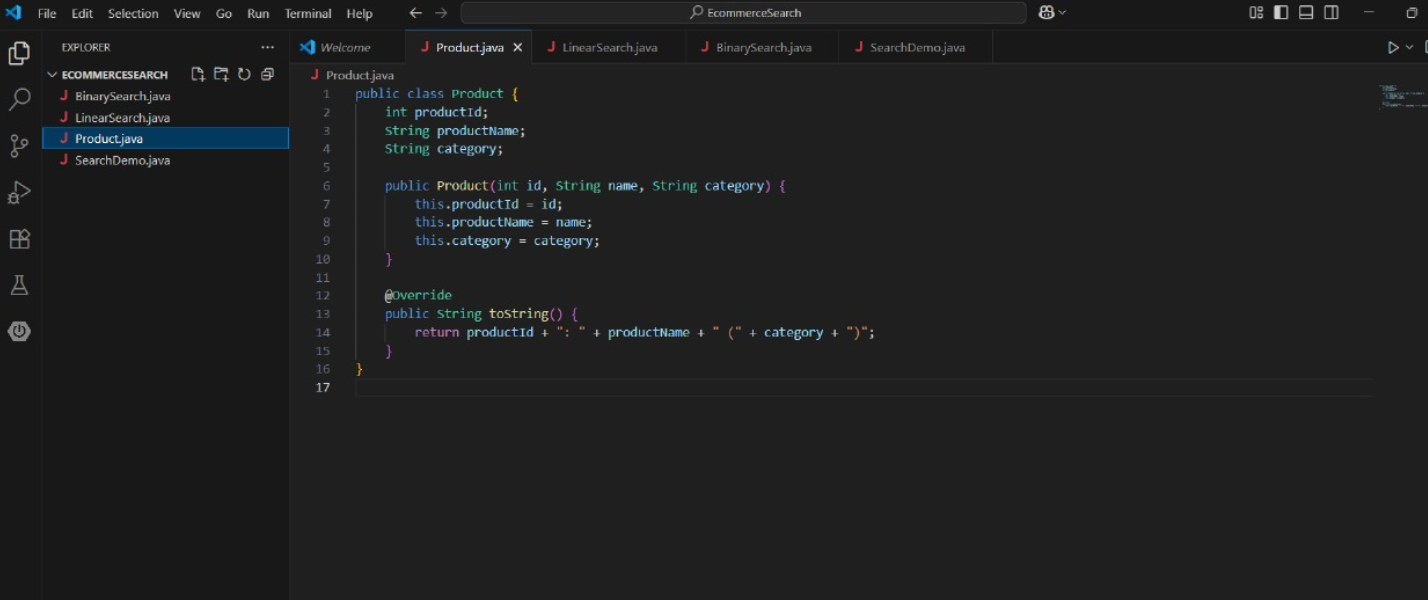
Performs a **linear search** over an unsorted array of products by name.

**3. BinarySearch.java**

Performs a **binary search** after sorting the array alphabetically.

**4. SearchDemo.java**

Contains the main() method to demonstrate both search types with sample data.





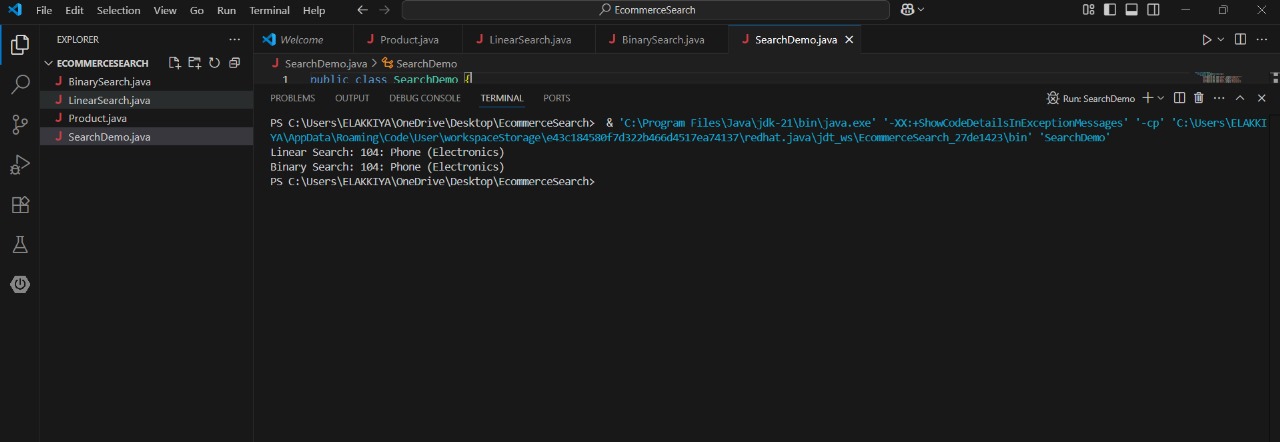




**Sample Output :**

Linear Search: 104: Phone (Electronics)

Binary Search: 104: Phone (Electronics)



**Performance Analysis :**

Binary Search performs better for large datasets but requires sorting.

Linear Search is simpler and better suited for small datasets or unsorted data.

**Conclusion :**

This project demonstrates the trade-offs between linear and binary search in terms of performance and data preparation. In a real-world e-commerce platform, search performance significantly affects user experience, and hence, optimized algorithms such as binary search or search indexing tools (e.g., Elasticsearch) are preferred.

4.Project Title:

**Financial Forecasting**

**Scenario :**

This project demonstrates a financial forecasting tool using recursive algorithms to calculate the future value of an investment based on past growth rates. It serves as a practical exercise in recursion and its optimization.

**Objectives :**

* Understand recursion and its application in problem solving.
* Implement a recursive method to forecast future values.
* Analyze the time complexity and explore optimization techniques.

**Step-by-Step Implementation :**

1. Understand Recursive Algorithms

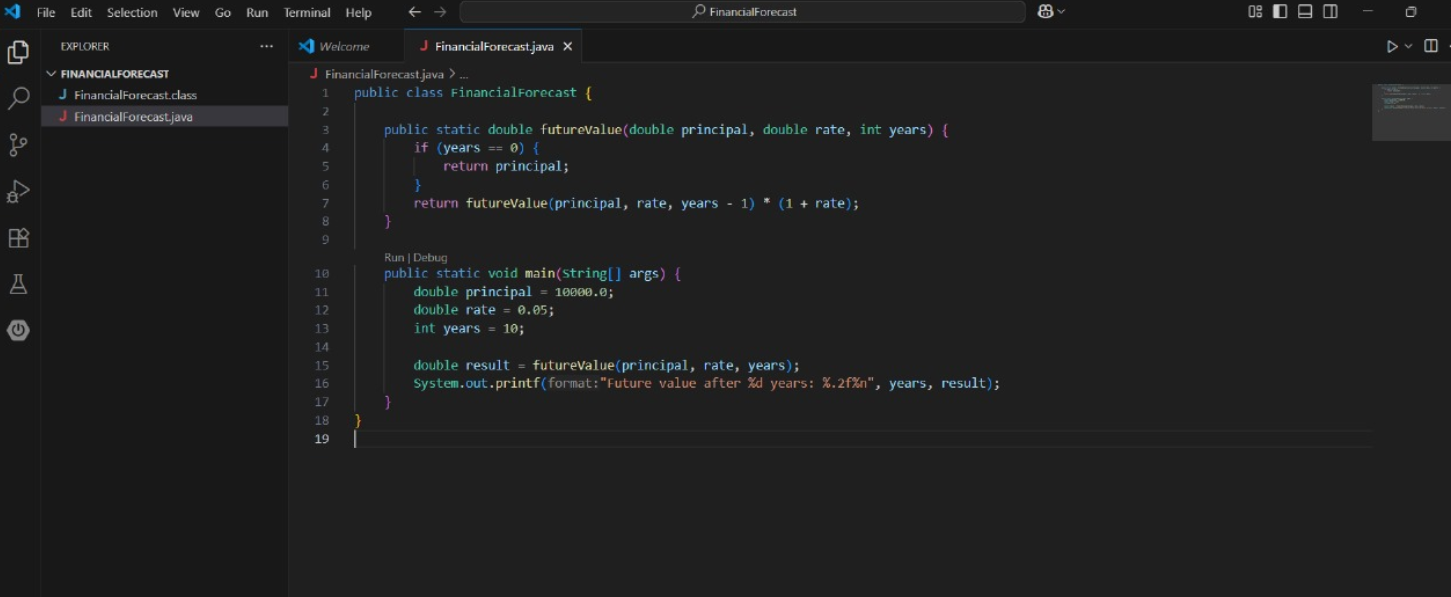
Recursion is a method where a function calls itself to solve smaller subproblems. In this case, recursion helps to calculate compound growth over multiple years.

2. Setup

A method futureValue(principal, rate, years) is implemented in Java to calculate the future value based on the formula:

**Future Value=Principal×(1+Rate)^Years**

**Java Code (Recursive Version)**

****

**How to Run :**

Run using:

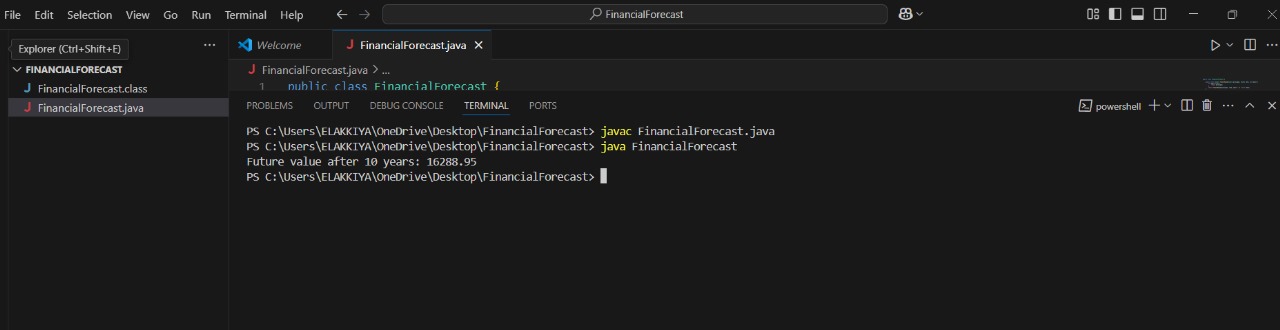
* Run button OR
* Terminal commands:

**javac FinancialForecast.java**

**java FinancialForecast**

**Output :**

Future value after 10 years: 16288.95



**Time Complexity :**

* Recursive: O(n) time, O(n) space due to recursive call stack
* Iterative version (optimized): O(n) time, O(1) space

**Conclusion :**

Recursion provides a simple and elegant solution for problems like financial forecasting. However, for performance-critical applications, an iterative approach is preferred due to its efficiency and low memory usage.