MANDATORY HANDS-ON

WEEK-1

**DESIGN PRINCIPLES AND PATTERNS**

**Exercise 1: Implementing the Singleton Pattern**

**Java Project Name: SingletonPatternExample**

**Logger.java**

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);

    }

}

**Main.java**

public class Main {

    public static void main(String[] args) {

        Logger logger1 = Logger.getInstance();

        logger1.log("Logging message from logger1.");

        Logger logger2 = Logger.getInstance();

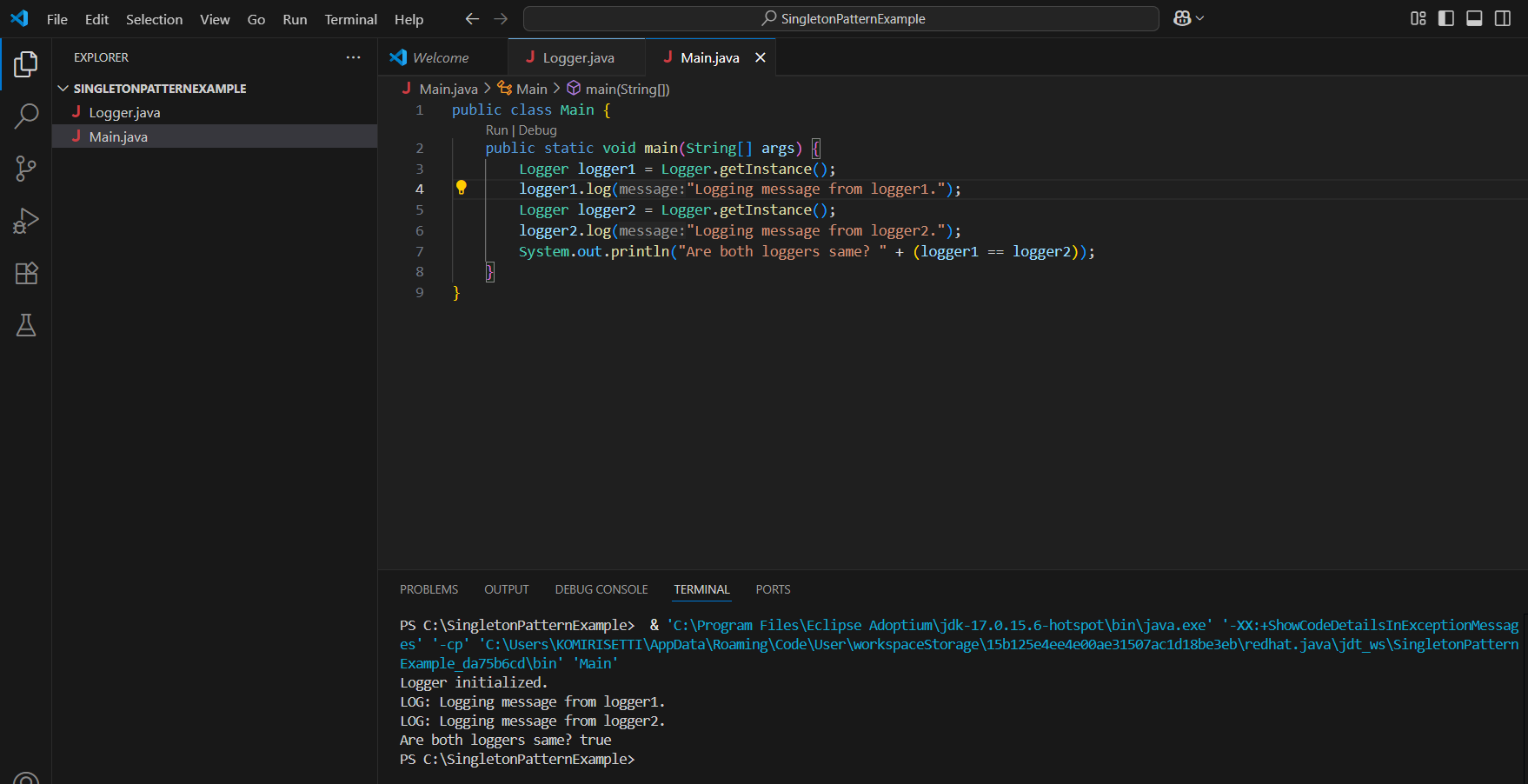
        logger2.log("Logging message from logger2.");

        System.out.println("Are both loggers same? " + (logger1 == logger2));

    }

}

**Output:**

****

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

**Java Project Name: FactoryMethodPatternExample**

**Document.java**

interface Document {

    void create();

}

**WordDocument.java**

class WordDocument implements Document {

    public void create() {

        System.out.println("Creating a Word document...");

    }

}

**PdfDocument.java**

class PdfDocument implements Document {

    public void create() {

        System.out.println("Creating a PDF document...");

    }

}

**ExcelDocument.java**

class ExcelDocument implements Document {

    public void create() {

        System.out.println("Creating an Excel document...");

    }

}

**DocumentFactory.java**

abstract class DocumentFactory {

    abstract Document createDocument();

}

**WordDocumentFactory.java**

class WordDocumentFactory extends DocumentFactory {

    Document createDocument() {

        return new WordDocument();

    }

}

**PdfDocumentFactory.java**

class PdfDocumentFactory extends DocumentFactory {

    Document createDocument() {

        return new PdfDocument();

    }

}

**ExcelDocumentFactory.java**

class ExcelDocumentFactory extends DocumentFactory {

    Document createDocument() {

        return new ExcelDocument();

    }

}

**Main.java**

class FactoryMain {

    public static void main(String[] args) {

        DocumentFactory wordFactory = new WordDocumentFactory();

        Document wordDoc = wordFactory.createDocument();

        wordDoc.create();

        DocumentFactory pdfFactory = new PdfDocumentFactory();

        Document pdfDoc = pdfFactory.createDocument();

        pdfDoc.create();

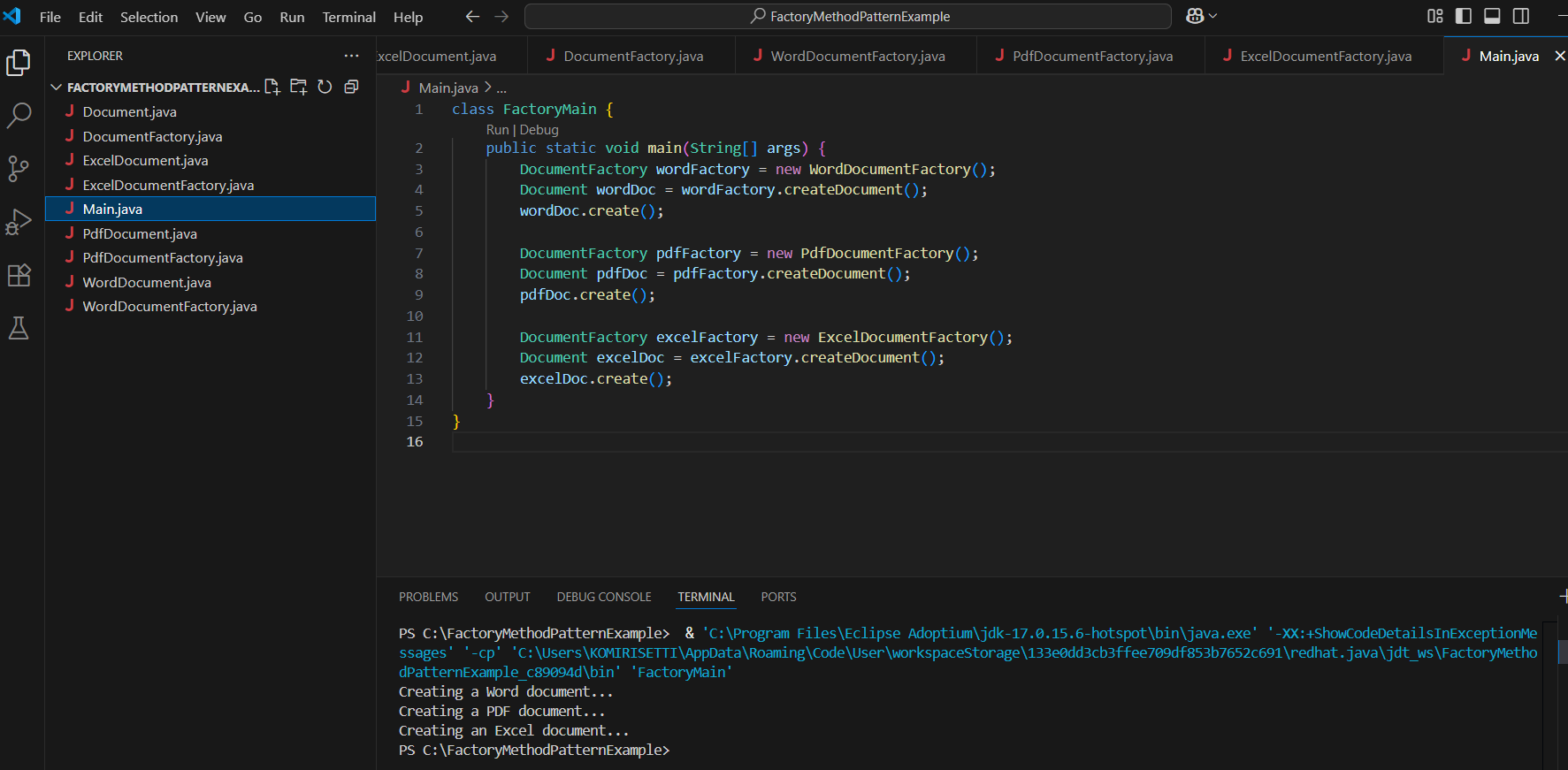
        DocumentFactory excelFactory = new ExcelDocumentFactory();

        Document excelDoc = excelFactory.createDocument();

        excelDoc.create();

    }

}

**Output:**

**DATA STRUCTURES AND ALGORITHMS**

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

**Java Project Name : E-commerce Platform Search Function**

**Step 1:**

Big O Notation is a mathematical notation used to describe the efficiency of an algorithm in terms of time and space complexity. It gives us an upper bound on the running time as the input size grows.

Time Complexities for Search:

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

* Best case: When the item is found in the first comparison.
* Average case: When the item is found somewhere in the middle.
* Worst case: When the item is not found or is at the last position (linear) / the search space is reduced completely (binary).

**Step 2:**

**Product.java**

public 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;

    }

}

**Step 3:**

**SearchEngine.java**

public class SearchEngine {

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

        for (int i = 0; i < products.length; i++) {

            if (products[i].productName.equalsIgnoreCase(name)) {

                return i;

            }

        }

        return -1;

    }

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

        int low = 0, high = products.length - 1;

        while (low <= high) {

            int mid = (low + high) / 2;

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

            if (cmp == 0) return mid;

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

            else high = mid - 1;

        }

        return -1;

    }

}

**Main.java**

import java.util.Arrays;

import java.util.Comparator;

public class Main {

    public static void main(String[] args) {

        Product[] products = {

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

            new Product(202, "Chair", "Furniture"),

            new Product(203, "Book", "Education")

        };

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

        int index1 = SearchEngine.linearSearch(products, "Chair");

        int index2 = SearchEngine.binarySearch(products, "Chair");

        System.out.println("Linear Search Index: " + index1);

        System.out.println("Binary Search Index: " + index2);

    }

}

**Step 4:**

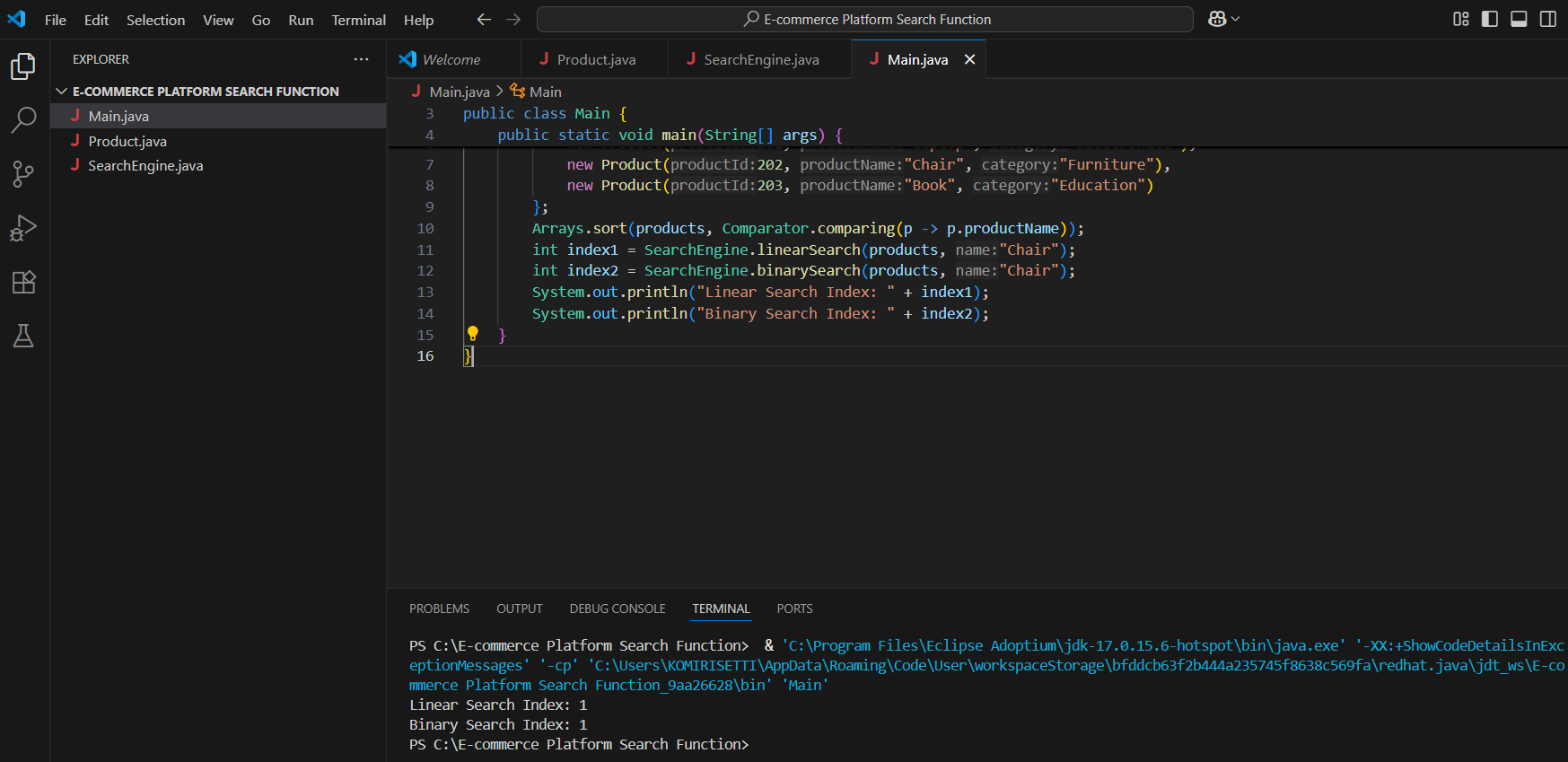
**Time Complexity Comparison:**

| Algorithm | Time Complexity |
| --- | --- |
| Linear Search | O(n) |
| Binary Search | O(log n), requires sorted data |

Suitability:

* Linear Search is simple and works on unsorted data, but is slow for large datasets.
* Binary Search is much faster, but the data must be sorted.
* For an e-commerce platform with large datasets, Binary Search is more suitable because it performs much faster due to logarithmic complexity.

**Output:**

****

**Exercise 7: Financial Forecasting**

**Java Project Name : Financial Forecasting**

**Step 1:**

Recursion is a programming technique where a function calls itself to solve smaller instances of a problem until it reaches a base condition. It is particularly useful when a problem can be broken down into similar subproblems.

In the context of financial forecasting, recursion simplifies the process of applying the compound interest formula repeatedly over a number of years. Instead of using loops, recursion allows us to express the logic more naturally and elegantly.

For example, to forecast the future value with a constant growth rate, we can define:

FutureValue(years) = FutureValue(years - 1) \* (1 + rate)

This approach continues until the number of years reaches zero, at which point the present value is returned. Thus, recursion offers a clean and structured way to calculate the result over multiple periods.

**Step 2:**

* I created a Java class named FinancialForecast.
* Inside this class, I defined a recursive method called forecast() that takes three parameters:
* presentValue (initial investment or value),
* rate (growth rate per year),
* years (number of years to forecast).

**Step 3:**

**FinancialForecast.java**

public class FinancialForecast {

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

        if (years == 0) return presentValue;

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

    }

    public static void main(String[] args) {

        double presentValue = 10000;

        double rate = 0.05;

        int years = 5;

        double futureValue = forecast(presentValue, rate, years);

        System.out.println("Future Value (Recursive): " + futureValue);

    }

}

**Step 4:**

**Time Complexity of the Recursive Algorithm:**

* The recursive method calls itself once for every year, decreasing the years by 1 each time until it reaches 0.
* Therefore, the **time complexity is O(n)**, where n is the number of years.

**Space Complexity:**

* Since each recursive call is added to the call stack, the **space complexity is also O(n)**.

**Issues with Recursion:**

* For **large values of years**, recursion may lead to a **stack overflow** due to deep call stacks.
* Each call holds memory until the base case is reached, which increases **memory usage**.

**Optimization Techniques:**

To avoid excessive computation and memory usage, we can optimize the recursive solution by:

1. **Using Iteration Instead**:

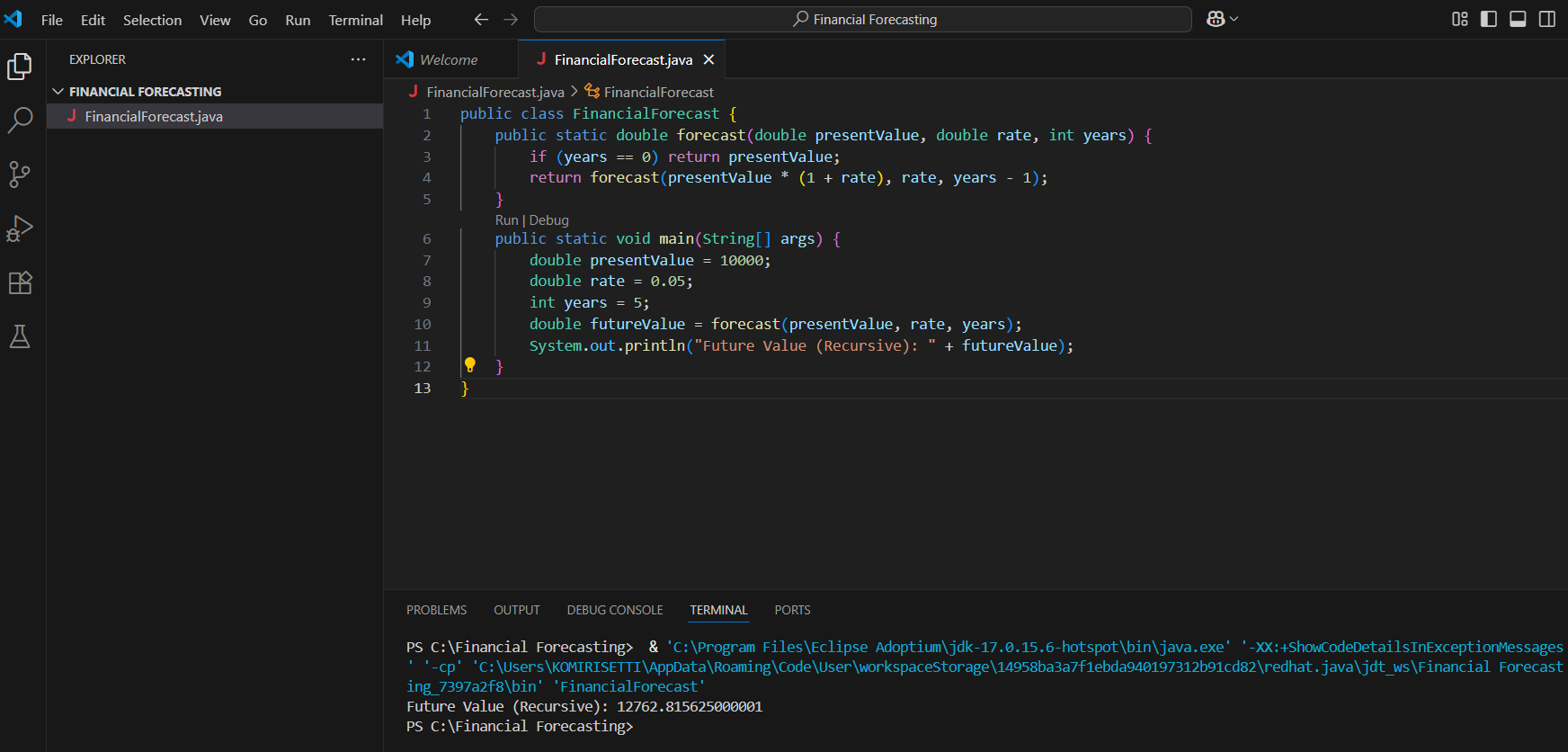
* A loop-based approach avoids call stack buildup.
* Time complexity remains **O(n)** but **space complexity reduces to O(1)**.

1. **Using the Compound Interest Formula** (if allowed):

Future Value=Present Value×(1+rate)years

* Time complexity: **O(1)**
* Space complexity: **O(1)**
* This is the most efficient method but replaces recursion.

**Output:**

****