**Design Patterns & Principles**

**Exercise 1: Implementing the Singleton Pattern**

**Scenario:**

You need to ensure that a logging utility class in your application has only one instance throughout the application lifecycle to ensure consistent logging.

**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("Application started");

        Logger logger2 = Logger.getInstance();

        logger2.log("Still running");

        if (logger1 == logger2) {

            System.out.println("Both instances are the same.");

        } else {

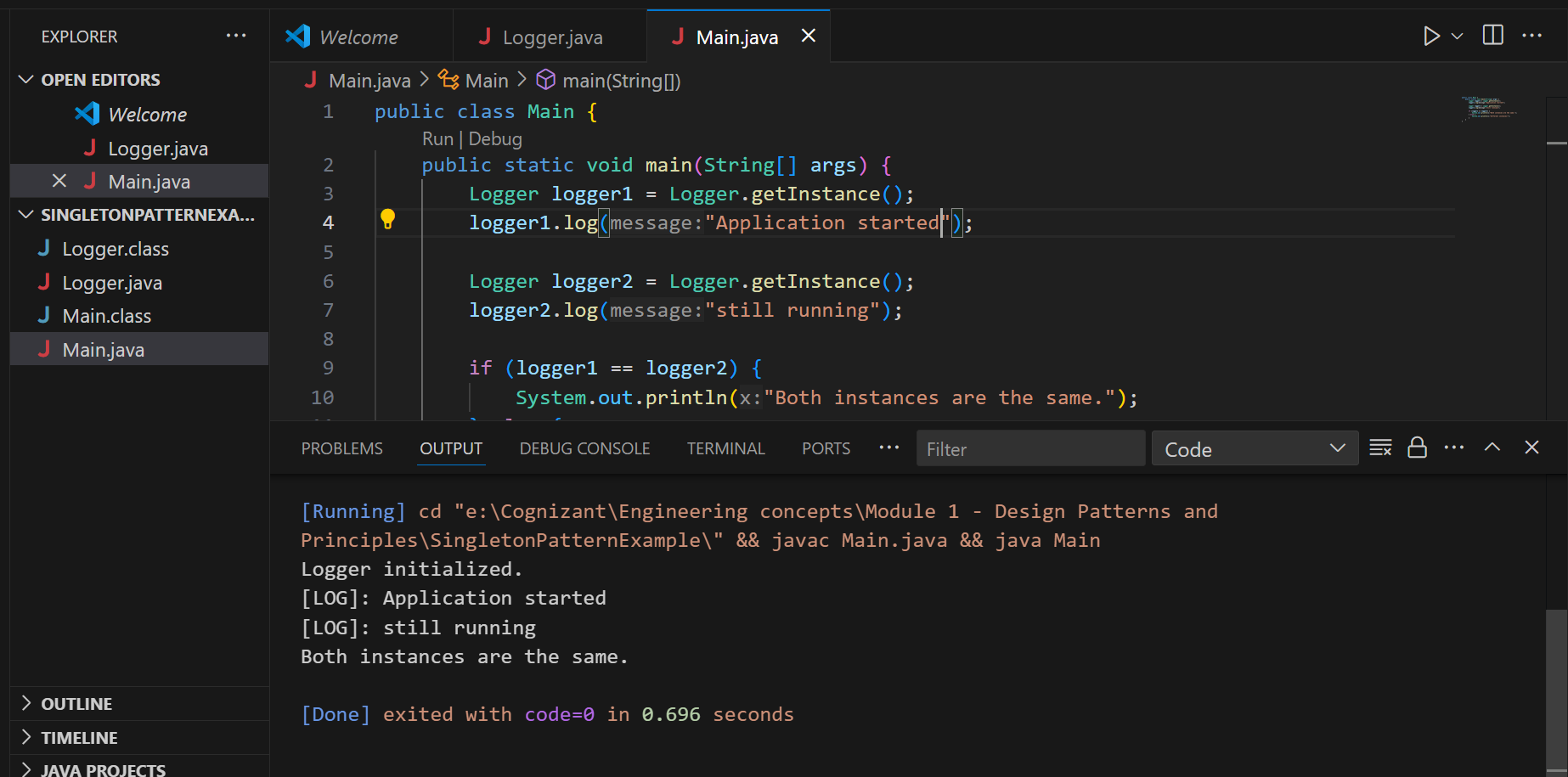
            System.out.println("Different instances!");

        }

    }

}

**Output**

****

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

**Scenario:**

You are developing a document management system that needs to create different types of documents (e.g., Word, PDF, Excel). Use the Factory Method Pattern to achieve this.

Ans).

1. **Document.java – Base Interface**

public interface Document {

    void open();

}

1. **WordDocument.java – Concrete class**

public class WordDocument implements Document {

    @Override

    public void open() {

        System.out.println("Opening Word document");

    }

}

1. **PdfDocument.java – Concrete class**

public class PdfDocument implements Document {

    @Override

    public void open() {

        System.out.println("Opening PDF document");

    }

}

1. **ExcelDocument.java – Concrete class**

public class ExcelDocument implements Document {

    @Override

    public void open() {

        System.out.println("Opening Excel document");

    }

}

1. **DocumentFactory.java – Abstract factory class**

public abstract class DocumentFactory {

    public abstract Document createDocument();

}

1. **WordDocumentFactory.java – Concrete factory class**

public class WordDocumentFactory extends DocumentFactory {

    @Override

    public Document createDocument() {

        return new WordDocument();

    }

}

1. **PdfDocumentFactory.java – Concrete factory class**

public class PdfDocumentFactory extends DocumentFactory {

    @Override

    public Document createDocument() {

        return new PdfDocument();

    }

}

1. **ExcelDocumentFactory.java – Concrete factory class**

public class ExcelDocumentFactory extends DocumentFactory {

    @Override

    public Document createDocument() {

        return new ExcelDocument();

    }

}

1. **Main.java**

public class Main {

    public static void main(String[] args) {

        DocumentFactory wordFactory = new WordDocumentFactory();

        Document wordDoc = wordFactory.createDocument();

        wordDoc.open();

        DocumentFactory pdfFactory = new PdfDocumentFactory();

        Document pdfDoc = pdfFactory.createDocument();

        pdfDoc.open();

        DocumentFactory excelFactory = new ExcelDocumentFactory();

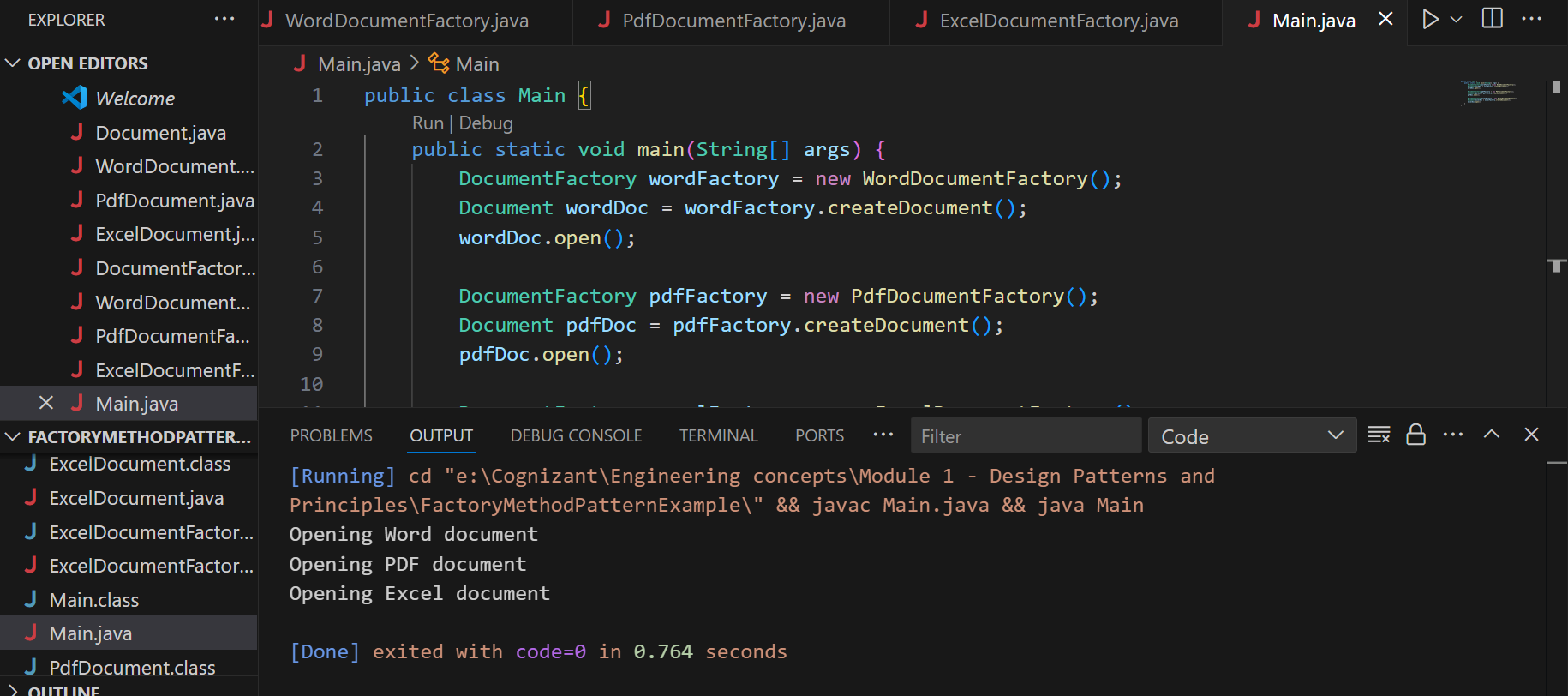
        Document excelDoc = excelFactory.createDocument();

        excelDoc.open();

    }

}

**Output:**

****

**Data Structures & Algorithms**

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

**Scenario:**

You are working on the search functionality of an e-commerce platform. The search needs to be optimized for fast performance.

1). **Understand Asymptotic Notation:**

**Explain Big O notation and how it helps in analyzing algorithms.**

Big O notation describes the **time complexity** or **performance** of an algorithm as input size grows.

It tells you **how fast** or **slow** an algorithm is in:

Best Case

Average Case

Worst Case

**Describe the best, average, and worst-case scenarios for search operations.**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **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) |
|  | |  | |  | |  |
|  | |  | |  | |  |
|  |  | |  | |
|  |  | |  | |
|  | |  | |  | |  |

**Code Implementation:**

**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 + ")";

    }

}

**SearchMethods.java**

import java.util.Arrays;

import java.util.Comparator;

public class SearchMethods {

    // 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

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

        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 sortProductsByName(Product[] products) {

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

    }

}

**Main.java**

public class Main {

    public static void main(String[] args) {

         Product[] products = {

            new Product(1, "Phone Charger", "Electronics"),

            new Product(2, "Pen", "Stationery"),

            new Product(3, "Shoes", "Clothing"),

            new Product(4, "Hand Bags", "Accessories"),

        };

        String searchItem = "Shoes";

        //Linear Search

        System.out.println("Linear Search:");

        Product result1 = SearchMethods.linearSearch(products, searchItem);

        if (result1 != null)

            System.out.println("Found: " + result1);

        else

            System.out.println("Product not found!");

        //Binary Search

        SearchMethods.sortProductsByName(products);

        System.out.println("\nBinary Search:");

        Product result2 = SearchMethods.binarySearch(products, searchItem);

        if (result2 != null)

            System.out.println("Found: " + result2);

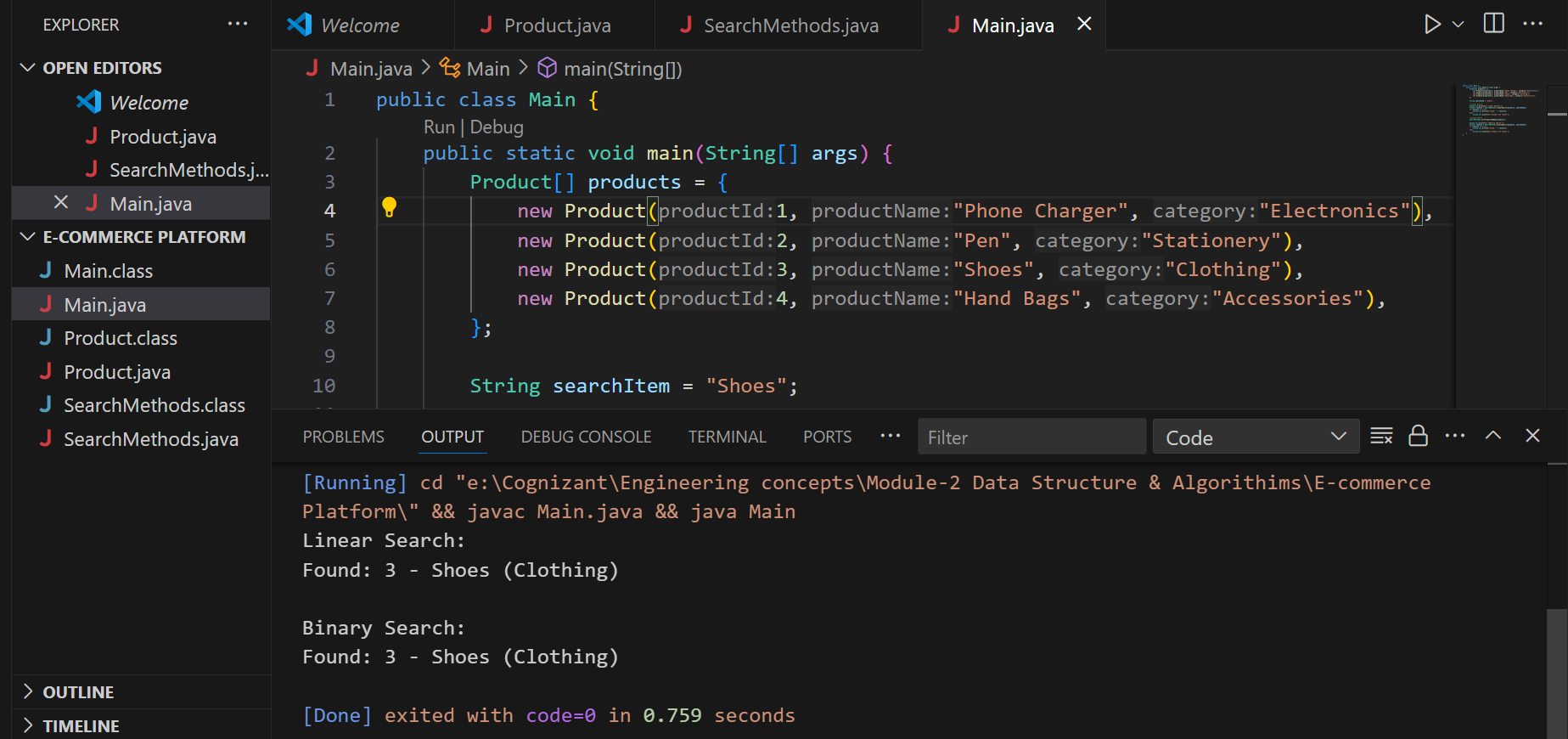
        else

            System.out.println("Product not found!");

    }

}

**OUTPUT:**



**Analysis:**

* + Compare the time complexity of linear and binary search algorithms.
  + Discuss which algorithm is more suitable for your platform and why.

|  |  |  |  |
| --- | --- | --- | --- |
| **Algorithm** | **Best Case** | **Average Case** | **Worst Case** |
| Linear Search | O(1) | O(n/2) = O(n) | O(n) |
| Binary Search | O(1) | O(log n) | O(log n) |

**Linear Search** is easy but slower for large data.

**Binary Search** is **much faster** but needs **sorted data**.

For an optimized e-commerce platform, **Binary Search** is **more suitable** for name-based searches with a sorted list.

**Exercise 7: Financial Forecasting**

**Scenario:**

You are developing a financial forecasting tool that predicts future values based on past data.

**Steps:**

1. **Understand Recursive Algorithms:**
   * Explain the concept of recursion and how it can simplify certain problems.
2. **Setup:**
   * Create a method to calculate the future value using a recursive approach.
3. **Implementation:**
   * Implement a recursive algorithm to predict future values based on past growth rates.
4. **Analysis:**
   * Discuss the time complexity of your recursive algorithm.
   * Explain how to optimize the recursive solution to avoid excessive computation.

**Ans.** 1). Recursion is a technique where a method calls **itself** to solve a smaller version of the same problem.

Examples:

* Factorial calculation
* Fibonacci numbers
* Tree traversal
* And here: **financial forecasting over years based on growth rate**

**Code Implementation:**

**Forecast.java**

public class Forecast {

    // Recursive method

    public static double futureValue(double initialVal, double growthRate, int years) {

        if (years == 0) {

            return initialVal;

        }

        return futureValue(initialVal, growthRate, years - 1) \* (1 + growthRate);

    }

    // using memoization (to avoid recomputation)

    public static double futureValMemo(double initialVal, double growthRate, int years, double[] dp) {

        if (years == 0) {

            return initialVal;

        }

        if (dp[years] != 0) {

            return dp[years];

        }

        dp[years] = futureValMemo(initialVal, growthRate, years - 1, dp) \* (1 + growthRate);

        return dp[years];

    }

}

**Main.java**

public class Main {

    public static void main(String[] args) {

        double initialVal = 1000.0;

        double growthRate = 0.10;

        int years = 5;

        System.out.println("Financial Forecast (Recursive):");

        double futureValue1 = Forecast.futureValue(initialVal, growthRate, years);

        System.out.printf("Value after %d years: Rs %.2f\n", years, futureValue1);

        System.out.println("\nFinancial Forecast (Memoization):");

        double[] dp = new double[years + 1];

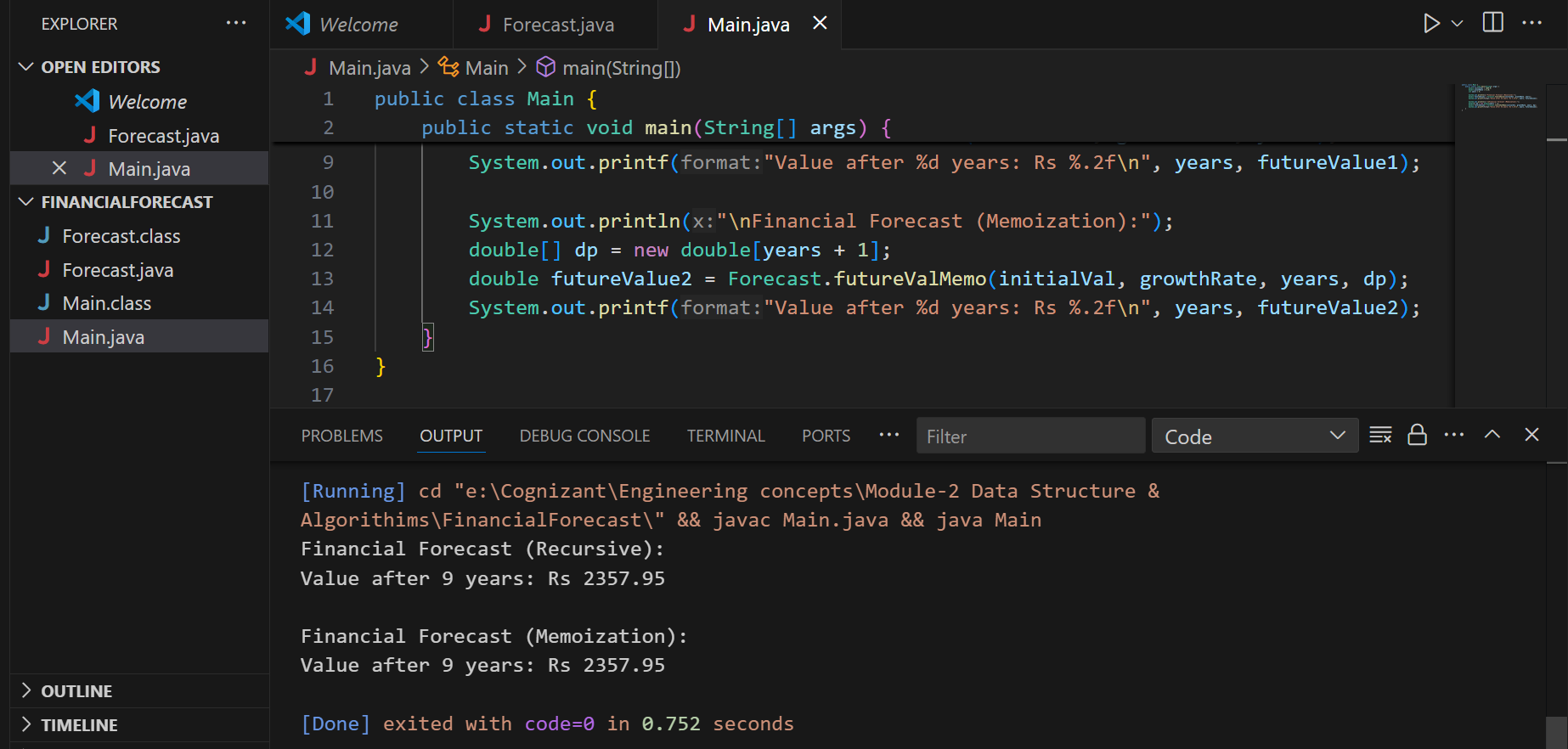
        double futureValue2 = Forecast.futureValMemo(initialVal, growthRate, years, dp);

        System.out.printf("Value after %d years: Rs %.2f\n", years, futureValue2);

    }

}

**Output:**

****

|  |  |  |
| --- | --- | --- |
| **Approach** | **Time Complexity** | **Space Complexity** |
| Recursive (Normal) | O(n) | O(n) call stack |
| Recursive + Memoization | O(n) | O(n) call stack +Array |

**Basic recursion** recalculates values many times.

**Memoization** avoids recomputation by storing previously computed results.

To optimize recursion:

1. **Using memoization** to store results in an array.
2. Or we can switch to **iterative approach** for better space usage.