**Week-1 Design Patterns and Principles**

1 - Implementing the Singleton Pattern

CODE :

package week1.ex1;

public class Logger {

    private static Logger instance;

    private Logger() {

        System.out.println("Logger instance created.");

    }

    public static Logger getInstance() {

        if (instance == null) {

            instance = new Logger();

        }

        return instance;

    }

    public void log(String message) {

        System.out.println("Log: " + message);

    }

}

package week1.ex1;

public class SingletonTest {

    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("Both logger1 and logger2 are the same instance.");

        } else {

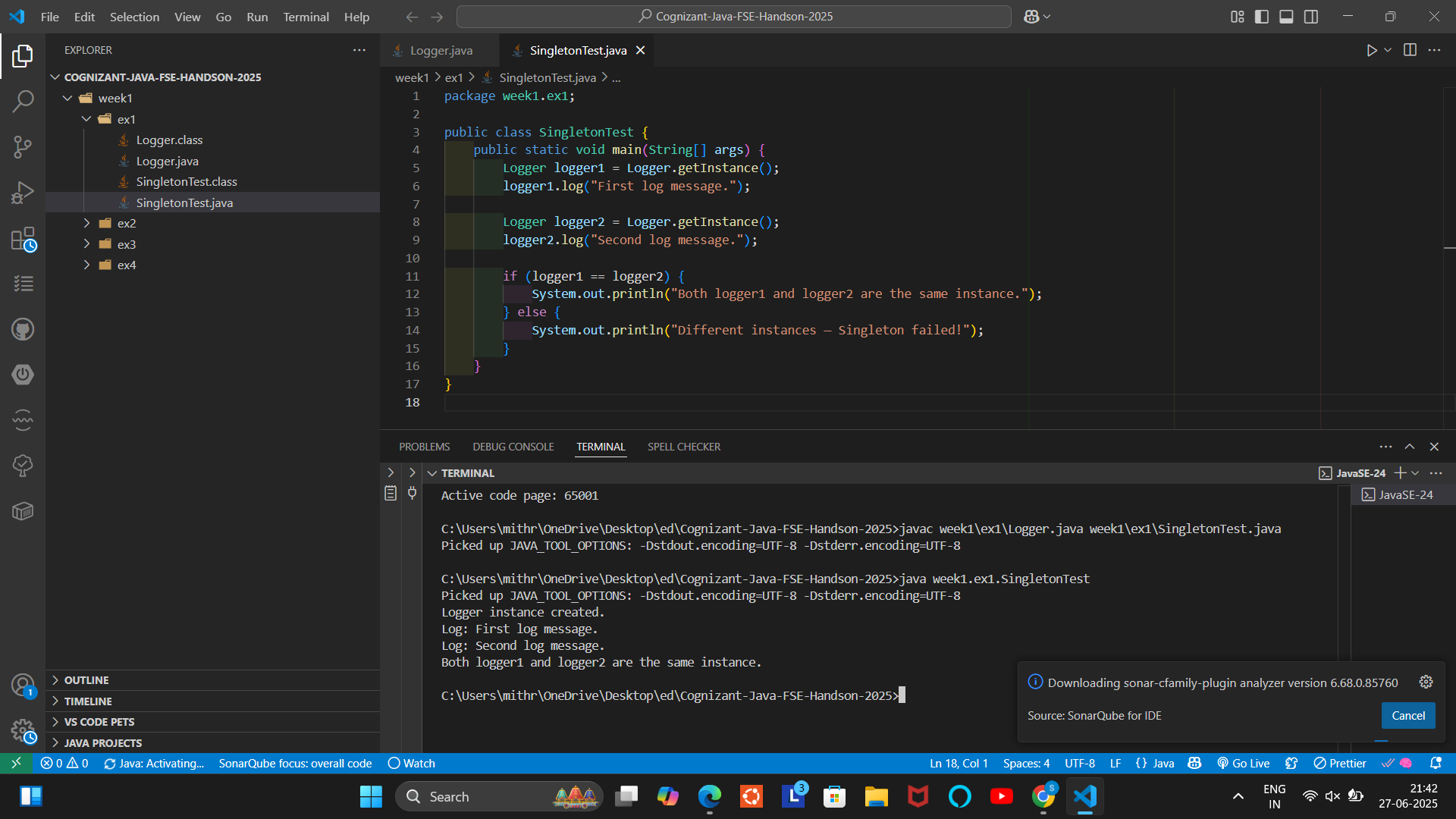
            System.out.println("Different instances — Singleton failed!");

        }

    }

}

OUTPUT :



2 - Implementing the Factory Method Pattern

CODE :

package week1.ex2;

public class FactoryPatternTest {

    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();

    }

}

package week1.ex2;

public interface Document {

    void open();

}

package week1.ex2;

public abstract class DocumentFactory {

    public abstract Document createDocument();

}

package week1.ex2;

public class ExcelDocument implements Document {

    public void open() {

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

    }

}

package week1.ex2;

public class ExcelDocumentFactory extends DocumentFactory {

    public Document createDocument() {

        return new ExcelDocument();

    }

}

package week1.ex2;

public class PdfDocument implements Document {

    public void open() {

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

    }

}

package week1.ex2;

public class PdfDocumentFactory extends DocumentFactory {

    public Document createDocument() {

        return new PdfDocument();

    }

}

package week1.ex2;

public class WordDocument implements Document {

    public void open() {

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

    }

}

package week1.ex2;

public class WordDocumentFactory extends DocumentFactory {

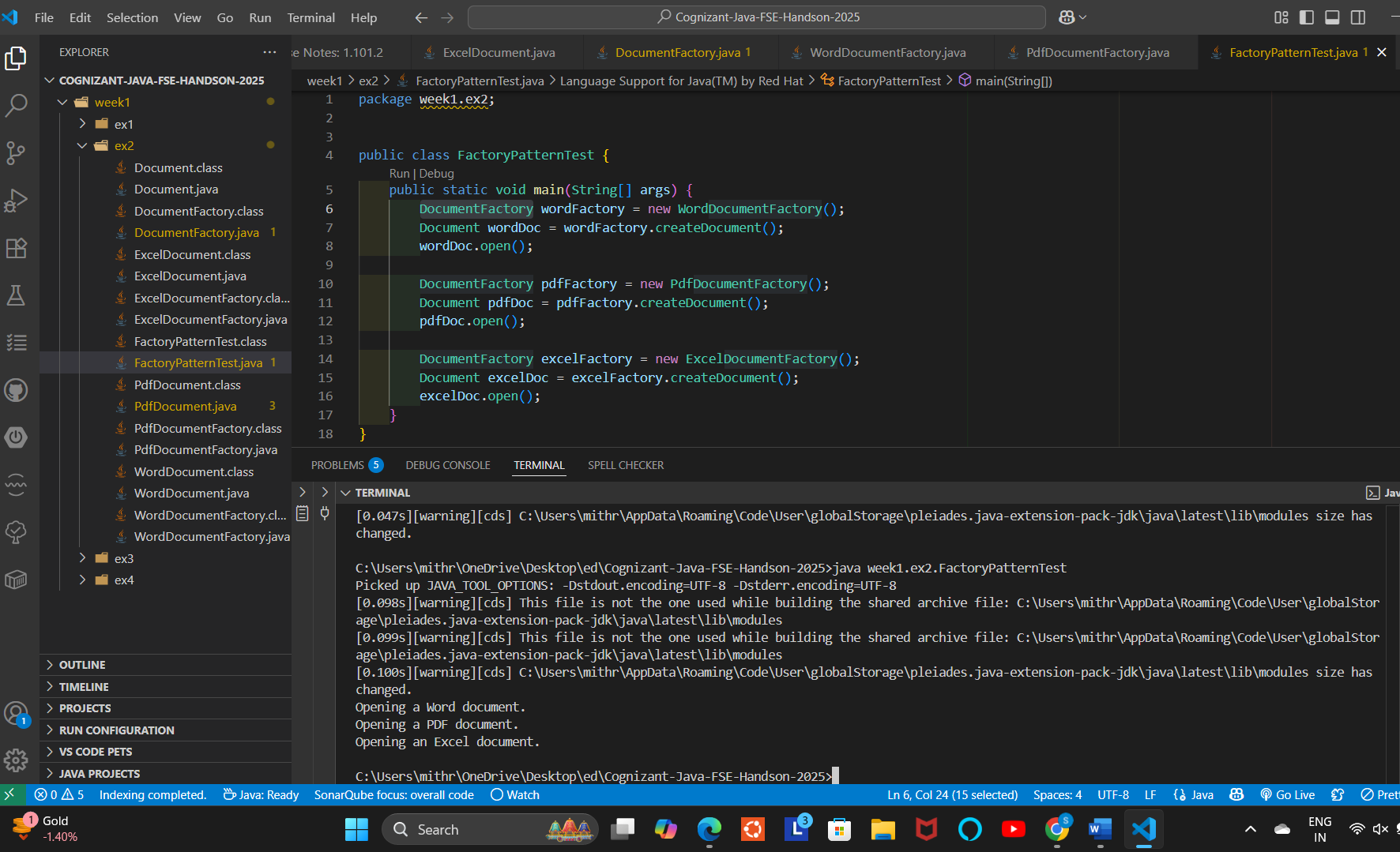
    public Document createDocument() {

        return new WordDocument();

    }

}

OUTPUT :



3 – E-commerce Platform Search Function

CODE :

package week1.ex3;

public class Product {

    private int id;

    private String name;

    private String category;

    private double price;

    public Product(int id, String name, String category, double price) {

        this.id = id;

        this.name = name.toLowerCase();

        this.category = category.toLowerCase();

        this.price = price;

    }

    public String getName() { return name; }

    public String getCategory() { return category; }

    public double getPrice() { return price; }

    @Override

    public String toString() {

        return "Product{" +

                "id=" + id +

                ", name='" + name + '\'' +

                ", category='" + category + '\'' +

                ", price=" + price +

                '}';

    }

}

package week1.ex3;

import java.util.\*;

public class ProductSearch {

    public static void main(String[] args) {

        List<Product> products = Arrays.asList(

            new Product(1, "iPhone 14", "Electronics", 69999),

            new Product(2, "Samsung Galaxy S21", "Electronics", 49999),

            new Product(3, "Nike Running Shoes", "Footwear", 3299),

            new Product(4, "Adidas Sports Shoes", "Footwear", 3899),

            new Product(5, "Sony Headphones", "Electronics", 2599),

            new Product(6, "The Alchemist Book", "Books", 399)

        );

        ProductTrie trie = new ProductTrie();

        for (Product p : products) {

            trie.insert(p);

        }

        Scanner scanner = new Scanner(System.in);

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

        String prefix = scanner.nextLine();

        List<Product> results = trie.searchByPrefix(prefix);

        if (results.isEmpty()) {

            System.out.println("No products found with prefix: " + prefix);

        } else {

            System.out.println("Products found:");

            for (Product p : results) {

                System.out.println(p);

            }

        }

    }

}

package week1.ex3;

import java.util.\*;

class TrieNode {

    Map<Character, TrieNode> children = new HashMap<>();

    List<Product> products = new ArrayList<>();

    boolean isEndOfWord = false;

}

public class ProductTrie {

    private TrieNode root;

    public ProductTrie() {

        root = new TrieNode();

    }

    public void insert(Product product) {

        TrieNode node = root;

        String name = product.getName();

        for (char c : name.toCharArray()) {

            node.children.putIfAbsent(c, new TrieNode());

            node = node.children.get(c);

            node.products.add(product);

        }

        node.isEndOfWord = true;

    }

    public List<Product> searchByPrefix(String prefix) {

        TrieNode node = root;

        prefix = prefix.toLowerCase();

        for (char c : prefix.toCharArray()) {

            if (!node.children.containsKey(c)) {

                return Collections.emptyList();

            }

            node = node.children.get(c);

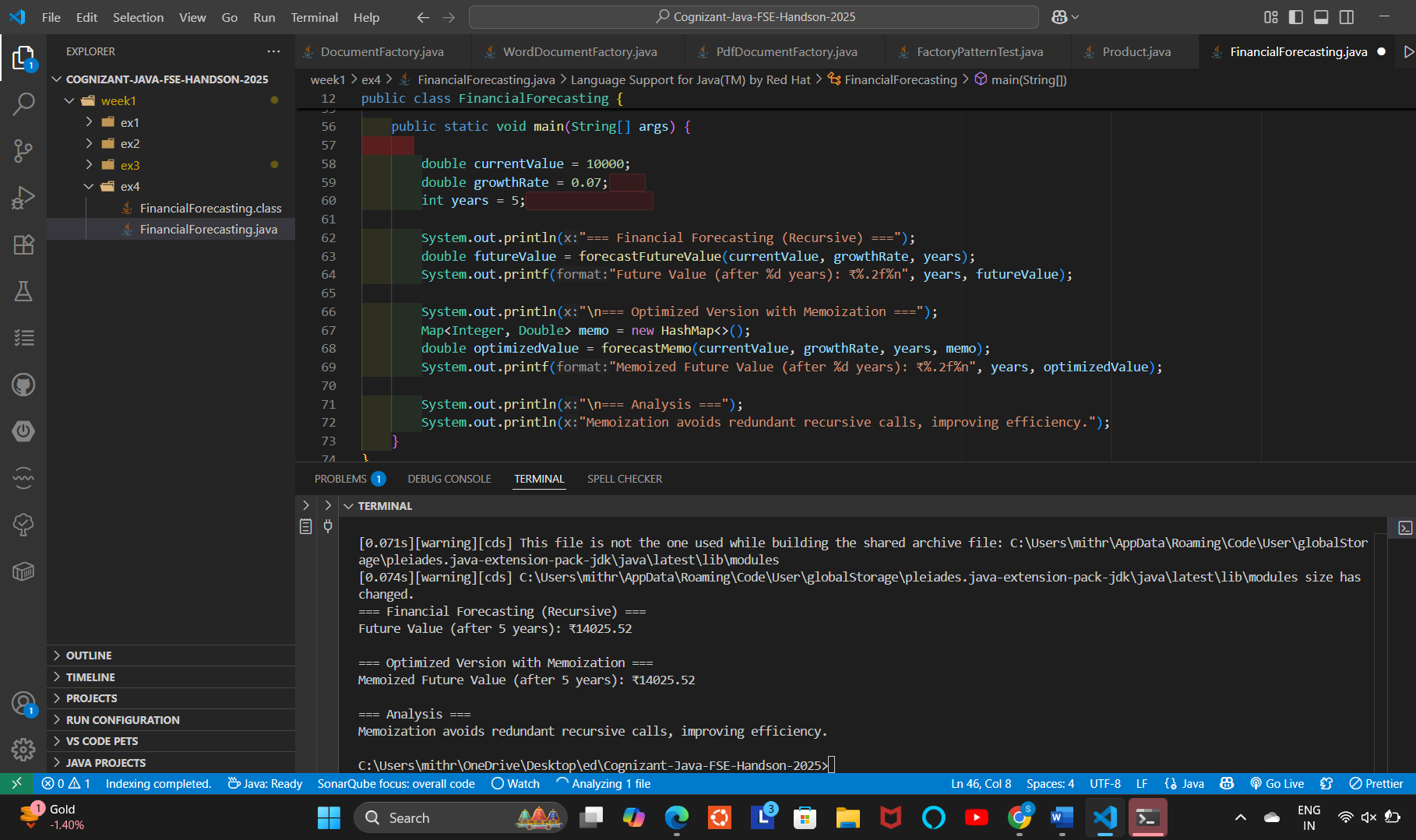
        }

        return node.products;

    }

}

OUTPUT



4 - Financial Forecasting

CODE :

package week1.ex4;

import java.util.HashMap;

import java.util.Map;

/\*\*

 \* FinancialForecasting

 \*

 \* This class demonstrates forecasting the future value of an investment

 \* using recursion and an optimized memoization technique.

 \*/

public class FinancialForecasting {

    /\*\*

     \* Recursively computes future value after 'years' years.

     \* Time Complexity: O(years)

     \*

     \* @param currentValue Current value of the investment

     \* @param growthRate   Annual growth rate (as decimal, e.g., 0.07 for 7%)

     \* @param years        Number of years to forecast

     \* @return Future value after given years

     \*/

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

        if (years == 0) {

            return currentValue;

        }

        // Recursive call reducing years by 1 and applying growth

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

    }

    /\*\*

     \* Optimized recursive method using memoization to store intermediate results.

     \* Avoids redundant calculations.

     \*

     \* Time Complexity: O(years)

     \* Space Complexity: O(years)

     \*

     \* @param currentValue Current value of the investment

     \* @param growthRate   Annual growth rate

     \* @param years        Number of years to forecast

     \* @param memo         Map to store computed results for years

     \* @return Future value after given years

     \*/

    public static double forecastMemo(double currentValue, double growthRate, int years, Map<Integer, Double> memo) {

        if (years == 0) {

            return currentValue;

        }

        if (memo.containsKey(years)) {

            return memo.get(years);

        }

        double result = forecastMemo(currentValue \* (1 + growthRate), growthRate, years - 1, memo);

        memo.put(years, result);

        return result;

    }

    public static void main(String[] args) {

        double currentValue = 10000;

        double growthRate = 0.07;

        int years = 5;

        System.out.println("=== Financial Forecasting (Recursive) ===");

        double futureValue = forecastFutureValue(currentValue, growthRate, years);

        System.out.printf("Future Value (after %d years): ₹%.2f%n", years, futureValue);

        System.out.println("\n=== Optimized Version with Memoization ===");

        Map<Integer, Double> memo = new HashMap<>();

        double optimizedValue = forecastMemo(currentValue, growthRate, years, memo);

        System.out.printf("Memoized Future Value (after %d years): ₹%.2f%n", years, optimizedValue);

        System.out.println("\n=== Analysis ===");

        System.out.println("Memoization avoids redundant recursive calls, improving efficiency.");

    }

}

OUTPUT :

