**DESIGN PATTERNS AND PRINCIPLE**

**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.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **SingletonPatternExample**.
2. **Define a Singleton Class:**
   * Create a class named Logger that has a private static instance of itself.
   * Ensure the constructor of Logger is private.
   * Provide a public static method to get the instance of the Logger class.
3. **Implement the Singleton Pattern:**
   * Write code to ensure that the Logger class follows the Singleton design pattern.
4. **Test the Singleton Implementation:**
   * Create a test class to verify that only one instance of Logger is created and used across the application.

**PROGRAM**

**package** CTS;

**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);

}

**public** **static** **void** main(String[] args)

{

Logger logger1=Logger.*getInstance*();

logger1.log("Firstmessage.");

Logger logger2=Logger.*getInstance*();

logger2.log("Secondmessage.");

**if** (logger1==logger2)

{

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

}

**else**

{

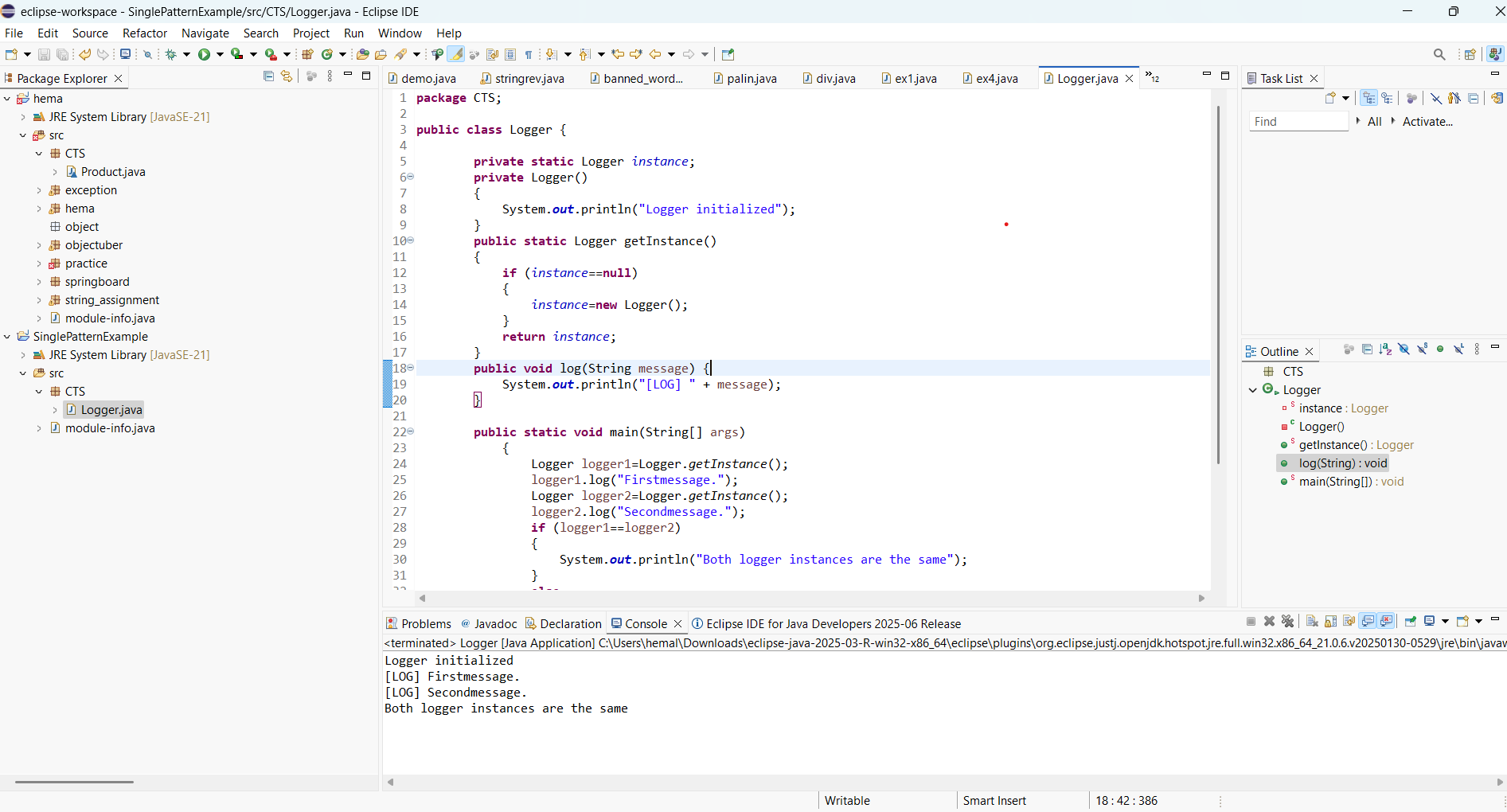
System.***out***.println("Logger instances are different");

}

}

}

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.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named **FactoryMethodPatternExample**.
2. **Define Document Classes:**
   * Create interfaces or abstract classes for different document types such as **WordDocument**, **PdfDocument**, and **ExcelDocument**.
3. **Create Concrete Document Classes:**
   * Implement concrete classes for each document type that implements or extends the above interfaces or abstract classes.
4. **Implement the Factory Method:**
   * Create an abstract class **DocumentFactory** with a method **createDocument()**.
   * Create concrete factory classes for each document type that extends DocumentFactory and implements the **createDocument()** method.
5. **Test the Factory Method Implementation:**
   * Create a test class to demonstrate the creation of different document types using the factory method.

**PROGRAM**

package factory;

public interface Document {

void open();

}

package factory;

public class WordDocument implements Document

{

public void open()

{

System.*out*.println("Opening a word document.");

}

}

package factory;

public class PdfDocument implements Document {

public void open() {

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

}

}

package factory;

public class ExcelDocument implements Document {

public void open() {

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

}

}

package factory;

public abstract class DocumentFactory

{

public abstract Document createDocument();

}

package factory;

public class WordDocumentFactory extends DocumentFactory {

public Document createDocument()

{

return new WordDocument();

}

}

package factory;

public class PdfDocumentFactory extends DocumentFactory {

public Document createDocument()

{

return new PdfDocument();

}

}

package factory;

public class ExcelDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new ExcelDocument();

}

}

package factory;

public class Main {

public static void main(String[] args)

{

DocumentFactory word = new WordDocumentFactory();

Document wordDoc = word.createDocument();

wordDoc.open();

DocumentFactory pdf = new PdfDocumentFactory();

Document pdfDoc = pdf.createDocument();

pdfDoc.open();

DocumentFactory excel = new ExcelDocumentFactory();

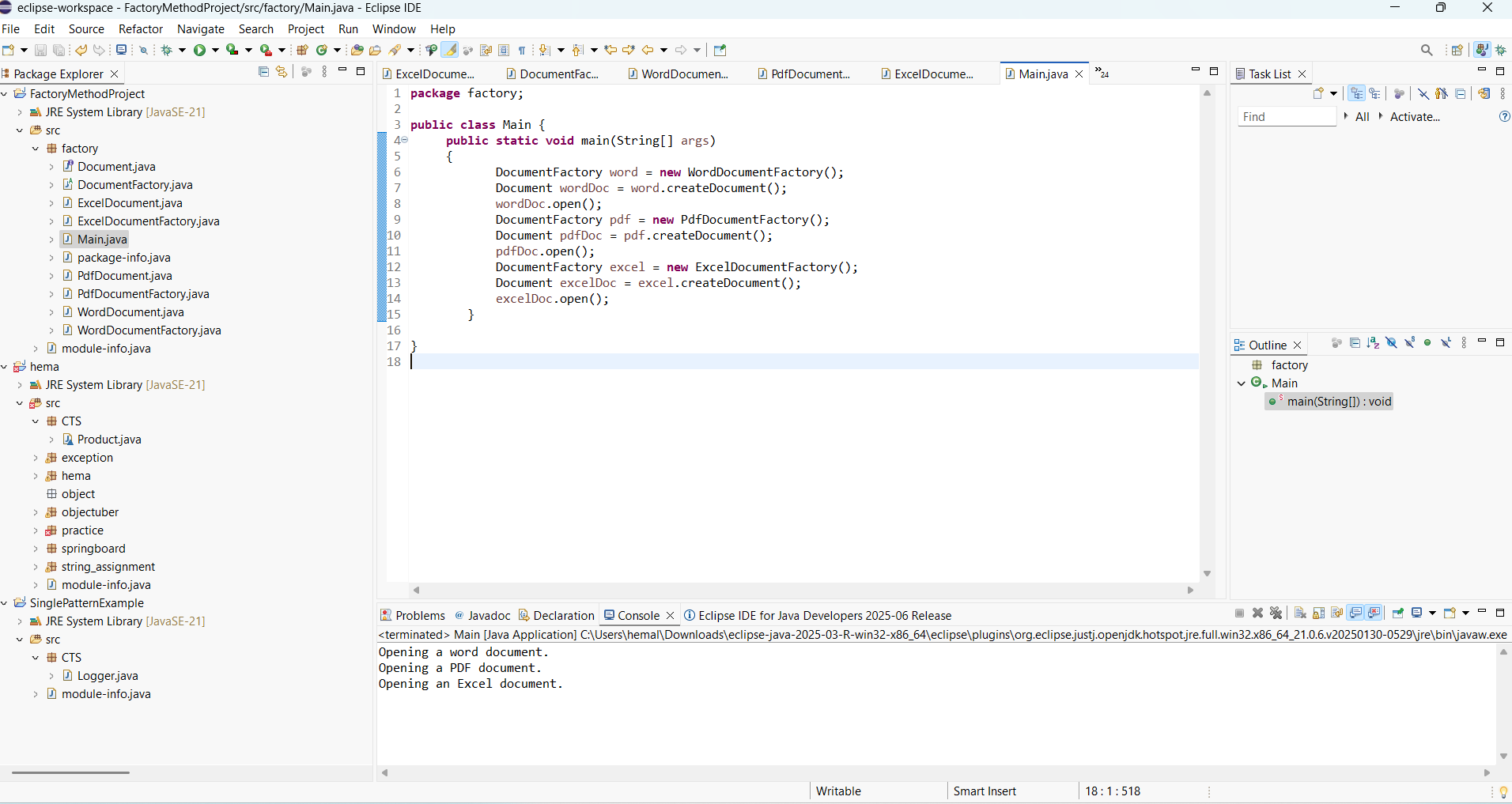
Document excelDoc = excel.createDocument();

excelDoc.open();

}

}

OUTPUT:



**DATA STRUCTURES AND 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.

**Steps:**

1. **Understand Asymptotic Notation:**
   * Explain Big O notation and how it helps in analyzing algorithms.
   * Describe the best, average, and worst-case scenarios for search operations.
2. **Setup:**
   * Create a class **Product** with attributes for searching, such as **productId, productName**, and **category**.
3. **Implementation:**
   * Implement linear search and binary search algorithms.
   * Store products in an array for linear search and a sorted array for binary search.
4. **Analysis:**
   * Compare the time complexity of linear and binary search algorithms.
   * Discuss which algorithm is more suitable for your platform and why.

**PROGRAM**

package CTS;

import java.util.Arrays;

import java.util.Comparator;

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 + ")";

}

public static Product linearsearch(Product[] products, String targetName) {

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

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

return products[i];

}

}

return null;

}

public static Product binarysearch(Product[] products, String targetName) {

int left=0, right=products.length - 1;

while (left<=right) {

int mid=(left+right)/2;

int result=products[mid].productName.compareToIgnoreCase(targetName);

if (result==0)

return products[mid];

else if (result<0)

left=mid + 1;

else

right=mid - 1;

}

return null;

}

public static void sortByName(Product[] products) {

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

}

public static void main(String[] args)

{

Product[] products = {

new Product(101, "Laptop", "Electronics"),

new Product(102, "Shoes", "Footwear"),

new Product(103, "Phone", "Electronics"),

new Product(104, "Table", "Furniture")

};

String searchName="phone";

Product linearresult=linearsearch(products, searchName);

if (linearresult!=null) {

System.out.println("Linear Product found: " + linearresult);

} else {

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

}

sortByName(products);

Product binaryresult=binarysearch(products , searchName);

if (binaryresult!=null) {

System.out.println("Binary search Product found: " + binaryresult);

} else {

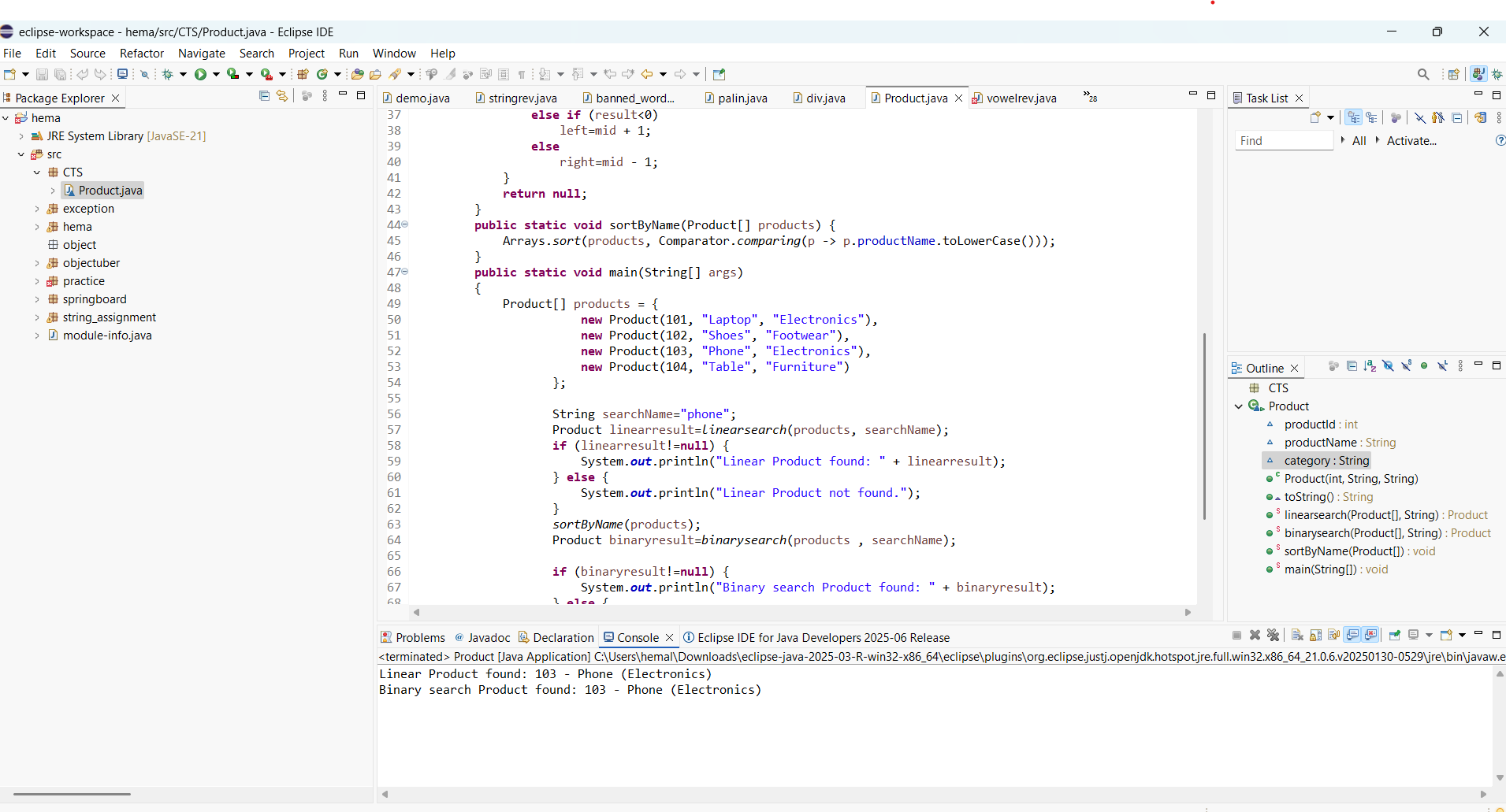
System.out.println("Binary search Product not found.");

}

}

}

**OUTPUT:**

****

**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.

**PROGRAM**

**package** CTS;

**public** **class** FinancialForecast {

**public** **static** **double** futureValue(**double** initialAmount, **double** growthRate, **int** periods)

{

**if** (periods == 0)

{

**return** initialAmount;

}

**return** *futureValue*(initialAmount \* (1 + growthRate), growthRate, periods - 1);

}

**public** **static** **void** main(String[] args) {

**double** initial = 1000.0;

**double** rate = 0.05;

**int** years = 10;

**double** result = *futureValue*(initial, rate, years);

System.***out***.printf("Predicted future value: ₹%.2f%n", result);

}

}

OUTPUT:

