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**.

code

class Logger {

private static Logger instance; private Logger() {

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

}

public static Logger getInstance() { if (instance == null) {

instance = new Logger();

}

return instance;

}

public void log(String message) { System.out.println("Log: " + message);

}

}

public class Main {

public static void main(String[] args) { Logger l1 = Logger.getInstance(); l1.log("First log message");

Logger l2 = Logger.getInstance(); l2.log("Second log message");

if (l1 == l2) {

System.out.println("Both are same instance");

} else {

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

}

}

}

if (logger1 == logger2) { System.out.println("Both are same instance");

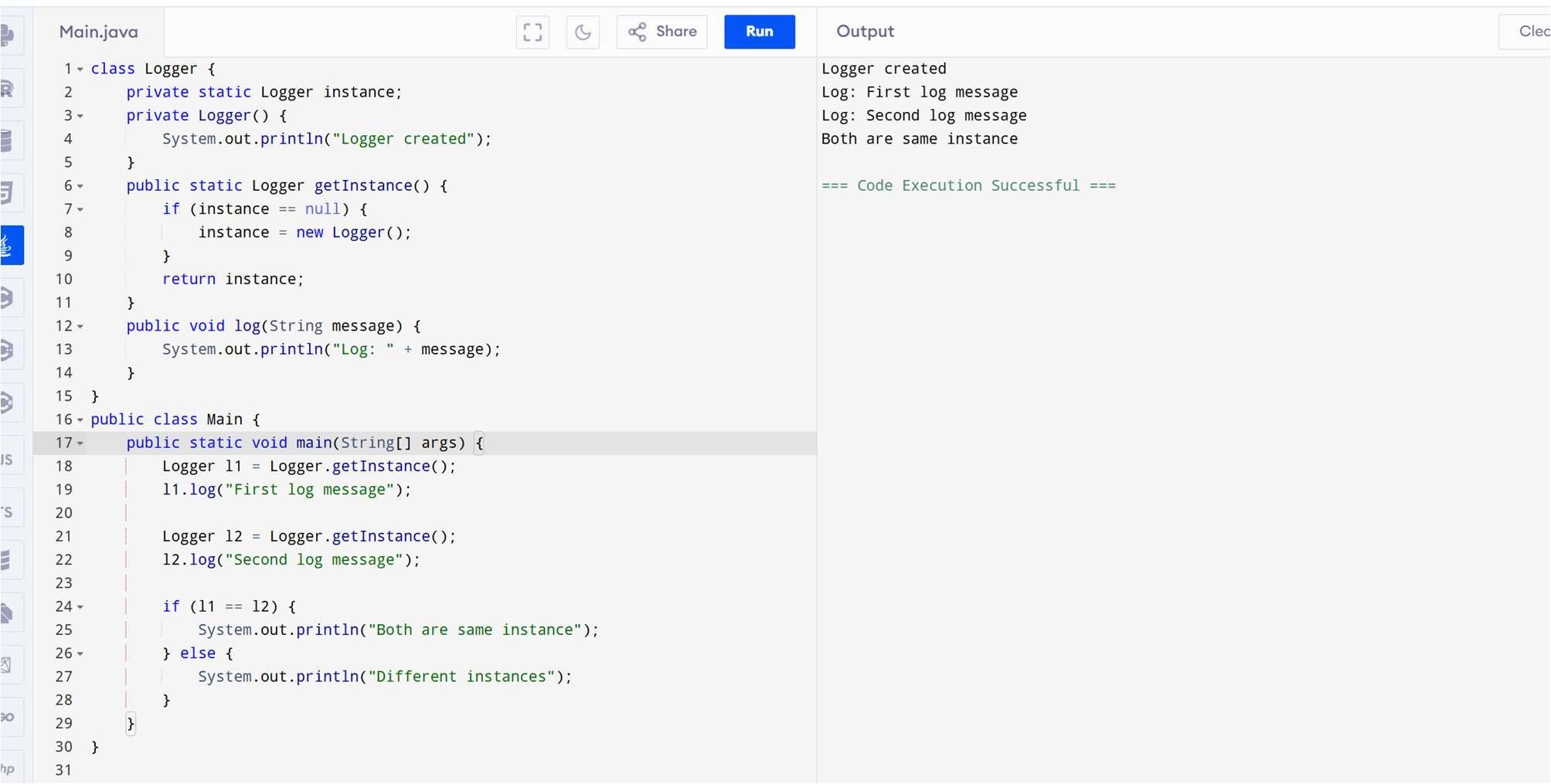
} else {

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

}

}

}



# 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**.

# Define Document Classes:

* + Create interfaces or abstract classes for different document types such as **WordDocument**, **PdfDocument**, and **ExcelDocument**.

# Create Concrete Document Classes:

* + Implement concrete classes for each document type that implements or extends the above interfaces or abstract classes.

# 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.

# Test the Factory Method Implementation:

* + Create a test class to demonstrate the creation of different document types using the factory method.

CODE

interface Document { void open();

}

class WordDocument implements Document { public void open() {

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

}

}

class PdfDocument implements Document { public void open() {

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

}

}

class ExcelDocument implements Document { public void open() {

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

}

}

class DocumentFactory {

public Document createDocument(String type) { if (type.equalsIgnoreCase("word"))

return new WordDocument();

else if (type.equalsIgnoreCase("pdf")) return new PdfDocument();

else if (type.equalsIgnoreCase("excel")) return new ExcelDocument();

else

return null;

}

}

public class Main {

public static void main(String[] args) { DocumentFactory factory = new DocumentFactory();

Document doc1 = factory.createDocument("word");

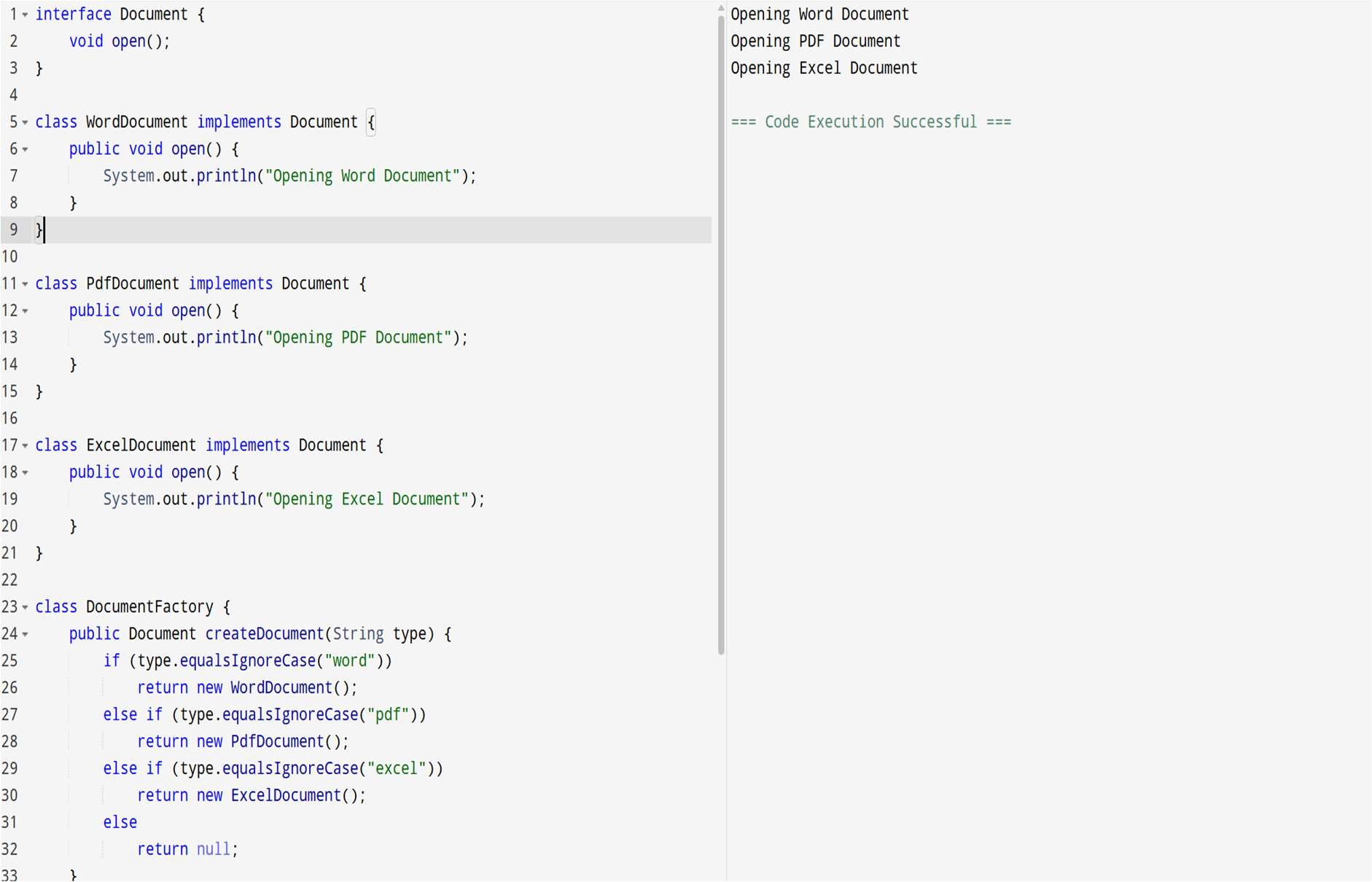
if (doc1 != null) doc1.open();

Document doc2 = factory.createDocument("pdf"); if (doc2 != null) doc2.open();

Document doc3 = factory.createDocument("excel"); if (doc3 != null) doc3.open();

}

}



# 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.

# Setup:

* + Create a class **Product** with attributes for searching, such as **productId, productName**, and **category**.

# Implementation:

* + Implement linear search and binary search algorithms.
  + Store products in an array for linear search and a sorted array for binary search.

# Analysis:

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

CODE

import java.util.Arrays; class Product {

int productId;

String productName; String category;

Product(int productId, String productName, String category) { this.productId = productId;

this.productName = productName; this.category = category;

}

}

public class SearchExample {

public static void main(String[] args) { Product[] products = {

new Product(102, "Laptop", "Electronics"), new Product(101, "Shirt", "Fashion"),

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

};

for (int i = 0; i < products.length; i++) { if (products[i].productId == 103) {

System.out.println("Linear Search: Found " + products[i].productName); break;

}

}

Arrays.sort(products, (a, b) -> a.productId - b.productId); int low = 0, high = products.length - 1;

while (low <= high) {

int mid = (low + high) / 2;

if (products[mid].productId == 103) {

System.out.println("Binary Search: Found " + products[mid].productName); break;

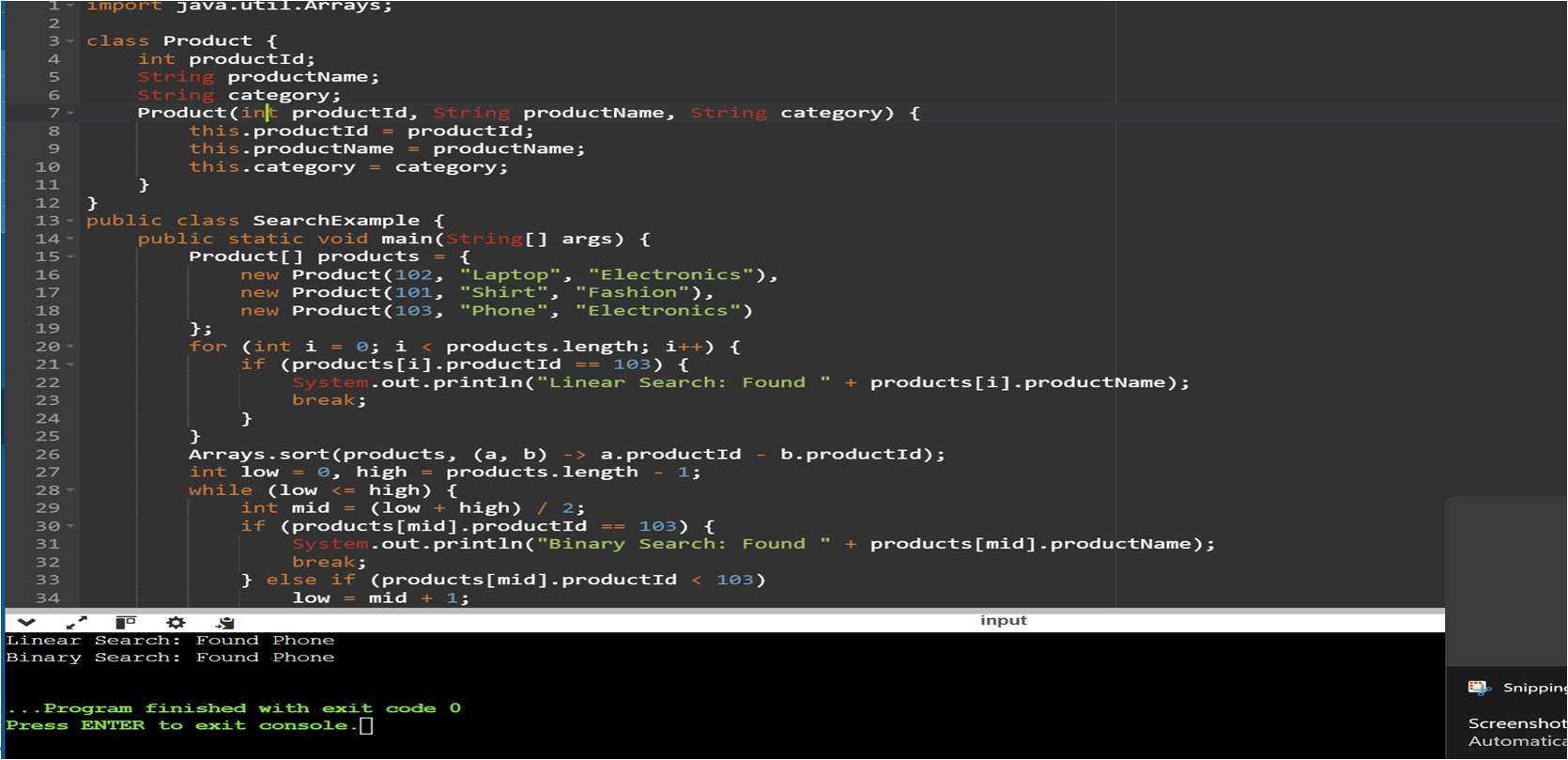
} else if (products[mid].productId < 103) low = mid + 1;

else

high = mid - 1;

}

}

}

# 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.

# Setup:

* + Create a method to calculate the future value using a recursive approach.

# Implementation:

* + Implement a recursive algorithm to predict future values based on past growth rates.

# Analysis:

* + Discuss the time complexity of your recursive algorithm.
  + Explain how to optimize the recursive solution to avoid excessive computation.

CODE

public class FinancialForecast {

static double futureValue(double presentValue, double growthRate, int years) { if (years == 0)

return presentValue; else

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

}

public static void main(String[] args) { double presentValue = 1000.0; double growthRate = 0.05;

int years = 3;

double result = futureValue(presentValue, growthRate, years); System.out.println("Future Value after " + years + " years: Rs " + result);

}

}

