Design principles & Patterns

**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:**

* **Create a New Java Project:**
  + Create a new Java project named **SingletonPatternExample**.
* **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.
* **Implement the Singleton Pattern:**
  + Write code to ensure that the Logger class follows the Singleton design pattern.
* **Test the Singleton Implementation:**
  + Create a test class to verify that only one instance of Logger is created and used across the application.

**CODE :**

Logger.java :

package com.singleton;

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);

}

}

LoggerTest.java :

package com.singleton;

public class LoggerTest {

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 refer to the same instance.");

} else {

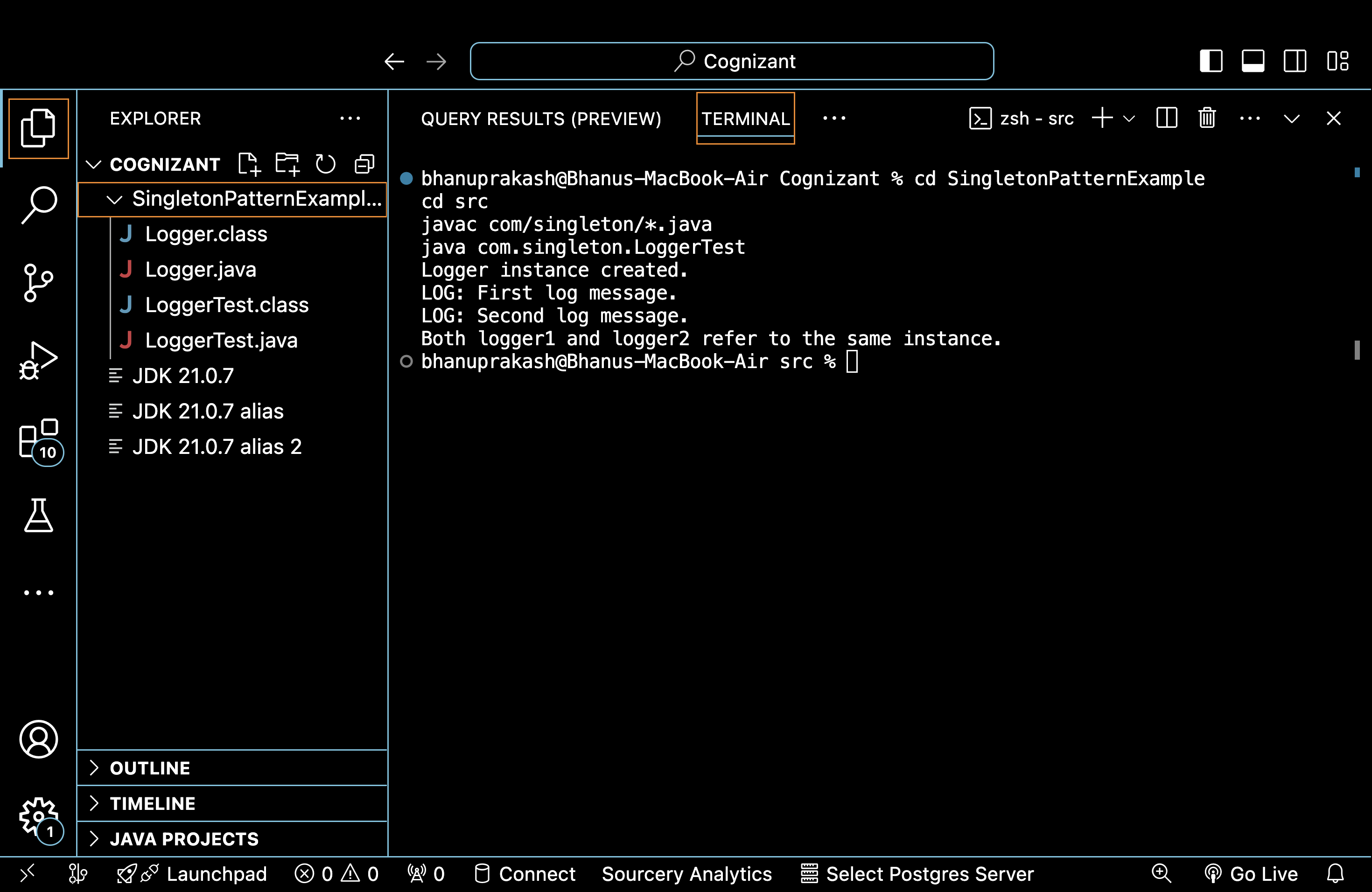
System.out.println("logger1 and logger2 are different instances.");

}

}

}

**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.

**Steps:**

* **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 :**

**Step 1: Create a new Java project named**

**FactoryMethodPatternExample**

**Step 2: Define Document Classes (Interfaces)**

// Document.java

package com.factory;

public interface Document {

void open();

}

**Step 3: Create Concrete Document Classes**

// WordDocument.java

package com.factory;

public class WordDocument implements Document {

public void open() {

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

}

}

// PdfDocument.java

package com.factory;

public class PdfDocument implements Document {

public void open() {

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

}

}

// ExcelDocument.java

package com.factory;

public class ExcelDocument implements Document {

public void open() {

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

}

}

**Step 4: Implement the Factory Method**

// DocumentFactory.java (abstract)

package com.factory;

public abstract class DocumentFactory {

public abstract Document createDocument();

}

// WordDocumentFactory.java

package com.factory;

public class WordDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new WordDocument();

}

}

// PdfDocumentFactory.java

package com.factory;

public class PdfDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new PdfDocument();

}

}

// ExcelDocumentFactory.java

package com.factory;

public class ExcelDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new ExcelDocument();

}

}

**Step 5: Test the Factory Method Implementation**

// FactoryTest.java

package com.factory;

public class FactoryTest {

public static void main(String[] var0) {

WordDocumentFactory var1 = new WordDocumentFactory();

Document var2 = var1.createDocument();

var2.open() PdfDocumentFactory var3 = new PdfDocumentFactory();

Document var4 = var3.createDocument();

var4.open();

ExcelDocumentFactory var5 = new ExcelDocumentFactory();

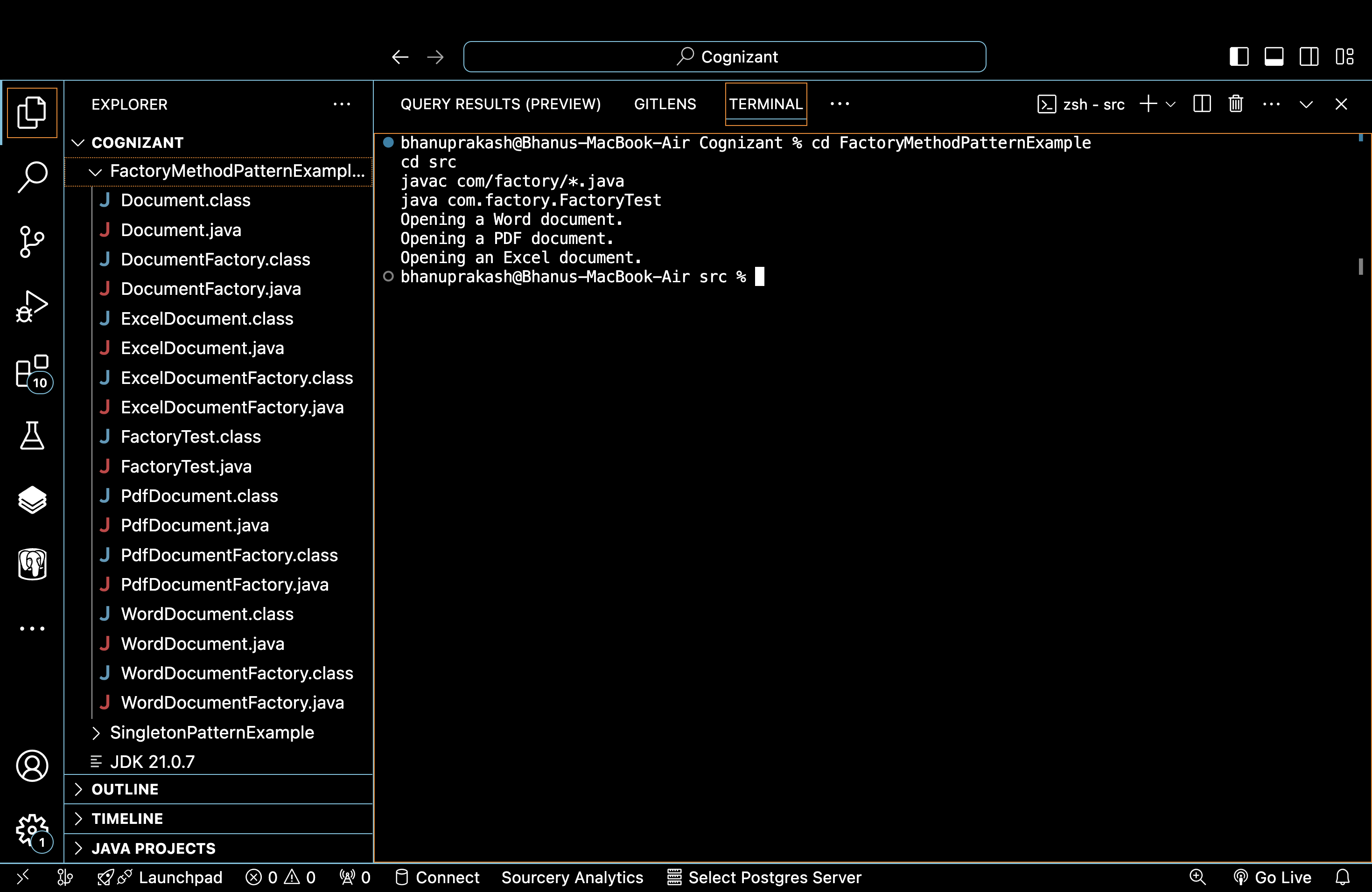
Document var6 = var5.createDocument();

var6.open();

}

}

**OUTPUT :**



Data structures and Algorithms

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

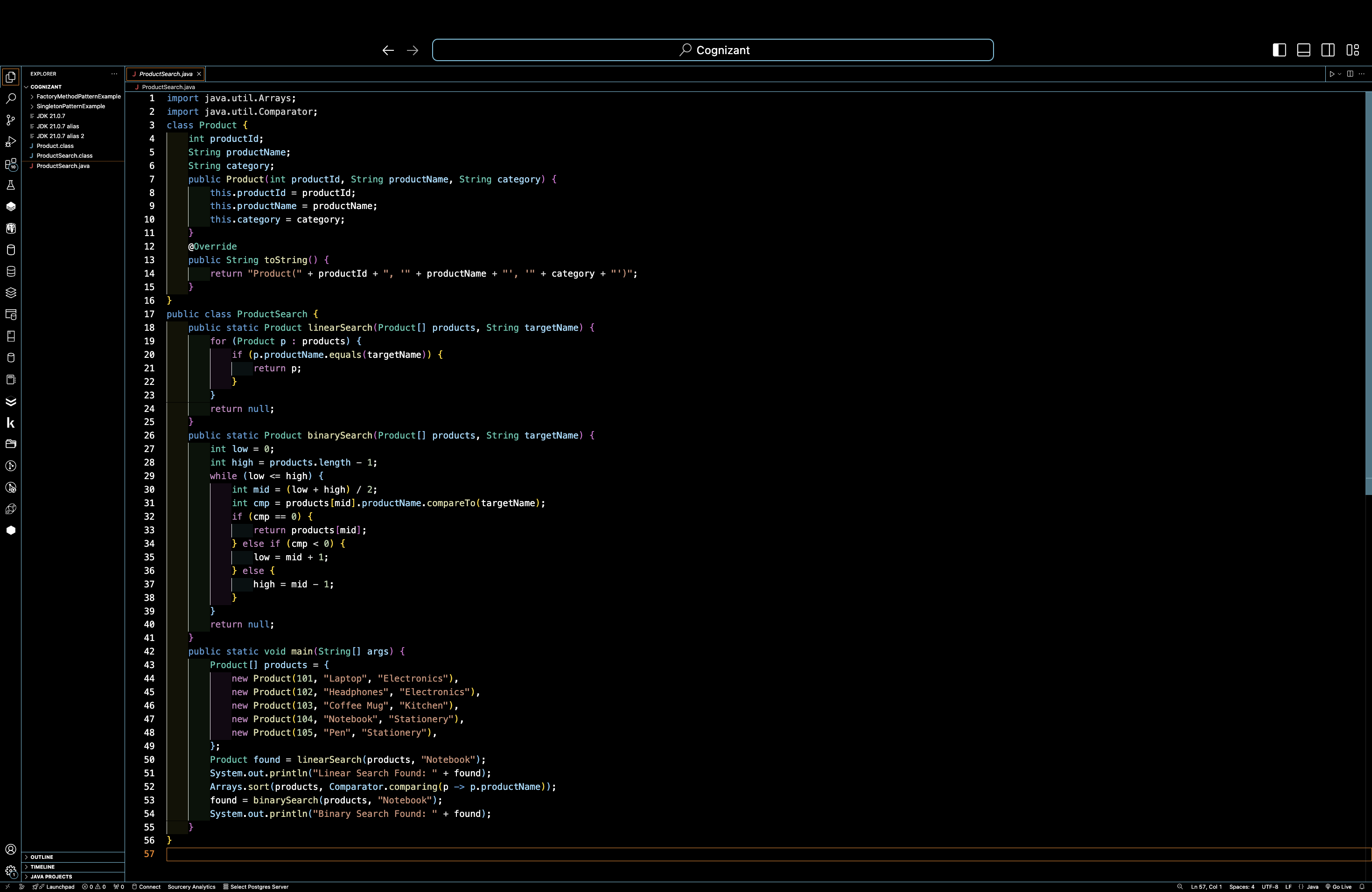
**Scenario:**

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

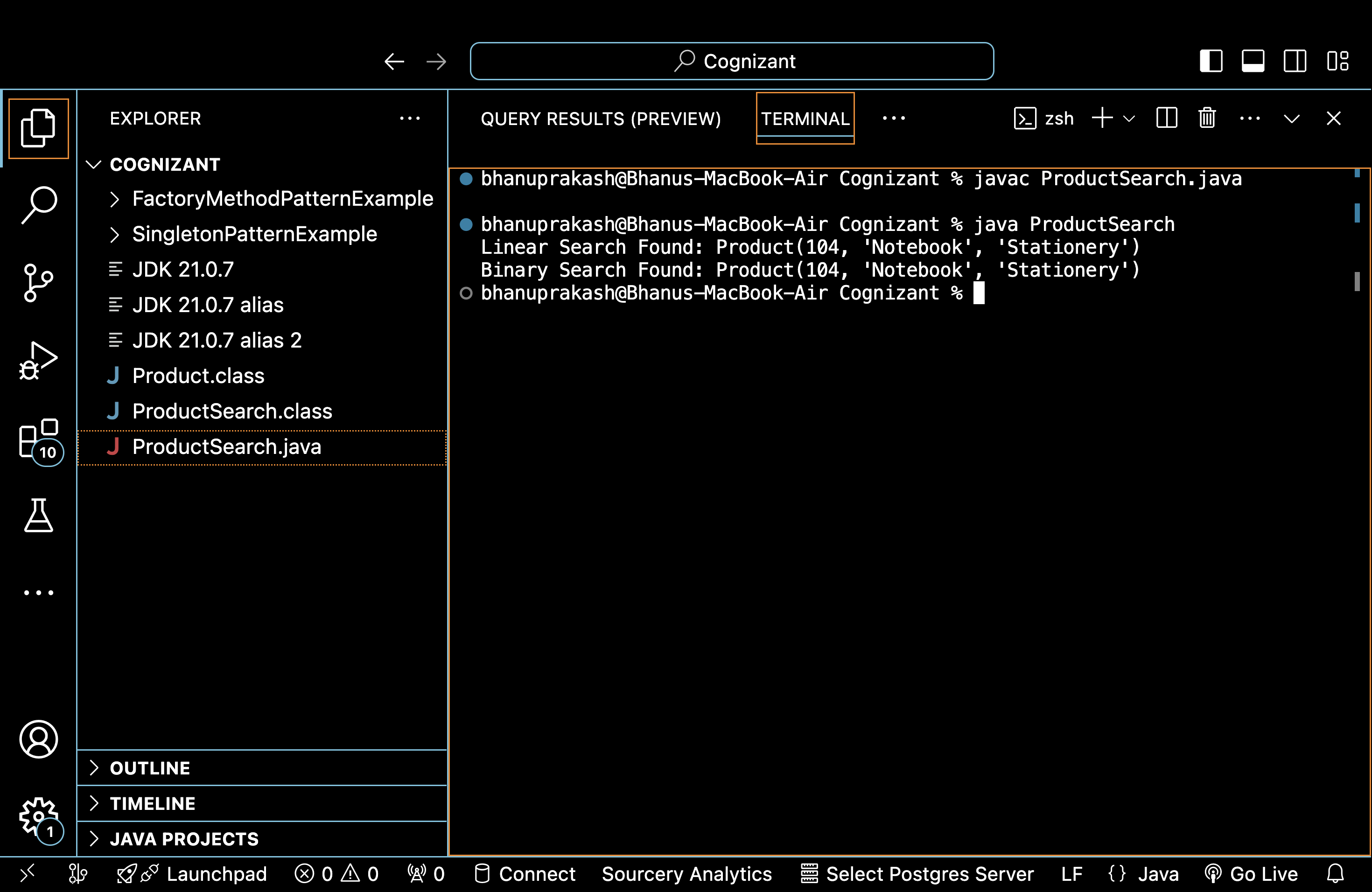
**Steps:**

* **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 :**



**OUTPUT :**



**Exercise 7: Financial Forecasting**

**Scenario:**

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

**Steps:**

* **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 AND OUTPUT :**

