**Week – 1 Design patterns and principles , Data structures and algorithms**

Design Patterns and Principles

Exercise 1: Implementing the Singleton Pattern :

Logger.java:

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);

}

}

Main.java:

public class Main {

public static void main(String[] args) {

Logger logger1 = Logger.getInstance();

Logger logger2 = Logger.getInstance();

logger1.log("First log message");

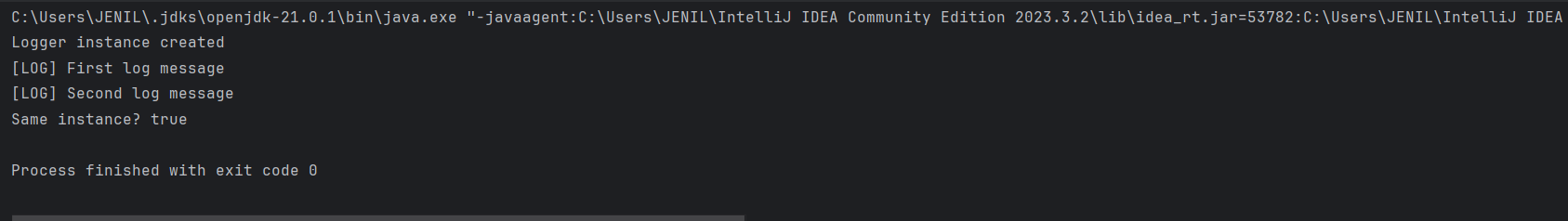
logger2.log("Second log message");

System.out.println("Same instance? " + (logger1 == logger2));

}

}

**Output :**

****

------------------------------------------------------------------------------------

Design Patterns and Principles

Exercise 2: Implementing the Factory Method Pattern

Main.java :

public class Main {

public static void main(String[] args) {

DocumentFactory wordFactory = new WordDocumentFactory();

DocumentFactory pdfFactory = new PdfDocumentFactory();

DocumentFactory excelFactory = new ExcelDocumentFactory();

System.out.println("Processing Word Document:");

wordFactory.processDocument();

System.out.println("\nProcessing PDF Document:");

pdfFactory.processDocument();

System.out.println("\nProcessing Excel Document:");

excelFactory.processDocument();

}

}

Document.java :

public interface Document {

void open();

void save();

void close();

}

WordDocument.java :

public class WordDocument implements Document {

public void open() {

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

}

public void save() {

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

}

public void close() {

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

}

}

PdfDocument.java :

public class PdfDocument implements Document {

public void open() {

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

}

public void save() {

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

}

public void close() {

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

}

}

ExcelDocument.java :

public class ExcelDocument implements Document {

public void open() {

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

}

public void save() {

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

}

public void close() {

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

}

}

DocumentFactory.java :

public abstract class DocumentFactory {

public abstract Document createDocument();

public void processDocument() {

Document doc = createDocument();

doc.open();

doc.save();

doc.close();

}

}

WordDocumentFactory.java

public class WordDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new WordDocument();

}

}

PdfDocumentFactory.java

public class PdfDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new PdfDocument();

}

}

ExcelDocumentFactory.java :

public class ExcelDocumentFactory extends DocumentFactory {

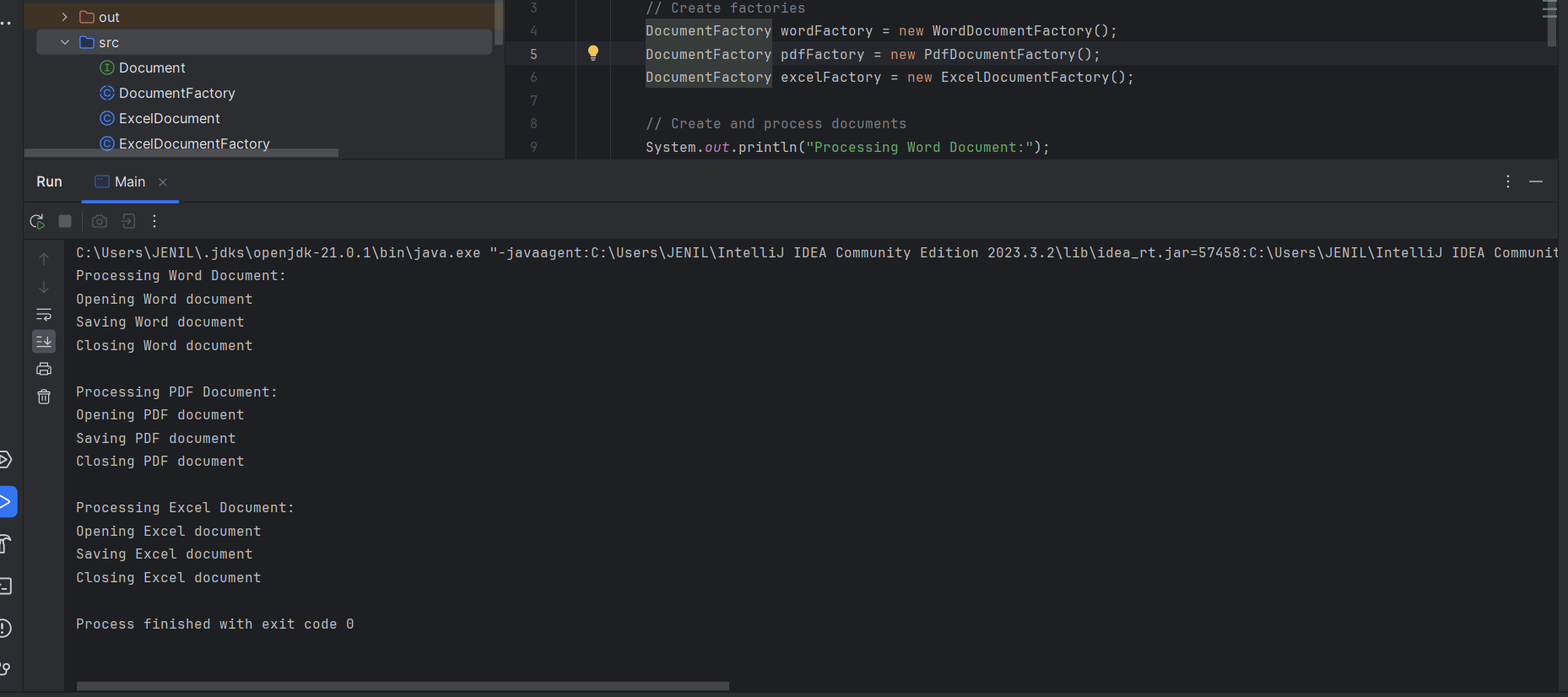
public Document createDocument() {

return new ExcelDocument();

}

}

**Output:**



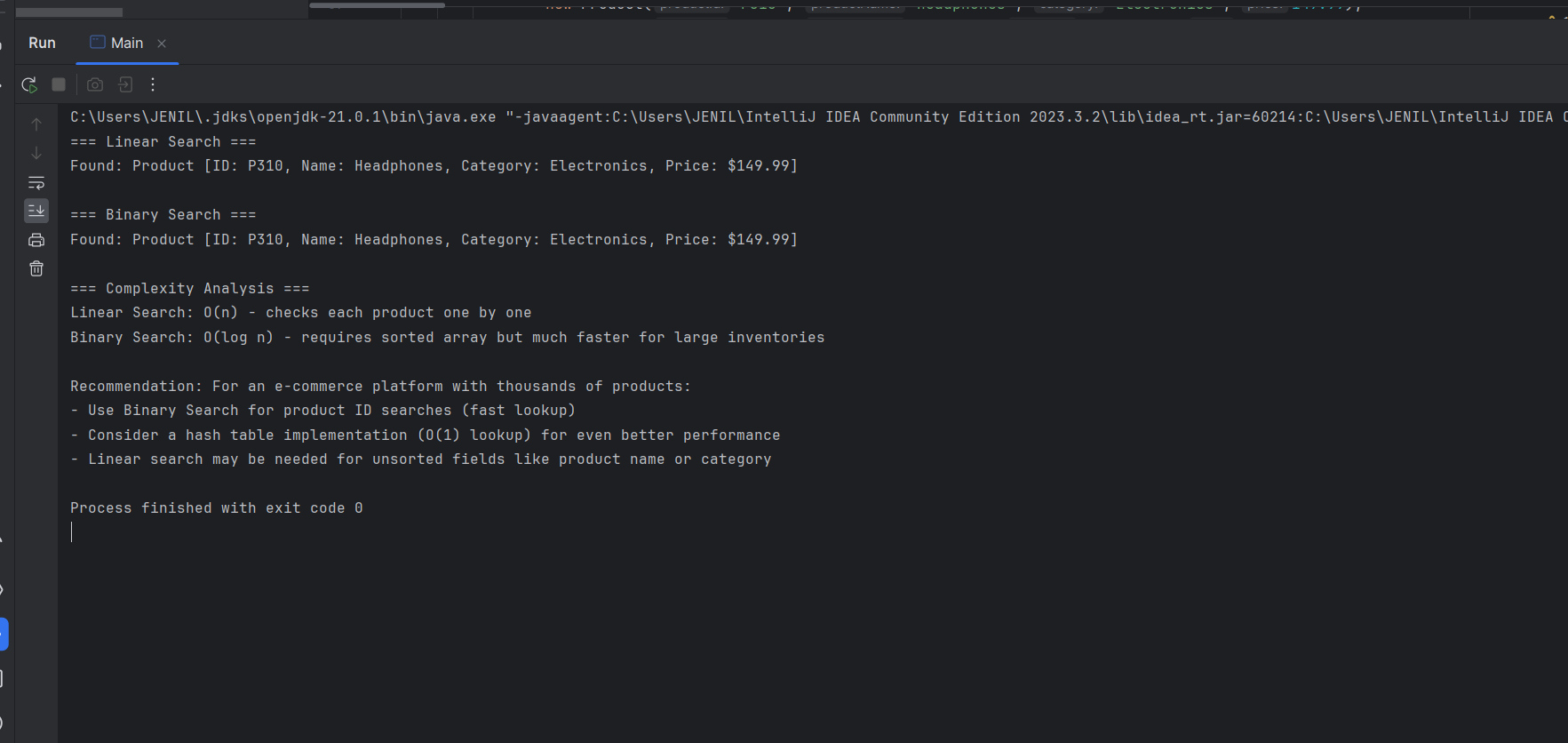
------------------------------------------------------------------------------------

Algorithms\_Data Structures

Exercise 2: E-commerce Platform Search Function

import java.util.Arrays;  
import java.util.Comparator;  
  
class Product {  
 private String productId;  
 private String productName;  
 private String category;  
 private double price;  
  
 public Product(String productId, String productName, String category, double price) {  
 this.productId = productId;  
 this.productName = productName;  
 this.category = category;  
 this.price = price;  
 }  
  
   
 public String getProductId() { return productId; }  
 public String getProductName() { return productName; }  
 public String getCategory() { return category; }  
 public double getPrice() { return price; }  
  
 public String toString() {  
 return "Product [ID: " + productId + ", Name: " + productName +  
 ", Category: " + category + ", Price: $" + price + "]";  
 }  
}  
  
public class Main {  
  
   
 public static Product linearSearch(Product[] products, String targetId) {  
 for (Product product : products) {  
 if (product.getProductId().equals(targetId)) {  
 return product;  
 }  
 }  
 return null;  
 }  
  
   
 public static Product binarySearch(Product[] sortedProducts, String targetId) {  
 int left = 0;  
 int right = sortedProducts.length - 1;  
  
 while (left <= right) {  
 int mid = left + (right - left) / 2;  
 int comparison = sortedProducts[mid].getProductId().compareTo(targetId);  
  
 if (comparison == 0) {  
 return sortedProducts[mid];  
 } else if (comparison < 0) {  
 left = mid + 1;  
 } else {  
 right = mid - 1;  
 }  
 }  
 return null;  
 }  
  
 public static void main(String[] args) {  
   
 Product[] products = {  
 new Product("P100", "Smartphone", "Electronics", 599.99),  
 new Product("P205", "Laptop", "Electronics", 999.99),  
 new Product("P310", "Headphones", "Electronics", 149.99),  
 new Product("P420", "T-Shirt", "Clothing", 19.99),  
 new Product("P530", "Jeans", "Clothing", 49.99)  
 };  
  
   
 Product[] sortedProducts = Arrays.*copyOf*(products, products.length);  
 Arrays.*sort*(sortedProducts, Comparator.*comparing*(Product::getProductId));  
  
   
 String targetId = "P310";  
  
 System.*out*.println("=== Linear Search ===");  
 Product resultLinear = *linearSearch*(products, targetId);  
 System.*out*.println(resultLinear != null ? "Found: " + resultLinear : "Product not found");  
  
 System.*out*.println("\n=== Binary Search ===");  
 Product resultBinary = *binarySearch*(sortedProducts, targetId);  
 System.*out*.println(resultBinary != null ? "Found: " + resultBinary : "Product not found");  
  
   
 System.*out*.println("\n=== Complexity Analysis ===");  
 System.*out*.println("Linear Search: O(n) - checks each product one by one");  
 System.*out*.println("Binary Search: O(log n) - requires sorted array but much faster for large inventories");  
 System.*out*.println("\nRecommendation: For an e-commerce platform with thousands of products:");  
 System.*out*.println("- Use Binary Search for product ID searches (fast lookup)");  
 System.*out*.println("- Consider a hash table implementation (O(1) lookup) for even better performance");  
 System.*out*.println("- Linear search may be needed for unsorted fields like product name or category");  
 }  
}

**output:**



------------------------------------------------------------------------------------

Algorithms\_Data Structures

Exercise 7: Financial Forecasting

import java.util.\*;  
import java.util.Arrays;  
  
public class Main{  
  
  
 public static double calculateFutureValue(double presentValue, double growthRate, int periods) {  
  
 if (periods == 0) {  
 return presentValue;  
 }  
  
  
 return (1 + growthRate) \* *calculateFutureValue*(presentValue, growthRate, periods - 1);  
 }  
  
  
 private static double[] *memo*;  
  
 public static double calculateFutureValueOptimized(double presentValue, double growthRate, int periods) {  
 *memo* = new double[periods + 1];  
 Arrays.*fill*(*memo*, -1);  
 return *calculateFutureValueMemo*(presentValue, growthRate, periods);  
 }  
  
 private static double calculateFutureValueMemo(double presentValue, double growthRate, int periods) {  
 if (periods == 0) {  
 return presentValue;  
 }  
  
 if (*memo*[periods] != -1) {  
 return *memo*[periods];  
 }  
  
 *memo*[periods] = (1 + growthRate) \* *calculateFutureValueMemo*(presentValue, growthRate, periods - 1);  
 return *memo*[periods];  
 }  
  
 public static void main(String[] args) {  
 double pv = 1000.0;  
 double rate = 0.05;   
 int years = 10;   
  
  
 double fv = *calculateFutureValue*(pv, rate, years);  
 System.*out*.printf("Basic recursive: Future value after %d years: $%.2f%n", years, fv);  
  
  
 double fvOpt = *calculateFutureValueOptimized*(pv, rate, years);  
 System.*out*.printf("Optimized recursive: Future value after %d years: $%.2f%n", years, fvOpt);  
  
  
 System.*out*.println("\n=== Complexity Analysis ===");  
 System.*out*.println("Basic recursive: O(n) time, O(n) space (call stack)");  
 System.*out*.println("Optimized recursive: O(n) time, O(n) space (but with memoization)");  
 System.*out*.println("\nFor financial forecasting with many periods:");  
 System.*out*.println("- Consider iterative approach for better space efficiency");  
 System.*out*.println("- For complex recursive models, memoization is essential");  
 System.*out*.println("- Matrix exponentiation can optimize certain recursive financial models");  
 }  
}

**Output:**

