**Algorithms\_Data Structures**

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

**CODE:**

**package** com.ecommerce.search;

**public** **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;

}

@Override

**public** String toString() {

**return** productName + " (ID: " + productId + ", Category: " + category + ")";

}

}

**package** com.ecommerce.search;

**import** java.util.Arrays;

**import** java.util.Comparator;

**public** **class** ProductSearch {

// Linear Search

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

**for** (Product product : products) {

**if** (product.productName.equalsIgnoreCase(targetName)) {

**return** product;

}

}

**return** **null**;

}

// Binary Search (requires sorted array)

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

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

**while** (left <= right) {

**int** mid = (left + right) / 2;

String midName = products[mid].productName.toLowerCase();

**if** (midName.equals(targetName.toLowerCase())) {

**return** products[mid];

} **else** **if** (midName.compareTo(targetName.toLowerCase()) < 0) {

left = mid + 1;

} **else** {

right = mid - 1;

}

}

**return** **null**;

}

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

Product[] products = {

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

**new** Product(102, "Smartphone", "Electronics"),

**new** Product(103, "Shirt", "Clothing"),

**new** Product(104, "Notebook", "Stationery"),

**new** Product(105, "Headphones", "Electronics")

};

System.***out***.println(" Linear Search for 'Notebook':");

Product result = *linearSearch*(products, "Notebook");

System.***out***.println("Result: " + result);

// Sort products by name before binary search

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

System.***out***.println("\n Binary Search for 'Notebook':");

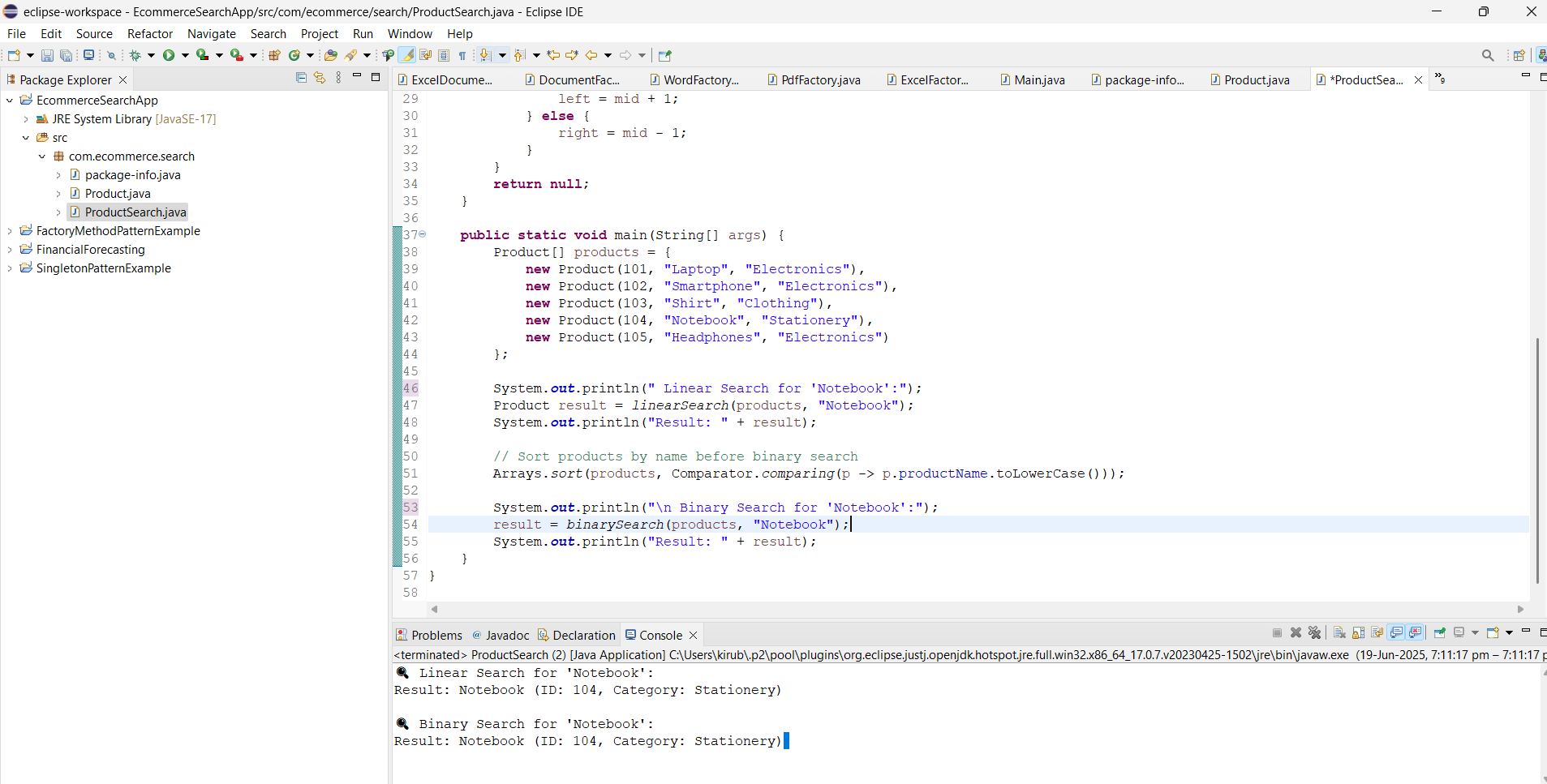
result = *binarySearch*(products, "Notebook");

System.***out***.println("Result: " + result);

}

}

**OUTPUT:**



**Exercise 7: Financial Forecasting**

**CODE:**

package com.finance;

public class RecursiveForecast {

public static double calculateFutureValue(double principal, double growthRate, int years) {

if (years == 0) {

return principal;

}

return calculateFutureValue(principal \* (1 + growthRate), growthRate, years - 1);

}

public static void main(String[] args) {

double initialAmount = 10000; // ₹10,000

double annualGrowthRate = 0.08; // 8%

int futureYears = 5;

double futureValue = calculateFutureValue(initialAmount, annualGrowthRate, futureYears);

System.out.printf(" Predicted value after %d years = ₹%.2f\n", futureYears, futureValue);

}

}

package com.finance;

public class RecursiveForecast {

public static double calculateFutureValue(double principal, double growthRate, int years) {

if (years == 0) {

return principal;

}

return calculateFutureValue(principal \* (1 + growthRate), growthRate, years - 1);

}

public static void main(String[] args) {

double initialAmount = 10000; // ₹10,000

double annualGrowthRate = 0.08; // 8%

int futureYears = 5;

double futureValue = calculateFutureValue(initialAmount, annualGrowthRate, futureYears);

System.out.printf(" Predicted value after %d years = ₹%.2f\n", futureYears, futureValue);

}

}

package com.finance;

public class RecursiveForecast {

public static double calculateFutureValue(double principal, double growthRate, int years) {

if (years == 0) {

return principal;

}

return calculateFutureValue(principal \* (1 + growthRate), growthRate, years - 1);

}

public static void main(String[] args) {

double initialAmount = 10000; // ₹10,000

double annualGrowthRate = 0.08; // 8%

int futureYears = 5;

double futureValue = calculateFutureValue(initialAmount, annualGrowthRate, futureYears);

System.out.printf(" Predicted value after %d years = ₹%.2f\n", futureYears, futureValue);

}

}

package com.finance;

public class RecursiveForecast {

public static double calculateFutureValue(double principal, double growthRate, int years) {

if (years == 0) {

return principal;

}

return calculateFutureValue(principal \* (1 + growthRate), growthRate, years - 1);

}

public static void main(String[] args) {

double initialAmount = 10000; // ₹10,000

double annualGrowthRate = 0.08; // 8%

int futureYears = 5;

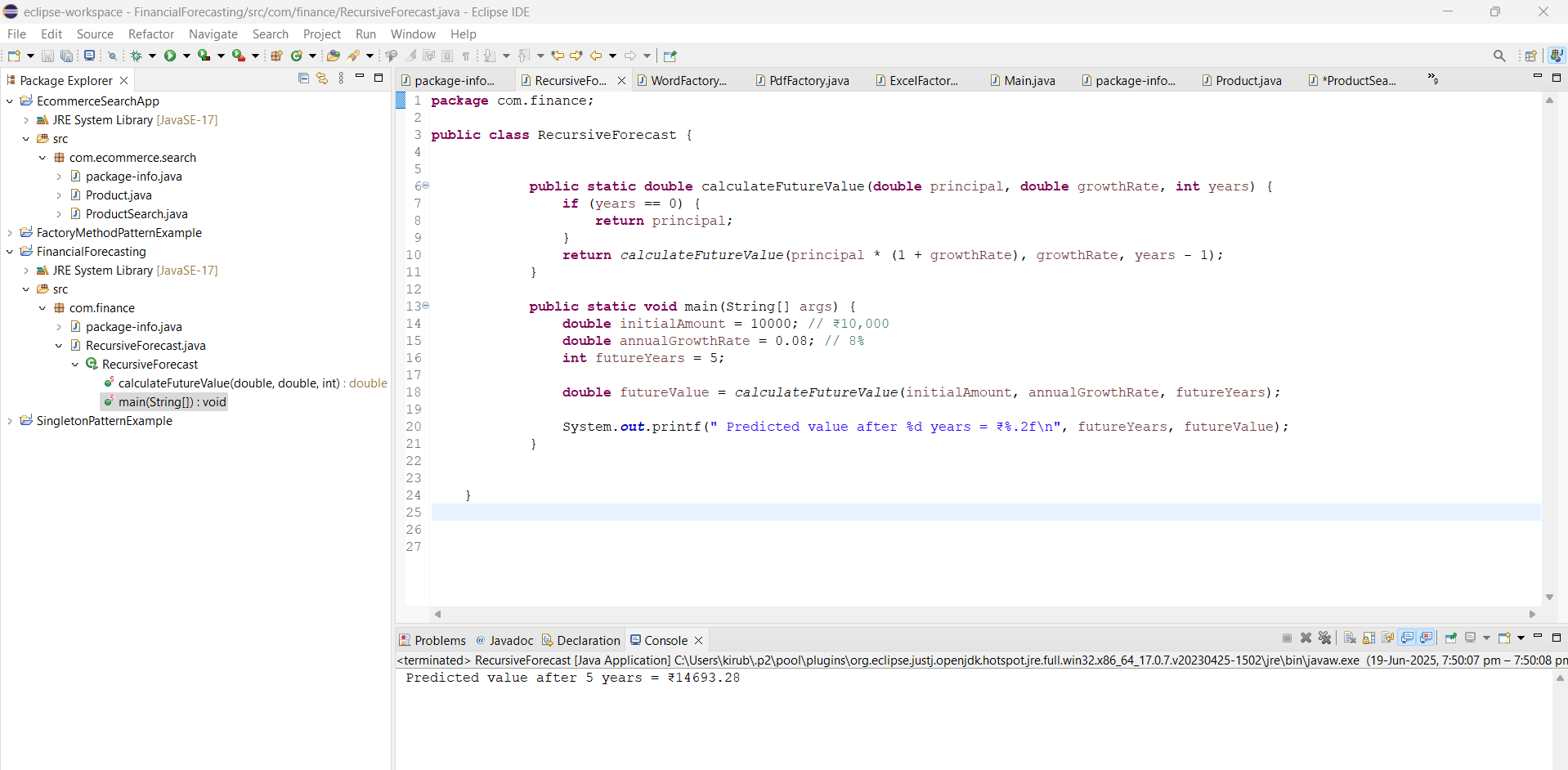
double futureValue = calculateFutureValue(initialAmount, annualGrowthRate, futureYears);

System.out.printf(" Predicted value after %d years = ₹%.2f\n", futureYears, futureValue);

}

}

**OUTPUT:**



**Design Patterns and Principles**

**Exercise 1: Implementing the Singleton Pattern**

CODE:

**package** com.singleton.example;

**public** **class** Logger {

// 1. Private static instance of the same class

**private** **static** Logger *instance*;

// 2. Private constructor to prevent instantiation

**private** Logger() {

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

}

// 3. Public static method to provide global access

**public** **static** Logger getInstance() {

**if** (*instance* == **null**) {

*instance* = **new** Logger();

}

**return** *instance*;

}

**public** **void** log(String message) {

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

}

}

**package** com.singleton.example;

**public** **class** Main {

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

// Get Logger instance 1

Logger logger1 = Logger.*getInstance*();

logger1.log("This is the first log message.");

// Get Logger instance 2

Logger logger2 = Logger.*getInstance*();

logger2.log("This is the second log message.");

// Compare both instances

System.***out***.println("Logger instance hash: " + logger1.hashCode());

System.***out***.println("Logger instance hash: " + logger2.hashCode());

**if** (logger1 == logger2) {

System.***out***.println(" Same instance used everywhere!");

} **else** {

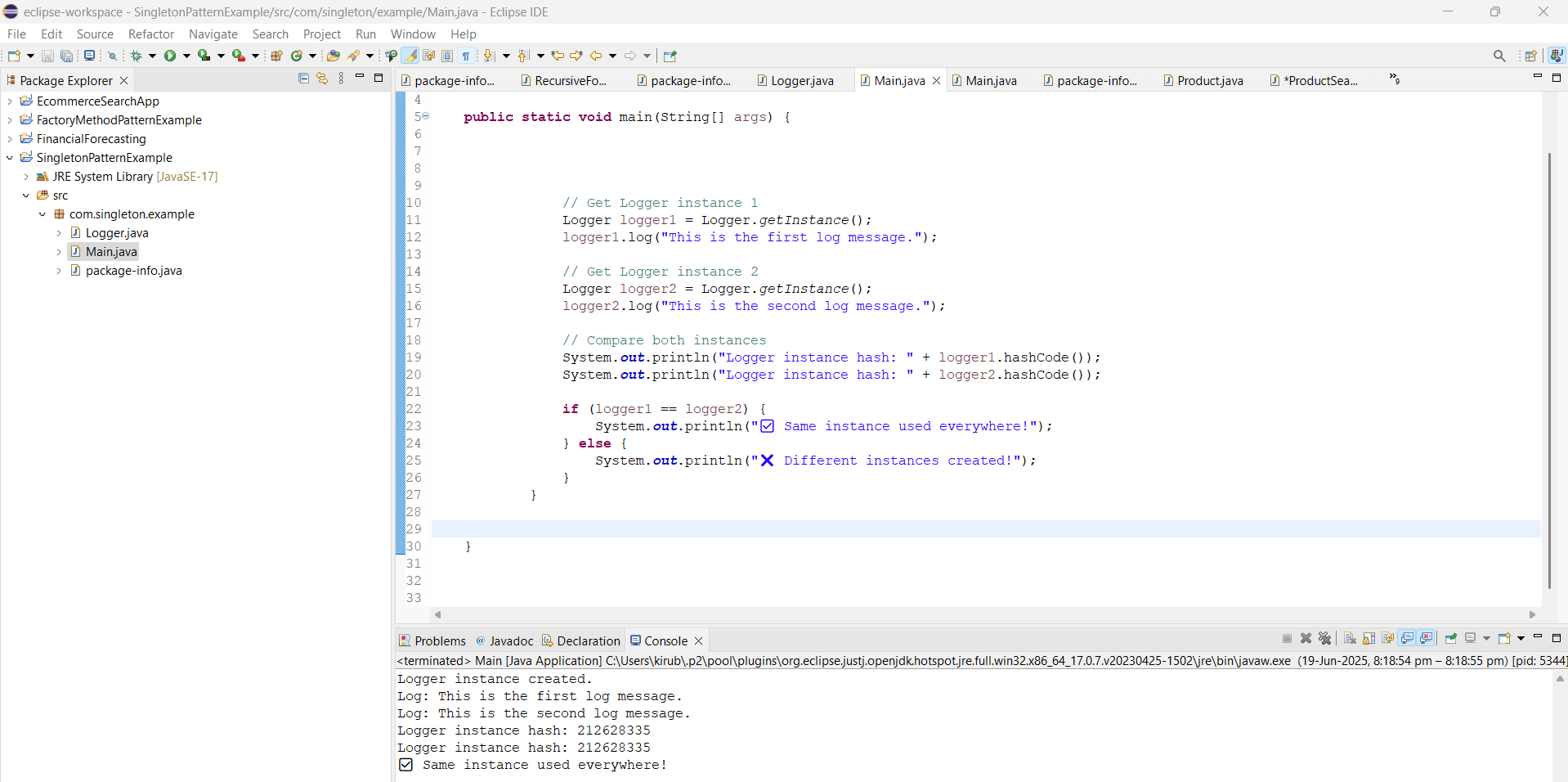
System.***out***.println(" Different instances created!");

}

}

}

OUTPUT:



**Exercise 2: Implementing the Factory Method Pattern**

CODE:

**package** com.factory.document;

**public** **interface** Document {

**void** open();

}

**package** com.factory.document;

**public** **abstract** **class** DocumentFactory {

**public** **abstract** Document createDocument(); // factory method

}

**package** com.factory.document;

**public** **class** ExcelDocument **implements** Document {

@Override

**public** **void** open() {

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

}

}

**package** com.factory.document;

**public** **class** ExcelFactory **extends** DocumentFactory {

@Override

**public** Document createDocument() {

**return** **new** ExcelDocument();

}

}

**package** com.factory.document;

**public** **class** Main {

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

// **TODO** Auto-generated method stub

DocumentFactory wordFactory = **new** WordFactory();

Document wordDoc = wordFactory.createDocument();

wordDoc.open();

// Create PDF document using factory

DocumentFactory pdfFactory = **new** PdfFactory();

Document pdfDoc = pdfFactory.createDocument();

pdfDoc.open();

// Create Excel document using factory

DocumentFactory excelFactory = **new** ExcelFactory();

Document excelDoc = excelFactory.createDocument();

excelDoc.open();

}

}

**package** com.factory.document;

**public** **class** PdfDocument **implements** Document {

@Override

**public** **void** open() {

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

}

}

**package** com.factory.document;

**public** **class** PdfFactory **extends** DocumentFactory {

@Override

**public** Document createDocument() {

**return** **new** PdfDocument();

}

}

**package** com.factory.document;

**public** **class** WordDocument **implements** Document {

@Override

**public** **void** open() {

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

}

}

**package** com.factory.document;

**public** **class** WordFactory **extends** DocumentFactory {

@Override

**public** Document createDocument() {

**return** **new** WordDocument();

}

}

OUTPUT:

