**Exercise 1: Implementing the Singleton Pattern**

**LAGGER CLASS**

package 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);

}

}

LAGGER TEST CLASS

package singleton;

public class LoggerTest {

public static void main(String[] args) {

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

Logger logger1 = Logger.*getInstance*();

logger1.log("First message");

Logger logger2 = Logger.*getInstance*();

logger2.log("Second message");

if (logger1 == logger2) {

System.***out***.println("Only one instance exists. Singleton confirmed.");

} else {

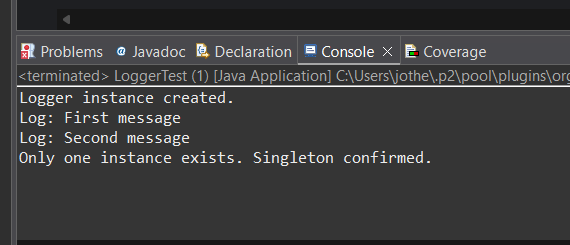
System.***out***.println("Different instances exist. Singleton failed.");

}

}

}

OUTPUT



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

**Document**

Java

package com.document;

// Interface for all document types

public interface Document {

void open();

void save();

void close();

}

**Word Document**

Java

package com.document;

// Concrete implementation for Word Document

public class WordDocument implements Document {

@Override

public void open() {

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

}

@Override

public void save() {

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

}

@Override

public void close() {

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

}

}

**PdfDocument**

Java

package com.document;

// Concrete implementation for PDF Document

public class PdfDocument implements Document {

@Override

public void open() {

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

}

@Override

public void save() {

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

}

@Override

public void close() {

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

}

}

**ExcelDocument**

Java

package com.document;

// Concrete implementation for Excel Document

public class ExcelDocument implements Document {

@Override

public void open() {

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

}

@Override

public void save() {

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

}

@Override

public void close() {

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

}

}

**DocumentFactory**

Java

package com.document;

// Abstract factory class with the factory method

public abstract class DocumentFactory {

public abstract Document createDocument();

}

**WordDocumentFactory**

Java

package com.document;

// Concrete factory for creating Word Documents

public class WordDocumentFactory extends DocumentFactory {

@Override

public Document createDocument() {

return new WordDocument();

}

}

**PdfDocumentFactory**

Java

package com.document;

// Concrete factory for creating PDF Documents

public class PdfDocumentFactory extends DocumentFactory {

@Override

public Document createDocument() {

return new PdfDocument();

}

}

**ExcelDocumentFactory**

Java

package com.document;

// Concrete factory for creating Excel Documents

public class ExcelDocumentFactory extends DocumentFactory {

@Override

public Document createDocument() {

return new ExcelDocument();

}

}

**DocumentCreationTest**

Java

package com.document;

// Test class to demonstrate the Factory Method Pattern

public class DocumentCreationTest {

public static void main(String[] args) {

// Create a Word Document using its factory

DocumentFactory wordFactory = new WordDocumentFactory();

Document wordDoc = wordFactory.createDocument();

System.out.println("--- Word Document Operations ---");

wordDoc.open();

wordDoc.save();

wordDoc.close();

System.out.println();

// Create a PDF Document using its factory

DocumentFactory pdfFactory = new PdfDocumentFactory();

Document pdfDoc = pdfFactory.createDocument();

System.out.println("--- PDF Document Operations ---");

pdfDoc.open();

pdfDoc.save();

pdfDoc.close();

System.out.println();

// Create an Excel Document using its factory

DocumentFactory excelFactory = new ExcelDocumentFactory();

Document excelDoc = excelFactory.createDocument();

System.out.println("--- Excel Document Operations ---");

excelDoc.open();

excelDoc.save();

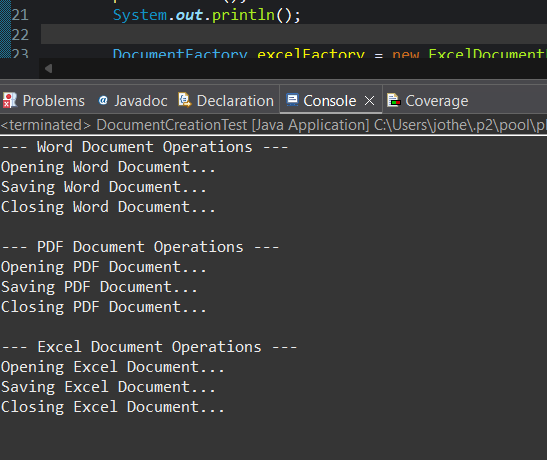
excelDoc.close();

System.out.println();

}

}

**Output**

****

**E-commerce Platform Search Function**

**PRODUCT**

package com.ecommerce.search;

public class Product {

private String productId;

private String productName;

private String category;

public Product(String productId, String productName, String category) {

this.productId = productId;

this.productName = productName;

this.category = category;

}

// Getters

public String getProductId() {

return productId;

}

public String getProductName() {

return productName;

}

public String getCategory() {

return category;

}

*@Override*

public String toString() {

return "Product [ID=" + productId + ", Name=" + productName + ", Category=" + category + "]"

}

}

**SEARCH ALGORITHMS**

package com.ecommerce.search;

public class SearchAlgorithms {

public Product linearSearch(Product[] products, String targetId) {

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

if (products[i].getProductId().equals(targetId)) {

return products[i];

}

}

return null;

}

public Product binarySearch(Product[] products, String targetId) {

int left = 0;

int right = products.length - 1;

while (left <= right) {

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

int comparisonResult = targetId.compareTo(products[mid].getProductId());

if (comparisonResult == 0) {

return products[mid];

} else if (comparisonResult < 0) {

right = mid - 1;

} else {

left = mid + 1;

}

}

return null;

}

}

**SEARCH TEST**

package com.ecommerce.search;

import java.util.Arrays;

import java.util.Comparator;

public class SearchTest {

public static void main(String[] args) {

Product[] products = {

new Product("P003", "Laptop", "Electronics"),

new Product("P001", "Mouse", "Electronics"),

new Product("P005", "Keyboard", "Electronics"),

new Product("P002", "Desk Lamp", "Home & Office"),

new Product("P004", "Notebook", "Stationery")

};

SearchAlgorithms searcher = new SearchAlgorithms();

System.***out***.println("--- Linear Search ---");

String targetId1 = "P002";

Product foundProduct1 = searcher.linearSearch(products, targetId1);

if (foundProduct1 != null) {

System.***out***.println("Found: " + foundProduct1);

} else {

System.***out***.println("Product with ID " + targetId1 + " not found.");

}

String targetId2 = "P006";

Product foundProduct2 = searcher.linearSearch(products, targetId2);

if (foundProduct2 != null) {

System.***out***.println("Found: " + foundProduct2);

} else {

System.***out***.println("Product with ID " + targetId2 + " not found.");

}

System.***out***.println();

System.***out***.println("--- Binary Search ---");

System.***out***.println("Original Products: " + Arrays.*toString*(products));

Arrays.*sort*(products, Comparator.*comparing*(Product::getProductId));

System.***out***.println("Sorted Products (by ID): " + Arrays.*toString*(products));

System.***out***.println();

String targetId3 = "P004";

Product foundProduct3 = searcher.binarySearch(products, targetId3);

if (foundProduct3 != null) {

System.***out***.println("Found: " + foundProduct3);

} else {

System.***out***.println("Product with ID " + targetId3 + " not found.");

}

String targetId4 = "P001";

Product foundProduct4 = searcher.binarySearch(products, targetId4);

if (foundProduct4 != null) {

System.***out***.println("Found: " + foundProduct4);

} else {

System.***out***.println("Product with ID " + targetId4 + " not found.");

}

String targetId5 = "P007";

Product foundProduct5 = searcher.binarySearch(products, targetId5);

if (foundProduct5 != null) {

System.***out***.println("Found: " + foundProduct5);

} else {

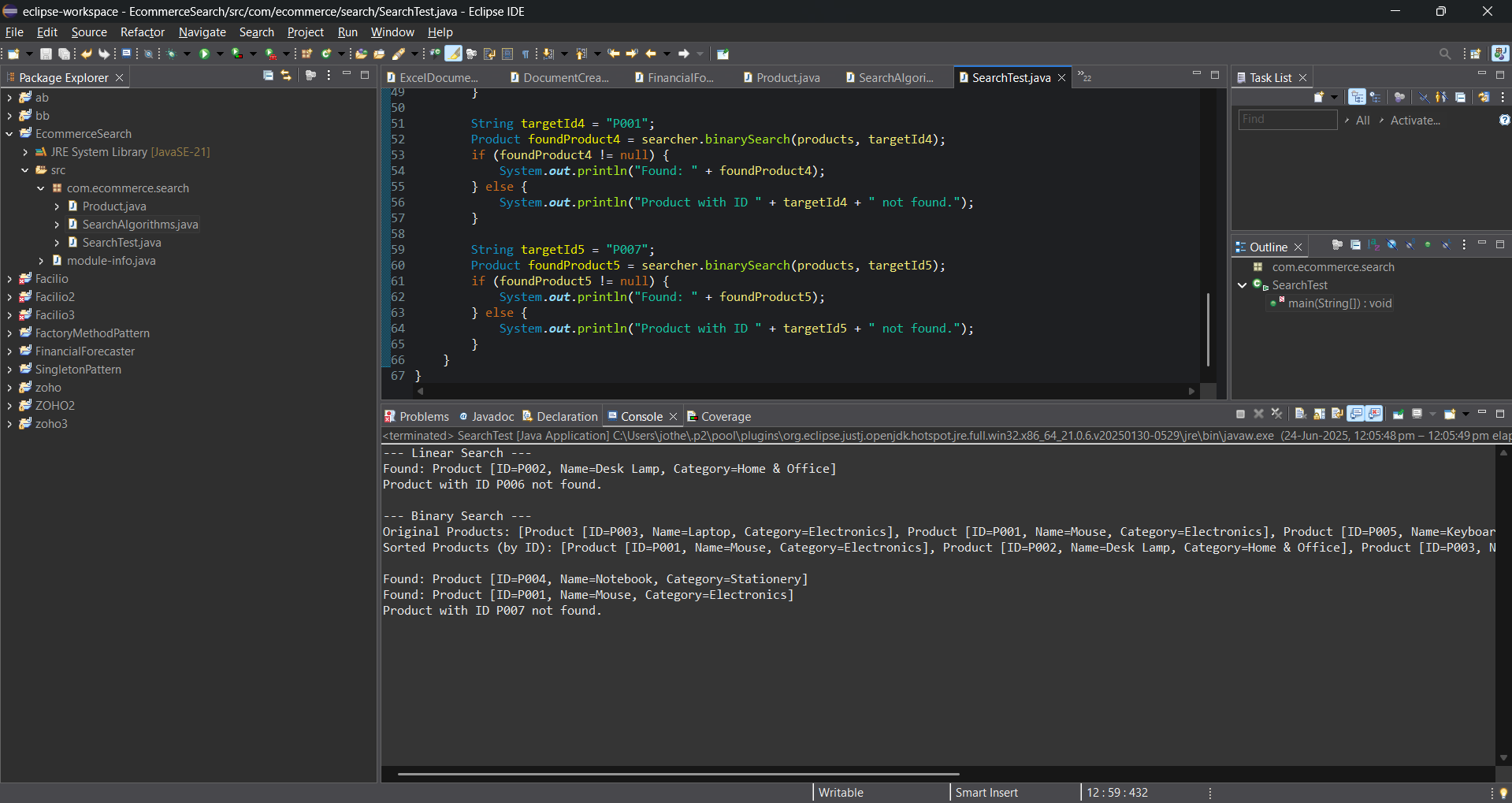
System.***out***.println("Product with ID " + targetId5 + " not found.");

}

}

}

**OUTPUT**



**Financial Forecasting**

package com.forecasting;

import java.util.HashMap;

import java.util.Map;

public class FinancialForecaster {

public double calculateFutureValueRecursive(double presentValue, double growthRate, int periods) {

if (periods < 0) {

throw new IllegalArgumentException("Number of periods cannot be negative.");

}

if (periods == 0) {

return presentValue;

} else {

double valueAfterPreviousPeriod = calculateFutureValueRecursive(presentValue, growthRate, periods - 1);

return valueAfterPreviousPeriod \* (1 + growthRate);

}

}

private Map<Integer, Double> memo = new HashMap<>();

private double initialPresentValueForMemo;

private double fixedGrowthRateForMemo;

public void resetMemoization(double pv, double rate) {

this.memo.clear();

this.initialPresentValueForMemo = pv;

this.fixedGrowthRateForMemo = rate;

}

public double calculateFutureValueRecursiveMemoized(double presentValue, double growthRate, int periods) {

if (periods < 0) {

throw new IllegalArgumentException("Number of periods cannot be negative.");

}

if (memo.containsKey(periods)) {

return memo.get(periods);

}

if (periods == 0) {

memo.put(periods, presentValue);

return presentValue;

} else {

double valueAfterPreviousPeriod = calculateFutureValueRecursiveMemoized(

this.initialPresentValueForMemo,

this.fixedGrowthRateForMemo,

periods - 1

);

double currentFutureValue = valueAfterPreviousPeriod \* (1 + fixedGrowthRateForMemo);

memo.put(periods, currentFutureValue);

return currentFutureValue;

}

}

public static void main(String[] args) {

FinancialForecaster forecaster = new FinancialForecaster();

double initialAmount = 1000.0;

double annualGrowthRate = 0.05;

int forecastPeriods = 10;

try {

double futureValue = forecaster.calculateFutureValueRecursive(initialAmount, annualGrowthRate, forecastPeriods);

System.*out*.printf("Initial Amount: %.2f%n", initialAmount);

System.*out*.printf("Annual Growth Rate: %.2f%%%n", annualGrowthRate \* 100);

System.*out*.printf("Forecast Periods: %d years%n", forecastPeriods);

System.*out*.printf("Predicted Future Value (Recursive): %.2f%n", futureValue);

double futureValueIterative = initialAmount \* Math.*pow*((1 + annualGrowthRate), forecastPeriods);

System.*out*.printf("Predicted Future Value (Iterative): %.2f%n", futureValueIterative);

System.*out*.println("\nTesting with 0 periods:");

System.*out*.printf("Future Value for 0 periods: %.2f%n", forecaster.calculateFutureValueRecursive(500.0, 0.10, 0));

System.*out*.println("\nTesting with negative periods:");

try {

forecaster.calculateFutureValueRecursive(100.0, 0.02, -5);

} catch (IllegalArgumentException e) {

System.*out*.println("Caught expected exception: " + e.getMessage());

}

} catch (IllegalArgumentException e) {

System.*err*.println("Error: " + e.getMessage());

}

System.*out*.println("\n--- Testing Memoized Recursive Version ---");

forecaster.resetMemoization(initialAmount, annualGrowthRate);

try {

double futureValueMemoized = forecaster.calculateFutureValueRecursiveMemoized(initialAmount, annualGrowthRate, forecastPeriods);

System.*out*.printf("Predicted Future Value (Recursive Memoized): %.2f%n", futureValueMemoized);

double futureValueMemoized2 = forecaster.calculateFutureValueRecursiveMemoized(initialAmount, annualGrowthRate, forecastPeriods);

System.*out*.printf("Predicted Future Value (Recursive Memoized, 2nd call): %.2f%n", futureValueMemoized2);

} catch (IllegalArgumentException e) {

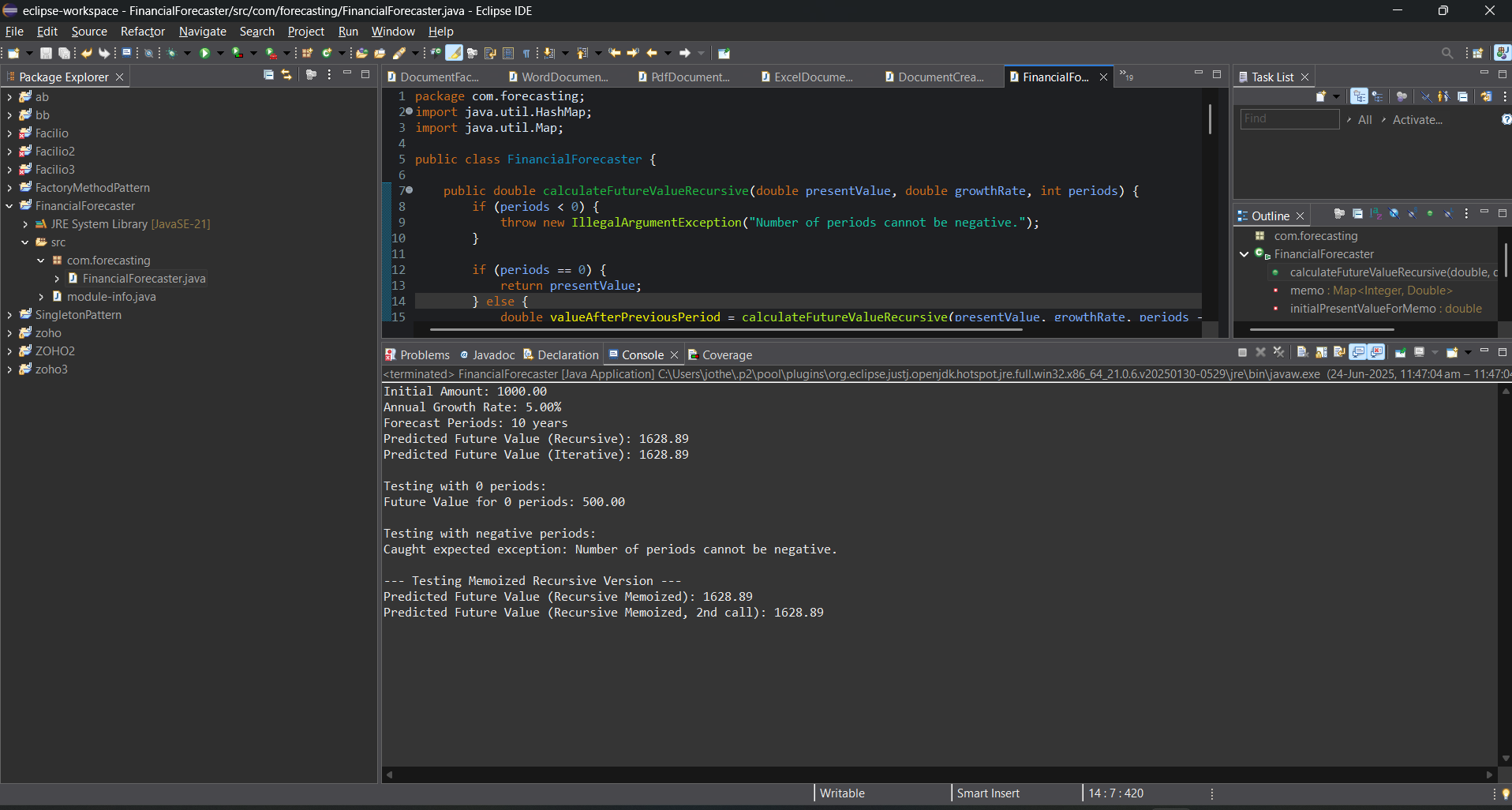
System.*err*.println("Error: " + e.getMessage());

}

}

}

**OUTPUT**

****