**WEEK - 1**

**Exercise 1: Implementing the Singleton Pattern**

**public class SingletonPatternExample {**

**static 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);**

**}**

**}**

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

**Logger logger1 = Logger.getInstance();**

**Logger logger2 = Logger.getInstance();**

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

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

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

**System.out.println("Both logger instances are the same (Singleton works)");**

**} else {**

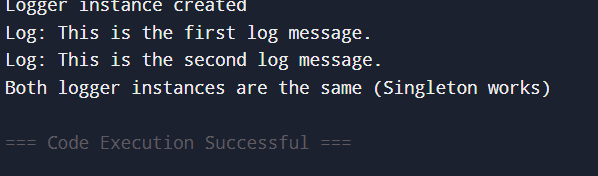
**System.out.println("Different instances (Singleton failed)");**

**}**

**}**

**}**

**OUTPUT :**



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

**public class FactoryMethodPatternExample {**

**interface Document {**

**void open();**

**}**

**static class WordDocument implements Document {**

**public void open() {**

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

**}**

**}**

**static class PdfDocument implements Document {**

**public void open() {**

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

**}**

**}**

**static class ExcelDocument implements Document {**

**public void open() {**

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

**}**

**}**

**abstract static class DocumentFactory {**

**public abstract Document createDocument();**

**}**

**static class WordFactory extends DocumentFactory {**

**public Document createDocument() {**

**return new WordDocument();**

**}**

**}**

**static class PdfFactory extends DocumentFactory {**

**public Document createDocument() {**

**return new PdfDocument();**

**}**

**}**

**static class ExcelFactory extends DocumentFactory {**

**public Document createDocument() {**

**return new ExcelDocument();**

**}**

**}**

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

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

**Document wordDoc = wordFactory.createDocument();**

**wordDoc.open();**

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

**Document pdfDoc = pdfFactory.createDocument();**

**pdfDoc.open();**

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

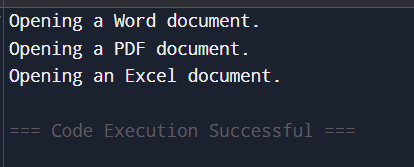
**Document excelDoc = excelFactory.createDocument();**

**excelDoc.open();**

**}**

**}**

**OUTPUT :**



Exercise 2: E-commerce Platform Search Function

import java.util.Arrays;

import java.util.Comparator;

public class EcommerceSearchExample {

static 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;

}

public String toString() {

return "ProductID: " + productId + ", Name: " + productName + ", Category: " + category;

}

}

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

for (Product product : products) {

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

return product;

}

}

return null;

}

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

int left = 0;

int right = products.length - 1;

while (left <= right) {

int mid = (left + right) / 2;

int comparison = products[mid].productName.compareToIgnoreCase(targetName);

if (comparison == 0) {

return products[mid];

} else if (comparison < 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, "Shirt", "Clothing"),

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

new Product(104, "Book", "Stationery"),

new Product(105, "Shoes", "Footwear")

};

String searchName1 = "Book";

Product foundLinear = linearSearch(products, searchName1);

System.out.println("Linear Search Result for \"" + searchName1 + "\": " + (foundLinear != null ? foundLinear : "Not Found"));

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

String searchName2 = "Mobile";

Product foundBinary = binarySearch(products, searchName2);

System.out.println("Binary Search Result for \"" + searchName2 + "\": " + (foundBinary != null ? foundBinary : "Not Found"));

/\*

Linear Search:

- Time Complexity: O(n)

- Simple but slow for large data sets.

Binary Search:

- Time Complexity: O(log n)

- Much faster, but requires sorted data.

Recommended for E-commerce:

- Binary search is better for large, sorted product listings.

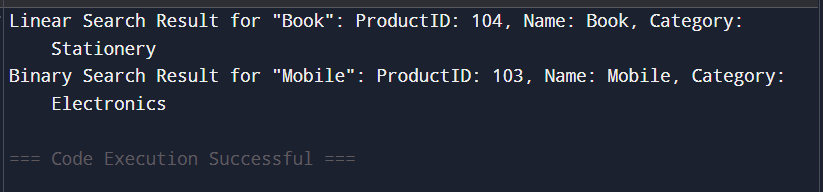
- For dynamic search where data is not sorted, use other fast structures like HashMap or indexing.

\*/

}

}

**OUTPUT :**



Exercise 7: Financial Forecasting

public class FinancialForecasting {

// Formula: futureValue = currentValue \* (1 + growthRate)^years

public static double predictFutureValue(double currentValue, double growthRate, int years) {

if (years == 0) {

return currentValue; // Base case: no growth

} else {

return (1 + growthRate) \* predictFutureValue(currentValue, growthRate, years - 1);

}

}

Time Complexity:

- The recursive function has a time complexity of O(n), where n = years.

Optimization:

- Use memoization or convert to an iterative version to avoid repeated calculations.

- For faster performance, use Math.pow(): currentValue \* Math.pow(1 + growthRate, years)

\*/

public static void main(String[] args) {

double currentValue = 10000.0; // Initial investment

double annualGrowthRate = 0.08; // 8% annual growth

int years = 5; // Predict after 5 years

double predictedValue = predictFutureValue(currentValue, annualGrowthRate, years);

System.out.printf("Predicted future value after %d years: ₹%.2f\n", years, predictedValue);

}

}

**OUTPUT :**

