**Module 1 - Design Patterns and Principles**

**Exercise 1: Implementing the Singleton Pattern**

public class SingletonPatternExample {

    static class Logger {

        private static Logger instance;

        private Logger() {

            System.out.println("Logger initialized");

        }

        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("Logging first message");

        logger2.log("Logging second message");

        if (logger1 == logger2) {

            System.out.println("Both logger1 and logger2 are the same instance.");

        } else {

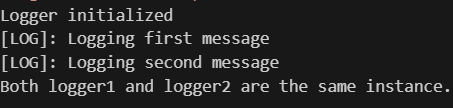
            System.out.println("Different logger instances exist!");

        }

    }

}

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.");

        }

    }

    static abstract class DocumentFactory {

        public abstract Document createDocument();

    }

    static class WordDocumentFactory extends DocumentFactory {

        public Document createDocument() {

            return new WordDocument();

        }

    }

    static class PdfDocumentFactory extends DocumentFactory {

        public Document createDocument() {

            return new PdfDocument();

        }

    }

    static class ExcelDocumentFactory extends DocumentFactory {

        public Document createDocument() {

            return new ExcelDocument();

        }

    }

    public static void main(String[] args) {

        DocumentFactory wordFactory = new WordDocumentFactory();

        DocumentFactory pdfFactory = new PdfDocumentFactory();

        DocumentFactory excelFactory = new ExcelDocumentFactory();

        Document wordDoc = wordFactory.createDocument();

        Document pdfDoc = pdfFactory.createDocument();

        Document excelDoc = excelFactory.createDocument();

        wordDoc.open();

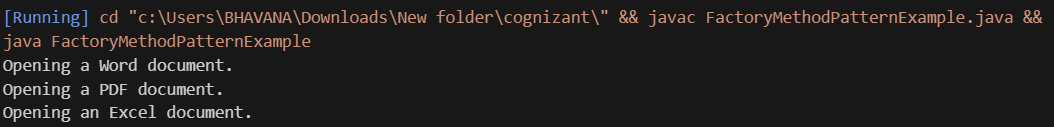
        pdfDoc.open();

        excelDoc.open();

    }

}

OUTPUT:



**Module 2 - Data Structures and Algorithms**

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

#### **Big O Notation**:

Big O notation describes how the runtime of an algorithm grows as the input size increases. It focuses on the **worst-case scenario** and helps compare the **efficiency** of different algorithms, especially as inputs grow large.

· **O(1)** – Constant time

· **O(log n)** – Logarithmic time

· **O(n)** – Linear time

· **O(n log n)** – Log-linear time

· **O(n²)** – Quadratic time

| **Case Type** | **Linear Search** | **Binary Search** |
| --- | --- | --- |
| **Best** | O(1) | O(1) |
| **Average** | O(n) | O(log n) |
| **Worst** | O(n) | O(log n) |

**Linear Search s**cans each item; worst case is when the item is at the end or not present.

**Binary Search o**nly works on sorted arrays; it halves the search space at each step.

**Linear Search:**

import java.util.Scanner;

public class LinearSearchDemo {

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

for (Product product : products) {

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

return product;

}

}

return null;

}

public static void main(String[] args) {

Product[] products = {

new Product(1, "Laptop", "Electronics"),

new Product(2, "Shirt", "Clothing"),

new Product(3, "Pen", "Stationery"),

new Product(4, "Phone", "Electronics")

};

Scanner sc = new Scanner(System.in);

System.out.print("Enter product name to search (Linear Search): ");

String input = sc.nextLine();

Product found = linearSearch(products, input);

if (found != null) {

System.out.println("Product found: " + found);

} else {

System.out.println("Product not found.");

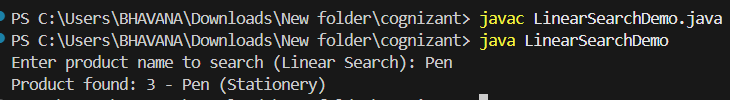
}

sc.close();

}

}

OUTPUT:



**Binary Search:**

import java.util.\*;

public class BinarySearchDemo {

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

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

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

while (left <= right) {

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

int cmp = products[mid].productName.compareToIgnoreCase(target);

if (cmp == 0) return products[mid];

else if (cmp < 0) left = mid + 1;

else right = mid - 1;

}

return null;

}

public static void main(String[] args) {

Product[] products = {

new Product(1, "Laptop", "Electronics"),

new Product(2, "Shirt", "Clothing"),

new Product(3, "Pen", "Stationery"),

new Product(4, "Phone", "Electronics")

};

Scanner sc = new Scanner(System.in);

System.out.print("Enter product name to search: ");

String input = sc.nextLine();

Product found = binarySearch(products, input);

if (found != null) {

System.out.println("Product found: " + found);

} else {

System.out.println("Product not found.");

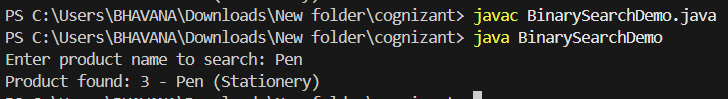
}

sc.close();

}

}

OUTPUT:



Analysis:

| **Algorithm** | **Time Complexity** | **Space Complexity** |
| --- | --- | --- |
| Linear Search | O(n) | O(1) |
| Binary Search | O(log n) | O(1) |

· **Linear Search** is simple and works on **unsorted** data, but is slow for large datasets.

· **Binary Search** is much faster but requires the array to be **sorted**, which may add overhead if data changes frequently.

**Conclusion**:

**Binary Search** is preferred if the product list is **pre-sorted or static**. It’s much faster and better for large datasets.Using Visual Studio, binary search makes the solution efficient and production-ready.

**Exercise 7: Financial Forecasting**

**Recursion** is a technique where a function calls itself to solve smaller sub-problems of the original problem.It helps simplify problems like factorial and Fibonacci series.

**Implementation:**

public class FinancialForecast {

public static double futureValue(double initialValue, double growthRate, int years) {

if (years == 0) {

return initialValue;

}

return futureValue(initialValue, growthRate, years - 1) \* (1 + growthRate);

}

public static void main(String[] args) {

explainRecursion();

double initialValue = 1000.0;

double growthRate = 0.10;

int years = 5;

System.out.println("Forecasting for " + years + " years with 10% growth..");

double resultRecursive = futureValue(initialValue, growthRate, years);

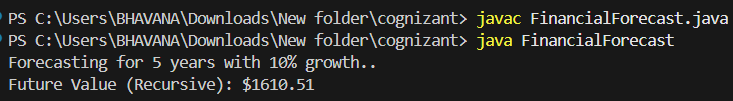
System.out.printf("Future Value (Recursive): $%.2f%n", resultRecursive);

System.out.println();

}

}

OUTPUT:



Time complexity of this recursive algorithm is O(n).This can be optimized to avoid excessive computation using iteration:

public class FinancialForecast {

public static double futureValueOptimized(double initialValue, double growthRate, int years) {

double result = initialValue;

for (int i = 1; i <= years; i++) {

result \*= (1 + growthRate);

}

return result;

}

public static void main(String[] args) {

double initialValue = 1000.0;

double growthRate = 0.10;

int years = 5;

double resultOptimized = futureValueOptimized(initialValue, growthRate, years);

System.out.printf("Future Value (Optimized): ₹%.2f%n", resultOptimized);

System.out.println();

}

}

OUTPUT:

