**DN 4.0 WEEK 1 MANDATORY PROBLEMS**

**MODULE 1: DESIGN PATTERNS AND PRINCIPLES**

**Exercise 1: Implementing the Singleton Pattern**

**Scenario:**

You need to ensure that a logging utility class in your application has only one instance throughout the application lifecycle to ensure consistent logging.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named SingletonPatternExample**.**
2. **Define a Singleton Class:**
   * Create a class named Logger that has a private static instance of itself.
   * Ensure the constructor of Logger is private.
   * Provide a public static method to get the instance of the Logger class.
3. **Implement the Singleton Pattern:**
   * Write code to ensure that the Logger class follows the Singleton design pattern.
4. **Test the Singleton Implementation:**
   * Create a test class to verify that only one instance of Logger is created and used across the application.

**CODE:**

**LOGGER.JAVA:**

public class Logger {

    private static Logger singleInstance;

    private Logger() {

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

    }

    public static Logger getInstance() {

        if (singleInstance == null) {

            singleInstance = new Logger();

        }

        return singleInstance;

    }

    public void log(String message) {

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

    }

}

**TESTLOGGER.JAVA:**

public class TestLogger {

    public static void main(String[] args) {

        Logger logger1 = Logger.getInstance();

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

        Logger logger2 = Logger.getInstance();

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

        if (logger1 == logger2) {

            System.out.println("Only one Logger instance is used.");

        } else {

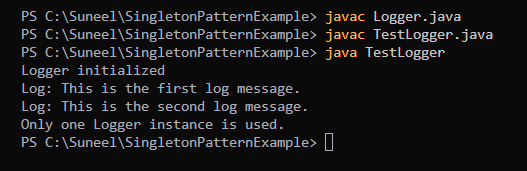
            System.out.println("Different Logger instances exist.");

        }

    }

}

**OUTPUT:**

****

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

**Scenario:**

You are developing a document management system that needs to create different types of documents (e.g., Word, PDF, Excel). Use the Factory Method Pattern to achieve this.

**Steps:**

1. **Create a New Java Project:**
   * Create a new Java project named FactoryMethodPatternExample.
2. **Define Document Classes:**
   * Create interfaces or abstract classes for different document types such as WordDocument, PdfDocument, and ExcelDocument.
3. **Create Concrete Document Classes:**
   * Implement concrete classes for each document type that implements or extends the above interfaces or abstract classes.
4. **Implement the Factory Method:**
   * Create an abstract class DocumentFactory with a method createDocument().
   * Create concrete factory classes for each document type that extends DocumentFactory and implements the createDocument() method.
5. **Test the Factory Method Implementation:**
   * Create a test class to demonstrate the creation of different document types using the factory method.

**CODE:**

**Document.java**

public interface Document {

    void open();

}

**ConcreteDocuments.java**

class WordDocument implements Document {

    public void open() {

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

    }

}

class PdfDocument implements Document {

    public void open() {

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

    }

}

class ExcelDocument implements Document {

    public void open() {

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

    }

}

**DocumentFatory.java**

public class DocumentFactory {

    public static Document createDocument(String type) {

        switch (type.toLowerCase()) {

            case "word":

                return new WordDocument();

            case "pdf":

                return new PdfDocument();

            case "excel":

                return new ExcelDocument();

            default:

                throw new IllegalArgumentException("Unknown document type: " + type);

        }

    }

}

**TestDocumentFactory.java**

public class TestDocumentFactory {

    public static void main(String[] args) {

        Document doc1 = DocumentFactory.createDocument("word");

        doc1.open();

        Document doc2 = DocumentFactory.createDocument("pdf");

        doc2.open();

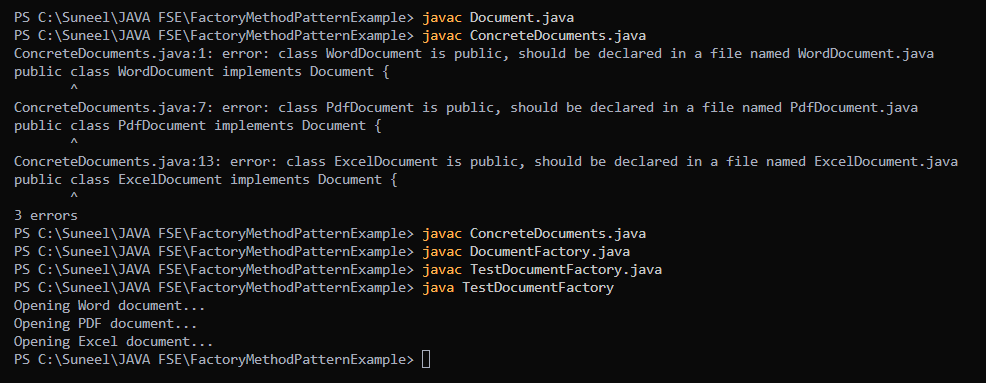
        Document doc3 = DocumentFactory.createDocument("excel");

        doc3.open();

    }

}

**OUTPUT:**

****

**MODULE 2: ALGORITHMS – DATA STRUCTURES**

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

**Scenario:**

You are working on the search functionality of an e-commerce platform. The search needs to be optimized for fast performance.

**Steps:**

1. **Understand Asymptotic Notation:**
   * Explain Big O notation and how it helps in analyzing algorithms.
   * Describe the best, average, and worst-case scenarios for search operations.
2. **Setup:**
   * Create a class **Product** with attributes for searching, such as **productId, productName**, and **category**.
3. **Implementation:**
   * Implement linear search and binary search algorithms.
   * Store products in an array for linear search and a sorted array for binary search.
4. **Analysis:**
   * Compare the time complexity of linear and binary search algorithms.

Discuss which algorithm is more suitable for your platform and why.

**Big O notation** describes the upper bound (worst-case performance) of an algorithm in terms of input size n.

Normally, For Linear Search it will be O(n)

For Binary Search it will be O (log n) because it uses divide and conquer method so, the time complexity will be in logarithms.

Best Case for Linear Search is when the search element is present in the first index and Worst case will be when the search element is present in the Last index of the array.

Best Case for Binary Search is when the search element is present in the Middle of the array and Worst case is when the search element is present in the First or Last index of the array.

**CODE**

**Product.java**

public class Product {

    int productId;

    String productName;

    String category;

    public Product(int id, String name, String category) {

*this*.productId = id;

*this*.productName = name;

*this*.category = category;

    }

    public String toString() {

        return "[" + productId + ", " + productName + ", " + category + "]";

    }

}

**Searchutil.java**

public class SearchUtil {

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

        for (Product p : products) {

            if (p.productName.equalsIgnoreCase(name)) {

                return p;

            }

        }

        return null;

    }

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

        int low = 0, high = products.length - 1;

        while (low <= high) {

            int mid = (low + high) / 2;

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

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

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

            else high = mid - 1;

        }

        return null;

    }

}

**SearchTest.java**

import java.util.Arrays;

import java.util.Comparator;

public class SearchTest {

    public static void main(String[] args) {

        Product[] products = {

            new Product(101, "iPhone", "Electronics"),

            new Product(102, "Shirt", "Apparel"),

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

            new Product(104, "Shoes", "Footwear"),

            new Product(105, "Watch", "Accessories")

        };

        Product found1 = SearchUtil.linearSearch(products, "Laptop");

        System.out.println("Linear Search: " + (found1 != null ? found1 : "Not found"));

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

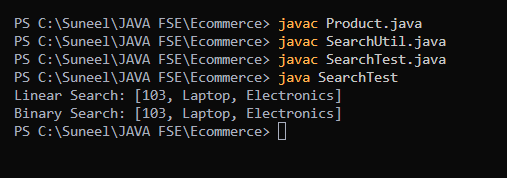
        Product found2 = SearchUtil.binarySearch(products, "Laptop");

        System.out.println("Binary Search: " + (found2 != null ? found2 : "Not found"));

    }

}

**OUTPUT:**



Linear Search works best on Unsorted data and small datasets and works poorly on large datasets and efficient with small datasets.

Binary Search works best on large datasets but the only problem is that the data should be sorted first to proceed further and results will be quicker when compared with linear Search.

**Conclusion:**

**So, if the data is small then we choose linear Search and if the data is large we sort the data and choose Binary Search.**

**Exercise 7: Financial Forecasting**

**Scenario:**

You are developing a financial forecasting tool that predicts future values based on past data.

**Steps:**

1. **Understand Recursive Algorithms:**
   * Explain the concept of recursion and how it can simplify certain problems.
2. **Setup:**
   * Create a method to calculate the future value using a recursive approach.
3. **Implementation:**
   * Implement a recursive algorithm to predict future values based on past growth rates.
4. **Analysis:**
   * Discuss the time complexity of your recursive algorithm.
   * Explain how to optimize the recursive solution to avoid excessive computation.

**Recursion:**

This means that if problem is bigger to solve then we try to divide the problem into smaller sub problems and solve each sub problem and finally combine to get the final answer which makes the work easier but in recursion we stack to store the function calls so if uses more storage when compared with iterative method but recursion is efficient in terms of time complexity when compared with iterative method.

**Recursive future formula:**

The future value FVFVFV of an investment can be predicted using the formula:

FV(n)=PV×(1+r) ^ (n)

Where:

* PV = Present Value (initial amount)
* r = growth rate (e.g., 5% → 0.05)
* n = number of periods (years)

**CODE:**

**FinancialForecast.java**

public class FinancialForecast {

    public static double futureValue(double presentValue, double rate, int years) {

        if (years == 0) {

            return presentValue;

        }

        return (1 + rate) \* futureValue(presentValue, rate, years - 1);

    }

    public static void main(String[] args) {

        double presentValue = 10000;

        double rate = 0.05;

        int years = 5;

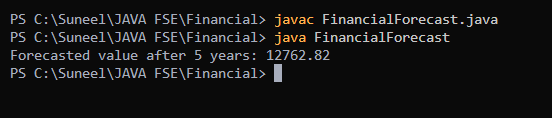
        double forecastedValue = futureValue(presentValue, rate, years);

        System.out.printf("Forecasted value after %d years: %.2f\n", years, forecastedValue);

    }

}

**OUTPUT:**

****