**Week-1 Mandatory Hands-on Exercises:**

**Skill: Design principles and patterns**

**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.

Code:

//Create a class named Logger that has a private static instance of itself.

public class Logger {

private static Logger instance;

// Ensure the constructor of Logger is private.

private Logger() {

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

}

// Provide a public static method to get the instance of the Logger class.

public static Logger getInstance() {

if (instance == null) {

instance = new Logger();

}

return instance;

}

// Example log method

public void log(String message) {

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

}

}

// Create a test class to verify that only one instance of Logger is created and used across the application.

public class loggerTest {

public static void main(String[] args) {

Logger logger1 = Logger.getInstance();

logger1.log("First log message.");

Logger logger2 = Logger.getInstance();

logger2.log("Second log message.");

if (logger1 == logger2) {

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

} else {

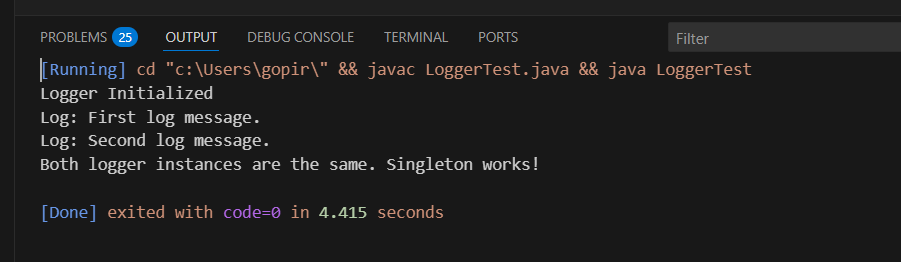
System.out.println("Logger instances are different. Singleton failed.");

}

}

}

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.

Code:

//Create interfaces or abstract classes for different document types such as WordDocument, PdfDocument, and ExcelDocument.

public interface Document {

void open();

}

// Implement concrete classes for each document type that implements or extends the above interfaces or abstract classes.

// WordDocument

public class WordDocument implements Document {

@Override

public void open() {

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

}

}

// PdfDocument

public class PdfDocument implements Document {

@Override

public void open() {

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

}

}

// ExcelDocument

public class ExcelDocument implements Document {

@Override

public void open() {

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

}

}

//Create an abstract class DocumentFactory with a method createDocument().

public abstract class DocumentFactory {

public abstract Document createDocument();

}

//Create concrete factory classes for each document type that extends DocumentFactory and implements the createDocument() method.

// WordDocumentFactory

public class WordDocumentFactory extends DocumentFactory {

@Override

public Document createDocument() {

return new WordDocument();

}

}

// PdfDocumentFactory

public class PdfDocumentFactory extends DocumentFactory {

@Override

public Document createDocument() {

return new PdfDocument();

}

}

// ExcelDocumentFactory

public class ExcelDocumentFactory extends DocumentFactory {

@Override

public Document createDocument() {

return new ExcelDocument();

}

}

// Create a test class to demonstrate the creation of different document types using the factory method.

public class DocumentFactoryTest {

public static void main(String [] args) {

// Word document

DocumentFactory wordFactory = new WordDocumentFactory();

Document wordDoc = wordFactory.createDocument();

wordDoc.open();

// PDF document

DocumentFactory pdfFactory = new PdfDocumentFactory();

Document pdfDoc = pdfFactory.createDocument();

pdfDoc.open();

// Excel document

DocumentFactory excelFactory = new ExcelDocumentFactory();

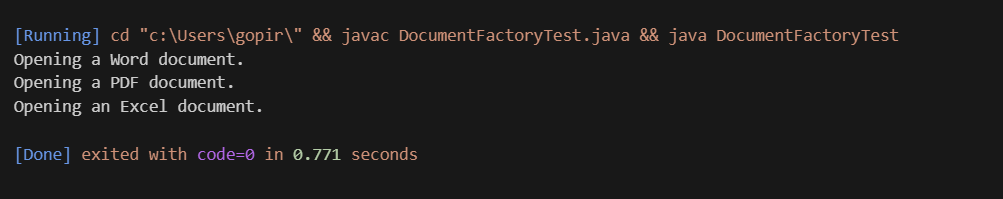
Document excelDoc = excelFactory.createDocument();

excelDoc.open();

}

}

Output:



**Skill: Data Structures and Algorithms**

**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.

Code:

// code for productid, product name and category

import java.util.Objects;

public class Product implements Comparable<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;

    }

    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;

    }

    // code to sort products which is required for binary search and we will sort products by its name

    @Override

    public int compareTo(Product other) {

        return this.productName.toLowerCase().compareTo(other.productName.toLowerCase());

    }

    //equals and hashCode for better object comparison

    @Override

    public boolean equals(Object o) {

        if (this == o) return true;

        if (o == null || getClass() != o.getClass()) return false;

        Product product = (Product) o;

        // Compare based on productName for search purposes (case-insensitive)

        return productName.equalsIgnoreCase(product.productName);

    }

    @Override

    public int hashCode() {

        return Objects.hash(productName.toLowerCase());

    }

}

// Search Algorithms (linear and binary search)

import java.util.ArrayList;

import java.util.Collections;

import java.util.List;

public class SearchAlgorithms {

    public Product linearSearch(List<Product> products, String searchKey) {

        for (Product product : products) {

            if (product.getProductName().equalsIgnoreCase(searchKey)) {

                return product;

            }

        }

        return null; // Product not found

    }

    public Product binarySearch(List<Product> sortedProducts, String searchKey) {

        int left = 0;

        int right = sortedProducts.size() - 1;

        while (left <= right) {

            int mid = left + (right - left) / 2; // Avoid potential overflow for very large lists

            String midProductName = sortedProducts.get(mid).getProductName().toLowerCase();

            int comparison = midProductName.compareTo(searchKey.toLowerCase());

            if (comparison == 0) {

                return sortedProducts.get(mid); // Found the product

            } else if (comparison < 0) {

                left = mid + 1; // Search in the right half

            } else {

                right = mid - 1; // Search in the left half

            }

        }

        return null; // Product not found

    }

    public static void main(String[] args) {

        // Create some products

        List<Product> productsData = new ArrayList<>();

        productsData.add(new Product("1", "Mobile Phones", "Electronics"));

        productsData.add(new Product("5", "Electric Pressure Cooker", "Home & Kitchen"));

        productsData.add(new Product("2", "Smart Tv", "Electronics"));

        productsData.add(new Product("7", "Desk Chair", "Furniture"));

        productsData.add(new Product("3", "Hand Sanitizer", "Health and Household"));

        productsData.add(new Product("6", "laptop", "Electronics"));

        productsData.add(new Product("4", "Headphones", "Audio"));

        SearchAlgorithms searcher = new SearchAlgorithms();

        // Linear search testing

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

        System.out.println("Original Products for Linear Search:");

        for (Product p : productsData) {

            System.out.println(p);

        }

        Product foundProduct = searcher.linearSearch(productsData, "Mobile Phones");

        System.out.println("Found 'Mobile Phones' by using linear search: " + foundProduct);

        foundProduct = searcher.linearSearch(productsData, "Tablet");

        System.out.println("Found 'Tablet' by using linear search: " + foundProduct);

        // Binary search testing

        System.out.println("\n--- Binary Search ---");

        // For binary search, the products must be sorted by the product name

        List<Product> sortedBinarySearchProducts = new ArrayList<>(productsData);

        Collections.sort(sortedBinarySearchProducts); // Sorts based on compareTo in Product class

        System.out.println("Sorted Products for Binary Search:");

        for (Product p : sortedBinarySearchProducts) {

            System.out.println(p);

        }

        foundProduct = searcher.binarySearch(sortedBinarySearchProducts, "Smart Tv");

        System.out.println("Found 'Smart Tv' by using binary search: " + foundProduct);

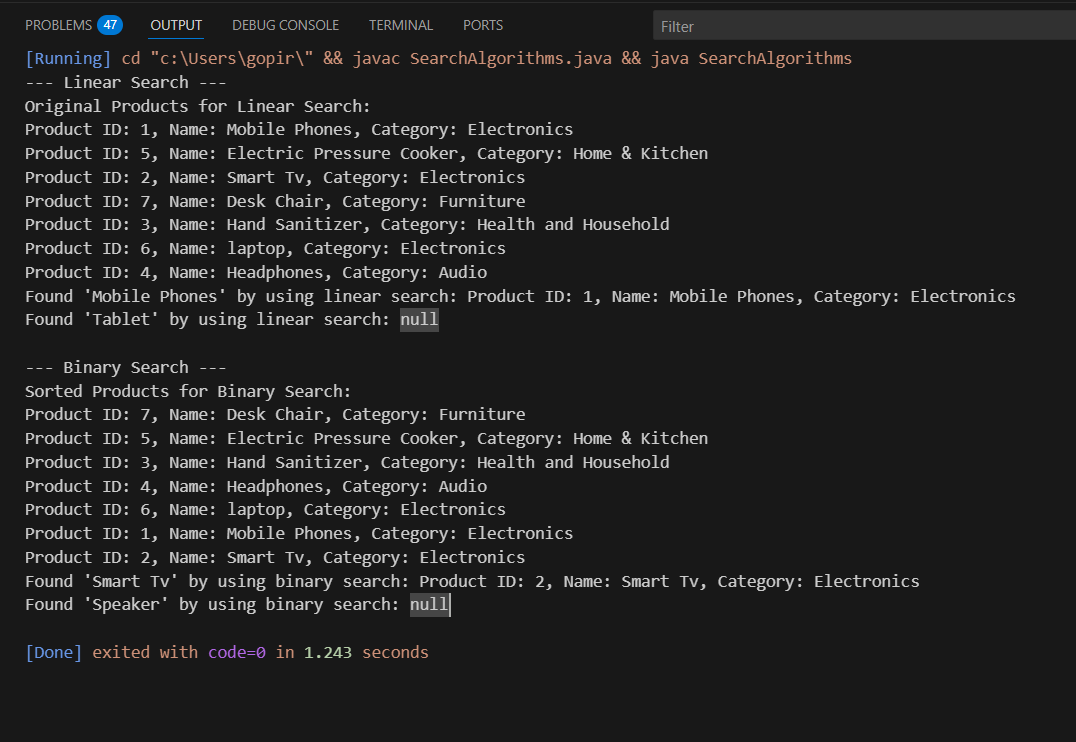
        foundProduct = searcher.binarySearch(sortedBinarySearchProducts, "Speaker");

        System.out.println("Found 'Speaker' by using binary search: " + foundProduct);

    }

}

Output:



which algorithm is more suitable for your platform and why?

For an e-commerce platform when we compare both linear and binary search functions, Binary search is more suitable than Linear search, mainly when the number of products grows. Because E-commerce platforms typically deal with millions of products and a Linear search for million products takes more time sometimes it takes minutes but for Binary search returns results instantaneously due to its logarithmic time complexity.

**Exercise 7: Financial Forecasting**

**Scenario:**

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

Code:

public class FinancialForecast {

    // Recursive method to calculate future value

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

        if (years == 0) {

            return currentValue;

        }

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

    }

    public static void main(String[] args) {

        double initialValue = 10000.0;  // ₹10,000

        double growthRate = 0.08;       // 8% growth per year

        int years = 5;

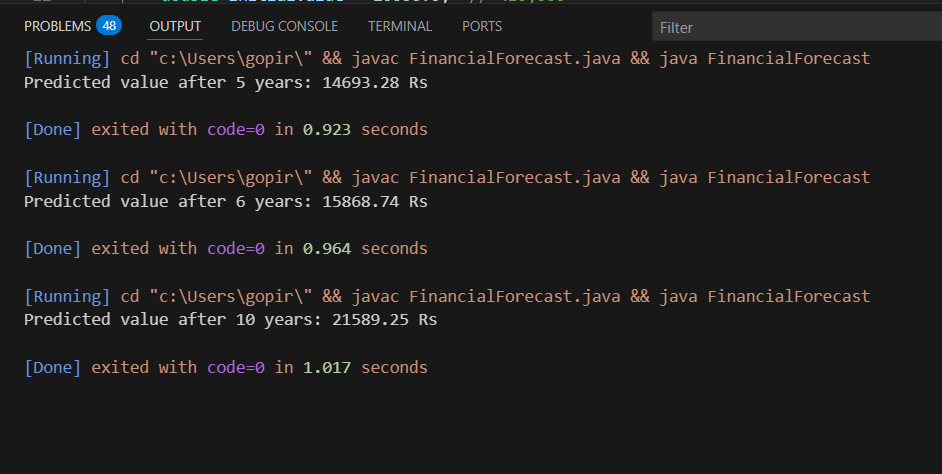
        double futureValue = predictFutureValue(initialValue, growthRate, years);

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

    }

}

Output:



The Time Complexity of above Recursive algorithm is O(n).

Explain how to optimize the recursive solution to avoid excessive computation?

To optimize the recursive algorithm we use iteration or memorization.

The benefit of using memorization is it avoids recomputation for already solved years.

The benefit of using iteration is there will be no stack overflow risk but by using recursive algorithm we get stack overflow risk. Iteration is more efficient in java.