**DIGITAL NURTURE 4.0 JavaFSE**

**WEEK 1**

**DESIGN PATTERNS**

**Exercise 1: Implementing the Singleton Pattern:**

public class Logger1 {

    private static Logger1 instance;

private Logger1() {

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

    }

public static Logger1 getInstance() {

        if (instance == null) {

            instance = new Logger1();

        }

        return instance;

    }

public void display() {

        System.out.println("Singleton Object");

    }

public static void main(String args[]) {

        Logger1 logger1 = Logger1.getInstance();

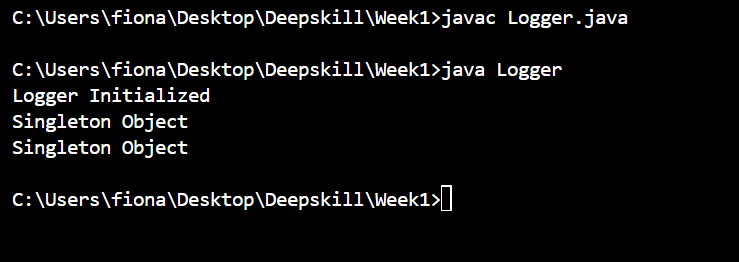
        Logger1 logger2 = Logger1.getInstance();

        logger1.display();

        logger2.display();

    }

}



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

public class FactoryMethodPattern {

 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 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();

        Document wordDoc = wordFactory.createDocument();

        wordDoc.open();

        DocumentFactory pdfFactory = new PdfDocumentFactory();

        Document pdfDoc = pdfFactory.createDocument();

        pdfDoc.open();

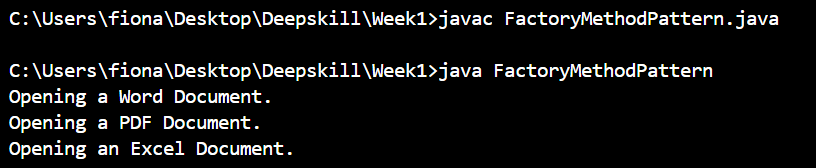
        DocumentFactory excelFactory = new ExcelDocumentFactory();

        Document excelDoc = excelFactory.createDocument();

        excelDoc.open();

    }

}



**ALGORITHMS DATA STRUCTURES**

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

**Big O :**

Big O Notation is a way to describe how the performance (time or space) of an algorithm grows as the input size increases.It focuses on the worst-case scenario

Best case : O(1)

Worst case : O(log n)

Average case : O(n)

import java.util.Arrays;

import java.util.Comparator;

import java.util.Scanner;

public class Ecommerce {

     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 + ", " + productName + ", " + category + "]";

        }

    }

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

        for (Product product : products) {

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

                return product;

            }

        }

        return null;

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

        int low = 0;

        int 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;

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

             Product[] products = {

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

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

            new Product(3, "Phone", "Electronics"),

            new Product(4, "Shoes", "Footwear"),

            new Product(5, "Keyboard", "Accessories") };

        Scanner scanner = new Scanner(System.in);

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

        String searchName = scanner.nextLine();

        System.out.println("\nLinear Search");

       Product result1 = linearSearch(products, searchName);

        if (result1 != null) {

            System.out.println("Product found: " + result1);

        } else {

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

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

        System.out.println("\nBinary Search");

       Product result2 = binarySearch(products, searchName);

        if (result2 != null) {

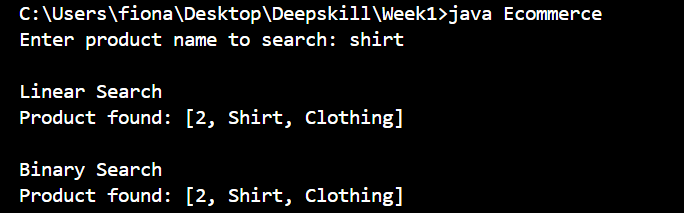
            System.out.println("Product found: " + result2);

        } else {

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

        }}

}



Comparing Linear Search and Binary Search Time Complexity :

Linear Search : O(n)

Binary Search : O(log n)

E-commerce platforms (like Amazon,meesho, etc.) typically deal with:

* Thousands to millions of products
* Fast user queries like searching by product name or ID

Binary Search is more Suitable as it takes O(log n) time .

**Exercise 7 :Financial Forecasting :**

**Recursion:**

Recursion is a programming concept where a method calls itself to solve a problem.

It breaks a complex problem into smaller sub-problems, solving each one using the same logic, until it reaches a base case

public class FinancialForecasting {

    public static double forecastRecursive(double presentValue, double growthRate, int years) {

        if (years == 0) {

            return presentValue;

        } else {

            return (1 + growthRate) \* forecastRecursive(presentValue, growthRate, years - 1);

        }

    }

    public static void main(String[] args) {

        double presentValue = 1000.0;

        double growthRate = 0.10;

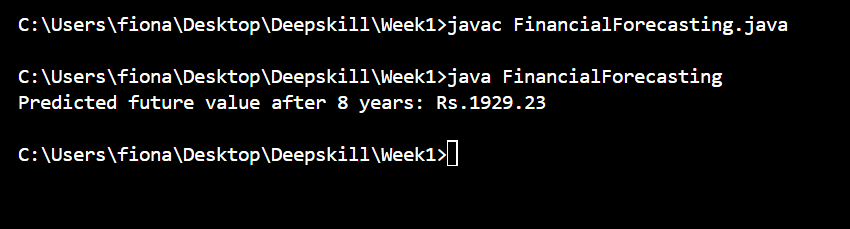
        int years = 5;

        double futureValue = forecastRecursive(presentValue, growthRate, years);

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

    }

}



**The Time Complexity : O(n)**

**Optimization of the recursive solution :**

* Stack overflow for large n (too many nested calls)
* Redundant computation if there were overlapping subproblems (not in this example, but common in other recursive problems).

For this problem, it’s more efficient to use an iterative approach because:

* There’s no overlapping subproblem to reuse ,so memoization doesn’t help.
* An iterative loop avoids the stack overhead.