# WEEK 1 – ENGINEERING CONCEPTS

# 1. DESIGN PATTERNS AND PRINCIPLES

## Exercise 1 : Implementing the Singleton Pattern

**Scenario**: Ensure that only one instance of a Logger utility class exists throughout the application lifecycle.

**Java Code:**

**// Logger.java**

package logger;

public class Logger {

    private static Logger instance;

    private Logger() {

        // private constructor to prevent instantiation

        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);

    }

}

**// SingletonTest.java**

package test;

import logger.Logger;

public class SingletonTest {

    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 loggers are the same instance.");

        } else {

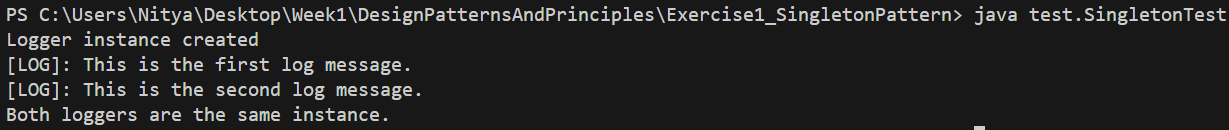
            System.out.println("Different logger instances exist.");

        }

    }

}

**OUTPUT :**



## Exercise 2 : Factory Method Pattern

**Scenario:** Create different types of documents (Word, PDF, Excel) using a factory method to manage instantiation.

Factory Pattern enables modular and scalable code design. O(1) creation time and easy to extend.

**Java Code:**

**// Document.java**

package documents;

public interface Document {

void open();

}

**// ExcelDocument.java**

package documents;

public class ExcelDocument implements Document {

    @Override

    public void open() {

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

    }

}

**// PdfDocument.java**

package documents;

public class PdfDocument implements Document {

    @Override

    public void open() {

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

    }

}

**// WordDocument.java**

package documents;

public class WordDocument implements Document {

    @Override

    public void open() {

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

    }

}

**// DocumentFactory.java**

package factories;

import documents.Document;

public abstract class DocumentFactory {

    public abstract Document createDocument();

}

**// ExcelDocumentFactory.java**

package factories;

import documents.Document;

import documents.ExcelDocument;

public class ExcelDocumentFactory extends DocumentFactory {

    @Override

    public Document createDocument() {

        return new ExcelDocument();

    }

}

**// PdfDocumentFactory.java**

package factories;

import documents.Document;

import documents.PdfDocument;

public class PdfDocumentFactory extends DocumentFactory {

    @Override

    public Document createDocument() {

        return new PdfDocument();

    }

}

**// WordDocumentFactory.java**

package factories;

import documents.Document;

import documents.WordDocument;

public class WordDocumentFactory extends DocumentFactory {

    @Override

    public Document createDocument() {

        return new WordDocument();

    }

}

**// FactoryPatternTest.java**

package test;

import documents.Document;

import factories.\*;

public class FactoryPatternTest {

    public static void main(String[] args) {

        DocumentFactory wordFactory = new WordDocumentFactory();

        Document word = wordFactory.createDocument();

        word.open();

        DocumentFactory pdfFactory = new PdfDocumentFactory();

        Document pdf = pdfFactory.createDocument();

        pdf.open();

        DocumentFactory excelFactory = new ExcelDocumentFactory();

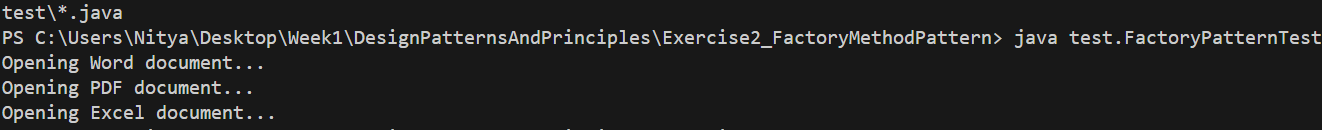
        Document excel = excelFactory.createDocument();

        excel.open();

    }

}

**OUTPUT:**



# 2. Data Structures and Algorithms

## Exercise 2 : E-commerce Platform Search Function

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

.

**Asymptotic Notation**

**Asymptotic Notation** helps evaluate algorithm performance based on input size.

-**Big O (O)**: Worst-case performance.

-**Omega (Ω)**: Best-case performance.

- **Theta (Θ)**: Average-case performance.

**Implementation**

Implemented two algorithms for search:

1. Linear Search: Scans each product one-by-one.

2. Binary Search: Divides and conquers on a sorted list.

**Java Code:**

**// Product.java**

package models;

public class Product {

    private int productId;

    private String productName;

    private String category;

    public Product(int productId, String productName, String category) {

        this.productId = productId;

        this.productName = productName;

        this.category = category;

    }

    public int getProductId() {

        return productId;

    }

    public String getProductName() {

        return productName;

    }

    public String getCategory() {

        return category;

    }

    @Override

    public String toString() {

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

    }

}

**// SearchService.java**

package search;

import models.Product;

public class SearchService {

    // Linear Search

    public static Product linearSearch(Product[] products, int targetId) {

        for (Product product : products) {

            if (product.getProductId() == targetId) {

                return product;

            }

        }

        return null;

    }

    // Binary Search

    public static Product binarySearch(Product[] products, int targetId) {

        int left = 0;

        int right = products.length - 1;

        while (left <= right) {

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

            int midId = products[mid].getProductId();

            if (midId == targetId) {

                return products[mid];

            } else if (midId < targetId) {

                left = mid + 1;

            } else {

                right = mid - 1;

            }

        }

        return null;

    }

}

**// SearchTest.java**

package test;

import java.util.Arrays;

import java.util.Comparator;

import models.Product;

import search.SearchService;

public class SearchTest {

    public static void main(String[] args) {

        Product[] products = {

            new Product(105, "Mouse", "Electronics"),

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

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

            new Product(102, "Notebook", "Stationery"),

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

        };

        // Sort for Binary Search (by productId)

        Arrays.sort(products, Comparator.comparingInt(Product::getProductId));

        int targetId = 103;

        // Linear Search

        Product resultLinear = SearchService.linearSearch(products, targetId);

        System.out.println("Linear Search Result: " + resultLinear);

        // Binary Search

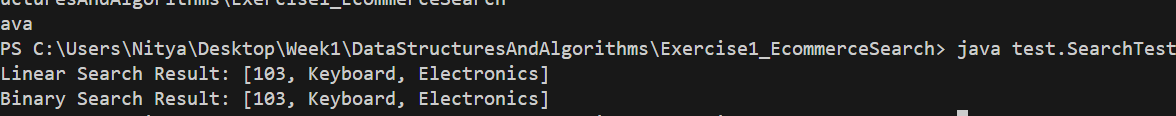
        Product resultBinary = SearchService.binarySearch(products, targetId);

        System.out.println("Binary Search Result: " + resultBinary);

    }

}

**OUTPUT:**



**Time Complexity**

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

**When to Use Which Search Algorithm**

| **Use Case** | **Recommended Search** |
| --- | --- |
| Small data, unsorted | Linear Search |
| Large data, sorted | Binary Search |
| Real-time updates to product list | Linear Search (simpler) |
| Static list or frequent repeated searches | Binary Search (more optimal) |

**CONCLUSION**

- Binary Search is efficient but only works on sorted data.

- Understanding complexity helps you choose the best algorithm for the job.

## EXERCISE 7 : Financial Forecasting

**Scenario**: Developing a financial forecasting tool that predicts future values based on past data.

Memoization reduces computation. Recursive time complexity O(n), space O(n).

**Recursive Algorithms**

**Recursion** is a method in programming where a function calls itself to solve a smaller part of the same problem.It makes solving big or repeated problems easier — like calculating factorials, going through trees, or predicting future values.  
In forecasting, each future value depends on the one before it, so recursion works well for this.

**Implementation**

- Predicts future financial value using the formula:

  Future = Present × (1 + GrowthRate)^Years

**Java Code:**

**// ForecastService.java**

package forecast;

public class ForecastService {

    // Calculating future value based on compound growth

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

        if (years == 0) {

            return currentValue;

        }

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

    }

    // Optimized recursive method using memoization

    public double predictFutureValueMemo(double currentValue, double growthRate, int years, double[] memo) {

        if (years == 0) {

            return currentValue;

        }

        if (memo[years] != 0.0) {

            return memo[years];

        }

        memo[years] = predictFutureValueMemo(currentValue \* (1 + growthRate), growthRate, years - 1, memo);

        return memo[years];

    }

}

**// ForecastTest.java**

package test;

import forecast.ForecastService;

public class ForecastTest {

    public static void main(String[] args) {

        ForecastService service = new ForecastService();

        double currentValue = 10000.0;

        double growthRate = 0.10;

        int years = 5;

        // Simple recursive forecast

        double futureValue = service.predictFutureValue(currentValue, growthRate, years);

        System.out.println("Future Value (Simple Recursion): ₹" + futureValue);

        // Optimized recursive forecast using memoization

        double[] memo = new double[years + 1];

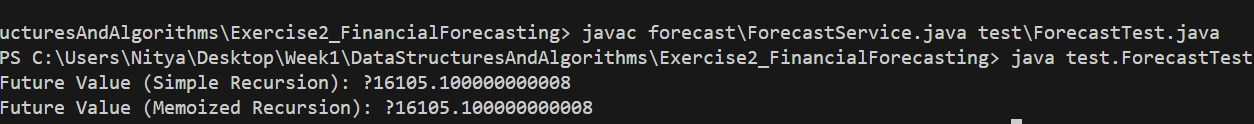
        double memoValue = service.predictFutureValueMemo(currentValue, growthRate, years, memo);

        System.out.println("Future Value (Memoized Recursion): ₹" + memoValue);

    }

}

**OUTPUT:**



**Analysis**

**Time Complexity:**

-**Simple Recursion:** O(n) - Linear time as the function calls itself n times.

- **Memoized Recursion:** O(n) - But faster due to caching previous results.

**Optimization**

Memoization prevents recalculating the same values, significantly improving performance for large inputs.

**CONCLUSION**

- Recursive solutions offer clarity but must be used carefully to avoid stack overflows or inefficiency.

- Always consider memoization or converting to iterative solutions when performance is critical.