WEEK 1

Design principles & Patterns

*Exercise 1: Singleton Pattern*

**Code for exercise 1:**

public 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();

logger1.log("First log message");

Logger logger2 = Logger.getInstance();

logger2.log("Second log message");

if (logger1 == logger2) {

System.out.println("Both are same instance (Singleton confirmed)");

} else {

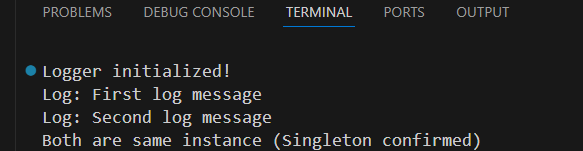
System.out.println("Different instances – Singleton failed");

}

}

}

**Output for exercise 1:**



***Exercise 2: Factory Method Pattern***

**Code for exercise 2:**

interface FileType {

void display();

}

class WordFile implements FileType {

public void display() {

System.out.println("Word file is now open.");

}

}

class PdfFile implements FileType {

public void display() {

System.out.println("PDF file is now open.");

}

}

class ExcelFile implements FileType {

public void display() {

System.out.println("Excel file is now open.");

}

}

abstract class FileFactory {

abstract FileType createFile();

}

class WordFileFactory extends FileFactory {

public FileType createFile() {

return new WordFile();

}

}

class PdfFileFactory extends FileFactory {

public FileType createFile() {

return new PdfFile();

}

}

class ExcelFileFactory extends FileFactory {

public FileType createFile() {

return new ExcelFile();

}

}

public class DocumentFactoryExample {

public static void main(String[] args) {

FileFactory word = new WordFileFactory();

word.createFile().display();

FileFactory pdf = new PdfFileFactory();

pdf.createFile().display();

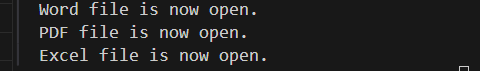
FileFactory excel = new ExcelFileFactory();

excel.createFile().display();

}

}

**Output for exercise 2:**

****

Data structures and Algorithms

*Exercise 2: E-commerce Platform Search Function*

**Asymptotic Notation:**

Big O notation is used to describe the efficiency of an algorithm in terms of input size. It helps analyze how an algorithm performs as the dataset grows.

* **Best Case**: The fastest possible performance (e.g., item found at the first index).
* **Average Case**: Typical expected performance with randomly distributed data.
* **Worst Case**: The slowest performance (e.g., item is not found).

Time Complexities:

**Linear Search**:

- Best Case: O(1)

- Average Case: O(n/2)

- Worst Case: O(n)

**Binary Search**:

- Best Case: O(1)

- Average/Worst Case: O(log n)

**Code for exercise 2:**

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 class SearchFunction {

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

        for (int i = 0; i < products.length; i++) {

            if (products[i].productName.equalsIgnoreCase(target)) {

                return i;

            }

        }

        return -1;

    }

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

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

        while (low <= high) {

            int mid = (low + high) / 2;

            int result = products[mid].productName.compareTo(target);

            if (result == 0) return mid;

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

            else high = mid - 1;

        }

        return -1;

}

    public static void main(String[] args) {

        Product[] products = new Product[] {

            new Product(105, "Book", "Education"),

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

            new Product(102, "Mobile", "Electronics"),

            new Product(103, "Shampoo", "Personal Care"),

            new Product(104, "T-Shirt", "Clothing")

        };

        String searchTerm = "Shampoo";

        int linearResult = linearSearch(products, searchTerm);

        int binaryResult = binarySearch(products, searchTerm);

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

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

    }

}

**Output for exercise 2:**



**Time Complexity Analysis:**

Linear search takes O(n) time in the worst case, while binary search takes only O(log n).

Binary search is more efficient for large, sorted datasets.

Therefore, for an e-commerce platform where speed matters, binary search is more suitable.

*Exercise 7: Financial Forecasting*

Recursion is a programming technique where a method calls itself to solve a smaller part of the problem.

It helps simplify complex tasks like mathematical sequences, tree traversals, or predictions by breaking them into smaller, manageable problems.

**Code for exercise 7:**

public class FinancialForecast {

    public static double forecastValue(double value, double rate, int years) {

        if (years == 0) return value;

        return forecastValue(value \* (1 + rate), rate, years - 1);

    }

    public static void main(String[] args) {

        double value = 10000;

        double rate = 0.10;

        int years = 5;

        double result = forecastValue(value, rate, years);

        System.out.println("Forecasted Value after " + years + " years: " + result);

    }

}

**Output for exercise 7:**



**Time Complexity Analysis:**

The time complexity of this recursive solution is O(n), where n is the number of years. Each recursive call performs one multiplication and reduces the year count by 1.

**Optimization Approach:**

The recursion is efficient for small values of n.

However, for large values, it may cause a stack overflow.

To optimize, we can convert the recursive logic into an iterative loop that calculates the final value using a simple for loop.