**WEEK – 01**

**ENGINEERING CONCEPTS**

**Module -01 Design Patterns and Principles**

**Superset ID: 6262264**

**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's lifecycle. This ensures consistent logging, prevents multiple log file handlers, and improves resource management.

**SOLUTION :**

**Logger.java : (Defines Singleton Pattern)**

Class

public class Logger {

// Private static instance (holds the one and only Logger object)

private static Logger instance;

// Private constructor so external classes cannot create new objects

private Logger() {

System.out.println("Logger instance created.");

}

}

**Logger.java : (Implementing Singleton Pattern)**

// Public static method to get the single instance

public static Logger getInstance() {

if (instance == null) {

instance = new Logger(); // create only if it doesn't exist

}

return instance;

}

// Method to simulate logging

public void log(String message) {

System.out.println("[LOG]: " + message);

}

}

**main.java : (Testing Singleton Pattern)**

public class Main {

public static void main(String[] args) {

System.out.println("Testing Singleton Logger...");

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("Confirmed: Both loggers are the same instance.");

} else {

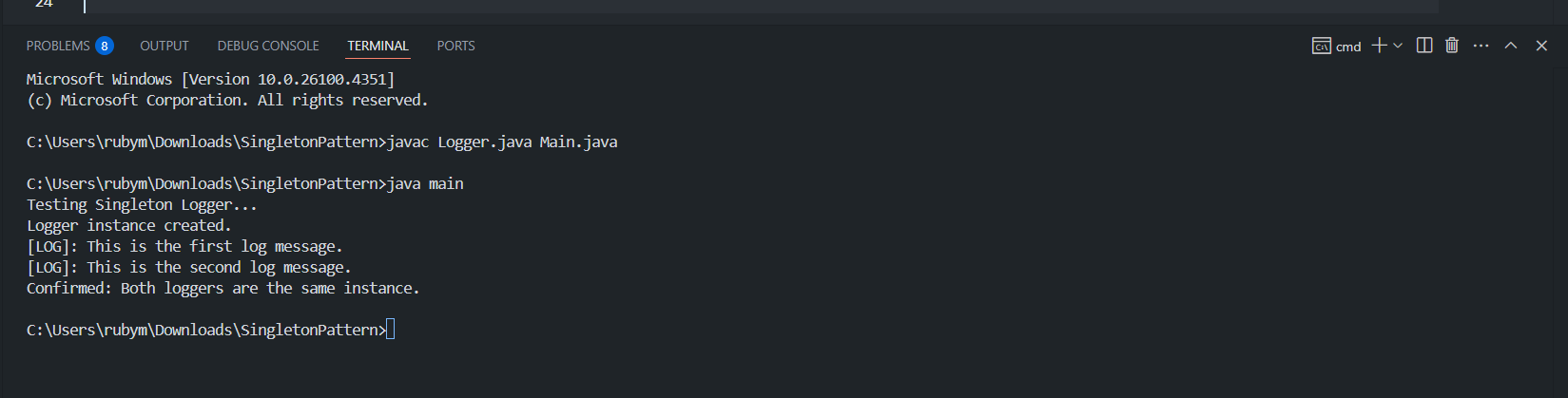
System.out.println("Error: Multiple logger instances exist!");

}

}

}

**Output :**



**Explanation:**

* The provided code follows the Singleton Design Pattern to ensure that only a single instance of the Logger class is created and shared across the application.
* It uses a private static variable to store the instance and a private constructor to restrict object creation from outside the class.
* A public static method getInstance() lazily initializes and returns the single shared Logger object whenever needed.

**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.

**Solution:**

**Main.java**

class main {

    public static void main(String[] args) {

        DocumentFactory wordFactory = new WordFactory();

        Document word = wordFactory.createDocument();

        word.preview();

        DocumentFactory pdfFactory = new PdfFactory();

        Document pdf = pdfFactory.createDocument();

        pdf.preview();

        DocumentFactory excelFactory = new ExcelFactory();

        Document excel = excelFactory.createDocument();

        excel.preview();

    }

}

**Document.java :**

public interface Document {

    void preview();

}

**WordDocument.java:**

public class WordDocument implements Document {

    public void preview() {

        System.out.println("Word File:");

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

        System.out.println("Format: .docx\n");

    }

}

**PdfDocument.java:**

public class PdfDocument implements Document {

    public void preview() {

        System.out.println("PDF File:");

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

        System.out.println("Format: .pdf\n");

    }

}

**ExcelDocument.java:**

public class ExcelDocument implements Document {

    public void preview() {

        System.out.println("Excel File:");

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

        System.out.println("Format: .xlsx\n");

    }

}

**DocumentFactroy.java:**

public abstract class DocumentFactory {

    public abstract Document createDocument();}

**WordFactroy.java:**

public class WordFactory extends DocumentFactory {

    public Document createDocument() {

        return new WordDocument();

    }

}

**PdfFactroy.java:**

public class PdfFactory extends DocumentFactory {

    public Document createDocument() {

        return new PdfDocument();

    }

}

**ExcelFactroy.java:**

public class ExcelFactory extends DocumentFactory {

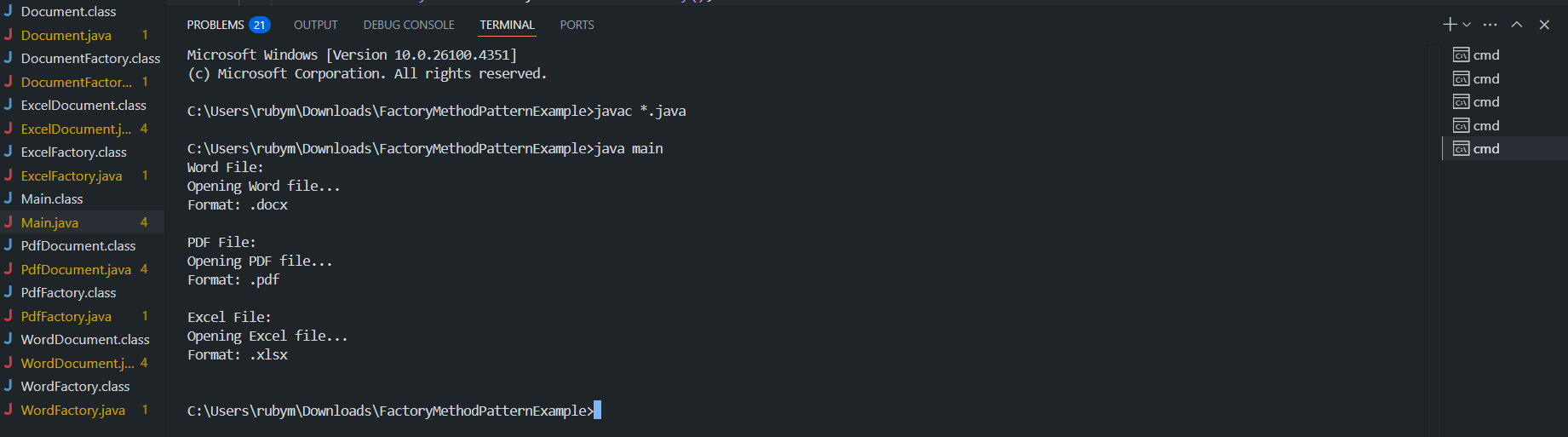
    public Document createDocument() {

        return new ExcelDocument();

    }

}

**Output :**



**Explanation:**

* This implementation follows the Factory Method Pattern to create and preview different types of documents like Word, PDF, and Excel.
* A common interface Document defines the preview() method, which each document type implements to show custom output.
* The abstract class DocumentFactory defines the factory method createDocument(), and concrete factories like WordFactory, PdfFactory, and ExcelFactory override it to return specific document instances.
* The Main class demonstrates how each document type is created using its respective factory without needing to know the underlying class structure.
* This approach promotes loose coupling, code reusability, and easy scalability when adding new document types in the future.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Module -02 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.

**SOLUTION :**

1. **Big O Notation and its Use :**

Big O notation is used to analyze the time and space complexity of algorithms. It helps determine how performance scales with input size and is crucial for comparing algorithms in large systems like e-commerce platforms.

1. **Best, Average, and Worst-Case Scenarios :**

| **Case** | **Linear Search** | **Binary Search** |
| --- | --- | --- |
| **Best Case** | **O(1) – First element match** | **O(1) – Middle element match** |
| **Average Case** | **O(n) – Element in middle** | **O(log n) – Halves search space** |
| **Worst Case** | **O(n) – Element not found** | **O(log n) – Element not found** |

1. **Setup – Defining Product Class :**

**Product.java:**

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 void display() {

System.out.println("Product ID: " + productId);

System.out.println("Product Name: " + productName);

System.out.println("Category: " + category);

}

}

**main.java:**

import java.util.Arrays;

import java.util.Comparator;

import java.util.Scanner;

public class main {

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

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

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

                return i;

            }

        }

        return -1;

    }

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

        int left = 0, right = products.length - 1;

        while (left <= right) {

            int mid = (left + right) / 2;

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

            if (cmp == 0)

                return mid;

            else if (cmp < 0)

                left = mid + 1;

            else

                right = mid - 1;

        }

        return -1;

    }

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        Product[] products = {

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

                new Product(102, "Shampoo", "Beauty"),

                new Product(103, "Shoes", "Footwear"),

                new Product(104, "Watch", "Accessories"),

                new Product(105, "Bag", "Travel")

        };

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

        String targetName = sc.nextLine();

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

        System.out.println("Opening product list...");

        System.out.println("Searching using linear method...");

        int linearResult = linear Search(products, targetName);

        if (linearResult != -1) {

            System.out.println("Product found using Linear Search:");

            products[linearResult].display();

        } else {

            System.out.println("Product not found using Linear Search.");

        }

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

        System.out.println("Sorting product list...");

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

        System.out.println("Searching using binary method...");

        int binaryResult = binarySearch(products, targetName);

        if (binaryResult != -1) {

            System.out.println("Product found using Binary Search:");

            products[binaryResult].display();

        } else {

            System.out.println("Product not found using Binary Search.");

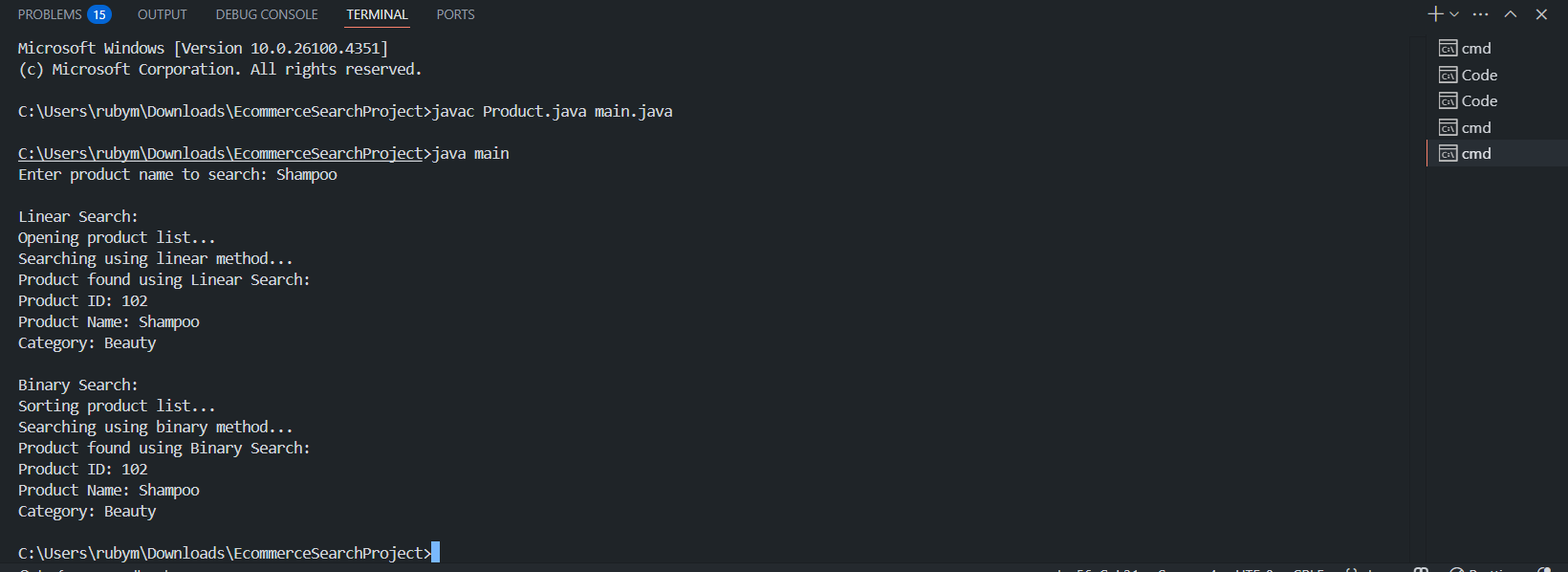
        }

        sc.close();

    }

}

**4. OUTPUT :**

****

**5. Analysis :**

| **Operation** | **Linear Search** | **Binary Search** |
| --- | --- | --- |
| **Time Complexity** | **O(n)** | **O(log n)** |
| **Sorting Needed?** | **No** | **Yes** |
| **Space Complexity** | **O(1)** | **O(1)** |

* **Binary search** is much faster for larger sorted arrays.
* **Linear search** is simple and does not require pre-sorting, but becomes slow with large datasets.
* For real-world e-commerce systems, **binary search or advanced indexing** is preferred for performance and scalability.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Exercise 7 : Financial Forecasting**

**Scenario :** You are building a forecasting module to assist financial planners in predicting the future value of an investment based on historical performance. The aim is to utilize recursive logic for this dynamic calculation.

**SOLUTION :**

1. **Recursive Algorithm :**

* Recursion is a process where a function calls itself to solve smaller instances of the same problem. In financial forecasting, this helps model investment growth year by year. Each year’s value depends on the previous, making recursion a natural fit.
* The recursive method simplifies the logic and aligns with the compound growth formula:

FutureValue(n)=FutureValue(n−1)×(1+Rate/100)

1. **Method Design :**

* Recursively calculates the compounded future value.
* Uses base case years == 0 to stop.

public static double predictFutureValue(double principal, double rate, int years)

* Adds an inflation adjustment layer to the growth calculation.
* Provides a realistic view of the value's future purchasing power.

public static double predictAdjustedForInflation(double principal, double rate,double inflationRate, int years)

1. **Recursive Implementation:**

**main.java:**

import java.util.Scanner;

public class main {

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        System.out.println("   Smart Financial Forecasting");

        System.out.print("Enter Initial Investment (Rs.): ");

        double base = sc.nextDouble();

        System.out.print("Enter Annual Growth Rate (%): ");

        double growthRate = sc.nextDouble();

        System.out.print("Enter Forecast Duration (years): ");

        int years = sc.nextInt();

        System.out.print("Enter Expected Inflation Rate (%): ");

        double inflationRate = sc.nextDouble();

        System.out.println("\nProcessing smart forecast...\n");

        double rawValue = ForecastCalculator.predictFutureValue(base, growthRate, years);

        double adjustedValue = ForecastCalculator.predictAdjustedForInflation(base, growthRate, inflationRate, years);

        System.out.printf("Projected Value after %d years: Rs%.2f (without inflation)\n", years, rawValue);

        System.out.printf("Adjusted Value after %d years: Rs%.2f (with %.2f%% inflation)\n", years, adjustedValue,

                inflationRate);

        sc.close();

    }

}

**ForecastCalculator .java(using Recursion)**

public class ForecastCalculator {

    // Recursive method to calculate future value

    public static double predictFutureValue(double principal, double rate, int years) {

        if (years == 0)

            return principal;

        return predictFutureValue(principal, rate, years - 1) \* (1 + rate / 100);

    }

    // Recursive method with inflation adjustment

    public static double predictAdjustedForInflation(double principal, double rate, double inflationRate, int years) {

        if (years == 0)

            return principal;

        double value = predictAdjustedForInflation(principal, rate, inflationRate, years - 1);

        value \*= (1 + rate / 100);

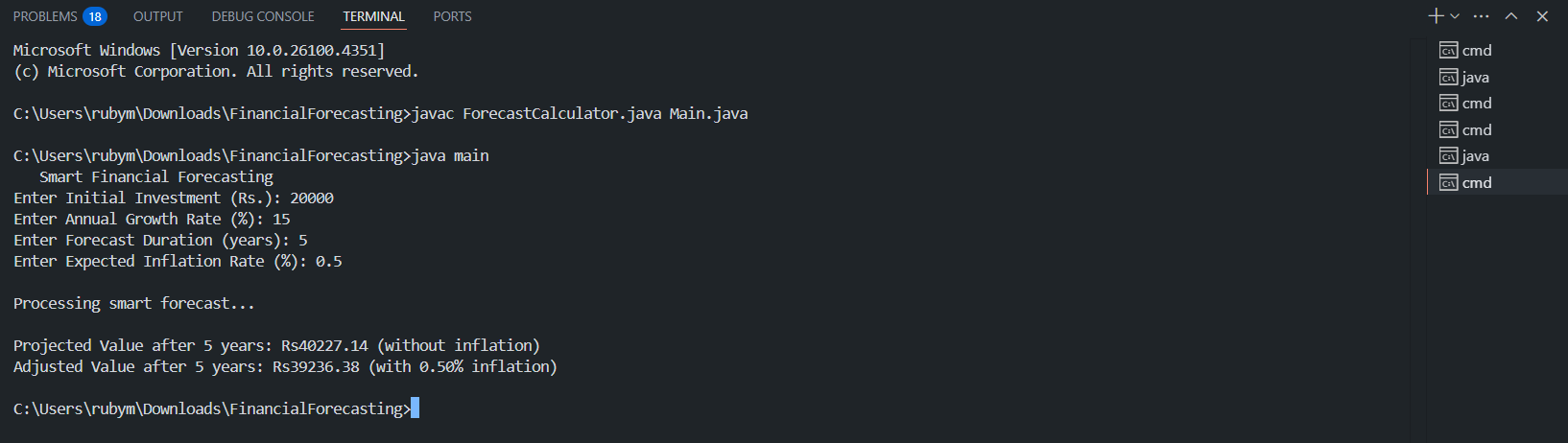
        value /= (1 + inflationRate / 100);

        return value;

    }

}

1. **OUTPUT :**



1. **Analysis:**

* **Time Complexity**:  
  The recursive algorithm has a time complexity of **O(n)**, where n is the number of years. Each recursive call processes one year, making the depth of recursion linear.
* **Optimization**:  
  To avoid stack overflow and improve efficiency for large inputs, the recursive logic can be **converted to an iterative approach,** which uses a simple loop and runs in constant space.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**