**Design Patterns and Principles**

**SingletonPatternExample**

Code:

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);

}

}

class SingletonTest {

public static void main(String[] args) {

Logger logger1 = Logger.getInstance();

Logger logger2 = Logger.getInstance();

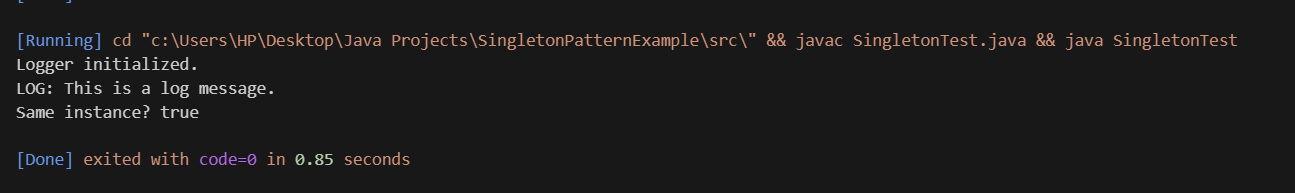
logger1.log("This is a log message.");

System.out.println("Same instance? " + (logger1 == logger2));

}

}

Output:



**FactoryMethodPattern**

Code:

interface Document {

void open();

}

class WordDocument implements Document {

public void open() {

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

}

}

class PdfDocument implements Document {

public void open() {

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

}

}

class ExcelDocument implements Document {

public void open() {

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

}

}

abstract class DocumentFactory {

public abstract Document createDocument();

}

class WordFactory extends DocumentFactory {

public Document createDocument() {

return new WordDocument();

}

}

class PdfFactory extends DocumentFactory {

public Document createDocument() {

return new PdfDocument();

}

}

class ExcelFactory extends DocumentFactory {

public Document createDocument() {

return new ExcelDocument();

}

}

class FactoryTest {

public static void main(String[] args) {

DocumentFactory factory = new PdfFactory();

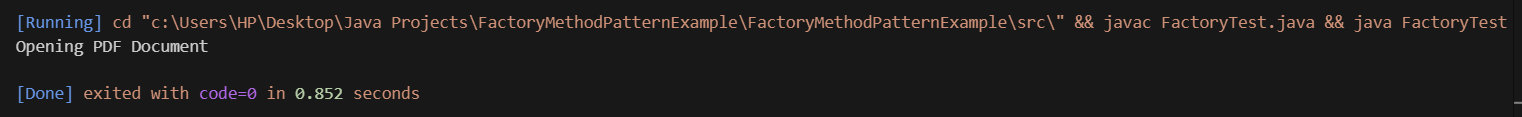
Document doc = factory.createDocument();

doc.open();

}

}

Output:



**Data Structures and Algorithms**

**Exercise2**

**Asymptotic Notation**

Big O Notation describes the upper bound of an algorithm's time or space complexity. It helps in comparing how algorithms scale with input size (n).

For searching:

Linear Search-

Worst case: O(n) – item at end or not found

Best case: O(1) – item at start

Binary Search-

Works only on sorted arrays

Worst/Average case: O(log n)

Best case: O(1) – item in middle

Code:

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 "ID: " + productId + ", Name: " + productName + ", Category: " + category;

    }

}

import java.util.Arrays;

import java.util.Comparator;

public class ProductSearch {

    // Linear Search by productId

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

        for (Product p : products) {

            if (p.productId == targetId) {

                return p;

            }

        }

        return null;

    }

    // Binary Search by productId

    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;

            if (products[mid].productId == targetId) {

                return products[mid];

            } else if (products[mid].productId < targetId) {

                left = mid + 1;

            } else {

                right = mid - 1;

            }

        }

        return null;

    }

    public static void main(String[] args) {

        Product[] productList = {

            new Product(104, "Laptop", "Electronics"),

            new Product(101, "Shirt", "Clothing"),

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

            new Product(102, "Phone", "Electronics")};

        // Linear Search

        int targetId = 103;

        Product found = linearSearch(productList, targetId);

        System.out.println("Linear Search Result: " + (found != null ? found : "Product not found"));

        // Sort products for binary search by productId

        Arrays.sort(productList, Comparator.comparingInt(p -> p.productId));

        // Binary Search

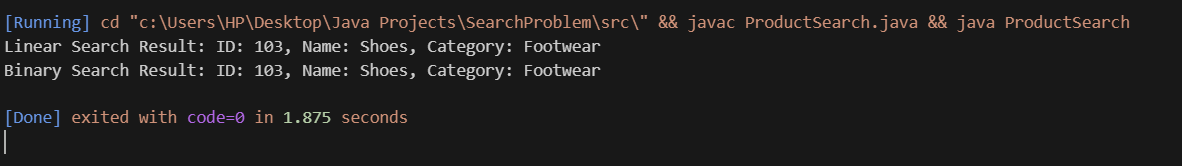
        found = binarySearch(productList, targetId);

        System.out.println("Binary Search Result: " + (found != null ? found : "Product not found"));

    }

}

Output:



Analysis:

Linear Search-

Time Complexity: O(n)

Description: Scans each element in the array until the target is found or the end is reached.

Use Case: Best suited for small datasets or when the data is not sorted.

Binary Search-

Time Complexity: O(log n)

Description: Continuously divides the sorted array in half to locate the target.

Use Case: Ideal for large datasets that are already sorted.

Recommendation:

-Use binary search for optimized performance only if the product list is sorted by productId.

-For dynamic or small lists, linear search is simpler and works directly.

**Exercise 7**

Explain the concept of recursion and how it can simplify certain problems.

Ans: Recursion is a method where a function calls itself to solve smaller instances of the same problem.

Code:

public class FinancialForecast {

    // Recursive method to calculate future value

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

        if (years == 0) {

            return currentValue;

        }

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

    }

    public static void main(String[] args) {

        double currentValue = 10000;  // Starting investment

        double growthRate = 0.08;     // 8% annual growth

        int years = 5;

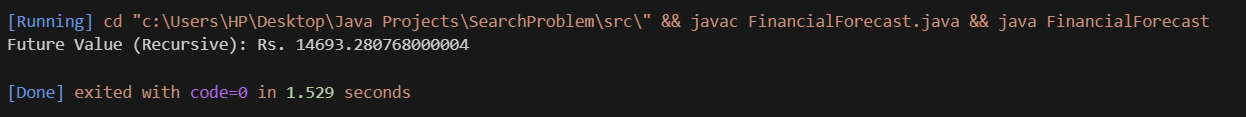
        double resultRecursive = forecastFutureValue(currentValue, growthRate, years);

        System.out.println("Future Value (Recursive): Rs. " + resultRecursive);

    }

}

Output:



Analysis:

Time Complexity-

Recursive version: O(n) — one recursive call per year.

Drawbacks of Recursion-

Stack overflow for large n (e.g., 10,000+ years).

Repeated calls can be inefficient if growth isn't constant

Optimization-

Prefer iteration over recursion for simple linear recurrence.