**Exercise 1: Implementing the Singleton Pattern**

**Code:**

class Logger {

    private static Logger instance;

    private Logger() {

    }

    public static Logger getInstance() {

        if (instance == null) {

            instance = new Logger();

        }

        return instance;

    }

    public void log(String message) {

        System.out.println("Log: " + message);

    }

}

public class SingletonPatternExample {

    public static void main(String[] args) {

        Logger logger1 = Logger.getInstance();

        Logger logger2 = Logger.getInstance();

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

        if (logger1 == logger2) {

            System.out.println("Both logger1 and logger2 refer to the same instance.");

        } else {

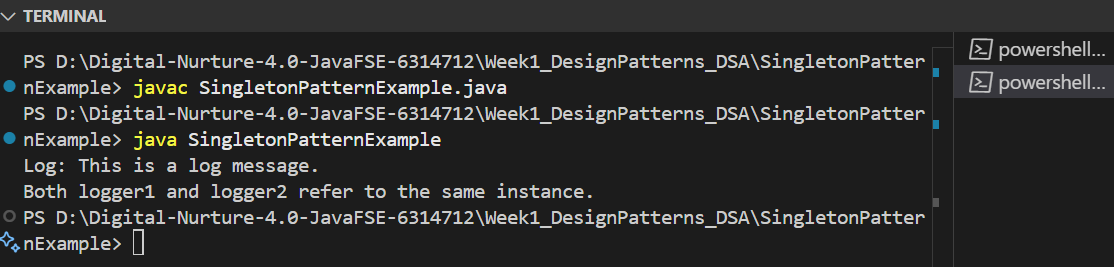
            System.out.println("Different instances exist!");

        }

    }

}

**Output:**

****

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

**Code:**

package Week1\_DesignPatterns\_DSA.FactoryMethodPatternExample;

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 {

    abstract Document createDocument();

}

class WordDocumentFactory extends DocumentFactory {

    Document createDocument() {

        return new WordDocument();

    }

}

class PdfDocumentFactory extends DocumentFactory {

    Document createDocument() {

        return new PdfDocument();

    }

}

class ExcelDocumentFactory extends DocumentFactory {

    Document createDocument() {

        return new ExcelDocument();

    }

}

public class FactoryMethodPatternExample {

    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();

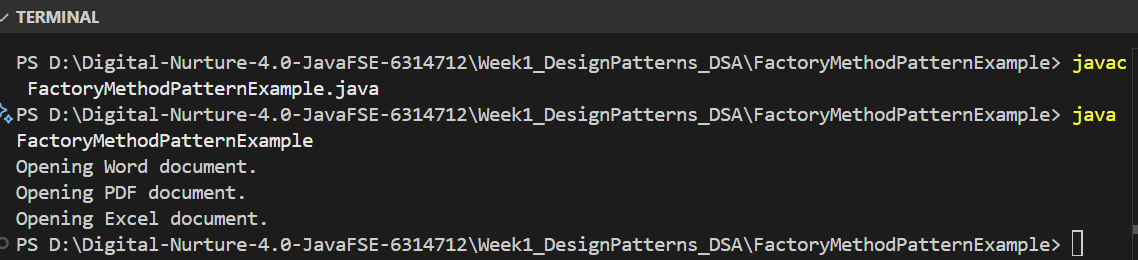
        Document excelDoc = excelFactory.createDocument();

        excelDoc.open();

    }

}

**Output:**



**Exercise 3: E-commerce Platform Search Function**

**Code:**

package Week1\_DesignPatterns\_DSA.EcommerceSearch;

/\*Big O notation describes the upper bound of an algorithm's running time as input size grows.

It helps analyze and compare the efficiency of algorithms.

Best case: minimum time for some input.

Average case: expected time over all inputs.

Worst case: maximum time for any input.

For search: Linear search O(1) best, O(n) average/worst. Binary search O(1) best, O(log n) average/worst (requires sorted data).

\*/

class Product {

    int productId;

    String productName;

    String category;

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

        this.productId = productId;

        this.productName = productName;

        this.category = category;

    }

}

class SearchUtils {

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

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

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

                return i;

            }

        }

        return -1;

    }

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

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

        while (left <= right) {

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

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

                return mid;

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

                left = mid + 1;

            } else {

                right = mid - 1;

            }

        }

        return -1;

    }

}

public class EcommerceSearch {

    public static void main(String[] args) {

        Product[] products = {

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

            new Product(205, "Shoes", "Footwear"),

            new Product(150, "Book", "Stationery"),

            new Product(300, "Phone", "Electronics"),

            new Product(120, "T-shirt", "Apparel")

        };

        int idx = SearchUtils.linearSearch(products, 150);

        if (idx != -1) {

            System.out.println("Linear Search: Found " + products[idx].productName);

        } else {

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

        }

        java.util.Arrays.sort(products, (a, b) -> a.productId - b.productId);

        idx = SearchUtils.binarySearch(products, 150);

        if (idx != -1) {

            System.out.println("Binary Search: Found " + products[idx].productName);

        } else {

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

        }

        // Analysis:

        // Linear search: O(n), checks each product one by one.

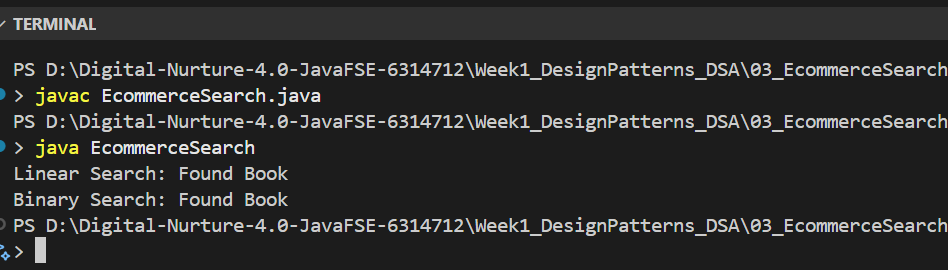
        // Binary search: O(log n), divides search space in half each time, but needs sorted data.

        // For large product lists, binary search is much faster and more suitable for an e-commerce platform.

    }

}

**Output:**

****

**Exercise 4: Financial Forecasting**

**Code:**

package Week1\_DesignPatterns\_DSA.FinancialForecasting;

/\*

Recursion is a programming technique where a method calls itself to solve a problem by breaking it down into smaller subproblems.

It can simplify problems that have a natural recursive structure, such as computing factorial, Fibonacci numbers, or traversing trees.

\*/

// Recursive future value calculation: FV = PV \* (1 + rate)^years

class FinancialForecasting {

    // Recursive method to calculate future value

    static double futureValue(double presentValue, double rate, int years) {

        if (years == 0) {

            return presentValue;

        }

        return futureValue(presentValue, rate, years - 1) \* (1 + rate);

    }

    public static void main(String[] args) {

        double presentValue = 1000.0;

        double rate = 0.08; // 8% annual growth

        int years = 5;

        double fv = futureValue(presentValue, rate, years);

        System.out.printf("Future value after %d years: %.2f\n", years, fv);

    }

}

/\*

Time complexity: O(n), where n is the number of years, since the function calls itself once per year.

To optimize and avoid excessive computation (especially if overlapping subproblems exist), use memoization or an iterative approach.

For this simple case, iteration is more efficient:

    static double futureValueIterative(double presentValue, double rate, int years) {

        double result = presentValue;

        for (int i = 0; i < years; i++) {

            result \*= (1 + rate);

        }

        return result;

    }

\*/

**Output:**

