**1.Implementing the Singleton Pattern**

**SOLUTION**

**Logger.java**

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);

    }

}

**Main.java**

public class Main {

    public static void main(String[] args) {

        Logger logger1 = Logger.getInstance();

        Logger logger2 = Logger.getInstance();

        logger1.log("First message");

        logger2.log("Second message");

        if (logger1 == logger2) {

            System.out.println("Singleton works! Both loggers are the same instance.");

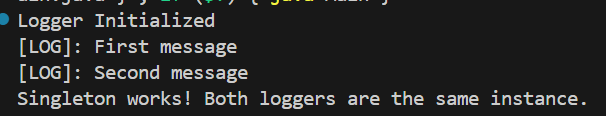
        } else {

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

        }

    }

}

**Output**

**2. Implementing the Factory Method Pattern**

**SOLUTION**

public interface Document {

    void open();

}

public class WordDocument implements Document {

    public void open() {

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

    }

}

public class PdfDocument implements Document {

    public void open() {

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

    }

}

public class ExcelDocument implements Document {

    public void open() {

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

    }

}

public abstract class DocumentFactory {

    public abstract Document createDocument();

}

public class WordDocumentFactory extends DocumentFactory {

    public Document createDocument() {

        return new WordDocument();

    }

}

public class PdfDocumentFactory extends DocumentFactory {

    public Document createDocument() {

        return new PdfDocument();

    }

}

public class ExcelDocumentFactory extends DocumentFactory {

    public Document createDocument() {

        return new ExcelDocument();

    }

}

**FactoryMethodTest.java**

public class FactoryMethodTest {

    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();

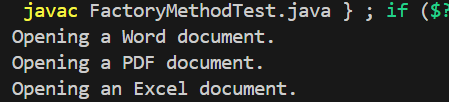
        DocumentFactory excelFactory = new ExcelDocumentFactory();

        Document excelDoc = excelFactory.createDocument();

        excelDoc.open();

    }

}

**Output**

1. **E-commerce Platform Search Function**

**SOLUTION**

**Big O Notation**

Big O notation describes the upper bound of an algorithm’s running time as the input size grows. It helps estimate performance and scalability of algorithms.  
  
  
**Best, Average, Worst Case**

Best: Minimum time. O(1)

Average: Expected time over many inputs. O(log n) for binary search

Worst: Maximum time. O(log n) for binary search

**ProductSearchExample.java**

import java.util.Arrays;

import java.util.Comparator;

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;

    }

    @Override

    public String toString() {

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

    }

}

class Search {

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

        for (Product product : products) {

            if (product.productName.equalsIgnoreCase(name)) {

                return product;

            }

        }

        return null;

    }

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

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

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

        while (left <= right) {

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

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

            if (cmp == 0) return products[mid];

            else if (cmp < 0) left = mid + 1;

            else right = mid - 1;

        }

        return null;

    }

}

public class ProductSearchExample {

    public static void main(String[] args) {

        Product[] products = {

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

            new Product(102, "Chair", "Furniture"),

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

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

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

        };

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

        Product result1 = Search.linearSearch(products, "Shoes");

        System.out.println(result1 != null ? result1 : "Product not found!");

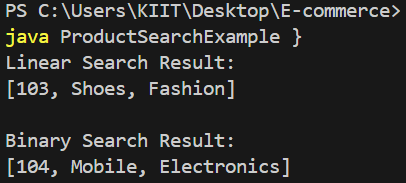
        System.out.println("\nBinary Search Result:");

        Product result2 = Search.binarySearch(products, "Mobile");

        System.out.println(result2 != null ? result2 : "Product not found!");

    }

}

**Output**

For an e-commerce platform, binary search is preferable if the product list is sorted, because:

1.Much faster on large datasets

2.Can scale well as product catalog grows

**4. Financial Forecasting**

**Solution**Recursive Algorithms:-  
Recursion is a technique where a function calls itself to solve smaller instances of the same problem.  
In financial forecasting, future value can often be modeled using recurrence relations.  
  
**FinancialForecast.java**

public class FinancialForecast {

    public static double calculateFutureValue(double baseValue, double growthRate, int years) {

        if (years == 0) {

            return baseValue;

        }

        return calculateFutureValue(baseValue, growthRate, years - 1) \* (1 + growthRate);

    }

    public static void main(String[] args) {

        double baseValue = 1000.0;

        double growthRate = 0.05;

        int years = 5;

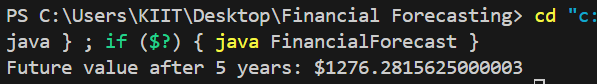
        double futureValue = calculateFutureValue(baseValue, growthRate, years);

        System.out.println("Future value after " + years + " years: $" + futureValue);

    }

}

**Output**



**Time Complexity:**

T(n) = T(n-1) + O(1)= Time complexity: O(n)

Every year involves one recursive call until n = 0

**Optimization Techniques:-**

1. Memoization (caching intermediate results)
2. Convert to Iterative Approach