**WEEK-1**

**Exercise-1: Implementing the Singleton Pattern**

**Logger.java**

package singleton;

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);

    }

}

**TestSingleton.java**

package singleton;

public class TestSingleton {

    public static void main(String[] args) {

        Logger logger1 = Logger.getInstance();

        Logger logger2 = Logger.getInstance();

        logger1.log("This is the first log message.");

        logger2.log("This is the second log message.");

        if (logger1 == logger2) {

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

        } else {

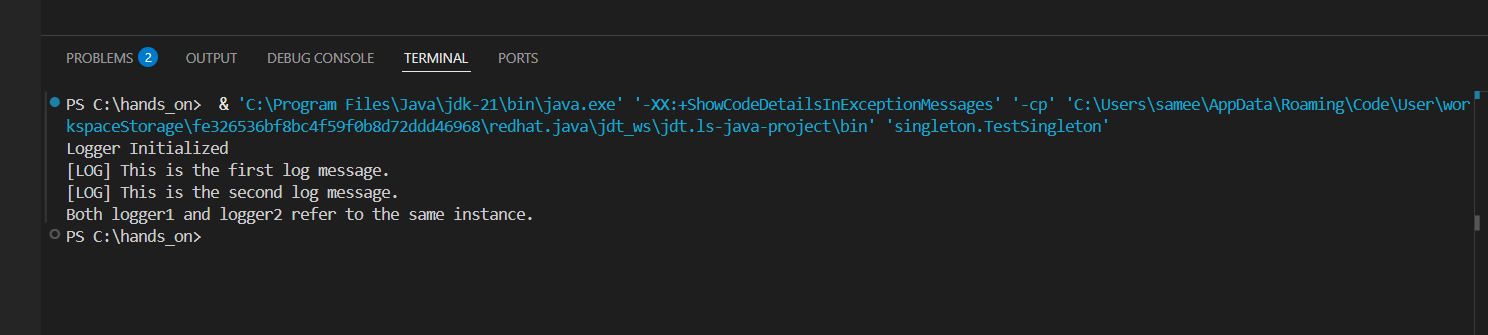
            System.out.println("Different instances exist!");

        }

    }

}

**OUTPUT:**

****

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

**FactoryMethodSingleFile.java**

public class FactoryMethodSingleFile {

    interface Document {

        void open();  }

    static class WordDocument implements Document {

        public void open() {

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

        }

    }

    static class PdfDocument implements Document {

        public void open() {

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

        }

    }

    static class ExcelDocument implements Document {

        public void open() {

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

        }   }

    static abstract class DocumentFactory {

        public abstract Document createDocument();

    }

    static class WordDocumentFactory extends DocumentFactory {

        public Document createDocument() {

            return new WordDocument();

        }

    }

    static class PdfDocumentFactory extends DocumentFactory {

        public Document createDocument() {

            return new PdfDocument();

        }

    }

    static class ExcelDocumentFactory extends DocumentFactory {

        public Document createDocument() {

            return new ExcelDocument();

        } }

    public static void main(String[] args) {

        DocumentFactory wordFactory = new WordDocumentFactory();

        Document word = wordFactory.createDocument();

        word.open();

        DocumentFactory pdfFactory = new PdfDocumentFactory();

        Document pdf = pdfFactory.createDocument();

        pdf.open();

        DocumentFactory excelFactory = new ExcelDocumentFactory();

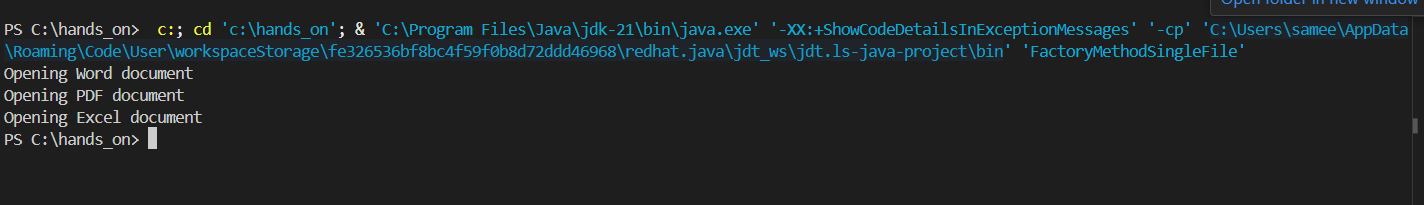
        Document excel = excelFactory.createDocument();

        excel.open();

    }

}

**OUTPUT:**



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

**ECommerceSearch.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;

}

public String toString() {

return productId + " - " + productName + " (" + category + ")";

}

}

public class ECommerceSearch {

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

for (Product p : products) {

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

return p;

}

}

return null;

}

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

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 static void main(String[] args) {

Product[] products = {

new Product(1, "Laptop", "Electronics"),

new Product(2, "Shoes", "Fashion"),

new Product(3, "Book", "Stationery"),

new Product(4, "Phone", "Electronics"),

new Product(5, "Backpack", "Accessories")

};

Product[] sortedProducts = Arrays.copyOf(products, products.length);

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

String searchName = "Phone";

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

Product result1 = linearSearch(products, searchName);

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

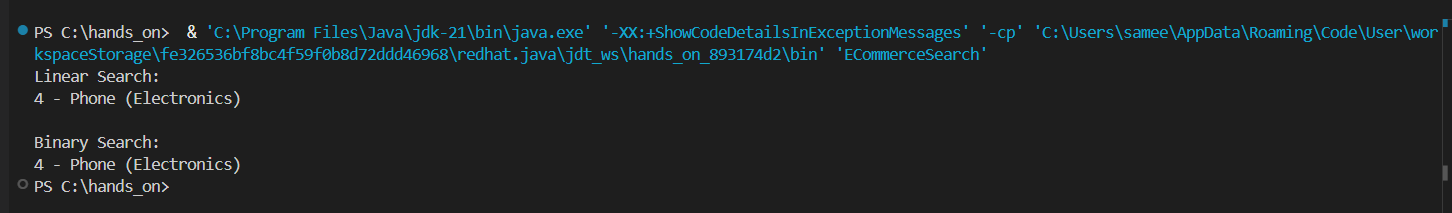
System.out.println("\nBinary Search:");

Product result2 = binarySearch(sortedProducts, searchName);

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

}}

**OUTPUT:**

****

**Time Complexity Analysis:**

Linear Search has a time complexity of O(n) and does not require the data to be sorted. It is simple to implement and works well for small or unsorted datasets, making it suitable for quick lookups where sorting isn't feasible.

Binary Search, in contrast, has a time complexity of O(log n) but requires the data to be sorted. It is much faster and more efficient for large datasets, especially when the product list can be maintained in a sorted order.

**Conclusion:**  
Use Linear Search for smaller or unsorted collections, and choose Binary Search when working with large, sorted datasets to ensure optimal search performance in an e-commerce platform.

**Exercise 4: Financial Forecasting**

**FinancialForecast.java**

public class FinancialForecast {

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

if (years == 0) {

return currentValue;

}

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

}

public static void main(String[] args) {

double currentValue = 10000;

double annualGrowthRate = 0.08;

int forecastYears = 5;

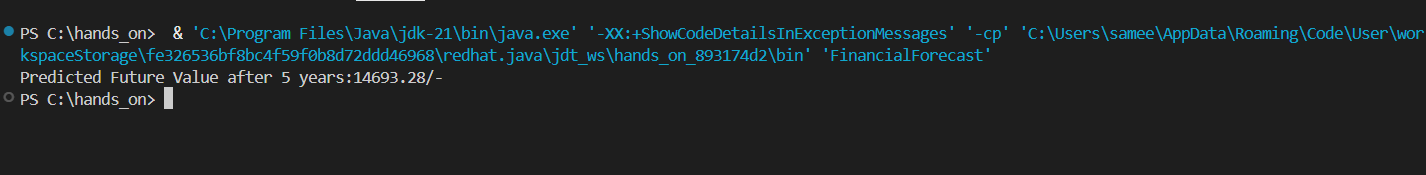
double futureValue = forecastFutureValue(currentValue, annualGrowthRate, forecastYears);

System.out.printf("Predicted Future Value after %d years: ₹%.2f\n", forecastYears, futureValue);

}

}

**OUTPUT:**



**Time Complexity:**

* The recursive function runs once per year:  
  Time Complexity = O(n) where n is the number of years.
* Space complexity is also O(n) due to the call stack.

**Optimization:** For a simple calculation like compound growth, **recursion isn't the most efficient** due to repeated calls, **iterative or mathematical exponentiation** is better.