Cognizant Week-1\_HandsOn

**Exercise 1: Implementing the Singleton Pattern**

**Code:**

public class main {

    public static void main(String[] args) {

        Logger logger1 = Logger.getInstance();

        Logger logger2 = Logger.getInstance();

        if (logger1 == logger2) {

            System.out.println("Verified Both logger instances are the same");

        } else {

            System.out.println("verification failed both instance are different");

        }

    }

}

public class Logger {

    private static Logger instance;

    private Logger() {

        //it defines how many time logger is initialized

        System.out.println("Logger initialized.");

    }

    public static Logger getInstance() {

        if (instance == null) {

            instance = new Logger();

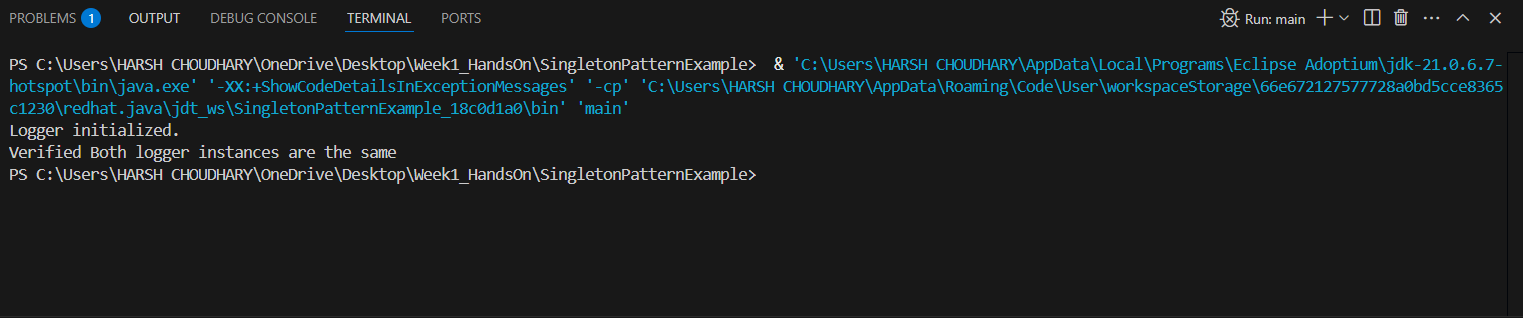
        }

        return instance;

    }

}

**Output:**



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

**Code:**

public class main {

    public static void main(String[] args) {

        DocumentFactory word = new WordDocumentFactory();

        Document obj1 = word.createDocument();

        obj1.open();

        DocumentFactory pdf = new pdfDocumentFactory();

        Document obj2 = pdf.createDocument();

        obj2.open();

        DocumentFactory excel = new excelDocumentFactory();

        Document obj3 = excel.createDocument();

        obj3.open();

    }

}

public interface Document {

    void open();

}

public class WordDocument implements Document {

    public void open() {

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

    }

}

public class pdfDocument implements Document{

    public void open(){

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

    }

}

public class excelDocument implements Document{

    public void  open(){

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

    }

}

public abstract class DocumentFactory {

    public abstract Document createDocument();

}

public class excelDocumentFactory extends DocumentFactory {

    @Override

    public Document createDocument() {

        return  new excelDocument();

    }

}

public class pdfDocumentFactory extends DocumentFactory{

    @Override

    public Document createDocument() {

        return  new pdfDocument();

    }

}

public class WordDocumentFactory extends DocumentFactory{

    @Override

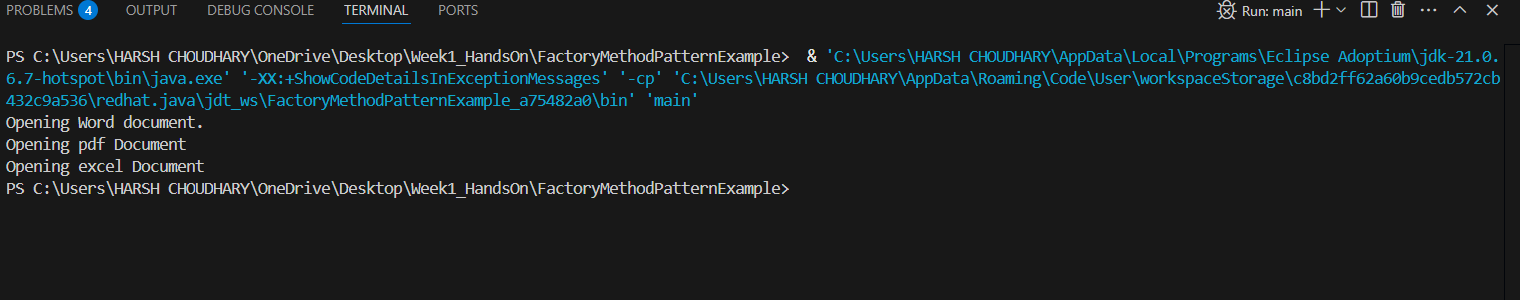
    public Document createDocument() {

           return new WordDocument();

    }

}

**Output:**



**Exercise 2: E-commerce Platform Search Function**

**Theory:**

**Big O Notation:**

Big O Notation describes the upper bound of an algorithm’s running time or space requirement in terms of input size. It helps in understanding the scalability of an algorithm.

* O(1) → Constant time (fastest)
* O(n) → Linear time
* O(log n) → Logarithmic time
* O(n²) → Quadratic time (slow for large n)

**1. Best Case**:

* This scenario represents the input that allows the algorithm to run with the fewest operations.
* For example, in a linear search, the best case is when the target element is the first element in the list.
* The algorithm finds the element immediately, requiring only one comparison.
* This is often represented by O(1), indicating constant time complexity.

**2. Worst Case**:

* This scenario represents the input that causes the algorithm to take the longest time or use the most resources.
* For a linear search, the worst case is when the target element is the last element in the list or not present at all.
* The algorithm has to compare the target with every element in the list.
* This is often represented by O(n), indicating linear time complexity, where 'n' is the input size.

**3. Average Case**:

* This scenario represents the typical performance of the algorithm when considering all possible inputs.
* It's determined by analyzing the average number of operations across all inputs.
* For example, in a linear search, the average case is when the target element is roughly in the middle of the list.
* This provides a more realistic view of the algorithm's performance in everyday use.
* Calculating the average case can be complex and often involves probability distributions of inputs.

**Code:**

public class Main {

    public static void main(String[] args) {

        Product[] products = {

            new Product(511, "Slipper", "Footwear"),

            new Product(15, "Tv", "Electronics"),

            new Product(110, "Mobile", "Accessories"),

            new Product(188, "Speaker", "Electronics")

        };

        int searchId = 15;

        Product foundLinear = SearchEngine.linearSearch(products, searchId);

       if(foundLinear != null){

          System.out.println("Linear Search Output: "+ foundLinear);

       }

       else{

          System.out.println("Linear Search Output: "+ "Not Found");

       }

        SearchEngine.sortProducts(products);

        Product foundBinary = SearchEngine.binarySearch(products, searchId);

        if(foundBinary != null){

            System.out.println("Binary Search Output: " + foundBinary);

        }

        else{

            System.out.println("Binary Search Output: " + "Not Found");

        }

    }

}

public class Product {

    private int productId;

    private String productName;

    private String category;

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

        this.productId = productId;

        this.productName = productName;

        this.category = category;

    }

    public int getProductId() {

        return productId;

    }

    public String getProductName() {

        return productName;

    }

    public String getCategory() {

        return category;

    }

    @Override

    public String toString() {

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

    }

}

public class SearchEngine {

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

        for (Product product : products) {

            if (product.getProductId() == productId) {

                return product;

            }

        }

        return null;

    }

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

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

        while (left <= right) {

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

            int midId = products[mid].getProductId();

            if (midId == productId) {

                return products[mid];

            } else if (midId < productId) {

                left = mid + 1;

            } else {

                right = mid - 1;

            }

        }

        return null;

    }

    public static void sortProducts(Product[] products) {

    int n = products.length;

    for (int i = 0; i < n - 1; i++) {

        for (int j = 0; j < n - i - 1; j++) {

            if (products[j].getProductId() > products[j + 1].getProductId()) {

                Product temp = products[j];

                products[j] = products[j + 1];

                products[j + 1] = temp;

            }

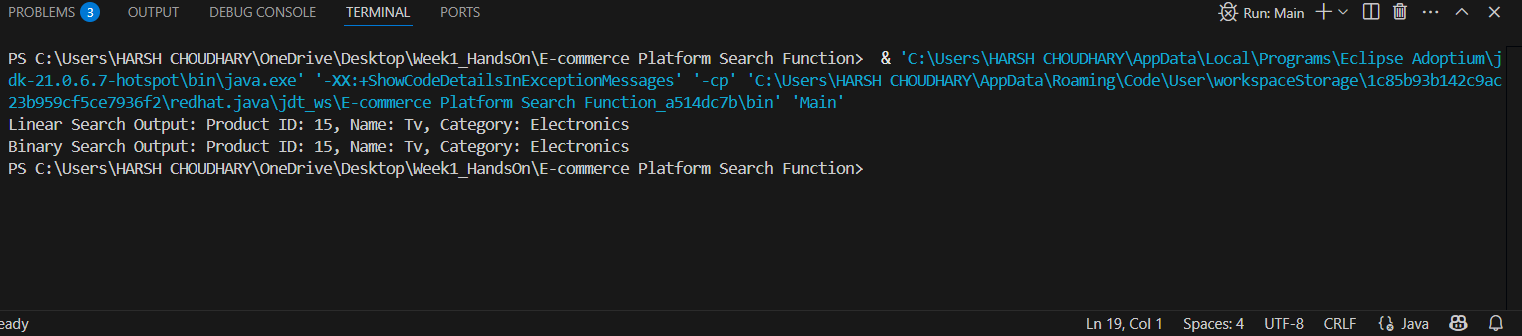
        }

    }

}

}

**Output:**



**Analysis**:

Linear search is simple to implement but less efficient for large datasets, while binary search is more complex but offers significant performance gains when working with sorted data.

**Conclusion**:

* **Linear search**: is suitable for small datasets or when the data is not sorted, and its simplicity makes it a good choice for quick implementations.
* **Binary search**: is the superior choice for large, sorted datasets where speed is critical. The logarithmic time complexity makes it significantly faster than linear search for large inputs. However, it requires the data to be sorted beforehand, which adds an extra step.

**Recommendation:**

Use Binary Search when your product list is sorted by product ID — it provides fast performance even with thousands of entries

**Exercise 7: Financial Forecasting**

**Theory:**

**Recursion:**

Recursion is a technique where a function calls itself to solve a smaller instance of the same problem. Every recursive function must have:

* A base case (stopping condition)
* A recursive step (where the function calls itself)

**Code:**

public class main {

    public static void main(String[] args) {

        int presentValue = 500;

        int growthRate = 2;

        int years = 4;

        int futureValue = FinanceForcasting.forecastValue(presentValue, growthRate, years);

        System.out.printf("Future Value after %d years: %d", years, futureValue);

    }

}

public class FinanceForcasting {

    public static int forecastValue(int presentValue, int rate, int years) {

        if (years == 0) {

            return presentValue;

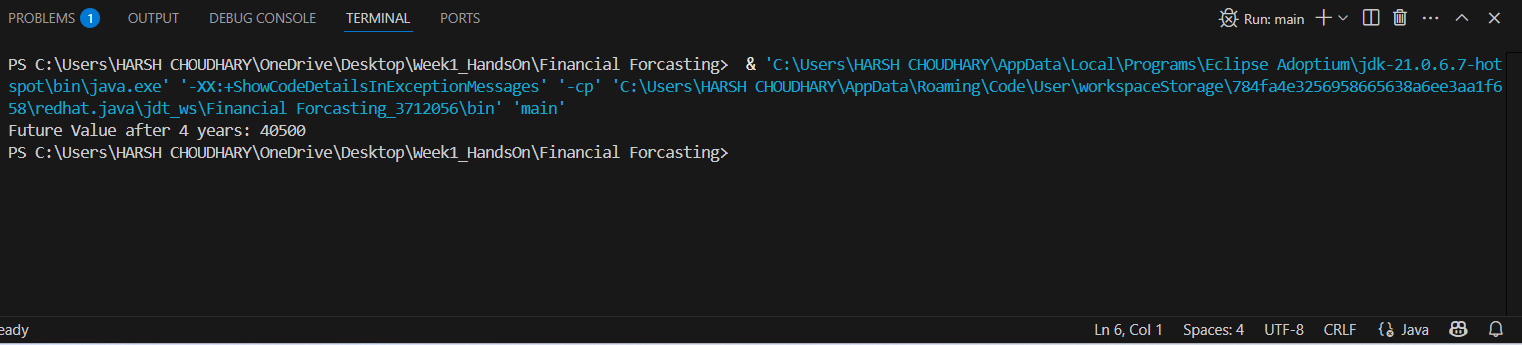
        }

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

    }

}

**Output:**



**Analysis:**

**** Time Complexity: O(n)

 Space Complexity: O(n) (due to call stack)

**Optimization :**

1.Use Iteration Instead (Tail Recursion or Loop)

2.Use Memoization (for more complex models)