COGNIZANT DN 4.0 – JAVA FSE

Week 1

**1) Implementing the Singleton Pattern**

Folder “SingletonPatternExample” in <https://github.com/Kunta-tech/Deepskilling4.0-JavaFSE/tree/master/week_1/SingletonPatternExample>

**Code**:

Logger.java

package SingletonPatternExample;

class Logger{

  private static Logger instance = new Logger();

  private Logger(){}

  public static Logger getInstance(){

    return instance;

  }

}

LoggerTest.java

package SingletonPatternExample;

public class LoggerTest {

    public static void main(String[] args) {

        Logger logger1 = Logger.getInstance();

        Logger logger2 = Logger.getInstance();

        if (logger1 == logger2) {

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

        } else {

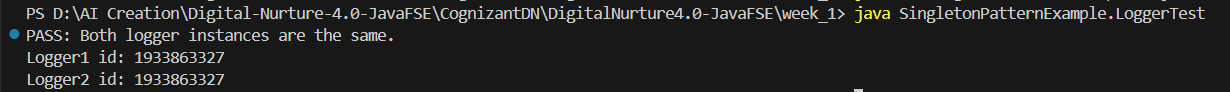
            System.out.println("FAIL: Logger instances are different.");

        }

    }

}

**Output:**



**2) Implementing the Factory Method Pattern**

Folder “[FactoryMethodPatternExample](https://github.com/Kunta-tech/Deepskilling4.0-JavaFSE/tree/master/week_1/FactoryMethodPatternExample)” in <https://github.com/Kunta-tech/Deepskilling4.0-JavaFSE/tree/master/week_1/FactoryMethodPatternExample>

**Code: (Interface classes and test class)**

Document.java

package FactoryMethodPatternExample;

public interface Document {

  public void open();

}

DocumentFactory.java

package FactoryMethodPatternExample;

public abstract class DocumentFactory {

  public abstract Document createDocument();

}

DocumentFactoryTest.java

package FactoryMethodPatternExample;

public class DocumentFactoryTest {

    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();

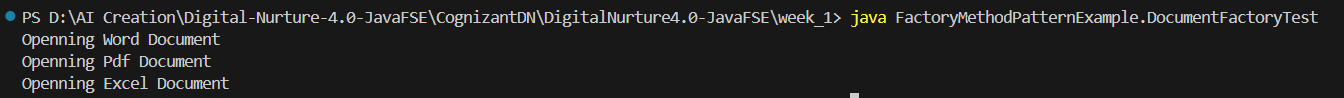
        Document excelDoc = excelFactory.createDocument();

        excelDoc.open();

    }

}

**Output:**



**3) E-commerce Platform Search Function**

**Questions:**

* Explain Big O notation and how it helps in analysing algorithms.
* **Big O notation** describes how an algorithm’s performance (time or space) grows with input size. It helps analyse and compare algorithms by focusing on their **efficiency** as inputs get larger. It helps to choose the most scalable and efficient algorithm, especially for large inputs.
* Describe the best, average, and worst-case scenarios for search operations.
* Best Case – if the item being searched for, got found in the first turn ( O(1) ).

Average Case – if the item take item is somewhere in the middle or based on a typical input. ( O(n) for Linear Search and O(log(n)) for Binary Search ).

Worst Case – if the item is not found or is at the end. (O(n) for Linear Search and O(log n) for Binary Search ).

Folder “ECommercePlatform” in <https://github.com/Kunta-tech/Deepskilling4.0-JavaFSE/tree/master/week_1/ECommercePlatform>

**Code:**

Product.java

package ECommercePlatform;

public class Product {

  String productName;

  int productID;

  String category;

  Product(int ID, String name, String category){

    this.productID = ID;

    this.productName = name;

    this.category = category;

  }

}

ProductList.java

package ECommercePlatform;

public class ProductList {

  Product[] list;

  final static int listLength = 7;

  ProductList(){

    list = new Product[listLength];

    String[] names = {"Phone", "Laptop", "Tablet", "Monitor", "Keyboard", "Mouse", "Camera"};

    String[] categories = {"Electronics", "Accessories", "Office", "Gaming"};

    for(int i=0;i<listLength;i++){

      int ID = i\*10 + 6;

      list[i] = new Product(ID, names[i], categories[(int)(Math.random()\*categories.length)]);

    }

  }

  public Product linearSearch(String name){

    for (Product product : list) {

      if(product.productName.equals(name))

        return product;

    }

    return null;

  }

  public Product binarySearch(int id){

    int leftLimit = 0,

        rightLimit = list.length-1,

        mid;

    while (leftLimit<=rightLimit) {

      mid = (leftLimit+rightLimit)/2;

      if(list[mid].productID == id) return list[mid];

      else if(list[mid].productID < id) leftLimit = mid+1;

      else if(list[mid].productID > id) rightLimit = mid-1;

    }

    return null;

  }

  static void result(Product product){

    if (product == null) System.out.println("Product not found");

    else System.out.println("Product found Id: "+product.productID+"\nname: "+product.productName+" category: "+product.category);

  }

  public static void main(String[] args) {

    ProductList productList = new ProductList();

    System.out.println("Searching with Linear Search");

    result(productList.linearSearch("Keyboard"));

    result(productList.linearSearch("Speaker"));

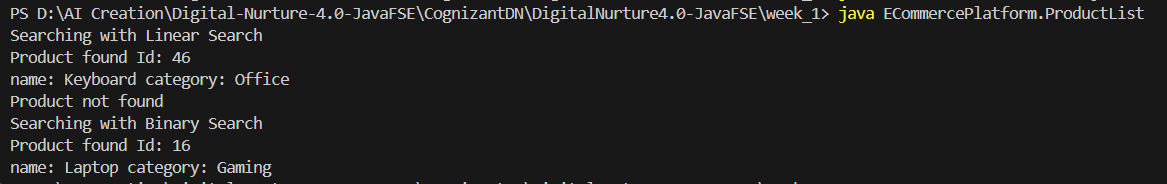
    System.out.println("Searching with Binary Search");

    result(productList.binarySearch(productList.list[1].productID));

  }

}

**Output:**



**Analysis:**

* Compare the time complexity of linear and binary search algorithms.

The worst case time complexity of Linear Search is O(n) and Binary Search is O(log n). This shows that Binary Search is the faster than Linear Search algorithm.

* Discuss which algorithm is more suitable for your platform and why.

For my platform – the product list is contains product sorted by id but unsorted by name. So for finding through id I used Binary Search and Linear Search for names.

**4) Financial Forecasting**

**Questions:**

* Explain the concept of recursion and how it can simplify certain problems.
* Recursion is a programming technique where a function calls itself to solve smaller instances of a problem. It simplifies complex problems like tree traversal or factorials by breaking them into easier subproblems, using a base case to stop the recursion.

Folder “FinancialForecaster” in <https://github.com/Kunta-tech/Deepskilling4.0-JavaFSE/tree/master/week_1/FinancialForecaster>

**Code:**

Forceaster.java

package FinancialForecaster;

public class Forecaster {

  static double rate = 20.0;

  static public double FutureValue(double presentValue, int years){

    if(years<=0) return presentValue;

    return FutureValue(presentValue, years-1)\*(1+rate/100);

  }

  public static void main(String[] args) {

    double currentValue = 1500.0;

    System.out.println("Value of "+currentValue+" in 5 years: "+FutureValue(currentValue, 5));

  }

}

**Output:**



**Analysis:**

* Discuss the time complexity of your recursive algorithm.

The Formula used for calculating the future value is:

F(n) = F(n-1) \* (1+r/100)

For ‘F(n)’ is the Value for the nth year from present and ‘r’ is rate.

So, the Time Complexity is O(n) (Linear Time)

* Explain how to optimize the recursive solution to avoid excessive computation.

Recursive solutions can be optimised by memorization or by using a constant formula.

Here, in this Financial Forecaster we can improve the time complexity by changing formula to:

F(n) = V \* (1+r/100)^n

For ‘F(n)’ is the Value for the nth year from present, ‘V’ is the current value and ‘r’ is rate.