WEEK 1 HANDS ON

TANIYA G REMULA (312422205116)

St. Joseph's Institute of Technology

Design Principles

1.Implementing the singleton pattern Code:

```
public class Main {

// Singleton Logger class
static class MyLogger {
    private static MyLogger instance;

    private MyLogger() {
        // private constructor to prevent external instantiation
    }

    public static MyLogger getInstance() {
        if (instance == null) {
            instance = new MyLogger();
        }
        return instance;
    }
}

public static void main(String[] args) {
```

```
        MyLogger log1 = MyLogger.getInstance();
        MyLogger log2 = MyLogger.getInstance();

        if (log1 == log2) {
            System.out.println("Same logger used");
        } else {
            System.out.println("Different loggers used");
        }
    }
}
```

Output:

```
Output                                                        Clear
Same logger used

=== Code Execution Successful ===
```

2.Factory Method Pattern

Code:

```
public class Main {

    public static void main(String[] args) {
        DocFactory wordFactory = new WordFactory();
        Doc word = wordFactory.create();
        word.open();
```

```java
        DocFactory pdfFactory = new PdfFactory();
        Doc pdf = pdfFactory.create();
        pdf.open();

        DocFactory excelFactory = new ExcelFactory();
        Doc excel = excelFactory.create();
        excel.open();
    }
}

interface Doc {
    void open();
}

class Word implements Doc {
    public void open() {
        System.out.println("opening word");
    }
}

class Pdf implements Doc {
    public void open() {
        System.out.println("opening pdf");
```

```java
    }
}

class Excel implements Doc {
    public void open() {
        System.out.println("opening excel");
    }
}

abstract class DocFactory {
    public abstract Doc create();
}

class WordFactory extends DocFactory {
    public Doc create() {
        return new Word();
    }
}

class PdfFactory extends DocFactory {
    public Doc create() {
        return new Pdf();
    }
}
```

```
class ExcelFactory extends DocFactory {
    public Doc create() {
        return new Excel();
    }
}
```

Output:

Output Clear

```
opening word
opening pdf
opening excel

=== Code Execution Successful ===
```

Data Structure and Algorithm Hands On

3. E_Commerce Platform Search Function

Big O notation: Big O notation describes the upper bound of an algorithms space and time complexity with the respective input size n. It helps us to understand the performance of the algorithm with respective to time and space there are three cases here best,average and worst cases.

Best case: When the element is found immediately Average case: when the element is found in middle region.

Worst Case: when the element is found at the last region.

Code:

```
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 class Main {

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

```
        for (Product product : products) {
            if (product.productname.equals(targetName)) {
                return product;
            }
        }
        return null;
    }
    public static Product binarySearch(Product[] products, String
targetName) {
        int left = 0, right = products.length - 1;
        while (left <= right) {
            int mid = left + (right - left) / 2;
            int result =
products[mid].productname.compareTo(targetName);
            if (result == 0) return products[mid];
            else if (result < 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, "Phone", "Electronics"),
```

```java
            new Product(3, "Shirt", "Apparel"),
            new Product(4, "Shoes", "Footwear"),
            new Product(5, "Watch", "Accessories")
        };
        Product result1 = linearSearch(products, "Shoes");
        if (result1 != null)
            System.out.println("Linear Search Result: " + result1.productname);

        // Sort products before binary search
        Arrays.sort(products, Comparator.comparing(p -> p.productname));
        Product result2 = binarySearch(products, "Shoes");
        if (result2 != null)
            System.out.println("Binary Search Result: " + result2.productname);
    }
}
```

Output:

Output	Clear

```
Linear Search Result: Shoes
Binary Search Result: Shoes

=== Code Execution Successful ===
```

Time Complexity of linear search and binary search

Linear Search

Best case: O(1)

Average case: O(n/2)

Worst case: O(n)

Binary Search:

Best case: O(1)

Average case: O(log n)

Worst case: O(log n)

Best Algorithm that fits this program

In this program E_Commerce Application the most suitable Algorithm is Binary Search because linear search is suitable for small data sets but e_commerce application is a growing application so for large data sets so binary search is the most suitable algorithm

4.Financial Forecasting

Recursion

Recursion is a concept where function call itself to solve the specific problem where the code logic need to be executed repeatedly.

Code:

```
public class Main {

    public static double predict(double currentValue, double growthRate, int years) {
        if (years == 0) return currentValue;
        return predict(currentValue * (1 + growthRate), growthRate, years - 1);
    }

    public static void main(String[] args) {
        double currentValue = 10000;
        double growthRate = 0.08;
        int years = 5;

        double futureValue = predict(currentValue, growthRate, years);
        System.out.println("Future Value (Recursive): " + futureValue);

        // Optimized using mathematical formula
        double optimizedValue = currentValue * Math.pow(1 + growthRate, years);
        System.out.println("Future Value (Optimized): " + optimizedValue);
    }
}
```

Output:

```
Output                                                          Clear
Future Value (Recursive): 14693.280768000004
Future Value (Optimized): 14693.280768000006

=== Code Execution Successful ===
```

Time Complexity:

Here in this java code the time complexity for recursion is O(n) where the recursion call is happening base on the no of years.

Optimization Approach:

The optimization approach for this problem is to use mathematical formula where we can get in output in O(1) time complexity which avoid unnecessary recursive calls.

The formula used in this problem is:
Future Value = Current Value*(1+growth rate)^years.