WEEK-1 HANDS ON

Design Patterns and principles

Exercise 1: Implementing the Singleton Pattern

- Created a java project named “SingletonPatternExample” in Eclipse IDE.
- Created two java files named “Logger.java” and “LoggerMain.java”.

Logger.java

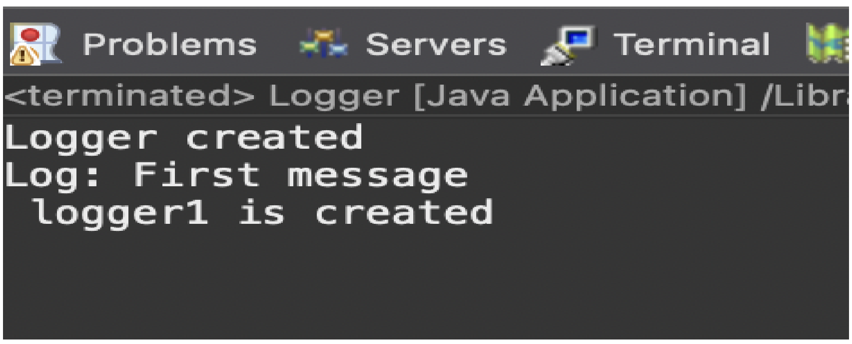
```
package com.example.logger;
public class Logger {
  private static Logger instance = null;
  private Logger() {
     System.out.println("Logger created");
  }
  public static Logger getInstance() {
     if (instance == null) {
        instance = new Logger();
     }
     return instance;
  }
  public void log(String message) {
     System.out.println("Log: " + message);
  }
}
```

LoggerMain.java

```
package com.example.logger;
public class LoggerMain {
        public static void main(String[] args) {
                Logger logger1= Logger.getInstance();
                  logger1.log("First message");

                  System.out.println(" logger1 is created");
  }
}
```

Output



Exercise 2: Implementing the Factory Method Pattern

- Created a java interface named “Document.java” in Eclipse IDE.
- Create respective classes for handling pdf, word and excel files.

FactoryMethodPatternExample.java

```
class FactoryMethodPatternExample {
   public Document createDocument(String type) {
       if (type.equalsIgnoreCase("worddocument")) {
           return new WordDocument();
       } else if (type.equalsIgnoreCase("pdfdocument")) {
           return new PdfDocument();
       } else if (type.equalsIgnoreCase("exceldocument")) {
           return new ExcelDocument();
       }
       return new WordDocument();
       }
   }
```

Document.java

```
public interface Document {
          void open();
}
```

WordDocument.java

```
public class WordDocument implements Document{
          public void open() {
       System.out.println("Opening Word document");
   }
}
```

PdfDocument.java

```
public class PdfDocument implements Document{
           public void open() {
                  System.out.println("Opening PDF document");
              }
}
```

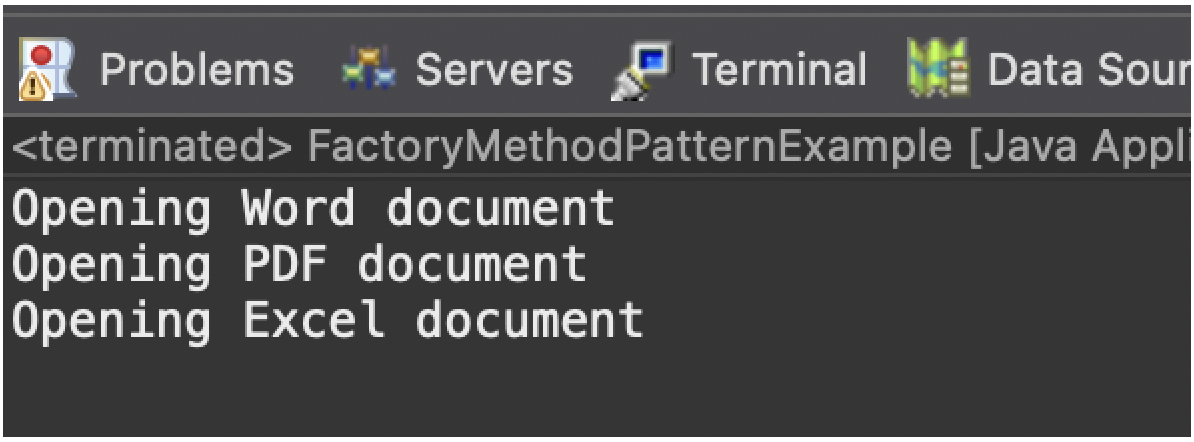
ExcelDocument.java

```java
public class ExcelDocument implements Document{
        public void open() {
                System.out.println("Opening Excel document");
        }
}
```

FactoryMain.java

```java
public class FactoryMain {
  public static void main(String[] args) {
     FactoryMethodPatternExample factory = new FactoryMethodPatternExample();
     Document word = factory.createDocument("worddocument");
     word.open();
     Document pdf = factory.createDocument("pdfdocument");
     pdf.open();
     Document excel = factory.createDocument("exceldocument");
     excel.open();
  }
}
```

Output:



Algorithms_Data Structures

Exercise 2: E-commerce Platform Search Function

Big O Notation - Big O notation is used to analyze the upper bound of an algorithm’s time and space complexity. It simply helps us to understand at what rate the algorithm runs with respect to the input size.

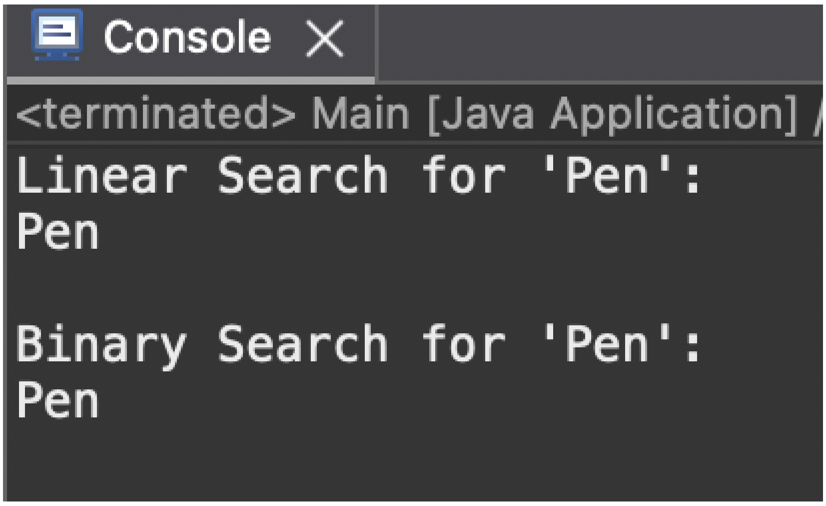
Linear Search
Best case- O(1) - When element is present at the start of the list/array.
Average case- O(n/2) - When the element is in the middle.
Worst case - O(n) - When the element is at the end of the list or not found.

Binary Search
Best case- O(1) - When element is present at the middle of the list/array.
Average case- O(logn) - When we keep on dividing the list until it's found.
Worst case - O(logn) - When we keep on dividing the list until it’s found or not.

```
package com.example.logger;
import java.util.*;
public class Main {
  public static void main(String[] args) {
      ArrayList<Product> products = new ArrayList<>();
     products.add(new Product(1, "Apple", "Fruit"));
     products.add(new Product(2, "Pen", "Stationary"));
     products.add(new Product(3, "Brinjal", "Vegetable"));
     System.out.println("Linear Search for 'Pen':");
     ArrayList<Product> linearResults = linearSearch(products, "Pen");
     for (Product p : linearResults) {
        System.out.println(p);
     }
     System.out.println("\nBinary Search for 'Pen':");
     Product result = binarySearch(products, "Pen");
     if (result != null) {
        System.out.println(result);
     } else {
        System.out.println("Product not found.");
     }
  }
   static class Product implements Comparable<Product> {
     int id;
     String name;
     String category;
     Product(int id, String name, String category) {
        this.id = id;
        this.name = name;
```

```
        this.category = category;
    }
    public String toString() {
        return name;
    }
    public int compareTo(Product other) {
        return this.name.compareToIgnoreCase(other.name);
    }
}
 public static ArrayList<Product> linearSearch(ArrayList<Product> products, String
keyword) {
    ArrayList<Product> found = new ArrayList<>();
    for (Product p : products) {
        if (p.name.toLowerCase().contains(keyword.toLowerCase())) {
            found.add(p);
        }
    }
    return found;
}
public static Product binarySearch(ArrayList<Product> products, String keyword) {
    int left = 0;
    int right = products.size() - 1;
    keyword = keyword.toLowerCase();
    while (left <= right) {
        int mid = (left + right) / 2;
        String midName = products.get(mid).name.toLowerCase();
        if (midName.equals(keyword)) {
            return products.get(mid);
        } else if (keyword.compareTo(midName) < 0) {
            right = mid - 1;
        } else {
            left = mid + 1;
        }
    }
    return null;
}
}
```

Output:



For this example, since the size of the list is small, it is preferred to use linear search.

Exercise 7: Financial Forecasting

- **Recursion :** Recursion is a concept in programming where a function calls itself to solve smaller parts of the code. It is very helpful to simplify certain problems by breaking the entire code into smaller sub problems. It handles the logic without having to call the entire function again and again.

```java
public class FinancialForecasting {
  public static double predict(int currVal, int time) {
     if (time == 0) {
        return currVal;
     }
     return predict(currVal*4+time, time-1);
  }
  public static void main(String[] args) {
     int currVal = 500;
     int time = 3;
     double futureValue = predict(currVal, time);
     System.out.printf("Predicted value after %d years: %f", time, futureValue);
  }
}
```

Output:

Time complexity = O(n)
We can optimize the recursive solution by simply following an iterative process or storing the values from previous calls into an array.