WEEK 1

Kavya V.R

St. Joseph's Institute of Technology

Superset ID: 6377320

Design Pattern and Principles

1.Implementing the singleton pattern

Code:

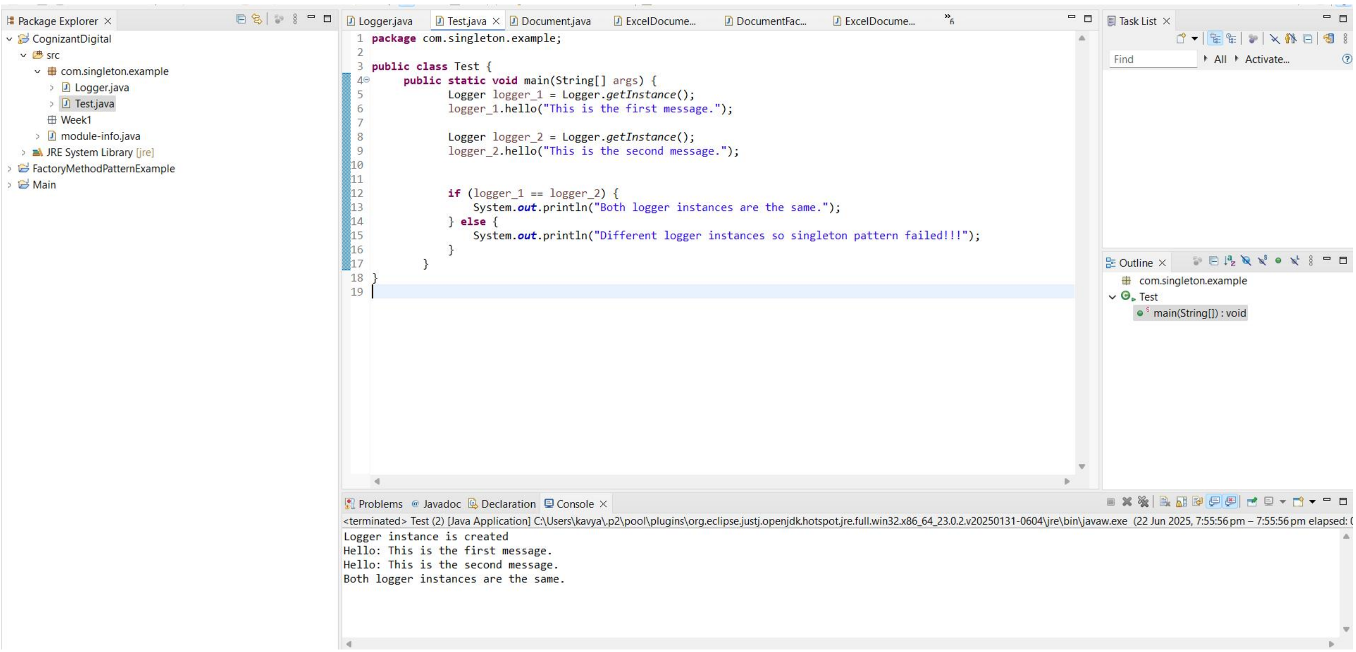
Logger.java

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

Test.java

```
package com.singleton.example;
public class Test {
 public static void main(String[] args) {
        Logger logger_1 = Logger.getInstance();
        logger_1.hello("This is the first message.");
        Logger logger_2 = Logger.getInstance();
        logger_2.hello("This is the second message.");
        if (logger_1 == logger_2) {
           System.out.println("logger instances are the
same.");
        } else {
           System.out.println("Different logger instances so
singleton pattern failed!!!");
        }
    }
}
```

Output:



2.Factory Method Pattern

Code:

TestFactoryMethod.java

```
package factorypattern;
public class TestFactoryMethod {
    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();
    }
}
```

Document.java

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

DocumentFactory.java

```
package factorypattern;
public abstract class DocumentFactory {
    public abstract Document createDocument();
}
```

ExcelDocument.java

```
package factorypattern;
public class ExcelDocument implements Document {
    public void open() {
        System.out.println("Opening the Excel Document!!!");
    }
}
```

ExcelDocumentFactory.java

```
package factorypattern;
```

```java
public class ExcelDocumentFactory extends DocumentFactory {
    public Document createDocument() {
        return new ExcelDocument();
    }
}
```

PdfDocument.java

```java
package factorypattern;
public class PdfDocument implements Document {
    public void open() {
        System.out.println("Opening the PDF Document!!!");
    }
}
```

PdfDocumentFactory.java

```java
package factorypattern;
public class PdfDocumentFactory extends DocumentFactory {
    public Document createDocument() {
        return new PdfDocument();
    }
}
```

WordDocument.java

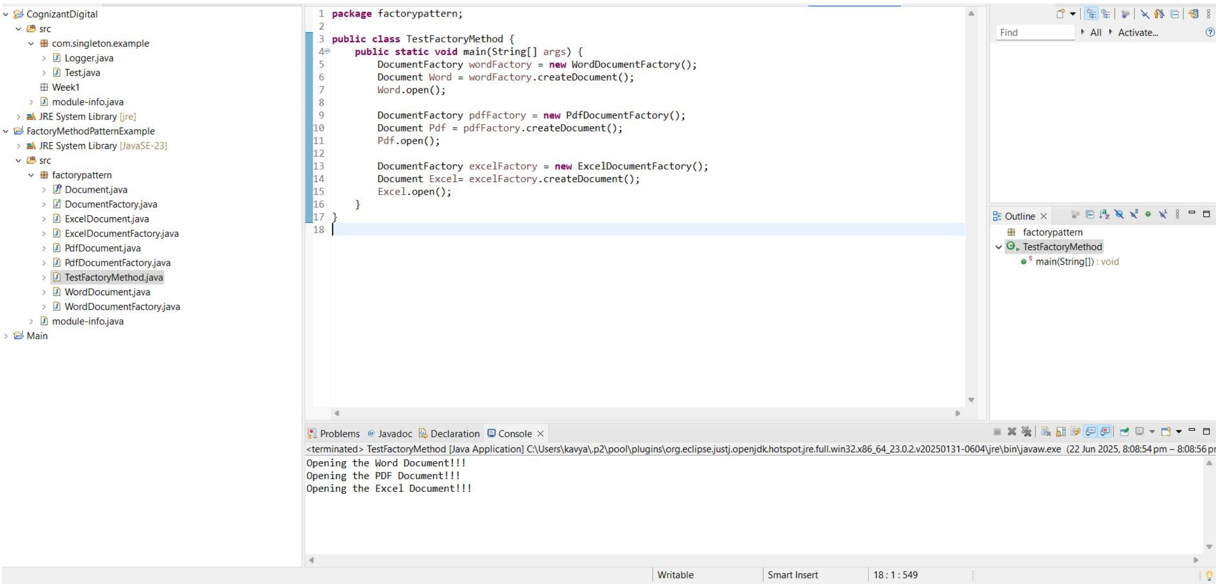
```java
package factorypattern;
public class WordDocument implements Document {
    public void open() {
        System.out.println("Opening the Word Document!!!");
```

```
    }
}
```

WordDocumentFactory.java

```
package factorypattern;
public class WordDocumentFactory extends DocumentFactory {
    public Document createDocument() {
        return new WordDocument();
    }
}
```

Output:



Data Structure and Algorithm

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 easily understand the performance of the algorithm with respective to time and space without running the actual code. there are three cases here best, average and worst cases.

Linear Search:

Best case: Target element is the first element

Average case: Target element in the middle or not present.

Worst Case: Target element is the last element or not present

Best case: Target element is the middle element

Average case: target element is found after repeatedly

halving the search space.

Worst Case: target element is found after repeatedly halving the search space.

Code:

Product.java:

```
package com.ecommerce.search;
public class Product {
 int productId;
    String productName;
```

```
    String category;
public Product(int id, String name, String category) {
        this.productId = id;
        this.productName = name;
        this.category = category;
    }
}
```

ProductSearch.java:

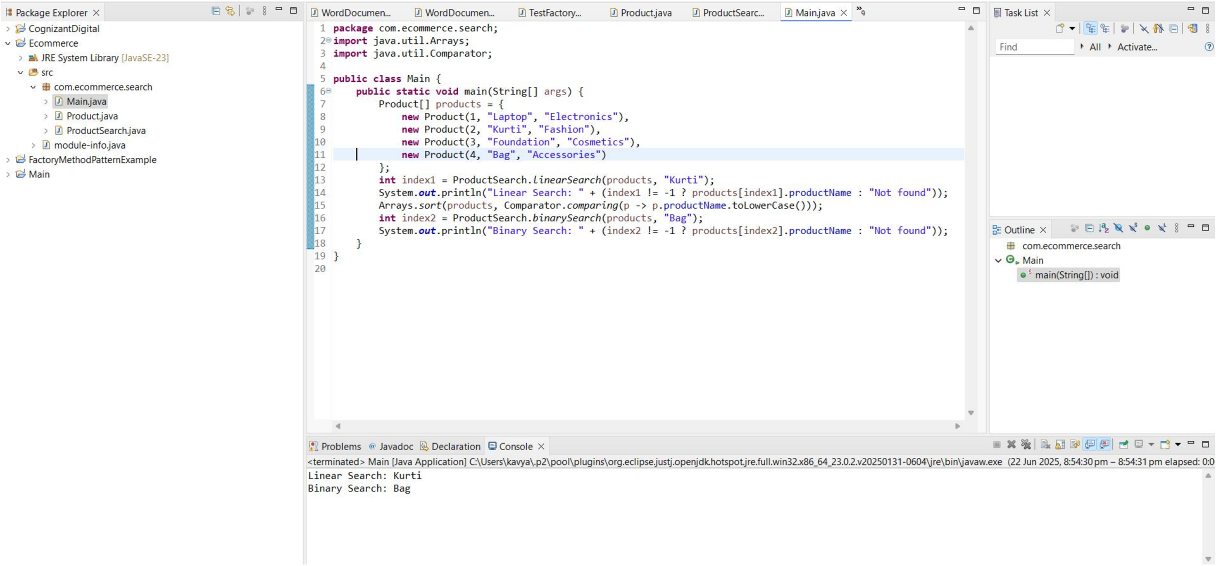
```
package com.ecommerce.search;
public class ProductSearch {
public static int linearSearch(Product[] products, String targetName)
{
        for (int i = 0; i < products.length; i++) {
            if (products[i].productName.equalsIgnoreCase(targetName))
{
                return i;
            }
        }
        return -1;
    }

public static int binarySearch(Product[] products, String targetName)
{
        int left = 0, right = products.length - 1;
        while (left <= right) {
            int mid = left + (right - left) / 2;
            int compare =
products[mid].productName.compareToIgnoreCase(targetName);
            if (compare == 0) return mid;
            else if (compare < 0) left = mid + 1;
            else right = mid - 1;
        }
        return -1;
    }
```

```
}
```

Main.java:

```
package com.ecommerce.search;
import java.util.Arrays;
import java.util.Comparator;

public class Main {
public static void main(String[] args) {
        Product[] products = {
            new Product(1, "Laptop", "Electronics"),
            new Product(2, "Kurti", "Fashion"),
            new Product(3, "Foundation", "Cosmetics"),
            new Product(4, "Bag", "Accessories")
        };
        int index1 = ProductSearch.linearSearch(products, "Kurti");
        System.out.println("Linear Search: " + (index1 != -1 ?
products[index1].productName : "Not found"));
        Arrays.sort(products, Comparator.comparing(p ->
p.productName.toLowerCase()));
        int index2 = ProductSearch.binarySearch(products, "Bag");
        System.out.println("Binary Search: " + (index2 != -1 ?
products[index2].productName : "Not found"));
    }
}
```

Output:



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 Practice for E-commerce:Best Algorithm for E-commerce is the Binary Search which is better for the real-world platforms because Linear search is best for the small dataset where binary search is very fast for the large,sorted data. As E-commerce application is large data application binary search algorithm will be most efficient.

4.Financial Forecasting

Recursion

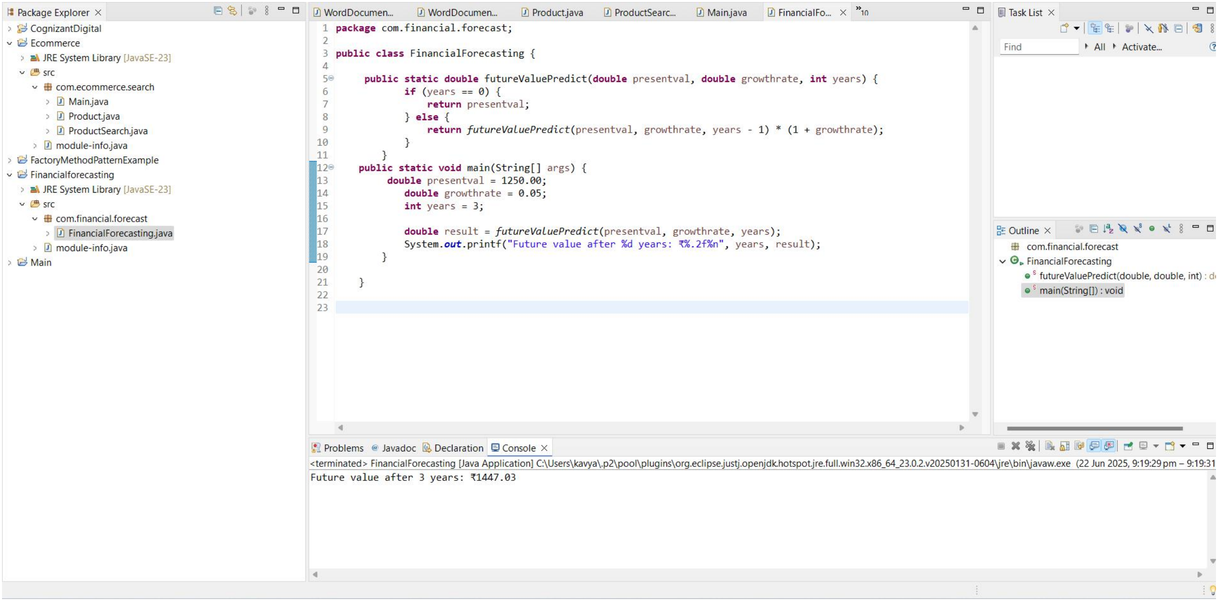
Recursion is a programming technique where a function calls itself to solve a problem by breaking it down into smaller, self-similar subproblems until the base case is reached.

Code:

```
package com.financial.forecast;

public class FinancialForecasting {

 public static double futureValuePredict(double presentval, double growthrate, int years) {

        if (years == 0) {

            return presentval;

        } else {

            return futureValuePredict(presentval, growthrate, years - 1) * (1 + growthrate);

        }

    }

public static void main(String[] args) {

 double presentval = 1250.00;

        double growthrate = 0.05;

        int years = 3;


        double result = futureValuePredict(presentval, growthrate, years);

        System.out.printf("Future value after %d years: ₹%.2f%n", years, result);
```

```
    }
}
```

Output:



Time Complexity:

O(n) where the recursion call for no of years.

One call for each year

Optimization Approach:

We can implement mathematical formula where is result time complexity of O(1) and O(1) space complexity.

The formula used in this problem is:
Future Value = presentvalue*(1+growthrate)^years.