JAVA FSE Mandatory Module:

MODULE1:

Exercise1: Implementing theSingleton Pattern:

Code:

Singleton.java:

public class Singleton {

    public static void main(String[] args) {

        Logger logger1 = Logger.getInstance();

        logger1.log("log message");

         Logger logger2 = Logger.getInstance();

        logger2.error(" error");

        if (logger1 == logger2) {

            System.out.println(" same instance");

        } else {

            System.out.println(" multiple instances");

        }

          logger1.warn("warning message");

    }

}

Logger.java:

public class Logger {

    private static Logger instance;

    private Logger() {

    System.out.println("Logger instance created");

    }

    public static Logger getInstance() {

        if (instance == null) {

            instance = new Logger();

        }

        return instance;

    }

    public void log(String message) {

        System.out.println("LOG: " + message);

    }

    public void error(String message) {

        System.out.println("ERROR: " + message);

    }

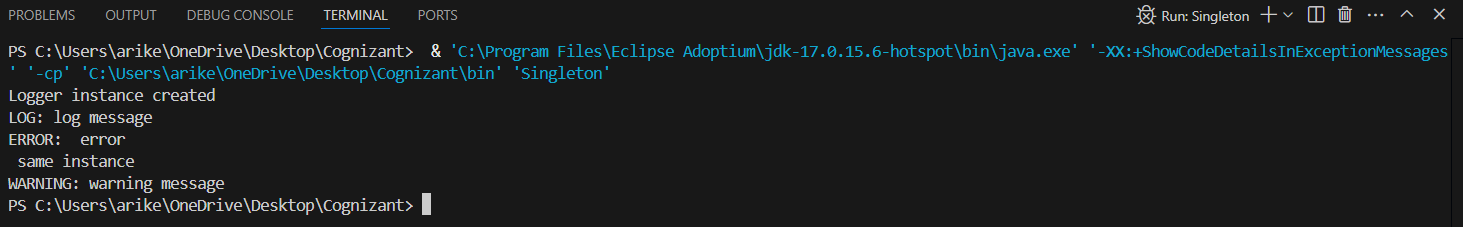
    public void warn(String message) {

        System.out.println("WARNING: " + message);

    }

}

Output:



Exercise 2: Implementing the Factory Method Pattern:

Code:

Document package:

Document.java:

package documents;

public interface Document {

    void open();

    void close();

    void save();

    }

ExcelDocument.java:

package documents;

public class ExcelDocument implements Document {

    @Override

    public void open(){

        System.out.println("exelsheet");

}

    @Override

    public void close(){

        System.out.println("close excelsheet");

    }

    @Override

    public void save(){

        System.out.println("save excel");

    }

}

PdfDocument.java:

package documents;

public class PdfDocument implements Document {

@Override

public void open() {

    System.out.println("Opening PDF file");

    }

@Override

public void close() {

        System.out.println("Closing PDF file");

    }

@Override

    public void save() {

     System.out.println("Saving PDF file");

    }

}

WordDocument.java:

package documents;

public class WordDocument implements Document {

    @Override

    public void open() {

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

    }

    @Override

    public void close() {

        System.out.println("Closing Word file");

    }

    @Override

    public void save() {

        System.out.println("Saving Word file");

    }

}

WordDocument.java:

package documents;

public class WordDocument implements Document {

    @Override

    public void open() {

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

    }

    @Override

    public void close() {

        System.out.println("Closing Word file");

    }

    @Override

    public void save() {

        System.out.println("Saving Word file");

    }

}

factory package:

DocumentFactory.java:

package factory;

import documents.Document;

public abstract class DocumentFactory {

    public abstract Document createDocument();

   public void processDocument() {

        Document doc1 = createDocument();

        doc1.open();

        doc1.save();

        doc1.close();

    }

}

PdfFactory.java:

package factory;

import documents.Document;

import documents.PdfDocument;

public class PdfFactory extends DocumentFactory {

    @Override

    public Document createDocument() {

  return new PdfDocument();

    }

    }

ExcelFactory.java:

package factory;

import documents.Document;

import documents.ExcelDocument;

public class ExcelFactory extends DocumentFactory {

     @Override

    public Document createDocument() {

        return new ExcelDocument();

    }

    }

WordFactory.java:

package factory;

import documents.WordDocument;

import documents.Document;

public class WordFactory extends DocumentFactory{

    @Override

    public Document createDocument() {

        return new WordDocument();

    }

    }

Test.java:

import factory.\*;

public class Test{

    public static void main(String[] args) {

        DocumentFactory wordFactory = new WordFactory();

        DocumentFactory pdfFactory = new PdfFactory();

        DocumentFactory excelFactory = new ExcelFactory();

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

        wordFactory.processDocument();

        System.out.println("\nPDF document:");

        pdfFactory.processDocument();

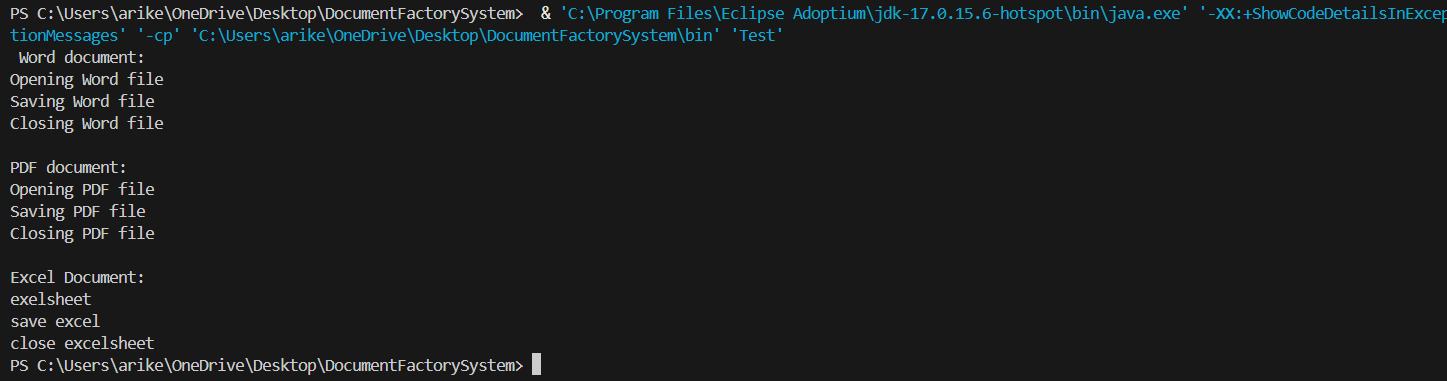
        System.out.println("\nExcel Document:");

        excelFactory.processDocument();

    }

}

Output:



MODULE2:

Exercise 2: E-commerce Platform Search Function:

Big-0 Notation:

Two functions f(n) and g(n), we say that f(n) is O(g(n)) if there exist constants c > 0 and n0 >= 0 such that f(n) <= c\*g(n) for all n >= n0.

Performance:

Linear and Binary comparision: Best case: both has O(1), in average case O(n) for linear search and O(logn) for binary search. Coming to worst cases O(n) for linear and O(logn) for binary search.

Usually the best way for searching the elements is Binary search as they are differed in Time complexity and comparing it with space complexity both has same space complexity.

Code:

import java.util.Arrays;

import java.util.Comparator;

public class Ecommerce {

// Product class with int id

static class Product {

int id;

String name;

String category;

public Product(int id, String name, String category) {

this.id = id;

this.name = name;

this.category = category;

}

}

//Here I cosidered string for searching

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

for (Product p : products) {

if (p.name.equalsIgnoreCase(name)) {

return p;

}

}

return null;

}

// where as here I used integer array elements

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

int left = 0;

int right = products.length - 1;

while (left <= right) {

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

int midId = products[mid].id;

if (midId == id) {

return products[mid];

} else if (midId < id) {

left = mid + 1;

} else {

right = mid - 1;

}

}

return null;

}

public static void main(String[] args) {

Product[] products = {

new Product(100, "Mouse", "Electronics"),

new Product(200, "Headphones", "Electronics"),

new Product(150, "Lamp", "Study table")

};

// Sort products by id for binary search

Arrays.sort(products, Comparator.comparingInt(p -> p.id));

// Output section

System.out.println(" Demo");

// Linear search by name

System.out.println("\n[Linear Search Using Name]");

String searchName = "Mouse";

Product foundByName = linearSearch(products, searchName);

if (foundByName != null) {

System.out.printf("Found: %s ,ID: %d\n",

foundByName.name, foundByName.id);

} else {

System.out.println("Product not found.");

}

// Binary search by ID

System.out.println("\n[Binary Search by ID]");

int searchId = 20;

Product foundById = binarySearch(products, searchId);

if (foundById != null) {

System.out.printf("Found: %s ,Category: %s\n",

foundById.name, foundById.category);

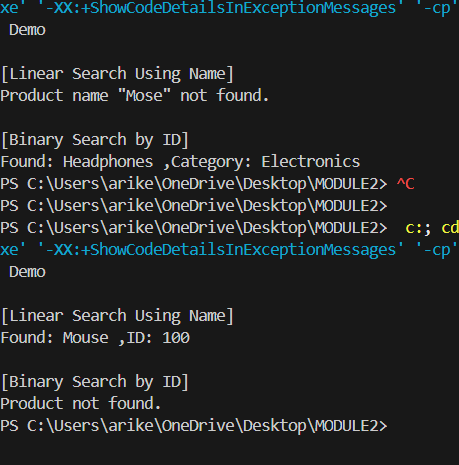
} else {

System.out.println("Product not found.");

}

}

}



Performance Analysis:

Binary Search is more optimal compared to linear search as it has time complexity of O(log n).

Exercise 7: Financial Forecasting:

Code:

public class Interest {

    // This method calculates the future value using compound interest

    public static double Intrest(

        double P,

        double annualRate

        int years,

        int N

    ) {

        // Formula: A = P \* (1 + r/n) ^ (n \* t)

        return P \* Math.pow(

            1 + (annualRate / 100) / N,

            N \* years

        );

    }

    public static void main(String[] args) {

        double P = 1000.0

        double annualRate = 6;

        int years = 5;

        int N = 365;

        double futureValue = Intrest(P, annualRate, years, N);

        System.out.printf(

            "$%.2f invested at %.1f%% annual rate for %d years  grows to $%.2f",

            P, annualRate, years, futureValue

        );

    }

}

Output:

