Candidate Name:- ROHITHGOWDA V

Superset ID:-6430364

Mail ID:- rohith2005v@gmail.com

## WEEK – 1 HANDS ON EXERCISE (JAVA FSE DEEPSKILLING)

## (DESIGN PATTERN AND PRINCIPLES)

### Exercise 2: Implementing the Factory Method Pattern

**Scenario:**

You are developing a document management system that needs to create different types of documents (e.g., Word, PDF, Excel). Use the Factory Method Pattern to achieve this.

**Steps:**

1. **Create a New Java Project:**
   - Create a new Java project named **FactoryMethodPatternExample**.
2. **Define Document Classes:**
   - Create interfaces or abstract classes for different document types such as **WordDocument**, **PdfDocument**, and **ExcelDocument**.
3. **Create Concrete Document Classes:**
   - Implement concrete classes for each document type that implements or extends the above interfaces or abstract classes.
4. **Implement the Factory Method:**
   - Create an abstract class **DocumentFactory** with a method **createDocument()**.
   - Create concrete factory classes for each document type that extends DocumentFactory and implements the **createDocument()** method.
5. **Test the Factory Method Implementation:**
   - Create a test class to demonstrate the creation of different document types using the factory method.

### Code for the above question:-

```
import java.util.*;

class FactoryMethodPatternExample {

  public static void main(String[] args) {

    DocumentFactory wordFactory = new WordDocumentFactory();

    DocumentFactory pdfFactory = new PdfDocumentFactory();
```

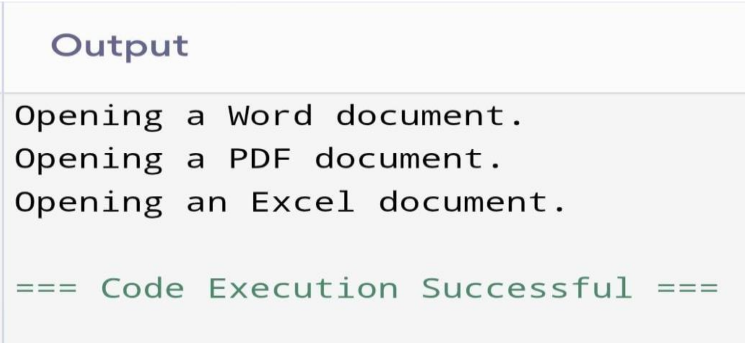
```
        DocumentFactory excelFactory = new ExcelDocumentFactory();
        Document wordDoc = wordFactory.createDocument();
        Document pdfDoc = pdfFactory.createDocument();
        Document excelDoc = excelFactory.createDocument();
        wordDoc.open();
        pdfDoc.open();
        excelDoc.open();
    }
}
interface Document {
    void open();
}
class WordDocument implements Document {
    public void open() {
        System.out.println("Opening a Word document.");
    }
}
class PdfDocument implements Document {
    public void open() {
        System.out.println("Opening a PDF document.");
    }
}
class ExcelDocument implements Document {
    public void open() {
        System.out.println("Opening an Excel document.");
    }
}
abstract class DocumentFactory {
    public abstract Document createDocument();
}
class WordDocumentFactory extends DocumentFactory {
    public Document createDocument() {
        return new WordDocument();
    }
}
class PdfDocumentFactory extends DocumentFactory {
    public Document createDocument() {
        return new PdfDocument();
    }
}
class ExcelDocumentFactory extends DocumentFactory {
    public Document createDocument() {
        return new ExcelDocument();
    }
}
```

**Output:-**

```
Opening a Word document.
Opening a PDF document.
Opening an Excel document.

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

**Output Image:-**



```
Opening a Word document.
Opening a PDF document.
Opening an Excel document.

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

# (DESIGN PATTERN AND PRINCIPLES)

**Exercise 1: Implementing the Singleton Pattern**

**Scenario:**

You need to ensure that a logging utility class in your application has only one instance throughout the application lifecycle to ensure consistent logging.

**Steps:**

1. **Create a New Java Project:**
   - Create a new Java project named **SingletonPatternExample**.
2. **Define a Singleton Class:**
   - Create a class named Logger that has a private static instance of itself.
   - Ensure the constructor of Logger is private.
   - Provide a public static method to get the instance of the Logger class.
3. **Implement the Singleton Pattern:**
   - Write code to ensure that the Logger class follows the Singleton design pattern.
4. **Test the Singleton Implementation:**
   - Create a test class to verify that only one instance of Logger is created and used across the application.

## Code for the above question:-

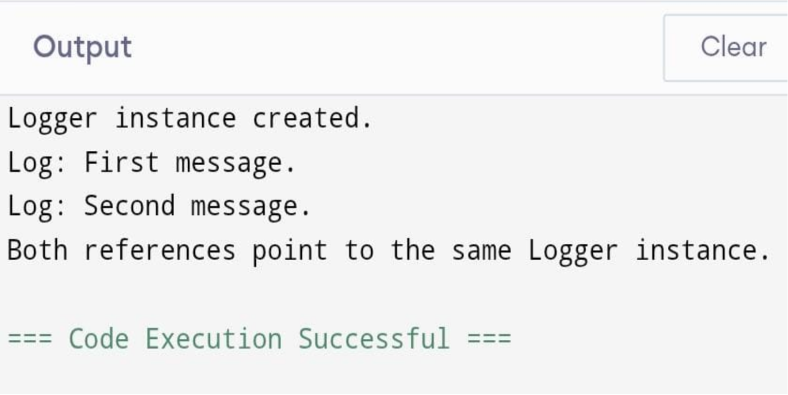
```
import java.util.*;
public class SingletonPatternExample {
  public static void main(String[] args) {
    Logger logger1 = Logger.getInstance();
    logger1.log("First message.");
    Logger logger2 = Logger.getInstance();
    logger2.log("Second message.");
    if (logger1 == logger2) {
      System.out.println("Both references point to the same Logger instance.");
    } else {
      System.out.println("Different Logger instances.");
    }
  }
}
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);
    }
}
```

Output:-

```
Logger instance created.
Log: First message.
Log: Second message.
Both references point to the same Logger instance.

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

Output Image:-



```
Logger instance created.
Log: First message.
Log: Second message.
Both references point to the same Logger instance.

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

# (ALGORITHM DATA STRUCTURES)

**Exercise 2: E-commerce Platform Search Function**

**Scenario:**

You are working on the search functionality of an e-commerce platform. The search needs to be optimized for fast performance.

**Steps:**

1. **Understand Asymptotic Notation:**
   - Explain Big O notation and how it helps in analyzing algorithms.
   - Describe the best, average, and worst-case scenarios for search operations.
2. **Setup:**
   - Create a class **Product** with attributes for searching, such as **productId, productName**, and **category**.
3. **Implementation:**
   - Implement linear search and binary search algorithms.
   - Store products in an array for linear search and a sorted array for binary search.
4. **Analysis:**
   - Compare the time complexity of linear and binary search algorithms.
   - Discuss which algorithm is more suitable for your platform and why.

## Code for above question:-

```
import java.util.*;

public class ECommerceSearch {

  public static void main(String[] args) {

    Product[] products = {

      new Product(103, "Laptop", "Electronics"),

      new Product(101, "Shirt", "Apparel"),

      new Product(105, "Headphones", "Electronics"),

      new Product(102, "Book", "Education"),

      new Product(104, "Shoes", "Footwear")

    };

    System.out.println("Linear Search:");

    Product result1 = linearSearch(products, 102);
```

```
        if (result1 != null) {
            System.out.println("Found: " + result1);
        } else {
            System.out.println("Product not found.");
        }

        Arrays.sort(products);

        System.out.println("Binary Search:");
        Product result2 = binarySearch(products, 104);
        if (result2 != null) {
            System.out.println("Found: " + result2);
        } else {
            System.out.println("Product not found.");
        }
    }

    public static Product linearSearch(Product[] products, int id) {
        for (Product product : products) {
            if (product.productId == id) {
                return product;
            }
        }
        return null;
    }

    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;
            if (products[mid].productId == id) {
                return products[mid];
            } else if (products[mid].productId < id) {
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }
        return null;
    }
}

class Product implements Comparable<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 String toString() {
        return productId + " - " + productName + " (" + category + ")";
    }

    public int compareTo(Product other) {
        return Integer.compare(this.productId, other.productId);
    }
}
```

Output:-

Linear Search:

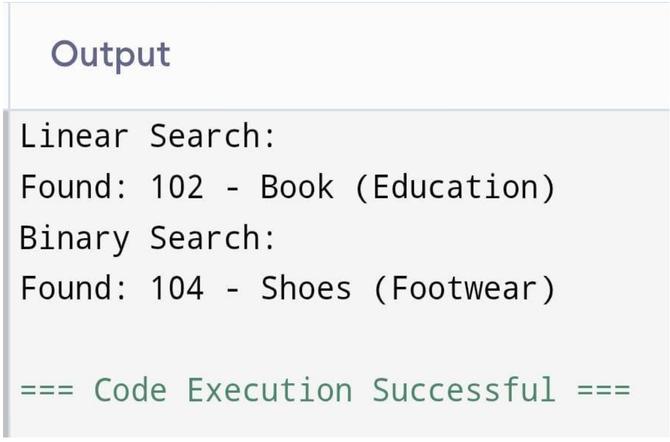
Found: 102 - Book (Education)

Binary Search:

Found: 104 - Shoes (Footwear)

=== Code Execution Successful ===

Output Image:-



```
Linear Search:
Found: 102 - Book (Education)
Binary Search:
Found: 104 - Shoes (Footwear)

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

## (ALGORITHM DATA STRUCTURES)

**Exercise 7: Financial Forecasting**

**Scenario:**

You are developing a financial forecasting tool that predicts future values based on past data.

**Steps:**

1. **Understand Recursive Algorithms:**
   - Explain the concept of recursion and how it can simplify certain problems.
2. **Setup:**
   - Create a method to calculate the future value using a recursive approach.
3. **Implementation:**
   - Implement a recursive algorithm to predict future values based on past growth rates.
4. **Analysis:**
   - Discuss the time complexity of your recursive algorithm.
   - Explain how to optimize the recursive solution to avoid excessive computation.

### Code for above question:-

```
import java.util.Scanner;
public class FinancialForecasting {
   public static double forecast(double initial, double rate, int years) {
      if (years == 0) {
         return initial;
      }
      return forecast(initial, rate, years - 1) * (1 + rate);
   }
   public static void main(String[] args) {
      Scanner scanner = new Scanner(System.in);
      System.out.print("Enter initial amount: ");
      double initial = scanner.nextDouble();
      System.out.print("Enter annual growth rate (e.g., 0.05 for 5%): ");
      double rate = scanner.nextDouble();
      System.out.print("Enter number of years: ");
```

```
        int years = scanner.nextInt();
        double futureValue = forecast(initial, rate, years);
        System.out.printf("Future value after %d years: %.2f\n", years, futureValue);
    }
}
```

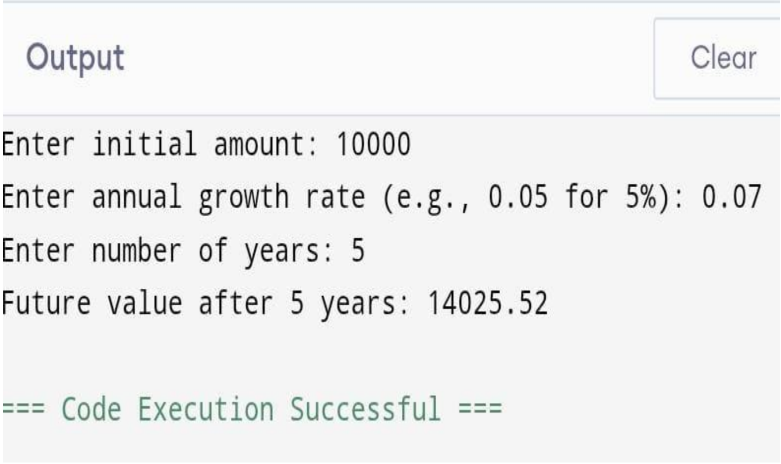
Input:-

```
Enter initial amount: 10000
Enter annual growth rate (e.g., 0.05 for 5%): 0.07
Enter number of years: 5
```

Output:-

```
Future value after 5 years: 14025.52
=== Code Execution Successful ===
```

Output Image:-



```
Enter initial amount: 10000
Enter annual growth rate (e.g., 0.05 for 5%): 0.07
Enter number of years: 5
Future value after 5 years: 14025.52

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