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.

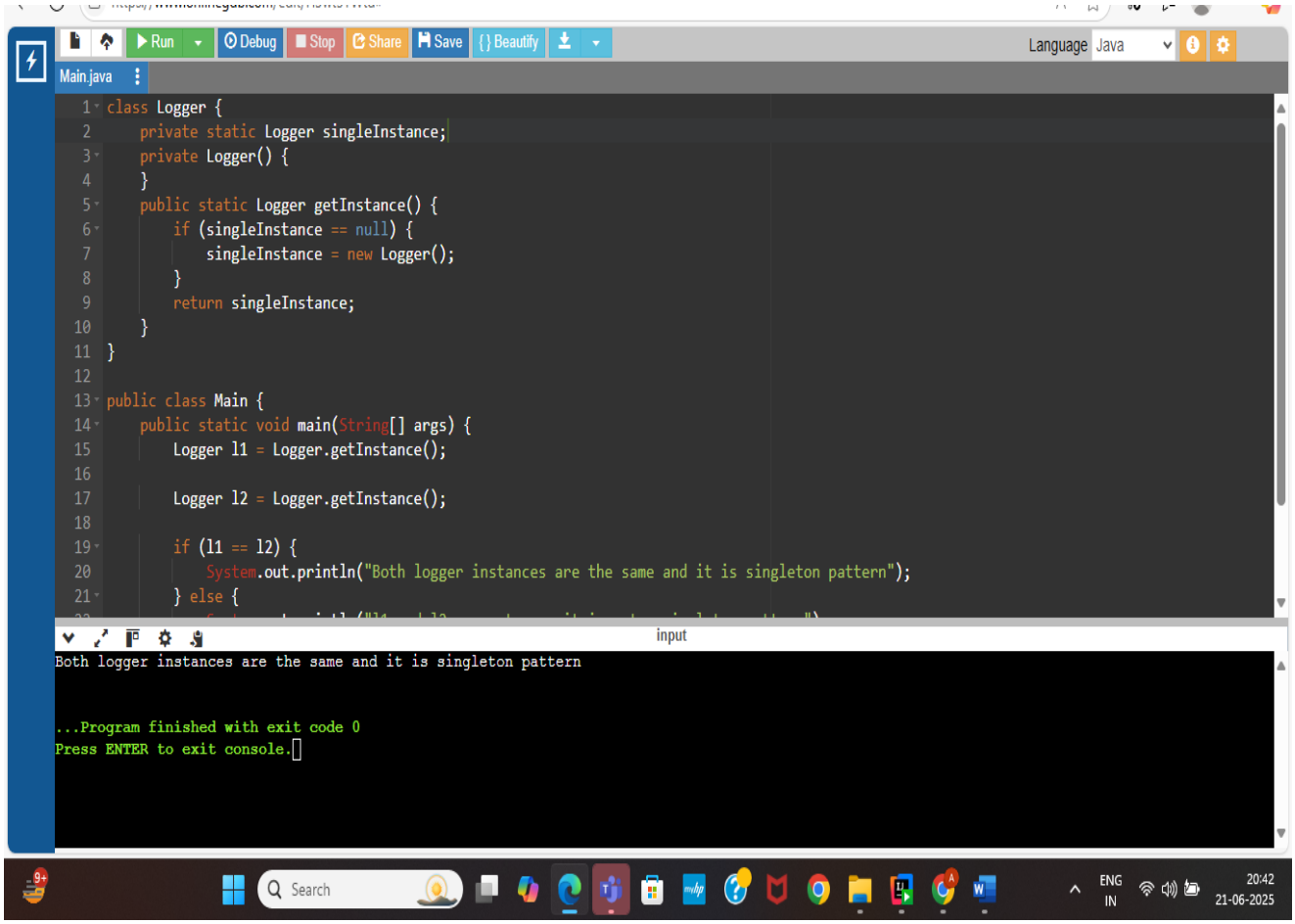
Answer

Program:

```
class Logger {
   private static Logger singleInstance;


   private Logger() {
   }


   public static Logger getInstance() {
      if (singleInstance == null) {
         singleInstance = new Logger();
      }
      return singleInstance;
   }
```

```
}

public class Main {
    public static void main(String[] args) {
        Logger l1 = Logger.getInstance();

        Logger l2 = Logger.getInstance();

        if (l1 == l2) {
            System.out.println("Both logger instances are the same and it is singleton pattern");
        } else {
            System.out.println("l1 and l2 are not same it is not a singleton pattern");
        }
    }
}
```

output:



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

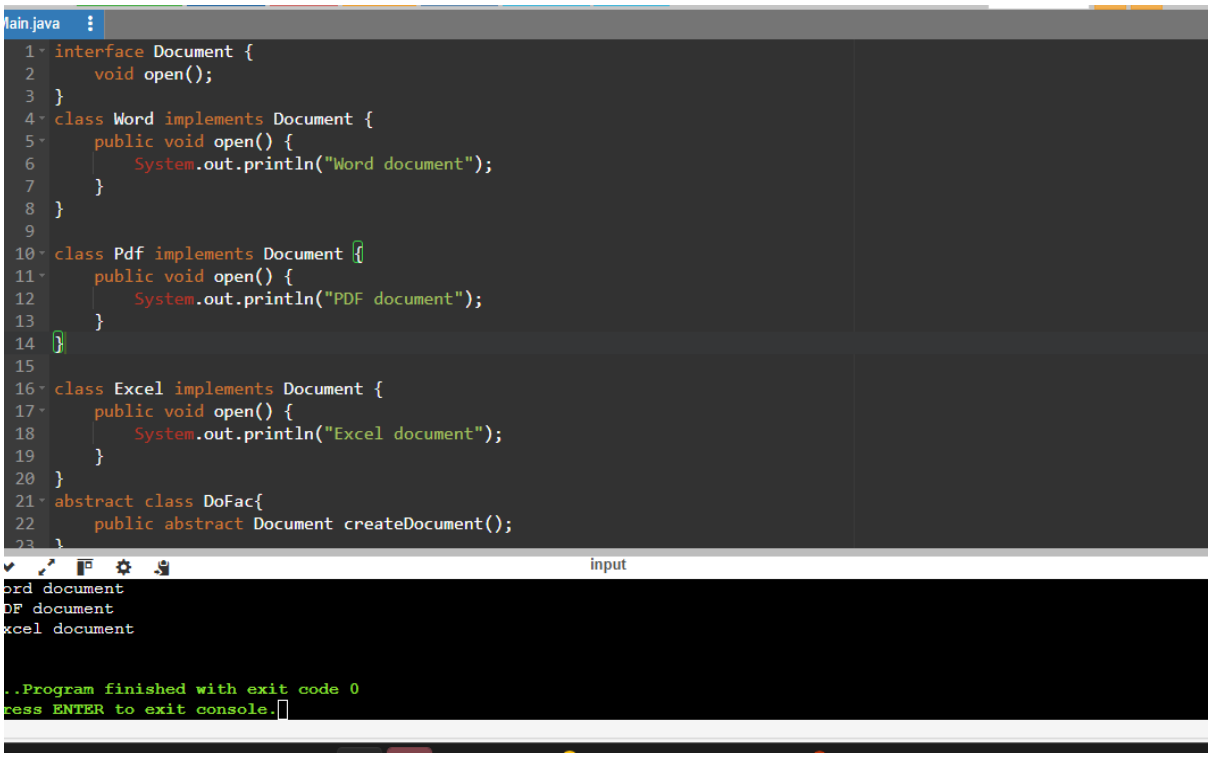
Answer:

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


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


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

```
}
abstract class DoFac{
    public abstract Document createDocument();
}

class WordFactory extends DoFac{
    public Document createDocument() {
        return new Word();
    }
}



class PdfFactory extends DoFac{
    public Document createDocument() {
        return new Pdf();
    }
}

class ExcelFactory extends DoFac{
    public Document createDocument() {
        return new Excel();
    }
}
public class Main {
    public static void main(String[] args) {
        DoFac wordFactory = new WordFactory();
        Document wordDoc = wordFactory.createDocument();
        wordDoc.open();

        DoFac pdfFactory = new PdfFactory();
        Document pdfDoc = pdfFactory.createDocument();
```

```
        pdfDoc.open();

        DoFac excelFactory = new ExcelFactory();
        Document excelDoc = excelFactory.createDocument();
        excelDoc.open();
    }
}
```

Output:



Exercise 3: 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.**

Answer:

1.

BIG O NOTATION:

Big O notation is a way to describe how fast or slow an algorithm works when the input gets bigger.

It helps us understand:

- How much time an algorithm takes
- Or how much space (memory) it uses

We don't care about the exact time.
We just want to know how the performance changes as the input grows.

Best case,Average and worst case

Best, Average, and Worst-Case Scenarios for Search Operations

When we search for an item in a list or array, the time it takes to find the item depends on where it is. There are three possible cases:

---

1. Best Case
The item is found at the beginning of the list.
It takes the least amount of time.

Example: Searching for "Amit" in ["Amit", "Bala", "John"]
Time Complexity: O(1) – very fast

---

2. Average Case
The item is found somewhere in the middle of the list.
It takes a medium amount of time.

Example: Searching for "John" in ["Amit", "Bala", "John", "Rahul"]
Time Complexity: O(n) – depends on list size

---

3. Worst Case
The item is at the end of the list or not present at all.
We have to check every item.

Example: Searching for "Zara" in ["Amit", "Bala", "John", "Rahul", "Zara"]
or searching for "Sam" which is not in the list.
Time Complexity: O(n) – slowest case

Summary Table

Scenario	Description	Time Complexity
Best Case	Found at the beginning	O(1)
Average Case	Found somewhere in the middle	O(n)
Worst Case	Found at the end or not at all	O(n)

2.

Program:

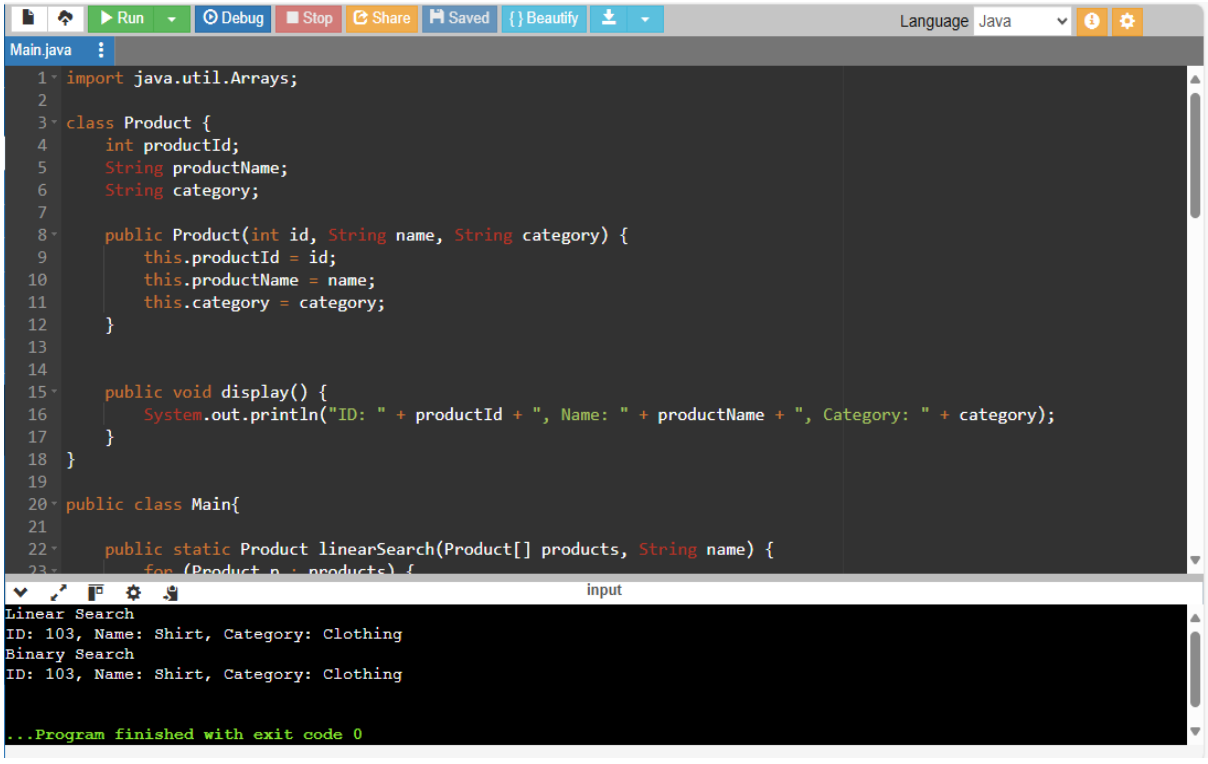
```
import java.util.Arrays;

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;
    }


    public void display() {
        System.out.println("ID: " + productId + ", Name: " + productName + ", Category: " + category);
    }
}

public class Main{

    public static Product linearSearch(Product[] products, String name) {
        for (Product p : products) {
            if (p.productName.equalsIgnoreCase(name)) {
                return p;
            }
        }
        return null;
    }


    public static Product binarySearch(Product[] products, String name) {
        int left = 0, right = products.length - 1;

        while (left <= right) {
            int mid = (left + right) / 2;
            int result = products[mid].productName.compareToIgnoreCase(name);

            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(101, "Phone", "Electronics"),
        new Product(102, "Laptop", "Electronics"),
        new Product(103, "Shirt", "Clothing"),
        new Product(104, "Book", "Stationery"),
        new Product(105, "Mouse", "Accessories")
    };



    Product l = linearSearch(products, "Shirt");
    if (l!= null) {
        l.display();
    } else {
        System.out.println("Product not found.");
    }

    Arrays.sort(products, (a, b) -> a.productName.compareToIgnoreCase(b.productName));
```

```
        Product b = binarySearch(products, "Shirt");
        if (b != null) {
            b.display();
        } else {
            System.out.println("Product not found.");
        }

    }
}
```

Output:



---

1. Compare the Time Complexity of Linear and Binary Search Algorithms

Search Type	Time Complexity (Best Case)	Time Complexity (Average & Worst Case)
Linear Search	O(1)	O(n)
Binary Search	O(1)	O(log n)

Explanation:

- Linear Search:
    - Checks each item one by one.
    - Takes more time as the number of products increases.
    - Works on unsorted arrays.
- Binary Search:
    - Divides the array in half each time (like guessing a number game).
    - Works faster but requires the array to be sorted.
    - Time grows very slowly even with a large number of products.

---

2. Which Algorithm is More Suitable for Your Platform and Why?

- If the product list is small or not sorted, use linear search. It's simple and works well.
- But if the product list is large and sorted, use binary search. It's much faster and saves time.
- So, for small or temporary data → linear search is fine.
  For big or growing data → binary search is better

Exercise 4: 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.**

Answer:

1.Recursion

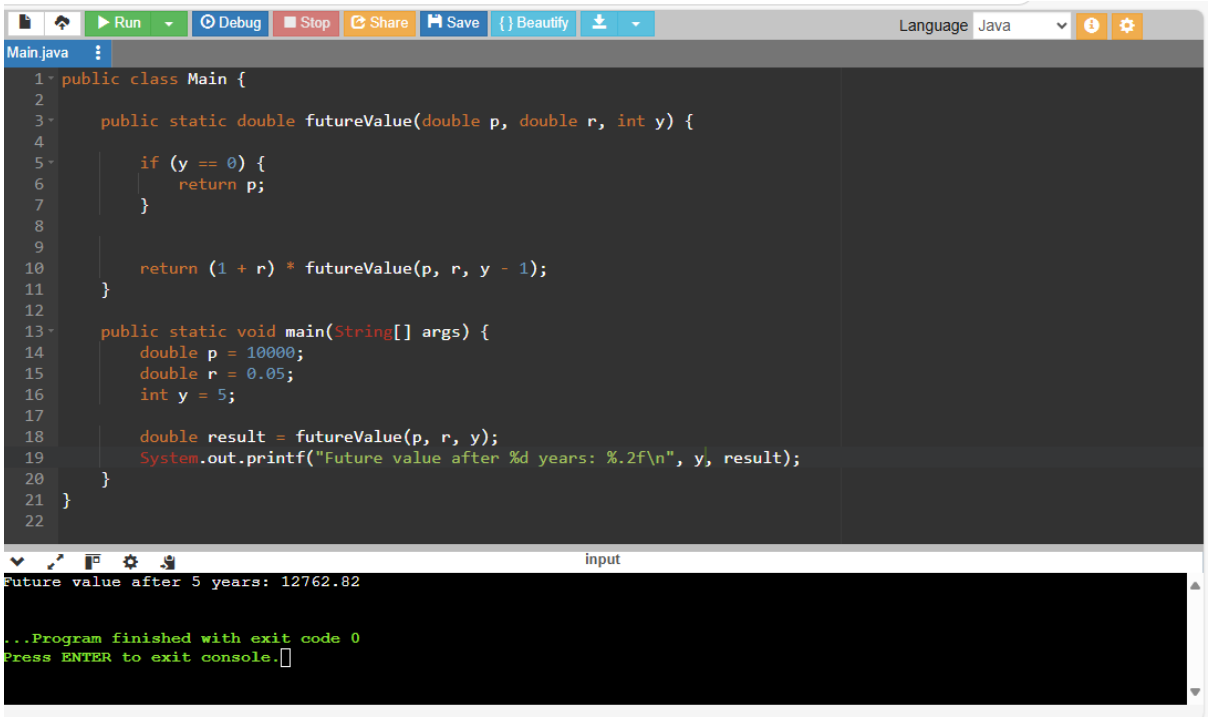
- Recursion is a way for a function to solve a problem by calling itself.
- The function keeps calling itself with smaller or simpler inputs.
- It continues doing this until it reaches a point where it stops — this is called the base case.
- After reaching the base case, the function starts going back and gives the final answer.
- It makes the code shorter and cleaner.
- It helps with problems that are built in layers or steps.
- It's easier to solve some problems with recursion than with loops.

2. Program:

```
public class Main {

    public static double futureValue(double p, double r, int y) {

        if (y == 0) {
            return p;
        }

        return (1 + r) * futureValue(p, r, y - 1);
    }

    public static void main(String[] args) {
        double p = 10000;
```

```
        double r = 0.05;
        int y = 5;

        double result = futureValue(p, r, y);
        System.out.printf("Future value after %d years: %.2f\n", y, result);
    }
}
```

Output:



4. In our recursive function, it calls itself one time for each year.

   So, if we want to calculate for 5 years, it runs 5 times.

   If we want to calculate for 100 years, it runs 100 times.

The time complexity is **O(n)**
This means the time grows **step by step** with the number of years.

Optimization:

Use memoization or iteration to improve:

- Memoization: Store results of already computed years (avoid re-computation)
- Or, convert to iterative version (better for large values)

Optimized solution:

Use iterative approach:

```
public static double futureValueIterative(double p, double r, int y) {
 for (int i = 0; i < y; i++) {
      p *= (1 + r);
 }
  return p;
}
```