**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.

**Solution:**

**Code:**

public class Logger {

private static Logger instance;

private Logger() {

System.out.println("Logger Initialized");

}

public static Logger getInstance() {

if (instance == null) {

instance = new Logger();

}

return instance;

}

public void log(String message) {

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

}

}

public class Main {

public static void main(String[] args) {

Logger logger1 = Logger.getInstance();

logger1.log("First log message");

Logger logger2 = Logger.getInstance();

logger2.log("Second log message");

if (logger1 == logger2) {

System.out.println("Logger is a Singleton. Ore object dha use panrom.");

} else {

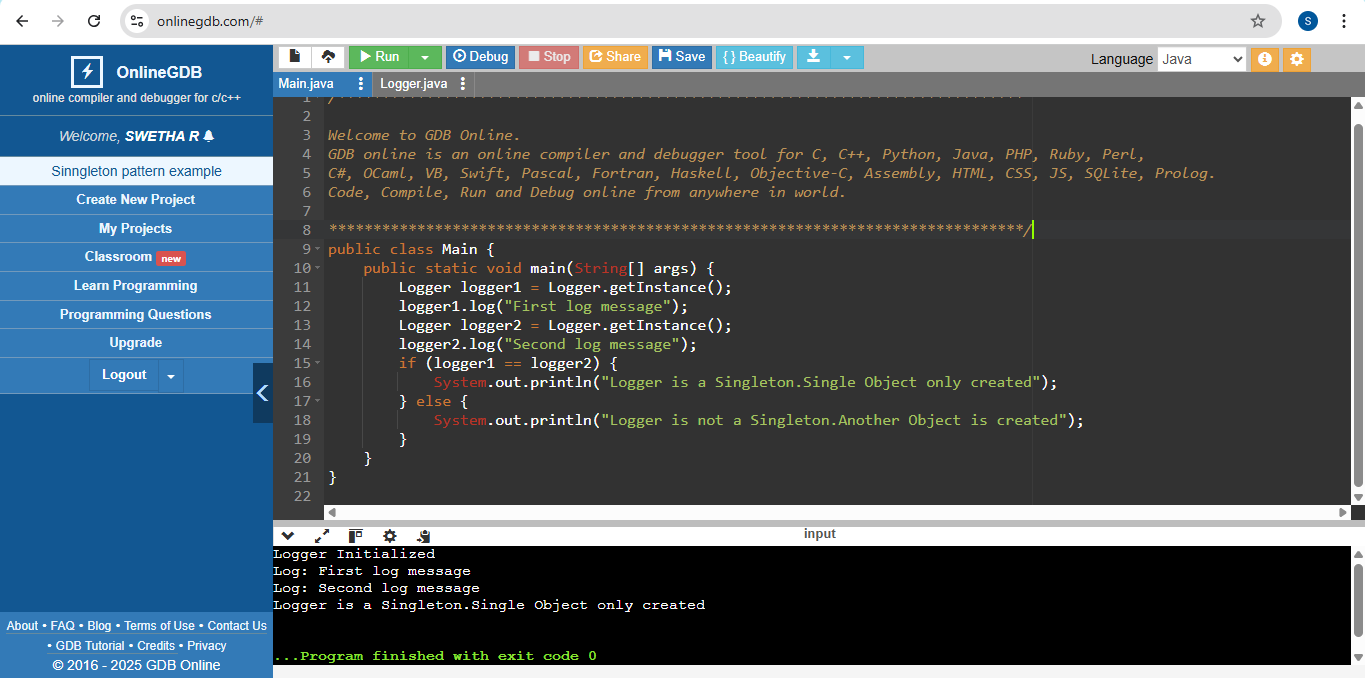
System.out.println("Logger is not a Singleton. Vera object create panniduchu.");

}

}

}

**Result:**



**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.

**Solution:**

**Code:**

public interface Document {

void open();

}

public class WordDocument implements Document {

public void open() {

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

}

}

public class PdfDocument implements Document {

public void open() {

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

}

}

public class ExcelDocument implements Document {

public void open() {

System.out.println("Opening Excel Document");

}

}

public abstract class DocumentFactory {

public abstract Document createDocument();

}

public class WordDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new WordDocument();

}

}

public class PdfDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new PdfDocument();

}

}

public class ExcelDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new ExcelDocument();

}

}

public class Main {

public static void main(String[] args) {

DocumentFactory wordFactory = new WordDocumentFactory();

//WordDocumentFactory

Document wordDoc = wordFactory.createDocument();

wordDoc.open();

//PdfDocumentFactory

DocumentFactory pdfFactory = new PdfDocumentFactory();

Document pdfDoc = pdfFactory.createDocument();

pdfDoc.open();

//ExcelDocumentFactory

DocumentFactory excelFactory = new ExcelDocumentFactory();

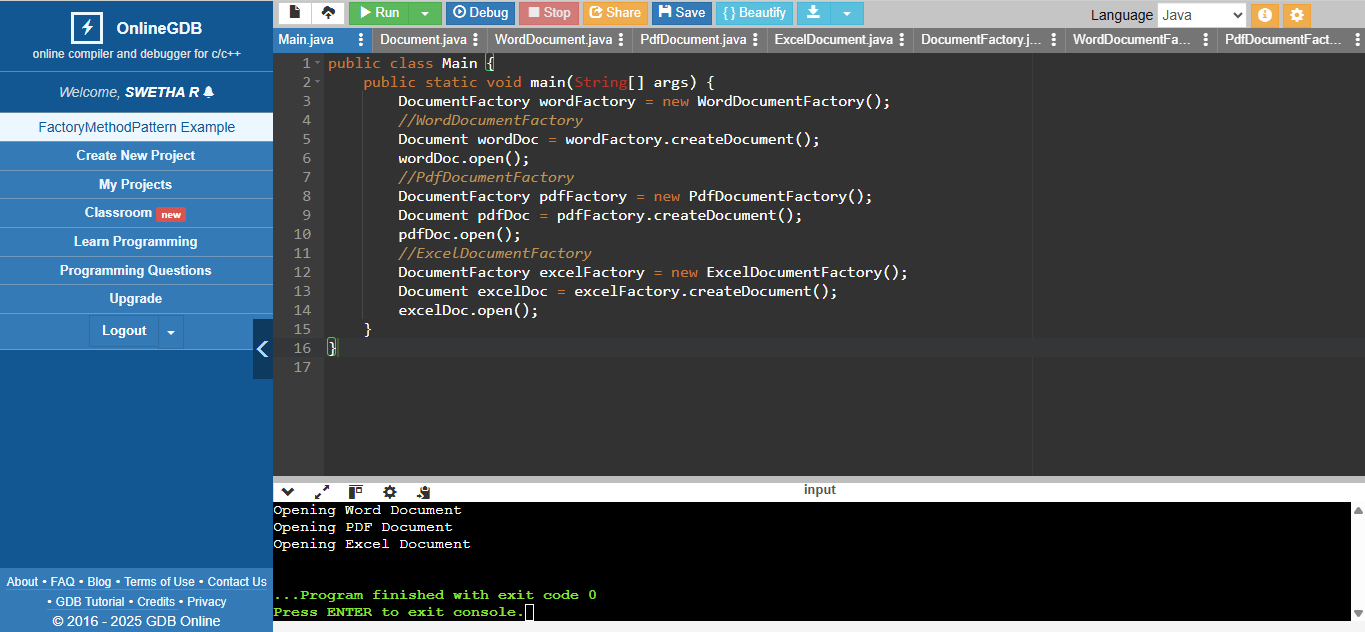
Document excelDoc = excelFactory.createDocument();

excelDoc.open();

}

}

**Result:**



**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.

**Solution:**

**Code:**

import java.util.Arrays;

import java.util.Comparator;

public class Main {

public static void main(String[] args) {

// Sample products

Product[] products = {

new Product(1, "Laptop", "Electronics"),

new Product(2, "Phone", "Electronics"),

new Product(3, "Shirt", "Clothing"),

new Product(4, "Book", "Stationery"),

new Product(5, "Watch", "Accessories")

};

// Linear Search

Product result1 = SearchAlgorithms.linearSearch(products, "Phone");

System.out.println("Linear Search Result: " + (result1 != null ? result1 : "Product not found"));

// Sort products for binary search (Sort by product name)

Arrays.sort(products, new Comparator<Product>() {

//@Override

public int compare(Product a, Product b) {

return a.getProductName().compareToIgnoreCase(b.getProductName());

}

});

// Binary Search

Product result2 = SearchAlgorithms.binarySearch(products, "Phone");

System.out.println("Binary Search Result: " + (result2 != null ? result2 : "Product not found"));

}

}

public class Product {

private int productId;

private String productName;

private String category;

public Product(int productId, String productName, String category) {

this.productId = productId;

this.productName = productName;

this.category = category;

}

public int getProductId() {

return productId;

}

public String getProductName() {

return productName;

}

public String getCategory() {

return category;

}

public String toString() {

return "Product ID: " + productId + ", Name: " + productName + ", Category: " + category;

}

}

import java.util.Arrays;

public class SearchAlgorithms {

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

for (Product product : products) {

if (product.getProductName().equalsIgnoreCase(productName)) {

return product;

}

}

return null;

}

public static Product binarySearch(Product[] products, String productName) {

int left = 0;

int right = products.length - 1;

while (left <= right) {

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

int compare = products[mid].getProductName().compareToIgnoreCase(productName);

if (compare == 0) {

return products[mid];

} else if (compare < 0) {

left = mid + 1;

} else {

right = mid - 1;

}

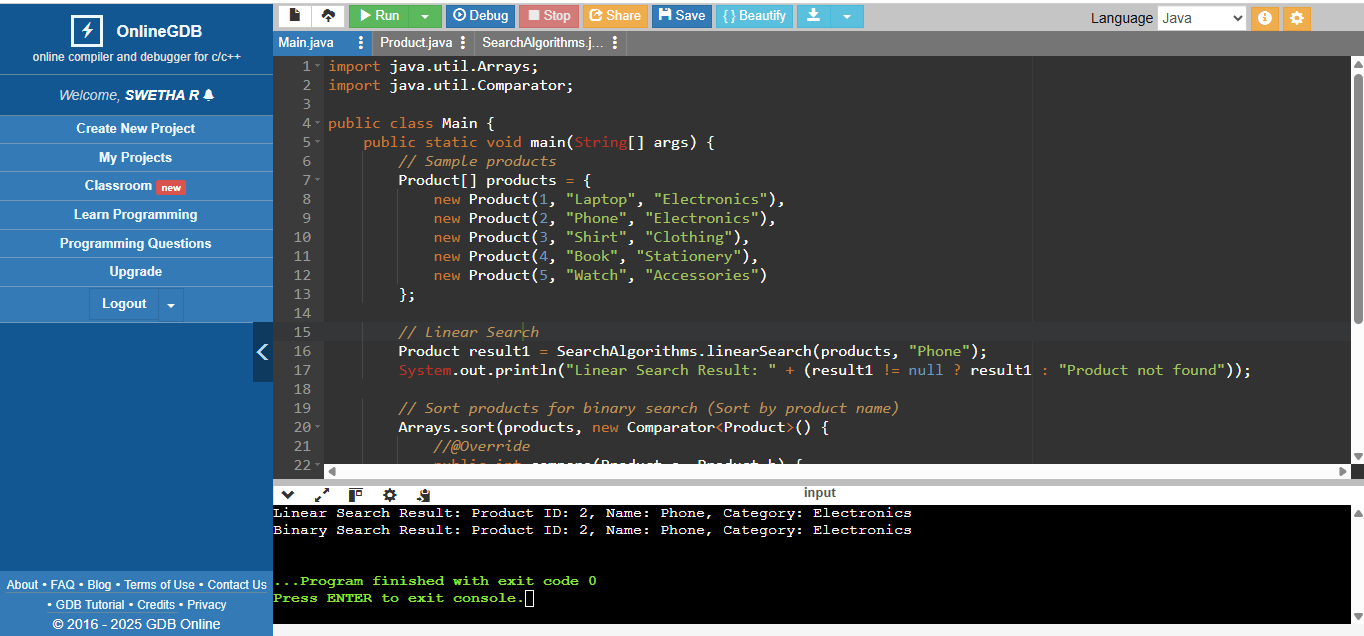
}

return null;

}

}

**Result:**



**Analysis Result:**

**Compare Linear and Binary Search**

1. **Linear Search**:
   * Time Complexity: O(n)
   * Use case: Works on **unsorted data** and is easy to implement.
2. **Binary Search**:
   * Time Complexity: O(log n)
   * Use case: Works on **sorted data** and is faster for large datasets.

**Which is Better?**

* **Binary Search** is more efficient but requires data to be sorted.
* If your platform frequently adds products, sorting might be expensive, making **Linear Search** suitable for small datasets.

**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.

**Solution:**

**Code:**

public class FinancialForecasting {

// Recursive method to calculate future value

public static double calculateFutureValue(double presentValue, double growthRate, int periods) {

// Base case: No periods left

if (periods == 0) {

return presentValue;

}

// Recursive case: Reduce periods and apply growth rate

return calculateFutureValue(presentValue, growthRate, periods - 1) \* (1 + growthRate);

}

public static void main(String[] args) {

// Example data

double presentValue = 20000; // Initial investment

double growthRate = 0.06; // 6% annual growth rate

int periods = 5; // Predict for 5 years

// Calculate future value using recursion

double futureValue = calculateFutureValue(presentValue, growthRate, periods);

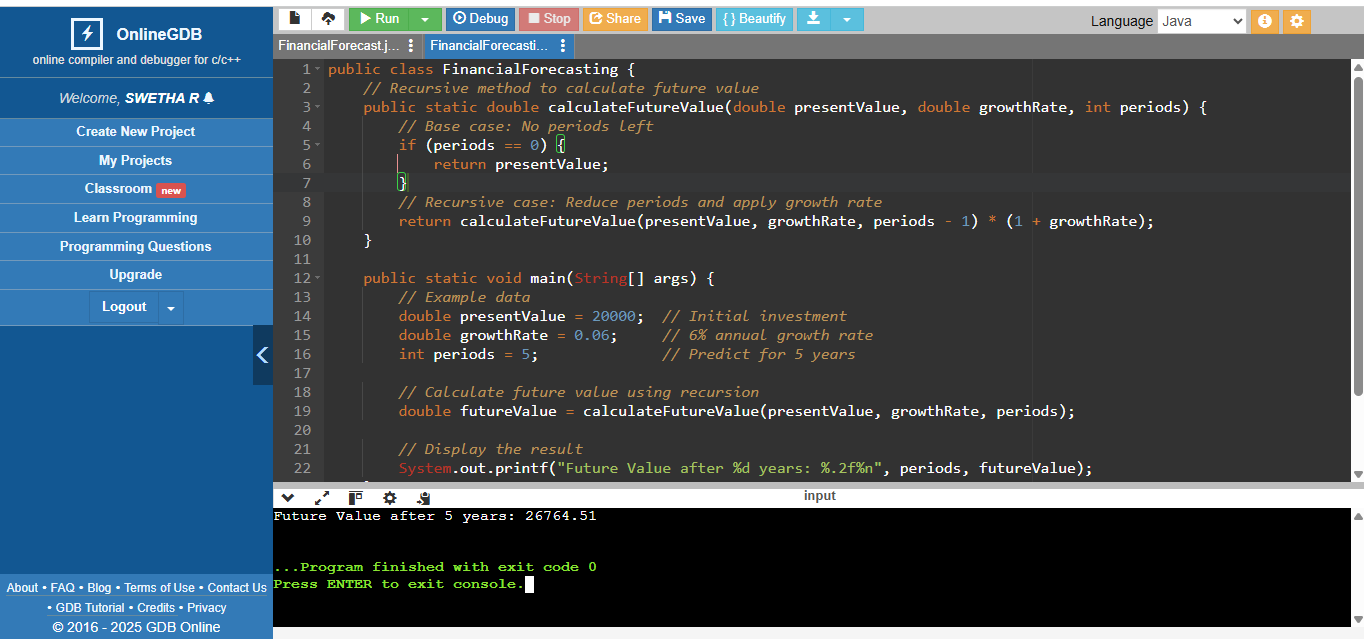
// Display the result

System.out.printf("Future Value after %d years: %.2f%n", periods, futureValue);

}

}

**Result:**



**2,Optimized Method:**

**Code:**

import java.util.HashMap;

public class FinancialForecast {

// Recursive method with memoization

public static double calculateFutureValueOptimized(double presentValue, double growthRate, int periods, HashMap<Integer, Double> memo) {

// Check if result is already computed

if (memo.containsKey(periods)) {

return memo.get(periods);

}

// Base case

if (periods == 0) {

return presentValue;

}

// Recursive case

double result = calculateFutureValueOptimized(presentValue, growthRate, periods - 1, memo) \* (1 + growthRate);

// Store result in memoization map

memo.put(periods, result);

return result;

}

public static void main(String[] args) {

// data

double presentValue = 20000; // Initial investment

double growthRate = 0.06; // 6% annual growth rate

int periods = 10; // Predict for 5 years

// Memoization map

HashMap<Integer, Double> memo = new HashMap<>();

// Calculate future value using optimized recursion

double futureValue = calculateFutureValueOptimized(presentValue, growthRate, periods, memo);

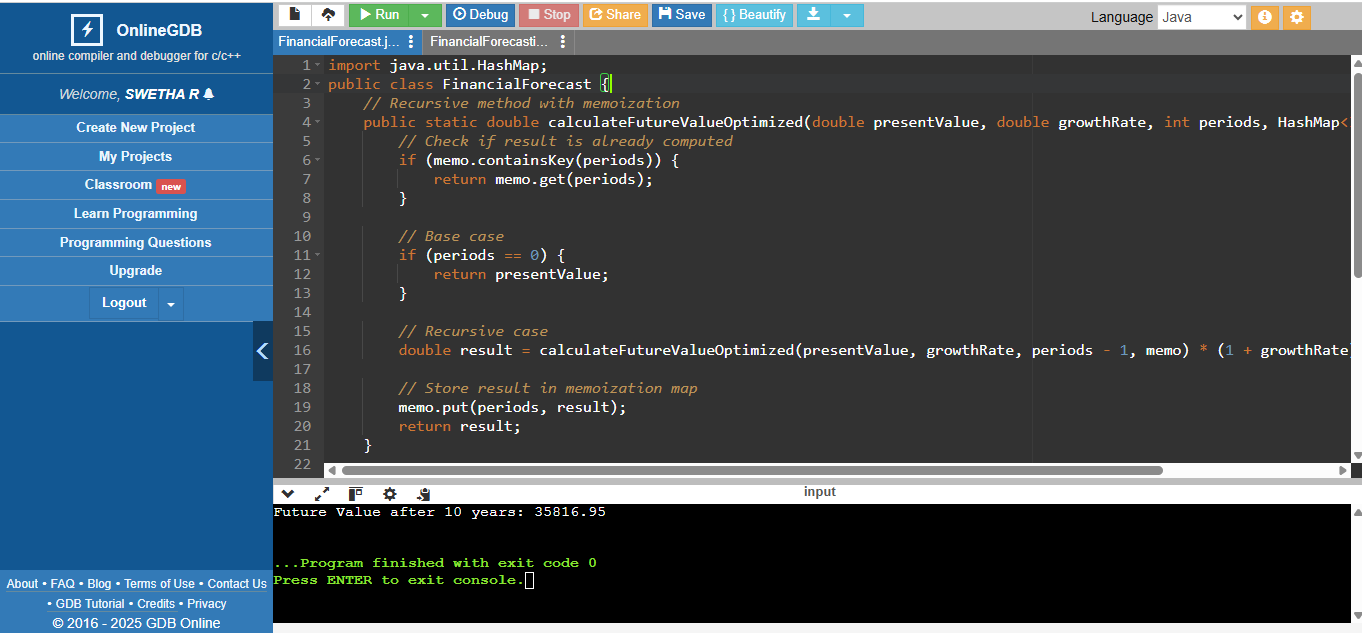
// Display the result

System.out.printf("Future Value after %d years: %.2f%n", periods, futureValue);

}

}

**Result:**



**Analysis Result:**

#### ****Non-Optimized Recursive Method****

1. **How it works**:
   * Recalculates values for each period independently.
   * Overlapping sub-problems lead to redundant computations.
2. **Complexity**:
   * **Time Complexity**: O(n)(but with redundant calculations).
   * **Space Complexity**: O(n)(due to stack usage).
3. **Drawbacks**:
   * Inefficient for large inputs.
   * Risk of stack overflow for deep recursion.

#### ****Optimized Recursive Method (Memoization)****

1. **How it works**:
   * Uses a HashMap to store intermediate results.
   * Reuses stored values instead of recalculating.
2. **Complexity**:
   * **Time Complexity**: O(n) (each period is calculated once).
   * **Space Complexity**: O(n)(stack + memoization storage).
3. **Benefits**:
   * Eliminates redundant calculations.
   * Efficient for large inputs, avoids stack overflow.