**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:**

**o Create a new Java project named SingletonPatternExample.**

**2. Define a Singleton Class:**

**o Create a class named Logger that has a private static instance of itself.**

**o Ensure the constructor of Logger is private.**

**o Provide a public static method to get the instance of the Logger class.**

**3. Implement the Singleton Pattern:**

**o Write code to ensure that the Logger class follows the Singleton design pattern.**

**4. Test the Singleton Implementation:**

**o Create a test class to verify that only one instance of Logger is created and used across the application.**

**SOLUTION:**

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();

Logger logger2 = Logger.getInstance();

logger1.log("This is the first log message.");

logger2.log("This is the second log message.");

if (logger1 == logger2) {

System.out.println("Both logger1 and logger2 are the same instance.");

} else {

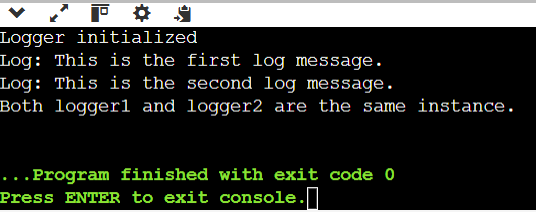
System.out.println("Different instances exist! Singleton failed.");

}

}

}

**OUTPUT SCREENSHOT:**

****

**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:**

**o Create a new Java project named FactoryMethodPatternExample.**

**2. Define Document Classes:**

**o Create interfaces or abstract classes for different document types such as WordDocument, PdfDocument, and ExcelDocument.**

**3. Create Concrete Document Classes:**

**o Implement concrete classes for each document type that implements or extends the above interfaces or abstract classes.**

**4. Implement the Factory Method:**

**o Create an abstract class DocumentFactory with a method createDocument().**

**o Create concrete factory classes for each document type that extends DocumentFactory and implements the createDocument() method.**

**5. Test the Factory Method Implementation:**

**o Create a test class to demonstrate the creation of different document types using the factory method.**

**SOLUTION:**

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();

}

}

public class Main {

public static void main(String[] args) {

DocumentFactory wordFactory = new WordDocumentFactory();

Document wordDoc = wordFactory.createDocument();

wordDoc.open();

DocumentFactory pdfFactory = new PdfDocumentFactory();

Document pdfDoc = pdfFactory.createDocument();

pdfDoc.open();

DocumentFactory excelFactory = new ExcelDocumentFactory();

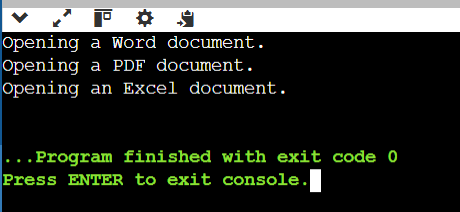
Document excelDoc = excelFactory.createDocument();

excelDoc.open();

}

}

**OUTPUT SCREENSHOT:**

****

**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:**

import java.util.Arrays;

import java.util.Comparator;

class 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 "ID: " + productId + ", Name: " + productName + ", Category: " + category;

}

}

public class Main {

// Linear Search

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

for (Product product : products) {

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

return product;

}

}

return null;

}

// Binary Search (requires sorted array)

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

Arrays.sort(products, Comparator.comparing(p -> p.productName.toLowerCase())); // sort first

int low = 0;

int high = products.length - 1;

while (low <= high) {

int mid = (low + high) / 2;

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

if (cmp == 0) return products[mid];

else if (cmp < 0) low = mid + 1;

else high = mid - 1;

}

return null;

}

public static void main(String[] args) {

Product[] products = {

new Product(101, "Laptop", "Electronics"),

new Product(102, "Shirt", "Clothing"),

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

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

new Product(105, "Watch", "Accessories")

};

System.out.println(" Linear Search for 'Phone':");

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

System.out.println(result1 != null ? result1 : "Not Found");

System.out.println("\n Binary Search for 'Phone':");

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

System.out.println(result2 != null ? result2 : "Not Found");

}

}

**OUTPUT SCREENSHOT:**

**A screenshot of a computer

AI-generated content may be incorrect.**

**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:**

public class Main {

// Recursive method to calculate future value

public static double futureValue(double amount, double growthRate, int years) {

if (years == 0) {

return amount;

}

return futureValue(amount, growthRate, years - 1) \* (1 + growthRate);

}

// Iterative version (for comparison)

public static double futureValueIterative(double amount, double growthRate, int years) {

for (int i = 0; i < years; i++) {

amount \*= (1 + growthRate);

}

return amount;

}

public static void main(String[] args) {

// Input values

double initialAmount = 10000; // Starting amount in Rs

double growthRate = 0.05; // 5% annual growth

int years = 5; // Forecast for 5 years

// Calculate using recursion

double resultRecursive = futureValue(initialAmount, growthRate, years);

System.out.printf("Future Value after %d years (Recursive): %.2f\n", years, resultRecursive);

// Calculate using iteration

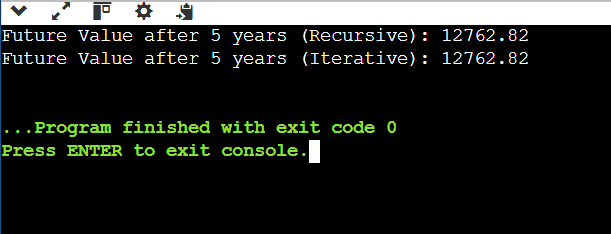
double resultIterative = futureValueIterative(initialAmount, growthRate, years);

System.out.printf("Future Value after %d years (Iterative): %.2f\n", years, resultIterative);

}

}

**OUTPUT SCREENSHOT:**

****