**CTS WEEK 1:**

**MODULE 1: Design patterns & 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.

**SOLUTION:**

package SingletonPatternExample;

class Logger {

private static Logger instance;

private Logger() {

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

}

public static Logger getInstance() {

if (instance == null) {

synchronized (Logger.class) {

if (instance == null) {

instance = new Logger();

}

}

}

return instance;

}

public void log(String message) {

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

}

}

public class SingletonPatternExample {

public static void main(String[] args) {

Logger logger1 = Logger.getInstance();

Logger logger2 = Logger.getInstance();

System.out.println("Are logger1 and logger2 the same instance? " + (logger1 == logger2));

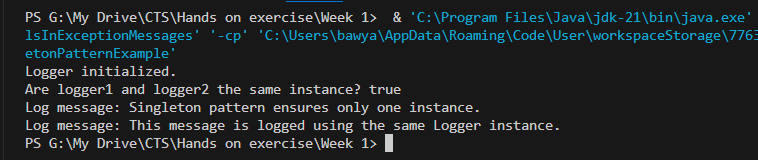
logger1.log("Singleton pattern ensures only one instance.");

logger2.log("This message is logged using the same Logger instance.");

}

}

**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.

**SOLUTION:**

package factorymethodpatternexample;

interface Document {

void open();

}

class WordDocument implements Document {

@Override

public void open() {

System.out.println("Opening a Word document.");

}

}

class PdfDocument implements Document {

@Override

public void open() {

System.out.println("Opening a PDF document.");

}

}

class ExcelDocument implements Document {

@Override

public void open() {

System.out.println("Opening an Excel document.");

}

}

abstract class DocumentFactory {

public abstract Document createDocument();

}

class WordDocumentFactory extends DocumentFactory {

@Override

public Document createDocument() {

return new WordDocument();

}

}

class PdfDocumentFactory extends DocumentFactory {

@Override

public Document createDocument() {

return new PdfDocument();

}

}

class ExcelDocumentFactory extends DocumentFactory {

@Override

public Document createDocument() {

return new ExcelDocument();

}

}

public 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();

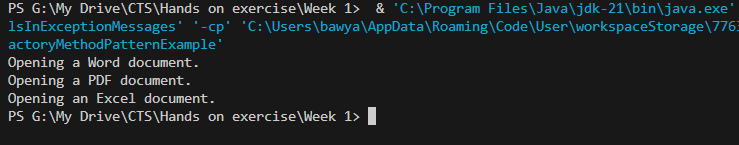
pdfDoc.open();

excelDoc.open();

}

}

**OUTPUT:**

****

**MODULE 2: Data Structures & Algorithms**

**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:**

package ecommerceplatform;

import java.util.Arrays;

import java.util.Scanner;

class Product implements Comparable<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;

}

@Override

public String toString() {

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

}

@Override

public int compareTo(Product other) {

return Integer.compare(this.productId, other.productId);

}

}

public class ECommerceSearch {

public static Product linearSearch(Product[] products, int productId) {

for (Product product : products) {

if (product.getProductId() == productId) {

return product;

}

}

return null;

}

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

int low = 0;

int high = products.length - 1;

while (low <= high) {

int mid = low + (high - low) / 2;

if (products[mid].getProductId() == productId) {

return products[mid];

} else if (products[mid].getProductId() < productId) {

low = mid + 1;

} else {

high = mid - 1;

}

}

return null;

}

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

Product[] products = {

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

new Product(205, "T-shirt", "Apparel"),

new Product(303, "Refrigerator", "Home Appliance"),

new Product(150, "Mobile Phone", "Electronics"),

new Product(420, "Microwave", "Home Appliance")

};

Product[] sortedProducts = products.clone();

Arrays.sort(sortedProducts);

System.out.print("Enter Product ID to search: ");

int searchId = scanner.nextInt();

System.out.println("\n--- Linear Search ---");

long startLinear = System.nanoTime();

Product linearResult = linearSearch(products, searchId);

long endLinear = System.nanoTime();

if (linearResult != null) {

System.out.println("Product Found: " + linearResult);

} else {

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

}

System.out.println("Time taken: " + (endLinear - startLinear) + " nanoseconds");

System.out.println("\n--- Binary Search ---");

long startBinary = System.nanoTime();

Product binaryResult = binarySearch(sortedProducts, searchId);

long endBinary = System.nanoTime();

if (binaryResult != null) {

System.out.println("Product Found: " + binaryResult);

} else {

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

}

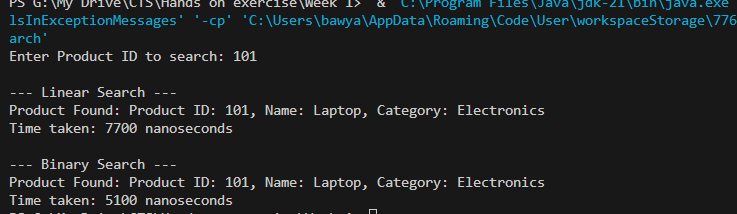
System.out.println("Time taken: " + (endBinary - startBinary) + " nanoseconds");

scanner.close();

}

}

**OUTPUT:**

****

**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:**

package financialforecasting;

public class FinancialForecasting {

public static double calculateFutureValueRecursive(double presentValue, double growthRate, int years) {

if (years == 0) {

return presentValue;

}

return calculateFutureValueRecursive(presentValue \* (1 + growthRate), growthRate, years - 1);

}

public static double calculateFutureValueOptimized(double presentValue, double growthRate, int years) {

double[] memo = new double[years + 1];

return calculateWithMemoization(presentValue, growthRate, years, memo);

}

private static double calculateWithMemoization(double presentValue, double growthRate, int years, double[] memo) {

if (years == 0) {

return presentValue;

}

if (memo[years] != 0) {

return memo[years];

}

memo[years] = calculateWithMemoization(presentValue \* (1 + growthRate), growthRate, years - 1, memo);

return memo[years];

}

public static void main(String[] args) {

double presentValue = 1000;

double growthRate = 0.05;

int years = 10;

double futureValueRecursive = calculateFutureValueRecursive(presentValue, growthRate, years);

System.out.println("Future Value (Recursive): " + futureValueRecursive);

double futureValueOptimized = calculateFutureValueOptimized(presentValue, growthRate, years);

System.out.println("Future Value (Optimized): " + futureValueOptimized);

}

}

**OUTPUT:**

