**Week 1 hands on exercise solutions**

**Design patterns 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.

**Solution:**

**Code:**

Logger.java

package sp;

public class Logger {

// Private static instance of itself

private static Logger *instance*;

// Ensure the constructor of Logger is private

private Logger() {

// Private constructor to prevent instantiation from outside

System.***out***.println("Logger instance created.");

}

// Provide a public static method to get the instance of the Logger class

public static Logger getInstance() {

if (*instance* == null) {

// Synchronize to prevent multiple threads from creating multiple instances

synchronized (Logger.class) {

if (*instance* == null) {

*instance* = new Logger();

}

}

}

return *instance*;

}

public void log(String message) {

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

}

}

SingletonTest.java

package sp;

public class SingletonTest {

public static void main(String[] args) {

// Get the first instance of Logger

Logger logger1 = Logger.*getInstance*();

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

// Get the second instance of Logger

Logger logger2 = Logger.*getInstance*();

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

// Get the third instance of Logger

Logger logger3 = Logger.*getInstance*();

logger3.log("This is the third log message.");

// Verify that all references point to the same instance

System.***out***.println("\n--- Instance Verification ---");

System.***out***.println("logger1 == logger2: " + (logger1 == logger2));

System.***out***.println("logger1 == logger3: " + (logger1 == logger3));

System.***out***.println("logger2 == logger3: " + (logger2 == logger3));

// Demonstrate hash codes to further verify they are the same object

System.***out***.println("logger1 hash code: " + logger1.hashCode());

System.***out***.println("logger2 hash code: " + logger2.hashCode());

System.***out***.println("logger3 hash code: " + logger3.hashCode());

}

}

Output:

Logger instance created.

Log Message: This is the first log message.

Log Message: This is the second log message.

Log Message: This is the third log message.

--- Instance Verification ---

logger1 == logger2: true

logger1 == logger3: true

logger2 == logger3: true

logger1 hash code: 920011586

logger2 hash code: 920011586

logger3 hash code: 920011586

**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:**

Document.java

package fp;

public interface Document {

void open();

void save();

void close();

}

WordDocument.java

package fp;

//Step 3: Concrete Word Document

public class WordDocument implements Document {

*@Override*

public void open() {

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

}

*@Override*

public void save() {

System.***out***.println("Saving Word Document...");

}

*@Override*

public void close() {

System.***out***.println("Closing Word Document.");

}

}

PdfDocument.java

package fp;

//Step 3: Concrete PDF Document

public class PdfDocument implements Document {

*@Override*

public void open() {

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

}

*@Override*

public void save() {

System.***out***.println("Saving PDF Document...");

}

*@Override*

public void close() {

System.***out***.println("Closing PDF Document.");

}

}

ExcelDocument.java

package fp;

//Step 3: Concrete Excel Document

public class ExcelDocument implements Document {

*@Override*

public void open() {

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

}

*@Override*

public void save() {

System.***out***.println("Saving Excel Document...");

}

*@Override*

public void close() {

System.***out***.println("Closing Excel Document.");

}

}

DocumentFactory.java

package fp;

//Step 4: Abstract Document Factory

public abstract class DocumentFactory {

// The factory method (abstract)

public abstract Document createDocument();

// You can also add other methods that operate on the created document

public void processDocument() {

Document document = createDocument(); // Call the factory method

document.open();

document.save();

document.close();

System.***out***.println("Document processing complete.\n");

}

}

WordDocumentFactory.java

package fp;

//Step 4: Concrete Factory for Word Documents

public class WordDocumentFactory extends DocumentFactory {

*@Override*

public Document createDocument() {

System.***out***.println("Creating Word Document via factory.");

return new WordDocument();

}

}

PdfDocumentFactory.java

package fp;

//Step 4: Concrete Factory for PDF Documents

public class PdfDocumentFactory extends DocumentFactory {

*@Override*

public Document createDocument() {

System.***out***.println("Creating PDF Document via factory.");

return new PdfDocument();

}

}

ExcelDocumentFactory.java

package fp;

//Step 4: Concrete Factory for Excel Documents

public class ExcelDocumentFactory extends DocumentFactory {

*@Override*

public Document createDocument() {

System.***out***.println("Creating Excel Document via factory.");

return new ExcelDocument();

}

}

FactoryMethodTest.java

package fp;

// Step 5: Test the Factory Method Implementation

public class FactoryMethodTest {

public static void main(String[] args) {

System.***out***.println("--- Creating Word Document ---");

DocumentFactory wordFactory = new WordDocumentFactory();

wordFactory.processDocument(); // Uses createDocument() internally

System.***out***.println("--- Creating PDF Document ---");

DocumentFactory pdfFactory = new PdfDocumentFactory();

pdfFactory.processDocument();

System.***out***.println("--- Creating Excel Document ---");

DocumentFactory excelFactory = new ExcelDocumentFactory();

excelFactory.processDocument();

// Direct creation using createDocument()

System.***out***.println("--- Creating another Word Document directly ---");

Document anotherWordDoc = wordFactory.createDocument();

anotherWordDoc.open();

anotherWordDoc.save();

anotherWordDoc.close();

System.***out***.println("Another Word Document operations complete.\n");

}

}

Output:

--- Creating Word Document ---

Creating Word Document via factory.

Opening Word Document...

Saving Word Document...

Closing Word Document.

Document processing complete.

--- Creating PDF Document ---

Creating PDF Document via factory.

Opening PDF Document...

Saving PDF Document...

Closing PDF Document.

Document processing complete.

--- Creating Excel Document ---

Creating Excel Document via factory.

Opening Excel Document...

Saving Excel Document...

Closing Excel Document.

Document processing complete.

--- Creating another Word Document directly ---

Creating Word Document via factory.

Opening Word Document...

Saving Word Document...

Closing Word Document.

Another Word Document operations complete.

**Data Structures and 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:**

**Code:**

Product.java

package dsa;

public class Product {

private String productId;

private String productName;

private String category;

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

this.productId = productId;

this.productName = productName;

this.category = category;

}

// Getters

public String 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;

}

}

SearchAlgorithms.java

package dsa;

import java.util.ArrayList;

import java.util.Collections;

import java.util.Comparator;

import java.util.List;

public class SearchAlgorithms {

public static Product linearSearch(List<Product> products, String searchAttribute, String searchValue) {

for (Product product : products) {

switch (searchAttribute) {

case "productId":

if (product.getProductId().equals(searchValue)) {

return product;

}

break;

case "productName":

if (product.getProductName().equals(searchValue)) {

return product;

}

break;

case "category":

if (product.getCategory().equals(searchValue)) {

return product;

}

break;

default:

throw new IllegalArgumentException("Invalid search attribute: " + searchAttribute);

}

}

return null; // Product not found

}

public static Product binarySearch(List<Product> sortedProducts, String searchAttribute, String searchValue) {

if (sortedProducts == null || sortedProducts.isEmpty()) {

return null;

}

if (!"productId".equals(searchAttribute)) {

throw new IllegalArgumentException("Binary search currently only supported for 'productId'.");

}

int low = 0;

int high = sortedProducts.size() - 1;

while (low <= high) {

int mid = low + (high - low) / 2; // Avoid potential integer overflow

Product midProduct = sortedProducts.get(mid);

int comparison = midProduct.getProductId().compareTo(searchValue);

if (comparison == 0) {

return midProduct; // Found

} else if (comparison < 0) {

low = mid + 1; // Search in the right half

} else {

high = mid - 1; // Search in the left half

}

}

return null; // Product not found

}

public static void main(String[] args) {

// 1. Create Product instances

List<Product> allProducts = new ArrayList<>();

allProducts.add(new Product("P001", "Laptop Pro", "Electronics"));

allProducts.add(new Product("P005", "Wireless Mouse", "Electronics"));

allProducts.add(new Product("P010", "Mechanical Keyboard", "Electronics"));

allProducts.add(new Product("P015", "Webcam HD", "Accessories"));

allProducts.add(new Product("P020", "Monitor 4K", "Electronics"));

allProducts.add(new Product("P025", "USB Hub", "Accessories"));

allProducts.add(new Product("P030", "Ergonomic Chair", "Furniture"));

System.***out***.println("--- Linear Search Examples ---");

// Linear Search by Product ID

Product foundProductId = SearchAlgorithms.*linearSearch*(allProducts, "productId", "P015");

System.***out***.println("Linear Search (ID P015): " + (foundProductId != null ? foundProductId : "Not Found"));

Product foundProductIdNotFound = SearchAlgorithms.*linearSearch*(allProducts, "productId", "P999");

System.***out***.println("Linear Search (ID P999 - Not Found): " + (foundProductIdNotFound != null ? foundProductIdNotFound : "Not Found"));

// Linear Search by Product Name

Product foundProductName = SearchAlgorithms.*linearSearch*(allProducts, "productName", "Monitor 4K");

System.***out***.println("Linear Search (Name Monitor 4K): " + (foundProductName != null ? foundProductName : "Not Found"));

// Linear Search by Category

Product foundProductCategory = SearchAlgorithms.*linearSearch*(allProducts, "category", "Furniture");

System.***out***.println("Linear Search (Category Furniture): " + (foundProductCategory != null ? foundProductCategory : "Not Found"));

<Product> sortedProductsById = new ArrayList<>(allProducts); // Create a copy to sort

Collections.*sort*(sortedProductsById, Comparator.*comparing*(Product::getProductId));

System.***out***.println("\n--- Binary Search Examples (Sorted by Product ID) ---");

System.***out***.println("Sorted Products: " + sortedProductsById);

// Binary Search by Product ID

Product foundProductBinaryId = SearchAlgorithms.*binarySearch*(sortedProductsById, "productId", "P015");

System.***out***.println("Binary Search (ID P015): " + (foundProductBinaryId != null ? foundProductBinaryId : "Not Found"));

Product foundProductBinaryIdNotFound = SearchAlgorithms.*binarySearch*(sortedProductsById, "productId", "P008");

System.***out***.println("Binary Search (ID P008 - Not Found): " + (foundProductBinaryIdNotFound != null ? foundProductBinaryIdNotFound : "Not Found"));

Product foundProductBinaryIdFirst = SearchAlgorithms.*binarySearch*(sortedProductsById, "productId", "P001");

System.***out***.println("Binary Search (ID P001 - First): " + (foundProductBinaryIdFirst != null ? foundProductBinaryIdFirst : "Not Found"));

Product foundProductBinaryIdLast = SearchAlgorithms.*binarySearch*(sortedProductsById, "productId", "P030");

System.***out***.println("Binary Search (ID P030 - Last): " + (foundProductBinaryIdLast != null ? foundProductBinaryIdLast : "Not Found"));

}

}

Output:

--- Linear Search Examples ---

Linear Search (ID P015): Product ID: P015, Name: Webcam HD, Category: Accessories

Linear Search (ID P999 - Not Found): Not Found

Linear Search (Name Monitor 4K): Product ID: P020, Name: Monitor 4K, Category: Electronics

Linear Search (Category Furniture): Product ID: P030, Name: Ergonomic Chair, Category: Furniture

--- Binary Search Examples (Sorted by Product ID) ---

Sorted Products: [Product ID: P001, Name: Laptop Pro, Category: Electronics, Product ID: P005, Name: Wireless Mouse, Category: Electronics, Product ID: P010, Name: Mechanical Keyboard, Category: Electronics, Product ID: P015, Name: Webcam HD, Category: Accessories, Product ID: P020, Name: Monitor 4K, Category: Electronics, Product ID: P025, Name: USB Hub, Category: Accessories, Product ID: P030, Name: Ergonomic Chair, Category: Furniture]

Binary Search (ID P015): Product ID: P015, Name: Webcam HD, Category: Accessories

Binary Search (ID P008 - Not Found): Not Found

Binary Search (ID P001 - First): Product ID: P001, Name: Laptop Pro, Category: Electronics

Binary Search (ID P030 - Last): Product ID: P030, Name: Ergonomic Chair, Category: Furniture

**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:**

FinancialForecasting.java

package dsa;

import java.util.HashMap;

import java.util.Map;

public class FinancialForecasting {

private static final Map<String, Double> ***memo*** = new HashMap<>(); // Store calculated values

public static double futureValueRecursive(double initialValue, double growthRate, int years) {

String key = initialValue + "-" + growthRate + "-" + years; // Unique key for each call

if (***memo***.containsKey(key)) {

return ***memo***.get(key); // Return cached result if available

}

double result;

if (years == 0) {

result = initialValue; // Base case

} else {

result = *futureValueRecursive*(initialValue, growthRate, years - 1) \* (1 + growthRate);

}

***memo***.put(key, result); // Store the result for future use

return result;

}

public static void main(String[] args) {

double initialInvestment = 10000;

double annualGrowthRate = 0.08; // 8%

int forecastYears = 10;

double futureValue = *futureValueRecursive*(initialInvestment, annualGrowthRate, forecastYears);

System.***out***.println("Future Value after " + forecastYears + " years: $" + String.*format*("%.2f", futureValue));

int longForecast = 30;

double longTermValue = *futureValueRecursive*(initialInvestment, annualGrowthRate, longForecast);

System.***out***.println("Future Value after " + longForecast + " years: $" + String.*format*("%.2f", longTermValue));

}

}

Output:

Future Value after 10 years: $21589.25

Future Value after 30 years: $100626.57