**SuperSet ID:  6389675**

**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:**

***// Java Project : Singleton design pattern:***

***// Logger.java***

**public class Logger {**

**//1. Created Class named Logger that has a private static instance of itself.**

**private static Logger *instance*;**

**// Ensured Logger is private.**

**private Logger(){**

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

**}**

**// Provided Public method to return the instance**

**public static Logger getInstance() {**

**if(*instance*==null) {**

***instance* = new Logger();**

**}**

**return *instance*;**

**}**

**public void log(String Message) {**

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

**}**

**public static void main(String[] args) {**

**// Getting Instance one**

**Logger logger1 = Logger.*getInstance*();**

**logger1.log("This is Instance one");**

**// Getting Instance Two**

**Logger logger2 = Logger.*getInstance*();**

**logger2.log("This is Instance Two");**

**if(logger1 == logger2) {**

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

**}else {**

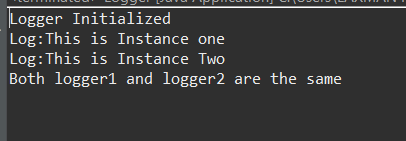
**System.*out*.println("Different instances exist");**

**}**

**}**

**}**

**// 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:**

**// Java Project : FactoryMethodPatternExample.**

**// Create Document Interface**

**// Different Document Classes**

**// Documents.java**

**public interface Documents {**

**void open();**

**}**

**// Concreate Classes**

**//WordDoc.java**

**public class WordDoc implements Documents {**

**public void open() {**

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

**}**

**}**

**//PdfDoc.java**

**public class PdfDoc implements Documents {**

**public void open() {**

**System.*out*.println("Open PDF");**

**}**

**}**

**//ExcelDoc.java**

**public class ExcelDoc implements Documents{**

**public void open() {**

**System.*out*.println("Open Excel");**

**}**

**}**

**// Implementation of Factory Methods**

**//Create a Abstract Class**

**//DocFactory**

**public abstract class DocFactory {**

**public abstract Documents createDoc();**

**}**

**//WordDocFactory.java**

**public class WordDocFactory extends DocFactory {**

***@Override***

**public Documents createDoc() {**

**// TODO Auto-generated method stub**

**return new WordDoc();**

**}**

**}**

**//PdfDocFactory.java**

**public class PdfFactory extends DocFactory {**

***@Override***

**public Documents createDoc() {**

**return new PdfDoc();**

**}**

**}**

**//ExcelDocFactory**

**public class ExcelDocFactory extends DocFactory{**

***@Override***

**public Documents createDoc(){**

**return new ExcelDoc();**

**}**

**}**

**// Testing Of Factory Method**

**//main.java**

**public class main {**

**public static void main(String[] args) {**

**DocFactory wordFactory = new WordDocFactory();**

**Documents word = wordFactory.createDoc();**

**word.open();**

**DocFactory pdfFactory = new PdfDocFactory();**

**Documents pdf = pdfFactory.createDoc();**

**pdf.open();**

**DocFactory excelFactory = new ExcelDocFactory();**

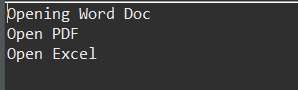
**Documents excel = excelFactory.createDoc();**

**excel.open();**

**}**

**}**

**OUTPUT:**



**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:**

**// Java Project Name: E-commerce Platform Search Function**

**Step 1: Understanding Asymptotic Notation**

**Big O (O) represents the upper bound of an algorithm’s run time as the input size grows .**

**It helps in evaluate of scalability & performance.**

**Case & Description**

**Best Case : fast scenario**

**Average Case :** **Expected time for a random input**

**Worst Case**: **Slowest scenario (e.g., item not found at all)**

**Step 2: Java Project : EcommerceSearch.java**

**// Product.java**

**public 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;**

**}**

***@Override***

**public String toString() {**

**return productId + " - " + productName + " (" + category + ")";**

**}**

**}**

**// SearchDemo.java**

**import java.util.Arrays;**

**public class SearchDemo {**

**// Linear Search**

**public static int linearSearch(Product[] products, String name) {**

**for (int i = 0; i < products.length; i++) {**

**if (products[i].productName.equalsIgnoreCase(name)) {**

**return i;**

**}**

**}**

**return -1;**

**}**

**// Binary Search**

**public static int binarySearch(Product[] products, String name) {**

**int low = 0;**

**int high = products.length - 1;**

**while (low <= high) {**

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

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

**if (compare == 0)**

**return mid;**

**else if (compare < 0)**

**low = mid + 1;**

**else**

**high = mid - 1;**

**}**

**return -1;**

**}**

**public static void main(String[] args) {**

**Product[] products = {**

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

**new Product(102, "Chair", "Furniture"),**

**new Product(103, "Shoes", "Footwear"),**

**new Product(104, "Watch", "Accessories"),**

**new Product(105, "Phone", "Electronics")**

**};**

**// For binary search: sort the array by productName**

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

**String searchKey = "Watch";**

**// Linear Search**

**int linearIndex = *linearSearch*(products, searchKey);**

**System.*out*.println("Linear Search Result: " + (linearIndex != -1 ? products[linearIndex] : "Not Found"));**

**// Binary Search**

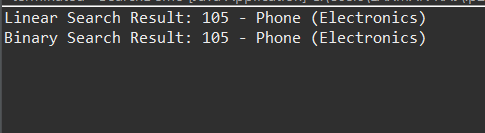
**int binaryIndex = *binarySearch*(products, searchKey);**

**System.*out*.println("Binary Search Result: " + (binaryIndex != -1 ? products[binaryIndex] : "Not Found"));**

**}**

**}**

**OUTPUT:**

****

**Step 4: Analysis**

**//Linear Search**

**Time Complexity:** **O(n)**

**Best Case:** **O(1) (first item)**

**Worst Case:** **O(n) (last/not found)**

**//Binary Search**

**Time Complexity:** **O(log n)**

**Best Case:** **O(1)**

**Worst Case:** **O(log n)**

**// Comparison**

* **Binary Search is faster (O(log n)) but requires the array to be sorted.**
* **Use Binary Search if:**
  + **Your products are pre-sorted or sorted once during load.**
  + **You need faster lookup.**
* **Use Linear Search for small lists or unsorted data.**

**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:**

**// Java Project: Financial Forecasting**

**Step 1: Understand Recursive Algorithms**

* **Recursion is when a method calls itself to solve smaller parts of a larger problem.**
* **It consists of:**
  1. **Base case – when to stop recursion.**
  2. **Recursive case – when the method calls itself with a modified parameter.**

**Use Case in Finance:  
Recursion is useful to model compound growth, e.g., predicting investment value over time.**

**// FinancialForecasting.java**

**public class ForecastTool {**

**// Recursive method to calculate future value**

**public static double predictFutureValue(double presentValue, double growthRate, int years) {**

**if (years == 0) {**

**return presentValue;**

**} else {**

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

**}**

**}**

**public static void main(String[] args) {**

**double presentValue = 10000; // starting amount**

**double annualGrowthRate = 0.08; // 8% growth per year**

**int years = 5;**

**double futureValue = predictFutureValue(presentValue, annualGrowthRate, years);**

**System.out.printf("Future Value after %d years = ₹%.2f%n", years, futureValue);**

**}**

**}**

**// Optimized Iterative Version**

**public static double predictFutureValueIterative(double presentValue, double growthRate, int years) {**

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

**presentValue \*= (1 + growthRate);**

**}**

**return presentValue;**

**}**

**Time Complexity**

**The function calls itself once per year ⇒ Time Complexity: O(n) (where n = years).**

**Optimization (to avoid excessive computation):**

**Use Memoization to cache already computed results (for overlapping subproblems).**

**Or use Iteration (preferred for simple compound calculations).**

**Conclusion:**

* **Recursive methods make the code clean and close to mathematical definitions.**
* **But for large input, iterative or memoized versions are more efficient and safe (avoid stack overflow).**