**Week 1- Mandatory Hands-on exercises**

Design Patterns And Principals:

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.

**Code**

public class Main{

public static void main(String[] args){

Logger logger=Logger.getInstance();

logger.log("Application started successfully.");

logger.log("User input received.");

logger.log("Processing data...");

logger.log("Task completed.");

Logger anotherLogger=Logger.getInstance();

if (logger==anotherLogger){

System.out.println("Singleton check passed: Only one Logger instance exists.");

}

else{

System.out.println("Singleton check failed: Multiple Logger instances found.");

}

}

}

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);

}

}

A screenshot of a computer

AI-generated content may be incorrect.

**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.

import java.util.Scanner;

public class Main{

public static void main(String[] args){

Scanner sc = new Scanner(System.in);

System.out.println("Choose document type to open:");

System.out.println("1.Word");

System.out.println("2.PDF");

System.out.println("3.Excel");

System.out.print("Enter your choice: ");

int choice = sc.nextInt();

DocumentFactory factory;

switch (choice){

case 1:

factory = new WordDocumentFactory();

break;

case 2:

factory = new PdfDocumentFactory();

break;

case 3:

factory = new ExcelDocumentFactory();

break;

default:

System.out.println("Invalid choice. Exiting.");

sc.close();

return;

}

Document document = factory.createDocument();

document.open();

sc.close();

}

}

// Document interface

interface Document{

void open();

}

// Concrete document classes

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 factory

abstract class DocumentFactory{

public abstract Document createDocument();

}

// Concrete factories

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();

}

}

OUTPUT:

A screenshot of a computer

AI-generated content may be incorrect.

**Algorithms\_Data Structure**

**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.

1. **Setup:**

* Create a class Product with attributes for searching, such as productId, productName, and category**.**

1. **Implementation:**

* Implement linear search and binary search algorithms.
* Store products in an array for linear search and a sorted array for binary search.

1. **Analysis:**

* Compare the time complexity of linear and binary search algorithms.
* Discuss which algorithm is more suitable for your platform and why.

**Step 1: Understanding Asymptotic Notation**

**Big O Notation**

Big O notation describes the **upper bound** of an algorithm’s running time in terms of input size (n). It helps in understanding how the performance of an algorithm scales.

* **O(1):** Constant time – best case.
* **O(n):** Linear time – performance degrades linearly.
* **O(log n):** Logarithmic time – very efficient for large datasets.

**Best, Average, and Worst Case for Search**

* **Linear Search:**
  + Best Case: O(1) (item is first)
  + Average Case: O(n/2) ≈ O(n)
  + Worst Case: O(n) (item is last or not found)
* **Binary Search (on sorted data):**
  + Best Case: O(1)
  + Average Case: O(log n)
  + Worst Case: O(log n)
* **Step 2: Setup – Product Class & Dataset**

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

}

}

**Step 3: Implementation – Search Algorithms + Main Method**

import java.util.Arrays;

import java.util.Comparator;

import java.util.Scanner;

// Product class to represent each item in the store

class Product{

int productId;

String productName;

String category;

// Constructor to initialize product fields

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

this.productId=productId;

this.productName=productName;

this.category=category;

}

// Method to display product details in a readable format

@Override

public String toString(){

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

}

}

// Main class containing search methods

public class ProductSearch{

// Linear search based on product ID

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

for(Product p:products){

if(p.productId==targetId){

return p;

}

}

return null;

}

// Binary search based on product ID (requires sorted array)

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

int left=0,right=products.length-1;

while(left<=right){

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

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

return products[mid];

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

left=mid+1;

}else{

right=mid-1;

}

}

return null;

}

// Case-insensitive exact match search by product name

public static Product searchByName(Product[] products,String name){

for(Product p:products){

if(p.productName.equalsIgnoreCase(name)){

return p;

}

}

return null;

}

// Case-insensitive partial match search by category

public static void searchByCategory(Product[] products,String category){

boolean found=false;

for(Product p:products){

if(p.category.toLowerCase().contains(category.toLowerCase())){

System.out.println(p);

found=true;

}

}

if(!found){

System.out.println("No products found in the category:"+category);

}

}

public static void main(String[] args){

Scanner sc=new Scanner(System.in);

// Sample dataset of products

Product[] products={

new Product(201,"Wireless Mouse","Electronics"),

new Product(202,"Gaming Keyboard","Electronics"),

new Product(203,"Yoga Mat","Fitness"),

new Product(204,"Water Bottle","Fitness"),

new Product(205,"Bluetooth Speaker","Electronics"),

new Product(206,"Notebook","Stationery"),

new Product(207,"Desk Lamp","Furniture"),

new Product(208,"Running Shoes","Fashion"),

new Product(209,"Leather Wallet","Fashion"),

new Product(210,"Noise Cancelling Headphones","Electronics")

};

// --- Linear and Binary Search by ID ---

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

int searchId=sc.nextInt();

sc.nextLine(); // Consume newline

// Linear search output

Product result1=linearSearch(products,searchId);

System.out.println("\nLinear Search Result:");

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

// Sort array for binary search

Arrays.sort(products,Comparator.comparingInt(p->p.productId));

// Binary search output

Product result2=binarySearch(products,searchId);

System.out.println("\nBinary Search Result:");

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

// Search by Name

System.out.print("\nEnter Product Name to search:");

String name=sc.nextLine();

Product nameResult=searchByName(products,name);

System.out.println("\nSearch by Name Result:");

System.out.println(nameResult!=null?nameResult:"Product not found.");

//Search by Category

System.out.print("\nEnter Category to search (partial or full):");

String category=sc.nextLine();

System.out.println("\nSearch by Category Result:");

searchByCategory(products,category);

sc.close(); // Close the scanner

}

}

**Step 4: Analysis**

**Time Complexity Comparison**

* **Linear Search:** O(n)
  + No sorting needed
  + Works on unsorted data
* **Binary Search:** O(log n) search, but O(n log n) for sorting initially
  + Only works on sorted data
  + Much faster on large datasets

**Which to Use and When?**

* Use **Linear Search** for **small datasets** or **unsorted lists**.
* Use **Binary Search** for **large datasets** where **sorting is maintained** or acceptable.

OUTPUT:

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.**

**1. Understand Recursive Algorithms**

**Recursion** is a technique where a method calls itself to solve a smaller version of a problem. It's often used when a problem can be divided into similar sub-problems.

**Example:** Calculating future value recursively is like saying:

FutureValue(year) = FutureValue(year - 1) \* (1 + growthRate)

**2. Setup**

We’ll create a recursive method that predicts the future value based on:

* Initial investment
* Annual growth rate
* Number of years

public class FinancialForecast{

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

if(years==0){

return initialValue;

}

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

}

}

3. Implementation

import java.util.Scanner;

public class FinancialForecast{

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

if(years==0){

return initialValue;

}

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

}

public static void main(String[]args){

Scanner sc=new Scanner(System.in);

System.out.print("Enter initial investment amount: ");

double initial=sc.nextDouble();

System.out.print("Enter annual growth rate (as decimal, e.g. 0.05 for 5%): ");

double rate=sc.nextDouble();

System.out.print("Enter number of years to forecast: ");

int years=sc.nextInt();

double result=futureValue(initial,rate,years);

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

}

}

**4. Analysis**

**Time Complexity:**

* Time complexity is **O(n)** where n is the number of years.
* Each call depends on the result of the previous year, forming a linear chain of calls.

**Optimization:**

* The current implementation is efficient enough since each recursive call only computes once.
* However, if reused values were involved (like in Fibonacci), we could optimize using **memoization** or **bottom-up dynamic programming**.

OUTPUT:

A screen shot of a computer

AI-generated content may be incorrect.