**WEEK-1**

**Design principles & Patterns**

**BONDA PRASANTH**

**6382932**

**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:

**Logger.java**:

public class Logger {

    private static Logger instance;

    private Logger() {

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

    }

    public static Logger getInstance() {

        if (instance == null) {

            instance = new Logger();

        }

        return instance;

    }

    public void log(String message) {

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

    }

}

**Main.java:**

public class Main {

    public static void main(String[] args) {

        Logger logger1 = Logger.getInstance();

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

        Logger logger2 = Logger.getInstance();

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

        if (logger1 == logger2) {

            System.out.println("Both logger instances are the same (singleton verified).");

        } else {

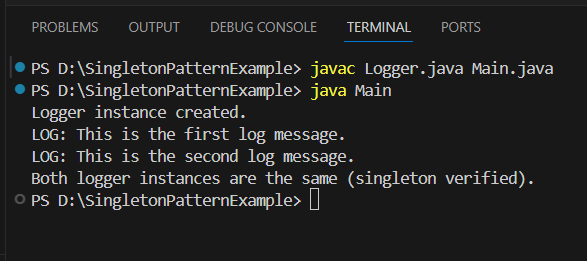
            System.out.println("Different logger instances (singleton failed).");

        }

    }

}

**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.

**CODE:**

**Document.java:**

public interface Document {

void open();

}

**WordDocument.java:**

public class WordDocument implements Document {

public void open() {

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

}

}

**PdfDocument.java:**

public class PdfDocument implements Document {

public void open() {

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

}

}

**ExcelDocument.java:**

public class ExcelDocument implements Document {

public void open() {

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

}

}

**DocumentFactory.java:**

public abstract class DocumentFactory {

public abstract Document createDocument();

}

**WordDocumentFactory.java:**

public class WordDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new WordDocument();

}

}

**PdfDocumentFactory.java:**

public class PdfDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new PdfDocument();

}

}

**ExcelDocumentFactory.java:**

public class ExcelDocumentFactory extends DocumentFactory {

public Document createDocument() {

return new ExcelDocument();

}

}

**Main.java:**

public class Main {

public static void main(String[] args) {

DocumentFactory wordFactory = new WordDocumentFactory();

Document word = wordFactory.createDocument();

word.open();

DocumentFactory pdfFactory = new PdfDocumentFactory();

Document pdf = pdfFactory.createDocument();

pdf.open();

DocumentFactory excelFactory = new ExcelDocumentFactory();

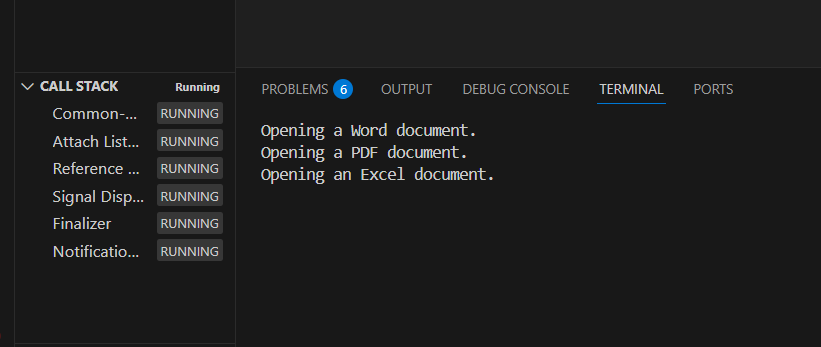
Document excel = excelFactory.createDocument();

excel.open();

}

}

**OUTPUT:**

****

**ALGORITHMS AND DATA STRUCTURES**

**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.

ANSWER:

**1)Big O notation** describes the upper bound of an algorithm's time or space complexity — how performance scales with input size n.

O(1) – Constant time

O(n) – Linear time

O(log n) – Logarithmic time

O(n log n) – Log-linear time

O(n²) – Quadratic time

**Linear Search**

**Best Case**: O(1) – Found at the beginning.

**Average Case**: O(n/2) → Simplified as O(n)

**Worst Case**: O(n) – Found at the end or not found.

**Binary Search**

**Best Case**: O(1) – Middle element is the target.

**Average/Worst Case**: O(log n) – Repeatedly halve the search space.

**CODE:**

**Product.java:**

public class 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 "[" + productId + "] " + productName + " (" + category + ")";

}

}

**SearchUtil.java:**

import java.util.Arrays;

import java.util.Comparator;

public class SearchUtil {

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

for (Product p : products) {

if (p.getProductName().equalsIgnoreCase(targetName)) {

return p;

}

}

return null;

}

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

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

while (left <= right) {

int mid = (left + right) / 2;

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

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

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

else right = mid - 1;

}

return null;

}

// Sort products alphabetically by name

public static void sortByName(Product[] products) {

Arrays.sort(products, Comparator.comparing(Product::getProductName, String.CASE\_INSENSITIVE\_ORDER));

}

}

**Main.java:**

public class Main {

public static void main(String[] args) {

Product[] products = {

new Product(1, "Laptop", "Electronics"),

new Product(2, "Shoes", "Footwear"),

new Product(3, "Watch", "Accessories"),

new Product(4, "Camera", "Electronics"),

new Product(5, "Backpack", "Bags")

};

// Linear Search

Product result1 = SearchUtil.linearSearch(products, "Camera");

System.out.println("Linear Search Result: " + result1);

// Sort before Binary Search

SearchUtil.sortByName(products);

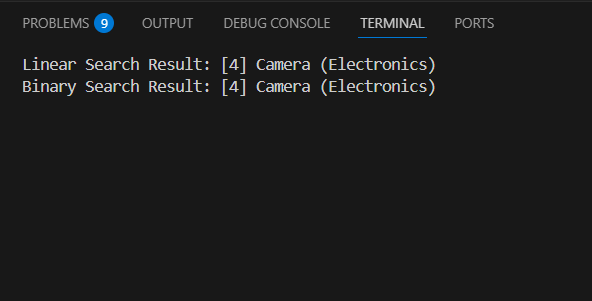
Product result2 = SearchUtil.binarySearch(products, "Camera");

System.out.println("Binary Search Result: " + result2);

}

}

**OUTPUT:**

****

Binary Search is significantly faster if data is sorted.

Linear Search works on unsorted data but is slower as the size grows.

**Final Recommendation for E-commerce Platform**

Use Binary Search for performance.

**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.**

**ANSWER:**

Recursion is when a method calls itself to solve a smaller instance of the same problem.

Why use recursion?

It simplifies problems that have repetitive structure, like calculating factorials, Fibonacci numbers, or in this case, future value over time.

We'll calculate **future value** using a compound growth formula recursively.

FutureValue = PresentValue × (1 + Rate)^Years

FV(P, r, y) = P × (1 + r) if y == 1

= FV(P, r, y-1) × (1 + r) if y > 1

**CODE:**

**FinancialForecast.java:**

public class FinancialForecast {

// Recursive method to calculate future value

public static double futureValue(double principal, double rate, int years) {

if (years == 0) {

return principal;

}

return futureValue(principal, rate, years - 1) \* (1 + rate);

}

public static void main(String[] args) {

double initialAmount = 10000; // $10,000

double annualRate = 0.05; // 5% growth

int forecastYears = 5;

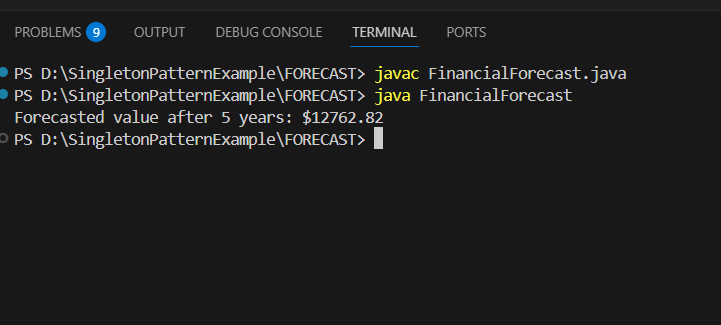
double result = futureValue(initialAmount, annualRate, forecastYears);

System.out.printf("Forecasted value after %d years: $%.2f\n", forecastYears, result);

}

}

**OUTPUT:**

****

The recursive method here has:

* Time Complexity: O(n) (1 recursive call per year)
* Space Complexity: O(n) due to call stack for recursion
* For large values of years, recursion can cause **stack overflow** or slow performance.
* Use Iteration Instead

public static double futureValueIterative(double principal, double rate, int years) {

double result = principal;

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

result \*= (1 + rate);

}

return result;

}

Or Use Formula Directly (Most Efficient):

public static double futureValueFormula(double principal, double rate, int years) {

return principal \* Math.pow(1 + rate, years);

}

| **Method** | **Time Complexity** | **Space Complexity** | **Notes** |
| --- | --- | --- | --- |
| Recursive | O(n) | O(n) | Simple, but not efficient |
| Iterative | O(n) | O(1) | Better for large n |
| Math.pow Formula | O(1) | O(1) | Fastest and most efficient |