**Week 1**

***Design Patterns and Principles***

**Implementing the Singleton Pattern**

**Logger.java**

package com.example.singleton;

import java.io.FileWriter;

import java.io.IOException;

import java.io.PrintWriter;

import java.time.LocalDateTime;

public class Logger {

private static volatile Logger instance; // Single static instance (volatile for thread-safety)

private PrintWriter writer;

// Private constructor prevents instantiation

private Logger() {

try {

writer = new PrintWriter(new FileWriter("app.log", true), true);

} catch (IOException e) {

e.printStackTrace(); // handle exception

}

}

// Thread-safe, lazy-loaded Singleton getter (double-checked locking)

public static Logger getInstance() {

if (instance == null) {

synchronized (Logger.class) {

if (instance == null) {

instance = new Logger();

}

}

}

return instance;

}

// Logging methods

public void log(String level, String message) {

String time = LocalDateTime.now().toString();

writer.printf("%s [%s]: %s%n", time, level, message);

}

public void info(String msg) { log("INFO", msg); }

public void warn(String msg) { log("WARN", msg); }

public void error(String msg) { log("ERROR", msg); }

}

**TestSingleton.java**

package com.example.singleton;

public class TestSingleton {

public static void main(String[] args) {

Logger loggerA = Logger.getInstance();

Logger loggerB = Logger.getInstance();

System.out.println("Same instance? " + (loggerA == loggerB)); // should print true

loggerA.info("Starting app...");

loggerB.warn("Low disk space");

loggerA.error("Unexpected error occurred");

}

}

**Output**



**Implementing the Factory Method Pattern**

**IDocument.java**

package com.example.factory;

public interface IDocument {

void open();

void close();

}

**WordDocument.java**

package com.example.factory;

public class WordDocument implements IDocument {

@Override

public void open() {

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

}

@Override

public void close() {

System.out.println("Closing Word document");

}

}

**PdfDocument.java**

package com.example.factory;

public class PdfDocument implements IDocument {

@Override

public void open() {

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

}

@Override

public void close() {

System.out.println("Closing PDF document");

}

}

**ExcelDocument.java**

package com.example.factory;

public class ExcelDocument implements IDocument {

@Override

public void open() {

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

}

@Override

public void close() {

System.out.println("Closing Excel document");

}

}

**DocumentFactory.java**

package com.example.factory;

public abstract class DocumentFactory {

public abstract IDocument createDocument();

}

**WordFactory.java**

package com.example.factory;

public class WordFactory extends DocumentFactory {

@Override

public IDocument createDocument() {

return new WordDocument();

}

}

**PdfFactory.java**

package com.example.factory;

public class PdfFactory extends DocumentFactory {

@Override

public IDocument createDocument() {

return new PdfDocument();

}

}

**ExcelFactory.java**

package com.example.factory;

public class ExcelFactory extends DocumentFactory {

@Override

public IDocument createDocument() {

return new ExcelDocument();

}

}

**TestFactory.java**

package com.example.factory;

public class TestFactory {

public static void main(String[] args) {

DocumentFactory factory;

// Create and use a Word document

factory = new WordFactory();

IDocument doc1 = factory.createDocument();

doc1.open();

doc1.close();

// Create and use a PDF document

factory = new PdfFactory();

IDocument doc2 = factory.createDocument();

doc2.open();

doc2.close();

// Create and use an Excel document

factory = new ExcelFactory();

IDocument doc3 = factory.createDocument();

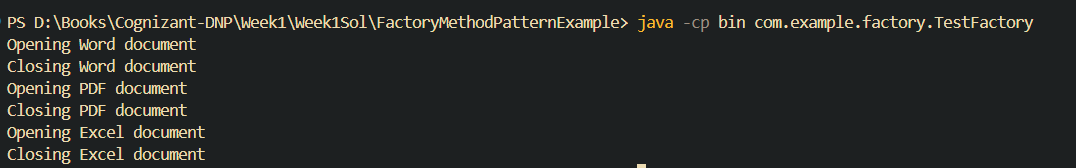
doc3.open();

doc3.close();

}

}

**Output**

****

***Algorithms Data Structures***

**E-commerce Platform Search Function**

**1. Understanding Asymptotic Notation**

* **Big O notation** expresses how algorithm run time grows with input size (n), focusing on the **upper bound** in the worst case
* **Best-case**: quickest scenario (e.g., O(1) if item found immediately).
* **Average-case**: based on typical inputs.
* **Worst-case**: guarantees max time taken (e.g., scanning whole data).

For search:

* **Linear Search**:
  + *Best*: O(1) (first element).
  + *Average/Worst*: O(n) (on average halfway through, worst when missing) .
* **Binary Search** (on sorted array):
  + *Best*: O(1).
  + *Average/Worst*: O(log n) (halving each step)

**SearchApp.java**

public class SearchApp {

public static void main(String[] args) {

Product[] products = {

new Product(10, "Keyboard", "Electronics"),

new Product(5, "Notebook", "Stationery"),

new Product(30, "Mouse", "Electronics"),

new Product(20, "Pen", "Stationery")

};

int idx1 = SearchAlgorithms.linearSearch(products, 30);

System.out.println("Linear found index: " + idx1 + " -> " +

(idx1 >= 0 ? products[idx1] : "Not Found"));

java.util.Arrays.sort(products,

java.util.Comparator.comparingInt(Product::getProductId));

int idx2 = SearchAlgorithms.binarySearch(products, 30);

System.out.println("Binary found index: " + idx2 + " -> " +

(idx2 >= 0 ? products[idx2] : "Not Found"));

}

}

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; }

@Override

public String toString() {

return productId + ": " + productName + " (" + category + ")";

}

}

class SearchAlgorithms {

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

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

if (arr[i].getProductId() == targetId) return i;

}

return -1;

}

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

int low = 0, high = arr.length - 1;

while (low <= high) {

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

int id = arr[mid].getProductId();

if (id == targetId) return mid;

else if (id < targetId) low = mid + 1;

else high = mid - 1;

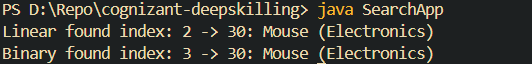
}

return -1;

}

}

**Output**

****

### Time Complexity Comparison

| **Algorithm** | **Best Case** | **Average Case** | **Worst Case** | **Requires Sorted Data** |
| --- | --- | --- | --- | --- |
| Linear Search | O(1) | O(n) | O(n) | ❌ |
| Binary Search | O(1) | O(log n) | O(log n) | ✔️ |

* **Linear Search** works on any array but becomes inefficient as n grows
* **Binary Search** is far faster for large n, halving the search space each time

### Choosing the Right Search

* For **small or unsorted lists,** or one-off searches: **Linear Search** is fine.
* For **large, frequently-searched datasets** (like product catalogs), after initial sorting cost:
  + Sorting is O(n log n)
  + Binary searches thereafter are O(log n) each—far superior as many searches accumulate.
* Best: **Store data sorted**, and use **binary search** for real-time queries.

**Financial Forecasting**

### 1. Understanding Recursive Algorithms

* **Recursion** occurs when a function calls itself to solve smaller instances of the same problem. Base cases prevent infinite loops.
* It simplifies problems like factorials, Fibonacci, or compound calculations by expressing problems in terms of themselves.
* **Example**: Computing future financial value recursively—each year’s value based on the prior one.

**FinancialForecastApp.java**

public class FinancialForecastApp {

public static void main(String[] args) {

double principal = 1000.0;

double[] rates = {0.05, 0.04, 0.06, 0.03};

int years = rates.length;

double result = FinancialForecast.forecast(principal, rates, years);

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

}

}

class FinancialForecast {

public static double forecast(double principal, double[] rates, int years) {

if (years == 0) return principal;

double prev = forecast(principal, rates, years - 1);

return prev \* (1 + rates[years - 1]);

}

}

**Output**

****

### Time Complexity

* Recurrence: T(n) = T(n-1) + O(1) leads to O(n) time complexity
* Each recursive call reduces years by 1 until base case, performing a constant-time multiplication.
* **Space complexity**: Also O(n) due to the recursion stack .

### Optimization Strategies

1. **Memoization**:
   * Not needed here—each year's result used exactly once.
2. **Convert to iterative form**:

double fv = principal;

for (double r : rates) {

fv \*= (1 + r);

}

* + Reduces space usage to O(1), same O(n) time.

1. **Tail recursion** (where supported):
   * Refactor so the recursive call is the last action—can be optimized by compilers/interpreters