**WEEK - 1**

**Design Patterns and Principles Hands On**

**Exercise 1: Implementing the Singleton Pattern**

Code:

Test.java

**public** **class** Test {

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

Logger logger1 = Logger.*getInstance*();

logger1.log("First message");

Logger logger2 = Logger.*getInstance*();

logger2.log("Second message");

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

System.***out***.println("Same instance");

} **else** {

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

}

}

}

Logger.java

**public** **class** Logger {

**private** **static** Logger *instance*; // private static instance of itself

**private** Logger() { // private constructor

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

}

**public** **static** Logger getInstance() { // public static method to get the instance

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

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

}

**return** *instance*;

}

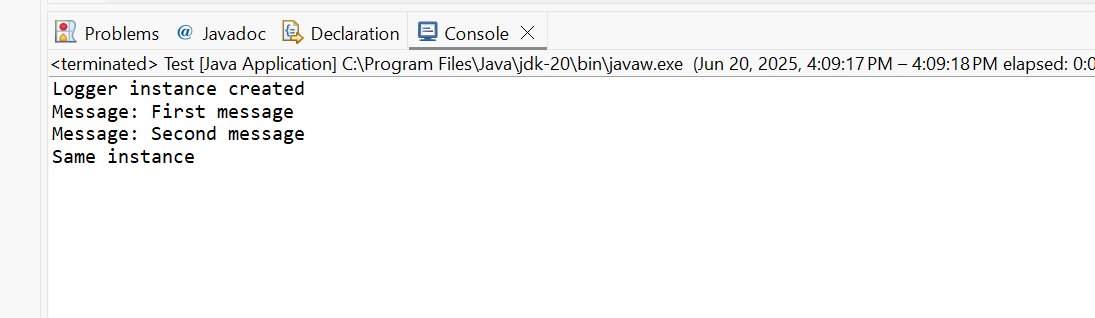
**public** **void** log(String message) { // to log the message

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

}

}

Output:



**Exercise 2: Implementing the Factory Method Pattern**

Code:

Main.java

**public** **class** Main {

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

DocumentFactory wordFactory = **new** WordFactory();

Document wordDoc = wordFactory.createDocument();

wordDoc.show();

DocumentFactory pdfFactory = **new** PDFFactory();

Document pdfDoc = pdfFactory.createDocument();

pdfDoc.show();

DocumentFactory excelFactory = **new** ExcelFactory();

Document excelDoc = excelFactory.createDocument();

excelDoc.show();

}

}

Document.java

**public** **interface** Document {

**void** show();

}

DocumentFactory.java

**public** **abstract** **class** DocumentFactory {

**public** **abstract** Document createDocument();

}

ExcelDocument.java

**public** **class** ExcelDocument **implements** Document {

**public** **void** show() {

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

}

}

ExcelFactory.java

**public** **class** ExcelFactory **extends** DocumentFactory {

**public** Document createDocument() {

**return** **new** ExcelDocument();

}

}

PDFDocument.java

**public** **class** PDFDocument **implements** Document {

**public** **void** show() {

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

}

}

PDFFactory.java

**public** **class** PDFFactory **extends** DocumentFactory {

**public** Document createDocument() {

**return** **new** PDFDocument();

}

}

WordDocument.java

**public** **class** WordDocument **implements** Document {

**public** **void** show() {

System.***out***.println("Opening WordDocument");

}

}

WordFactory.java

**public** **class** WordFactory **extends** DocumentFactory {

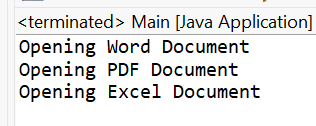
**public** Document createDocument() {

**return** **new** WordDocument();

}

}

Output:



**Algorithms and Data Structures Hands On**

**Exercise 2: E-commerce Platform Search Function**

1. **Understand Asymptotic Notation:**

Big O Notation describes how the runtime of an algorithm grows with input size n.

* O (1): Constant time
* O (n): Linear time
* O (log n): Logarithmic time
* O (n log n): Linearithmic
* O(n²): Quadratic (nested loops)
* It helps predict how an algorithm will scale, not just how fast it is on one input.

For Linear Search:

* Best Case – O (1) (when element is matched at first index)
* Average Case – O(N/2) ~ O(N) (when element is matched at middle index)
* Worst Case – O(N) (when element is matched at last index)

For Binary Search:

* Best Case – O (1) (when element is matched at middle index)
* Average Case, Worst Case– O(logN) (when element is matched at first/ last or other indexes)

1. **Code:**

**Product.java**

**public** **class** Product {

**int** productId;

String productName;

String category;

// constructor

**public** Product(**int** id, String name, String category) {

**this**.productId = id;

**this**.productName = name;

**this**.category = category;

}

}

Search.java

**import** java.util.Arrays;

**import** java.util.Comparator;

**public** **class** Search {

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

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

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

**return** i;

}

}

**return** -1;

}

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

Arrays.*sort*(products, Comparator.*comparing*(p -> p.productName.toLowerCase()));

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

**while** (left <= right) {

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

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

**if** (cmp == 0) **return** mid;

**else** **if** (cmp < 0) right = mid - 1;

**else** left = mid + 1;

}

**return** -1;

}

}

Main.java

**public** **class** Main {

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

Product[] products = {

**new** Product(1, "p1", "c1"),

**new** Product(2, "p2", "c2"),

**new** Product(3, "p3", "c1"),

**new** Product(4, "p4", "c2"),

**new** Product(5, "p5", "c2"),

**new** Product(6, "p6", "c2")

};

// Linear Search

**int** result1 = Search.*linearSearch*(products, "p5");

System.***out***.println("Linear Search Index:(0-based) " + result1);

// Binary Search

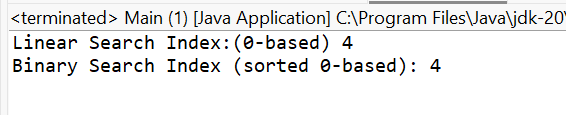
**int** result2 = Search.*binarySearch*(products, "p5");

System.***out***.println("Binary Search Index (sorted 0-based): " + result2);

}

}

1. **Output:**



1. **Analysis:**

Unsorted data calls for Linear Search with O(N) time complexity and sorted data calls for Binary Search with O(logN). In this scenario, Binary search is ideal where products are indexed & sorted, and it reduces the search speed.

**Exercise 7: Financial Forecasting**

1. Recursive Algorithms:

Recursion is a programming paradigm where a function calls itself to solve a smaller instance of the same problem. It is particularly useful in cases where a problem can be broken down into smaller subproblems with the same structure. It has three main parts: base case, assumptions and main logic.

In a financial forecasting tool, we can use recursion to calculate the future value of an investment by applying a constant growth rate over several time periods. The recursive method would take as input the current value, the growth rate, and the number of periods, and then repeatedly apply the growth rate until it reaches the final period.

For example, if the value grows at 5% annually, we can model this by multiplying the current value by 1.05 each year recursively.

1. Code:

Main.java

**public** **class** Main {

**static** **final** **double** ***baseValue*** = 10000; // starting investment

**static** **final** **double** ***growthRate*** = 0.05; // 5% growth per year

**public** **static** **double** predictFutureValue(**int** year) {

**if** (year == 0) {

**return** ***baseValue***;

}

**return** *predictFutureValue*(year - 1) \* (1 + ***growthRate***);

}

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

**int** targetYear = 5;

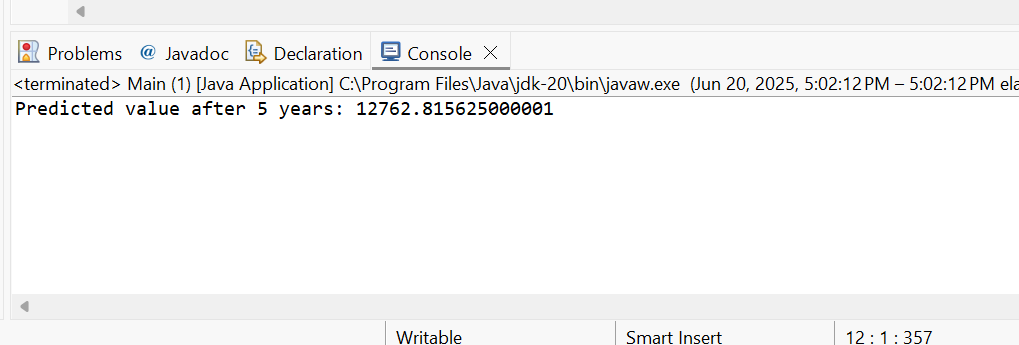
**double** futureValue = *predictFutureValue*(targetYear);

System.***out***.println("Predicted value after " + targetYear + " years: " + futureValue);

}

}

1. Output:



1. Analysis:

This method works by reducing the number of years at each recursive call and applying the growth rate to the current value. When the number of years reaches zero, it returns the final value.

The time complexity of this recursive algorithm is O(n), where n is the number of years. This is because each recursive call processes one year and the function is called n times.

To optimize this recursion, we can use iteration instead of recursion, which avoids the overhead of repeated function calls. Alternatively, we can use memoization to store results of previously computed subproblems if the logic is more complex and involves overlapping subproblems.