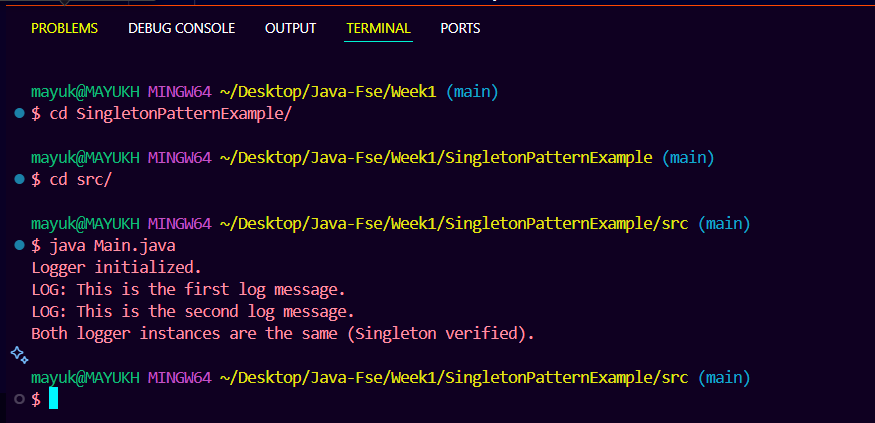
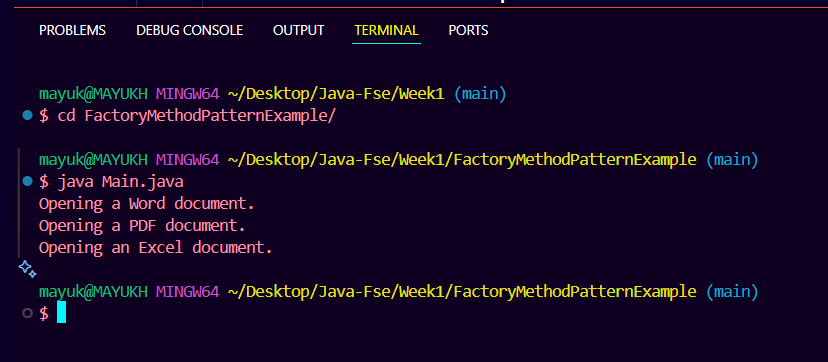
**WEEK 1-MANDATORY HANDS-ON EXERCISES**

**Module 1 - Design Patterns and Principles**

**Exercise 1: Singleton Pattern output**

**Exercise 2: Factory Method Pattern output**

****

**Module 2 - Data Structures and Algorithms**

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

**1. Understand Asymptotic Notation**

* **Big O notation** is a mathematical representation used to describe the upper bound of the time complexity of an algorithm relative to the size of its input (n). It helps analyse and compare the efficiency and scalability of different algorithms, allowing us to predict performance for large input sizes regardless of hardware or implementation.
* In the context of search operations:
  + **Best case:** The target product is found immediately. For example, in linear search, if the target is the first element, the runtime is O(1).
  + **Average case:** The algorithm searches about half the elements on average (linear) or runs log n steps (binary), representing typical expected performance.
  + **Worst case:** The target is found at the last position or not found at all, leading to O(n) complexity for linear search and O(log n) for binary search.

**2. Setup: Product Class**

* The **Product** class models an item in the e-commerce platform, containing searchable attributes:
  + **productId** (int): A unique identifier for each product.
  + **productName** (String): Name of the product, e.g., "Laptop".
  + **category** (String): Category of the product, e.g., "Electronics".
* The constructor initializes these fields, and the overridden **toString()** method nicely formats the product information for printing.

**3. Implementation: Search Algorithms**

* **Linear Search:**
  + Implemented as **linearSearch(Product[] products, int targetId)** in **SearchUtil**.
  + The method iterates through each product in the array and compares its **productId** with the target **targetName**.
  + Returns the matching **Product** if found, otherwise **null**.
  + The array is unsorted during this search, so it checks all elements in sequence.
* **Binary Search:**
  + Implemented as **binarySearch(Product[] products, int targetId)** in **SearchDemo**.
  + Requires the product array to be sorted by **productId** first.
  + Uses a while loop to narrow down the search by comparing the middle element's **productId** with the target.
  + If the middle element equals the target, it returns the product. Otherwise, it adjusts the search bounds **low** and **high** accordingly.
  + Returns **null** if the product is not found.
  + Sorting is done via **sortProducts(Product[] products)**, which uses Java’s **Arrays.sort** with a comparator based on **productId**.
* In **Main**, products are:

Product[] products = {

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

new Product(2, "Smartphone", "Electronics"),

new Product(3, "Tablet", "Electronics"),

new Product(4, "Headphones", "Accessories"),

new Product(5, "Smartwatch", "Accessories")

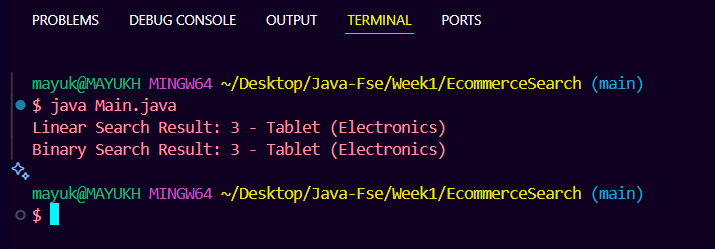
};

The **targetName** set to 3 demonstrates searching for the product "Mobile".

**4. Analysis**

* **Time Complexity Comparison:**
  + Linear Search runs in **O(n)** time for search, where n is the number of products — because it checks each product one by one.
  + Binary Search runs in **O(log n)** time, which is significantly faster for larger datasets, but it requires the input array to be sorted. Sorting itself adds an overhead of **O(n log n)** but can be done once if many searches occur.
* **Suitability for the E-commerce Platform:**
  + Binary Search is preferred for this platform when dealing with large and relatively stable product catalogues since it offers faster search performance after sorting.
  + The code sorts the product array before performing binary search to ensure correctness.
  + Linear Search can be useful for small datasets or when products are frequently added or removed, making sorting costly or infeasible.
  + Overall, for scalability and speed, using binary search on a sorted array is ideal, as implemented here.

**Output**



**Exercise 7: Financial Forecasting**

**1. Understand Recursive Algorithms**

Recursion is a programming technique where a function calls itself to solve smaller instances of a problem. It continues to do this until it reaches a base case that stops the recursion.

In this project, recursion is used to calculate the future value of an investment by reducing the time (number of years) one step at a time. Each call computes interest for the current year and then recurses with the updated principal for the remaining years.

2. Setup: Recursive Method for Future Value

The class Recursion contains two methods:

* recursion: A standard recursive implementation.
* recursionOptimized: A loop-based version for optimization.

Parameters:

1. principal: The initial investment amount (double).
2. rate: The annual interest or growth rate (as a decimal, e.g., 0.05 for 5%).
3. time: The number of years for which the investment grows.

*Recursive logic:*

If time == 0 (base case), return the current principal.

Otherwise, calculate interest for the current year and call the function again with the increased principal and time - 1.

**3. Implementation: Recursive and Optimized Calculations**

*Recursive Method*

public static double recursion(double principal, double rate, int time) {

if (time == 0) {

return principal;

}

double interest = principal \* rate;

return recursion(principal + interest, rate, time - 1);

}

This function calculates compound interest recursively by accumulating interest every year.

*Optimized (Iterative) Method*

public static double recursionOptimized(double principal, double rate, int time) {

double result = principal;

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

result \*= (1 + rate);

}

return result;

}

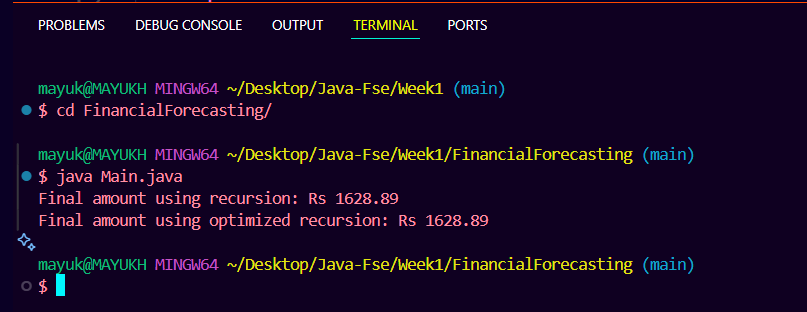
This loop-based method simulates compound growth without recursion.

It improves performance and avoids stack overflow risks.

**4. Analysis**

|  |  |  |  |
| --- | --- | --- | --- |
| **Method** | **Time Complexity** | **Stack Overhead** | **Performance** |
| recursion | O(time) | Yes | Moderate |
| recursionOptimized | O(time) | No | High |

* **Optimization:**
  + Recursion can lead to excessive memory use and slower execution due to function call overhead, especially for large values of **time**.
  + The loop-based method acts like tail recursion by iterating directly, optimizing both memory and performance.
  + For more advanced optimization, techniques like **memoization** or closed-form mathematical formulas (e.g., using **Math.pow**) could be used to calculate future values in O(1) time, further improving efficiency.

**OUTPUT**