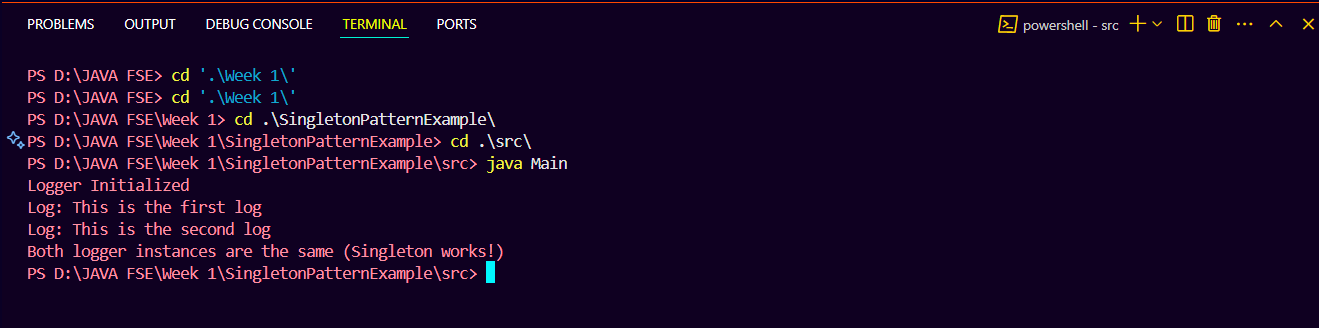
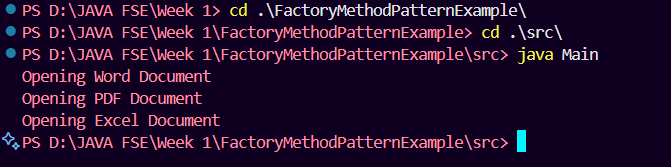
**WEEK 1-MANDATORY HANDS ON EXERCISES**

**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 analyze 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 **targetId**.
  + 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 **SearchUtil**.
  + 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 **sortProductsById(Product[] products)**, which uses Java’s **Arrays.sort** with a comparator based on **productId**.
* In **Main**, products are:

Product[] products = {

new Product(104, "Laptop", "Electronics"),

new Product(101, "T-shirt", "Clothing"),

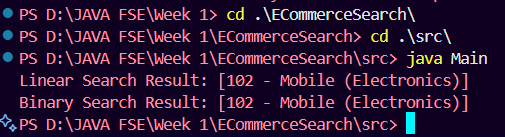
new Product(102, "Mobile", "Electronics"),

new Product(103, "Book", "Stationery")

};

The **targetId** set to 102 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 catalogs 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 method calls itself to solve smaller instances of a problem until a base case is met. It simplifies problems by breaking them down into identical subproblems, often making solutions more intuitive and elegant.
* In this case, recursion helps calculate future values by reducing the number of years left step-by-step until zero, representing the base case.

**2. Setup: Recursive Method for Future Value**

* The **calculateFutureValue** method computes the future value of an investment recursively, using three parameters:
  + **principal**: the initial investment amount.
  + **rate**: the growth rate per year (as a decimal).
  + **years**: the number of years to calculate for.
* If **years** equals zero (base case), it returns the principal (current value). Otherwise, it multiplies the value of the previous year’s investment by **(1 + rate)**, recursively reducing the year count.

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

* The recursive method:

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

if (years == 0) {

return principal;

} else {

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

}

}

* An **optimized method** replaces recursion with a simple loop to achieve the same calculation without the overhead of recursive calls:

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

double result = principal;

for (int i = 1; i <= years; i++) {

result \*= (1 + rate);

}

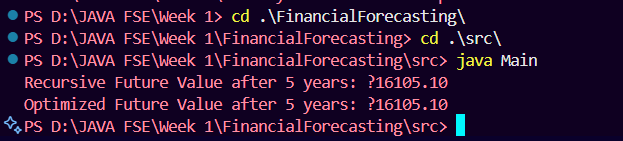
return result;

}

* In **Main**, both methods are called using sample data: a principal of ₹10,000, a 10% growth rate, and a 5-year period. The results are printed formatted to two decimal places.

**4. Analysis**

* **Time Complexity:**
  + The recursive method runs in **O(years)** time because it makes one recursive call per year until it reaches zero.
  + The optimized loop-based method also runs in **O(years)** but with less overhead since it avoids recursive call stack usage.
* **Optimization:**
  + Recursion can lead to excessive memory use and slower execution due to function call overhead, especially for large values of **years**.
  + 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 🡪