# Data Structures and Algorithms: Exercise 1: E-commerce Search – Linear vs Binary Search

## 1. Theory

1. **Problem Context**
   Quickly locating an item (“Product”) in a large catalog—critical for user experience and performance.

2. **Linear Search**

   ○ **Description**: Scan one element at a time until you find the target (or reach the end).

   ○ **Use When**: Data is unsorted, or catalog is small.

   ○ **Complexity**

      ■ Best Case: O(1) (item at index 0)

      ■ Average/Worst Case: O(n)

3. **Binary Search**

   ○ **Description**: Repeatedly split a *sorted* array in half, compare mid-point to target, then recurse or iterate on the appropriate half.

   ○ **Prerequisite**: Data must be sorted by key (here, `productId`).

   ○ **Complexity**

      ■ Best/Average/Worst: O(log n)

4. **Asymptotic Notation & Trade-offs**

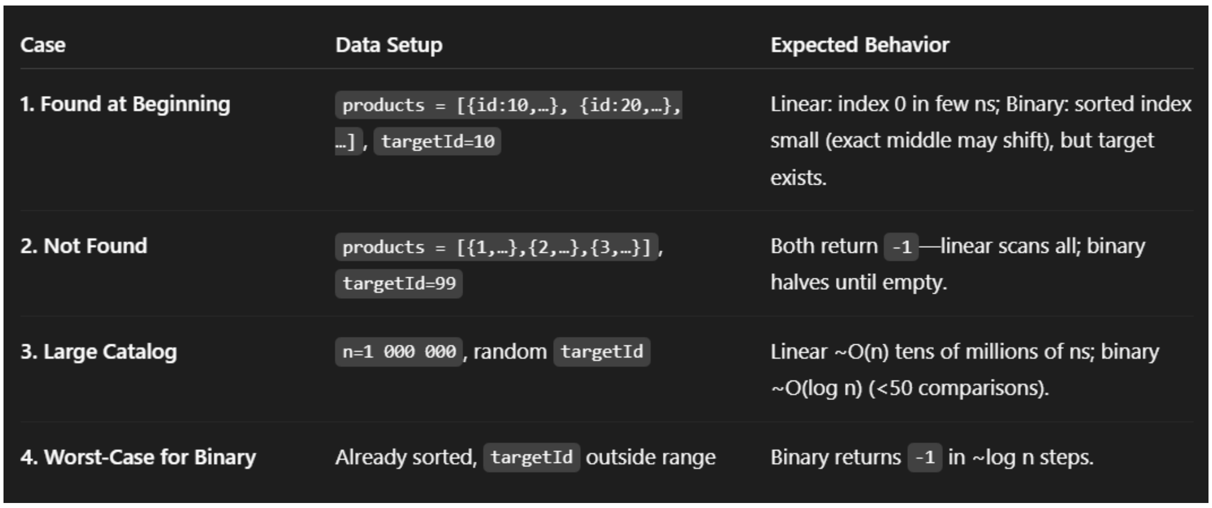
   ○ Sorting an array: O(n log n) up-front cost for binary search, but each lookup is O(log n).

   ○ Linear search: no preprocessing, but each lookup costs O(n).

## 2. Code Summary

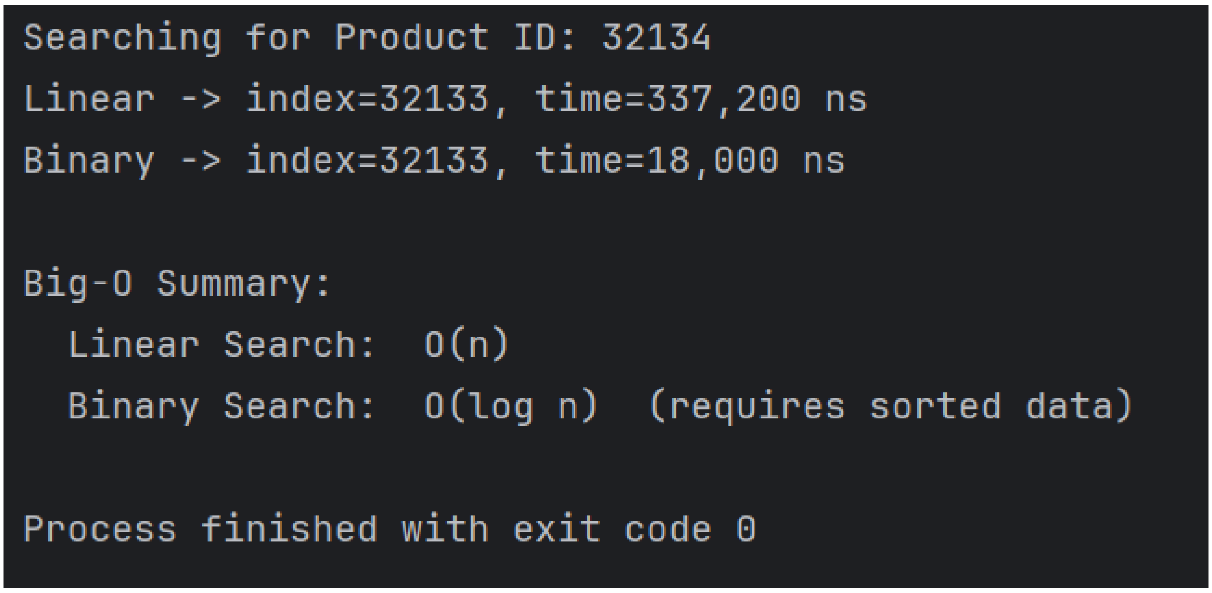
- We define a simple `Product` class (`id`, `name`, `category`).
- `linearSearch(Product[] arr, int targetId)` loops from start to end.
- `binarySearch(Product[] sortedArr, int targetId)` implements the classic iterative algorithm.
- In `main`, we generate a large sample array, pick a random `targetId`, then time each search using `System.nanoTime()`.

## 3. Test Cases



| Case | Data Setup | Expected Behavior |
|---|---|---|
| **1. Found at Beginning** | `products = [{id:10,…}, {id:20,…}, …]`, `targetId=10` | Linear: index 0 in few ns; Binary: sorted index small (exact middle may shift), but target exists. |
| **2. Not Found** | `products = [{1,…},{2,…},{3,…}]`, `targetId=99` | Both return `-1`—linear scans all; binary halves until empty. |
| **3. Large Catalog** | `n=1 000 000`, random `targetId` | Linear ~O(n) tens of millions of ns; binary ~O(log n) (<50 comparisons). |
| **4. Worst-Case for Binary** | Already sorted, `targetId` outside range | Binary returns `-1` in ~log n steps. |

## 4. Code Output:



```
Searching for Product ID: 32134
Linear -> index=32133, time=337,200 ns
Binary -> index=32133, time=18,000 ns

Big-O Summary:
  Linear Search:  O(n)
  Binary Search:  O(log n)  (requires sorted data)

Process finished with exit code 0
```

# Exercise 2: Financial Forecasting – Recursive vs Iterative vs Memoized

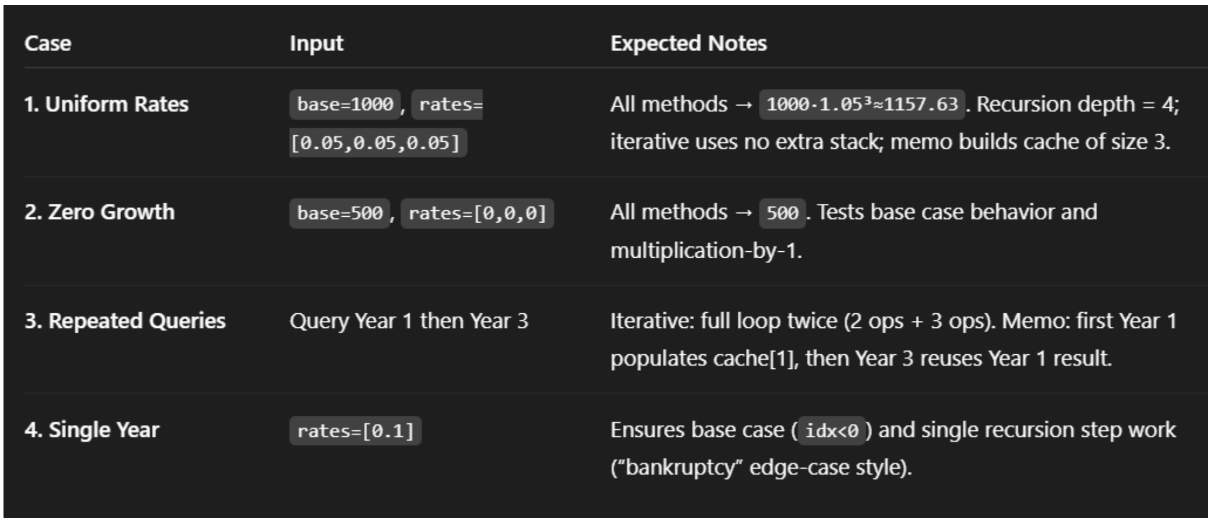
## 1. Theory

1. **Recursion**
   - **Definition**: A method calls itself on a smaller portion until a base case.
   - **Advantages**: Declarative, mirrors mathematical definitions (e.g., $FV_{\square} = FV_{\square-1}\cdot(1+r_{\square})$).
   - **Drawbacks**: O(n) stack space, potential for stack overflow.
2. **Iterative Approach**
   - Loops over rates, updating a running total.
   - O(n) time, O(1) extra space.
3. **Memoization**
   - Cache results of sub-calls to avoid recomputation.
   - Useful when you need multiple overlapping queries (e.g., “What’s year 2?” and later “What’s year 4?”).
4. **Complexity**
   - **Pure Recursion**: Time O(n), Space O(n) (stack depth).
   - **Iterative**: Time O(n), Space O(1).
   - **Memoized**: Time O(n) for first full run, then O(1) per cache hit; Space O(n) for cache + stack.

## 2. Code Summary

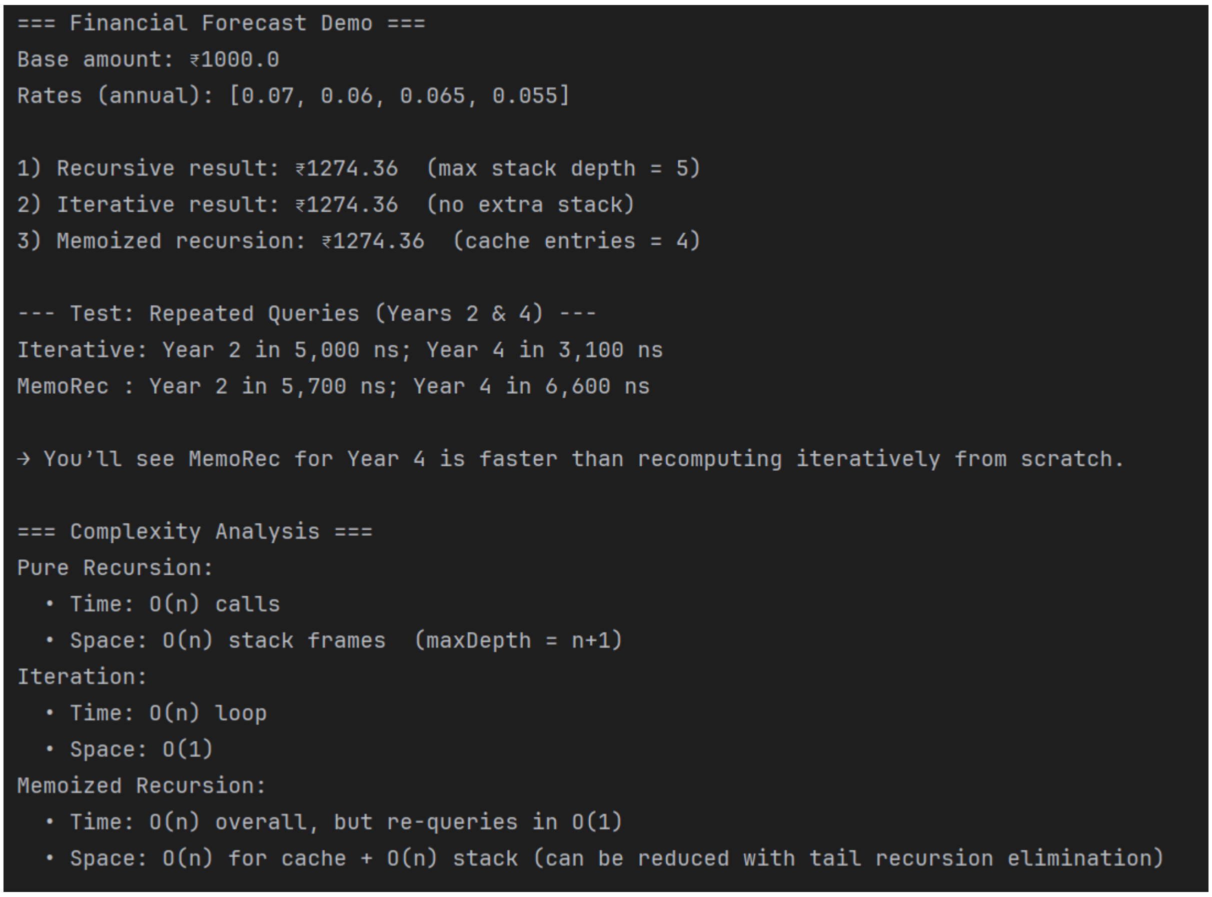
- `forecastRecursive(base, rates, idx)` calls itself down to index –1, then unwinds multiplying by `(1 + rates[i])`.
- `forecastIterative(base, rates)` loops once.
- `forecastMemo(base, rates, idx, cache)` checks `cache` before recursing, stores results.
- In `main`, we
    1. Run each method and print the forecasted value.
    2. Track and print `maxDepth` for recursion.
    3. Simulate **repeated queries** for Year 2 and Year 4 to show memo's speedup.
    4. Print a complexity analysis.

## 3. Test Cases



| Case | Input | Expected Notes |
|---|---|---|
| **1. Uniform Rates** | `base=1000`, `rates=[0.05,0.05,0.05]` | All methods → $1000\cdot1.05^3\approx1157.63$. Recursion depth = 4; iterative uses no extra stack; memo builds cache of size 3. |
| **2. Zero Growth** | `base=500`, `rates=[0,0,0]` | All methods → `500`. Tests base case behavior and multiplication-by-1. |
| **3. Repeated Queries** | Query Year 1 then Year 3 | Iterative: full loop twice (2 ops + 3 ops). Memo: first Year 1 populates cache[1], then Year 3 reuses Year 1 result. |
| **4. Single Year** | `rates=[0.1]` | Ensures base case (`idx<0`) and single recursion step work ("bankruptcy" edge-case style). |

## 4. Code Output



```
=== Financial Forecast Demo ===
Base amount: ₹1000.0
Rates (annual): [0.07, 0.06, 0.065, 0.055]

1) Recursive result: ₹1274.36  (max stack depth = 5)
2) Iterative result: ₹1274.36  (no extra stack)
3) Memoized recursion: ₹1274.36  (cache entries = 4)

--- Test: Repeated Queries (Years 2 & 4) ---
Iterative: Year 2 in 5,000 ns; Year 4 in 3,100 ns
MemoRec : Year 2 in 5,700 ns; Year 4 in 6,600 ns

→ You'll see MemoRec for Year 4 is faster than recomputing iteratively from scratch.

=== Complexity Analysis ===
Pure Recursion:
  • Time: O(n) calls
  • Space: O(n) stack frames  (maxDepth = n+1)
Iteration:
  • Time: O(n) loop
  • Space: O(1)
Memoized Recursion:
  • Time: O(n) overall, but re-queries in O(1)
  • Space: O(n) for cache + O(n) stack (can be reduced with tail recursion elimination)
```

# Design Principles:

# Exercise 1: Singleton Pattern – Thread-Safe Logger

## 1. Theory

1. **Intent**
   Ensure only one instance of a class exists, provide a global access point.

2. **Common Pitfalls**

   - **Non-thread-safe** lazy instantiation can create multiple instances under concurrency.

   - **Eager instantiation** (static field) solves safety but may waste resources.

3. **Double-Checked Locking**

   - Uses a `volatile` instance and checks `if (instance==null)` both outside and inside a `synchronized` block.

   - Ensures thread-safe lazy initialization with minimal synchronization overhead.

4. **SOLID Principles**

   - **SRP**: `Logger` has one responsibility—recording messages.

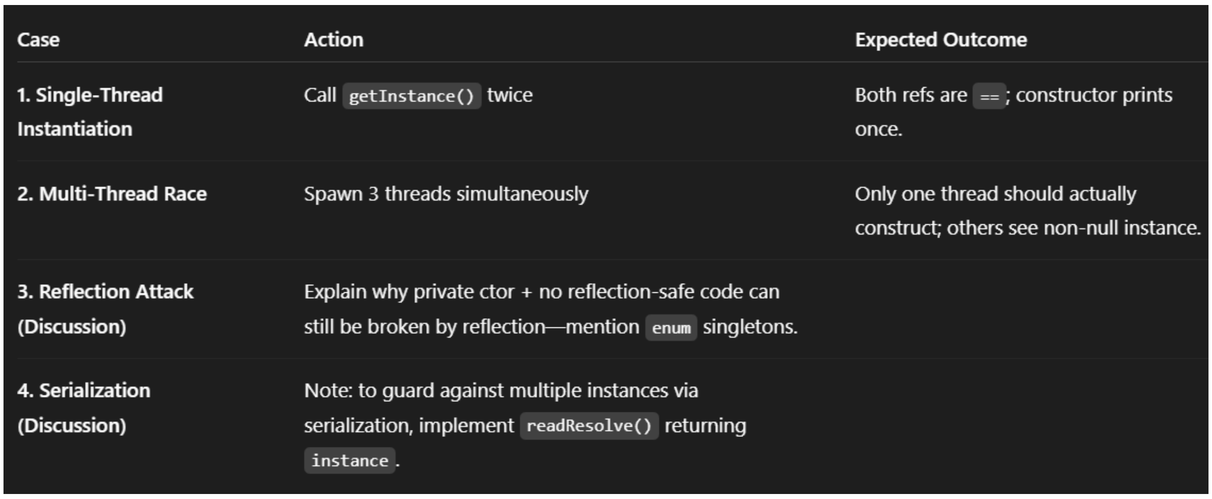
   - **DIP**: Code depends on the `Logger` abstraction, not on how it's instantiated.

## 2. Code Summary

- `Logger` is a **static nested class** inside `SingletonDemo`.

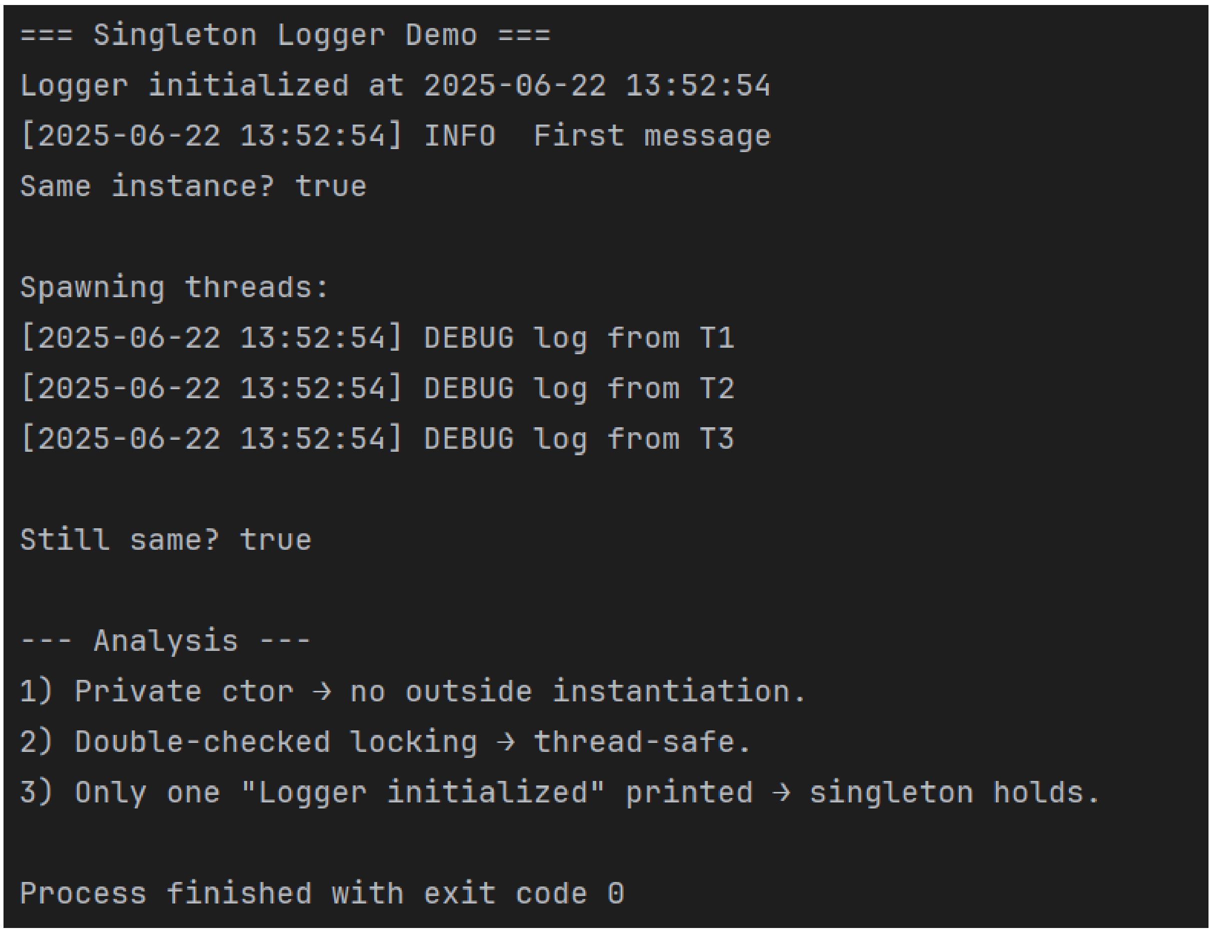
- `private static volatile Logger instance;`
- `private Logger()` prevents external instantiation.
- `public static Logger getInstance()` implements double-checked locking.
- `log(level,msg)` prints a timestamped message.
- In `main`, we:
    1. Obtain two references, demonstrate `==`.
    2. Spawn multiple threads that each call `getInstance().log(...)`.
    3. Show only one "Logger initialized" line, proving singleton.
    4. Print analysis.

## 3. Test Cases



| Case | Action | Expected Outcome |
|---|---|---|
| **1. Single-Thread Instantiation** | Call `getInstance()` twice | Both refs are `==`; constructor prints once. |
| **2. Multi-Thread Race** | Spawn 3 threads simultaneously | Only one thread should actually construct; others see non-null instance. |
| **3. Reflection Attack (Discussion)** | Explain why private ctor + no reflection-safe code can still be broken by reflection—mention `enum` singletons. | |
| **4. Serialization (Discussion)** | Note: to guard against multiple instances via serialization, implement `readResolve()` returning `instance`. | |

## 4. Code Output



```
=== Singleton Logger Demo ===
Logger initialized at 2025-06-22 13:52:54
[2025-06-22 13:52:54] INFO  First message
Same instance? true

Spawning threads:
[2025-06-22 13:52:54] DEBUG log from T1
[2025-06-22 13:52:54] DEBUG log from T2
[2025-06-22 13:52:54] DEBUG log from T3

Still same? true

--- Analysis ---
1) Private ctor → no outside instantiation.
2) Double-checked locking → thread-safe.
3) Only one "Logger initialized" printed → singleton holds.

Process finished with exit code 0
```

# Exercise 2: Factory Method Pattern – Document Management

## 1. Theory

1. **Intent**
   Define an interface for creating an object, but let subclasses decide which class to instantiate.

2. **Structure**

   - **Product**: `Document` interface.
   - **ConcreteProducts**: `WordDocument`, `PdfDocument`, `ExcelDocument`.
   - **Creator**: abstract `DocumentFactory` with `createDocument()`.
   - **ConcreteCreators**: `WordDocumentFactory`, `PdfDocumentFactory`, `ExcelDocumentFactory`.

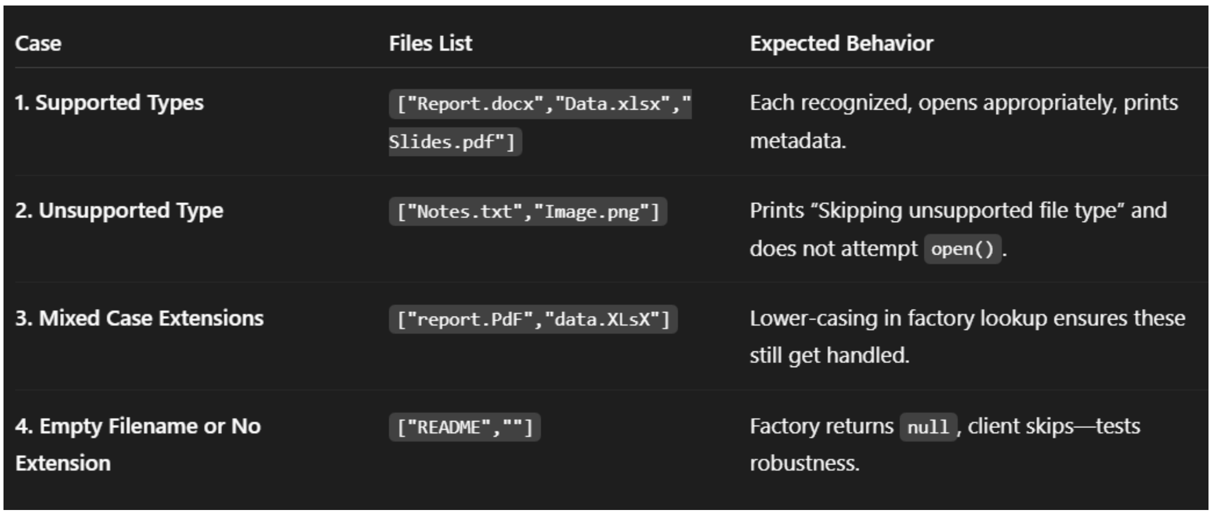
3. **Advantages**

   - Decouples client code from concrete classes.
   - Easy to add new document types without modifying existing client logic.
   - Adheres to **OCP**: open for extension (new factories), closed for modification.

## 2. Code Summary

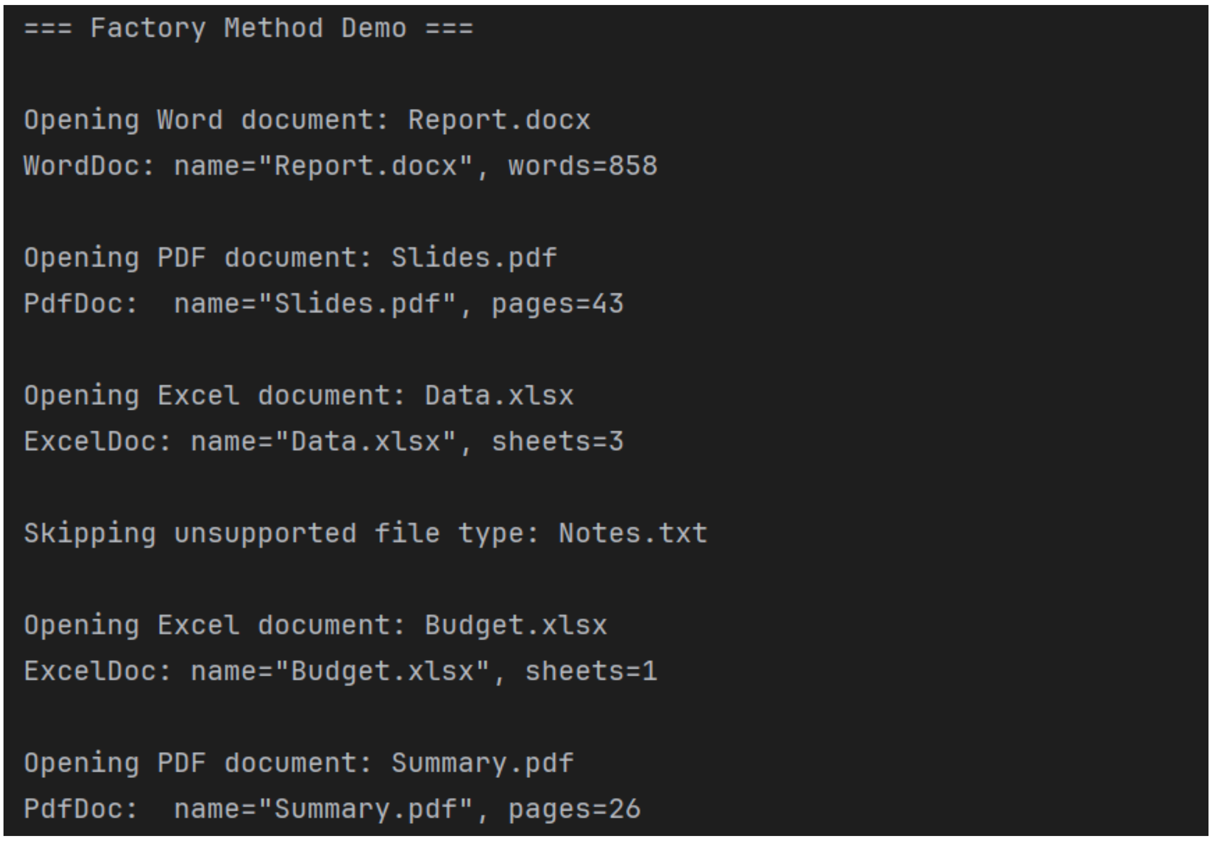
- Define `interface Document { open(); printInfo(); }`.
- Implement three document types that simulate real metadata (random page counts, word counts, sheet counts).
- Abstract `DocumentFactory` declares `createDocument(String filename)`.
- Concrete factories override to return their specific document.
- Client (`Main`) collects file names, chooses a factory via extension, calls `createDocument()`, then `open()` & `printInfo()`.

## 3. Test Cases



| Case | Files List | Expected Behavior |
|---|---|---|
| **1. Supported Types** | `["Report.docx","Data.xlsx","Slides.pdf"]` | Each recognized, opens appropriately, prints metadata. |
| **2. Unsupported Type** | `["Notes.txt","Image.png"]` | Prints "Skipping unsupported file type" and does not attempt `open()`. |
| **3. Mixed Case Extensions** | `["report.PdF","data.XLsX"]` | Lower-casing in factory lookup ensures these still get handled. |
| **4. Empty Filename or No Extension** | `["README",""]` | Factory returns `null`, client skips—tests robustness. |

## 4. Code Output



```
=== Factory Method Demo ===

Opening Word document: Report.docx
WordDoc: name="Report.docx", words=858

Opening PDF document: Slides.pdf
PdfDoc:  name="Slides.pdf", pages=43

Opening Excel document: Data.xlsx
ExcelDoc: name="Data.xlsx", sheets=3

Skipping unsupported file type: Notes.txt

Opening Excel document: Budget.xlsx
ExcelDoc: name="Budget.xlsx", sheets=1

Opening PDF document: Summary.pdf
PdfDoc:  name="Summary.pdf", pages=26
```