**Exercise 1:**

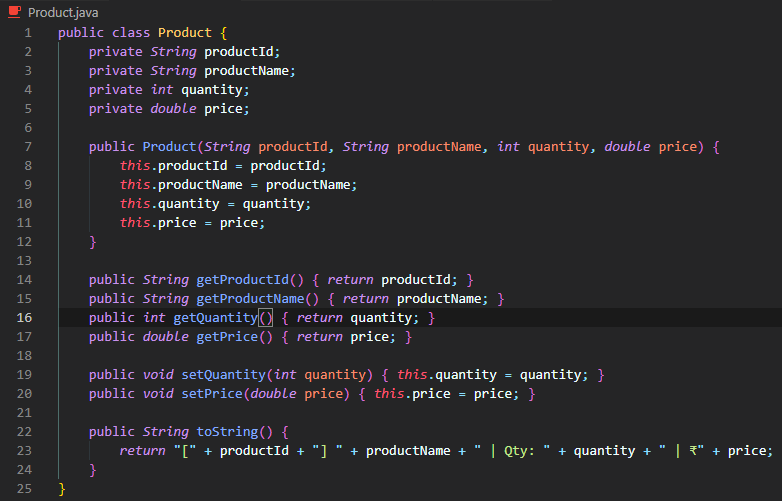
**1. Why are data structures and algorithms essential in handling large inventories?**

In an inventory system, especially one that deals with thousands of items, data structures help in organizing and managing data efficiently. Algorithms play a key role in operations like searching for a product, updating stock, or deleting items. The right combination ensures the system stays fast and reliable as it scales.

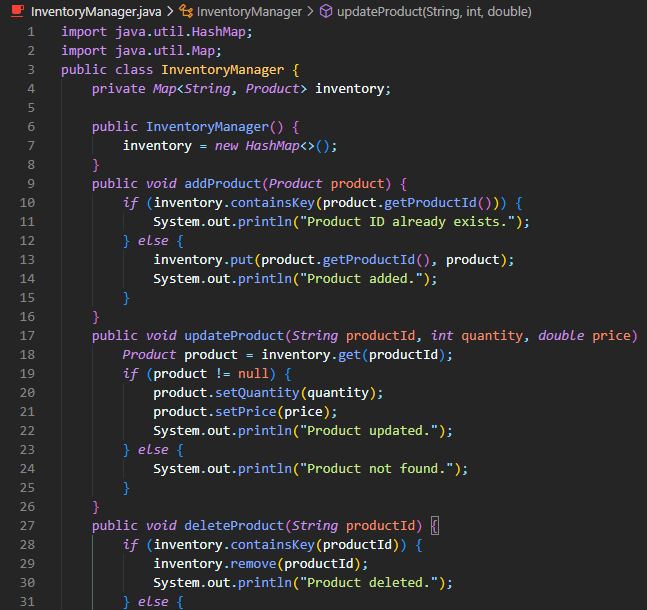
**2. Suitable data structures for this problem:**

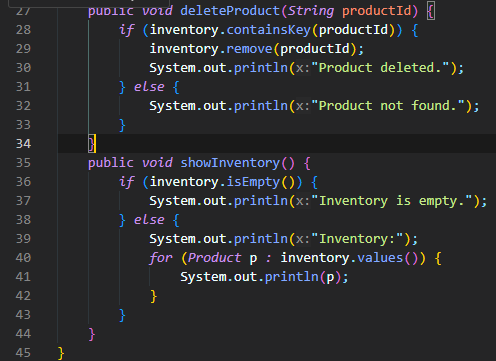
A HashMap is great when we need fast lookups based on product IDs. It provides constant-time performance for add, update, and delete operations. An ArrayList can be useful if we want to maintain the order of products, but it’s slower for search and delete. Combining both can offer balance.

Product.java

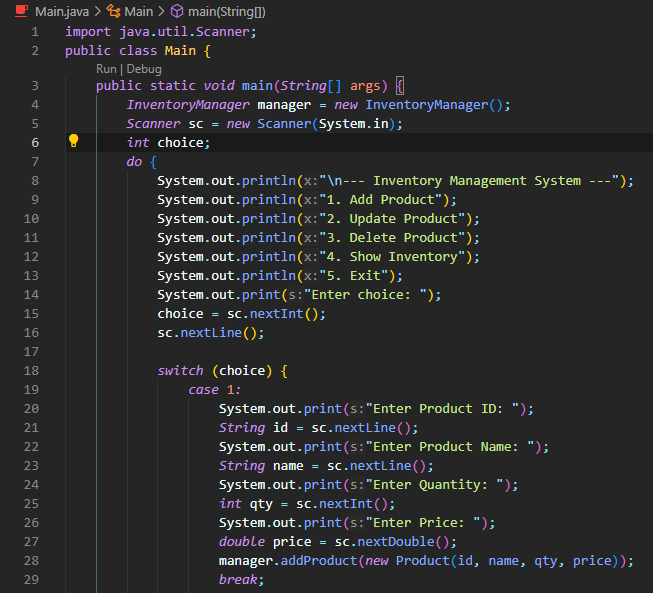


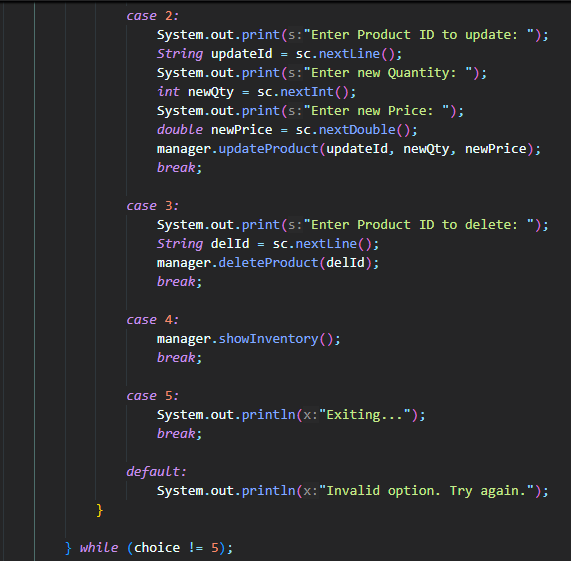
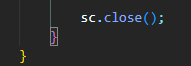
InventoryManager.java



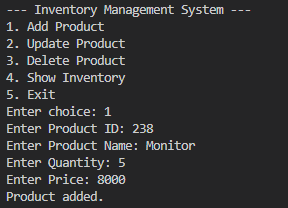


Main.java



Output –



**3. Time complexity of operations:**

* Add (HashMap): O(1)
* Update: O(1)
* Delete: O(1)
* Add/Search/Delete in ArrayList: O(n)

**4. How to optimize these operations:**

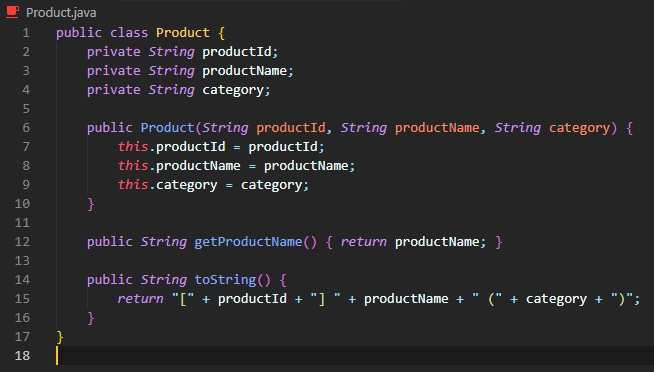
To optimize, we should use a HashMap with productId as the key. For batch updates or sorted views, we can occasionally convert this map into a list. Indexing and caching frequently accessed items can also improve performance.

**Exercise 2:**

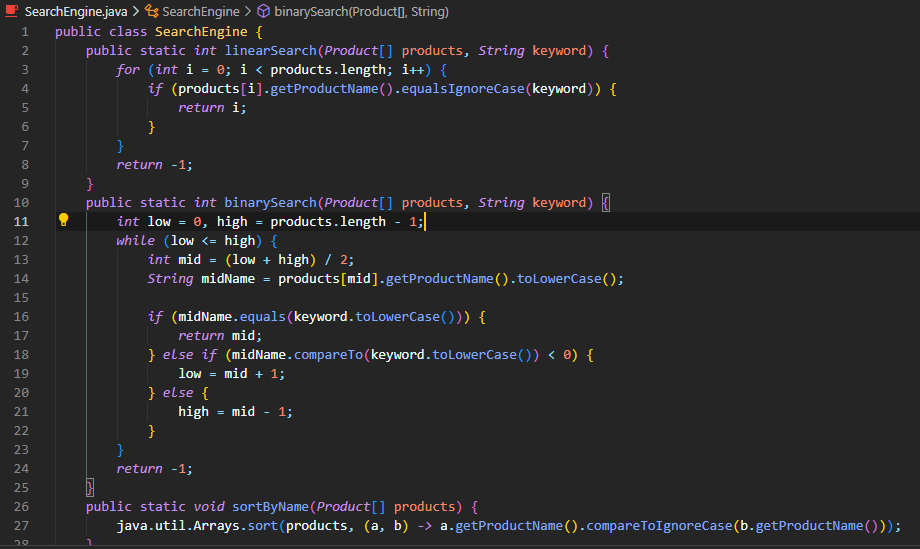
**1. What is Big O notation and why is it useful?**

Big O notation is a way to describe how the performance of an algorithm changes as the input size grows. It helps us compare different algorithms not based on exact time, but on how they scale. It’s like a rough estimate of how "heavy" or "light" an algorithm is.

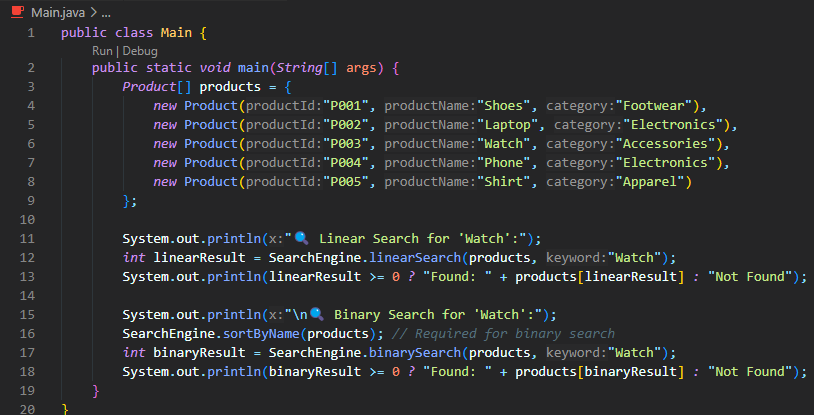
Product.java



SearchEngine.java



Main.java



**2. Best, average, and worst cases for search:**

* **Linear Search:**
  + Best: O(1) (if first item matches)
  + Average: O(n/2) ≈ O(n)
  + Worst: O(n)
* **Binary Search:**
  + Best: O(1)
  + Average: O(log n)
  + Worst: O(log n)

**3. Which search is better?**

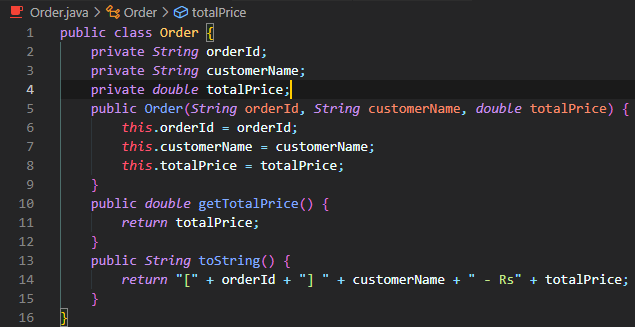
Binary search is better if the data is sorted, as it reduces the search space by half each time. Linear search works on unsorted data but is slower. For a large e-commerce site, binary search is preferred after sorting product data.

**Exercise 3:**

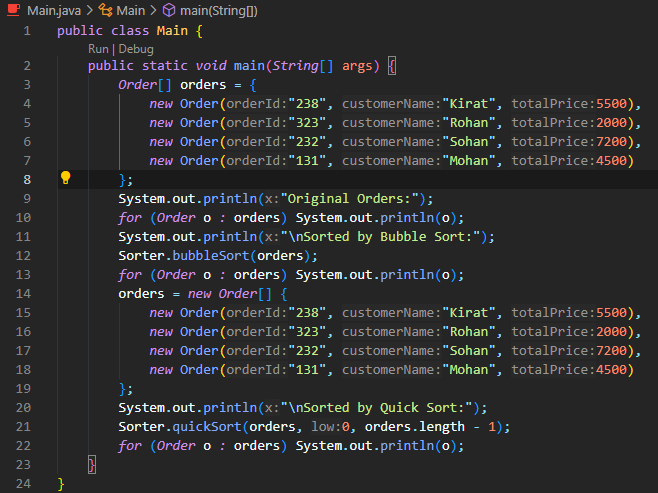
**1. Sorting algorithms explained:**

* **Bubble Sort** compares each pair and swaps if needed. It’s simple but very slow (O(n²)).
* **Insertion Sort** is better on small datasets; it builds the sorted list gradually.
* **Quick Sort** is fast for large datasets, with average case O(n log n).
* **Merge Sort** also gives O(n log n) but uses more memory.

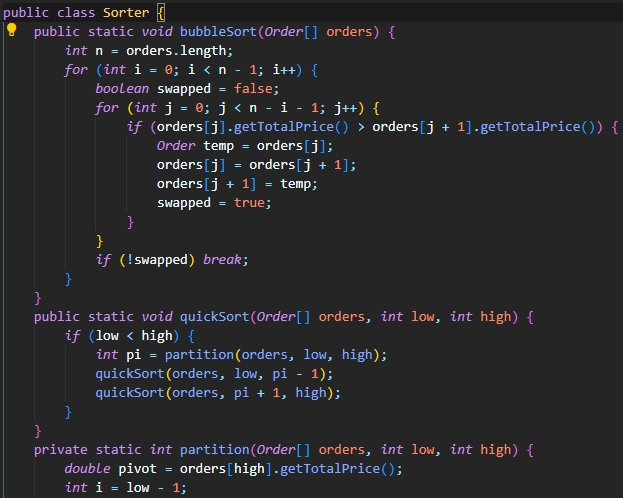
Order.java

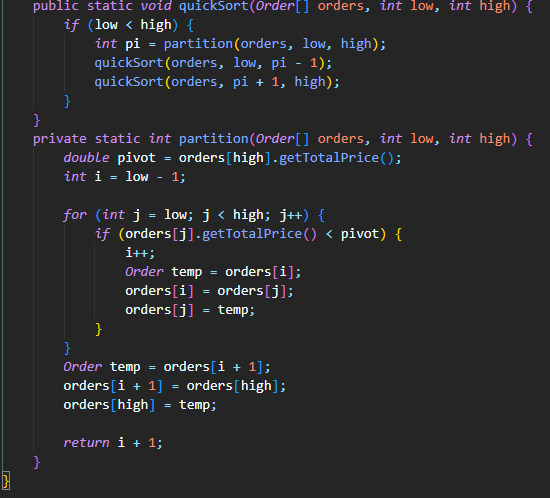


Main.java



Sorter.java





Output –



**2. Why Quick Sort is preferred over Bubble Sort:**

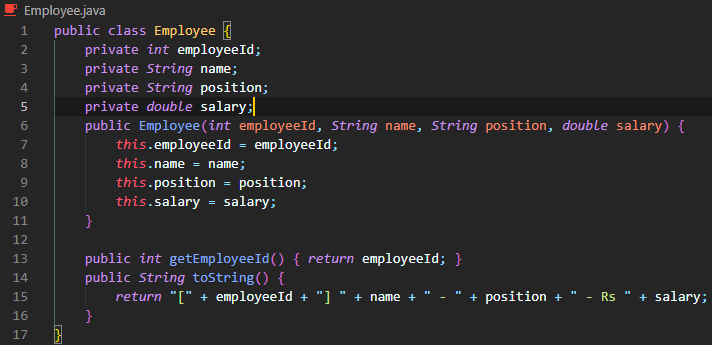
Quick Sort is much faster because it divides the list and processes each part efficiently. Bubble Sort makes repeated unnecessary comparisons, so it’s inefficient for practical use.

**Exercise 4:**

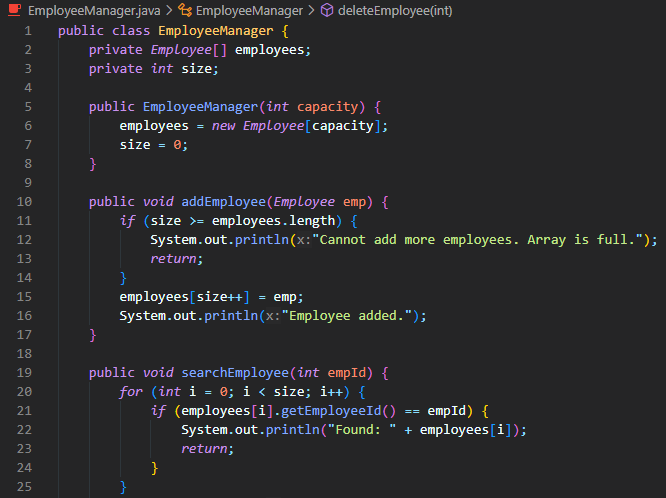
**1. How arrays are represented and their advantages:**

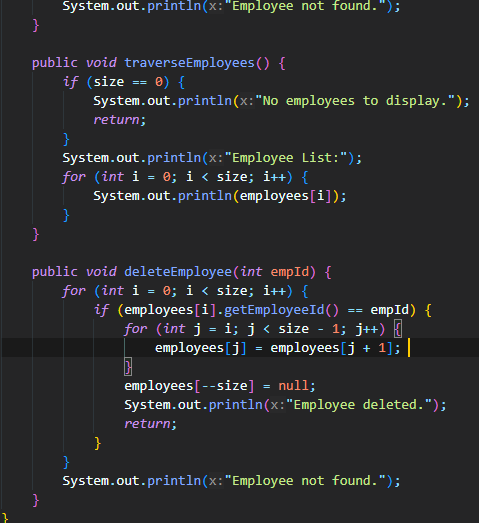
Arrays are stored in continuous memory locations, which allows fast access using indexes (O(1)). They are easy to implement and work well when we know the number of elements in advance.

Employee.java

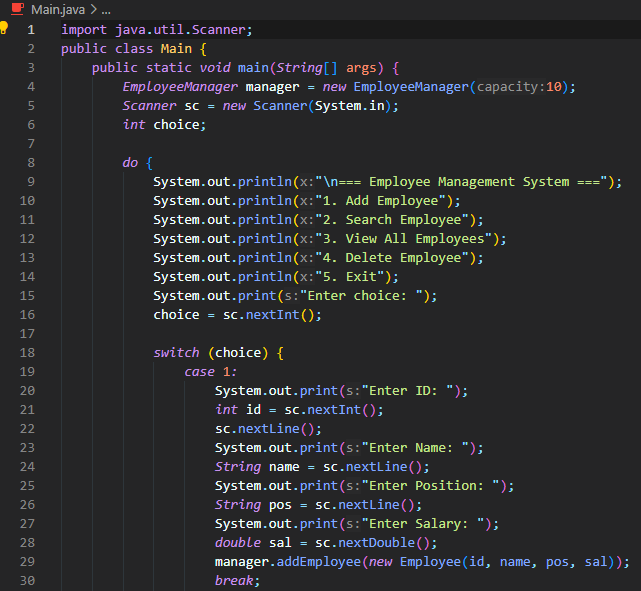


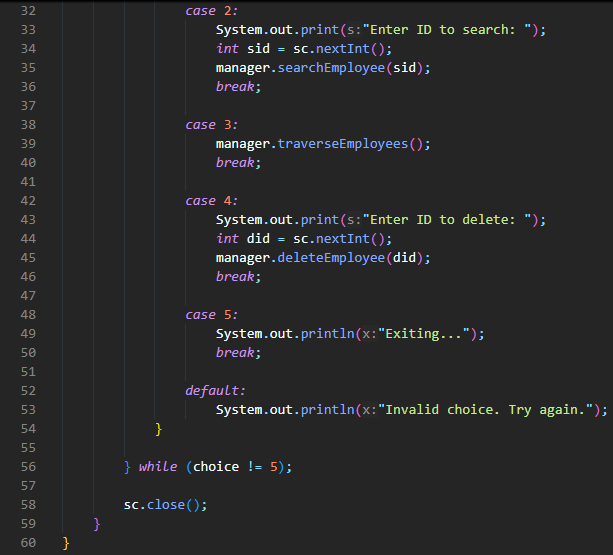
EmployeeManager.java



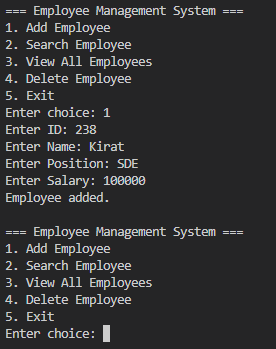


Main.java





Output-



**2. Time complexity of operations:**

* Add: O(1) (if space is available at end)
* Search: O(n)
* Traverse: O(n)
* Delete: O(n) (requires shifting elements)

**3. Limitations of arrays:**

Arrays are fixed in size and not dynamic. Inserting and deleting elements takes more time. For growing datasets, dynamic structures like ArrayList or LinkedList are better.

**Exercise 5:**

**1. Types of linked lists:**

* **Singly Linked List:** Each node has a pointer to the next one. Memory efficient but one-way traversal.
* **Doubly Linked List:** Nodes have both next and previous pointers. Easier to navigate backward but uses more memory.

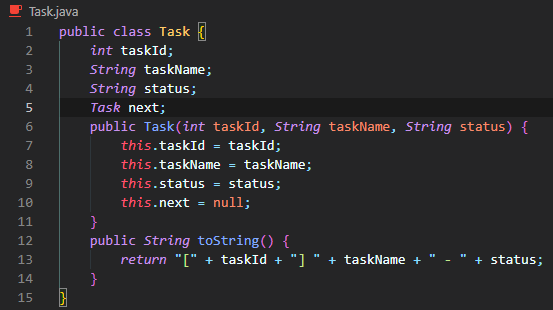
**2. Why use linked lists over arrays?**

Linked lists are dynamic — we don’t need to declare a fixed size. Adding or deleting elements is fast (O(1) if we have the node reference). Arrays need shifting, which makes those operations slower.

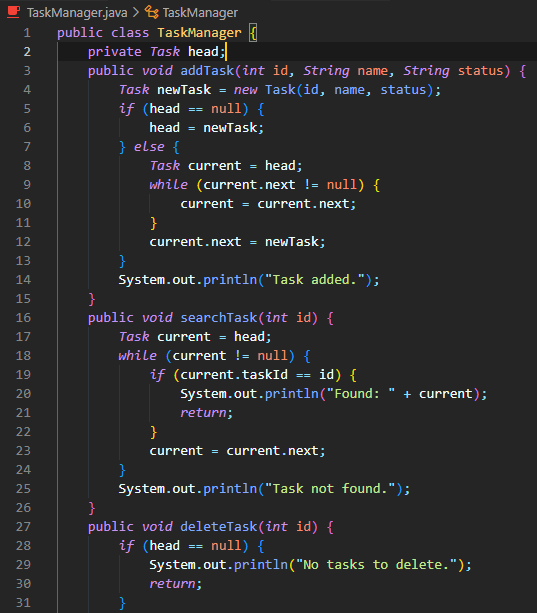
**3. Time complexity:**

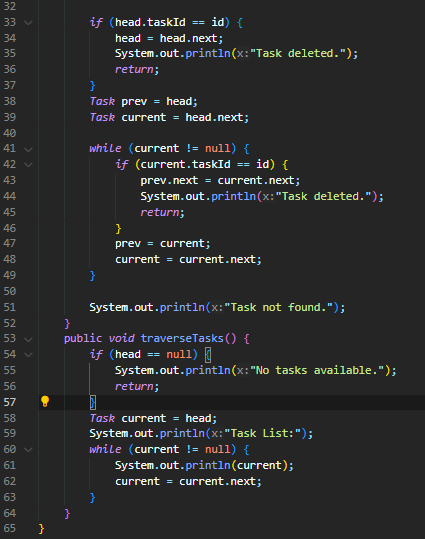
* Add: O(1) (at head) or O(n) (at tail)
* Search: O(n)
* Delete: O(1) with reference; O(n) without
* Traverse: O(n)

Task.java

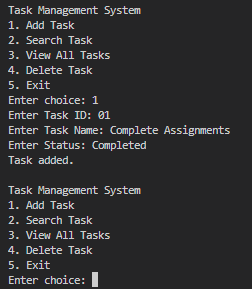


TaskManager.java





Output –

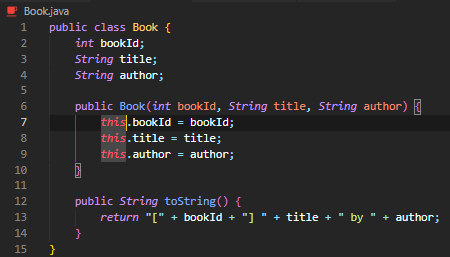


**Exercise 6:**

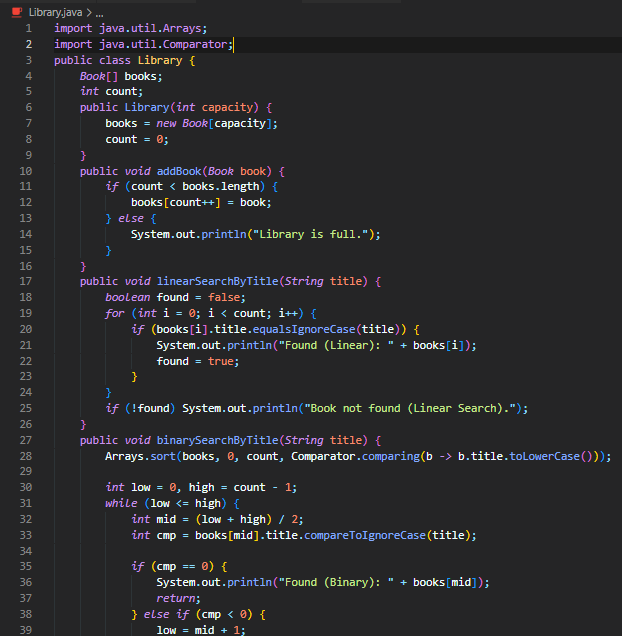
**1. Linear and Binary Search:**

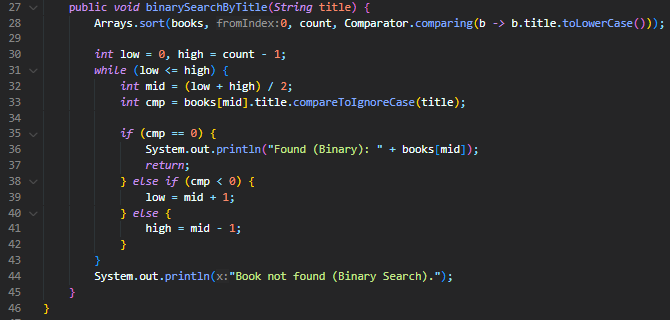
* **Linear Search**: Goes through every item until it finds the match. Good for unsorted data.
* **Binary Search**: Works on sorted data. Divides the list into halves and narrows down the search — very fast.

Book.java

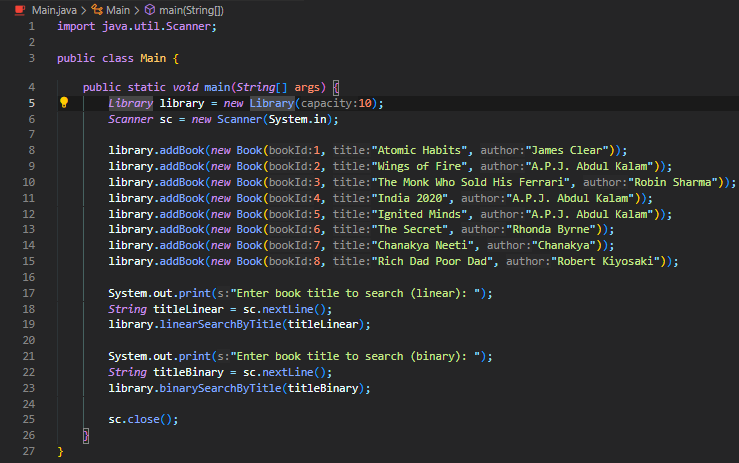


Library.java





Main.java



**2. Time complexity:**

* **Linear Search:** O(n)
* **Binary Search:** O(log n)

**3. When to use what:**

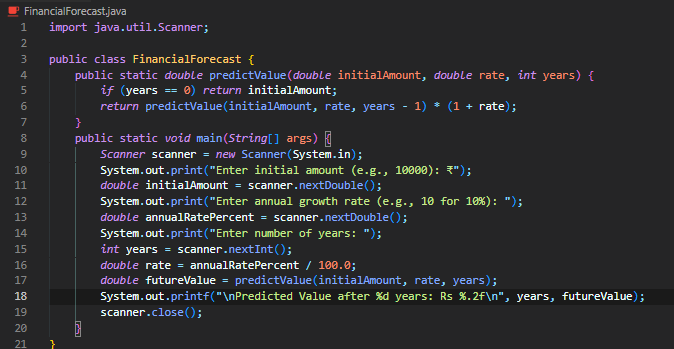
If the book list is small or unsorted (like user-entered data), linear search is fine. For large, sorted book collections, binary search gives a clear performance edge.

**Exercise 7:**

**1. What is recursion and why use it?**

Recursion is when a method calls itself to solve smaller subproblems. It’s a clean way to write code for problems like calculating compound growth or Fibonacci numbers. It also helps reduce code size and complexity when used correctly.

FinancialForecast.java



**2. Time complexity of recursive approach:**

It’s generally O(n), but each recursive call uses stack space, so if not optimized, it can lead to stack overflow or performance issues.

**3. How to optimize recursion:**

We can:

* Use **iteration** instead of recursion to save stack space.
* Use **memoization** to store results of previous calculations.
* Use **tail recursion** (if supported by the language/compiler).