**Java Collections framework:**

Java Collections framework is a collection of libraries that provide implementation of basic data structures and algorithms. For example, if you are solving a big problem or complex problem as part of software development or interview process or maybe solving a competitive programming problem and while you are solving this problem there's an intermediate step where you need to sort certain items.

One way is to write our own sort function. There are many problems with this approach. The problems are since you are writing your own sort function, it's going to take time. It may not be efficient and there are chances of errors.

So the idea is to use sort function readily available as part of collections framework.

Another example can be say you want to use hash tables while you're solving a big problem. Hash tables are efficient implementations or efficient data structures which have search insert and delete operations really really fast and say while you are solving this big problem, you need search insert and delete operations.

So one way is to implement your own hash and again you have the same problems and another way is to use Library implementation and all these basic data structures like hash, linked list, array, queue, dequeue, they are implemented as part of this collections framework and all these standard basic algorithms like binary search, sorting, fill, shuffle they are also available as part of this collections class.

Java Collections framework can be divided into two parts. The first part is implementation of standard basic data structures.

Here are the classes that we have in this first part, the first class is array list.

* Array list is implementation of dynamic size arrays, arrays which grew automatically in size when you add more and more items, you don't have to worry about specifying the capacity when you're creating them, you don't have to worry about resizing them, deallocating the old memory, allocating the new memory, you don't have to worry about all that, it happens automatically inside this class.
* Next is linked list. Linked list is implementation of doubly linked list.
* Next one is vector. Vector is another dynamic sized array.
* Next one is stack, stack is implementation of standard stack data structure, which is last in first out, LIFO structure. You think about these two classes is, these two are legacy classes, stack class inherits from vector class and the difference between Vector Dynamic sized array and array list dynamic sized arrays is first difference is it's a legacy class and the second difference is it's a thread safe class. Since stack inherits from vector, stack is also thread safe class and as a general practice, it is not recommended to use these two classes, especially in a single-threaded environment because they're going to cause extra overhead, you can always use array list over vector especially in a single-threaded environment and place of stack you can use arrayDeque.
* Next is Hashset, hashset is implementation of hashing. It provides you a collection of keys, where it provides you data structure where you can store collection of keys, you can quickly search insert and delete these keys.
* Next one is TreeSet. TreeSet is again a collection of keys. The difference is hashset implements hashing and treeset implements self-balancing binary search tree like red, black tree.
* The next one is linked hash set, linked hashset stores hash table items or hash items in an insertion order. Hashing, hashset does not guarantee any order, when you insert items, there's no guarantee when you access them, what order they will come out. Linked hashset internally uses hash set and a linked list to ensure that when you traverse these items, you get the items in the same order as you inserted them.
* Next one is linked list. Linked list is coming again here, because linked list implements queue interface as well. What it means is linked list can be used as a queue, you can have queue operations on linked list. Queue is first in first out data structure, whatever comes in first, goes out first.
* ArrayDeque is another class which can be used as a queue. You can use ArrayDeque as a queue and the difference between these two is, it's a linked list implementation, it's an array implementation.
* The next class is priority queue which is implementation of heap data structure. You can have maxheap or minheap data structure implemented by this priority queue class. This class is useful when you want to have a queue with priorities among the items like a doctor's queue where the patients have priorities, the VIP patients, they are treated first. So if items are priorities, then instead of using normal queue, we can use priority queue.
* Next one is linked list again and it can be used as a deque, you can use a linked list as a deque also. What is deque? Deque is doubly ended queue, you can insert and delete items at both the ends and this doubly ended queue can either be implemented using linked list or a ArrayDeque again the difference is it's a linked list implementation and it's an array implementation.
* The next set of classes are hashmap, tree map and LinkedHashMap. These classes are map and these were set. What is the difference between set and map? Set is a collection of keys and map is a collection of key-value pairs. So, for example, if you have only an array of integers or say you have an array of prices of items, you can put them in a set, but if you have key value pairs, for example, every integer also has frequency associated with it, then you need to use map, for example, another example say you have employees IDs, you have employee IDs which are keys and every key has a corresponding employee object. So you need to use a map here. Employee ID is key and employee object is value.

So hashmap is implementation of hashing, it again is the same similar concept as Hashset. They both implement hash table and hashing concepts.

Treemap is implementation of self-balancing binary search tree especially red black tree. It's there on the Java documentation that tree map and treeset, the both implement red black tree, so the advantage that you get with red black tree is you get the items in sorted order and the disadvantage of red black tree over hashing is your search insert and delete times are higher, in hashing it they are big O of one on average, in treemap you have big O of log n.

The next one is linked hashmap, this is similar to linked hashset, in normal hashmap you do not have any order among the items in linked HashMap you have items stored according to their insertion order and also uses hash map and the linked list internally.

The next part here is collections class. It's a single class and it has implementations of basic algorithms like binary search, sort, max, min, reverse, fill and many more and these algorithms are implemented on these collections. For example, sort can be used for list collections where you have an array list or linked list you can use sort function on this list, the list collection is to sort these collections.

**Java Collections Framework Overview**

The Java Collections Framework is a set of classes and interfaces that implement commonly used data structures and algorithms. It helps reduce development time by providing ready-to-use, efficient, and error-free implementations.

Why Use the Collections Framework?

* Avoid writing complex data structures from scratch.
* Ensures optimized and tested performance.
* Reduces chances of bugs.
* Simplifies code readability and maintenance.

Core Components

1. Collections (Set, List, Queue, Deque)

List (Ordered Collection)

* ArrayList: Dynamic arrays (grows automatically).
* LinkedList: Doubly linked list (efficient insertions/deletions).
* Vector (Legacy): Thread-safe dynamic array.
* Stack (Legacy): Inherits from Vector, LIFO structure.

Vector and Stack are legacy and not recommended in single-threaded environments. Use ArrayList or ArrayDeque instead.

Set (Unique Elements)

* HashSet: No order, fast operations using hashing.
* LinkedHashSet: Maintains insertion order (uses HashSet + LinkedList).
* TreeSet: Sorted set (uses Red-Black Tree).

Queue / Deque

* Queue (FIFO):
  + LinkedList (implements Queue)
  + ArrayDeque (array-based queue)
  + PriorityQueue: Based on heap (min-heap or max-heap), used when elements have priority.
* Deque (Double-ended queue):
  + LinkedList and ArrayDeque implement Deque.
  + Allows insertions/deletions at both ends.

Map (Key-Value Pairs)

* HashMap: No order, uses hashing (fast access).
* LinkedHashMap: Maintains insertion order.
* TreeMap: Sorted map (Red-Black Tree), slower than HashMap but keeps keys sorted.

Use Map when you want to associate keys with values (e.g., ID → Employee).

Utility Class: Collections

* A helper class that provides static methods for operations like:
  + sort()
  + binarySearch()
  + reverse()
  + fill()
  + max(), min(), and more.

These work on collections like List, e.g., sorting an ArrayList.

Performance Comparison (Summary)

| Feature | HashSet / HashMap | TreeSet / TreeMap | LinkedHashSet / LinkedHashMap |
| --- | --- | --- | --- |
| Order | No | Sorted | Insertion Order |
| Time (Avg) | O(1) | O(log n) | O(1) |
| Use Case | Fast lookups | Sorted elements | Predictable iteration order |

Best Practices

* Prefer ArrayList over Vector in single-threaded environments.
* Use ArrayDeque instead of Stack and LinkedList for queues.
* Use HashMap for fast lookups; TreeMap if sorted keys are needed.
* Use Collections class for common algorithms to save time and effort.

Java Collections Framework is a collection of libraries that provide implementation of basic data structures and algorithms. The idea is this: say you are solving a big or complex problem as part of software development, an interview process, or maybe a competitive programming problem. While solving this problem, there might be an intermediate step where you need to sort certain items.

One way is to write your own sort function. However, there are many problems with this approach. Writing your own sort function takes time, may not be efficient, and there are chances of errors. So, the idea is to use the sort function readily available as part of the Collections Framework.

Another example: say you want to use hash tables while solving a big problem. Hash tables are efficient data structures with fast search, insert, and delete operations. One way is to implement your own hash table, but again you face the same problems—time consumption, inefficiency, and risk of bugs. Instead, you can use the library implementation. All these basic data structures like hash, linked list, array, queue, and deque are implemented as part of the Collections Framework. Standard algorithms like binary search, sorting, fill, and shuffle are also available in the Collections class.

Java Collections Framework can be divided into two parts. The first part is the implementation of standard basic data structures. These are the classes in this first part:

* **ArrayList**: Implementation of dynamic-size arrays. Arrays that grow automatically in size when you add more items. You don’t have to worry about specifying capacity, resizing, or memory management—it’s handled internally.
* **LinkedList**: Implementation of a doubly linked list.
* **Vector**: Another dynamic-size array.
* **Stack**: Implementation of a standard stack (LIFO - Last In First Out). Stack inherits from Vector, so both are legacy and thread-safe classes. However, using them is not recommended in a single-threaded environment due to extra overhead. Use ArrayList instead of Vector and ArrayDeque instead of Stack.

Next:

* **HashSet**: Implementation of hashing. Stores a collection of keys. You can quickly search, insert, and delete keys.
* **TreeSet**: Also a collection of keys, but implements a self-balancing binary search tree (e.g., red-black tree). Provides sorted order.
* **LinkedHashSet**: Stores hash items in insertion order. Combines HashSet and LinkedList to maintain order.

**LinkedList** comes again because it implements the **Queue** interface, meaning it can be used as a queue (FIFO - First In First Out).

* **ArrayDeque**: Another queue implementation using arrays instead of a linked list.
* **PriorityQueue**: Implementation of a heap data structure (min-heap or max-heap). Useful when queue items have priorities (e.g., VIP patients in a hospital queue).

LinkedList can also be used as a **Deque** (Double-Ended Queue), where you can insert and delete items at both ends. Deque can be implemented using either LinkedList or ArrayDeque (linked list vs array implementation).

Next are the **Map** classes:

* **HashMap**, **TreeMap**, and **LinkedHashMap**.

These classes are for maps, not sets. The difference:

* A **Set** is a collection of keys.
* A **Map** is a collection of key-value pairs.

For example, if you have only item prices, a set works. But if each item has a corresponding quantity or ID, you need a map.

* **HashMap**: Implements hashing. Similar to HashSet but stores key-value pairs.
* **TreeMap**: Implements a self-balancing binary search tree (e.g., red-black tree). Items are stored in sorted order. Search, insert, and delete operations are slower than hashing (O(log n) vs O(1) on average).
* **LinkedHashMap**: Maintains insertion order using a combination of HashMap and LinkedList internally. Normal HashMap does not guarantee any order.

Write the least integer and say this function works on this list and does something. This function does **not** use any LinkedList-specific functions or ArrayList-specific methods. It uses only the functions available in the List interface and Collection interface.

That means you can use this function for a LinkedList, an ArrayList, or even a Vector, because the function names are the same. Suppose you are adding items, removing items, getting items at an index, or inserting an item at an index — these functions are present in all the implementations: ArrayList, LinkedList, Vector. So you can pass any object of these classes to your function and use the same logic.

The function using the List interface does not need to know the internal implementation. If you're passing a List object, you can later change the implementation from a LinkedList to an ArrayList (or vice versa) without modifying your logic. That's the advantage of using an interface that defines common functions — you get flexibility and abstraction.

**Deque Interface**

Queue in Java is typically **First In First Out (FIFO)**. But Deque (Double-Ended Queue) allows insertion and deletion from both ends. The Deque interface provides additional functions such as:

* offerFirst(), offerLast() – Add items at the beginning or end
* peekFirst(), peekLast() – Look at items from the beginning or end
* pollFirst(), pollLast() – Remove and return items from the beginning or end

The Deque interface is implemented by:

* LinkedList
* ArrayDeque

So, you can create a deque using either new LinkedList<>() or new ArrayDeque<>().

**Set Interface**

The Set interface extends the Collection interface and contains functions such as:

* add()
* remove()
* clear()
* size()
* contains()

A major difference is that Set implementations **do not allow duplicates**.

Implementations of Set:

* HashSet
* LinkedHashSet
* TreeSet

Each implementation has a different underlying data structure:

* HashSet uses hashing
* TreeSet uses a **self-balancing binary search tree (BST)**
* LinkedHashSet maintains insertion order using a combination of hashing and a linked list

**SortedSet Interface**

SortedSet adds the requirement that elements must be **comparable**. It is implemented by TreeSet, which uses a self-balancing BST.

When you insert elements into a TreeSet, it ensures:

* Items are stored in sorted order (via in-order traversal)
* Items are compared to determine left/right position or detect duplicates

**Map Interface**

A Map stores **key-value pairs**. It is useful when you want fast lookup, insert, or delete operations based on keys. For example:

* Storing student records by roll number
* Tracking item frequencies

Functions in the Map interface include:

* put(key, value)
* get(key)
* containsKey(key)
* containsValue(value)
* remove(key)
* clear()
* size()

The Map interface is **not** part of the Collection interface hierarchy.

**Implementations of Map**

* HashMap: Uses hashing. Keys are unique. Fast lookup.
* LinkedHashMap: HashMap + maintains insertion order.
* TreeMap: Implements SortedMap and stores keys in sorted order using a self-balancing BST.

Just like:

* HashSet ⇔ Keys only (no values)
* HashMap ⇔ Key-Value pairs

Similarly:

* TreeSet ⇔ Keys in sorted order
* TreeMap ⇔ Key-Value pairs in sorted order
* LinkedHashSet ⇔ Keys with insertion order
* LinkedHashMap ⇔ Key-Value with insertion order

**Imports and Usage**

All these interfaces and classes are part of the java.util package. You can import:

* All at once: import java.util.\*;
* Individually:

import java.util.ArrayList;

import java.util.List;

import java.util.Set;

**Note on Primitive Types**

Java Collections **do not support primitive types** (like int, char, etc.). You must use **wrapper classes**:

* int → Integer
* char → Character

Example:

List<Integer> list = new ArrayList<>();

Set<Character> set = new HashSet<>();

This design allows features like returning null (a reference) from functions like poll() when the collection is empty — something not possible with primitives.