Java Collections Framework - Detailed Guide

The Java Collections Framework (JCF) provides a set of interfaces and classes to store, manipulate, and retrieve data efficiently. The diagram illustrates the hierarchy of key interfaces and classes in the framework. Below, we describe each structure, its time complexity, and when to use it

Structure / Class	Random access	Add (end)	Add (middle)	Remove (by index)	Remove (value)	Contains / Search	Iteration
ArrayList	**O(1)**	**Amortized O(1)**	**O(n)**	**O(n)**	**O(n)**	**O(n)**	**O(n)**
LinkedList	**O(n)**	**O(1) (at ends)**	**O(n)**	**O(n)**	**O(n)**	**O(n)**	**O(n)**
Vector	**O(1)**	**Amortized O(1) (synchronized)**	**O(n)**	**O(n)**	**O(n)**	**O(n)**	**O(n)**
Stack (legacy)	**O(1)**	**O(1)**	—	**O(1)**	—	—	**O(n)**
ArrayDeque	**no random access (O(n))**	**O(1)**	**O(n)**	**O(n)**	**O(n)**	**O(n)**	
PriorityQueue	—	**O(log n)**	—	**O(n)**	**O(n)**	**O(n)**	**O(n)**
HashSet	—	**O(1) avg**	—	—	**O(1) avg**	**O(1) avg**	**O(n)**
LinkedHashSet	—	**O(1) avg**	—	—	**O(1) avg**	**O(1) avg**	**O(n) (insertion order)**
TreeSet	—	**O(log n)**	—	—	**O(log n)**	**O(log n)**	**O(n) (sorted order)**

Detailed Descriptions & When to Use

Iterable & Collection (interfaces)

- Purpose: Iterable<T> is the base interface that allows objects to be iterated with for-each. Collection<E> extends Iterable and is the root for groups of objects (List, Set, Queue).
- When to use: Use these interface types for API signatures so you accept multiple implementations.

List (interface)

- **Properties:** Ordered collection, allows duplicates, positional access via index.
- **Common implementations:** ArrayList, LinkedList, Vector.
- **Use when:** You need an ordered sequence with positional access or duplicates allowed.

ArrayList (class)

- **Internal structure:** Resizable array.
- **Performance:** get(i) O(1); add at end amortized O(1); insert/remove at middle O(n).
- **Memory:** Compact contiguous array; requires resizing growth (usually by ~1.5–2×).
- **When to use:** Default go-to list. Use when you need fast random access and mostly append or iterate.
- **Caveat:** Inserting/removing from front or middle is expensive.

LinkedList (class)

- **Internal structure:** Doubly-linked list; implements List and Deque.
- **Performance:** Fast inserts/removes at ends O(1); random access O(n).
- **When to use:** Use when you must frequently insert/remove from both ends or when you have frequent removals via iterators. Rarely faster than ArrayList in practice due to cache inefficiency.
- **Caveat:** Higher memory per element (node overhead).

Vector & Stack (legacy)

- **Vector:** Synchronized, similar API to ArrayList. Avoid unless you require legacy synchronized behavior.
- **Stack:** Extends Vector. Historically used for LIFO, but prefer ArrayDeque or Deque interface as a modern stack.

Queue & Deque (interfaces)

- **Queue:** FIFO semantics; implementations include LinkedList, ArrayDeque, PriorityQueue.
- **Deque:** Double-ended queue — add/remove from both ends. Implementations: ArrayDeque, LinkedList.

ArrayDeque

- **Internal structure:** Circular resizable array.
- **Performance:** O(1) add/remove at both ends; faster than LinkedList in practice for deque operations.
- **When to use:** Use as a queue or stack (push/pop) when you need high performance, and null elements are not required (it does **not** allow null).
- **Preferred over:** LinkedList for queue/stack roles in most cases.

PriorityQueue

- **Internal structure:** Binary heap.
- **Performance:** add O(log n), poll O(log n), peek O(1).
- **Ordering:** Natural ordering or custom Comparator. Not sorted iteration — only head is guaranteed minimal/maximal.
- **When to use:** Scheduling tasks, Dijkstra-like algorithms, event simulation — anytime you need to always access the smallest/largest element quickly.
- **Caveat:** Does not permit null elements.

Set (interface)

- **Properties:** No duplicate elements. No guaranteed order (depends on implementation).
- **Common implementations:** HashSet, LinkedHashSet, TreeSet.

HashSet

- **Internal structure:** Backed by a HashMap.
- **Performance:** Average O(1) for add/remove/contains.
- **When to use:** Fast membership testing, uniqueness enforcement without order requirement.
- **Nulls:** Allows one null.

LinkedHashSet

- **Properties:** Hash-based but maintains insertion (or access) order via linked list.
- **When to use:** When deterministic iteration order is required in addition to O(1) operations.

TreeSet

- **Internal structure:** Red-black tree (TreeMap backing).
- **Performance:** O(log n) for add/remove/contains.
- **Ordering:** Sorted order (natural or via comparator). Supports range operations (subSet, headSet, tailSet).
- **When to use:** Need sorted unique elements, navigable set operations, or range queries.
- **Nulls:** Usually disallowed unless comparator handles null.

Table 1 — Memory & Practical Performance Notes

Comparison	Memory Overhead	Speed Notes	Recommendation
ArrayList vs LinkedList	ArrayList: low overhead (contiguous array). LinkedList: high overhead (node object + 2 pointers per element).	ArrayList faster for iteration & random access; LinkedList only better for frequent insert/remove at ends.	Default to ArrayList unless you truly need frequent end insertions/removals.
ArrayDeque vs LinkedList	ArrayDeque: low overhead (circular array). LinkedList: high overhead.	ArrayDeque typically faster for queue/stack ops due to better cache locality.	Prefer ArrayDeque for most double-ended queue needs.
HashSet vs TreeSet	HashSet: lower overhead, based on hash table. TreeSet: more overhead (tree nodes).	HashSet O(1) avg ops; TreeSet O(log n) but provides sorted order & range queries.	Use HashSet for raw speed, TreeSet for ordering.
Null Elements	HashSet allows 1 null; ArrayList allows multiple nulls; PriorityQueue, ArrayDeque, TreeSet usually disallow.	—	Always check null element policy before choosing.

Table 2 — Concurrency & Thread Safety

Approach / Class	Thread Safety	Performance Impact	When to Use
ArrayList, HashSet, etc.	Not thread-safe	Highest performance in single-threaded contexts	Use in single-threaded or externally synchronized code
Collections.synchronizedList / synchronizedSet	Thread-safe (via synchronized wrapper)	Adds locking overhead	For small-scale multithreading where simple sync is enough
Vector / Stack (legacy)	Thread-safe (synchronized methods)	Higher overhead, outdated	Maintain legacy code requiring these types; avoid in new code
ConcurrentHashMap	Thread-safe (concurrent locks)	High scalability for maps	For high-performance concurrent map usage
CopyOnWriteArrayList	Thread-safe (copy-on-write)	Good for read-heavy, bad for write-heavy workloads	Use when reads dominate and modifications are rare
ConcurrentLinkedQueue	Thread-safe (non-blocking)	Scales well for concurrent queues	Use in producer-consumer models