Java Collections Framework cheatsheet

Collection	Type	Internal Data Structure	Load Factor /	Key Features	Usage Example
ArrayList	List	**Dynamic array** (resizable array backed by `Object[]`)	Default initial capacity: 10, grows by **1.5x**	**Fast random access** (O(1)), **slow insert/ remove** (O(n) for shifting), allows duplicates, maintains	`List<String> list = new ArrayList<>(); list.add("A");`
LinkedList	List, Deque	**Doubly linked list**, where each node contains data and pointers to the previous and next nodes.	No predefined capacity; nodes are dynamical	**Efficient for insertions/removals** (O(1)), slower for random access (O(n)), allows duplicates,	`List<String> list = new LinkedList<>(); list.add("A");`
HashMap	Map	**Hash table** with an array of buckets, where each bucket is a linked list or a red-black tree for large buckets.	Default initial capacity: 16, **load factor: 0.75**,	**Fast key lookups** (average O(1), worst O(n)), unordered, allows one `null` key and multiple `null` values, tree	`Map<String, Integer> map = new HashMap<>(); map.put("A", 1);`
LinkedHashMap	Map	Extends `HashMap` with a **doubly linked list** to maintain insertion/ access order.	Same as `HashMap` (load factor:	Predictable iteration order, slightly slower than `HashMap`, allows `null` keys/	`Map<String, Integer> map = new LinkedHashMap<>(); map.put("A", 1);`
TreeMap	Map, Sorted Map	**Red-black tree** (self-balancing binary search tree).	No load factor; uses tree balancing	**Sorted by key** (O(log n) operations), does not allow `null` keys, **tailMap/headMap**	`Map<String, Integer> map = new TreeMap<>(); map.put("A", 1);`
HashSet	Set	Backed by a `HashMap` where elements are stored as keys with	Same as `HashMap` (load	**No duplicates**, unordered, **fast add/ remove** (average	`Set<String> set = new HashSet<>(); set.add("A");`
LinkedHashSet	Set	Extends `HashSet` with a **linked list** to maintain insertion order.	Same as `HashSet` (load factor:	Predictable iteration order, no duplicates, slightly slower than `HashSet`, allows one	`Set<String> set = new LinkedHashSet<>(); set.add("A");`
TreeSet	Set, Sorted Set	Backed by a `TreeMap` (red-black tree).	No load factor; uses tree balancing	No duplicates, **sorted by natural order** or custom comparator, thread-unsafe, does	`Set<String> set = new TreeSet<>(); set.add("A");`
PriorityQueue	Queue	**Binary heap** (min-heap by default).	No load factor; dynamical ly resizes	**Efficient for priority-based tasks**, allows duplicates, thread-unsafe, **peek/poll**	`Queue<Integer> queue = new PriorityQueue<>(); queue.add(10);`
ArrayDeque	Deque, Queue	**Resizable array** (uses circular indexing for efficient front/back operations).	No load factor; dynamical ly resizes when	**Fast deque operations** (O(1)), no capacity restrictions, faster than `LinkedList` for	`Deque<String> deque = new ArrayDeque<>(); deque.addFirst("A");`
Stack	List, Stack	Extends `Vector` and works as a **LIFO stack**. All methods are synchronized.	Default capacity: 10, resizes by	**Thread-safe**, slower due to synchronization, has been largely replaced	`Stack<String> stack = new Stack<>(); stack.push("A");`

Vector	List	**Synchronized dynamic array** (similar to `ArrayList`).	Default capacity: 10, resizes by **2x**	**Thread-safe**, slower than `ArrayList`, allows duplicates, maintains insertion	`List<String> list = new Vector<>(); list.add("A");`
Hashtable	Map	**Synchronized hash table**, where keys/ values are hashed into buckets.	Default capacity: 11, load factor: 0.75,	**Thread-safe**, slower than `HashMap`, replaced by `ConcurrentHashMap` in most modern	`Map<String, Integer> map = new Hashtable<>(); map.put("A", 1);`
ConcurrentHashMap	Map	Uses **segment-based locking** (or bucket-level locking) for thread-safe access.	Default load factor: 0.75; uses adaptive concurren	**Thread-safe**, high performance, allows `null` values but not `null` keys, suitable for concurrent environments.	`Map<String, Integer> map = new ConcurrentHashMap<>(); map.put("A", 1);`
CopyOnWriteArrayList	List	Creates a **new copy of the underlying array** on every modification.	No load factor; grows dynamical	**Thread-safe**, high read performance, slower write operations, suitable for	`List<String> list = new CopyOnWriteArrayList<>();`
CopyOnWriteArraySet	Set	Backed by a `CopyOnWriteArrayList`.	No load factor; grows dynamical	**Thread-safe**, prevents duplicates, suitable for frequent read-heavy scenarios with	`Set<String> set = new CopyOnWriteArraySet<>();`
ConcurrentLinkedQueue	Queue	Implements a **non-blocking queue** using a **linked node structure**.	No load factor; dynamically grows as	**Thread-safe**, suitable for producer-consumer scenarios, uses CAS (Compare-And-Swap) for atomic updates,	`Queue<String> queue = new ConcurrentLinkedQueue<>(); queue.add("A");`

Additions to Internal Details:

1. **Load Factor**: Determines when resizing occurs, typically at 75% capacity for most hash-based collections.
2. **Tree Balancing**:
   - Red-black trees ensure logarithmic depth for search and insertion/removal operations.
3. **Circular Indexing** (e.g., `ArrayDeque`): Optimizes front/back operations using modulo arithmetic for array indices.
4. **Thread-Safety**:
   - Synchronized classes like `Vector` and `Stack` have high overhead.
   - Modern alternatives like `ConcurrentHashMap` and `CopyOnWriteArrayList` provide efficient thread safety.