Collection Cheat Sheet

```
// List
List<Integer> list = new ArrayList<>(List.of(1, 2));
List<String> linkedList = new LinkedList<>();
List<Integer> vector = new Vector<>(); // thread safe but slow compare to
List

list.add(3);
list.remove(1); // Removes by index
list.addAll(List.of(4, 5));
list.clear();
list.contains(2);
list.get(0);
list.size();

linkedList.addFirst("A"); // Adds to the head
linkedList.addLast("B"); // Adds to the tail
linkedList.removeFirst(); // Removes the head
linkedList.removeLast(); // Removes the tail
linkedList.getFirst(); // Retrieves the head without removing
linkedList.getLast(); // Retrieves the tail without removing
linkedList.offer("C"); // Adds to the tail (queue behavior)
linkedList.poll(); // Retrieves and removes the head (queue behavior)
linkedList.peek(); // Retrieves the head without removing (queue behavior)

// Vector (Legacy and thread-safe operations)
vector.removeElementAt(0); // Removes the element at a specific index
vector.insertElementAt(7, 1); // Inserts an element at a specific index
vector.firstElement(); // Retrieves the first element
vector.lastElement(); // Retrieves the last element
vector.capacity(); // Returns the current capacity of the vector
vector.trimToSize(); // Reduces capacity to match current size


List<Integer> list = new ArrayList<>(List.of(1, 2, 3));
list.add(4);
for (Integer i : list) System.out.println(i); // Enhanced for loop
list.forEach(System.out::println); // Lambda
Iterator<Integer> listIterator = list.iterator();
while (listIterator.hasNext()) System.out.println(listIterator.next());

// Set
```

```java
Set<Integer> hashSet = new HashSet<>(Set.of(1, 2));
Set<Integer> linkedHashSet = new LinkedHashSet<>(Set.of(1, 2));
Set<Integer> treeSet = new TreeSet<>(Set.of(1, 2));Set<Integer> hashSet =
new HashSet<>(Set.of(1, 2));
hashSet.add(3);
hashSet.remove(1); // Removes by value
hashSet.addAll(Set.of(4, 5));
hashSet.clear();
hashSet.contains(2);
hashSet.size();

Set<Integer> hashSet = new HashSet<>(Set.of(1, 2, 3));
hashSet.add(4);
for (Integer i : hashSet) System.out.println(i); // Enhanced for loop
hashSet.forEach(System.out::println); // Lambda
Iterator<Integer> setIterator = hashSet.iterator();
while (setIterator.hasNext()) System.out.println(setIterator.next());

// Queue
Queue<String> queue = new LinkedList<>();
Queue<String> priorityQueue = new PriorityQueue<>();
Deque<String> deque = new ArrayDeque<>();

Queue<String> queue = new LinkedList<>();
queue.add("A");
queue.remove(); // Removes head
queue.poll(); // Removes and returns head, or null if empty
queue.peek(); // Returns head without removing
queue.offer("B"); // Adds element to the queue

Deque<String> deque = new ArrayDeque<>();
deque.addFirst("A");
deque.addLast("B");
deque.removeFirst();
deque.removeLast();
deque.peekFirst();
deque.peekLast();

Queue<String> queue = new LinkedList<>(List.of("A", "B", "C"));
queue.add("D");
for (String s : queue) System.out.println(s); // Enhanced for loop
queue.forEach(System.out::println); // Lambda
Iterator<String> queueIterator = queue.iterator();
while (queueIterator.hasNext()) System.out.println(queueIterator.next());

Deque<String> deque = new ArrayDeque<>(List.of("X", "Y", "Z"));
```

```
deque.add("W");
for (String s : deque) System.out.println(s); // Enhanced for loop
deque.forEach(System.out::println); // Lambda
Iterator<String> dequeIterator = deque.iterator();
while (dequeIterator.hasNext()) System.out.println(dequeIterator.next());

// Map
Map<Integer, String> hashMap = new HashMap<>(Map.of(1, "A", 2, "B"));
Map<Integer, String> linkedHashMap = new LinkedHashMap<>(Map.of(1, "A", 2,
"B"));
Map<Integer, String> treeMap = new TreeMap<>(Map.of(1, "A", 2, "B"));
Map<Integer, String> hashtable = new Hashtable<>(Map.of(1, "A", 2, "B"));

hashMap.put(3, "C");
hashMap.remove(1); // Removes by key
hashMap.putAll(Map.of(4, "D", 5, "E"));
hashMap.clear();
hashMap.containsKey(2);
hashMap.containsValue("A");
hashMap.get(2);
hashMap.size();
hashMap.keySet(); // Returns all keys
hashMap.values(); // Returns all values
// LinkedHashMap (Order-sensitive operations)
linkedHashMap.replace(2, "B", "NewB"); // Conditional replace
linkedHashMap.computeIfAbsent(6, k -> "F"); // Adds new entry if key is
absent
linkedHashMap.computeIfPresent(2, (k, v) -> v + " Updated"); // Updates
value if key exists

// TreeMap (Navigable map operations)
treeMap.firstKey(); // Returns the lowest key
treeMap.lastKey(); // Returns the highest key
treeMap.headMap(3); // Keys less than 3
treeMap.tailMap(3); // Keys greater than or equal to 3
treeMap.subMap(2, 4); // Keys between 2 (inclusive) and 4 (exclusive)

// Hashtable (Thread-safe operations)
hashtable.replace(2, "B", "NewB"); // Conditional replace
hashtable.computeIfAbsent(6, k -> "F"); // Adds new entry if key is absent
hashtable.computeIfPresent(2, (k, v) -> v + " Updated"); // Updates value if
key exists


Map<Integer, String> hashMap = new HashMap<>(Map.of(1, "A", 2, "B", 3,
"C"));
```

```
hashMap.put(4, "D");
for (Map.Entry<Integer, String> entry : hashMap.entrySet())
System.out.println(entry.getKey() + "=" + entry.getValue());
hashMap.forEach((k, v) -> System.out.println(k + "=" + v)); // Lambda
Iterator<Map.Entry<Integer, String>> mapIterator =
hashMap.entrySet().iterator();
while (mapIterator.hasNext()) {
    Map.Entry<Integer, String> entry = mapIterator.next();
    System.out.println(entry.getKey() + "=" + entry.getValue());
}


Sort
// Arrays.sort() - Sorts array in ascending order
int[] arr = {3, 1, 4, 1, 5, 9};
Arrays.sort(arr);  // Sorts array in ascending order

// Arrays.sort() with Comparator - Custom sorting (descending order)
Integer[] arr = {3, 1, 4, 1, 5, 9};
Arrays.sort(arr, (a, b) -> b - a);  // Custom comparator for descending
order

// List.sort() - Sorts List in ascending order
List<Integer> list = Arrays.asList(3, 1, 4, 1, 5, 9);
list.sort((a, b) -> a - b);  // Sorts list in ascending order

// Collections.sort() - Sorts List in ascending order
List<Integer> list = Arrays.asList(3, 1, 4, 1, 5, 9);
Collections.sort(list);  // Sorts list in ascending order

// Collections.sort() with Comparator - Custom sorting (descending order)
List<Integer> list = Arrays.asList(3, 1, 4, 1, 5, 9);
Collections.sort(list, (a, b) -> b - a);  // Sorts list in descending order

// Stream.sorted() - Sorts stream in ascending order
List<Integer> list = Arrays.asList(3, 1, 4, 1, 5, 9);
list.stream().sorted().forEach(System.out::println);  // Sorted in ascending
order

// Stream.sorted() with Comparator - Custom sorting (descending order)
List<Integer> list = Arrays.asList(3, 1, 4, 1, 5, 9);
list.stream().sorted((a, b) -> b - a).forEach(System.out::println);  //
Sorted in descending order

// Arrays.parallelSort() - Sorts array in parallel for better performance
int[] arr = {3, 1, 4, 1, 5, 9};
```

```
Arrays.parallelSort(arr);  // Sorts array in parallel

// TreeSet - Automatically sorts elements in ascending order (or custom
comparator)
Set<Integer> set = new TreeSet<>(Arrays.asList(3, 1, 4, 1, 5, 9));
System.out.println(set);  // Automatically sorted in ascending order

// PriorityQueue - Sorts elements using heap (min-heap by default)
Queue<Integer> pq = new PriorityQueue<>(Arrays.asList(3, 1, 4, 1, 5, 9));
System.out.println(pq.poll());  // Polling returns the smallest element
(min-heap behavior)

System.out.println("PriorityQueue after heap sort: ");
while (!pq.isEmpty()) {
    System.out.print(pq.poll() + " ");  // Poll elements in sorted order
(heap behavior)
}
```