# Java Collections Cheat Sheet

Java Concept Of The Day

## Basics

**What is Java Collection Framework?**

Java Collection Framework is a framework which provides some predefined classes and interfaces to store and manipulate the group of objects. Using Java collection framework, you can store the objects as a List or as a Set or as a Queue or as a Map and perform basic operations like adding, removing, updating, sorting, searching etc... with ease.

**Why Java Collection Framework?**

Earlier, arrays are used to store the group of objects. But, arrays are of fixed size. You can't change the size of an array once it is defined. It causes lots of difficulties while handling the group of objects. To overcome this drawback of arrays, Java Collection Framework is introduced from JDK 1.2.

**Java Collections Hierarchy :**

All the classes and interfaces related to Java collections are kept in java.util package. List, Set, Queue and Map are four top level interfaces of Java collection framework. All these interfaces (except Map) inherit from java.util.Collection interface which is the root interface in the Java collection framework.

## List

**Intro :**

- List is a sequential collection of objects.
- Elements are positioned using zero-based index.
- Elements can be inserted or removed or retrieved from any arbitrary position using an integer index.

**Popular Implementations :**

- ArrayList, Vector And LinkedList

**Internal Structure :**

- **ArrayList :** Internally uses re-sizable array which grows or shrinks as we add or delete elements.
- **Vector :** Same as ArrayList but it is synchronized.
- **LinkedList :** Elements are stored as Nodes where each node consists of three parts – Reference To Previous Element, Value Of The Element and Reference To Next Element.

**Null Elements :**

- **ArrayList :** Yes
- **Vector :** Yes
- **LinkedList :** Yes

**Duplicate Elements :**

- **ArrayList :** Yes
- **Vector :** Yes
- **LinkedList :** Yes

**Order Of Elements :**

- **ArrayList :** Insertion Order
- **Vector :** Insertion Order
- **LinkedList :** Insertion Order

**Synchronization :**

- **ArrayList :** Not synchronized
- **Vector :** Synchronized
- **LinkedList :** Not synchronized

**Performance :**

- **ArrayList :** Insertion -> O(1) (if insertion causes restructuring of internal array, it will be O(n)), Removal -> O(1) (if removal causes restructuring of internal array, it will be O(n)), Retrieval -> O(1)
- **Vector :** Similar to ArrayList but little slower because of synchronization.
- **LinkedList :** Insertion -> O(1), Removal -> O(1), Retrieval -> O(n)

**When to use?**

- **ArrayList :** Use it when more search operations are needed then insertion and removal.
- **Vector :** Use it when you need synchronized list.
- **LinkedList :** Use it when insertion and removal are needed frequently.

## Queue

**Intro :**

- Queue is a data structure where elements are added from one end called tail of the queue and elements are removed from another end called head of the queue.
- Queue is typically FIFO (First-In-First-Out) type of data structure.

**Popular Implementations :**

- PriorityQueue, ArrayDeque and LinkedList (implements List also)

**Internal Structure :**

- **PriorityQueue :** It internally uses re-sizable array to store the elements and a Comparator to place the elements in some specific order.
- **ArrayDeque :** It internally uses re-sizable array to store the elements.

**Null Elements :**

- **PriorityQueue :** Not allowed
- **ArrayDeque :** Not allowed

**Duplicate Elements :**

- **PriorityQueue :** Yes
- **ArrayDeque :** Yes

**Order Of Elements :**

- **PriorityQueue :** Elements are placed according to supplied Comparator or in natural order if no Comparator is supplied.
- **ArrayDeque :** Supports both LIFO and FIFO

**Synchronization :**

- **PriorityQueue :** Not synchronized
- **ArrayDeque :** Not synchronized

**Performance :**

- **PriorityQueue :** Insertion -> O(log(n)), Removal -> O(log(n)), Retrieval -> O(1)
- **ArrayDeque :** Insertion -> O(1) , Removal -> O(n), Retrieval -> O(1)

**When to use?**

- **PriorityQueue :** Use it when you want a queue of elements placed in some specific order.
- **ArrayDeque :** You can use it as a queue OR as a stack.

## Set

**Intro :**

- Set is a linear collection of objects with no duplicates.
- Set interface does not have its own methods. All its methods are inherited from Collection interface. It just applies restriction on methods so that duplicate elements are always avoided.

**Popular Implementations :**

- HashSet, LinkedHashSet and TreeSet

**Internal Structure :**

- **HashSet :** Internally uses HashMap to store the elements.
- **LinkedHashSet :** Internally uses LinkedHashMap to store the elements.
- **TreeSet :** Internally uses TreeMap to store the elements.

**Null Elements :**

- **HashSet :** Maximum one null element
- **LinkedHashSet :** Maximum one null element.
- **TreeSet :** Doesn't allow even a single null element

**Duplicate Elements :**

- **HashSet :** Not allowed
- **LinkedHashSet :** Not allowed
- **TreeSet :** Not allowed

**Order Of Elements :**

- **HashSet :** No order
- **LinkedHashSet :** Insertion order
- **TreeSet :** Elements are placed according to supplied Comparator or in natural order if no Comparator is supplied.

**Synchronization :**

- **HashSet :** Not synchronized
- **LinkedHashSet :** Not synchronized
- **TreeSet :** Not synchronized

**Performance :**

- **HashSet :** Insertion -> O(1), Removal -> O(1), Retrieval -> O(1)
- **LinkedHashSet :** Insertion -> O(1), Removal -> O(1), Retrieval -> O(1)
- **TreeSet :** Insertion -> O(log(n)), Removal -> O(log(n)), Retrieval -> O(log(n))

**When to use?**

- **HashSet :** Use it when you want only unique elements without any order.
- **LinkedHashSet :** Use it when you want only unique elements in insertion order.
- **TreeSet :** Use it when you want only unique elements in some specific order.

## Map

**Intro :**

- Map stores the data in the form of key-value pairs where each key is associated with a value.
- Map interface is part of Java collection framework but it doesn't inherit Collection interface.

**Popular Implementations :**

- HashMap, LinkedHashMap And TreeMap

**Internal Structure :**

- **HashMap :** It internally uses an array of buckets where each bucket internally uses linked list to hold the elements.
- **LinkedHashMap :** Same as HashMap but it additionally uses a doubly linked list to maintain insertion order of elements.
- **TreeMap :** It internally uses Red-Black tree.

**Null Elements :**

- HashMap : Only one null key and can have multiple null values
- LinkedHashMap : Only one null key and can have multiple null values.
- TreeMap : Doesn't allow even a single null key but can have multiple null values.

**Duplicate Elements :**

- **HashMap :** Doesn't allow duplicate keys but can have duplicate values.
- **LinkedHashMap :** Doesn't allow duplicate keys but can have duplicate values.
- **TreeMap :** Doesn't allow duplicate keys but can have duplicate values.

**Order Of Elements :**

- **HashMap :** No Order
- **LinkedHashMap :** Insertion Order
- **TreeMap :** Elements are placed according to supplied Comparator or in natural order of keys if no Comparator is supplied.

**Synchronization :**

- **HashMap :** Not synchronized
- **LinkedHashMap :** Not Synchronized
- **TreeMap :** Not Synchronized

**Performance :**

- **HashMap :** Insertion -> O(1), Removal -> O(1), Retrieval -> O(1)
- **LinkedHashMap :** Insertion -> O(1), Removal -> O(1), Retrieval -> O(1)
- **TreeMap :** Insertion -> O(log(n)), Removal -> O(log(n)), Retrieval -> O(log(n))

**When to use?**

- **HashMap :** Use it if you want only key-value pairs without any order.
- **LinkedHashMap :** Use it if you want key-value pairs in insertion order.
- **TreeMap :** Use it when you want key-value pairs sorted in some specific order.