Collection Framework

Array limitation

- Array size is fixed and array does not grow if its full
- Inserting and deleting elements need reorganization of array
- Array elements are not automatically sorted

Collection framework

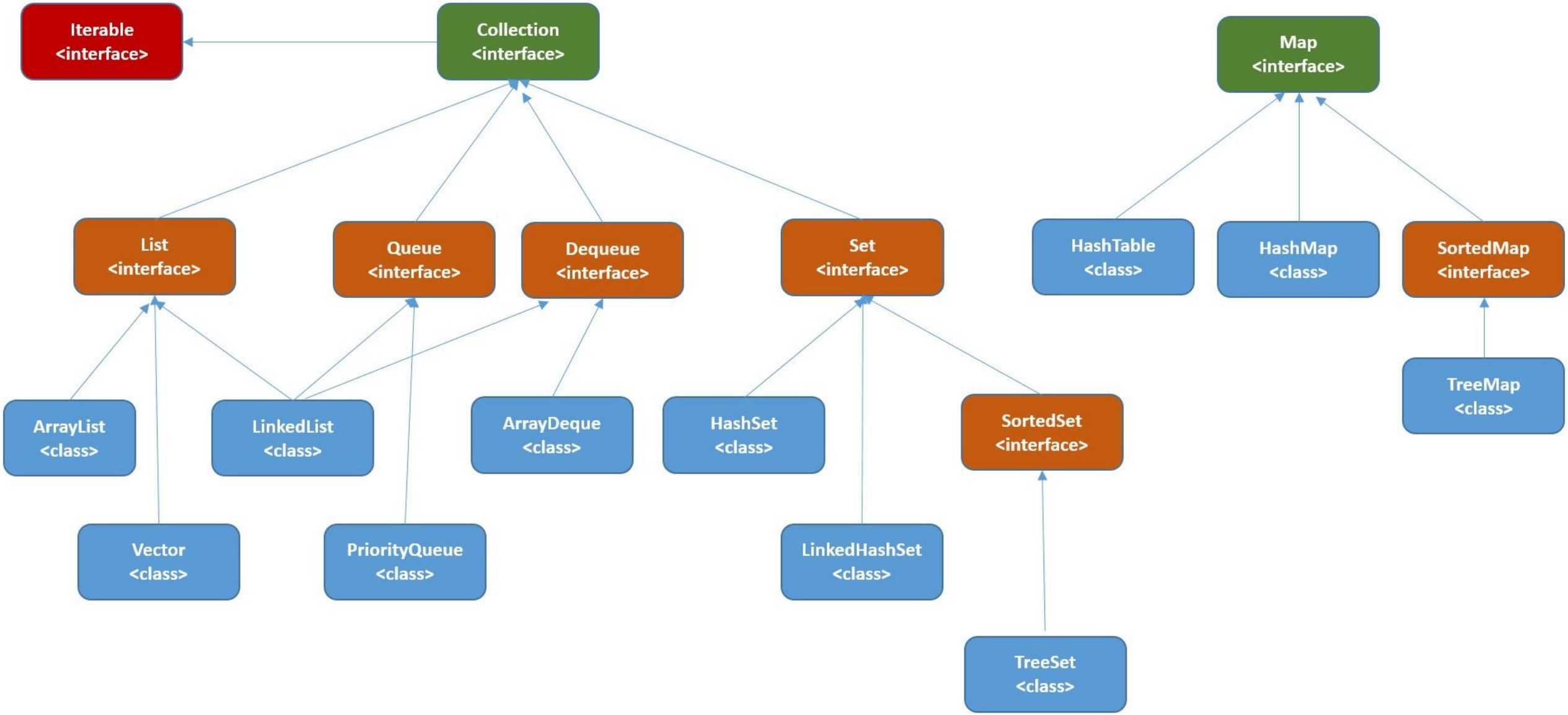
- Collection is group of data elements/objects referred as single unit.
- Collection framework has

  1. **Interfaces**
  2. **Implementations**
  3. **Algorithms**

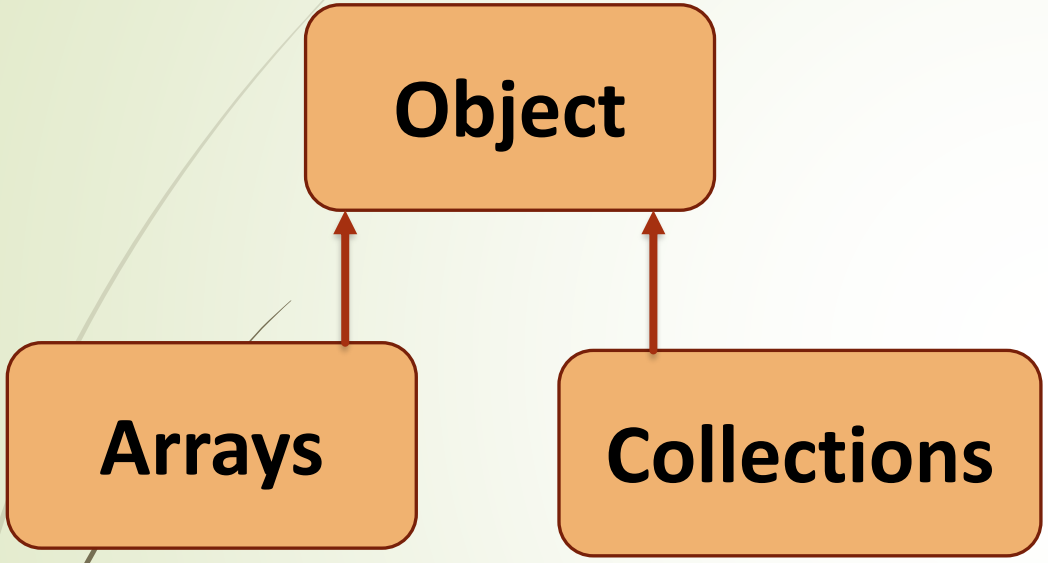
Why Collections?

- Collections are auto growable and shrinkable. It avoid shortage or wastage of memory.
- Collection are generic which can store any object types. Generic -> Parametrized types and are applicable for : classes , interfaces, enums , methods , constructors.
- Collection framework has data structures, Tuned algorithms with quality
- Reduced Programming effort
- Reusability

Collection Framework Hierarchy



Collections Utility classes



- Array**s**: Utility class for common Array operations
  - Searching, Sorting, comparison, toString operations etc
- Collection**s**: Utility class for common Collection operations
  - Searching, Sorting, reverse, synchronization, unmodifiable operations etc

Collection<E> interface

- Its sub-interface of Iterable<T>
- Has behaviours for general purpose operation on collection
- No concrete implementation classes
- Ex. add, addAll, clear, contains, containsAll, remove, removeAll, removeIf, iterataor, isEmpty, stream, toArray etc.

Abstract classes in Collection

- AbstractList
- AbstractCollection
- AbstractQueue
- AbstractSequentialList
- AbstractSet
- AbstractMap

These classes can be extended to implement our own collection extensions

List<E> interface

- Its sub-interface of Collection<E>
- It represents **Ordered Collection.**
- Maintains insertion order
- Supports index based operations
- Allow duplicates values
- Allow null values
- Concrete implementations of List<E>
  - ArrayList<E>, LinkedList<E> (Thread unsafe)
  - Vector<E> (Thread safe)

ListInterator<E> interface

- Its Sub-interface of Iterator<T>
- Its used for traversal of Lists
- Can be used only with List implementations
- We can get ListIterator using below List methods

ListIterator<E>	listIterator()	Returns a list iterator over the elements in this list (in proper sequence).
ListIterator<E>	listIterator(int index)	Returns a list iterator over the elements in this list (in proper sequence), starting at the specified position in the list.

Exceptions related to List or Iterator

- **java.util.NoSuchElementException** : thrown whenever trying to access the element beyond the size of list via Iterator/ListIterator
- **java.lang.IllegalStateException** : thrown whenever trying to remove element before calling next().
- **java.util.ConcurrentModificationException** :thrown when trying to use same iterator/list iterator after modification. This is fail-fast behavior of the Iterator/ListIterator
- **java.lang.IndexOutOfBoundsException** -- thrown while trying to access elements beyond size

Sorting

- Collections can be sorted using Collections **sort** method
- Sorting criteria can be passed to Collections.Sort(..)
- If sorting criteria is not passed externally the it will do implicit or natural sorting
- Important Interfaces for defining sorting criteria
  - Comparable
  - Comparator

Sorting

Natural Sorting (Implicit/ internal ordering)	Custom Sorting (Explicit)
Sorting Criteria will be within the UDT or the objects to be sorted	Sorting Criteria will be outside the
UDT implements java.**lang**.Comparable<T> Must override method public int **compareTo**(T o);	Separate class implements java.**util**.Comparator<T> Must override method public int **compare**(T o1,T o2);
Use java.util.Collections class API Method public static void sort(List<T> l1)	Use java.util.Collections class API Method public static void sort(List<T> l1,Comparator<T> c)

Wild cards in Generics

- **<?>** wild card char ? Denotes any type
- **<? Extends T>** : It defines upper bound i.e any type extending or sub class of T and T itself is allowed
- **<? super T>** : It defined lower bound i.e any super type of T and T itself is allowed

```
List<? extends Employee> empList1 = new ArrayList<PermEmployee>();
List<? extends Employee > empList2 = new ArrayList<Employee>();

List<? super PermEmployee> list1 = new ArrayList<PermEmployee>();
List<? super PermEmployee > list2 = new LinkedList<Employee>();
```

Set<E> interface

- Its sub-interface of Collection<E>
- Does **not** supports index based operations
- Does **not** allow duplicates values
- Allow single null values
- Concrete implementations of Set<E>
  - Hashset<E> (un ordered),
  - LinkedHashSet<E> (ordered),
  - TreeSet<E>(sorted but unordered) though SortedSet<E> interface

Map<K,V> interface

- A map cannot contain duplicate keys; each key can map to at most one value.
- The Map interface provides three collection views, which allow a map's contents to be viewed as a set of keys, collection of values, or set of key-value mappings.
- Map implementation, TreeMap<K,V> is ordered and HashMap<K,V> is un-ordered.
- Map has initial capacity & load factor.
- Allows only one null key reference
- Map implantations are thread unsafe.

HashMap<K,V>

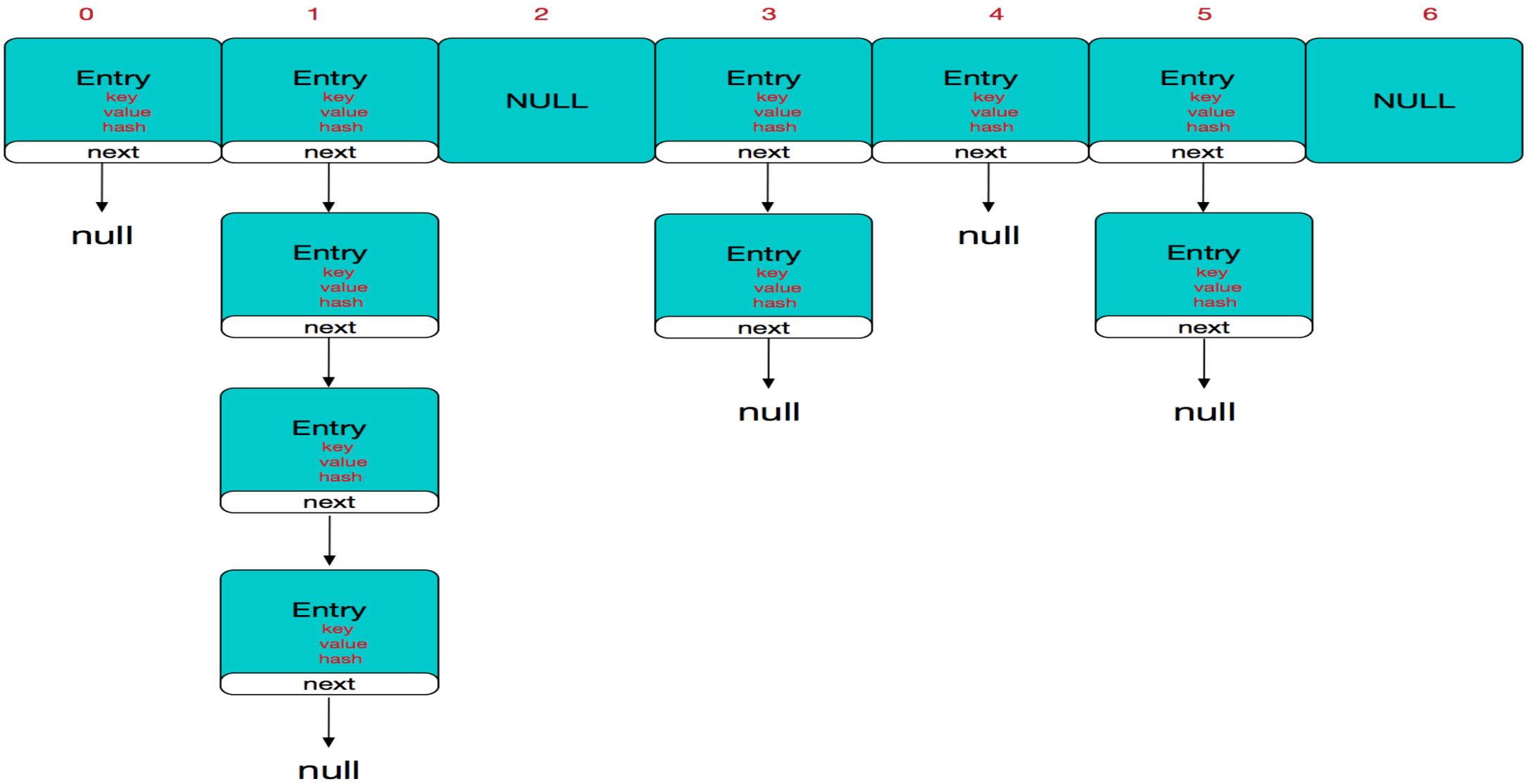
- HashMap has default capacity 16 and load factor 0.75f.
- HashMap works on hashing principle
- HashMap contains an array of Nodes and Node can represent as below

```
static class Node<K,V> implements Map.Entry<K,V> {
    final int hash;
    final K key;
    V value;
    Node<K,V> next;
```

HashMap<K,V>

- **Hashing** is a process of converting an object into integer form by using the method hashCode(). It's necessary to write the hashCode() method properly for better performance of HashMap
- **Buckets:** Bucket is one element of the HashMap array. It is used to store nodes. Two or more nodes can have the same bucket. In that case, a link list structure is used to connect the nodes. Buckets are different in capacity.

Internal Structure of HashMap



HashMap<K,V>

- **Hash collision** degrades the performance of HashMap significantly.
- Java 8 hash elements use balanced trees instead of linked lists after a certain threshold is reached. Which means HashMap starts with storing Entry objects in linked list but after the number of items in a bucket becomes larger than a certain threshold, the bucket will change from using a linked list to a balanced tree, this will improve the worst case

Internal Structure of HashMap