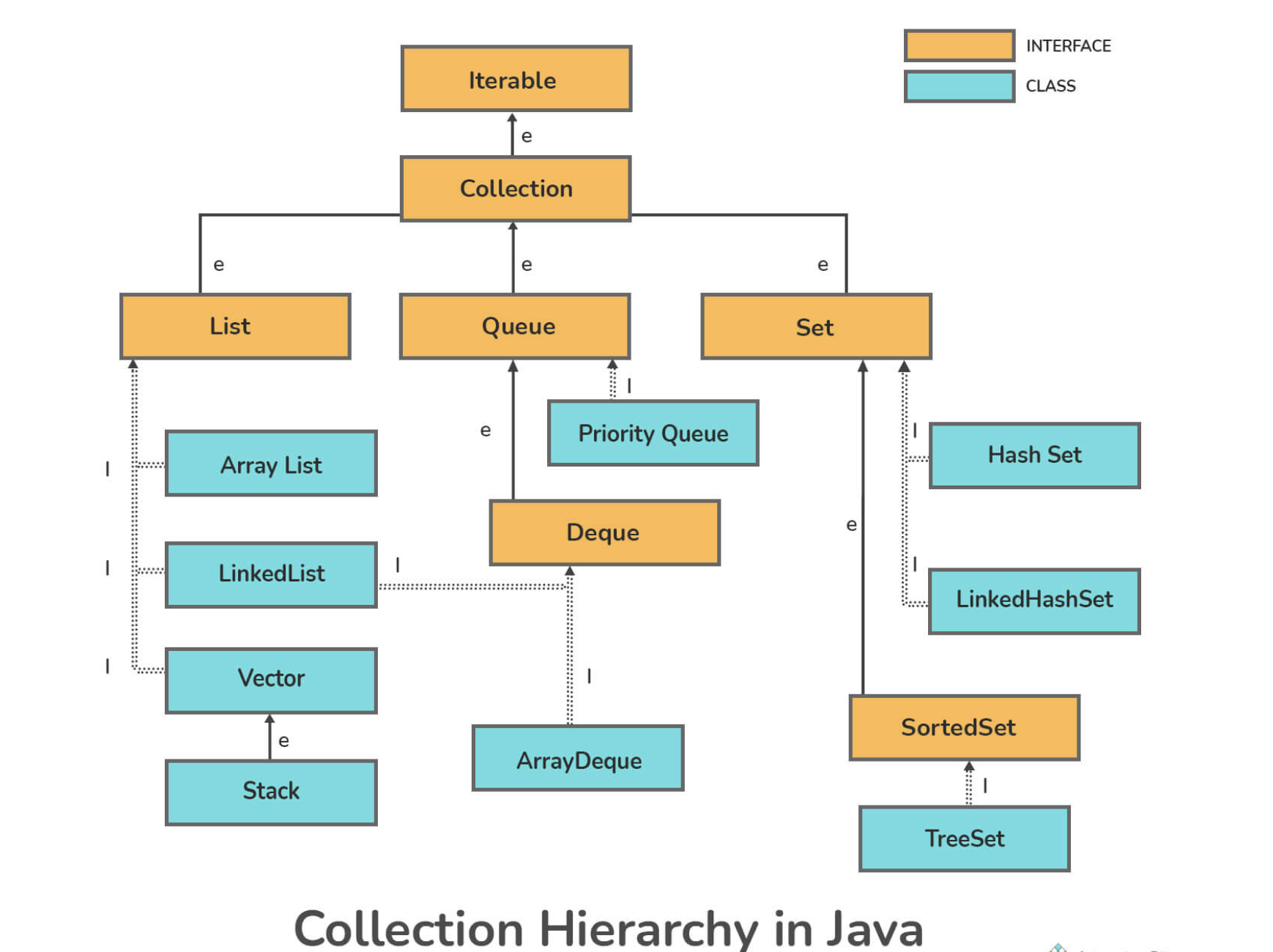
****

**ARRAY**

Arrays.*asList*(arr)

**COLLECTIONS**

Collections.*shuffle*(list)

Collections.*unmodifiableList*(list);

Collections.*unmodifiableMap*(map);

**ARRAYLIST**

arrayList.add("D")

arrayList.add(2, "E");

arrayList.remove(1)

arrayList1.equals(arrayList2)

**LIST**

Duplicate entries allowed

Contents are ordered

list1.add("Jack")

On converting arraylist to linkedlist

linkedList.addAll(arrayList)

**SET**

No duplicate entries

HashSet - contents are unordered

HashTable - key value pair , contents are unordered

TreeSet - unordered

Methods :

hs.add(arr[i])

hashSet.addAll(Arrays.*asList*(str))

hashtable.put(1, "Sam")

hashtable.elements()

On converting hashtable to enumeration

en.hasMoreElements()

en.nextElement()

treeSet.add("Hello");

treeSet.clone()

On converting list to set

set.addAll(list);

**MAP**

Key-value pairs

Methods:

hashMap.put(1, "A")

hashMap.values()

Clone method is only enabled when map is declared like this:

HashMap<K,V> hashMap = new HashMap<>();

Not when - Map<K,V> hashMap = new HashMap<>();

hashMap.clone()

hashMap.containsKey(5)

hashMap.entrySet()

Occurrence :

for(String word : splitString){

if(hashmap.containsKey(word))

hashmap.put(word, hashmap.get(word)+1);

else hashmap.put(word, 1);

}

Clone :

//ENTRYSET  
for(Map.Entry<String, Integer> entry : map1.entrySet()){

map2.put(entry.getKey(), entry.getValue());

}

System.*out*.println(map2);

map3.putAll(map1); //PUTALL

map4 = new HashMap<>(map1); //COPY CONSTRUCTOR

map5 = map1; //ASSIGNMENT

**DIFFERENCES**

| **COLLECTION** | **COLLECTIONS** |
| --- | --- |
| It is an interface | It is a utility class |
| It represents a collection of separate objects as a single entity | It defines useful methods for working with collections |
| Since Java8 collection interface contains static methods, along with abstract and default methods | It contains only static methods |

**Which is preferred - ArrayList/LinkedList ?**

* Arraylist is preferred because of its fast random access, but if there is requirement of insertion/deletion at specific locations then linked list is preferred

| **ARRAYLIST** | **LINKEDLIST** |
| --- | --- |
| Fast random access - O(1) accessing elements by index | Slow random access - O(n) traversing elements from start to reach specific element |
| Slow performance - no insertion/deletion in the middle - O(n) | Fast performance - doubly linked list efficient insertion/deletion in the middle - O(1) |
| Less memory | More memory - node reference (pointers) |

| **ARRAYLIST** | **VECTOR** |
| --- | --- |
| Non-synchronized - Not Thread Safe - Multiple threads running at a time | Synchronized - Thread Safe - one thread running at a time |
| If data limit exceeds, arraylist expands size by 50% | If data limit exceeds, vector expands size by 100% |
| Fast | Slow - when one thread is running it has lock, causing other threads to wait until lock is released |
| Traversal of elements using Iterator | Traversal of elements using Enumeration and Iterator |

| **LIST** | **SET** |
| --- | --- |
| Ordered | Un-ordered |
| Duplicate elements allowed | No duplicate elements |
| Multiple null elements | One null element |
| Elements can be accessed from any position | Elements can’t be accessed from specific position |

| **ITERATOR** | **LISTITERATOR** |
| --- | --- |
| Forward traversal | Forward and backward traversal |
| Used by List, Set, Map, Queue | Used by only List |
| Cannot fetch indexes | Can fetch indexes using nextIndex(), previousIndex() methods |
| Cannot add elements, throws ConcurrentModificationException | Can add elements quickly |
| Methods: next(), hasNext(), remove() | Methods: previous(), next(), hasPrevious(), hasNext(), add() |

| **SET** | **MAP** |
| --- | --- |
| No duplicate values | Unique keys, duplicate values |
| Traversal - keySet(), entrySet() | Traversal - not possible. Need to convert map to set use keySet or entrySet |
| Insertion order not kept, except LinkedHashSet tracks insertion order | Insertion order not tracked, neither does TreeMap or LinkedHashMap |

| **HASHMAP** | **HASHTABLE** |
| --- | --- |
| Not thread-safe, Not synchronized | Thread-safe, synchronized |
| Allows one null key and multiple null values | Null keys not allowed |

| **HASHMAP** | **TREEMAP** |
| --- | --- |
| Implements map interface based on HashTables | Implements map interface based on Tree Structure |
| Map, Clonable, Serializable interfaces are - implemented by HashMap | NavigableMap, Clonable, Serializable interfaces are - implemented by TreeMap |
| Faster - O(1) | Slower - O(log(n)) |
| Single null key, multiple null values | No null key, multiple null values |
| No tracking | Tracking in ascending order |
| Unsorted key-value pair - HashMap | Sorted key-value pair - TreeMap |

| **ITERATOR** | **ENUMERATION** |
| --- | --- |
| Universal cursor | Applies to only legacy classes |
| Changes can be made during traversal - eg. delete() can remove element from collection during traversal | It is read-only interface, no changes can be made during traversal |
| remove() method available | No remove() method |
| Used for traversing - LinkedList, ArrayList, HashSet, TreeSet, HashMap, TreeMap | Used for traversing - HashTable and Vector |