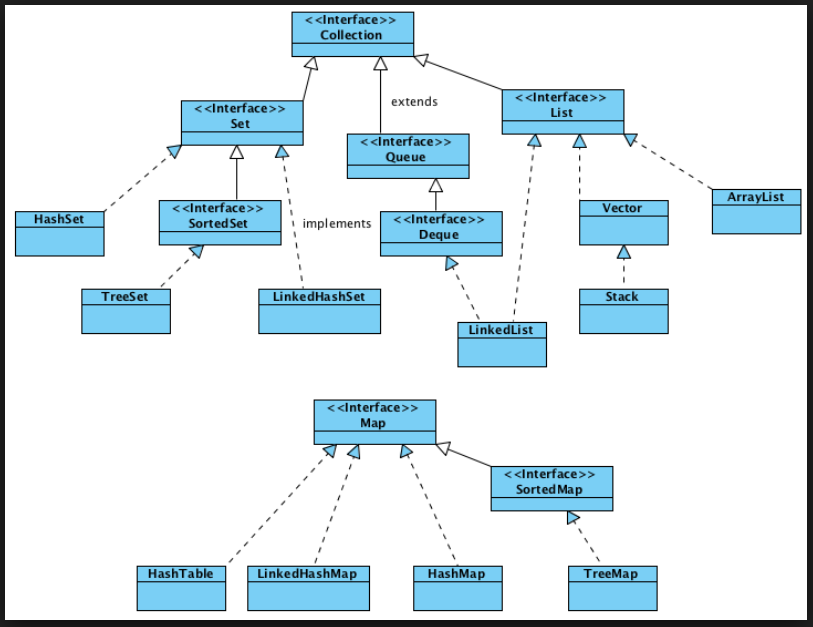
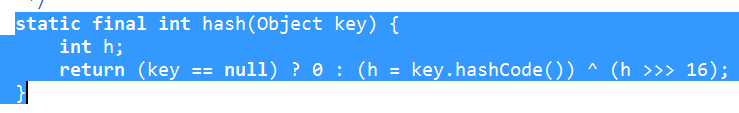
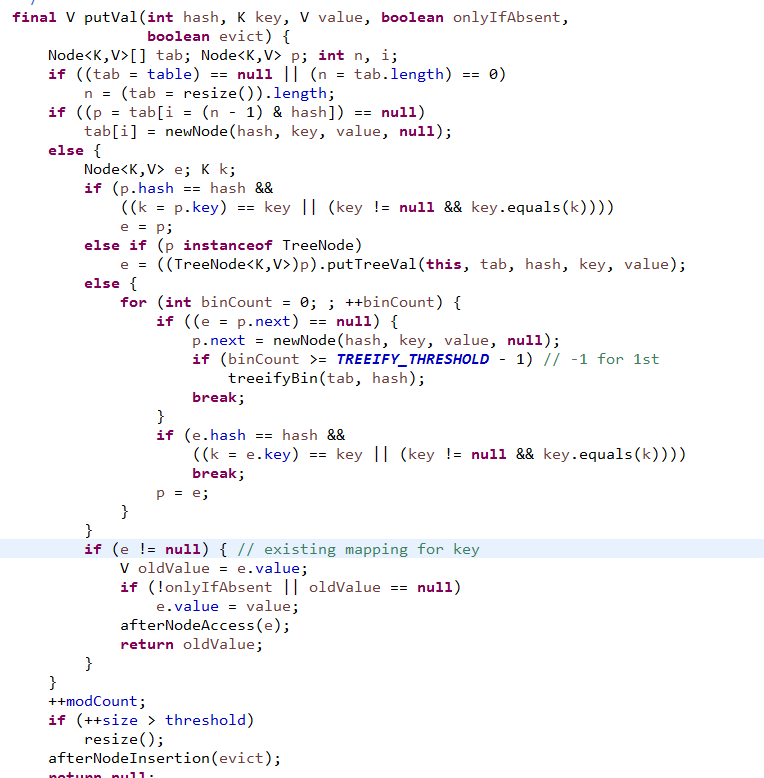
**Collection Framework:** Collection framework gives state-of-art ways to manage group of objects.



1. **Collection interface:** Collection<E> **extends** Iterable<E>
   1. **Iterable** signify all collections are target of for-each loop
   2. **Methods**:
      1. **Modification**: add(obj), addAll(collection), remove(obj), removeAll(collection), removeIf(predicate), retainAll(collection)
      2. **Iteration**: spliterator(), iterator()
      3. **Conversion**: stream(), parallelStream(), toArray([T::new])
      4. **Query**: size(), isEmpty(), clear(), contains(obj), containAll(collection)
2. **List interface**: List<E> **extends** Collection<E>
   1. List defines collection that stores sequence of elements and can be accessed via zero based index.
   2. **Methods**:
      1. **Modification**: add(index, obj), addAll(index, collection), set(index, obj) remove(index), replaceAll()
         1. Add method inserts element where set replaces element
      2. **Iteration**: listIterator([index])
      3. **Query**: get(index), subList(start, end), indexOf(obj), lastIndexOf(obj)
      4. **Others**: sort(comparator)
3. **Set interface**: Set<E> **extends** Collection<E>
   1. Set defines a behavior of a collection that doesn’t allow duplicate elements. add() return false if duplicate is added.
   2. **SortedSet interface**: SortedSet<E> **extends** Set<E>
      1. Defines a set sorted in ascending order.
      2. **Methods**:
         1. **Query**: comparator(), first(), last(), headset(), tailSet(), subset(start, end)
   3. **NavigableSet interface**:
      1. Supports retrieval based on closet match.
      2. **Method**:
         1. **Query**: ceiling(obj), floor(obj), higher(obj), lower(obj)
4. **Queue interface**: Queue<E> **extends** Collection<E>
   1. Declares the behavior of queue.
   2. **Methods**:
      1. **Modification**: poll() [null if empty], remove() [exception if empty], peek() [null if empty], element[exception if empty], offer(obj)
   3. **Deque interface:** Deque<E> **extends** Queue<E>:
      1. Declares behavior of a double-ended queue.
      2. **Methods**:
         1. **Modification**: addFirst(), addLast(), offerFirst(), offerLast(), pollFirst(), pollLast(), pop(), push(), removeFirst(), removeLast(), removeFirstOccurance(), removeLastOccurance()
         2. **Query**: getFirst(), getLast(), peekFirst(), peekLast(),
         3. **Iteration**: decendingIterator
5. **ArrayList**: ArrayList<E> **extends** AbstractList<E> **implements** List<E>, RandomAccess
   1. Default capacity of AL is 10 and incremented 1.5 to resize.
   2. ensureCapacity() increase capacity trimToSize() decrese capacity.
6. **LinkedList**: LinkedList<E> **extends** AbstractSequentialList<E> **implements** List<E>, Deque<E>
   1. Provide doubly linked data structure.
7. **ArraList vs LinkedList**: <http://javarevisited.blogspot.in/2012/02/difference-between-linkedlist-vs.html>
8. **HashSet**: HashSet<E> **extends** AbstractSet<E> **implements** Set<E>
   1. Provide set implementation by using hash table for storage.
   2. Because of hash table implementation add, contains, remove, size shows constant time behavior.
   3. Default size and fill ratio is 16 and .75 respectively.
   4. HashSet doesn’t guarantee the order of elements.
   5. Working: <https://www.geeksforgeeks.org/internal-working-of-sethashset-in-java/>
9. **LinkedHasSet**: LinkedHashSet<E> **extends** HashSet<E> **implements** Set<E>
   1. LinkedHashSet maintains insertion order in a linked list.
10. **TreeSet**: TreeSet<E> **extends** AbstractSet<E> **implements** NavigableSet<E>
    1. Complexity: O(log n)
11. **PriorityQueue**: PriorityQueue<E> **extends** AbstractQueue<E>
    1. Creates a queue that is prioritized based on queue’s comparator.
    2. Initial capacity is 11.
    3. If size is less than 64 grow twice as fast or grow at 50 %
12. **ArrayDeque**: creates dynamic array based queue with initial capacity of 16.
13. **EnumSet**:
    1. EnumSet replaces typical use bit flag system providing type safe and high efficiency.
    2. EnumSet is an abstract class with factory methods: of(), copyOf(), allOf(), noneOf(), complementOf()
    3. A specialized Set implementation for use with enum types. Elements in an enum set must come from a single enum type that is specified, explicitly or implicitly, when the set is created. Enum sets are represented internally as bit vectors. This representation is extremely compact and efficient. The space and time performance of this class should be good enough to allow its use as a high-quality, typesafe alternative to traditional int-based "bit flags." Even bulk operations (such as containsAll and retainAll) should run very quickly if their argument is also an enum set.
    4. <http://javarevisited.blogspot.in/2014/03/how-to-use-enumset-in-java-with-example.html>
14. **Iterator**: Provides support to remove element from collection while iterating.
15. **Splitarators**: Splitarator provides support iterating portion of a sequence in parallel.
    1. **Methods**: tryAdvance(consumer), trySplit(), estimateSize(), forEachRemaining()
16. **Map Interface**: Map<K, V>
    1. map defines unique key and corresponding values
    2. maps don’t implement Iterable/Collection interface
    3. We can get collection view of a map using entrySet(), keyset(), values()
    4. **Methods**:
       1. **Modification**: put(k, v), putAll(k, v), putIfAbsent(k,v), remove(k), remove(k,v), replace(k, vo, vn), replace(k,v), replaceAll(bifunc), compute(k, bifunc), computeIfAbsent(k, bifunc), computeIfPresent(k, bifunc), merge(k,v, bifunc), replaceAll(bifunc)
       2. **Query**: get(k), getOrDefault(k, v), keyset(), values(), isEmpty(), containsKey(k), containsValue(v), entrySet(), size()
       3. **Iteration**: forEach(biconsumer)
17. **SortedMap**: SortedMap<K,V> **extends** Map<K,V>
    1. Maintains entries in ascending order based on key
    2. Null is not allowed.
    3. **Methods**:
       1. **Query**: getFirst(k), getLast(k), comparator(), headMap(k), tailMap(k), submap(k1, k2)
18. **NavigableMap**: NavigableMap<K,V> **extends** SortedMap<K,V>
    1. Support retrieval of entries based on closest match on given key.
    2. **Query**: ceilingEntry(k), floorKey(k), ceilingKey(k), desendingKeySet(), decendingMap()
19. **Map.Entry:** Enables to work with map entry.
    1. **Methods**: getKey(), getValue(), setValue(v), camparingByKey() [comparator compares entries by key], comparingByValue() [comparator compares entries by value]
20. **HashMap**: HashMap<K,V> **extends** AbstractMap<K,V> **implements** Map<K,V>
    1. **Hash**():to prevent against bad hash function this method spreads higher bits into lower.



* 1. Location in hashtable determined by, i = (n - 1) & hash
  2. TREEIFY\_THRESHOLD, 8: The bin count threshold for using a tree rather than list for a bin. Bins are converted to trees when adding an element to a bin with at least this many nodes.
  3. UNTREEIFY\_THRESHOLD, 6: The bin count threshold for untreeifying a (split) bin during a resize operation
  4. MIN\_TREEIFY\_CAPACITY, 64: The smallest table capacity for which bins may be treeified
  5. **putVal**: Hashmap rehashing during putVal may results in infinite loop. <http://javabypatel.blogspot.in/2016/01/infinite-loop-in-hashmap.html>



1. **TreeMap**: TreeMap<K,V> **extends** AbstractMap<K,V> **implements** NavigableMap<K,V>
2. **LinkedHashMap**: LinkedHashMap<K,V> **extends** HashMap<K,V> **implements** Map<K,V>
   1. Maintains linked list of the entries in the map.
   2. **Methods**: removeEldestEntry(Entry): to keep oldest entry return false.
3. **IdentityHashMap**: HashMap but uses reference equality for comparison.
4. **EnumMap**: It is specifically for use with keys of an enum type.
5. **WeakHashMap**: WeakHashMap<K,V> AbstractMap<K,V> **implements** Map<K,V>
   1. WeakHashMap is a hashtable-based implementation of the Map interface, with keys that are of a WeakReference type. An entry in a WeakHashMap will automatically be removed when its key is no longer in ordinary use, meaning that there is no single Reference that point to that key.
6. **Comparator<T>**: It is used to specify ordering of elements in collection.
   1. **Methods**: compare(o1, o2), equals(o)
   2. **Defaults**: reversed(), thenComparing(comparator), thenComparing(func keyExtractor, [func keyComparator])
   3. **Static**: naturalOrder(), nullFirst(comparator), nullLast(comparator), reverseOrder()
   4. **Others**: comparing(function keyselector, [func comparator]), naturalOrder() [defined by behavior of the *Comparable* interface implementation.
   5. **Primitive comparision:** comparingInt(toIntFunction), comparingDouble(toDoubleFunction), comparingLong(toLongFunction), thenComparingInt(toIntFunction), thenComparingDouble(toDoubleFunction), thenComparingLong(toLongFunction),
7. **Collections**: Collections class defines bunch of algorithm can be applied on collections and map class.
   1. **Methods**: binarySearch(list, value, comp), disjoint(a, b), shuffle(), swap(list, in1, in2), synchronizedCollection(collection), unmodifiableCollection(collection), sort(list, [comparator])
8. **Arrays**: Arrays provides utility methods to work with arrays.
   1. **Methods**: asList(arr), binarySearch(arr, val), copyOf(arr, len),copyOfRange(arr, s, e), equals(a1, a2), deepEqual(a1, a2), fill(arr, s, e, v), sort(arr, [comp]), parallelSort(arr, [comp]), stream(), parallelPrefix(arr, BinaryOp) [culmilative operation applied on previous elements]
   2. Sort() method uses tim sort for object types with binary insertion sort threshold of 32
      1. for primitive array uses mixed strategy with dual pivot quicksort threshold of 286 and insertion sort threshold of 286.