Collections Framework

1. Collection <E> interface (E – the type of elements in the collection)

a) If we want to represent a group of individual objects as single entity then we should go for Collection.

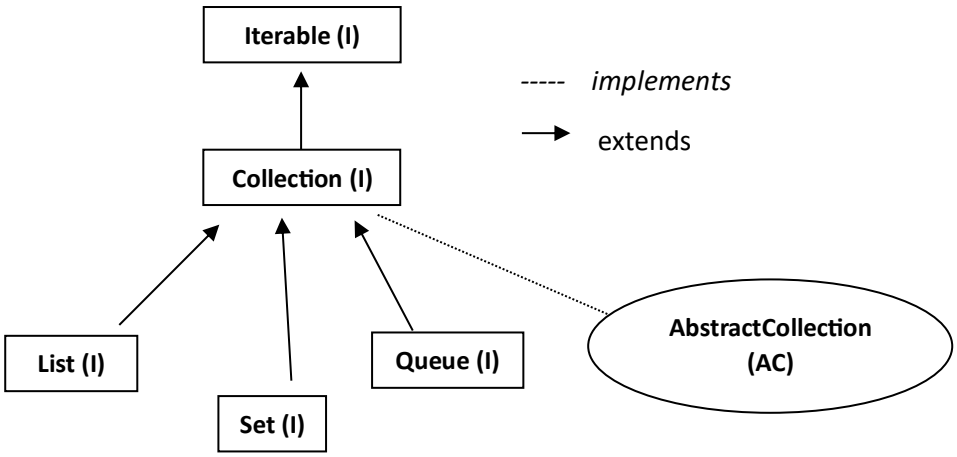
b) Collection interface acts as the root interface of the collections framework & it contains most commonly used methods for any collection object.

c) Collection interface is typically used to pass Collections around & manipulate them where maximum generality is desired.

d) The JDK doesn't provide any direct implementations of Collection interface.

e) **Methods in Collection interface**

1. **boolean add (E e) :** To add an object to the collection
2. **boolean addAll (Collection<? Extends E> c) :** To add a group of objects in the Collection
3. **boolean remove (Object o) :** removes a particular object from the collection
4. **boolean removeAll (Collection<?> c) :** removes a group of objects from the collection
5. **boolean retainAll (Collection<?> c) :** except a particular group of objects, remove all other objects
6. **void clear() :** removes all the objects/elements from the collection
7. **boolean contains (Object o) :** checks a particular object is available or not
8. **boolean contains (Collection<?> c) :** checks a group of objects available or not
9. **boolean isEmpty () :** checks if the collection is empty or not
10. **int size () :** returns the number of objects in the collection
11. **Object [] toArray () :** returns an array containing all of the elements in the collection
12. **Iterator<E> iterator () :** returns an iterator over the elements in the collection i.e., to get object one by one from the collection
13. **default boolean removeIf (Predicate<? super E> filter) :** removes all of the elements of this collection that satisfy the given predicate.
14. **default Stream<E> parallelStream () :** returns a possibly parallel Stream with this collection as its source.
15. **default Stream<E> stream () :** returns a sequential stream with this collection as its source
16. **default Spliterator<E> spliterator () :** creates a Spliterator over the elements in this collection



2. List <E> interface extends Collection<E> (E – the type of elements in this list)

[Ordered Collection]

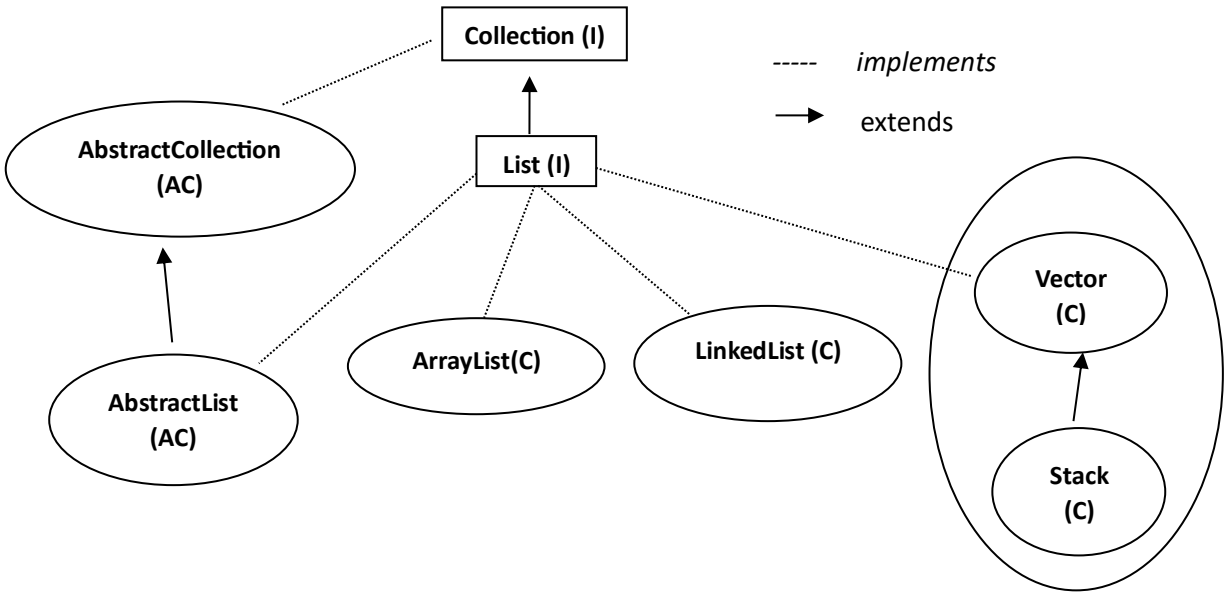
a) Child interface of Collection interface

b) If we want to represent a group of individual objects as single entity where duplicates are allowed & insertion order must be preserved then we should go for List interface.

c) We can preserve insertion order via index & we can differentiate duplicate objects by using index, hence index will play very important role in List.

d) Methods in List interface

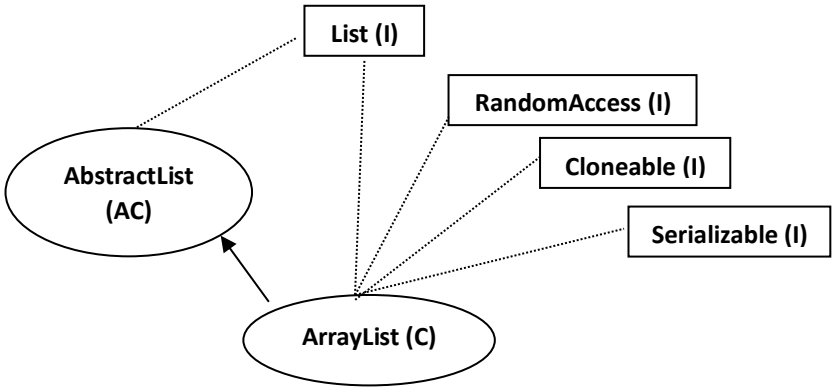
1. **void add (int index, E element) :** adds a particular object at a particular index.
2. **boolean addAll (int index, Collection<? Extends E> c) :** adds a group of objects started from this index onwards.
3. **E get (int index) :** returns the element at the specified position in this list.
4. **E set (int index, E element) :** replaces the element at the specified position in this list with the specified element.
5. **E remove (int index) :** removes a specified index object.
6. **int indexOf (Object o) :** returns index of first occurrence of specified object.
7. **int lastIndexOf (Object o) :** returns the last index of occurrence of specified object.
8. **ListIterator<E> listIterator ()** : returns a list iterator over the elements in this list (in proper sequence).
9. **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.
10. **List<E> subList (int fromIndex, int toIndex) :** returns a view of the portion of this list between the specified fromIndex (inclusive) & toIndex (exclusive).
11. **default void sort (Comparator<? super E> c) :** sorts this list according to the order induced by the specified Comparator.
12. **default void replaceAll (UnaryOperator<E> operator) :** replaces each element of this list with the result of applying the operator to that element.



List Interface implemented classes

2.1 ArrayList<E> class extends **AbstractList<E>** implements **List<E>, RandomAccess,** Cloneable, Serializable
2.2 LinkedList<E> class extends **AbstractSequentialList<E>** implements **List<E>, Deque<E>,** Cloneable, Serializable
2.3 Vector<E> class extends **AbstractList<E>** implements **List<E>, RandomAccess,** Cloneable, Serializable
2.4 Stack<E> class extends **Vector<E>**

2.1 ArrayList<E> class extends **AbstractList<E>** implements **List<E>, RandomAccess,** Cloneable, Serializable



a) The underlying data structure is Resizable array or Growable array.
b) Duplicates objects are allowed & insertion order is preserved.
c) Heterogenous objects are allowed (except TreeSet & TreeMap, everywhere heterogenous objects are allowed)
d) Null insertion is possible.
e) By default, ArrayList is non – synchronized but we can get synchronized version of **ArrayList** object by using **synchronizedList ()** method of Collections class.

public static List synchronizedList (List l)

e.g.,

```
ArrayList l = new ArrayList ();
List l1 = Collections.synchronizedList (l);
```

f) Constructors in ArrayList class

1. **ArrayList l = new ArrayList ()** : creates an empty ArrayList object with **default initial capacity 10**. Once it reaches its max capacity then a new ArrayList object will be created with formula

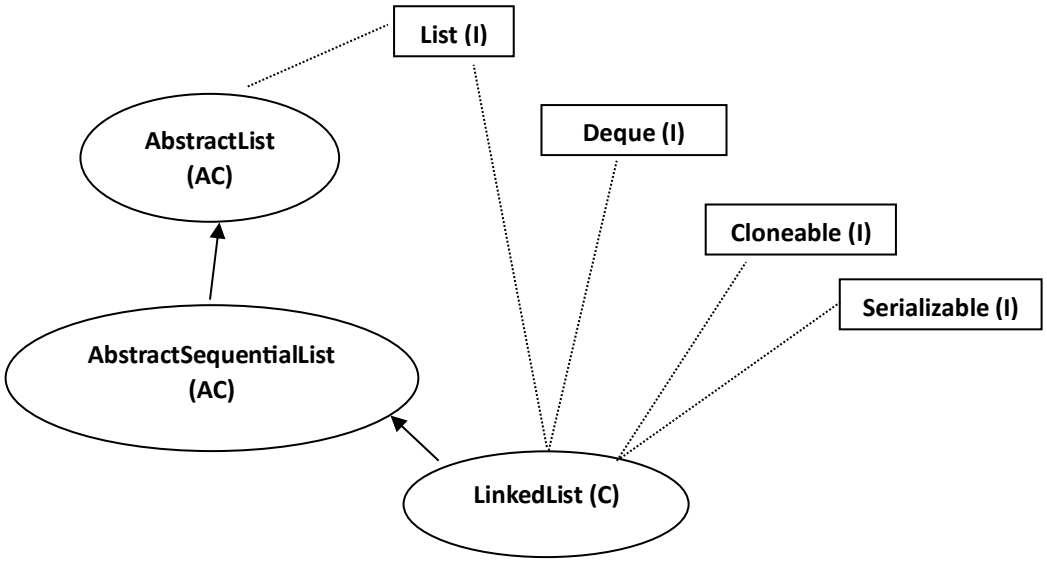
   New capacity = (Current Capacity * 3/2) + 1

2. **ArrayList l = new ArrayList (int initialCapacity)** : constructs an empty list with the specified initial capacity.
3. **ArrayList l = new ArrayList (Collection<? extends E> c)**
   - creates an equivalent ArrayList object for the given Collection.
   - This constructor meant for interconversion between Collection objects.

g) ArrayList is the best choice if our frequent operation is retrieval operation (because ArrayList implements RandomAccess interface which is a marker interface)

h) ArrayList is the worst choice if our frequent operation is insertion or deletion in middle (because of shift operation)

2.2 LinkedList<E> class extends **AbstractSequentialList<E>** implements **List<E>, Deque<E>,** Cloneable, Serializable



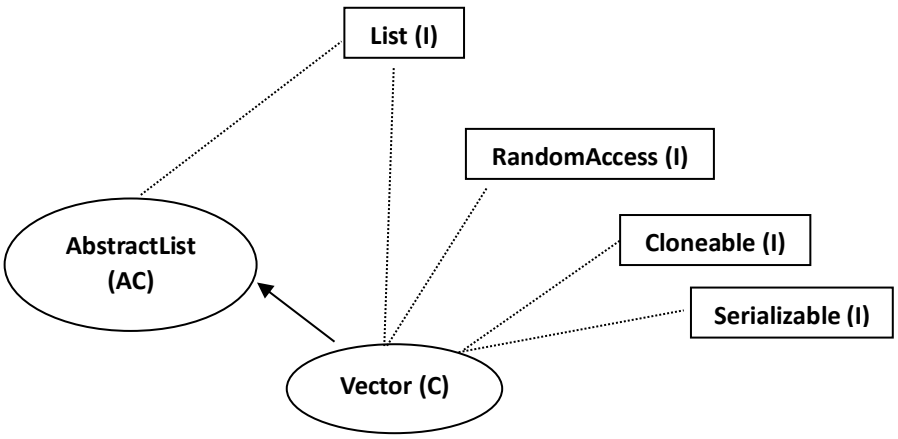
a) The underlying data structure is doubly – linked list & deque.
b) Duplicate objects are allowed & Insertion order is preserved.
c) Null insertion is possible.
d) By default, LinkedList is non – synchronized but we can get synchronized version same as ArrayList.

e) Constructors in LinkedList class
1. **LinkedList l = new LinkedList ()** : creates an empty linked list object.
2. **LinkedList l = new LinkedList (Collection c)** : creates an equivalent LinkedList object for the given collection.

f) **Methods in LinkedList<E>** class (Usually we can use LinkedList to develop Stack & Queues. To provide support for this requirement, LinkedList class defines following specific methods)
1. **void addFirst (E e) :** inserts the specified element at the beginning of this list.
2. **void addLast (E e) :** inserts the specified element to the end of this list.
3. **E getFirst () :** returns the first element in this list.
4. **E getLast () :** returns the last element in this list.
5. **E removeFirst () :** removes & returns the first element from this list.
6. **E removeLast () :** removes & returns the last element from this list.

g) LinkedList is the best choice if our frequent operation is insertion or deletion in the middle.
h) LinkedList is the worst choice if our frequent operation is retrieval operation.

2.3 Vector<E> class extends **AbstractList<E>** implements **List<E>, RandomAccess,** Cloneable, Serializable



a) The underlying data structure is Resizable array or Growable array.
b) Everything is same as ArrayList<E> except Vector object are synchronized.

c) Constructors in Vector class

1. **Vector v = new Vector ()**
   - Creates an empty Vector object with default initial capacity 10.
   - Once vector reaches its max capacity, then a new Vector object will be created with new capacity using formula

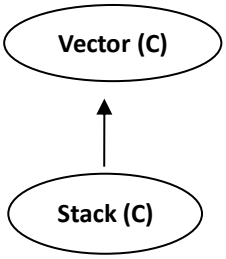
     New capacity = Current capacity * 2

2. **Vector v = new Vector (int initial_capacity)**
3. **Vector v = new Vector (int initial_capacity, int increment_capacity)** : creates an empty Vector object with specified initial_capacity & how much increment in size of vector is specified by increment_capacity.
4. **Vector v = new Vector (Collection c)**

d) Vector specific methods

1. **void addElement (E obj) :** adds the specified component to the end of this vector, increasing its size by one.
2. **boolean removeElement (Object obj) :** removes the first (lowest-indexed) occurrence of the argument from this vector.
3. **void removeElementAt (int index) :** deletes the component at the specified index.
4. **void removeAllElements () :** removes all components from this vector & sets its size to zero.
5. **E firstElement () :** returns the first component(the item at index 0) of this vector.
6. **E lastElement () :** returns the last component of the vector
7. **E elementAt (int index) :** returns the component at the specified index.

2.4 Stack<E> class extends **Vector<E>**



a) Stack class is specially designed class for LIFO (Last In First Out) order.

b) Constructor in Stack class: **Stack s = new Stack ();**

c) Methods in Stack class

1. **E push (E item) :** pushes an item onto the top of this stack.

2. **E pop () :** removes the object at the top of this stack & returns that object as the value of this function.

3. **E peek () :** looks at the object at the top of this stack without removing it from the stack.

4. **boolean empty () :** tests if this stack is empty.

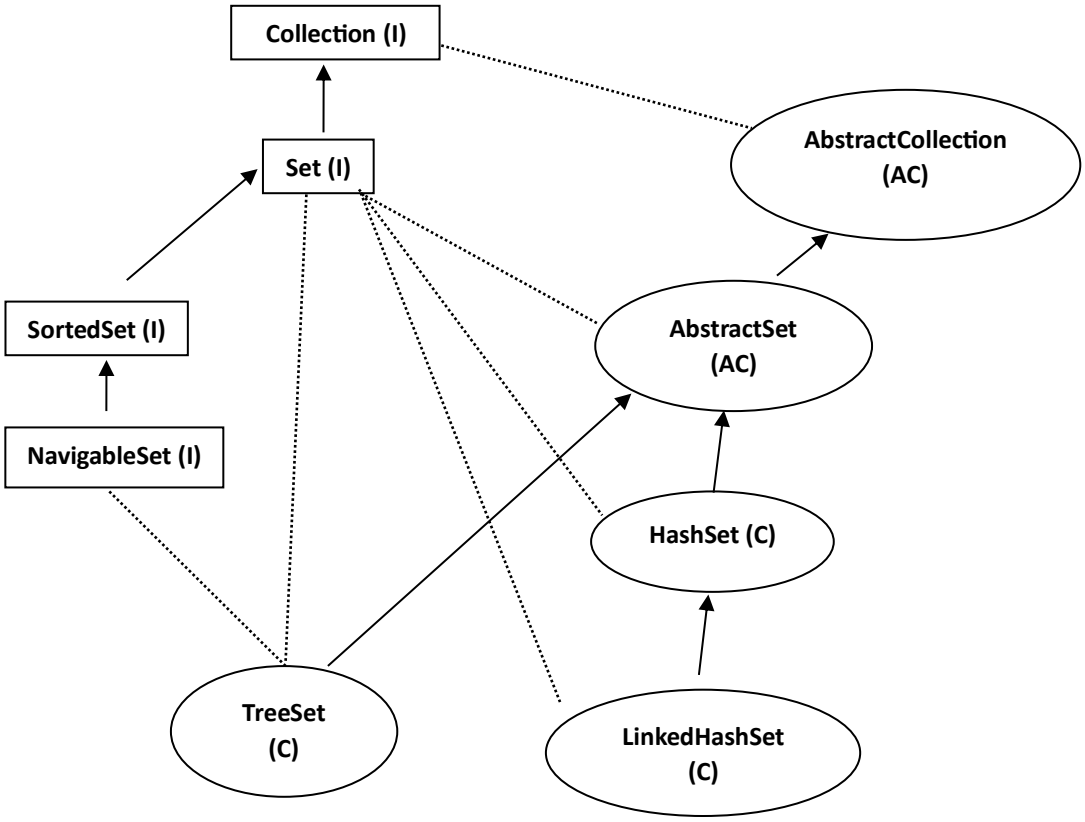
5. **int search (Object o) :** returns offset if the element is available otherwise returns -1.
   - Offset means position from the top.

3. Set<E> interface extends **Collection<E>** (E – the type of elements maintained by this set)

a) If we want to represent a group of individual objects as a single entity where duplicates are not allowed & insertion order not preserved then we should go for Set interface.

b) **Set interface doesn't contain any new method & we have to use only Collection interface methods.**

Set Interface implemented interfaces & classes



3.1 HashSet<E> interface extends **AbstractSet<E>** implements **Set<E>,** Cloneable, Serializable

3.2 LinkedHashSet<E> class extends **HashSet<E>** implements **Set<E>,** Cloneable, Serializable

3.3 SortedSet<E> interface extends **Set<E>**

3.4 NavigableSet<E> interface extends **SortedSet<E>**

3.5 TreeSet<E> class extends **AbstractSet<E>** implements **NavigableSet<E>,** Cloneable, Serializable

3.1 **HashSet<E> interface** extends **AbstractSet<E>** implements **Set<E>,** Cloneable, Serializable

a) The underlying data structure is hash table (actually a HashMap instance).
b) Duplicate objects are not allowed & insertion order is not preserved (it is based on hashcode of the objects)
c) Null insertion is possible only once
d) Heterogenous objects are allowed.
e) HashSet is the best choice if our frequent operation is search operation. This class offers constant time performance for the basic operations (add, remove, contains & size).

f) By default, HashSet is non – synchronized but we can get synchronized version of **HashSet** object by using **synchronizedSet ()** method of Collections class.

public static Set synchronizedSet (Set s)

e.g.,

Set s = Collections.synchronizedSet (new HashSet (…));

g) Constructors in HashSet<E>

1. HashSet h = new HashSet () : creates an empty HashSet object with default initial capacity 16 & default fill ratio 0.75
2. HashSet h = new HashSet (int initial_capacity) : creates an empty HashSet object with specified initial_capacity & default fill ratio 0.75
3. HashSet h = new HashSet (int initial_capacity, float fillRatio)
4. HashSet h = new HashSet (Collection c)
   - Create an equivalent HashSet for the given collection.
   - This constructor meant for interconversion b/w Collection objects.

FillRatio / Load Factor – After filling how much ratio, a new HashSet object will be created, this ratio is called FillRatio/ LoadFactor. For e.g., FillRatio 0.75 means after filling 75% ratio, a new HashSet object will be created automatically.

3.2 **LinkedHashSet<E> class** extends **HashSet<E>** implements **Set<E>,** Cloneable, Serializable

a) The underlying data structure is a combination of Hash table & linked list.
b) This implementation differs from HashSet in that it maintains a doubly – linked list running through all of its entries. This linked list defines the iteration ordering, which is the order in which elements were inserted into the set (insertion – order).
c) Insertion order preserved & duplicated not allowed. Other features same as HashSet
d) Constructors in LinkedHashSet<E> same as HashSet<E>

3.3 SortedSet<E> interface extends Set<E>

a) If we want to represent a group of individual objects according to some sorting order without duplicates then we should go for SortedSet.

b) The elements are ordered using their natural ordering, or by a Comparator typically provided at sorted set creation time.

c) All elements inserted into a sorted set must implement the Comparable interface (or be accepted by the specified comparator). Furthermore, all such elements must be mutually comparable.

d) Methods in SortedSet<E>

1. **E first () :** returns the first (lowest) element currently in this set.
2. **E last () :** returns the last (highest) element currently in this set.

3. **SortedSet<E> headSet (E toElement) :** returns a view of the portion of this set whose elements are strictly less than **toElement**.
4. **SortedSet<E> tailSet (E fromElement) :** returns a view of the portion of this set whose elements are greater than or equal to **fromElement**.

5. **SortedSet<E> subset (E fromElement, E toElement) :** returns a view of the portion of this set whose elements range from **fromElement** (inclusive) to **toElement** (exclusive).

6. **Comparator<? super E> comparator () :** returns the comparator used to order the elements in this set, or null if this set uses the natural ordering of its element.

3.4 NavigableSet<E> interface extends SortedSet<E>

a) It is the child interface of SortedSet & it defines several methods for Navigation purposes.

b) NavigableSet specific methods:

1. **E ceiling(E e) :** returns the least element in this set greater than or equal to the given element or null if there is no such element.
2. **E floor (E e):** returns the greatest element in this set less than or equal to the given element or null if there is no such element.

3. **E higher (E e):** returns the least element in this set strictly greater than the given element, or null if there is no such element.
4. **E lower (E e):** returns the greatest element in this set strictly less than the given element, or null if there is no such element.

5. **E pollFirst() :** retrieves/returns & removes the first (lowest) element, or returns null if this set is empty.
6. **E pollLast() :** retrieves & removes the last (highest) element, or returns null if this set is empty.

7. **NavigableSet<E> descendingSet():** returns a reverse order view of the elements contained in this set.

8. **Iterator<E> descendingIterator() :** returns an iterator over the elements in this set, in descending order.

3.5 TreeSet<E> class extends **AbstractSet<E>** implements **NavigableSet<E>,** Cloneable, Serializable

a) TreeSet uses a Tree for storage & is basically an implementation of a self – balancing binary search tree like a Red – Black tree.
b) Therefore operations like add, remove & search takes O(log N) time.
c) Since a set doesn't retain insertion order, the navigable set interface provides the implementation to navigate through the Set.
d) The elements are ordered using their natural ordering or by a Comparator provided at set creation time, depending on which constructor is used.
e) Operations like printing N elements in the sorted order takes O(N) time.
f) Duplicate objects not allowed.
g) TreeSet serves as an excellent choice for storing large amounts of sorted information which are supposed to be accessed quickly because of its faster access & retrieval time.
h) By default, TreeSet is non – synchronized but we can get synchronized version of **TreeSet** object by using **synchronizedSet ()** method of Collections class.

public static Set synchronizedSet (Set s)

e.g.,

Set s = Collections.synchronizedSet (new TreeSet());

i) **Constructors in TreeSet<E>**
1. TreeSet ts = new TreeSet();
2. TreeSet ts = new TreeSet (Collection<? extends E> c)
3. TreeSet ts = new TreeSet (Comparator<? super E> comparator) : Constructs a new, empty tree set, sorted according to the specified comparator.
4. TreeSet ts = new TreeSet (SortedSet<E> s): Constructs a new tree set containing the same elements & using the same ordering as the specified sorted set.

j) Methods in TreeSet<E> is same as in Collection interface & Navigable interface.

Note: The ordering maintained by a set (whether or not an explicit comparator is provided) must be *consistent with equals* if it is to correctly implement the Set interface. (See Comparable or Comparator for a precise definition of *consistent with equals*.) This is so because the Set interface is defined in terms of the equals operation, but a TreeSet instance performs all element comparisons using its **compareTo** (or compare) method, so two elements that are deemed equal by this method are, from the standpoint of the set, equal. The behaviour of a set *is* well-defined even if its ordering is inconsistent with equals; it just fails to obey the general contract of the Set interface.

4. Queue<E> interface extends **Collection<E>**

- **a)** If we want to represent a group of individual objects prior to processing then we should go for Queue.
- **b)** E.g., Before sending SMS message, we need to have all the mobile nos. in some data structure in which FIFO order is maintained.
- **c)** Usually Queue follows FIFO (First In First Out) order but based on our requirement we can implement our own Priority order also.
- **d)** From 1.5 version onwards, **LinkedList** class also implements **Deque** interface (child interface of **Queue**)
- **e)** **Methods in Queue interface**
    1. **boolean offer (E e) :** inserts the specified element into this queue if it's possible to do so immediately without violating capacity restrictions.
    2. **boolean add (E e):** same as offer () but raises RE: **IllegalStateException** if no space available.
    3. **E peek () :** returns the head element of the queue. If queue is empty, return null.
    4. **E element ():** same as peek () but raises RE: **NoSuchElementException** if queue is empty.
    5. **E poll ():** removes & returns the head element of the queue. If queue is empty, returns null.
    6. **E remove ():** same as poll () but this method raises RE: **NoSuchElementException** if queue is empty.

Queue Interface implemented interfaces & classes

4.1 PriorityQueue<E> class extends **AbstractQueue<E>** implements Serializable
4.2 Deque<E> interface extends **Queue<E>**

4.3 BlockingQueue<E> interface extends **Queue<E>**
4.4 LinkedBlockingQueue<E> class extends **AbstractQueue<E>** implements **BlockingQueue<E>,** Serializable
4.5 PriorityBlockingQueue<E> class extends **AbstractQueue<E>** implements **BlockingQueue<E>,** Serializable

4.1 PriorityQueue<E> class extends **AbstractQueue<E>** implements **Serializable**

a) If we want to represent a group of individual objects prior to processing according to some priority, then we should go for PriorityQueue.

b) The Priority can be either default natural sorting order or customized sorting order i.e., defined by Comparator.

c) Insertion order is not preserved & it is based on some priority.

d) Duplicated objects are not allowed.

e) If we're depending on default natural sorting, then compulsory the objects should be homogeneous & Comparable but if we're depending on Comparator sorting order then objects need not to be homogenous & Comparable.

f) This implementation provides O(log(n)) time for the enqueuing and dequeuing methods (offer, poll, remove() and add); linear time for the remove(Object) and contains(Object) methods; and constant time for the retrieval methods (peek, element, and size).

g) Constructors

1. PriorityQueue pq = new PriorityQueue(): creates a PriorityQueue with default initial capacity as 11 & all objects will be inserted in default natural sorting order.
2. PriorityQueue pq = new PriorityQueue(int initialCapacity)
3. PriorityQueue pq = new PriorityQueue(int initialCapacity, Comparator c)
4. PriorityQueue pq = new PriorityQueue(Comparator<? Extends E> c)
5. PriorityQueue pq = new PriorityQueue(Collection<? extends E> c)
6. PriorityQueue pq = new PriorityQueue(SortedSet<? Extends E> c)
7. PriorityQueue pq = new PriorityQueue(PriorityQueue<? Extends E> c)

h) Some platforms won't provide proper support for Thread priorities & PriorityQueues

4.2 Deque<E> interface extends Queue<E>

a) A linear collection that supports element insertion & removal at both ends.
b) The name deque is short for "double ended queue"
c) When a deque is used as a queue, FIFO (First – In – First – Out) behaviour result i.e., Elements are added at the end of the deque & removed from the beginning.
d) **Imp. Point:** When a deque is used as a stack, LIFO (Last – In – First – Out) behaviour result i.e., Elements are pushed & popped from the beginning of the deque.

Note: Deque should be used as stack instead of legacy Stack class.

e) Deque specific methods

1. **void addFirst (E e):** inserts the specified element at the front of this deque if it is possible to do so immediately without violating capacity restrictions, throwing an **IllegalStateException** if no space is currently available.

2. **void addLast (E e):** inserts the specified element at the end of this deque if it is possible to do so immediately without violating capacity restrictions, throwing an **IllegalStateException** if no space is currently available.

3. **boolean offerFirst (E e):** inserts the specified element at the front of this deque unless it would violate capacity restriction.
4. **boolean offerLast (E e):** inserts the specified element at the end of this deque unless it would violate capacity restrictions.

5. **E getFirst ():** retrieves, but doesn't remove, the first element of this deque.
6. **E getLast ():** retrieves, but doesn't remove, the last element of this deque.

7. **E peekFirst ():** retrieves, but doesn't remove, the first element of this deque, or returns null if this deque is empty.
8. **E peekLast ():** retrieves, but doesn't remove, the last element of this deque, or returns null if this deque is empty.

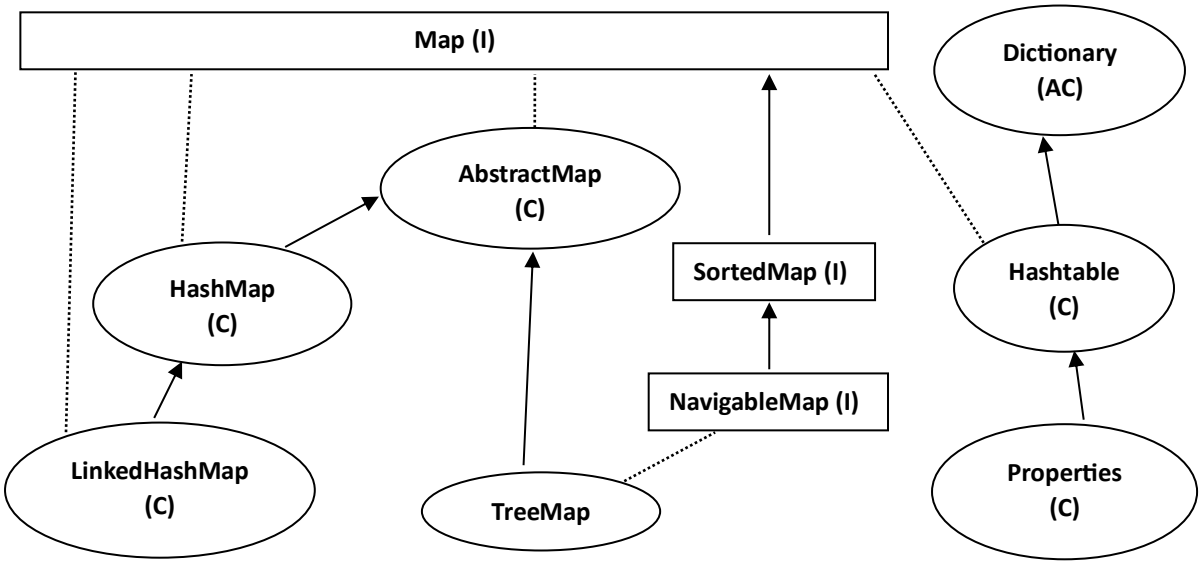
9. **E pollFirst ():** retrieves & removes the first element for this deque, or returns null if this deque is empty.
10. **E pollLast ():** retrieves & removes the last element for this deque, or returns null if this deque is empty.

11. **E removeFirst ():** returns & removes the first element of this deque.
12. **E removeLast ():** returns & removes the last element of this deque

4.3 BlockingQueue<E> interface extends Queue<E>

- A deque that additionally supports blocking operations that wait for the deque to become non – empty when retrieving an element, and wait for space to become available in the deque when storing an element.

5. Map<K, V> interface [K – the type of keys maintained by this map, V – the type of mapped values]



a) If we want to represent a group of objects as key value pairs then we should go for Map.
b) Both keys & values are object only.
c) Duplicate keys are not allowed but values can be duplicated.
d) Each key-value pair is called Entry. Hence Map is considered as a collection of Entry objects.
e) The Map interface provides three collection views, which allows a map's content to be viewed as a set of keys, collection of values, or set of key – value mappings.
f) The order of a map is defined as the order in which the iterators on the map's collection views return their elements. Some map implementations, like the **TreeMap** class maintains order, others like the **HashMap** class don't.
g) Nested class (Inner interface inside Map<K, V>)

- **public static interface Map.Entry<K, V>**

- Methods inside Map.Entry interface
  1. **K getKey () :** returns the key corresponding to this entry.
  2. **V getValue () :** returns the value corresponding to this entry.
  3. **V setValue () :** replaces the value corresponding to this entry with the specified value.

h) Methods in Map<K, V> interface

1. **V put (K key, V value)** : associates the specified value with the specified key in this map
2. **void putAll (Map<? Extends K, ? extends V) m)** : copies all of the mapping from the specified map to this map.
3. **V get (Object key)** : returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.
4. **V remove (Object key)** : removes the entry associated with specified key.
5. **boolean containsKey (Object key)** : return true if this map contains a mapping for the specified key.
6. **boolean containsValue (Object value)**: return true if this map maps one or more keys to the specified value.
7. **void clear () :** removes all of the entries from this map.
8. **boolean isEmpty ()** : returns true if this map contains no key – value mappings.
9. **int size ()** : returns the number of key – value mapping in this map.
10. **Set<Map.Entry<K, V>> entrySet ()** : returns a Set view of the mapping contained in this map.

11. **Set<K> keySet ()** : returns a Set view of the keys contained in this map.

12. **Collection<V> values () :** returns a Collection view of the values contained in this map.

13. **default V compute (K key, BiFunction<? super K, ? super V, ? extends V> remappingFunction) :** attempts to compute a mapping for the specified key & its current mapped value (or null if there is no current mapping).
14. **default V computeIfAbsent (K key, Function<? super K, ? extends V> mappingFunction) :** If the specified key is not already associated with a value (or is mapped to null), attempts to compute its value using the given mapping function & enters it into this map unless null.
15. **default V computeIfPresent (K key, BiFunction<? super K, ? super V, ? extends V> remappingFunction) :** If the value for the specified key is present & non-null, attempts to compute a new mapping given the key & its current mapped value.

16. **default void forEach (BiConsumer<? super K, ? super V> action) :** performs the given action for each entry in this map until all entries have been processed or the action throws an exception.

17. **default getOrDefault (Object key, V defaultValue) :** returns the value to which the specified key is mapped, or default value if the map contains no mapping for the key.
18. **default V putIfAbsent (K key, V value) :** If the specified key is not already associated with a value (or is mapped to null) associates it with the given value & returns null, else returns the current value.

19. **default V replace (K key, V value) :** replaces the entry for the specified key only if it's currently mapped to some value.

20. **default void replaceAll (BiFunction<? super K, ? super V, ? extends V> function) :** replaces each entry's value with the result of invoking the given function on that entry until all entries have been processed or the function throws an exception.

Map Interface implemented interfaces & classes

5.1 HashMap<K, V> class extends **AbstractMap<K, V>** implements **Map<K, V>,** Cloneable, Serializable

a) HashMap is called HashMap because it uses a technique called **Hashing**. **Hashing** is a technique of converting a large String into small String that represents the same string. A shorter value helps in indexing & faster searches.
b) HashSet also uses HashMap internally.
c) The underlying data structure is **Hashtable**.
d) The HashMap class is roughly like Hashtable, except that HashMap is unsynchronized & permits null key (only once) & null value (any no. of times).
e) Insertion order is not preserved & it's based on Hashcode of keys
f) Duplicate keys are not allowed but values can be duplicated
g) This implementation provides O (1) or constant time for get & put operations, assuming the hash function disperses the elements properly among the buckets.

h) **Iteration over HashMap requires time proportional** to the "**capacity**" of the HashMap instance (the no. of buckets) **plus** its **size** (the no. of key – value mappings). Thus, it's very important not to set the initial capacity too high (or the load factor too low) if iteration performance is important.

i) By default, HashMap is non – synchronized but we can get synchronized version of HashMap by using synchronizedMap () method of Collections classes.

public static Map synchronizedMap (Map m)

e.g.,

Map m = Collections. synchronizedMap (new HashMap (…));

j) Constructors in HashMap <K, V>

1. HashMap m = new HashMap () : constructs an empty HashMap with the default initial capacity (16) & the default load factor (0.75)
2. HashMap m = new HashMap (int initialCapacity) : constructs an empty HashMap with the specified initial capacity & the default load factor (0.75)
3. HashMap m = new HashMap (int initialCapacity, float LoadFactor);
4. HashMap m = new HashMap (Map<? extends K, ? extends V> m);

k) Methods same as Map interface methods

l) Performance of HashMap depends on 2 parameters

1. **Initial Capacity** – It is the capacity of HashMap at the time of its creation (i.e., no. of buckets a HashMap can hold when the HashMap is instantiated). Initially default capacity is 2^4 = 16 i.e., it can hold 16 key – values pairs.

2. **Load Factor** – It is the percent value of the capacity after which the capacity of HashMap is to be increased. (i.e., the percentage fill of buckets after which Rehashing takes place). In Java, default load factor is 0.75 i.e., the rehashing takes place after filling 75% of the capacity.

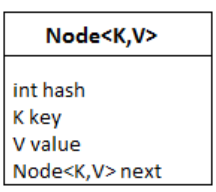
3. **Threshold** – Product of Load Factor & initial Capacity. In java, it's by default (0.75 * 16 = 12) i.e., rehashing takes place after filling 75% of the capacity.

4. **Rehashing** – It is the process of doubling the capacity of HashMap after it reaches its threshold. In java, HashMap continues to rehash in the following sequence: 2^4, 2^5, 2^6, …. so on.

- As a general rule, the default load factor (0.75) offers a good tradeoff between time & space costs. Higher values decrease the space overhead but increase the lookup cost. The expected no. of entries in the map & its load factor should be taken into account when setting its initial capacity, so as to minimize the no. of rehash operations.
- If initial capacity is greater than the maximum no. of entries divided by the load factor, no rehash operations will ever occur.
- If many mappings are to be stored in a HashMap instance, creating it with a sufficiently large capacity will allow the mappings to be stored more efficiently than letting it perform automatic rehashing as needed to grow the table.
- Note that using many keys with same hashcode () is a sure way to slow down performance of any Hashtable. To ameliorate impact, when keys are Comparable, this class may use comparison order among keys to help break ties.

Internal Structure of HashMap

- **Internally HashMap contains an array of Node & a node is represented as a class that contains 4 fields:**
  1. int hash
  2. K key
  3. V value
  4. Node next: reference to next node.



- It can be seen that the node is containing a reference to its own object. So, it's a linked list.

- Major update: From Java 8 onwards, Java has started using Self Balancing BST instead of a linked list for chaining. The advantage of self – balancing BST is, we can get the worst case (when every key maps to same slot) search time as O(log N).

- Before understanding the internal working of HashMap, we must be aware of hashCode () & equals () method.
    1. **equals ()**: It checks the equality of 2 objects. It compares the key, whether they are equal or not. If we override the equals () method, then it's mandatory to override the hashCode() method.
    2. **hashCode ():** It returns the memory reference of the object in integer form. The value received from this method is used as bucket number. The bucket no. is the address of the element inside the map. HashCode of null key is 0.
    3. **Buckets:** Array of the node is called bucket. Each node has a data structure like a LinkedList (self-balancing BST from java 8). More than one node can share the same bucket.

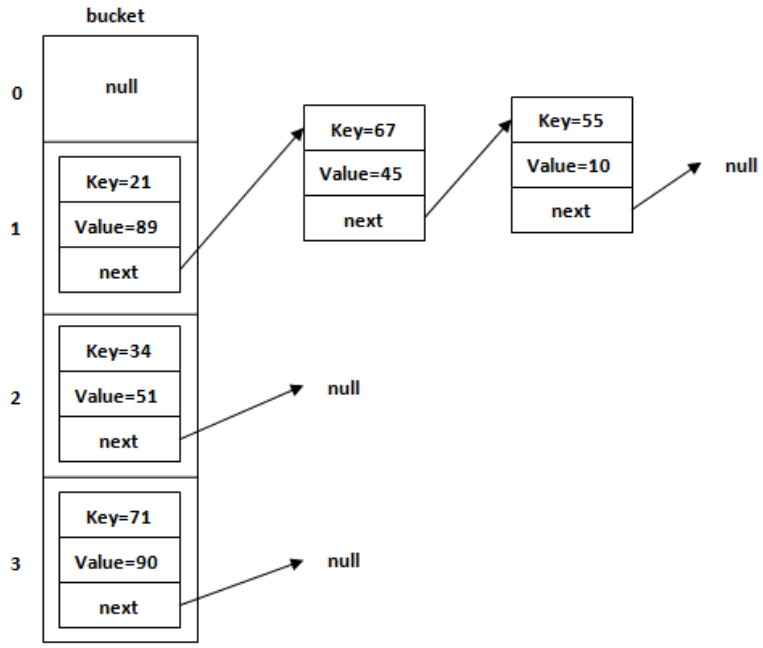


Figure: Allocation of nodes in Bucket

Let's take example & understand HashMap internal functioning

1. Insert key, value pair in HashMap

- We use put () method to insert the key & the value pair in the HashMap. The default size is 16 (0 to 15)
- e.g.,

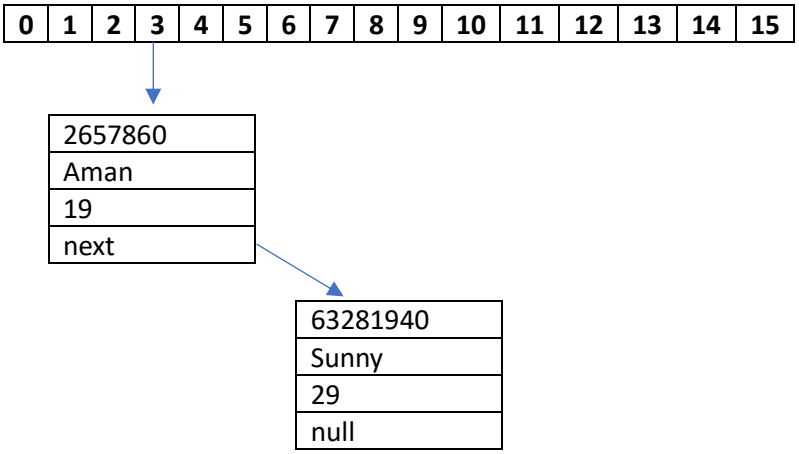
```
HashMap<String, Integer> map = new HashMap<> ();
map.put ("Aman", 19);   // Suppose hashcode (Aman) = 2657860
map.put ("Sunny", 29); // Suppose hashcode (Sunny) = 63281940
map.put ("Ritesh", 39);
```

- Calculating the index using the formula: (n is the size of array)

  Index = hashcode(key) & (n – 1)

  e.g.,
  Index (Aman) = hashcode (Aman) * (16 – 1) = 2657860 * 15 = 4
  Index (Sunny) = hashcode (Sunny) * (16 – 1) = 63281940 * 15 = 4

- This is the case of **Hash Collision** i.e.; the calculated index value is same for 2 or more keys.
  In this case, equals () method checks that keys are equal or not
    - If keys are same, replace the value with the current value.
    - Otherwise, connect this node object to the existing node object through the LinkedList. Hence both the keys will be stored at index 4.



2. get () method in HashMap

- When get (K key) method is called, it calculates the hashcode of the key then using Index formula, index for the key will be generated.
- For e.g., suppose index is 4 for key Aman, then get () method search for the index value 4.
  - If both keys are equal, then it returns the value
  - Else check for the next element in the node if it exists.

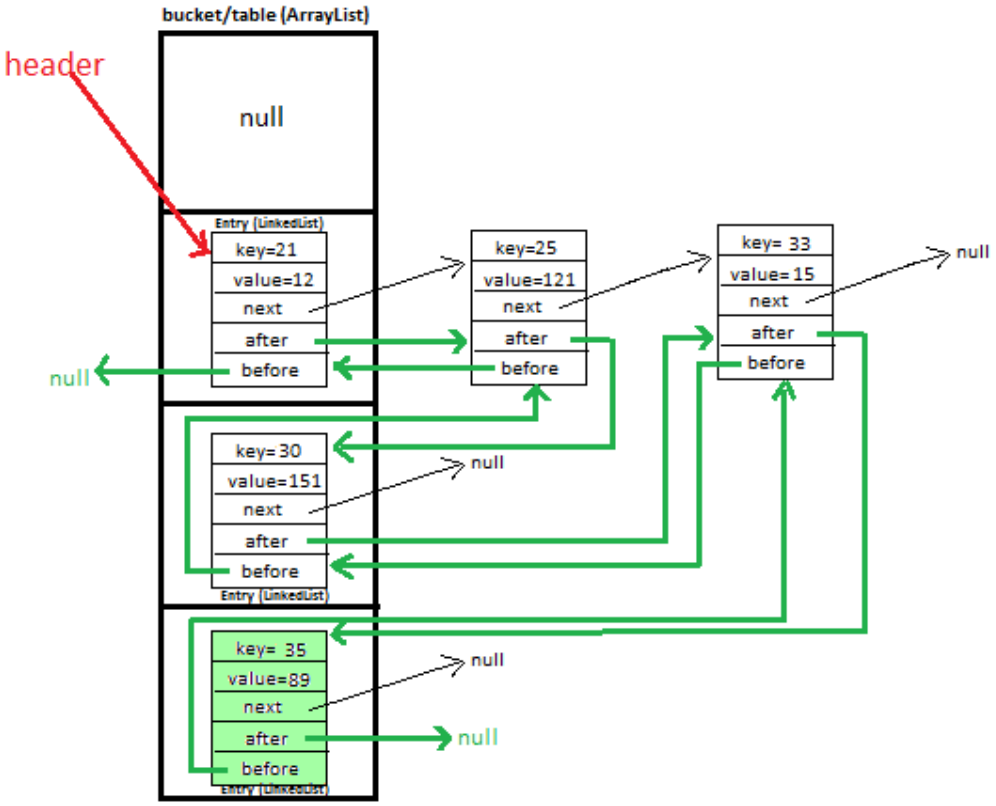
5.2 LinkedHashMap<K, V> class extends **HashMap<K, V>** implements **Map<K, V>**

- The LinkedHashMap class is just like HashMap with an additional feature of maintaining an order of elements inserted into it.
- Hashtable & doubly – linked list implementation of the Map interface, with predictable iteration order. This implementation differs from the HashMap in that it maintains a doubly – linked list running through all of its entries.
- This linked list defines the iteration ordering, which is normally the order in which keys were inserted into the map (insertion order)
- This implementation spares its clients from the unspecified, generally chaotic ordering provided by HashMap (and Hashtable), without incurring the increased cost associated with TreeMap. It can be used to produce a copy of a map that has same order as the original, regardless of the original map's implementation.

- Each node of the LinkedHashMap is represented as:

Hash	Key	Value	Next	Before/head	After/tail

- **Hash:** All the input keys are converted into a hash which is a shorter form of the key so that the search & insertion are faster.
- **Key:** Since this class extends HashMap, the data is stored in the form of a key – value pair. Therefore, this parameter is the key to the data.
- **Value:** For every key, there is value associated with it.
- **Next:** Since the LinkedHashMap stores the insertion order, this contains the address to the next node of the LinkedHashMap.
- **Previous:** This parameter contains the address of the previous node of the LinkedHashMap.



5.3 WeakHashMap<K, V> class extends **AbstractMap<K, V>** implements **Map<K, V>**

- It's exactly same as HashMap except the following difference:
  - In the case of HashMap, even though Object doesn't have any reference, it's not eligible for garbage collection (GC) if the object is associated with HashMap i.e., HashMap dominates Garbage Collector.
  - In the case of WeakHashMap, if object doesn't contain any references, it's eligible for Garbage collection even though object associated with WHM i.e., Garbage Collector dominates WHM.

5.4 IdentityHashMap<K, V> class extends **AbstractMap<K, V>** implements **Map<K, V>,** Serializable, Cloneable

- In general, == operator meant for reference comparison (address comparison) whereas .equals() method meant for content comparison.

```
e.g., Integer i1 = new Integer (10);
Integer i2 = new Integer (10);
System.out.println (i1 == i2);          // false
System.out.println (i1. equals(i2));    // true
```

- It's exactly same as HashMap (including methods & constructors) except the following difference:
  - In case of Normal HashMap, JVM will use .equals() method to identify duplicate keys, which is meant for content comparison.
  - But in case of IdentityHashMap, JVM will use "==" operator to identify duplicate keys which is meant for reference comparison (address comparison)

```
e.g.,
IdentityHashMap m = new IdentityHashMap ();
Integer i1 = new Integer (10);
Integer i2 = new Integer (10);
m.put (i1, "Shivam");
m.put (i2, "Raj");
System.out.println (m);  // {10 = Shivam, 10 = Raj}
```

5.5 SortedMap<K, V> interface extends **Map<K, V>**

- If we want to represent a group of Object as a group of key – value pairs according to some sorting order of keys (using Comparator), then we should go for SortedMap.

- **SortedMap specific methods**
  1. **K firstKey ():** returns the first (lowest) key currently in this map.
  2. **K lastKey ():** returns the last (highest) key currently in this map.
  3. **SortedMap<K, V> headMap (K toKey):** returns a view of the portion of this map whose keys are strictly less than **toKey**.
  4. **SortedMap<K, V> tailMap (K fromKey):** returns a view of the portion of this map where keys are greater than or equal to **fromKey**.
  5. **SortedMap<K, V> submap (K fromKey, K toKey):** returns a view of the portion of this map whose keys range from fromKey (inclusive) to toKey (exclusive)
  6. **Comparator<? super K> comparator ():** returns the comparator used to order the keys in this map, or null if this map uses the natural ordering of its keys.

5.6 NavigableMap<K, V> interface extends **SortedMap<K, V>**

- A SortedMap extended with navigation methods returning the closet matches for a given search targets.

- **NavigableMap specific methods**
  1. **K ceilingKey (K key):** returns the least key greater than or equal to the given key, or null if there is no such key.
  2. **K higherKey (K key):** returns the least key strictly greater than the given key, or null if there is no such key.
  3. **K floorKey (K key):** returns the greatest key less than or equal to the given key, or null if there is no such key.
  4. **K lowerKey (K key):** returns the greatest key strictly less than the given key, or null if there is no such key.
  5. **Map.Entry<K, V> pollFirstEntry ():** removes & returns a key – value mapping associated with the least key in this map, or null if the map is empty.
  6. **Map.Entry<K, V> pollLastEntry ():** removes & returns a key – value mapping associated with the greatest key in this map, or null if the map is empty.
  7. **NavigableMap<K, V> descendingMap ():** returns a reverse order view of the mappings contained in this map.

5.7 TreeMap<K, V> class extends **AbstractMap<K, V>** implements **NavigableMap<K, V>,** Cloneable, Serializable

- A TreeMap is implemented using a Red – black tree, which is a type of self – balancing BST.
- This provides efficient performance for common operations such as adding, removing, & retrieving elements, with an average time complexity of O(log N)
- The TreeMap in Java is a concrete implementation of the **java.util.SortedMap** interface. It provides an ordered collection of key – value pairs, where the keys are ordered based on their natural order or a custom Comparator passed to the constructor.
- By default, TreeMap is non – synchronized but we can get synchronized version of TreeMap by using **synchronizedSortedMap** () method of Collections classes.
  e.g.,

  SortedMap m = Collections. synchronizedSortedMap (new TreeMap (…));

- **Constructors in TreeMap**
    1. TreeMap t = new TreeMap ();
    2. TreeMap t = new TreeMap (Comparator<? super K> comparator);
    3. TreeMap t = new TreeMap (Map<? extends K, ? extends V> m);
    4. TreeMap t = new TreeMap (SortedMap< K, ? extends V> m);

5.8 Dictionary<K, V> abstract class extends Object

- The Dictionary class is an abstract class & can't be instantiated directly. Instead, it provides the basic operations for accessing the key – value pairs stored in the collection, which are implemented by its concrete subclass java.util.Hashtables

5.9 Hashtable<K, V> class extends Dictionary<K, V> implements Map<K, V>, Cloneable, Serializable

- This class implements a hash table, which maps keys to values.
- Insertion order not preserved
- Duplicate keys are not allowed.
- Every method present in Hashtable is synchronized & hence Hashtable object is thread – safe. If a thread – safe implementation is not needed, it is recommended to use HashMap in place of Hashtable.
- Hashtable is the best choice, if our frequent operation is Search operation.
- Constructors in TreeMap
    1. Hashtable h = new Hashtable ();
    2. Hashtable h = new Hashtable (int initialCapacity);
    3. Hashtable h = new Hashtable (int initialCapacity, float loadFactory);
    4. Hashtable h = new Hashtable (Map<? extends K, ? extends V> t);

5.10 Properties class extends Hashtable<Object, Object>

- In our program, if anything which changes frequently (like user_name, password, mail_id etc.) are not recommended to hard code in Java program because if there is any change, to reflect that change, recompilation, rebuild & redeploy application are required even sometime server restart also required which creates a big business impact to the client.
- We can overcome this problem by using Properties file. We can assign our frequently changing properties in Properties file & from there we can read into Java Program.
- The main advantage of this approach is if there is a change in Properties file, to reflect that change, just redeployment is enough, which won't create any business impact to the client.
- **Constructor in Properties class**
    - Properties p = new Properties ();

- **Properties class specific methods**
    1. **String getProperty (String key):** searches for the property with the specified key in this property list.
    2. **Object setProperty (String key, String value):** to set a new Property. Calls the Hashtable method put.
    3. **Enumeration propertyNames ():** returns all property Names present in Properties object.
    4. **void load (InputStream is):** to load properties from Properties file into java Properties object.
    5. **void store (OutputStream os, String comment):** to store properties from Java Properties object into Properties file.