**COLLECTION**

C:\Users\nn031615\AppData\Local\Microsoft\Windows\INetCache\Content.Word\collection-hierarchy.png

1. **Iterable – Interface**
   1. **Methods:**
      1. void forEach() – Java 1.8
      2. [Iterator](https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html)<[T](https://docs.oracle.com/javase/8/docs/api/java/lang/Iterable.html)> iterator()
      3. [Spliterator](https://docs.oracle.com/javase/8/docs/api/java/util/Spliterator.html)<[T](https://docs.oracle.com/javase/8/docs/api/java/lang/Iterable.html)> spliterator() – 1.8
2. **Collection – Interface (1.8)**
   1. **Methods**
      1. boolean add(E e)
      2. boolean addAll(Collection<? extends E> c)
      3. void clear()
      4. boolean contains (Object o)
      5. boolean containsAll(Collection<?> c)
      6. boolean equals(Object o)
      7. int hashCode()
      8. boolean isEmpty()
      9. Iterator<E> iterator()
      10. boolean remove(Object o)
      11. boolean removeAll(Collection<?> c)
      12. boolean removeIf(Predicate<? super E> filter)
      13. boolean retainAll(Collection<?> c)
      14. int size()
      15. Stream <E> stream()
      16. Object[] toArray()
      17. <T> T[] toArray(T[]
   2. **Desctiption** : Root interface of collection hierarchy
3. **Set – Interface**
   1. **Methods**
      1. boolean add(E e)
      2. boolean addAll(Collection<? extends E> c)
      3. void clear()
      4. boolean contains (Object o)
      5. boolean containsAll(Collection<?> c)
      6. static <E> Set<E> copyOf​(Collection<? extends E> coll)
      7. boolean equals(Object o)
      8. int hashCode()
      9. boolean isEmpty()
      10. Iterator<E> iterator()
      11. static <E> Set<E> of​(E e ) – (up to 10 elements) : Returns an unmodifiable set containing no. of arguments passed.
      12. Spliterator <E> spliterator()
      13. boolean remove(Object o)
      14. boolean removeAll(Collection<?> c)
      15. boolean retainAll(Collection<?> c)
      16. int size()
      17. Object[] toArray()
      18. <T> T[] toArray(T[]
   2. **Description** : No Duplicate elements and can have 1 null
4. **HashSet – Class**
   1. Methods:
      1. boolean add(E e)
      2. boolean addAll(Collection<? extends E> c)
      3. void clear()
      4. Object clone();
      5. boolean contains (Object o)
      6. boolean containsAll(Collection<?> c)
      7. boolean equals(Object o)
      8. int hashCode()
      9. boolean isEmpty()
      10. Iterator<E> iterator()
      11. Spliterator <E> spliterator()
      12. boolean remove(Object o)
      13. boolean removeAll(Collection<?> c)
      14. boolean retainAll(Collection<?> c)
      15. int size()
      16. Object[] toArray()
      17. <T> T [] toArray(T[])
   2. **Description:** Backed by hash table. Insertion order is not maintained. Can have 1 null element. It is not synchronized. The iterator returned is **fail-fast.** If Set is modified after iterator is created, ConcurrentModificationException is thrown.
   3. **Working:** It creates a map with key as data (e) in inserted and value as Object PRESENT

Ex : private static final Object PRESENT = new Object();

map.put(e, PRESENT)

* 1. **Constructors:** 
     1. [HashSet](https://docs.oracle.com/javase/9/docs/api/java/util/HashSet.html#HashSet--)​() - Constructs a new, empty set; the backing HashMap instance has default initial capacity (16) and load factor (0.75).
     2. [HashSet](https://docs.oracle.com/javase/9/docs/api/java/util/HashSet.html#HashSet-int-)​(int initialCapacity): Creates set with size “initialCapacity” and load factor 0.75
     3. [HashSet](https://docs.oracle.com/javase/9/docs/api/java/util/HashSet.html#HashSet-int-float-)​(int capacity, float loadFactor)
     4. [HashSet](https://docs.oracle.com/javase/9/docs/api/java/util/HashSet.html#HashSet-java.util.Collection-)​([Collection](https://docs.oracle.com/javase/9/docs/api/java/util/Collection.html)<? extends [E](https://docs.oracle.com/javase/9/docs/api/java/util/HashSet.html)> c) : Constructs a new set containing the elements in the specified collection.

1. **LinkedHashSet – Class**
   1. **Methods:** *Same as HASHSET*
   2. **Desc**: Hash table and linked list implementation of the Set. Insertion order is maintained. Its not synchronized*.* (Rest same as HashSet and **Constructors**)
2. **SortedSet – Interface**
   1. **Methods:** 
      1. 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 elements.
      2. E first()
      3. SortedSet<E> headSet​(E toElement) : Returns a view of the portion of this set whose elements are strictly less than toElement
      4. E last()
      5. default Spliterator<E> spliterator()
      6. SortedSet<E> subSet​(E fromElement, E toElement)
      7. SortedSet<E> tailSet​(E fromElement) : Returns a view of the portion of this set whose elements are greater than or equal to fromElement.
   2. **Desc:** Elements are ordered in their natural order or by a comparator.
   3. **Constructor:** 1) A void (no arguments) constructor, to sort according to the natural ordering. 2) A constructor with a single argument of type Comparator, which creates an empty sorted set sorted according to the specified comparator. 3) A constructor with a single argument of type Collection, which creates a new sorted set with the same elements as its argument, sorted according to the natural ordering of the elements. 4) A constructor with a single argument of type SortedSet, which creates a new sorted set with the same elements and the same ordering as the input sorted set.
3. **NavigableSet - Interface**
   1. **Method:**
      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. Iterator<E> descendingIterator()
      3. NavigableSet<E> descendingSet()
      4. E floor​(E e)
      5. SortedSet<E> headSet​(E toElement) : less that toElement (i.e exclusive)
      6. NavigableSet<E> headSet​(E toElement, boolean inclusive)
      7. 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.
      8. Iterator<E> iterator()
      9. E lower​(E e)
      10. E pollFirst() : Retrieves and removes the first (lowest) element, or returns null if this set is empty.
      11. E pollLast() : {for highest}
      12. NavigableSet<E> subSet​(E fromElement, boolean fromInclusive,E toElement, boolean toInclusive)
      13. SortedSet<E> subSet​(E fromElement, E toElement)
      14. SortedSet<E> tailSet​(E fromElement) : from fromElement, inclusive, to toElement, exclusive.
      15. NavigableSet<E> tailSet​(E fromElement, boolean inclusive)
   2. **Desc:** Sameas SortedSet but has navigation methods like descendingSet(), lower(E), ceiling(E) etc
4. **Treeset - Class**
   1. **Methods:** 
      1. boolean add​(E e)
      2. boolean addAll​(Collection<? extends E> c)
      3. E ceiling​(E e)
      4. void clear()
      5. Object clone()
      6. boolean contains​(Object o)
      7. Iterator<E> descendingIterator()
      8. NavigableSet<E> descendingSet()
      9. E first()
      10. E floor​(E e)
      11. SortedSet<E> headSet​(E toElement)
      12. NavigableSet<E> headSet​(E toElement, boolean inclusive)
      13. 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.
      14. boolean isEmpty()
      15. Iterator<E> iterator()
      16. E last() : Returns the last (highest) element currently in this set.
      17. 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.
      18. E pollFirst()
      19. E pollLast()
      20. boolean remove​(Object o)
      21. int size()
      22. Spliterator<E> spliterator()
      23. NavigableSet<E> subSet​(E fromElement, boolean fromInclusive,E toElement, boolean toInclusive)
      24. SortedSet<E> subSet​(E fromElement, E toElement)
      25. SortedSet<E> tailSet​(E fromElement)
      26. NavigableSet<E> tailSet​(E fromElement, boolean inclusive)
   2. **Desc :** A NavigableSet implementation based on a TreeMap. The elements are ordered using their natural ordering, or by a Comparator provided at set creation time, depending on which constructor is used.

Does not allow to insert Heterogeneous objects.

* 1. **Constructor** :
     1. **TreeSet();**
     2. **TreeSet(Comparator comp);**
     3. **TreeSet(Collection col);**
     4. **TreeSet(SortedSet s);**

1. **Queue - Interface**
   1. **Methods**
      1. boolean add​(E e) : Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions, returning true upon success and throwing an IllegalStateException if no space is currently available.
      2. E element() : Retrieves, but does not remove, the head of this queue.
      3. boolean offer​(E e) : Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions.
      4. E peek() : Retrieves, but does not remove, the head of this queue, or returns null if this queue is empty.
      5. E poll() : Retrieves and removes the head of this queue, or returns null if this queue is empty.
      6. E remove() : Retrieves and removes the head of this queue.
   2. **Desc:** Queues typically, but do not necessarily, order elements in a FIFO (first-in-first-out) manner. Among the exceptions are priority queues, which order elements according to a supplied comparator, or the elements' natural ordering, and LIFO queues (or stacks) which order the elements LIFO (last-in-first-out). Whatever the ordering used, the head of the queue is that element which would be removed by a call to remove() or poll(). In a FIFO queue, all new elements are inserted at the tail of the queue. Other kinds of queues may use different placement rules. Every Queue implementation must specify its ordering properties
2. **Deque ­- Interface**
   1. **Methods:**
      1. boolean add​(E e)
      2. boolean addAll​(Collection<? extends E> c) : Adds all element at end of queue like calling addLast(E e) for all element
      3. 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.
      4. void addLast​(E e)
      5. boolean contains​(Object o)
      6. Iterator<E> descendingIterator()
      7. E element() : Retrieves, but does not remove, the head of the queue represented by this deque (in other words, the first element of this deque).
      8. E getFirst() : Retrieves, but does not remove, the first element of this deque.
      9. E getLast()
      10. Iterator<E> iterator()
      11. boolean offer​(E e)
      12. boolean offerFirst​(E e) : Inserts the specified element at the front of this deque unless it would violate capacity restrictions.
      13. boolean offerLast​(E e){… end … }
      14. E peek() : Retrieves, but does not remove, the head of the queue represented by this deque (in other words, the first element of this deque), or returns null if this deque is empty.
      15. E peekFirst() : Retrieves, but does not remove, the first element of this deque, or returns null if this deque is empty.
      16. E peekLast() : Retrieves, but does not remove, the last element of this deque, or returns null if this deque is empty.
      17. E poll() : Retrieves and removes the head of the queue represented by this deque (in other words, the first element of this deque), or returns null if this deque is empty.
      18. E pollFirst()
      19. E pollLast()
      20. E pop() : Pops an element from the stack represented by this deque.
      21. void push​(E e) : Pushes an element onto the stack represented by this deque (in other words, at the head of this deque) throws an IllegalStateException if no space is currently available.
      22. E remove()
      23. boolean remove​(Object o) : Retrieves and removes the head of the queue represented by this deque (in other words, the first element of this deque).
      24. E removeFirst() : Retrieves and removes the first element of this deque.
      25. boolean removeFirstOccurrence​(Object o)
      26. E removeLast()
      27. boolean removeLastOccurrence​(Object o)
      28. int size()
   2. **Desc :**A linear collection that supports element insertion and removal at both ends. The name deque is short for "double ended queue" and is usually pronounced "deck". Most Deque implementations place no fixed limits on the number of elements they may contain, but this interface supports capacity-restricted deques as well as those with no fixed size limit.
3. **ArrayDeque – Class**
   1. **Methods:**
      1. boolean add​(E e)
      2. boolean addAll​(Collection<? extends E> c)
      3. void addFirst​(E e)
      4. void addLast​(E e)
      5. void clear()
      6. ArrayDeque<E> clone()
      7. boolean contains​(Object o)
      8. E element()
      9. void forEach​(Consumer<? super E> action)
      10. E getFirst()
      11. E getLast()
      12. boolean isEmpty()
      13. Iterator<E> iterator()
      14. boolean offer​(E e) : Inserts the specified element at the end of this deque.
      15. boolean offerFirst​(E e)
      16. boolean offerLast​(E e)
      17. E peek()
      18. E poll()
      19. E pop() : Pops an element from the stack represented by this deque.
      20. void push​(E e)
      21. E remove() : Retrieves and removes the head of the queue represented by this deque.
      22. boolean remove​(Object o)
      23. boolean removeAll​(Collection<?> c)
      24. E removeFirst() : Retrieves and removes the first element of this deque.
      25. boolean removeFirstOccurrence​(Object o)
      27. E removeLast()
      28. boolean removeLastOccurrence​(Object o)
      29. boolean retainAll​(Collection<?> c)
      30. int size()
      31. Spliterator<E> spliterator()
      32. Object[] toArray()
      33. <T> T[] toArray​(T[] a)
   2. **Desc :** Resizable-array implementation of the Deque interface. Array deques have no capacity restrictions; they grow as necessary to support usage. They are not thread-safe; in the absence of external synchronization, they do not support concurrent access by multiple threads. Null elements are prohibited. This class is likely to be faster than Stack when used as a stack, and faster than LinkedList when used as a queue.
   3. **Constructor:**
      1. ArrayDeque() : Constructs an empty array deque with an initial capacity sufficient to hold 16 elements.
      2. ArrayDeque​(int numElements) : Constructs an empty array deque with an initial capacity sufficient to hold the specified number of elements.
      3. ArrayDeque​(Collection<? extends E> c) : Constructs a deque containing the elements of the specified collection, in the order they are returned by the collection's iterator.
4. **PriorityQueue – Class**
   1. **Methods :** 
      1. boolean add​(E e)
      2. void clear()
      3. Comparator<? super E> comparator() : Returns the comparator used to order the elements in this queue, or null if this queue is sorted according to the natural ordering of its elements.
      4. boolean contains​(Object o)
      5. Iterator<E> iterator()
      6. boolean offer​(E e)
      7. boolean remove​(Object o)
      8. Spliterator<E> spliterator()
      9. Object[] toArray()
      10. <T> T[] toArray​(T[] a)
   2. **Desc :** PriorityQueue doesn’t allow null We can’t create PriorityQueue of Objects that are non-comparable The elements of the priority queue are ordered according to their natural ordering, or by a Comparator provided at queue construction time, depending on which constructor is used. The head of this queue is the least element with respect to the specified ordering. If multiple elements are tied for least value, the head is one of those elements — ties are broken arbitrarily.
   3. **Constructor:** 
      1. PriorityQueue() : Creates a PriorityQueue with the default initial capacity (11) that orders its elements according to their natural ordering.
      2. PriorityQueue​(int initialCapacity) : Creates a PriorityQueue with the specified initial capacity that orders its elements according to their natural ordering.
      3. PriorityQueue​(int initialCapacity, Comparator<? super E> comparator) : Creates a PriorityQueue with the specified initial capacity that orders its elements according to the specified comparator.
      4. PriorityQueue​(Collection<? extends E> c) : Creates a PriorityQueue containing the elements in the specified collection.
      5. PriorityQueue​(Comparator<? super E> comparator) : Creates a PriorityQueue with the default initial capacity and whose elements are ordered according to the specified comparator.
      6. PriorityQueue​(PriorityQueue<? extends E> c) : Creates a PriorityQueue containing the elements in the specified priority queue.
      7. PriorityQueue​(SortedSet<? extends E> c) : Creates a PriorityQueue containing the elements in the specified sorted set.

1. **List – Interface**
   1. **Methods :** 
      1. void add​(int index, E element)
      2. boolean add​(E e)
      3. boolean addAll​(int index, Collection<? extends E> c)
      4. boolean addAll​(Collection<? extends E> c)
      5. void clear()
      6. boolean contains​(Object o)
      7. boolean containsAll​(Collection<?> c)
      8. static <E> List<E> copyOf​(Collection<? extends E> coll) : Returns an unmodifiable List containing the elements of the given Collection, in its iteration order.
      9. boolean equals​(Object o)
      10. E get​(int index)
      11. int hashCode()
      12. int indexOf​(Object o)
      13. boolean isEmpty()
      14. Iterator<E> iterator()
      15. int lastIndexOf​(Object o)
      16. ListIterator<E> listIterator()
      17. 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.
      18. static <E> List<E> of() : Returns an unmodifiable list containing zero elements.
      19. static <E> List<E> of​(E e1) : can have upto 10 arguments
      20. static <E> List<E> of​(E... elements) : Returns an unmodifiable list containing an arbitrary number of elements.
      21. E remove​(int index)
      22. boolean remove​(Object o)
      23. boolean removeAll​(Collection<?> c)
      24. default void replaceAll​(UnaryOperator<E> operator) : Replaces each element of this list with the result of applying the operator to that element.
      25. boolean retainAll​(Collection<?> c)
      26. E set​(int index, E element)
      27. int size()
      28. default void sort​(Comparator<? super E> c)
      29. default Spliterator<E> spliterator()
      30. List<E> subList​(int fromIndex, int toIndex)
      31. Object[] toArray()
      32. <T> T[] toArray​(T[] a)
   2. **Desc :** List is an ordered collection of objects in which duplicate values can be stored. Since List preserves the insertion order it allows positional access and insertion of elements. List Interface is implemented by ArrayList, LinkedList, Vector and Stack classes.

The List interface provides a special iterator, called a **ListIterator**, that allows element insertion and replacement, and bidirectional access in addition to the normal operations that the Iterator interface provides. A method is provided to obtain a list iterator that starts at a specified position in the list.

1. **ArrayList – Interface**
   1. Methods:
      1. void add​(int index, E element) : Inserts the specified element at the specified position in this list. Shifts the element by 1 position. Throws [IndexOutOfBoundsException](https://docs.oracle.com/javase/10/docs/api/java/lang/IndexOutOfBoundsException.html) - if the index is out of range (index < 0 || index > size())
      2. boolean add​(E e)
      3. boolean addAll​(int index,Collection<? extends E> c)
      4. boolean addAll​(Collection<? extendsE> c)
      5. void clear()
      6. Object clone() : Returns a shallow copy of this ArrayList instance.
      7. boolean contains​(Object o)
      8. void ensureCapacity​(int minCapacity) : Increases the capacity of this ArrayList instance, if necessary, to ensure that it can hold at least the number of elements specified by the minimum capacity argument.
      9. void forEach​(Consumer<? superE> action)
      10. E get​(int index)
      11. int indexOf​(Object o) : -1 if this list does not contain the element.
      12. boolean isEmpty()
      13. Iterator<E> iterator()
      14. int lastIndexOf​(Object o)
      15. ListIterator<E> listIterator()
      16. ListIterator<E> listIterator​(int index)
      17. E remove​(int index)
      18. boolean remove​(Object o)
      19. boolean removeAll​(Collection<?> c)
      20. boolean removeIf​(Predicate<? superE> filter)
      21. protected void removeRange​(int fromIndex, int toIndex) : Removes from this list all of the elements whose index is between fromIndex, inclusive, and toIndex, exclusive.
      22. boolean retainAll​(Collection<?> c)
      23. E set​(int index, E element)
      24. int size()
      25. Spliterator<E> spliterator()
      26. List<E> subList​(int fromIndex, int toIndex)
      27. Object[] toArray()
      28. <T> T[] toArray​(T[] a)
      29. void trimToSize() : Trims the capacity of this ArrayList instance to be the list's current size.
   2. **Desc**: ArrayList is initialized by a size, however the size can increase if collection grows or shrunk if objects are removed from the collection. Initial capacity is 10.

Load factor is 1 and growth rate = current\_size + 0.5 \* current\_size = 1.5 current\_size

* 1. **Working :** Uses Object[] .
  2. **Constructor :**
     1. **ArrayList()** : Constructs an empty list with an initial capacity of ten.
     2. **ArrayList​(int initialCapacity) :**  Constructs an empty list with the specified initial capacity.
     3. **ArrayList​(Collection<? extendsE> c) :** Constructs a list containing the elements of the specified collection, in the order they are returned by the collection's iterator.

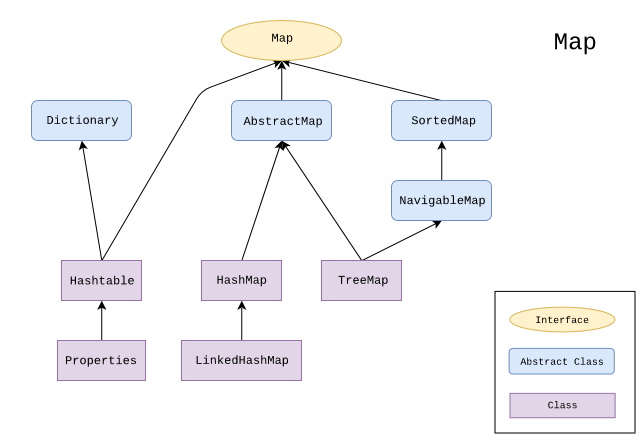
1. **Vector – Class**
   1. **Methods:**
      1. void add​(int index, E element)
      2. Boolean add​(E e)
      3. Boolean addAll​(int index,Collection<? extends E> c)
      4. Boolean addAll​(Collection<? extendsE> c)
      5. void addElement​(E obj) : Adds the specified component to the end of this vector, increasing its size by one.
      6. int capacity() : Returns the current capacity of this vector.
      7. void clear()
      8. Object clone()
      9. boolean contains​(Object o)
      10. boolean containsAll​(Collection<?> c)
      11. void copyInto​(Object[] anArray) : Copies the components of this vector into the specified array.
      12. E elementAt​(int index)
      13. Enumeration<E> elements() : Returns an enumeration of the components of this vector.
      14. void ensureCapacity​(int minCapacity) : Increases the capacity of this vector, if necessary, to ensure that it can hold at least the number of components specified by the minimum capacity argument.
      15. boolean equals​(Object o)
      16. E firstElement()
      17. void forEach​(Consumer<? superE> action)
      18. E get​(int index)
      19. int hashCode()
      20. int indexOf​(Object o) : Returns the index of the first occurrence of the specified element in this vector, or -1 if this vector does not contain the element.
      21. int indexOf​(Object o, int index) : Returns the index of the first occurrence of the specified element in this vector, searching forwards from index, or returns -1 if the element is not found.
      22. void insertElementAt​(E obj, int index)
      23. boolean isEmpty()
      24. Iterator<E> iterator()
      25. E lastElement()
      26. int lastIndexOf​(Object o)
      27. int lastIndexOf​(Object o, int index) : Returns the index of the last occurrence of the specified element in this vector, searching backwards from index, or returns -1 if the element is not found.
      28. ListIterator<E> listIterator()
      29. ListIterator<E> listIterator​(int index)
      30. E remove​(int index)
      31. boolean remove​(Object o)
      32. boolean removeAll​(Collection<?> c)
      33. void removeAllElements()
      34. boolean removeElement​(Object obj) : Removes the first (lowest-indexed) occurrence of the argument from this vector.
      35. void removeElementAt​(int index)
      36. boolean removeIf​(Predicate<? superE> filter)
      37. protected void removeRange​(int fromIndex, int toIndex) : Removes from this list all of the elements whose index is between fromIndex, inclusive, and toIndex, exclusive.
      38. void replaceAll​(UnaryOperator<E> operator)
      39. boolean retainAll​(Collection<?> c)
      40. E set​(int index, E element)
      41. void setElementAt​(E obj, int index)
      42. void setSize​(int newSize)
      43. int size()
      44. Spliterator<E> spliterator()
      45. List<E> subList​(int fromIndex, int toIndex) : Returns a view of the portion of this List between fromIndex, inclusive, and toIndex, exclusive.
      46. Object[] toArray()
      47. <T> T[] toArray​(T[] a)
      48. String toString()
      49. void trimToSize()
   2. **Desc:** Vector implements a dynamic array that means it can grow or shrink as required. Like an array, it contains components that can be accessed using an integer index

They are very similar to ArrayList but Vector is **synchronised** and have some legacy method which collection framework does not contain.

* 1. **Constructor:**
     1. **Vector()** : Constructs an empty vector so that its internal data array has size 10 and its standard capacity increment is zero.
     2. **Vector​(int initialCapacity)** : Constructs an empty vector with the specified initial capacity and with its capacity increment equal to zero.
     3. **Vector​(int initialCapacity, int capacityIncrement)** : Constructs an empty vector with the specified initial capacity and capacity increment.
     4. **Vector​(Collection<? extends E> c)** : Constructs a vector containing the elements of the specified collection, in the order they are returned by the collection's iterator.

1. **Stack – Class**
   1. **Method:**
      1. boolean empty()
      2. E peek() : Looks at the object at the top of this stack without removing it from the stack.
      3. E pop() : Removes the object at the top of this stack and returns that object as the value of this function.
      4. E push​(E item) : Pushes an item onto the top of this stack.
      5. int search​(Object o) : It determines whether an object exists in the stack. If the element is found, it returns the position of the element from the top of the stack. Else, it returns -1
   2. **Desc:** It is a collection that is based on the last in first out (LIFO) principle. On Creation, a stack is empty.
   3. **Constructor:**
      1. **Stack() :** Creates an empty Stack.

**MAPS**



1. **Map - Interface:**
   1. **Methods:**
      1. void clear() : default V compute​(K key, BiFunction<? super K,? super V,? extends V> remappingFunction) : Attempts to compute a mapping for the specified key and its current mapped value (or null if there is no current mapping).
      2. 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 and enters it into this map unless null.
      3. default V computeIfPresent​(K key, BiFunction<? super K,? super V,? extends V> remappingFunction) : If the value for the specified key is present and non-null, attempts to compute a new mapping given the key and its current mapped value.
      4. boolean containsKey​(Object key) :
      5. boolean containsValue​(Object value)
      6. static <K,V> Map<K,V> copyOf​(Map<? extends K,? extends V> map) : Returns an [unmodifiable Map](https://docs.oracle.com/javase/10/docs/api/java/util/Map.html#unmodifiable) containing the entries of the given Map
      7. static <K,V> Map.Entry<K,V> entry​(K k, V v)
      8. Set<Map.Entry<K,V>> entrySet()
      9. boolean equals​(Object o)
      10. default void forEach​(BiConsumer<? super K,? super V> action)
      11. V get​(Object key)
      12. default V getOrDefault​(Object key, V defaultValue) : Returns the value to which the specified key is mapped, ordefaultValue if this map contains no mapping for the key.
      13. int hashCode()
      14. boolean isEmpty()
      15. Set<K> keySet()
      16. default V merge​(K key, V value, BiFunction<? super V,? super V,? extends V> remappingFunction) : If the specified key is not already associated with a value or is associated with null, associates it with the given non-null value.
      17. static <K,V> Map<K,V> of() : Returns an unmodifiable map containing zero mappings.
      18. static <K,V> Map<K,V> of​(K k1, V v1)
      19. static <K,V> Map<K,V> ofEntries​(Map.Entry<? extends K,? extends V>... entries) : Returns an unmodifiable map containing keys and values extracted from the given entries.
      20. V put​(K key, V value)
      21. void putAll​(Map<? extends K,? extends V> m) : Copies all of the mappings from the specified map to this map (optional operation).
      22. 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 and returns null, else returns the current value.
      23. V remove​(Object key)
      24. default boolean remove​(Object key, Object value)
      25. default V replace​(K key, V value) : Replaces the entry for the specified key only if it is currently mapped to some value.
      26. default boolean replace​(K key, V oldValue, V newValue)
      27. default void replaceAll​(BiFunction<? super K,? super V,? extends V> function)
      28. int size()
      29. Collection<V> values() : Returns a [Collection](https://docs.oracle.com/javase/10/docs/api/java/util/Collection.html) view of the values contained in this map
   2. **Desc :** An object that maps keys to values. A map cannot contain duplicate keys; each key can map to at most one value. Some implementations allow null key and null value ([HashMap](https://www.geeksforgeeks.org/hashmap-treemap-java)and [LinkedHashMap](https://www.geeksforgeeks.org/linkedhashmap-class-java-examples)) but some do not ([TreeMap](https://www.geeksforgeeks.org/hashmap-treemap-java)).
2. **Dictionary - Interface:**
   1. **Methods:**
      1. abstract Enumeration<V> elements()
      2. abstract V get​(Object key)
      3. abstract boolean isEmpty()
      4. abstract Enumeration<K> keys()
      5. abstract V put​(K key, V value)
      6. abstract V remove​(Object key)
      7. abstract int size()
   2. **Desc:** The Dictionary class is the abstract parent of any class, such as Hashtable, which maps keys to values. Every key and every value is an object. In any one Dictionaryobject, every key is associated with at most one value. Given a Dictionary and a key, the associated element can be looked up. Any non-null object can be used as a key and as a value.
3. **HashTable – Class**
   1. **Methods:**
      1. void clear()
      2. Object clone()
      3. V compute​(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)
      4. V computeIfAbsent​(K key, Function<? superK,? extends V> mappingFunction)
      5. V computeIfPresent​(K key, BiFunction<? super K,? super V,? extendsV> remappingFunction)
      6. boolean contains​(Object value)
      7. boolean containsKey​(Object key)
      8. boolean containsValue​(Object value)
      9. Enumeration<V> elements()
      10. Set<Map.Entry<K,V>> entrySet()
      11. boolean equals​(Object o)
      12. V get​(Object key)
      13. int hashCode()
      14. boolean isEmpty()
      15. Enumeration<K> keys()
      16. Set<K> keySet()
      17. V merge​(K key, V value, BiFunction<? superV,? super V,? extendsV> remappingFunction)
      18. V put​(K key, V value)
      19. void putAll​(Map<? extends K,? extends V> t)
      20. protected void rehash()
      21. V remove​(Object key)
      22. int size()
      23. String toString()
      24. Collection<V> values()
   2. **Desc:** Any non-null object can be used as a key or as a value. To successfully store and retrieve objects from a hashtable, the objects used as keys must implement the hashCode method and the equals method. LoadFactor – 0.75 (default) Default size 11
      1. It is similar to HashMap, but is synchronised.
      2. Hashtable stores key/value pair in hash table.
      3. In Hashtable we specify an object that is used as a key, and the value we want to associate to that key. The key is then hashed, and the resulting hash code is used as the index at which the value is stored within the table.
   3. **Constructor:**
      1. Hashtable(): This is the default constructor.
      2. Hashtable(int size): This creates a hash table that has initial size specified by size.
      3. Hashtable(int size, float loadRatio): This version creates a hash table that has initial size specified by size and fill ratio specified by loadRatio. fill ratio: Basically it determines how full hash table can be before it is resized upward.and its Value lie between 0.0 to 1.0
      4. Hashtable(Map m): This creates a hash table that is initialised with the elements in m.
4. **Properties – Class**
   1. **Methods:**
      1. String getProperty​(String key) : Searches for the property with the specified key in this property list.
      2. String getProperty​(String key,String defaultValue) : Searches for the property with the specified key in this property list.
      3. void list​(PrintStream out) : Prints this property list out to the specified output stream.
      4. void list​(PrintWriter out) : Prints this property list out to the specified output stream.
      5. void load​(InputStream inStream) : Reads a property list (key and element pairs) from the input byte stream.
      6. void load​(Reader reader) : Reads a property list (key and element pairs) from the input character stream in a simple line-oriented format.
      7. void loadFromXML​(InputStream in) : Loads all of the properties represented by the XML document on the specified input stream into this properties table.
      8. Enumeration<?> propertyNames() : Returns an enumeration of all the keys in this property list, including distinct keys in the default property list if a key of the same name has not already been found from the main properties list.
      9. void save​(OutputStream out,String comments) : Deprecated. This method does not throw an IOException if an I/O error occurs while saving the property list.
      10. Object setProperty​(String key,String value) : Calls the Hashtable method put.
      11. void store​(OutputStream out,String comments) : Writes this property list (key and element pairs) in this Properties table to the output stream in a format suitable for loading into a Properties table using the load(InputStream) method.
      12. void store​(Writer writer,String comments) : Writes this property list (key and element pairs) in this Properties table to the output character stream in a format suitable for using the load(Reader) method.
      13. void storeToXML​(OutputStream os,String comment) : Emits an XML document representing all of the properties contained in this table.
      14. void storeToXML​(OutputStream os,String comment,String encoding) : Emits an XML document representing all of the properties contained in this table, using the specified encoding.
      15. void storeToXML​(OutputStream os,String comment,Charset charset) : Emits an XML document representing all of the properties contained in this table, using the specified encoding.
      16. Set<String> stringPropertyNames() : Returns an unmodifiable set of keys from this property list where the key and its corresponding value are strings, including distinct keys in the default property list if a key of the same name has not already been found from the main properties list.
   2. **Desc:** Properties is a very specialized class that's designed to hold configuration and/or resources that are usually stored in some file.
   3. **Constructor:**
      1. public Properties()
      2. public Properties​(Properties defaults) : Creates an empty property list with the specified defaults.
      3. public Properties​(int initialCapacity) : Creates an empty property list with no default values, and with an initial size accommodating the specified number of elements without the need to dynamically resize.
5. **AbstractMap – Class**
   1. **Methods:**
      1. void clear()
      2. protected Object clone()
      3. boolean containsKey​(Object key)
      4. boolean containsValue​(Object value)
      5. boolean equals​(Object o)
      6. V get​(Object key)
      7. int hashCode()
      8. boolean isEmpty()
      9. Set<K> keySet()
      10. V put​(K key, V value)
      11. void putAll​(Map<? extends K,? extendsV> m)
      12. V remove​(Object key)
      13. int size()
      14. String toString()
      15. Collection<V> values()
   2. **Desc:** This class provides a skeletal implementation of the Map interface, to minimize the effort required to implement this interface.
6. **HashMap – Class** 
   1. **Methods:**
      1. void clear()
      2. Object clone()
      3. V compute​(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)
      4. V computeIfAbsent​(K key, Function<? superK,? extends V> mappingFunction)
      5. V computeIfPresent​(K key, BiFunction<? super K,? super V,? extendsV> remappingFunction)
      6. boolean containsKey​(Object key)
      7. boolean containsValue​(Object value)
      8. Set<Map.Entry<K,V>> entrySet()
      9. V get​(Object key)
      10. boolean isEmpty()
      11. Set<K> keySet()
      12. V merge​(K key, V value, BiFunction<? superV,? super V,? extendsV> remappingFunction)
      13. V put​(K key, V value)
      14. void putAll​(Map<? extends K,? extends V> m)
      15. V remove​(Object key)
      16. int size()
      17. Collection<V> values()
   2. **Desc:** Hash table based implementation of the Map interface. This implementation provides all of the optional map operations, and permits null values and the null key. (The HashMap class is roughly equivalent to Hashtable, except that it is **unsynchronized** and permits nulls.) This class makes no guarantees as to the order of the map; in particular, it does not guarantee that the order will remain constant over time.

Default size - 16 , loadfactor – 0.75

* 1. **Working:** Internally HashMap contains an array of Node. and a node is represented as a class which contains 4 field
     1. int hash
     2. K key
     3. V value
     4. Node next

We can see that node is containing a reference of its own object. So it’s a linked list.

* 1. **Constructor:**
     1. **HashMap()** : It is the default constructor which creates an instance of HashMap with initial capacity 16 and load factor 0.75.
     2. **HashMap(int initial capacity)** : It creates a HashMap instance with specified initial capacity and load factor 0.75.
     3. **HashMap(int initial capacity, float loadFactor)** : It creates a HashMap instance with specified initial capacity and specified load factor.
     4. **HashMap(Map map)** : It creates instance of HashMapwith same mappings as specified map.

1. **TreeMap – Class:**
   1. **Methods:**
      1. Map.Entry<K,V> ceilingEntry​(K key) : Returns a key-value mapping associated with the least key greater than or equal to the given key, or null if there is no such key.
      2. K ceilingKey​(K key) : Returns the least key greater than or equal to the given key, or null if there is no such key.
      3. void clear()
      4. Object clone()
      5. boolean containsKey​(Object key)
      6. boolean containsValue​(Object value)
      7. NavigableSet<K> descendingKeySet()
      8. NavigableMap<K,V> descendingMap()
      9. Set<Map.Entry<K,V>> entrySet()
      10. Map.Entry<K,V> firstEntry() : Returns a key-value mapping associated with the least key in this map, or null if the map is empty.
      11. K firstKey() : Returns the first (lowest) key currently in this map.
      12. Map.Entry<K,V> floorEntry​(K key) : Returns a key-value mapping associated with the greatest key less than or equal to the given key, or null if there is no such key.
      13. K floorKey​(K key) : Returns the greatest key less than or equal to the given key, or null if there is no such key.
      14. V get​(Object key)
      15. SortedMap<K,V> headMap​(K toKey) : Returns a view of the portion of this map whose keys are strictly less than toKey.
      16. NavigableMap<K,V> headMap​(K toKey, boolean inclusive)
      17. Map.Entry<K,V> higherEntry​(K key) : Returns a key-value mapping associated with the least key strictly greater than the given key, or null if there is no such key.
      18. K higherKey​(K key) : Returns the least key strictly greater than the given key, or null if there is no such key.
      19. Set<K> keySet()
      20. Map.Entry<K,V> lastEntry() : Returns a key-value mapping associated with the greatest key in this map, or null if the map is empty.
      21. K lastKey() : Returns the last (highest) key currently in this map.
      22. Map.Entry<K,V> lowerEntry​(K key) : Returns a key-value mapping associated with the greatest key strictly less than the given key, or null if there is no such key.
      23. K lowerKey​(K key) : Returns the greatest key strictly less than the given key, or null if there is no such key.
      24. NavigableSet<K> navigableKeySet()
      25. Map.Entry<K,V> pollFirstEntry() : Removes and returns a key-value mapping associated with the least key in this map, or null if the map is empty.
      26. Map.Entry<K,V> pollLastEntry() : Removes and returns a key-value mapping associated with the greatest key in this map, or null if the map is empty.
      27. V put​(K key, V value)
      28. void putAll​(Map<? extends K,? extends V> map)
      29. V remove​(Object key)
      30. int size()
      31. NavigableMap<K,V> subMap​(K fromKey, boolean fromInclusive,K toKey, boolean toInclusive) : Returns a view of the portion of this map whose keys range from fromKey to toKey
      32. 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.
      33. SortedMap<K,V> tailMap​(K fromKey)
      34. NavigableMap<K,V> tailMap​(K fromKey, boolean inclusive) : Returns a view of the portion of this map whose keys are greater than or equal to fromKey.
      35. Collection<V> values()
   2. **Desc:** A TreeMap stores its data in a hierarchical tree with the ability to sort the elements with the help of a custom Comparator. It is same as HashMap instead maintains ascending order(Sorted using the natural order of its key). It cannot have null key but can have multiple null values
   3. **Constructor:**
      1. public TreeMap()
      2. public TreeMap​(Comparator<? super K> comparator)
      3. public TreeMap​([SortedMap](https://docs.oracle.com/javase/10/docs/api/java/util/SortedMap.html)<[K](https://docs.oracle.com/javase/10/docs/api/java/util/TreeMap.html),? extends [V](https://docs.oracle.com/javase/10/docs/api/java/util/TreeMap.html)> m) : Constructs a new tree map containing the same mappings and using the same ordering as the specified sorted map. This method runs in linear time.
      4. public TreeMap​([Map](https://docs.oracle.com/javase/10/docs/api/java/util/Map.html)<? extends [K](https://docs.oracle.com/javase/10/docs/api/java/util/TreeMap.html),? extends [V](https://docs.oracle.com/javase/10/docs/api/java/util/TreeMap.html)> m) : Constructs a new tree map containing the same mappings as the given map, ordered according to the *natural ordering* of its keys. All keys inserted into the new map must implement the [Comparable](https://docs.oracle.com/javase/10/docs/api/java/lang/Comparable.html) interface. Furthermore, all such keys must be *mutually comparable*: k1.compareTo(k2) must not throw a ClassCastException for any keys k1 and k2 in the map.
2. **LinkedHahMap – Class:**
   1. **Method:**
      1. boolean containsValue​(Object value)
      2. Set<Map.Entry<K,V>> entrySet()
      3. V get​(Object key)
      4. Set<K> keySet()
      5. protected boolean removeEldestEntry​(Map.Entry<K,V> eldest)
      6. Collection<V> values()
   2. **Desc :** LinkedHashMap is a subclass of HashMap. That means it inherits the features of HashMap. In addition, the linked list preserves the insertion-order.
   3. **Constructor :**
      1. **LinkedHashMap() :** Constructs an empty insertion-ordered LinkedHashMap instance with the default initial capacity (16) and load factor (0.75).
      2. **LinkedHashMap​(int initialCapacity) :** Constructs an empty insertion-ordered LinkedHashMap instance with the specified initial capacity and a default load factor (0.75).
      3. **LinkedHashMap​(int initialCapacity, float loadFactor) :** Constructs an empty insertion-ordered LinkedHashMap instance with the specified initial capacity and load factor.
      4. **LinkedHashMap​(int initialCapacity, float loadFactor, boolean accessOrder) :** Constructs an empty LinkedHashMap instance with the specified initial capacity, load factor and ordering mode.

accessOrder - the ordering mode - true for access-order, false for insertion-order

* + 1. **LinkedHashMap​(Map<? extends K,? extends V> m) :** Constructs an insertion-ordered LinkedHashMap instance with the same mappings as the specified map.