# JAVA
# Collection

By :

Subodh Sharma

An array is an indexed collection of fixed number of homogeneous data element. The main advantage of arrays is, we can represent multiple values with a single variable. So that reusability of the code will be improved.

Limitations of Arrays :

1. Arrays are fixed in size.
2. Arrays can hold only homogeneous data elements.
3. Arrays concept is not implemented based on some standard data structure hence readymade method support is not available for every requirement, so we have to write the code explicitly.

**To overcome the above limitations of Arrays (and other data structures like linked list, quest, tree), we should go for Collections.**

1. Collections are growable in natures.
2. Collections can hold both homogeneous & Heterogeneous elements.
3. Every Collection class is implemented based on some standard data structure. Hence readymade method support is available for every requirement.

**Collection Framework:** Java collection framework works with objects only.

# Collection in java :

The **Collection (API) in Java** is a framework that provides an architecture to store and manipulate the group of objects. Java Collections can achieve all the operations that you perform on a data such as searching, sorting, insertion, manipulation, and deletion.

All classes of the collection framework (ArrayList, LinkedList, Vector, HashSet, LinkedHashSet, TreeSet, PriorityQueue, ArrayDeque, etc.) deal with objects only. Java Collection means a single unit of objects.

The Collection framework represents a unified architecture for storing and manipulating a group of objects. It has:

1. Interfaces and its implementations, i.e., classes
2. Algorithm

Java Collection framework provides many interfaces (Set, List, Queue, Deque) and classes (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet).

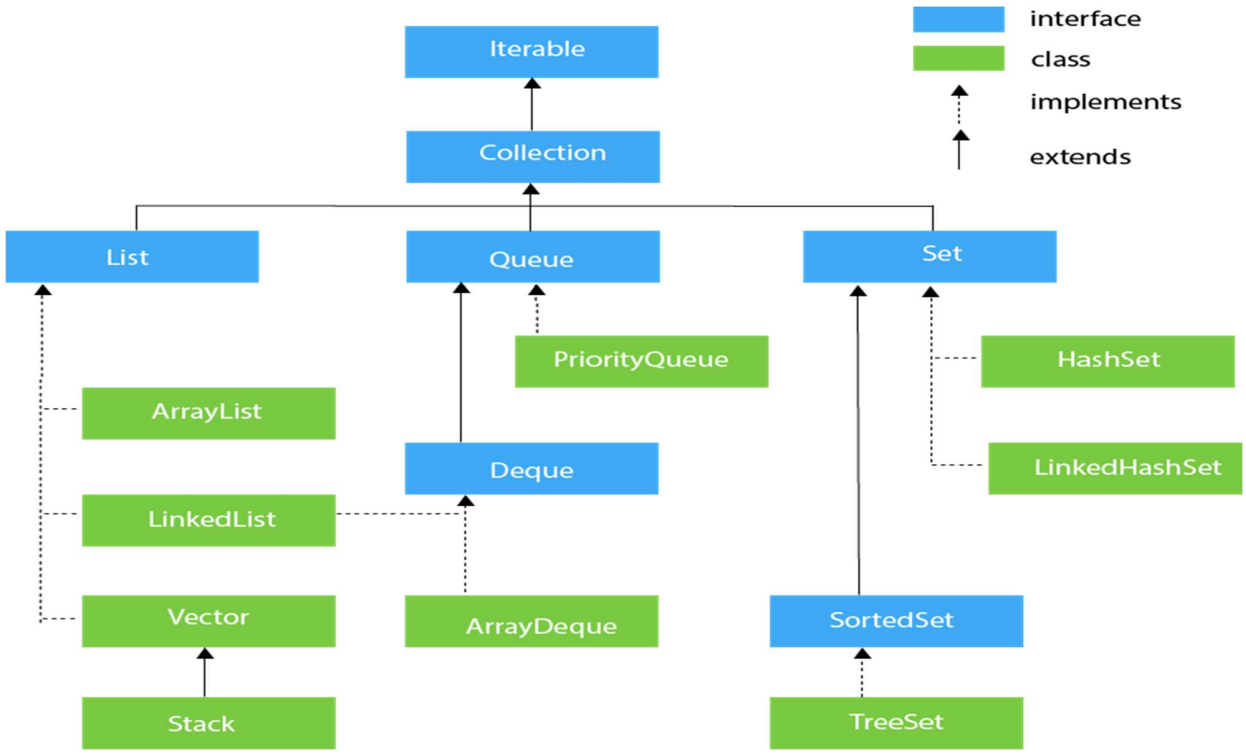
**Benefits of using Collection Framework :**

1. Collection framework can contain only non-primitive type of data. (not int but Integer datatypes)
2. Can store heterogeneous type of data.
3. We can increase or decrease the size of collections at runtime.
4. It is an API (Application Programming Interface) which provides predefined classes, interface & methods.

**Collection framework contains 2 parts : java.util.Collection java.util.Map (key value pairs)**

## Hierarchy of Collection Framework

Let us see the hierarchy of Collection framework. The **java.util** package contains all the classes and interfaces for the Collection framework.




# Iterator Interface

Iterator interface provides the facility of iterating the elements in a forward direction only. The Iterable interface is the root interface for all the collection classes. The Collection interface extends the Iterable interface and therefore all the subclasses of Collection interface also implement the Iterable interface.

### *Methods of Iterator interface*

There are only three methods in the Iterator interface. They are:

| No. | Method | Description |
|---|---|---|
| 1 | public boolean hasNext() | It returns true if the iterator has more elements otherwise it returns false. |
| 2 | public Object next() | It returns the element and moves the cursor pointer to the next element. |
| 3 | public void remove() | It removes the last elements returned by the iterator. It is less used. |

It contains only one abstract method. i.e., Iterator<T> iterator()
It returns the iterator over the elements of type T.

**'ListIterator'** in Java is an Iterator which allows users to traverse Collection in both direction. It contains the following methods:

1. **void add(Object object)**: It inserts object immediately before the element that is returned by the next( ) function.
2. **boolean hasNext( )**: It returns true if the list has a next element.
3. **boolean hasPrevious( )**: It returns true if the list has a previous element.
4. **Object next( )**: It returns the next element of the list. It throws 'NoSuchElementException' if there is no next element in the list.
5. **Object previous( )**: It returns the previous element of the list. It throws 'NoSuchElementException' if there is no previous element.
6. **void remove( )**: It removes the current element from the list. It throws 'IllegalStateException' if this function is called before next( ) or previous( ) is invoked.

## Collection Interface

The Collection interface is the interface which is implemented by all the classes in the collection framework. It declares the methods that every collection will have. In other words, we can say that the Collection interface builds the foundation on which the collection framework depends. Some of the methods of Collection interface are Boolean add ( Object obj), Boolean addAll ( Collection c), void clear(), etc. which are implemented by all the subclasses of Collection interface.

## Methods of Collection interface

There are many methods declared in the Collection interface. They are as follows:

| No. | Method | Description |
|---|---|---|
| 1 | public boolean add(E e) | It is used to insert an element in this collection. |
| 2 | public boolean addAll(Collection<? extends E> c) | It is used to insert the specified collection elements in the invoking collection. |
| 3 | public boolean remove(Object element) | It is used to delete an element from the collection. |
| 4 | public boolean removeAll(Collection<?> c) | It is used to delete all the elements of the specified collection from the invoking collection. |
| 5 | default boolean removeIf(Predicate<? super E> filter) | It is used to delete all the elements of the collection that satisfy the specified predicate. |
| 6 | public boolean retainAll(Collection<?> c) | It is used to delete all the elements of invoking collection except the specified collection. |
| 7 | public int size() | It returns the total number of elements in the |

| | | |
|---|---|---|
| | | collection. |
| 8 | public void clear() | It removes the total number of elements from the collection. |
| 9 | public boolean contains(Object element) | It is used to search an element. |
| 10 | public boolean containsAll(Collection<?> c) | It is used to search the specified collection in the collection. |
| 11 | public Iterator iterator() | It returns an iterator. |
| 12 | public Object[] toArray() | It converts collection into array. |
| 13 | public <T> T[] toArray(T[] a) | It converts collection into array. Here, the runtime type of the returned array is that of the specified array. |
| 14 | public boolean isEmpty() | It checks if collection is empty. |
| 15 | default Stream<E> parallelStream() | It returns a possibly parallel Stream with the collection as its source. |
| 16 | default Stream<E> stream() | It returns a sequential Stream with the collection as its source. |
| 17 | default Spliterator<E> spliterator() | It generates a Spliterator over the specified elements in the collection. |
| 18 | public boolean equals(Object element) | It matches two collections. |
| 19 | public int hashCode() | It returns the hash code number of the collection. |

## List Interface

List interface is the child interface of Collection interface. It inhibits a list type data structure in which we can store the ordered collection of objects. It can have duplicate values.
List interface is implemented by the classes ArrayList, LinkedList, Vector, and Stack.
To instantiate the List interface, we must use :

1. List <data-type> list1= **new** ArrayList();
2. List <data-type> list2 = **new** LinkedList();
3. List <data-type> list3 = **new** Vector();
4. List <data-type> list4 = **new** Stack();

There are various methods in List interface that can be used to insert, delete, and access the elements from the list.
The classes that implement the List interface are given below.

## ArrayList

The ArrayList class implements the List interface. It uses a dynamic array to store the duplicate element of different data types. The ArrayList class maintains the insertion order and is non-synchronized. The elements stored in the ArrayList class can be randomly accessed. Consider the following example.

## LinkedList

LinkedList implements the Collection interface. It uses a doubly linked list internally to store the elements. It can store the duplicate elements. It maintains the insertion order and is not synchronized. In LinkedList, the manipulation is fast because no shifting is required.

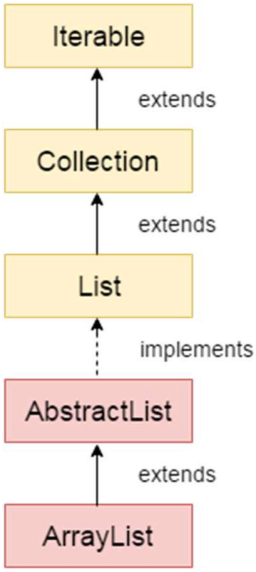
## Vector

Vector uses a dynamic array to store the data elements. It is similar to ArrayList. However, It is synchronized and contains many methods that are not the part of Collection framework.
Consider the following example.

# Java ArrayList

ArrayList is a part of collection framework and is present in java.util package. It provides us dynamic arrays in Java. Though, it may be slower than standard arrays but can be helpful in programs where lots of manipulation in the array is needed.

- ArrayList inherits AbstractList class and implements List interface.
- ArrayList is initialized by a size, however the size can increase if collection grows or shrunk if objects are removed from the collection.
- Java ArrayList allows us to randomly access the list.
- ArrayList can not be used for primitive types, like int, char, etc. We need a wrapper class for such cases.
- ArrayList in Java can be seen as similar to vector in C++.




The difference between a built-in array and an ArrayList in Java, is that the size of an array cannot be modified (if you want to add or remove elements to/from an array, you have to create a new one). While elements can be added and removed from an ArrayList whenever you want.

ArrayList has the constructors show here :
ArrayList()
ArrayList(Collection c)
ArrayList(int capcacity)

### Create an array list :

```
ArrayList <String>  list = new ArrayList<String>();
```

**Methods in Java ArrayList:**

1. forEach(Consumer<? super E> action): Performs the given action for each element of the Iterable until all elements have been processed or the action throws an exception.
2. retainAll(Collection<?> c): Retains only the elements in this list that are contained in the specified collection.

3. removeIf(Predicate<? super E> filter): Removes all of the elements of this collection that satisfy the given predicate.
4. contains(Object o): Returns true if this list contains the specified element.
5. remove(int index): Removes the element at the specified position in this list.
6. remove(Object o): Removes the first occurrence of the specified element from this list, if it is present.
7. get(int index): Returns the element at the specified position in this list.
8. subList(int fromIndex, int toIndex): Returns a view of the portion of this list between the specified fromIndex, inclusive, and toIndex, exclusive.
9. spliterator(): Creates a late-binding and fail-fast Spliterator over the elements in this list.
10. set(int index, E element): Replaces the element at the specified position in this list with the specified element.
11. size(): Returns the number of elements in this list.
12. removeAll(Collection<?> c): Removes from this list all of its elements that are contained in the specified collection.
13. 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.
14. listIterator(): Returns a list iterator over the elements in this list (in proper sequence).
15. listIterator(int index): Returns a list iterator over the elements in this list (in proper sequence), starting at the specified position in the list.
16. isEmpty(): Returns true if this list contains no elements.
17. removeRange(int fromIndex, int toIndex): Removes from this list all of the elements whose index is between fromIndex, inclusive, and toIndex, exclusive.
18. void clear(): This method is used to remove all the elements from any list.
19. void add(int index, Object element): This method is used to insert a specific element at a specific position index in a list.
20. void trimToSize(): This method is used to trim the capacity of the instance of the ArrayLis to the list's current size.
21. int indexOf(Object O): The index the first occurrence of a specific element is either returned, or -1 in case the element is not in the list.
22. int lastIndexOf(Object O): The index the last occurrence of a specific element is either returned, or -1 in case the element is not in the list.
23. Object clone(): This method is used to return a shallow copy of an ArrayList.
24. Object[] toArray(): This method is used to return an array containing all of the elements in the list in correct order.
25. Object[] toArray(Object[] O): It is also used to return an array containing all of the elements in this list in the correct order same as the previous method.
26. boolean addAll(Collection C): This method is used to append all the elements from a specific collection to the end of the mentioned list, in such a order that the values are returned by the specified collection's iterator.
27. boolean add(Object o): This method is used to append a specificd element to the end of a list.
28. boolean addAll(int index, Collection C): Used to insert all of the elements starting at the specified position from a specific collection into the mentioned list.

## Obtaining an Array from ArrayList:

Object[] toArray()

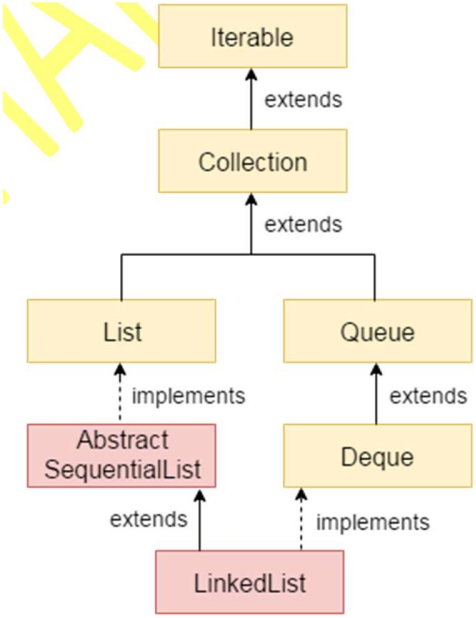
```
Object   oarray = arrayList.toArray()
For(int i=0;i<orray.length;i++)
  System.out.println((String)oarray[i]);
```

# Java LinkedList class

Linked List are linear data structures where the elements are not stored in contiguous locations and every element is a separate object with a data part and address part. The elements are linked using pointers and addresses. Each element is known as a node. Due to the dynamicity and ease of insertions and deletions, they are preferred over the arrays.
Java LinkedList class uses a doubly linked list to store the elements. It provides a linked-list data structure. It inherits the AbstractList class and implements List and Deque interfaces.

The important points about Java LinkedList are:

- Java LinkedList class can contain duplicate elements.
- Java LinkedList class maintains insertion order.
- Java LinkedList class is non synchronized.
- In Java LinkedList class, manipulation is fast because no shifting needs to occur.
- Java LinkedList class can be used as a list, stack or queue




## Hierarchy of LinkedList class

As shown in the above diagram, Java LinkedList class extends AbstractSequentialList class and implements List and Deque interfaces.

**Methods for Java LinkedList:**

1. **add(int index, E element):** This method Inserts the specified element at the specified position in this list.
2. **add(E e):** This method Appends the specified element to the end of this list.
3. **addAll(int index, Collection c):** This method Inserts all of the elements in the specified collection into this list, starting at the specified position.
4. **addAll(Collection c):** This method Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator.
5. **addFirst(E e):** This method Inserts the specified element at the beginning of this list.
6. **addLast(E e):** This method Appends the specified element to the end of this list.
7. **clear():** This method removes all of the elements from this list.
8. **clone():** This method returns a shallow copy of this LinkedList.
9. **contains(Object o):** This method returns true if this list contains the specified element.

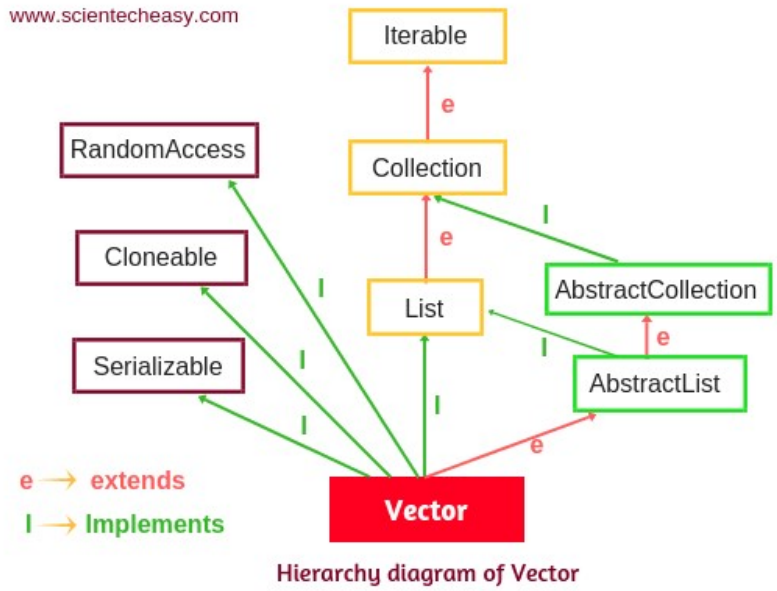
10. **descendingIterator():** This method returns an iterator over the elements in this deque in reverse sequential order.
11. **element():** This method retrieves, but does not remove, the head (first element) of this list.
12. **get(int index):** This method returns the element at the specified position in this list.
13. **getFirst():** This method returns the first element in this list.
14. **getLast():** This method returns the last element in this list.
15. **indexOf(Object o):** This method returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
16. **lastIndexOf(Object o):** This method returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element.
17. **listIterator(int index):** This method returns a list-iterator of the elements in this list (in proper sequence), starting at the specified position in the list.
18. **offer(E e):** This method Adds the specified element as the tail (last element) of this list.
19. **offerFirst(E e):** This method Inserts the specified element at the front of this list.
20. **offerLast(E e):** This method Inserts the specified element at the end of this list.
21. **peek():** This method retrieves, but does not remove, the head (first element) of this list.
22. **peekFirst():** This method retrieves, but does not remove, the first element of this list, or returns null if this list is empty.
23. **peekLast():** This method retrieves, but does not remove, the last element of this list, or returns null if this list is empty.
24. **poll():** This method retrieves and removes the head (first element) of this list.
25. **pollFirst():** This method retrieves and removes the first element of this list, or returns null if this list is empty.
26. **pollLast():** This method retrieves and removes the last element of this list, or returns null if this list is empty.
27. **pop():** This method Pops an element from the stack represented by this list.
28. **push(E e):** This method Pushes an element onto the stack represented by this list.
29. **remove():** This method retrieves and removes the head (first element) of this list.
30. **remove(int index):** This method removes the element at the specified position in this list.
31. **remove(Object o):** This method removes the first occurrence of the specified element from this list, if it is present.
32. **removeFirst():** This method removes and returns the first element from this list.
33. **removeFirstOccurrence(Object o):** This method removes the first occurrence of the specified element in this list (when traversing the list from head to tail).
34. **removeLast():** This method removes and returns the last element from this list.
35. **removeLastOccurrence(Object o):** This method removes the last occurrence of the specified element in this list (when traversing the list from head to tail).
36. **set(int index, E element):** This method replaces the element at the specified position in this list with the specified element.
37. **size():** This method returns the number of elements in this list.
38. **spliterator():** This method Creates a late-binding and fail-fast Spliterator over the elements in this list.
39. **toArray():** This method returns an array containing all of the elements in this list in proper sequence (from first to last element).
40. **toArray(T[] a):** This method returns an array containing all of the elements in this list in proper sequence (from first to last element); the runtime type of the returned array is that of the specified array.

# Java Vector

**Vector** is like the *dynamic array* which can grow or shrink its size. Unlike array, we can store n-number of elements in it as there is no size limit. It is a part of Java Collection framework since Java 1.2. It is found in the java.util package and implements the *List* interface, so we can use all the methods of List interface here.

It is recommended to use the Vector class in the thread-safe implementation only. If you don't need to use the thread-safe implementation, you should use the ArrayList, the ArrayList will perform better in such case.

Vector class has same features as ArrayList. Vector class also extends **AbstractList** class and implements **List interface**. It also implements 3 marker interfaces – **RandomAccess**, **Cloneable** and **Serializable**. Below is the hierarchy diagram of Vector class.



Hierarchy diagram of Vector

It is similar to the ArrayList, but with two differences-

- Vector is synchronized.
- Java Vector contains many legacy methods that are not the part of a collections framework. ( **legacy** means "should not be used anymore in new code".)
- Resize : ArrayList grow by half of its size when resized while Vector doubles the size of itself by default when grows.

## Java Vector class Declaration

```
public class Vector<E>
extends Object<E>
implements List<E>, Cloneable, Serializable
```

## Java Vector Constructors :

Vector class supports four types of constructors. These are given below:

| SN | Constructor | Description |
|---|---|---|
| 1) | vector() | It constructs an empty vector with the default size as 10. |
| 2) | vector(int initialCapacity) | It constructs an empty vector with the specified initial capacity and with its capacity increment equal to zero. |
| 3) | vector(int initialCapacity, int capacityIncrement) | It constructs an empty vector with the specified initial capacity and capacity increment. |
| 4) | Vector( Collection<? extends E> c) | It constructs a vector that contains the elements of a collection c. |

## Java Vector Methods:

The following are the list of Vector class methods:

| SN | Method | Description |
|---|---|---|
| 1) | add() | It is used to append the specified element in the given vector. |
| 2) | addAll() | It is used to append all of the elements in the specified collection to the end of this Vector. |
| 3) | addElement() | It is used to append the specified component to the end of this vector. It increases the vector size by one. |
| 4) | capacity() | It is used to get the current capacity of this vector. |
| 5) | clear() | It is used to delete all of the elements from this vector. |
| 6) | clone() | It returns a clone of this vector. |
| 7) | contains() | It returns true if the vector contains the specified element. |
| 8) | containsAll() | It returns true if the vector contains all of the elements in the specified collection. |
| 9) | copyInto() | It is used to copy the components of the vector into the specified array. |
| 10) | elementAt() | It is used to get the component at the specified index. |
| 11) | elements() | It returns an enumeration of the components of a vector. |
| 12) | ensureCapacity() | It is used to increase the capacity of the vector which is in use, if necessary. It ensures that the vector can hold at least the number of components specified by the minimum capacity argument. |
| 13) | equals() | It is used to compare the specified object with the vector for equality. |

| | | |
|---|---|---|
| 14) | firstElement() | It is used to get the first component of the vector. |
| 15) | forEach() | It is used to perform the given action for each element of the Iterable until all elements have been processed or the action throws an exception. |
| 16) | get() | It is used to get an element at the specified position in the vector. |
| 17) | hashCode() | It is used to get the hash code value of a vector. |
| 18) | indexOf() | It is used to get the index of the first occurrence of the specified element in the vector. It returns -1 if the vector does not contain the element. |
| 19) | insertElementAt() | It is used to insert the specified object as a component in the given vector at the specified index. |
| 20) | isEmpty() | It is used to check if this vector has no components. |
| 21) | iterator() | It is used to get an iterator over the elements in the list in proper sequence. |
| 22) | lastElement() | It is used to get the last component of the vector. |
| 23) | lastIndexOf() | It is used to get the index of the last occurrence of the specified element in the vector. It returns -1 if the vector does not contain the element. |
| 24) | listIterator() | It is used to get a list iterator over the elements in the list in proper sequence. |
| 25) | remove() | It is used to remove the specified element from the vector. If the vector does not contain the element, it is unchanged. |
| 26) | removeAll() | It is used to delete all the elements from the vector that are present in the specified collection. |
| 27) | removeAllElements() | It is used to remove all elements from the vector and set the size of the vector to zero. |
| 28) | removeElement() | It is used to remove the first (lowest-indexed) occurrence of the argument from the vector. |
| 29) | removeElementAt() | It is used to delete the component at the specified index. |
| 30) | removeIf() | It is used to remove all of the elements of the collection that satisfy the given predicate. |
| 31) | removeRange() | It is used to delete all of the elements from the vector whose index is between fromIndex, inclusive and toIndex, exclusive. |
| 32) | replaceAll() | It is used to replace each element of the list with the result of |

| | | |
|---|---|---|
| | | applying the operator to that element. |
| 33) | retainAll() | It is used to retain only that element in the vector which is contained in the specified collection. |
| 34) | set() | It is used to replace the element at the specified position in the vector with the specified element. |
| 35) | setElementAt() | It is used to set the component at the specified index of the vector to the specified object. |
| 36) | setSize() | It is used to set the size of the given vector. |
| 37) | size() | It is used to get the number of components in the given vector. |
| 38) | sort() | It is used to sort the list according to the order induced by the specified Comparator. |
| 39) | spliterator() | It is used to create a late-binding and fail-fast Spliterator over the elements in the list. |
| 40) | subList() | It is used to get a view of the portion of the list between fromIndex, inclusive, and toIndex, exclusive. |
| 41) | toArray() | It is used to get an array containing all of the elements in this vector in correct order. |
| 42) | toString() | It is used to get a string representation of the vector. |
| 43) | trimToSize() | It is used to trim the capacity of the vector to the vector's current size. |

# Set

Set is the child interface of Collection, If we want to represent a group of individual objects as a single entity, where duplicate are not allowed and insertion order is not preserved then we should go for Set.

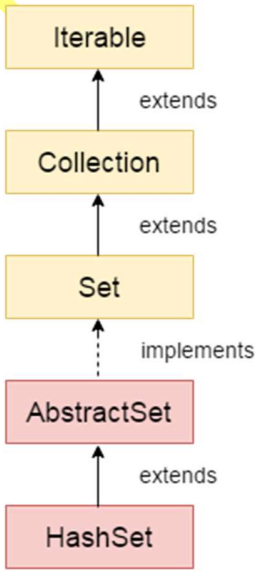
Set interface does not contain any new methods, So we have to use only Collection interface method.

# Java HashSet

Java HashSet class is used to create a collection that uses a hash table for storage. It inherits the AbstractSet class and implements Set interface.

## Hierarchy of HashSet class:

The HashSet class extends AbstractSet class which implements Set interface. The Set interface inherits Collection and Iterable interfaces in hierarchical order




The important points about Java HashSet class are:

1. Underlyin data structure is HashTable.
2. Duplicates are not allowed.If we are trying to insert duplicate, we would not get any compile time or run time error. If duplicate element, add function return false.
3. Insertion order not preserved, all objects will be inserted based on Hash code of objects.
4. Hetrogeneous element are allowed to store.
5. Null insertion allowed.
6. Implements Serialisable under clonable interface but not random access.
7. Searching is easy because there is Hash code of object.

## Difference between List and Set

A list can contain duplicate elements whereas Set contains unique elements only.

## There are four constructor of HashSet

**HashSet h = new HashSet();**

Empty with cap : 16 cap with ( fill ration) 0.75 load ratio
(after 75% filling 16 is againt alloated)

**HashSet h = new HashSet(int initial cap)**

(with 75% load capacity)

**HashSet h= new HashSet(int incap, float loadfactor)**

**HashSet= new HashSet(Collection c);**

```
import java.util.*;
class ColS1
{
  public static void main(String s[])
  {
     HashSet  h = new HashSet ();
       h.add("A");
       h.add("B");
       h.add("C");
       h.add(null);
       //h.add(10);
       h.add("A");
System.out.println(h);
  }
}
```

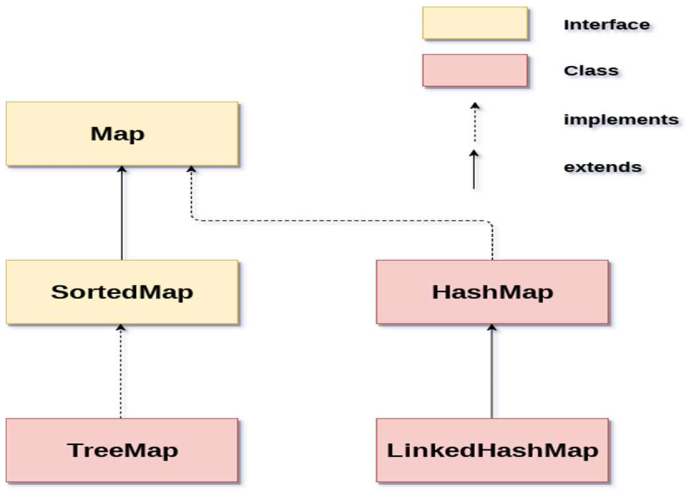
# Java Map Interface

Note that map has no relation with collection.
Since collection is group of individual objects while map is group of objects having key value pair i.e. a map contains values on the basis of key, i.e. key and value pair.
Each key and value pair is known as an (one) entry. A Map contains unique keys.
A Map is useful if you have to search, update or delete elements on the basis of a key.

## Java Map Hierarchy

There are two interfaces for implementing Map in java: **Map** and **SortedMap,** and three classes:
**HashMap :** HashMap is the implementation of Map, but it doesn't maintain any order.
**LinkedHashMap:** LinkedHashMap is the implementation of Map. It inherits HashMap class. It maintains insertion order.
**TreeMap** : TreeMap is the implementation of Map and SortedMap. It maintains ascending order.

The hierarchy of Java Map is given below:




A Map doesn't allow duplicate keys, but you can have duplicate values. HashMap and LinkedHashMap allow null keys and values, but TreeMap doesn't allow any null key or value.

A Map can't be traversed, so you need to convert it into Set using *keySet()* or *entrySet() method*.

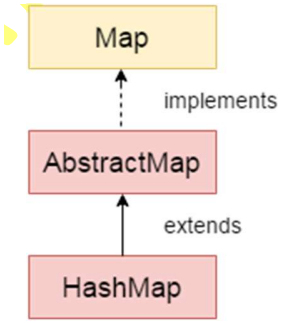
# Java HashMap

Java **HashMap** class implements the Map interface which allows us *to store key and value pair*, where keys should be unique. If you try to insert the duplicate key, it will replace the element of the corresponding key. It is easy to perform operations using the key index like updation, deletion, etc. HashMap class is found in the java.util package.

HashMap in Java is like the legacy Hashtable class, but it is not synchronized. It allows us to store the null elements as well, but there should be only one null key. Since Java 5, it is denoted as HashMap<K,V>, where K stands for key and V for value. It inherits the AbstractMap class and implements the Map interface.

## Hierarchy of HashMap class

As shown in the above figure, HashMap class extends AbstractMap class and implements Map interface.




## Important Points:

- Java HashMap contains values based on the key.
- Java HashMap contains only unique keys.
- Java HashMap may have one null key and multiple null values.
- Java HashMap is non synchronized.
- Java HashMap maintains no order.
- The initial default capacity of Java HashMap class is 16 with a load factor of 0.75.

```
import java.util.*;
class ColH1
{
  public static void main(String s[])
   {
   HashMap<String,String> hm= new HashMap<String,String>();
    hm.put("Ram", "March");
    hm.put("Sita", "October");
    hm.put("Mohan", "December");

Set<String> ks= hm.keySet();
for(String  name :  ks)
System.out.println(  name+":"+hm.get(name));
 }
}
```