## Why Collection Framework?

****Collections**** are nothing but group of objects stored in well defined manner. Earlier, Arrays are used to represent these group of objects. But, arrays are not re-sizable. size of the arrays are fixed. Size of the arrays can not be changed once they are defined. This causes lots of problem while handling group of objects. To overcome this drawback of arrays, ****Collection framework**** or simply collections are introduced in java from **JDK 1.2.**

**Collection Framework in java is a centralized and unified theme to store and manipulate the group of objects.** Java Collection Framework provides some pre-defined classes and interfaces to handle the group of objects. Using collection framework, you can store the objects as a ****list**** or as a ****set**** or as a ****queue**** or as a ****map**** and perform operations like adding an object or removing an object or sorting the objects without much hard work.

All classes and interfaces related to Collection Framework are placed in ****java.util**** package. ****java.util.Collection**** interface is at the top of class hierarchy of Collection Framework.

The entire collection framework is divided into four interfaces.

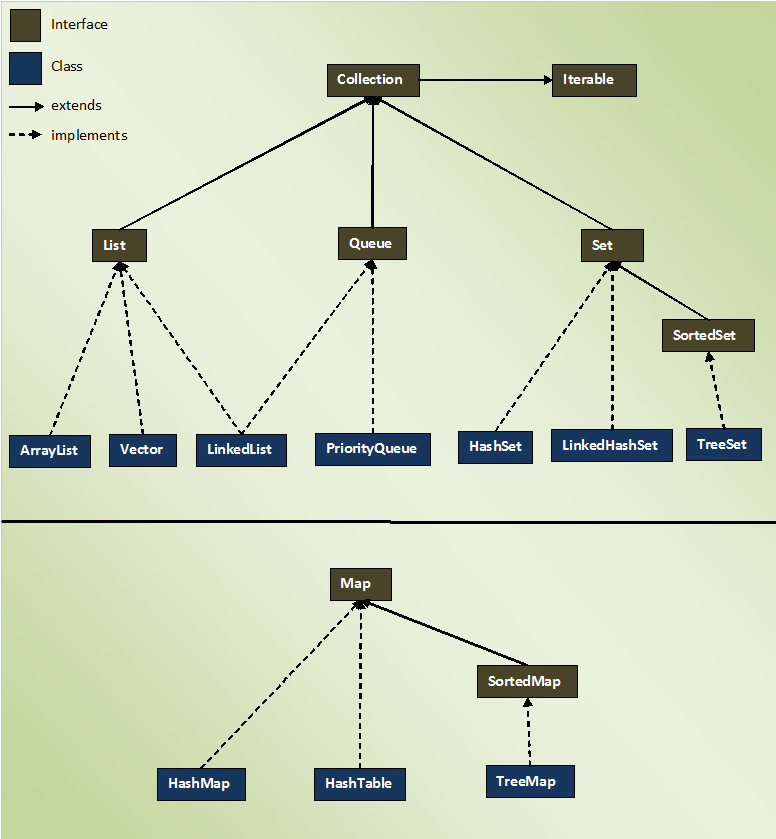
****1) List****  —> It handles sequential list of objects. ****ArrayList****, ****Vector**** and ****LinkedList**** classes implement this interface.

****2) Queue****  —> It handles the special group of objects in which elements are removed only from the head. ****LinkedList**** and ****PriorityQueue**** classes implement this interface.

****3) Set****  —> It handles the group of objects which must contain only unique elements. This interface is implemented by ****HashSet**** and ****LinkedHashSet**** classes and extended by ****SortedSet**** interface which in turn, is implemented by ****TreeSet****.

****4) Map****  —> This is the one interface in Collection Framework which is not inherited from *Collection* interface. It handles the group of objects as Key/Value pairs. It is implemented by ****HashMap**** and ****HashTable**** classes and extended by ****SortedMap**** interface which in turn is implemented by ****TreeMap****.

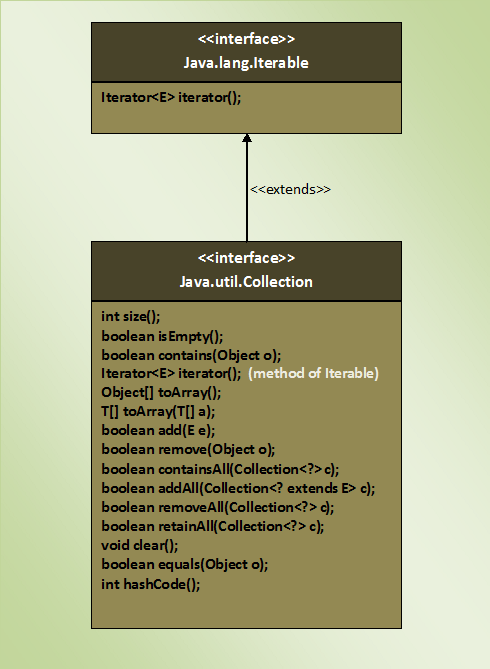
Three of above interfaces (*List*, *Queue* and *Set*) inherit from *Collection* interface. Although, *Map* is included in collection framework it does not inherit from *Collection* interface.



# [Collection Interface](https://javaconceptoftheday.com/collection-framework-collection-interface/)

****Collection interface**** is the root level interface in the collection framework. List, Queue and Set are all sub interfaces of Collection interface. JDK does not provide any direct implementations of this interface. But, JDK provides direct implementations of it’s sub interfaces.

Collection interface extends ****Iterable interface**** which is a member of java.lang package. Iterable interface has only one method called iterator(). It returns an Iterator object, using that object you can iterate over the elements of Collection. Here is the class diagram of Collection interface.



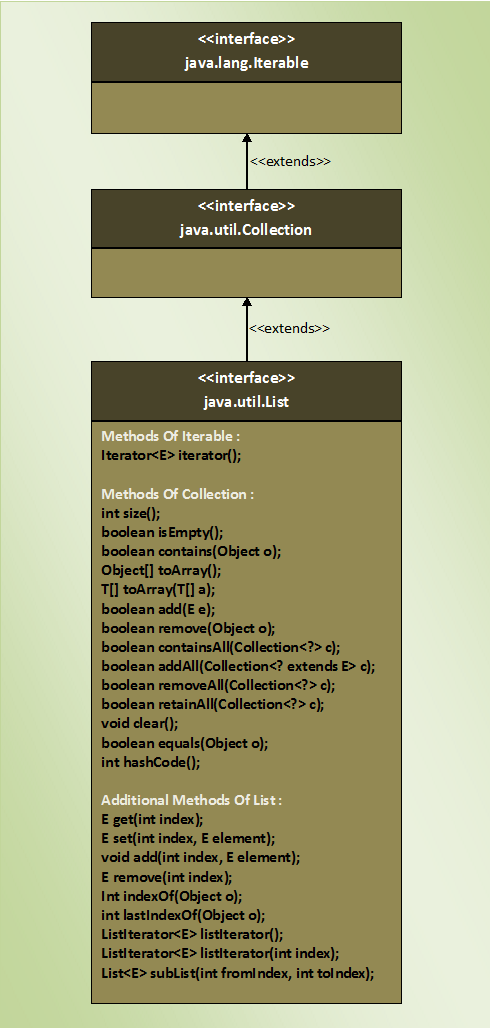
*equals()* and *hashcode()* methods in the *Collection* interface are not the methods of *java.lang.Object* class. Because, interfaces does not inherit from *Object* class. Only classes in java are inherited from *Object* class. Any classes implementing *Collection* interface must provide their own version of *equals()* and *hashcode()* methods or they can retain default version inherited from *Object* class.

# [List Interface](https://javaconceptoftheday.com/collection-framework-list-interface/)

****List**** ****Interface**** represents an ordered or sequential collection of objects. This interface has some methods which can be used to store and manipulate the ordered collection of objects. The classes which implement the List interface are called as ****Lists****. ArrayList, Vector and LinkedList are some examples of lists. You have the control over where to insert an element and from where to remove an element in the list.

Here are some properties of lists.

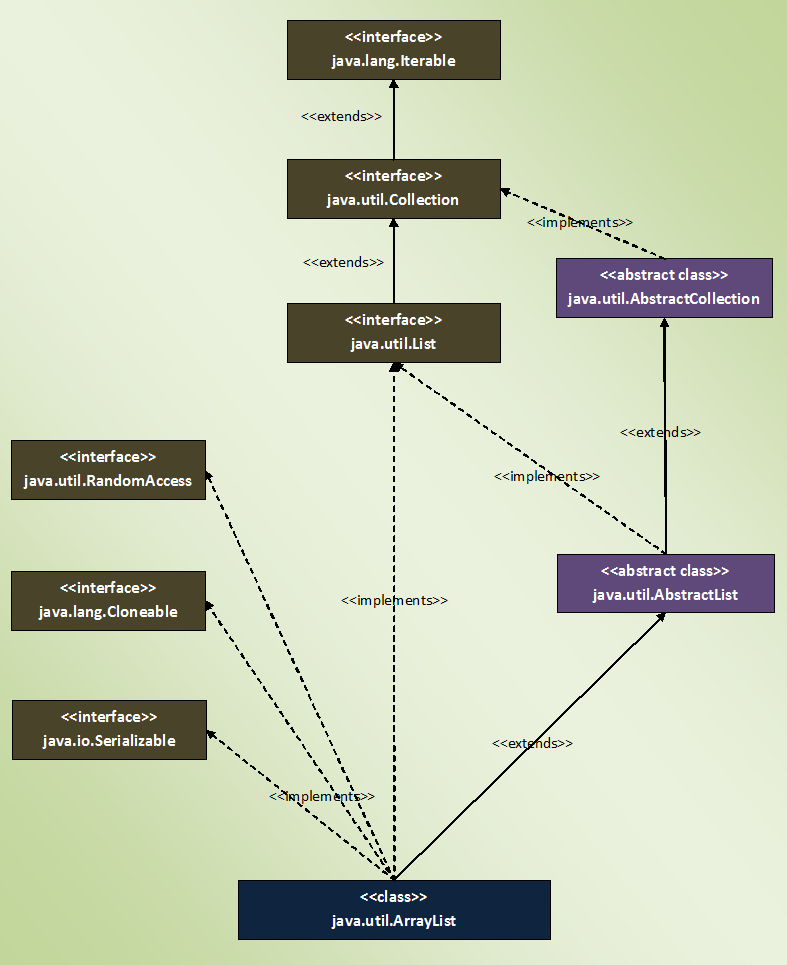
1. Elements of the lists are ordered using Zero based index.
2. You can access the elements of lists using an integer index.
3. Elements can be inserted at a specific position using integer index. Any pre-existing elements at or beyond that position are shifted right.
4. Elements can be removed from a specific position. The elements beyond that position are shifted left.
5. A list may contain duplicate elements.A list may contain multiple null elements.



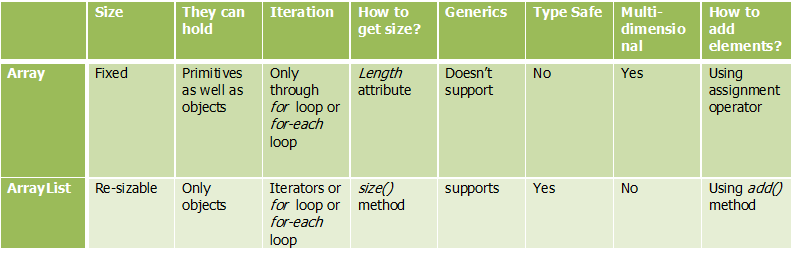
# [The ArrayList Class](https://javaconceptoftheday.com/collection-framework-arraylist-class/)

****ArrayList****, in simple terms, can be defined as **re-sizable array**. ArrayList is same like normal array but it can grow and shrink dynamically to hold any number of elements. **ArrayList is a sequential collection of objects which increases or decreases in size as we add or delete the elements.**

* In ArrayList, elements are positioned according to ****Zero-based index****. That means, elements are inserted from index 0.
* ****Default initial capacity** of an ArrayList is 10**. This capacity increases automatically as we add more elements to arraylist. You can also specify initial capacity of an ArrayList while creating it.
* ArrayList class implements ****List interface**** and extends ****AbstractList****. It also implements 3 marker interfaces – ****RandomAccess****, ****Cloneable**** and ****Serializable****. Here is hierarchy diagram of ArrayList class.
* When you insert an element in the middle of the ArrayList, the elements at the right side of that position are shifted one position right and when you delete an element, they will be shifted one position left. This feature of the ArrayList causes some performance issues as shifting of elements is time consuming if ArrayList has lots of elements.



### Array Vs ArrayList In Java :



# [Advantages Of Using ArrayList Over Arrays](https://javaconceptoftheday.com/advantages-of-using-arraylist-over-arrays/)

what are the drawbacks of arrays:

* Arrays are of fixed length. You can not change the size of the arrays once they are created.
* You can not accommodate an extra element in an array after they are created.
* Memory is allocated to an array during it’s creation only, much before the actual elements are added to it.

some advantages of using ArrayList over arrays:

1. You can define ArrayList as ****re-sizable array****. Size of the ArrayList is not fixed. ArrayList can grow and shrink dynamically.
2. Elements can be inserted at or deleted from a particular position.
3. ArrayList class has many methods to manipulate the stored objects.
4. ArrayList class has methods to perform solo modifications ( add(), remove()… ), bulk modifications ( addAll(), removeAll(), retainAll()… ), searching( indexOf(), lasIndexOf() ) and iterations( iterator() ).
5. If generics are not used, ArrayList can hold any type of objects.
6. You can traverse an ArrayList in both the directions – forward and backward using ListIterator.
7. ArrayList can hold multiple null elements.
8. ArrayList can hold duplicate elements.

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

list.add("ONE");

list.add("TWO");

list.add("THREE");

list.add("FOUR");

ListIterator iterator = list.listIterator();

System.out.println("Elements in forward direction");

while (iterator.hasNext()){

System.out.println(iterator.next());

}

System.out.println("Elements in backward direction");

while (iterator.hasPrevious()){

System.out.println(iterator.previous());

}

## Java ArrayList Programming Questions :

1. ****Explain the different ways of constructing an ArrayList?****ArrayList can be created in 3 ways.
2. ****ArrayList()**** —> It creates an empty ArrayList with initial capacity of 10.
3. ****ArrayList(int initialCapacity)**** —> It creates an empty ArrayList with supplied initial capacity.
4. ****ArrayList(Collection c)**** —> It creates an ArrayList containing the elements of the supplied collection.
5. ****How do you increase the current capacity of an ArrayList? ensureCapacity() method**** is used to increase the current capacity of an ArrayList. However, capacity of an ArrayList is automatically increased when we try to add more elements than the current capacity. To manually increase the current capacity, ensureCapacity() method is used.
6. **How do you decrease the current capacity of an ArrayList to the current size?**

****trimToSize() method**** is used to trim the capacity of arrayList to the current size of ArrayList. Developers use this method to minimize the storage area of an ArrayList.

1. ****How do you find the number of elements present in an ArrayList?****

Using ****size()**** method. size() method returns number of elements present in an ArrayList.

1. ****How do you find out whether the given ArrayList is empty or not? : isEmpty()**** method of ArrayList is used to check whether the given ArrayList is empty or not. This method returns true if an ArrayList contains no elements otherwise returns false.
2. ****How do you check whether the given element is present in an ArrayList or not?** Using **contains() method** of ArrayList, we can examine whether the ArrayList contains the given element or not. This method returns true if ArrayList has that element otherwise returns false.**
3. ****How do you get the position of a particular element in an ArrayList?** We can use **indexOf() and lastIndexOf() methods** to find out the position of a given element in an ArrayList. indexOf() method returns index of first occurrence of a specified element where as lastIndexOf() method returns index of last occurrence of a specified element in an ArrayList. If element is not found, they will return -1.**
4. ****How do you convert an ArrayList to Array?** Using **toArray() method** of ArrayList class. toArray() method returns an array containing all elements of the ArrayList. This method acts as a bridge between normal arrays and collection framework in java.** Object[] array = list.toArray();
5. ****How do you retrieve an element from a particular position of an ArrayList? get() method** returns an element from a specified position of an ArrayList. This method takes index of the element as an argument.** System.out.println(list.get(3));
6. ****How do you replace a particular element in an ArrayList with the given element? set()**** method replaces a particular element in an Arraylist with the given element. This method takes two arguments. One is the index of the element to be replaced and another one is the element to be placed at that position.list.set(1, 000);
7. ****How do you append an element at the end of an ArrayList or insert an element at a particular position? add()**** method appends an element at the end of an ArrayList. list.add("TWO"); list.add(3, "BBB");
8. **remove() method** which takes int type as an argument is used to remove an element from a particular position of an ArrayList.
9. ****remove(Object obj)**** method removes the first occurrence of the specified element ‘****obj****‘. If that element doesn’t exist, ArrayList will be unchanged.
10. ****clear()**** method removes all elements of an ArrayList. ArrayList will be empty after this method is executed.
11. ****How do you retrieve a portion of an ArrayList?**** Using ****subList()**** method of ArrayList, we can retrieve a portion of an ArrayList. subList() method returns a view of a portion of an ArrayList in the given range. The returned subList is backed by original ArrayList. That means any changes made to subList will be reflected in original ArrayList or Vice-Versa.
12. ****How do you insert more than one element at a particular position of an ArrayList?**** addAll() method which takes two arguments, one is index and another one is Collection type, can be used for this requirement.

## What is the difference between ArrayList and Vector Class?

## 1) Thread Safety

This is the main difference between ArrayList and Vector class. ArrayList class is not thread safety where as Vector class is thread safety. Vector class is a synchronized class. Only one thread can enter into Vector object at any moment of time during execution. Where as ArrayList class is not synchronized. Multiple threads can access ArrayList object simultaneously.

## 2) Performance

ArrayList has better performance compared to Vector. It is because, Vector class is synchronized. It makes the threads to wait for object lock to enter into vector object. Where as ArrayList class is not synchronized. Threads need not to wait for object lock to access ArrayList object. This makes ArrayList faster than the Vector class.

## 3) Capacity Increment

Whenever the size of the ArrayList exceeds it’s capacity, the capacity is increased by half of the current capacity. Where as in case of Vector, the capacity is increased by ****Capacity Increment**** passed while creating the Vector object. If Capacity increment is not passed, capacity will be doubled automatically when the size exceeds it’s capacity. In ArrayList, there is no provision to pass Capacity increment while creating it. It’s capacity is automatically increased by half of the current capacity whenever size exceeds capacity.

## 4) Size

You can manually change the current size of the vector. Vector class has a method called ****setSize().**** Using this method, you can change the current size of the vector. If the new size is greater than the current size, new slots will be filled with null elements and if the new size is smaller than the current size, extra elements will be discarded. But in case of ArrayList, you can’t change the current size manually. It doesn’t have methods which alter it’s size. The size of the ArrayList will be changed only when you add or delete it’s elements.

## 5) Traversing The Elements.

ArrayList elements can be traversed using Iterator, ListIterator and using either normal or advanced for loop. But, vector elements can be traversed using Enumeration also along with these methods. Vector class has a method called ****elements()**** which returns Enumeration object containing all elements of the vector. Where as ArrayList does not have such methods.

## 6) Searching The Elements.

In ArrayList, you have to start searching for a particular element from the beginning of an Arralist. But in the Vector, you can start searching for a particular element from a particular position in a vector. This makes the search operation in Vector faster than in ArrayList.

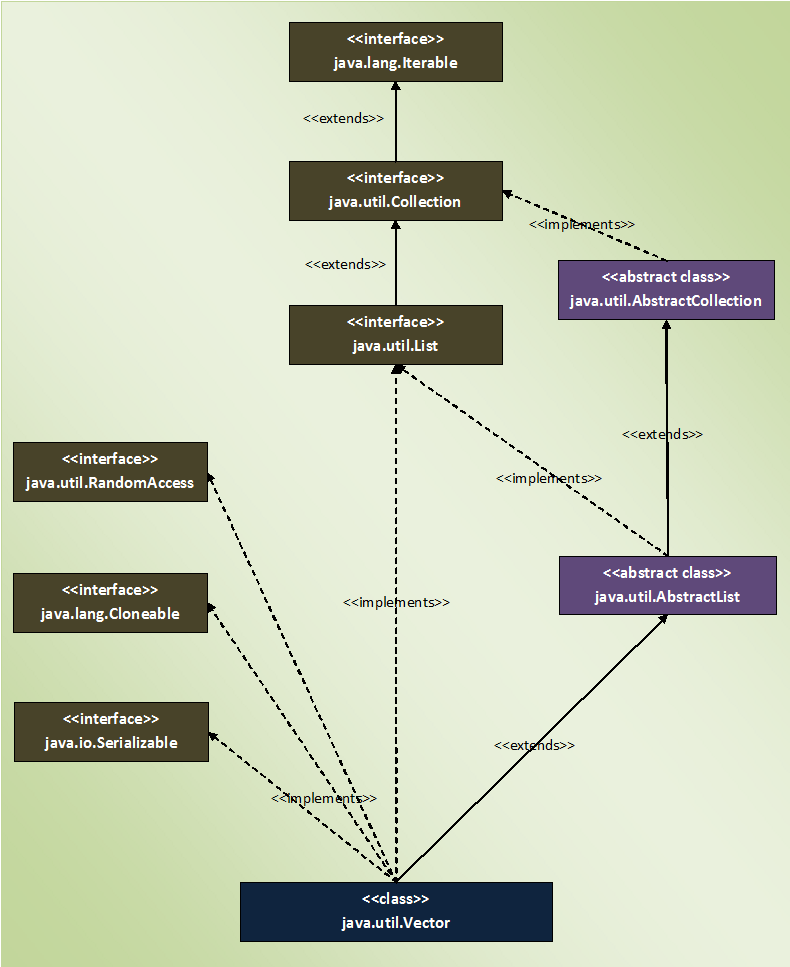
## 7) Legacy Code

Vector class is considered as Legacy code. Because, it exist in Java before the introduction of Collection Framework. Earlier it was not a part of Collections. Later it has been included in Collections. But, the older methods of vector class have been retained as it is.

# [The Vector Class](https://javaconceptoftheday.com/collection-framework-vector-class/)

****The Vector Class**** is also dynamically grow-able and shrink-able collection of objects like an [ArrayList](https://javaconceptoftheday.com/collection-framework-arraylist-class/" \o "Collection Framework – The ArrayList Class" \t "https://javaconceptoftheday.com/collection-framework-vector-class/_blank) class. But, the main difference between ArrayList and Vector is that ****Vector class is synchronized****. That means, only one thread can enter into vector object at any moment of time.

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.



## Properties Of Vector Class :

* **Vector class is preferred over ArrayList class when you are developing a multi threaded application**. But, precautions need to be taken because vector may reduce the performance of your application as it is thread safety and only one thread is allowed to have object lock at any moment of time and remaining threads have to wait until a thread releases the object lock which is held by it. So, it is always recommended that if you don’t need thread safety environment, it is better to use ArrayList class than the Vector class.
* **Capacity Increment** : Capacity increment is an amount by which the capacity of the vector is automatically incremented whenever size of the vector exceeds it’s capacity. You can pass this capacity increment while creating a vector. If you don’t pass, capacity increment will be treated as zero and capacity of the vector will be doubled whenever size exceeds capacity.
* **Unlike an ArrayList, you can set the size of the Vector manually.** If the new size is greater than the current size, the new slots will be filled with null elements. If the new size is smaller than current size, then the extra elements will be discarded.
* You can traverse the vector using **Enumeration**object. Vector class has a method called **elements()** which returns an Enumeration object consisting of all elements of Vector.
* Vector class has **separate methods to retrieve first and last element of vector object**. You will not find these methods in ArrayList class. **firstElement**() retrieves first element and **lastElement**() method retrieves last element of the vector.

# [Why Not To Use Vector Class In Your Code?](https://javaconceptoftheday.com/not-use-vector-class-code/)

1. **You can achieve Thread Safety without Vector.**

If a thread-safe implementation of List interface is required, **we can either use [CopyOnWriteArrayList](https://docs.oracle.com/javase/9/docs/api/java/util/CopyOnWriteArrayList.html" \t "https://www.techiedelight.com/why-vector-class-java-obsolete/_blank) class**, **which is a thread-safe variant of ArrayList or synchronize ArrayList externally using the synchronizedList() method of the Collections class**.

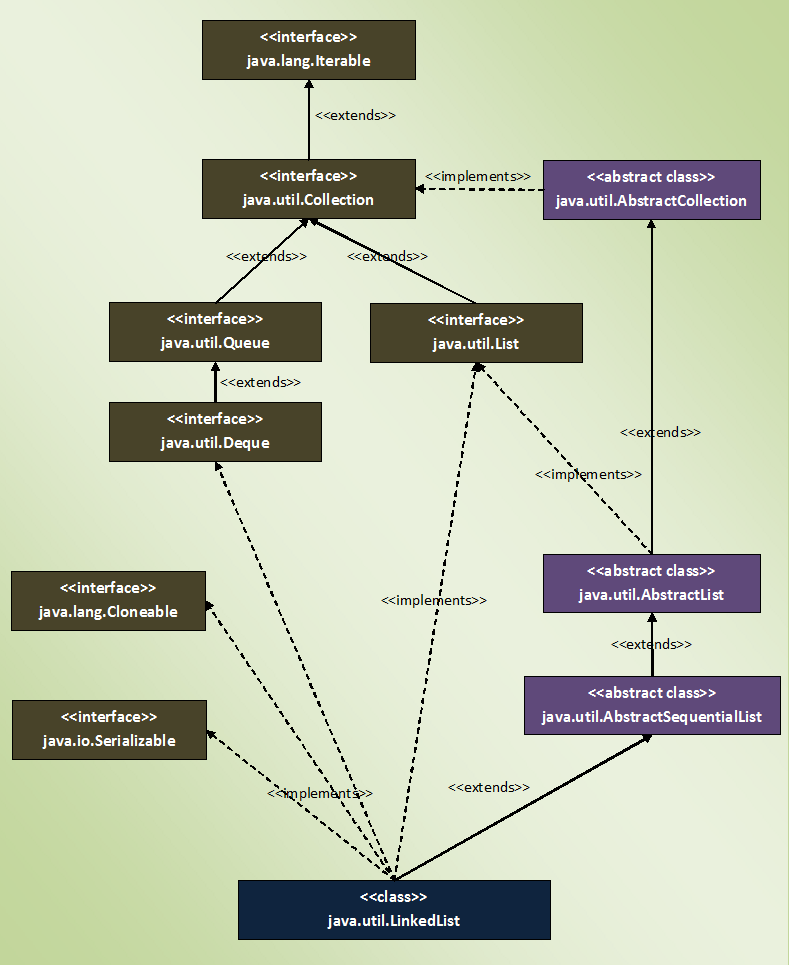
1. **Thread Safeness of Vector class is time consuming**. All methods of Vector class are synchronized. This makes each and every operation on Vector object thread safe. But, it is time consuming. Because, you need to acquire object lock for each operation you want to perform on vector object. Usually, you need set of operations to be synchronized not each and every operation. Isn’t make sense to take the object lock once, perform the operations you want and then release the lock when you are done. Why acquire the lock again and again for each operations?. This is the time consuming process and decreases the performance of your application.
2. **Bad Design** :The major drawback of Vector class is that it is synchronized but not completely thread-safe.

This is because Vector synchronizes on each individual operation and does not synchronize the whole Vector instance itself. This is not desired in real-world applications where whole set of operations needs to be synchronized and not individual operations. **If one thread is iterating over the Vector and another thread modifies the Vector instance structurally, the iterator will throw a ConcurrentModificationException**.

**In other words, two threads can operate on a Vector instance at the same time provided they’re performing different operations.**

1. **Enumeration Vs Iterator:** Vector class has a method which return Enumeration over the elements of Vector object. Although, Enumerations are faster than the Iterator, but it is not backed by the original collection. That means, any changes made to original collection does not reflect in Enumeration object. They ignore the modifications done during iteration. This may cause issues.
2. **Is Vector class poorly designed?** Vector class combines two features – “Re-sizable Array” and “Synchronization“. This makes poor design. Because, if you need just “Re-sizable Array” and you use Vector class for that, you will get “synchronized Resizable Array” not just re-sizable array. This may reduce the performance of your application. Therefore, instead of using Vector class, always use ArrayList class. You will have re-sizable array and whenever you want to make it synchronized, use Collections.SynchronizedList().

# [The LinkedList Class](https://javaconceptoftheday.com/java-collection-framework-linkedlist-class/)

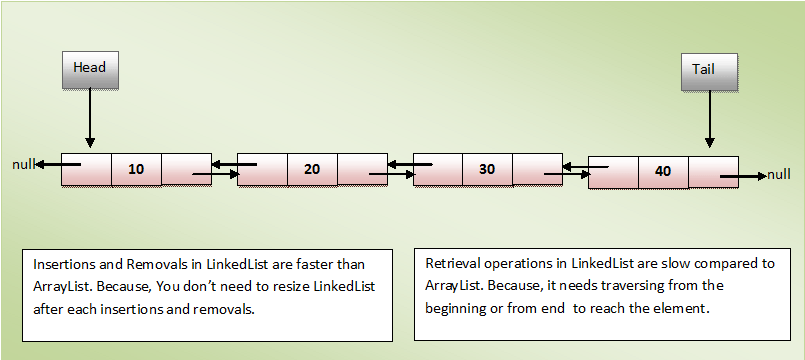


The LinkedList class extends ****AbstractSequentialList**** and implements ****List**** and ****Deque**** interfaces. It also implements 2 marker interfaces – ****Cloneable**** and ****Serializable****. Here is the hierarchy diagram of LinkedList class in Java.

The ****LinkedList**** class in Java is an implementation of doubly linked list which can be used both as a ****List**** as well as ****Queue****. The LinkedList in java can have any type of elements including null and duplicates. Elements can be inserted and can be removed from both the ends and can be retrieved from any arbitrary position.

## Properties Of LinkedList Class In Java:

* Elements in the LinkedList are called as ****Nodes****. Where each node consist of three parts – Reference To Previous Element, Value Of The Element and Reference To Next Element. Below diagram shows how LinkedList looks like.



* Reference To Previous Element of first node and Reference To Next Element of last node are null as there will be no elements before the first node and after the last node.

**You can insert the elements at both the ends and also in the middle of the LinkedList.**

|  |  |  |
| --- | --- | --- |
| Insertion At Head | Insertion In The Middle | Insertion At Tail |
| addFirst(E e) | add(int index, E e) | add(E e) |
| offerFirst(E e) | addAll(int index, Collection c) | addAll(Collection c) |
|  |  | offer(E e) |
|  |  | offerLast(E e) |

* You can remove the elements from the head, from the tail and also from the middle of the LinkedList.

|  |  |  |
| --- | --- | --- |
| Removing from head | Removing from the middle | Removing from the tail |
| poll() | Remove(int index) | pollLast() |
| pollFirst() |  | removeLast() |
| remove() |  |  |
| removeFirst() |  |  |

* You can retrieve the elements form the head, from the middle and from the tail of the LinkedList. Below is the list of retrieval methods.

|  |  |  |
| --- | --- | --- |
| Retrieving from the head | Retrieving from the middle | Retrieving from the tail |
| element() | get(int index) | getLast() |
| getFirst() |  | peekLast() |
| peek() |  |  |
| peekFirst() |  |  |

* Insertion and removal operations in LinkedList are faster than the ArrayList. Because in LinkedList, there is no need to shift the elements after each insertion and removal. only references of next and previous elements need to be changed.
* Retrieval of the elements is very slow in LinkedList as compared to ArrayList. Becaues in LinkedList, you have to traverse from beginning or end (whichever is closer to the element) to reach the element.
* The LinkedList can be used as ****stack****. It has the methods pop() and push() which make it to function as Stack.
* The LinkedList can also be used as ArrayList, Queue, SIngle linked list and doubly linked list.
* LinkedList can have multiple ****null**** elements.
* LinkedList can have ****duplicate**** elements.
* LinkedList class in Java is not of type ****Random Access****. i.e the elements can not be accessed randomly. To access the given element, you have to traverse the LinkedList from beginning or end (whichever is closer to the element) to reach the given element.

## ArrayList Vs LinkedList In Java :

|  |  |
| --- | --- |
| ****ArrayList**** | ****LinkedList**** |
| Structure | ArrayList is an index based data structure where each element is associated with an index. | Elements in the LinkedList are called as nodes, where each node consists of three things – Reference to previous element, Actual value of the element and Reference to next element. |
| Insertion And Removal | Insertions and Removals in the middle of the ArrayList are very slow. Because after each insertion and removal, elements need to be shifted. | Insertions and Removals from any position in the LinkedList are faster than the ArrayList. Because there is no need to shift the elements after every insertion and removal. Only references of previous and next elements are to be changed. |
| Insertion and removal operations in ArrayList are of order O(n). | Insertion and removal in LinkedList are of order O(1). |
| Retrieval(Searching or getting an element) | Retrieval of elements in the ArrayList is faster than the LinkedList . Because all elements in ArrayList are index based. | Retrieval of elements in LinkedList is very slow compared to ArrayList. Because to retrieve an element, you have to traverse from beginning or end (Whichever is closer to that element) to reach that element. |
| Retrieval operation in ArrayList is of order of O(1). | Retrieval operation in LinkedList is of order of O(n). |
| Random Access | ArrayList is of type Random Access. i.e elements can be accessed randomly. | LinkedList is not of type Random Access. i.e elements can not be accessed randomly. you have to traverse from beginning or end to reach a particular element. |
| Usage | ArrayList can not be used as a Stack or Queue. | LinkedList, once defined, can be used as ArrayList, Stack, Queue, Singly Linked List and Doubly Linked List. |
| Memory Occupation | ArrayList requires less memory compared to LinkedList. Because ArrayList holds only actual data and it’s index. | LinkedList requires more memory compared to ArrayList. Because, each node in LinkedList holds data and reference to next and previous elements. |
| When To Use | If your application does more retrieval than the insertions and deletions, then use ArrayList. | If your application does more insertions and deletions than the retrieval, then use LinkedList. |

# [The Queue Interface](https://javaconceptoftheday.com/collection-framework-queue-interface/)

The Queue Interface extends Collection interface. It defines queue data structure which is normally ****First-In-First-Out****.

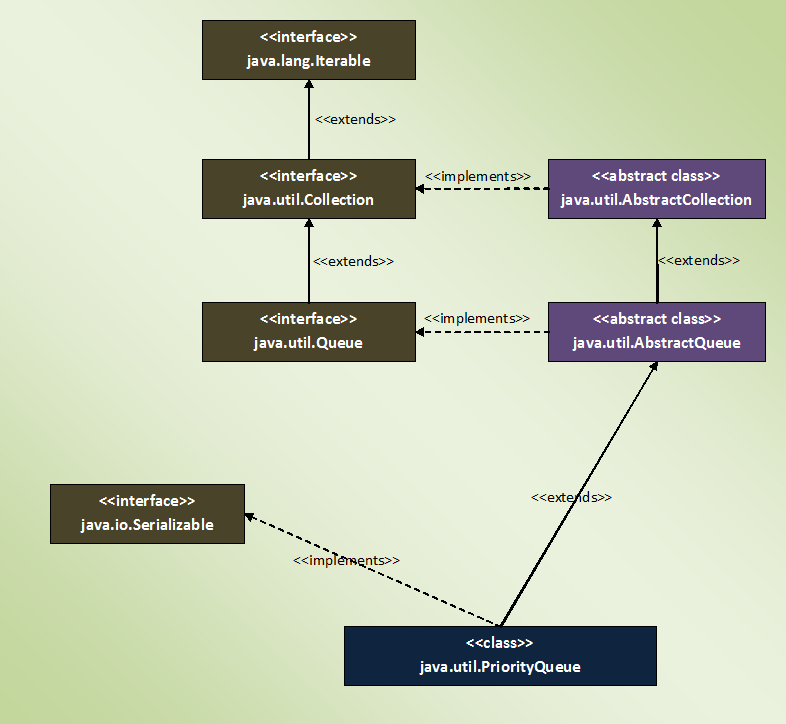
## Properties Of Queue :

* ****Null**** elements are not allowed in the queue. If you try to insert null object into the queue, it throws NullPointerException.
* Queue can have ****duplicate**** elements.
* Unlike a normal list, queue is ****not random access****. i.e you can’t set or insert or get elements at an arbitrary positions.
* In most of cases, elements are inserted at one end called ****tail**** of the queue and elements are removed or retrieved from another end called ****head**** of the queue.
* In the Queue Interface, there are two methods to obtain and remove the elements from the head of the queue. They are ****poll()**** and ****remove()****. The difference between them is, poll() returns null if the queue is empty and remove() throws an exception if the queue is empty.
* There are two methods in the Queue interface to obtain the elements but don’t remove. They are ****peek()**** and ****element()****. peek() returns null if the queue is empty and element() throws an exception if the queue is empty.

## Methods Of Queue Interface:

|  |  |  |
| --- | --- | --- |
| Operation | Throws An Exception If operation is not possible | Returns null or false if operation is not possible |
| Add an element to the queue. | add() | offer() |
| Retrieve an element from the head of the queue. | element() | peek() |
| Retrieve And Remove an element from the head of the queue. | remove() | poll() |

# [The PriorityQueue Class](https://javaconceptoftheday.com/java-collection-framework-priorityqueue-class/)



PriorityQueue class extends ****AbstractQueue**** class which in turn implements ****Queue**** interface. PriorityQueue also implements one marker interface – ****java.io.Serializable**** interface.

## Properties Of PriorityQueue Class :

1. Elements in the PriorityQueue are ordered according to supplied Comparator. **If Comparator is not supplied, elements will be placed in their natural order**.
2. The PriorityQueue is unbounded. That means the capacity of the PriorityQueue increases automatically if the size exceeds capacity. But, how it grows is not specified.
3. The PriorityQueue **can have duplicate elements but can not have null elements.**
4. **All elements of the PriorityQueue must be of Comparable type. Otherwise ClassCastException will be thrown at run time.**
5. The head element of the PriorityQueue is always the least element and tail element is always the largest element according to specified Comparator.
6. **The default initial capacity of PriorityQueue is 11.**
7. You can retrieve the Comparator used to order the elements of the PriorityQueue using comparator() method.
8. PriorityQueue is not a thread safe.

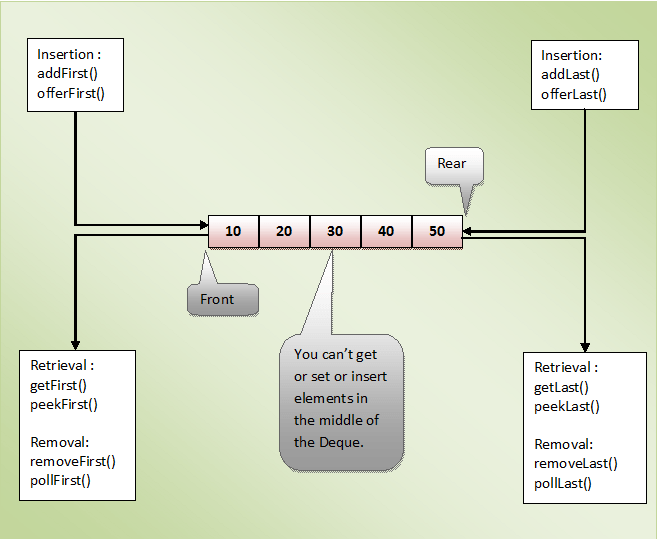
# [The Deque Interface](https://javaconceptoftheday.com/collection-framework-deque-interface/)

The Deque is the short name for “Double Ended Queue“. As the name suggest, Deque is a linear collection of objects which supports insertion and removal of elements from both the ends. The Deque interface defines the methods needed to insert, retrieve and remove the elements from both the ends.

The Deque interface is introduced in Java SE 6. It extends Queue interface

The main advantage of Deque is that you can use it as both ****Queue**** (FIFO) as well as ****Stack**** (LIFO).

|  |  |  |  |
| --- | --- | --- | --- |
| Operation | | Throws an exception if operation fails. | Returns null or false if operation fails. |
| Insertion | Front End | addFirst() | offerFirst() |
| Rear End | addLast() | offerLast() |
| Retrieval | Front End | getFirst() | peekFirst() |
| Rear End | getLast() | peekLast() |
| Retrieval And Removal | Front End | removeFirst() | pollFirst() |
| Rear End | removeLast() | pollLast() |



## Properties Of Deque :

* Unlike Queue, Deque can have ****null**** elements. But, it is recommended not to insert null elements as many methods return null to indicate Deque is empty.
* Deque can have ****duplicate**** elements.
* You can’t set or get or insert the elements at an arbitrary position of Deque. i.e ****Random access**** is not possible with the Deque.
* You can use removeFirstOccurrenec(E e), removeLastOccurrence(E e) and remove(E e) methods to delete the elements from the Deque.

## Deque As Queue :

As Deque interface extends Queue interface, it inherits all methods of Queue interface. So, you can use all those inherited methods to perform Queue operations. Along with them, methods defined in the Deque interface can also be used for Queue operations. Below is the list of Queue methods and their equivalent Deque methods.

|  |  |
| --- | --- |
| Queue Methods | Equivalent Deque Methods |
| add() | addLast() |
| offer() | OfferLast() |
| element() | getFirst() |
| peek() | peekFirst() |
| remove() | removeFirst() |
| poll() | pollFirst() |

## Deque As Stack :

Deque interface has two more methods – ****pop()**** and ****push()****. These two methods make Deque to function as a stack (Last-In-First-Out). Along with these two methods, you can also use addFirst(), peekFirst() and removeFirst() for stack operations. Below is the list of Stack methods and their equivalent methods of Deque.

|  |  |
| --- | --- |
| Stack Methods | Equivalent Deque Methods |
| push() | addFirst() |
| pop() | removeFirst() |
| peek() | peekFirst() |

# [The ArrayDeque Class](https://javaconceptoftheday.com/java-collection-framework-arraydeque-class/)

****The ArrayDeque class****in Javais introduced from JDK 1.6. It is an implementation of ****Deque Interface**** which allows insertion of elements at both the ends. It does not have any restrictions on capacity. It expands automatically as we add more elements. The ArrayDeque class extends ****AbstractCollection**** class and implements ****Deque**** interface. It also implements ****Cloneable**** and ****Serializable**** marker interfaces.

## Properties Of ArrayDeque Class :

* ArrayDeque is a r****esizable-array**** implementation of Deque interface like ArrayList class which is a resizable-array implementation of List interface. But, ****ArrayDeque is not a List.****
* ArrayDeque does not have any capacity limit. It will grow automatically as we add elements.
* Default initial capacity of ArrayDeque is ****16****. It will increase at a power of 2 (24, 25, 26 and so on) when size exceeds capacity.
* ArrayDeque can be used as a ****stack**** (LIFO) as well as a ****queue****(FIFO). ArrayDeque is faster than the Stack class when used as a stack and faster than the LinkedList class when used as a queue.
* Performance of ArrayDeque is sometimes considered as the best among the collection framework. It gives performance of ****O(1)**** for insertion, removal and retrieval operations. ArrayDeque class is recommended instead of Stack class (when you want stack data structure) and instead of LinkedList class (when you want queue data structure).
* You can’t perform ****indexed operations**** on ArrayDeque. ArrayDeque doesn’t have the methods to support those operations.
* ArrayDeque is not a thread safe.

# [The Set Interface](https://javaconceptoftheday.com/collection-framework-set-interface/)

The Set interface defines a set. The ****set**** is a linear collection of objects with no duplicates. Duplicate elements are not allowed in a set. The ****Set interface**** extends Collection interface. Set interface does not have it’s own methods. All it’s methods are inherited from Collection interface. The only change that has been made to Set interface is that add() method will return false if you try to insert an element which is already present in the set.

## Properties Of Set :

* Set contains only unique elements. It does not allow ****duplicates****.
* Set can contain only one ****null**** element.
* ****Random access**** of elements is not possible.
* ****Order of elements**** in a set is implementation dependent. ****HashSet**** elements are ordered on hash code of elements. ****TreeSet**** elements are ordered according to supplied Comparator (If no Comparator is supplied, elements will be placed in ascending order) and ****LinkedHashSet**** maintains insertion order.
* Set interface contains only methods inherited from Collection interface. It does not have it’s own methods. But, applies restriction on methods so that duplicate elements are always avoided.
* One more good thing about Set interface is that the ****stronger contract**** between equals() and hashCode() methods. According to this contract, you can compare two set instances of different implementation types (HashSet, TreeSet and LinkedHashSet).
* Two set instances, irrespective of their implementation types, are said to be equal if they contain same elements.

# [The HashSet Class](https://javaconceptoftheday.com/java-collection-framework-hashset-class/)

****The HashSet class**** in Java is an implementation of ****Set interface****. HashSet is a collection of objects which contains only unique elements. Duplicates are not allowed in HashSet. HashSet gives constant time performance for insertion, removal and retrieval operations. It allows only one null element.

HashSet class extends ****AbstractSet class**** and implements ****Set interface****. It also implements ****Cloneable**** and ****Serializable**** marker interfaces. Below is the hierarchy diagram of HashSet class.

## Properties Of HashSet Class In Java :

1. The HashSet internally uses ****HashMap**** to store the objects. The elements you insert in HashSet will be stored as keys of that HashMap object and their values will be a constant called ****PRESENT****. This constant is defined as ****private static final Object PRESENT = new Object()****in the source code of HashSet class.
2. HashSet does not allow ****duplicate**** elements. If you try to insert a duplicate element, older element will be overwritten.
3. HashSet can have maximum one ****null**** element.
4. HashSet doesn’t maintain any order. The order of the elements will be largely unpredictable. And it also doesn’t guarantee that order will remain constant over time.
5. HashSet offers constant time performance for insertion, removal and retrieval operations.HashSet class is not synchronized.
6. If you want synchronized HashSet, use ****Collections.synchronizedSet()**** method.

## How HashSet Works Internally In Java?

Whenever you insert an element into HashSet using ****add()**** method, it actually creates an entry in the internally backing HashMap object with element you have specified as it’s key and constant called “****PRESENT****” as it’s value. This “PRESENT” is defined in the HashSet class as below.

|  |  |
| --- | --- |
| 1  2 | // Dummy value to associate with an Object in the backing Map  **private** **static** **final** Object PRESENT = **new** Object(); |

Let’s have a look at add() method of HashSet class.

**public** **boolean** add(E e)

{

**return** map.put(e, PRESENT)==**null**;

}

You can notice that, add() method of HashSet class internally calls put() method of backing HashMap object by passing the element you have specified as a key and constant “PRESENT” as it’s value.

****remove()**** method also works in the same manner.

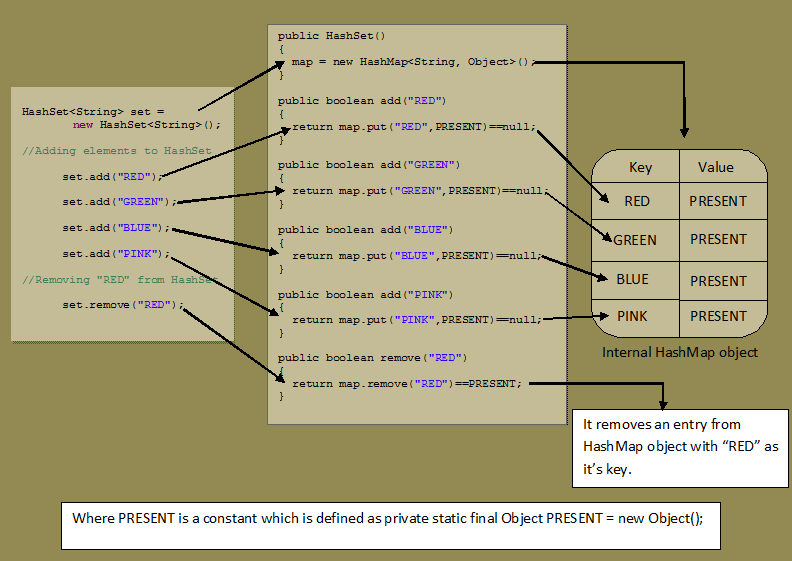
**public** **boolean** remove(Object o)

{

**return** map.remove(o)==PRESENT;

}

In the same manner, all methods of HashSet class process internally backing HashMap object to get the desired result.



# [The LinkedHashSet Class](https://javaconceptoftheday.com/java-collection-framework-linkedhashset-class/)

The ****LinkedHashSet**** in java is an ordered version of ****HashSet**** which internally maintains one ****doubly linked list**** running through it’s elements. This doubly linked list is responsible for maintaining the insertion order of the elements. Unlike HashSet which maintains no order, LinkedHashSet maintains ****insertion order**** of elements. i.e elements are placed in the order they are inserted. LinkedHashSet is recommended over HashSet if you want a unique collection of objects in an insertion order.

The ****LinkedHashSet class**** extends ****HashSet class**** and implements ****Set interface****. It also implements ****Cloneable**** and ****Serializable**** marker interfaces.

## Properties Of LinkedHashSet Class In Java:

* LinkedHashSet internally uses ****LinkedHashMap**** to store it’s elements just like HashSet which internally uses HashMap to store it’s elements.
* LinkedHashSet maintains ****insertion order****. This is the main difference between LinkedHashSet and HashSet.
* LinkedhashSet also gives ****constant time performance**** for insertion, removal and retrieval operations. The performance of LinkedHashSet is slightly less than the Hashset as it has to maintain doubly linked list internally to order it’s elements.
* Iterator returned by LinkedHashSet is ****fail-fast****. i.e if the LinkedHashSet is modified at any time after the Iterator is created, it throws ConcurrentModificationException.

LinkedHashSet<String> set = **new** LinkedHashSet<String>();

        //Adding elements to LinkedHashSet

        set.add("JDBC");

        set.add("HIBERNATE");

        //Getting Iterator object

        Iterator<String> it = set.iterator();

        //Modifying the LinkedHashSet after the Iterator is created

        set.add("JSF");

**while** (it.hasNext()){

            //This statement will throw ConcurrentModificationException

            System.out.println(it.next());

        }

* LinkedHashSet doesn’t allow ****duplicate**** elements and allows only one ****null**** element.

LinkedHashSet<String> set = **new** LinkedHashSet<String>();

set.add("BLUE");

        set.add("GREEN");

        set.add("BLUE");     //duplicate element

        set.add("BLACK");

        set.add("WHITE");

        //Adding two null elements

        set.add(**null**); set.add(**null**);

        //printing the elements of LinkedHashSet

        System.out.println(set);     //Output : [BLUE, RED, GREEN, BLACK, WHITE, null]

        //You can notice that LinkedHashSet doesn't allow duplicates and allows only one null element

* LinkedHashSet is not ****synchronized****. To get the synchronized LinkedHashSet, use ****Collections.synchronizedSet()**** method.

# [How LinkedHashSet Works Internally In Java?](https://javaconceptoftheday.com/how-linkedhashset-works-internally-in-java/)

LinkedHashSet is an ****extended version**** of HashSet. HashSet doesn’t follow any order where as LinkedHashSet maintains ****insertion order****. HashSet uses ****HashMap object**** internally to store it’s elements where as LinkedHashSet uses ****LinkedHashMap object**** internally to store and process it’s elements.

HashSet(**int** initialCapacity, **float** loadFactor, **boolean** dummy)

{

        map = **new** LinkedHashMap<>(initialCapacity, loadFactor);

}

How LinkedHashSet Maintains Insertion Order?

LinkedHashSet uses LinkedHashMap object to store it’s elements. The elements you insert in the LinkedHashSet are stored as keys of this LinkedHashMap object. Each ****key, value pair**** in the LinkedHashMap are instances of it’s static inner class called ****Entry<K, V>****. This Entry<K, V> class extends ****HashMap.Entry**** class. The insertion order of elements into LinkedHashMap are maintained by adding two new fields to this class. They are ****before**** and ****after****. These two fields hold the references to previous and next elements. These two fields make LinkedHashMap to function as a doubly linked list.

**private** **static** **class** Entry<K,V> **extends** HashMap.Entry<K,V>

{

        // These fields comprise the doubly linked list used for iteration.

        Entry<K,V> before, after;

        Entry(**int** hash, K key, V value, HashMap.Entry<K,V> next) {

**super**(hash, key, value, next);

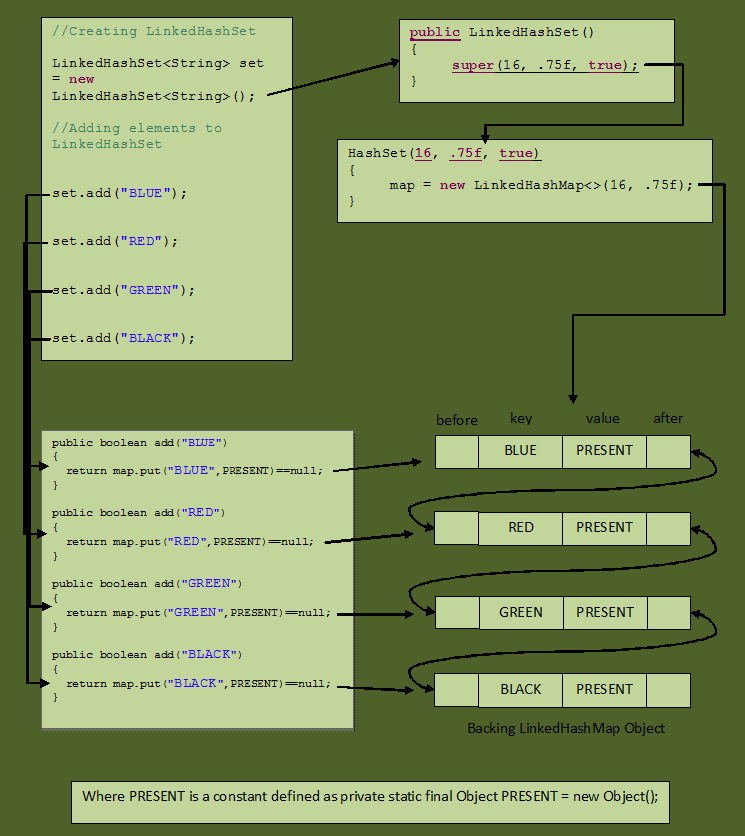
        }

}

The first two fields of above inner class of LinkedHashMap – ****before**** and ****after**** are responsible for maintaining the insertion order of the LinkedHashSet. The header field of LinkedHashMap stores the head of this doubly linked list.

**private** **transient** Entry<K,V> header;     //Stores the head of the doubly linked list

As LinkedHashSet maintains doubly linked list (along with HashMap), the performance of LinkedHashSet is slightly slower than the HashSet. But, LinkedHashSet will be very useful when you need a collection of elements placed in the order they have inserted. We will see one such example of LinkedHashSet in this article.



# [The SortedSet Interface](https://javaconceptoftheday.com/collection-framework-sortedset-interface/)

****The SortedSet interface**** extends Set interface. SortedSet is a set in which elements are placed according to supplied comparator. This Comparator is supplied while creating a SortedSet. If you don’t supply comparator, elements will be placed in ascending order.

## Properties Of SortedSet Interface :

* SortedSet can not have ****null**** elements. If you try to insert null element, it gives NullPointerException at run time.
* As SortedSet is a set, ****duplicate**** elements are not allowed.
* SortedSet elements are sorted according to supplied ****Comparator****. If you don’t mention any Comparator while creating a SortedSet, elements will be placed in ascending order.
* Inserted elements must be of ****Comparable**** type and they must be mutually Comparable.
* You can retrieve first element and last elements of the SortedSet. You can’t access SortedSet elements randomly. i.e ****Random access**** is denied.
* SortedSets returned by ****headSet()****, ****tailSet()**** and ****subSet()**** methods are just views of the original set. So, changes in the returned set are reflected in the original set and vice versa.

# [The NavigableSet Interface](https://javaconceptoftheday.com/collection-framework-navigableset-interface/)

The ****NavigableSet**** is a SortedSet with navigation facilities. The ****NavigableSet interface**** provides many methods through them you can easily find closest matches of any given element. It has the methods to find out less than, less than or equal to, greater than and greater than or equal of any element in a SortedSet.

## Properties Of NavigableSet Interface :

* NavaigableSet can’t have null elements.
* NavigableSet doesn’t support duplicate elements.
* NavigableSet can be traversed and accessed in either ascending or descending order.
* Methods subSet(), headSet() and tailSet() differ from SortedSet interface in taking additional arguments describing whether upper bound and lower bound are inclusive or exclusive.

# [The TreeSet Class](https://javaconceptoftheday.com/java-collection-framework-treeset-class/)

****The TreeSet class**** in java is a direct implementation of ****NavigableSet**** interface which in turn extends ****SortedSet**** interface (which in turn extends Set interface).

We have seen other two implementations of Set interface – [HashSet](https://javaconceptoftheday.com/java-collection-framework-hashset-class/" \o "Java Collection Framework – The HashSet Class" \t "https://javaconceptoftheday.com/java-collection-framework-treeset-class/_blank) and [LinkedHashSet](https://javaconceptoftheday.com/java-collection-framework-linkedhashset-class/" \o "Java Collection Framework – The LinkedHashSet Class" \t "https://javaconceptoftheday.com/java-collection-framework-treeset-class/_blank). HashSet doesn’t maintain any order where as LinkedHashSet maintains insertion order. The main difference between these two implementations and Treeset is, elements in TreeSet are ****sorted**** according to supplied ****Comparator****. You need to supply this Comparator while creating a TreeSet itself. If you don’t pass any Comparator while creating a TreeSet, elements will be placed in their ****natural ascending order****.

## Properties Of TreeSet Class In Java :

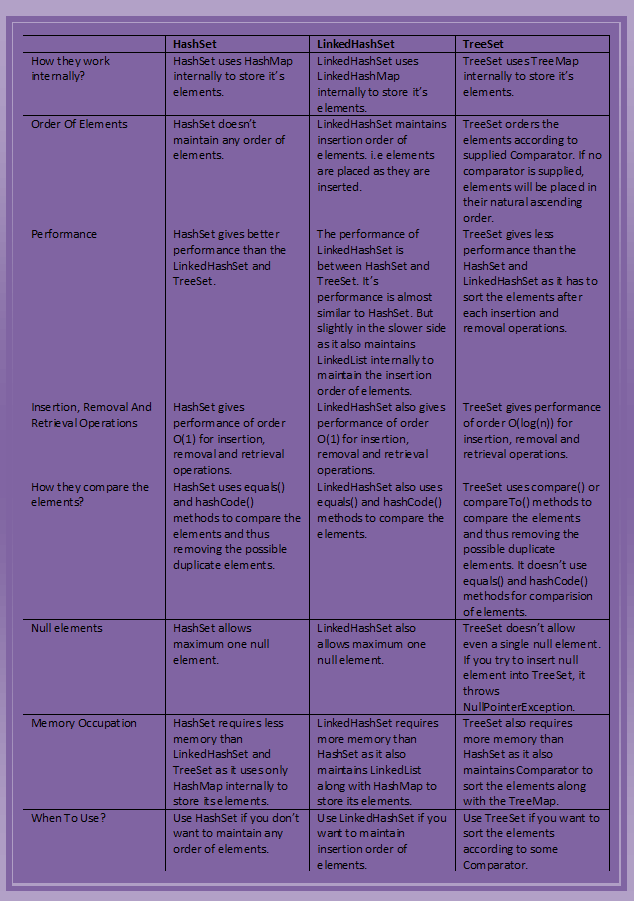
* The elements in TreeSet are sorted according to specified ****Comparator****. If no Comparator is specified, elements will be placed according to their natural ascending order.
* Elements inserted in the TreeSet must be of ****Comparable**** type and elements must be mutually comparable. If the elements are not mutually comparable, you will get ****ClassCastException**** at run time.
* TreeSet does not allow even a single ****null**** element.

set.add(**null**);    //It will throw NullPointerException

* TreeSet is not ****synchronized****. To get a synchronized TreeSet, use ****Collections.synchronizedSortedSet()**** method.
* TreeSet gives performance of order ****log(n)**** for insertion, removal and retrieval operations.
* Iterator returned by TreeSet is of ****fail-fast**** nature. That means, If TreeSet is modified after the creation of Iterator object, you will get ****ConcurrentModificationException****.
* TreeSet internally uses ****TreeMap**** to store it’s elements just like HashSet and LinkedHashSet which use HashMap and LinkedHashMap respectively to store their elements.

## Similarities Between HashSet, LinkedHashSet and TreeSet In Java :

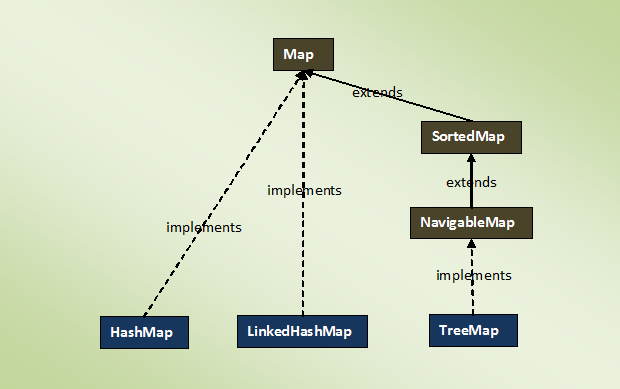
* All three doesn’t allow duplicate elements.
* All three are not synchronized.
* All three are Cloneable and Serializable.
* Iterator returned by all three is fail-fast in nature. i.e You will get ConcurrentModificationException if they are modified after the creation of Iterator object.



# [The Map Interface](https://javaconceptoftheday.com/java-collection-framework-map-interface/)

The ****Map interface**** in java is one of the four top level interfaces of Java Collection Framework along with [List](https://javaconceptoftheday.com/collection-framework-list-interface/" \o "Collection Framework – List Interface" \t "https://javaconceptoftheday.com/java-collection-framework-map-interface/_blank), [Set](https://javaconceptoftheday.com/collection-framework-set-interface/" \o "Collection Framework – The Set Interface" \t "https://javaconceptoftheday.com/java-collection-framework-map-interface/_blank) and [Queue](https://javaconceptoftheday.com/collection-framework-queue-interface/" \o "Collection Framework – The Queue Interface" \t "https://javaconceptoftheday.com/java-collection-framework-map-interface/_blank) interfaces. But, unlike others, it doesn’t inherit from [Collection](https://javaconceptoftheday.com/collection-framework-collection-interface/" \o "Collection Framework – Collection Interface" \t "https://javaconceptoftheday.com/java-collection-framework-map-interface/_blank) interface. Instead it starts it’s own interface hierarchy for maintaining the ****key-value associations****. Map is an object of key-value pairs where each key is associated with a value. This interface is the replacement for ‘****Dictionary****‘ class which is an abstract class introduced in JDK 1.0.

Map Interface was introduced in JDK 1.2.



## Properties Of Map Interface In Java :

1) Map interface is a part of Java Collection Framework, but it doesn’t inherit ****Collection Interface****.

2) Map interface stores the data as a ****key-value pairs**** where each key is associated with a value.

3) A map can not have duplicate ****keys**** but can have duplicate ****values****.

4) Each key ****at most**** must be associated with one value.

5) Each key-value pairs of the map are stored as ****Map.Entry**** objects. Map.Entry is an inner interface of Map interface.

6) The common implementations of Map interface are ****HashMap****, ****LinkedHashMap**** and ****TreeMap****.

7) Order of elements in map is implementation dependent. ****HashMap**** doesn’t maintain any order of elements. ****LinkedHashMap**** maintains ****insertion order**** of elements. Where as ****TreeMap**** places the elements according to supplied ****Comparator****.

8) The Map interface provides three methods, which allows map’s contents to be viewed as a ****set of keys****(keySet() method), ****collection of values****(values() method), or ****set of key-value mappings****(entrySet() method).

## Methods Of Map Interface In Java :

|  |  |  |
| --- | --- | --- |
| SL NO. | Methods | Descriptions |
| 1 | int size() | Returns number of key-value pairs in this map. |
| 2 | boolean isEmpty() | Checks whether this map is empty or not. |
| 3 | boolean containsKey(Object key) | Returns true if this map contains a mapping for the specified key. |
| 4 | boolean containsValue(Object value) | Returns true if this map contains one or more keys associated with the specified value. |
| 5 | V get(Object key) | Returns value associated with the specified key. |
| 6 | V put(K key, V value) | Adds the specified key-value pair to this map. If the specified key already exist in the map, old value will be replaced by the specified value. |
| 7 | V remove(Object key) | Removes the specified key along with it’s value from this map. |
| 8 | void putAll(Map<? extends K, ? extends V> m) | Copies all key-value pairs from the specified map to this map. |
| 9 | void clear() | Removes all mappings from this map. |
| 10 | Set<K> keySet() | Returns a set containing all keys of this map. The returned set is backed by actual map. So, changes made to the map are reflected in the set and vice-versa. |
| 11 | Collection<V> values() | Returns a collection of values of this map. The returned collection is backed by actual map. So, any changes made to the map is reflected in collection and vice-versa. |
| 12 | Set<Map.Entry<K, V>> entrySet() | Returns set view of the mappings contained in this map. |
| 13 | boolean equals(Object o) | Compares the specified object with this map. |
| 14 | int hashCode() | Returns hashcode value of this map. |

# [HashMap In Java With Example](https://javaconceptoftheday.com/hashmap-in-java-with-example/)

The *java.util.HashMap* is a popular implementation of *Map* interface which holds the data as key-value pairs. *HashMap* extends *AbstractMap* class and implements *Cloneable* and *Serializable* interfaces.

### Properties Of HashMap In Java :

1. *HashMap* holds the data in the form of key-value pairs where each key is associated with one value.
2. *HashMap* **doesn’t allow duplicate keys. But it can have duplicate values**.
3. *HashMap* **can have multiple null values and only one null key**.
4. *HashMap***is not synchronized. To get the synchronized *HashMap*, use *Collections.synchronizedMap()* method**.
5. *HashMap* maintains no order.
6. *HashMap* gives constant time performance for the operations like *get()* and *put()* methods.
7. **Default initial capacity of *HashMap* is 16**.

### Important Methods Of HashMap In Java :

1. public V put(K key, V value): This method inserts specified key-value mapping in the map. If map already has a mapping for the specified key, then it rewrites that value with new value.

2) public void putAll(Map m) : This method copies all of the mappings of the map *m* to this map.

3) public V get(Object key) : This method returns the value associated with a specified key.

4) public int size() :This method returns the number of key-value pairs in this map.

5) public boolean isEmpty() :This method checks whether this map is empty or not.

6) public boolean containsKey(Object key) :This method checks whether this map contains the mapping for the specified key.

7) public boolean containsValue(Object value) :This method checks whether this map has one or more keys mapping to the specified value.

8) public V remove(Object key):This method removes the mapping for the specified key.

9) public void clear() :This method removes all the mappings from this map.

10) public Set<K> keySet() : This method returns the Set view of the keys in the map.

11) public Collection<V> values() : This method returns Collection view of the values in the map.

12) public Set<Map.Entry<K, V>> entrySet() : This method returns the Set view of all the mappings in this map.

13) public V putIfAbsent(K key, V value) : This method maps the given value with specified key if this key is currently not associated with a value or mapped to a null.

14) public boolean remove(Object key, Object value) : This method removes the entry for the specified key if this key is currently mapped to a specified value.

15) public boolean replace(K key, V oldValue, V newValue) : This method replaces the oldValue of the specified key with newValue if the key is currently mapped to oldValue.

16) public V replace(K key, V value) : This method replaces the current value of the specified key with new value.

# [How HashMap Works Internally In Java?](https://javaconceptoftheday.com/how-hashmap-works-internally-in-java/)

If your application demands faster insertion and faster retrieval then *HashMap* is the ultimate choice. While selecting the data structure, you must keep two things in your mind. First one is that the data structure must give better performance while inserting the new elements and second one is that it should give even more better performance while searching for an element. Because insertion and retrieval are two operations which you perform very frequently in your applications. These things will matter even more when you are handling the big data. *HashMap* is the most sought after data structure when you are handling the big data with more preference to insertion and retrieval operations.

*HashMap* is the most used data structure in java because it gives almost constant time performance of O(1) for put and get operations irrespective of how big is the data. As you already know, *HashMap* stores the data in the form of key-value pairs. In this post, we will see how HashMap works internally in java and how it stores the elements to give O(1) performance for put and get operations.

### HashMap Internal Structure :

*HashMap* stores the data in the form of key-value pairs. Each key-value pair is stored in an object of *Entry<K, V>* class.

*Entry<K, V>* class is the static inner class of *HashMap* which is defined like below.

**static** **class** Entry<K,V> **implements** Map.Entry<K,V>

{

**final** K key;

        V value;

        Entry<K,V> next;

**int** hash;

        //Some methods are defined here

}

As you see, this inner class has four fields. *key*, *value*, *next* and *hash*.

****key**** : It stores the key of an element and its final.

****value**** : It holds the value of an element.

****next**** : It holds the pointer to next key-value pair. ***This attribute makes the key-value pairs stored as a linked list.***

****hash**** : It holds the hashcode of the key.

These *Entry* objects are stored in an array called *table[]*. This array is initially of size 16. It is defined like below.

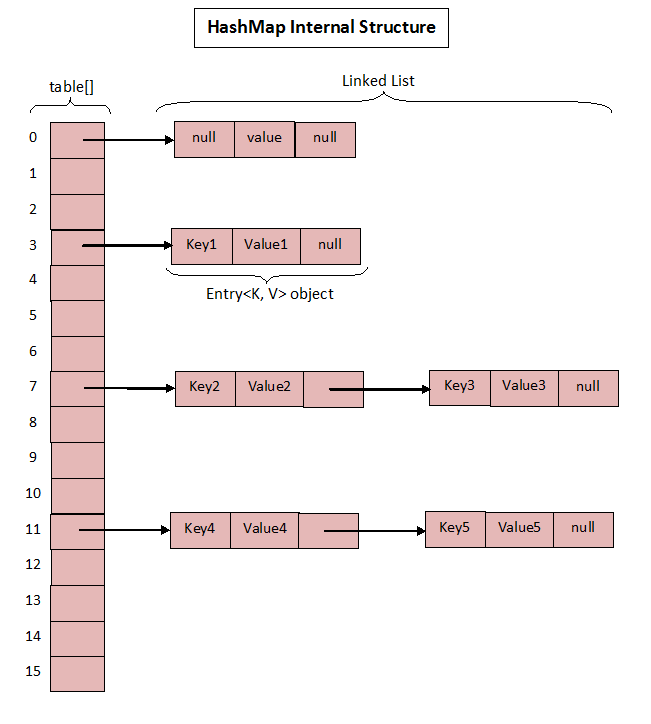
/\*\*

     \* The table, resized as necessary. Length MUST Always be a power of two.

     \*/

**transient** Entry<K,V>[] table;

To summarize the whole *HashMap* structure, each key-value pair is stored in an object of *Entry<K, V>* class. This class has an attribute called *next* which holds the pointer to next key-value pair. This makes the key-value pairs stored as a linked list. All these *Entry<K, V>* objects are stored in an array called *table[]*. The below image best describes the *HashMap* structure.



The above image roughly shows how the *HashMap* stores its elements. Internally it uses an array of *Entry<K, V>* class called *table[]* to store the key-value pairs. But how *HashMap* allocates slot in *table[]* array to each of its key-value pair is very interesting. It doesn’t inserts the objects as you put them into *HashMap* i.e first element at index 0, second element at index 1 and so on. Instead it uses the hashcode of the key to decide the index for a particular key-value pair. It is called ***Hashing***.

### What Is Hashing?

The whole *HashMap* data structure is based on the principle of ****Hashing****. Hashing is nothing but the function or algorithm or method which when applied on any object/variable returns an unique integer value representing that object/variable. This unique integer value is called ***hash code***. Hash function or simply hash said to be the best if it returns the same hash code each time it is called on the same object. Two objects can have same hash code.

Whenever you insert new key-value pair using *put()* method, *HashMap* blindly doesn’t allocate slot in the *table[]* array. Instead it calls **hash function** on the key. *HashMap* has its own hash function to calculate the hash code of the key. This function is implemented so that it overcomes poorly implemented *hashCode()* methods.

***hash()***.

/\*\*

     \* Retrieve object hash code and applies a supplemental hash function to the

     \* result hash, which defends against poor quality hash functions.  This is

     \* critical because HashMap uses power-of-two length hash tables, that

     \* otherwise encounter collisions for hashCodes that do not differ

     \* in lower bits. Note: Null keys always map to hash 0, thus index 0.

     \*/

**final** **int** hash(Object k) {

**int** h = 0;

**if** (useAltHashing) {

**if** (k **instanceof** String) {

**return** sun.misc.Hashing.stringHash32((String) k);

            }

            h = hashSeed;

        }

        h ^= k.hashCode();

        // This function ensures that hashCodes that differ only by

        // constant multiples at each bit position have a bounded

        // number of collisions (approximately 8 at default load factor).

        h ^= (h >>> 20) ^ (h >>> 12);

**return** h ^ (h >>> 7) ^ (h >>> 4);

}

After calculating the hash code of the key, it calls ***indexFor()***method by passing the hash code of the key and length of the *table[]* array. This method returns the index in the *table[]* array for that particular key-value pair.

/\*\*

     \* Returns index for hash code h.

     \*/

**static** **int** indexFor(**int** h, **int** length) {

**return** h & (length-1);

    }

### How put() method works?

/\*\*

     \* Associates the specified value with the specified key in this map.

     \* If the map previously contained a mapping for the key, the old

     \* value is replaced.

     \*

     \* @param key key with which the specified value is to be associated

     \* @param value value to be associated with the specified key

     \* @return the previous value associated with <tt>key</tt>, or

     \*         <tt>null</tt> if there was no mapping for <tt>key</tt>.

     \*         (A <tt>null</tt> return can also indicate that the map

     \*         previously associated <tt>null</tt> with <tt>key</tt>.)

     \*/

**public** V put(K key, V value) {

**if** (key == **null**)

**return** putForNullKey(value);

**int** hash = hash(key);

**int** i = indexFor(hash, table.length);

**for** (Entry<K,V> e = table[i]; e != **null**; e = e.next) {

            Object k;

**if** (e.hash == hash && ((k = e.key) == key || key.equals(k))) {

                V oldValue = e.value;

                e.value = value;

                e.recordAccess(**this**);

**return** oldValue;

            }

        }

        modCount++;

        addEntry(hash, key, value, i);

**return** **null**;

    }

Let’s see how this code works step by step.

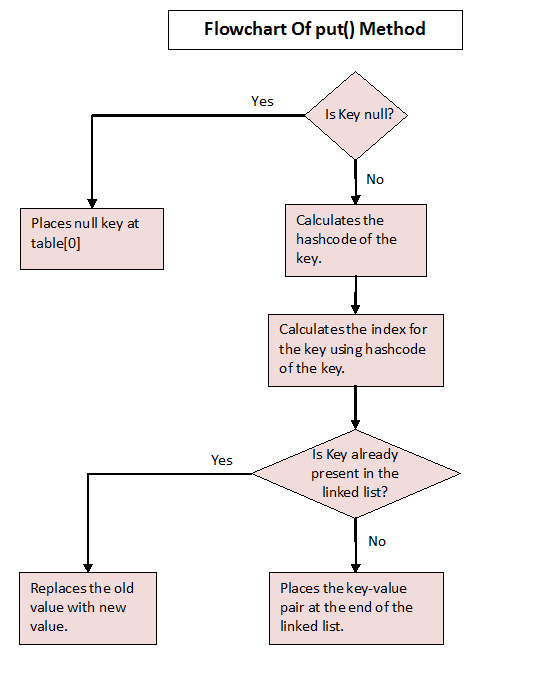
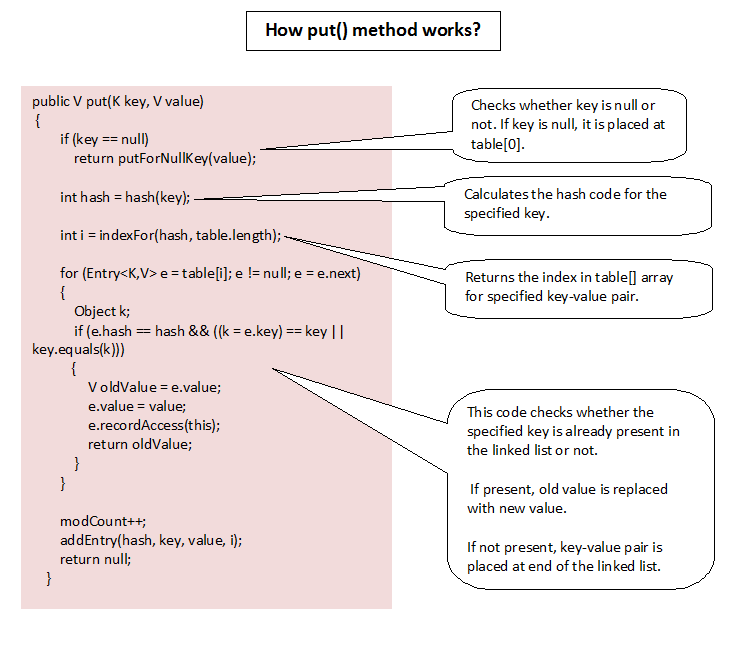
Step 1 : First checks whether the key is null or not. If the key is null, it calls *putForNullKey()* method. *table[0]* is always reserved for null key. Because, hash code of null is 0.

Step 2 : If the key is not null, then it calculates the hash code of the key by calling *hash()* method.

Step 3 : Calls *indexFor()* method by passing the hash code calculated in step 2 and length of the *table[]* array. This method returns index in *table[]* array for the specified key-value pair.

Step 4 : After getting the index, it checks all keys present in the linked list at that index ( or bucket). If the key is already present in the linked list, it replaces the old value with new value.

Step 5 : If the key is not present in the linked list, it appends the specified key-value pair at the end of the linked list.



### How get() method Works?

**public** V get(Object key) {

**if** (key == **null**)

**return** getForNullKey();

**int** hash = hash(key.hashCode());

**for** (Entry<K , V> e = table[indexFor(hash, table.length)]; e != **null**; e = e.next) {

        Object k;

**if** (e.hash == hash && ((k = e.key) == key || key.equals(k)))

**return** e.value;

    }

**return** **null**;

}

Step 1 : First checks whether specified key is null or not. If the key is null, it calls *getForNullKey()* method.

Step 2 : If the key is not null, hash code of the specified key is calculated.

Step 3 : *indexFor()* method is used to find out the index of the specified key in the *table[]* array.

Step 4 : After getting index, it will iterate though linked list at that position and checks for the key using equals() method. If the key is found, it returns the value associated with it. otherwise returns null.

# [What Are Initial Capacity And Load Factor Of HashMap In Java?](https://javaconceptoftheday.com/initial-capacity-and-load-factor-of-hashmap-in-java/)

*HashMap* almost gives constant time performance for most frequent operations – insertion and retrieval. That’s why *HashMap* is the first choice for the big sized data having requirement of faster retrieval and faster insertion operations. There are two factors which affect the performance of *HashMap*. One is the ****load factor**** and another one is ****initial capacity****. You have to choose these two factors very carefully while constructing an *HashMap* object.

### Initial Capacity Of HashMap :

The capacity of an *HashMap* is the number of buckets in the hash table. The initial capacity is the capacity of an *HashMap* at the time of its creation. **The default initial capacity of the *HashMap* is 24 i.e 16.** The capacity of the *HashMap* is doubled each time it reaches the threshold. i.e the capacity is increased to 25=32, 26=64, 27=128….. when the threshold is reached.

### Load Factor Of HashMap :

Load factor is the measure which decides when to increase the capacity of the *HashMap*. The default load factor is 0.75f.

### How The Threshold Is Calculated?

The threshold of an *HashMap* is the product of current capacity and load factor.

****Threshold = (Current Capacity) \* (Load Factor)****

For example, if the *HashMap* is created with initial capacity of 16 and load factor of 0.75f, then threshold will be,

Threshold = 16 \* 0.75 = 12

That means, the capacity of the *HashMap* is increased from 16 to 32 after the 12th element (key-value pair) is added into the *HashMap*.

### How Initial Capacity And Load Factor Affect Performance Of HashMap?

Whenever *HashMap* reaches its threshold, ****rehashing**** takes place. Rehashing is a process where new *HashMap* object with new capacity is created and all old elements (key-value pairs) are placed into new object after recalculating their hashcode. This process of rehashing is both space and time consuming. So, you must choose the initial capacity, by keeping the number of expected elements (key-value pairs) in mind, so that rehashing process doesn’t occur too frequently.

You also have to be very careful while choosing the load factor. According to *HashMap* doc, the default load factor of 0.75f always gives best performance in terms of both space and time. For example,

If you choose load factor as 1.0f, then rehashing takes place after filling 100% of the current capacity. This may save the space but it will increase the retrieval time of existing elements. Suppose if you choose load factor as 0.5f, then rehashing takes place after filling 50% of the current capacity. This will increase the number of rehashing operations. This will further degrade the HashMap in terms of both space and time.

So, you have to be very careful while choosing the initial capacity and load factor of an *HashMap* object. Choose the initial capacity and load factor such that they minimize the number of rehashing operations.

### Similarities Between HashMap And HashSet In Java :

1) Both data structures don’t maintain any order for the elements.

2) Both use *hashCode()* and *equals()* method to maintain the uniqueness of the data.

3) The iterators returned by both are fail-fast in nature.

4) Both give constant time performance for insertion and removal operations.

5) Both are not synchronized.

### Differences Between HashMap And HashSet In Java :

****1) Hierarchy****

*HashSet* implements the *Set* interface which in turn extends the *Collection* interface, the top level interface in the Java Collection Framework. But, *HashMap* implements the *Map* interface which starts it’s own hierarchy totally different from the *Collection* interface.

****2) Data Storage****

*HashSet* stores the data as objects where as *HashMap* stores the data as key-value pairs. Where each value is recognized and retrieved by it’s key.

****3) Internal Structure****

*HashSet* internally uses *HashMap* to store it’s elements [[See more](https://javaconceptoftheday.com/how-hashset-works-internally-in-java/" \t "https://javaconceptoftheday.com/differences-between-hashmap-vs-hashset-in-java/_blank)]. *HashSet* is sometimes considered as a wrapper around the *HashMap*. On the other hand, *HashMap* internally uses an array of *Entry<K, V>* objects to store the data.

****4) Duplicate Values****

*HashSet* doesn’t allow duplicate elements. If you try to insert duplicate element, *HashSet* will be unchanged. Where as *HashMap* allows duplicate values but doesn’t allow duplicate keys.

****5) null values****

*HashSet* can hold only one null value where as *HashMap* can hold multiple null values but allows only one null key.

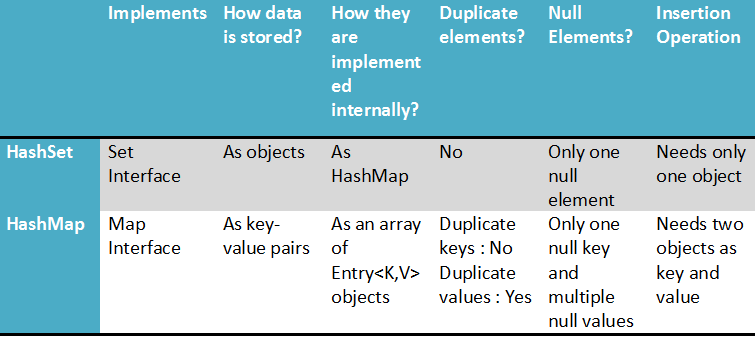
****6) Insertion Operation****

Insertion operation on *HashSet* requires only one object where as insertion operation on *HashMap* requires two objects, key and value.

****7) Performance****

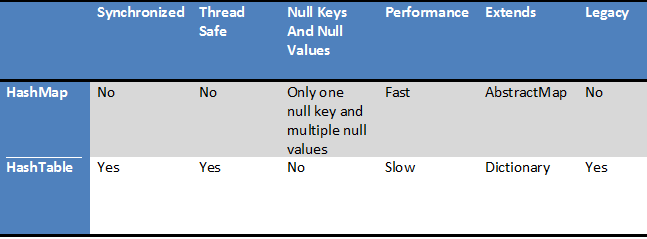
Performance of both is almost the same. But, some developers say that *HashMap* is slightly faster than the *HashSet*.

****8) Usage:**** Use the *HashSet* when you need uniqueness of the data. Otherwise, *HashMap* is always preferred as *HashSet* internally uses *HashMap*to store the data.



### Differences Between HashMap And HashTable In Java :

|  |  |
| --- | --- |
| ****HashMap**** | ****HashTable**** |
| HashMap is not synchronized and therefore it is not thread safe. | HashTable is internally synchronized and therefore it is thread safe. |
| HashMap allows maximum one null key and any number of null values. | HashTable doesn’t allow null keys and null values. |
| Iterators returned by the HashMap are fail-fast in nature. | Enumeration returned by the HashTable are fail-safe in nature. |
| HashMap extends AbstractMap class. | HashTable extends Dictionary class. |
| HashMap returns only iterators to traverse. | HashTable returns both Iterator as well as Enumeration for traversal. |
| HashMap is fast. | HashTable is slow. |
| HashMap is not a legacy class. | HashTable is a legacy class. |
| HashMap is preferred in single threaded applications. If you want to use HashMap in multi threaded application, wrap it using Collections.synchronizedMap() method. | Although HashTable is there to use in multi threaded applications, now a days it is not at all preferred. Because, ConcurrentHashMap is better option than HashTable. |



Similarities Between HashMap And HashTable In Java :

* Both store the data in the form of key-value pairs.
* Both use Hashing technique to store the key-value pairs.
* Both implement Map interface.
* Both doesn’t maintain any order for elements.
* Both give constant time performance for insertion and retrieval operations.