**The difference between add() and set() method**

Both of these methods add an object to a list at a specific index. But add inserts an item, shifting items "to the right" if necessary. Set replaces the item at that index.  
For example, if your list is [1,2,3,4,5]  
  
list.add(3, 10) will give [1,2,3,10,4,5,]  
list.set(3, 10) will give [1,2,3,10,5]

**What is the use of l.remove()**

The remove(Object obj) method in list interface java is used to remove the first occurrence of the specified element obj from this list if it is present in the list.

Eg: list:{1,2,10,20}

obj = 10

List.remove (obj)={1,2,20}

**What are Vector, Queue and Dequeue ?**

Vector class:

The Vector class implements a growable array of objects. Vectors basically fall in legacy classes but now it is fully compatible with collections.

* Vector implements a dynamic array that means it can grow or shrink as required. Like an array, it contains components that can be accessed using an integer index
* They are very similar to ArrayList but Vector is synchronised and have some legacy method which collection framework does not contain.
* It extends AbstractList and implements List interfaces.

Queue:

* The Queue interface is available in java.util package and extends the Collection interface.
* The queue collection is used to hold the elements about to be processed
* Provides various operations like the insertion, removal etc.
* It is an ordered list of objects with its use limited to insert elements at the end of the list and deleting elements from the start of list (FIFO principle).
* Being an interface the queue needs a concrete class for the declaration and the most common classes are the [PriorityQueue](https://www.geeksforgeeks.org/priority-queue-class-in-java-2/" \t "_blank) and [LinkedList](https://www.geeksforgeeks.org/linked-list-in-java/) in Java.
* Both the implementations are not thread safe.
* PriorityBlockingQueue is one alternative implementation if thread safe implementation is needed.

Methods in queue:

* add()- This method is used to add elements at the tail of queue.
* peek()- This method is used to view the head of queue without removing it. It returns Null if the queue is empty.
* element()- This method is similar to peek(). It throws NoSuchElementException when the queue is empty.
* remove()- This method removes and returns the head of the queue. It throws NoSuchElementException when the queue is empty.
* poll()- This method removes and returns the head of the queue. It returns null if the queue is empty.
* size()- This method return the no. of elements in the queue.

Dequeue :

* The java.util.Deque interface is a subtype of the [java.util.Queue](https://www.geeksforgeeks.org/queue-interface-java/) interface.
* The Deque is related to the double-ended queue that supports addition or removal of elements from either end of the data structure.
* It can be used as a [queue (first-in-first-out/FIFO)](https://www.geeksforgeeks.org/queue/) or as a [stack (last-in-first-out/LIFO)](https://www.geeksforgeeks.org/stack/).
* These are faster than Stack and LinkedList.

Methods of Dequeue

* a[dd(element)](https://www.geeksforgeeks.org/deque-add-method-in-java/): Adds an element to the tail.
* [addFirst(element)](https://www.geeksforgeeks.org/deque-addfirst-method-in-java-with-examples/): Adds an element to the head.
* [addLast(element)](https://www.geeksforgeeks.org/deque-addlast-method-in-java/): Adds an element to the tail.
* [offer(element)](https://www.geeksforgeeks.org/deque-offer-method-in-java/): Adds an element to the tail and returns a boolean to explain if the insertion was successful.
* [offerFirst(element)](https://www.geeksforgeeks.org/deque-offerfirst-method-in-java/): Adds an element to the head and returns a boolean to explain if the insertion was successful.
* [offerLast(element)](https://www.geeksforgeeks.org/deque-offerlast-method-in-java/): Adds an element to the tail and returns a boolean to explain if the insertion was successful.
* [iterator()](https://www.geeksforgeeks.org/deque-iterator-method-in-java/): Returns an iterator for this deque.
* [descendingIterator()](https://www.geeksforgeeks.org/deque-descendingiterator-method-in-java/): Returns an iterator that has the reverse order for this deque.

**Logic behind Hashset**

<https://javaconceptoftheday.com/how-hashset-works-internally-in-java/>

**Hashset** uses HashMap internally to store it’s objects. Whenever you create a HashSet object, one **hashmap** object associated with it is also created. This HashMap object is used to store the elements you enter in the HashSet. The elements you add into HashSet are stored as **keys** of this HashMap object. The value associated with those keys will be a **constant**. Every constructor of HashSet class internally creates one HashMap object.

**Difference between HashMap and HashTable.**

* HashMap is non synchronized. It is not-thread safe and can’t be shared between many threads without proper synchronization code whereas Hashtable is synchronized. It is thread-safe and can be shared with many threads.
* HashMap allows one null key and multiple null values whereas Hashtable doesn’t allow any null key or value.
* HashMap is generally preferred over HashTable if thread synchronization is not needed
* HashMap is a new class introduced in JDK1.2, whereas HashTable is a legacy class.
* HashMap is fast and HashTable is slow.
* HashMap is traversed by Iterator whereas HashTable can be traversed by Enumerator and Iterator.
* HashMap inherits AbstractMap class and HashTable inherits Dictionary class.
* Vectors are good for random read access and insertion and deletion in the back (takes amortized constant time), but bad for insertions and deletions in the front or any other position (linear time, as items have to be moved). Vectors are usually laid out contiguously in memory, so traversing one is efficient because the CPU memory cache gets used effectively.
* Linked list on the other hand are good for inserting and deleting items in the front or back (constant time), but not particularly good for much else: For example deleting an item at an arbitrary index in the middle of the list takes linear time because you must first find the node. On the other hand, once you have found a particular node you can delete it or insert a new item after it in constant time, something you cannot do with a vector. Linked lists are also very simple to implement, which makes them a popular data structure.

**Difference between vector and linked list.**

Vectors are good for random read access and insertion and deletion in the back (takes amortized constant time), but bad for insertions and deletions in the front or any other position (linear time, as items have to be moved). Vectors are usually laid out contiguously in memory, so traversing one is efficient because the CPU memory cache gets used effectively. Linkedlists  on the other hand are good for inserting and deleting items in the front or back (constant time), but not particularly good for much else: For example deleting an item at an arbitrary index in the middle of the list takes linear time because you must first find the node. On the other hand, once you have found a particular node you can delete it or insert a new item after it in constant time, something you cannot do with a vector. Linked lists are also very simple to implement, which makes them a popular data structure.