Map.Entry interface in Java provides certain methods to access the entry in the Map. By gaining access to the entry of the Map we can easily manipulate them. Map.Entry is a generic and is defined in the java.util package.

Some of the methods are:

* equals (Object o) – It compares the object (invoking object) with the Object o for equality.
* getKey()– Returns the key for the corresponding map entry.
* getValue() – Returns the value for the corresponding map entry.

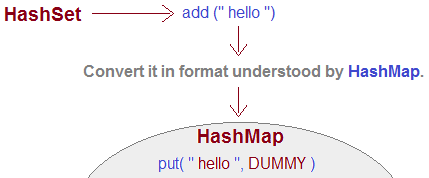
My most used use case for Map.Entry is for iterating through the map.

**HashSet....** is a collection class that implements **Set** Interface. HashSet stores its element by a process called **hashing**. In hashing, hash code of every element is computed internally and the value of hash code determines the index at which the element is stored in HashSet. Hence, the order of elements in HashSet is unknown and cannot be a guaranteed.

**Important points about HashSet Collection class**

* Duplicate elements are **NOT** allowed to be stored in a HashSet.
* The order of elements in a HashSet may appear **jumbled** in the output.
* Execution time for adding/removing element remains constant, irrespective of the size of HashSet.
* HashSet class doesn't provide any way to access its element by its index.
* HashSet class is **not synchronized** and hence not thread-safe by default.

HashSet internally extends AbstractSet and implements Set interface.



**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**.

Hash map and Hash table::-----

One of the major differences between HashMap and Hashtable is that  HashMap is non-synchronized whereas Hashtable is synchronized

The HashMap class is roughly equivalent to Hashtable, except that it  permits nulls. (HashMap allows null values as key and value whereas  Hashtable doesn’t allow nulls).

HashMap does not guarantee that the order of the map will remain constant over time.

**HashMap:**

* Non-synchronized
* Allows one null key and multiple null values
* New class introduced in JDK 1.2
* Fast

**Hashtable:**

* Synchronized
* Doesn't allow any null key or value
* Legacy class
* Slow

Linked list-----

**LinkedList**

1. LinkedList internally uses **doubly linked list**to store the elements.
2. Manipulation with LinkedList is **faster** than ArrayList because it uses doubly linked list so no bit shifting is required in memory.
3. LinkedList class can **act as a list and queue**both because it implements List and Deque interfaces.
4. LinkedList is **better for manipulating** data.

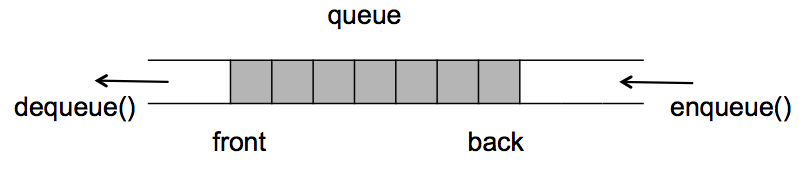
**Vector :**

1. Vector is ***implementation***class of List interface (i.e.; Vector implements List)
2. Vector uses ***resizable array or grow-able array*** to store element/objects
3. ***Duplicate*** element/objects are allowed to be inserted
4. ***Insertion order*** is maintained
5. Allows ***NULL*** insertion (no limit, any number of NULL insertion is allowed

Queue and de-queue:-------------

Deque is an interface which extends Queue Interface.  
Inorder to understand more, we must learn about Queue interface.

**Queue is an interface** which extends Collection interface, in other words queue is a subtype of collection interface.



Queue is LIFO datastructure(an effective way to sort or organise data) where elements are inserted from one end, and deleted from another end.

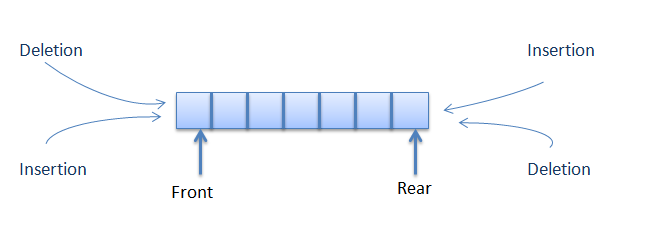
It has methods like add(),element(),preek(),search(),remove()

1. add()→Adding elements to the tail of Queue.
2. peek()→Returns head element of Queue without removing it, if Queue is empty returns null.
3. remove()→Returns and remove head element of Queue. **Returns NoSuchElementException if queue is empty.**
4. poll()→Returns and Remove head element of Queue. **Returns null if Queue is empty.**
5. element() → Returns head element of Queue without removing it. Returns null, if the queue is empty.

Note: As Queue is an interface which needs implemented classes like (LinkedList, PriorityQueue, etc).

**Difference between poll() and remove()?**  
Both are used for removing top most element of a queue, but when perform remove() on empty Queue, it return an exception, where as poll() will return null.

Now let’s **put some words on Deque**.  
Deque is double ended queue, which allows insertion and deletions from both end of Queue.  
It can be used as **LIFO(Queue) or Stack(FIFO)**.



Methods in Deque:

1)add() :→ add element to tail of Queue  
2)addFirst():→ add element to head of Queue  
3)addLast():→add element to tail of Queue  
4)removeFirst(): → remove element from head of Queue  
5)removeLast(): → remove element from tail of Queue  
6)pop(): → returns and remove first element of Queue  
7)offers(element)/offfersLast(element):→ adds element to the tail, and returns a boolean value.  
8)OffersFirst(element): → adds elements to the head, and returns a boolean value.  
9)iterator():→ Gives an iterator object, which helps in iterating an element.  
10)decendingIterator():→ Returns iterator, which returns reverse order of queue  
11)push(): → push elements to head of the Queue.

Any error events that occur during execution of program and break the normal flow are called **Exception**.

To handle such exception, Java provides a way known as Exception Handling. Some keywords used in Java for exception handling are :

1. **throw** – This keyword throws the exception to the runtime to handle it.

2. **throws** –This keyword is used to throw the exception but are not handled it, which let caller program find the exception that is thrown by the method,

3.**try-catch** – These keywords are used to handle the exception in the code, where try is the beginning of the block followed by catch block which require an argument that can be any type of exception.

4. **finally** – It is an optional block, which is mostly used with try-catch block. As exception blocks the code from execution, finally block as a backup which will always be execute, no matter whether exception occurred or not.

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** ListDemo {

**public** **static** **void** main(String args[])

{

List l=**new** ArrayList();

l.add(800);//auto boxing

l.add(800);//auto boxing

l.add(10);

System.***out***.println(l);//auto unboxing

Integer i=90;//auto boxing

String s="maruthi546";

System.***out***.println(i+s);

}

}

**import** java.util.HashSet;

**import** java.util.Set;

**public** **class** SetDemo {

**public** **static** **void** main(String args[])

{

Set s=**new** HashSet();

s.add(12);

s.add(67);

s.add(6);

s.add(6);

s.add(1);

s.add(**null**);

s.add(**null**);

s.add(**null**);

System.***out***.println(s);

//System.out.println("2 ele"+s.get(2));

System.***out***.println(s.contains(67));

System.***out***.println(s.isEmpty());

}

}

**import** java.util.Set;

**import** java.util.TreeSet;

**public** **class** TreeSetDemo {

**public** **static** **void** main(String args[])

{

Set<String> treeset=**new** TreeSet();

//treeset.add(8);

//treeset.add(1);

treeset.add("b");

treeset.add("B");

treeset.add("hi this is java");

System.***out***.println(treeset);

}

//output in ascending order,remove duplication,preference given to capital letters

}