Collection Framework

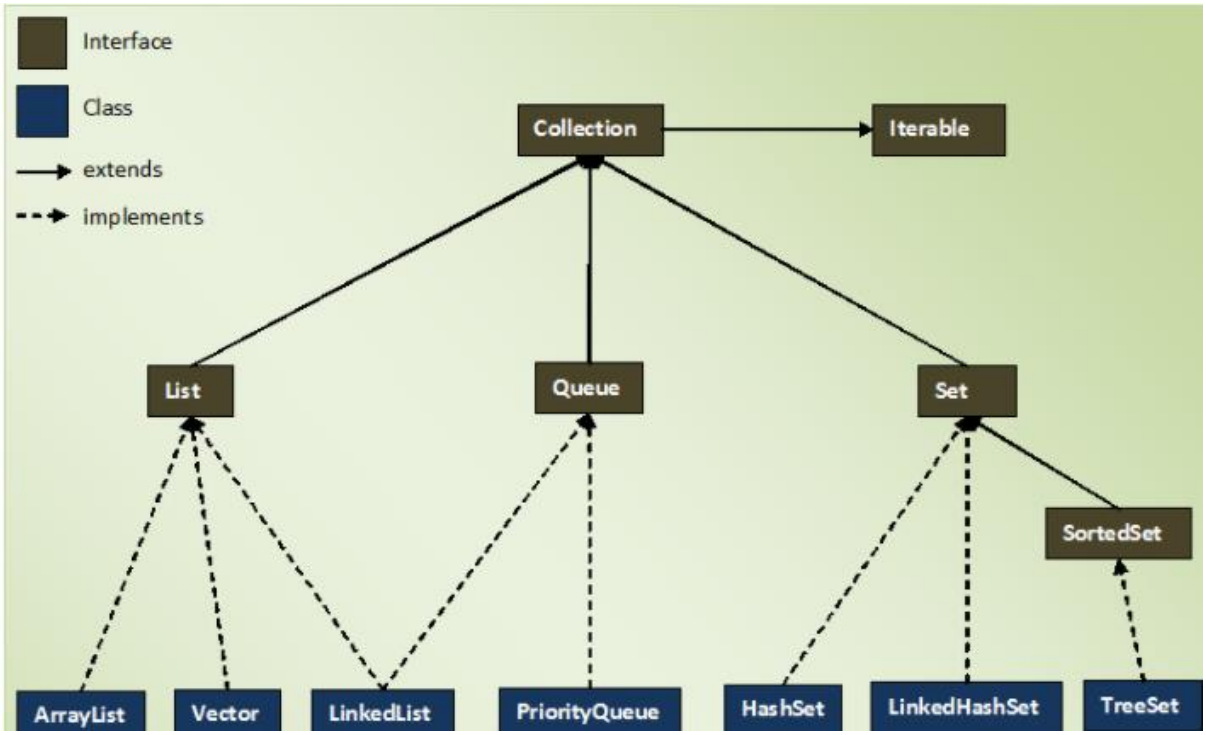
Collection Framework is architecture to manipulate (storing, insertion, deletion, sorting and searching) group of objects.

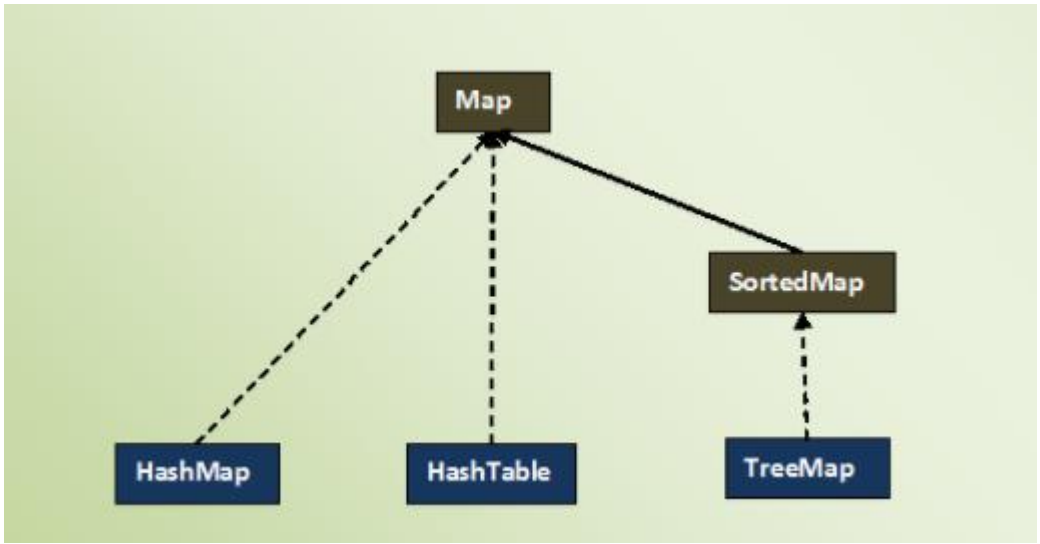
Parent Package :

Entire collection framework is divided into four interfaces-

1. List-Handles sequential (index basis) objects.
2. Set- Feature to store unique elements only.
3. Queue - Works on FIFO principle.
4. Map- Works on Key-Value pair principle.

Collection Hierarchy





Iterable interface has only one method iterator() which returns an Iterator object.

Exploring Collection Interface

The interface contains total 15 abstract methods out of which one is inherited from Iterable interface.

List of methods in Collection Interface

```
int size();
boolean isEmpty();
boolean contains(Object o);
Iterator<E> iterator();  (method of Iterable)
Object[] toArray();
T[] toArray(T[] a);
boolean add(E e);
boolean remove(Object o);
boolean containsAll(Collection<?> c);
boolean addAll(Collection<? extends E> c);
boolean removeAll(Collection<?> c);
boolean retainAll(Collection<?> c);
void clear();
boolean equals(Object o);
int hashCode();
```

Difference between Object[] toArray() and T[] toArray(T[] a)

The main difference is first one return object irrespective of content type of list while the second will return the exact type present in the list.

```
int size=list.size();
Object array1[]=list.toArray();
System.out.println(array1[2]);
String array2[]=list.toArray(new String[size]);
```

Note : 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 sub classes of 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

It works on index basis. So elements are added sequentially. It can contain duplicate and multiple null elements.

ArrayList, LinkedList and Vector are three classes implementing List Interface.

List interface provides ListIterator and Iterator interface to traverse a list. Using ListIterator list can be traversed both forward and backward direction. Iterator interface allows only traversing in forward direction.

Example-

```
public class Test {
public static void main(String[] args) {
                List<String> list=new ArrayList<>();
                list.add("one");
                list.add("two");
                list.add("three");
                list.add("four");
                list.add("five");
                ListIterator<String> lt=list.listIterator();
                System.out.println("Elements in forward direction");
                while(lt.hasNext())
                {
                        System.out.print(lt.next()+"  -  ");
                }
                System.out.println("");
                System.out.println("Elements in Backward direction");
                while(lt.hasPrevious())
                {
                        System.out.print(lt.previous()+"  -  ");
                }
        }
```

Output

```
Elements in forward direction
one - two - three - four - five -
Elements in Backward direction
five - four - three - two - one -
```

If you try like-

```
while(lt.hasPrevious())
                {
                        System.out.print(lt.next()+"  -  ");
                }
```

Then NoSuchElementFound exception will be thrown

ArrayList

- Internally uses dynamic array to store the elements.
- Maintains insertion order.
- Can contain duplicate elements.
- Can contain multiple null elements.
- Is non-Synchronized.
- Default initial capacity of an ArrayList is 10.
- If the know the element, you can retrieve the element.
- You can also specify initial capacity of an ArrayList while creating it. But no exception will be thrown if number of added elements exceeds than the defined size as it is re-sizable array.

Advantage of ArrayList over Arrays

- Resizable-Grow and Shrink dynamically.
- Insertion and deletion from particular position.
- Both side traversal.
- If generics not used it can hold any type of object which is not possible in array.

Different ways to Iterating an ArrayList

```
public class Test {

	public static void main(String[] args) {
		ArrayList<String>list=new ArrayList<>(2);
		list.add("One");
		list.add("Two");
		list.add("Three");
		list.add("Four");
		list.add("Five");
		/*Using For Loop*/
		for(int i=0;i<list.size();i++)
		{
			//System.out.println(list.get(i));
		}
		/*Using Iterator*/
		Iterator<String>iterator=list.iterator();
		while(iterator.hasNext())
		{
			iterator.remove();//Removing from list
		}
		/*Using ListIterator*/
		ListIterator<String> listIterator=list.listIterator();
		while(listIterator.hasPrevious())
		{
			System.out.println(listIterator.previous());
		}
		/*Using Enhanced For Loop*/

		for(String element : list)
		{
			System.out.println(element);
		}

		}
                                        }
```

Difference between Iterator and ListIterator

Both are two different interfaces in Java used for iteration. But there are some of the differences.

1. Using Iterator you can traverse List, Queue, Set and Map. But ListIterator can iterate List objects only no Set, Queue or Map.
2. Iterator can iterate in forward direction only while ListIterator can iterate in both forward and backward direction.
3. Using ListIterator you can perform modifications (insert,replace,remove) but using Iterator you can remove only.

```
package test;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.ListIterator;

public class Test {

	public static void main(String[] args) {
		ArrayList<String> list = new ArrayList<>();
		list.add("One");
		list.add("Two");
		list.add("Three");
		list.add("Four");
		list.add("Five");
		Iterator<String> itr=list.iterator();
		while(itr.hasNext())
		{
			itr.next();--Remove this and observe
			itr.remove();

		}
		System.out.println(list);

		ArrayList<String> list1=new ArrayList<>();
		list1.add("One");
		list1.add("Two");
		list1.add("Three");
		list1.add("Four");
		list1.add("Five");
		ListIterator<String> litr= list1.listIterator();
		while(litr.hasNext())
		{
			litr.next();--Remove this and observe
			litr.set("Modified");
		}
		System.out.println(list1);
	}
                                        }
```

Output

```
[]

[Modified, Modified, Modified, Modified, Modified]
```

Note:

In most of the practices it is found that user experiences Illegal State Exception while doing modifications using iterator or listIterator. The reason is in while loop they forget to include Itr.next() so iterator does not event points to first element so Illegal State Exception is thrown.

4. Using ListIterator, you can iterate a list from the specified index. It is not possible with Iterator.It will always iterate from start index.

```
ArrayList<String> list1=new ArrayList<>();
              list1.add("One");
              list1.add("Two");
              list1.add("Three");
              list1.add("Four");
              list1.add("Five");
              ListIterator<String> litr= list1.listIterator(2);//It will pick
elements from index 2
              while(litr.hasNext())
              {
                     System.out.println(litr.next());

                                                  }
```

Output

```
Three,Four,Five
```

5. Using ListIterator you can get index of next and previous element which is not possible in Iterator. It has method previousIndex() and nextIndex().

```
public class Test {

        public static void main(String[] args) {

                ArrayList<String> list1=new ArrayList<>();
                list1.add("One");
                list1.add("Two");
                list1.add("Three");
                list1.add("Four");
                list1.add("Five");
                ListIterator<String> litr= list1.listIterator();
                while(litr.hasNext())
                {
                        System.out.println(litr.nextIndex()+" : "+litr.next());

                }
                System.out.println("----------------------------------------------------");
                while(litr.hasPrevious())
                {
                        System.out.println(litr.previousIndex()+" : "+litr.previous());
                }

                                                }
```

Output

```
0 : One
1 : Two
2 : Three
3 : Four
4 : Five
-------------------------------------------------------
4 : Five
3 : Four
2 : Three
1 : Two
0 : One
```

Questions on ArrayList

How to increase current capacity of arraylist?-Using ensureCapacity() method.

Example

```
ArrayList<String> list1=new ArrayList<>();
            //initial default capacity is 10
                    list1.ensureCapacity(20);//Now capacity is increased to 20
```

How to decrease capacity of an arraylist?-using trimToSize() method.

```
ArrayList<String> list1=new ArrayList<>();//Default size is 10
            list1.ensureCapacity(20);//increased to 20
            list1.add("One");
            list1.add("Two");
            list1.add("Three");
            list1.add("Four");
            list1.add("Five");
                list1.trimToSize();//Reduces current capacity to current size of
                                    arraylist
```

```
ArrayList<String> list=new ArrayList<>();
            list.add("JSP");
            list.add("Java");
            list.add("Hibernate");
            list.add("Java");
            list.add("JSP");
            list.add("Java");
            list.add("Spring");
            System.out.println(list.indexOf("Java"));//1
            System.out.println(list.lastIndexOf("Java"));//5
            System.out.println(list.indexOf("JSP"));//0
            System.out.println(list.lastIndexOf("JSP"));//4
            /*Sub List in ArrayList*/
            List<String> subList=list.subList(2, 5);
            for(String str : subList)
            {
                  System.out.println(str);
```

```
            }
            /*How to join two ArrayList*/
            List<String> list1=new ArrayList<>();
            list1.add("TCS");
            list1.add("Wipro");
            //Here we are joining two ArrayList
            list.addAll(list1);
            System.out.println(list);
            /*How to join more than one elements a point in an ArrayList*/
            //Solution is adding a list at a a specific position of another list
            list.addAll(2, list1);
            System.out.println(list);
```

```
Output

1

5

0

4

Hibernate

Java

JSP

[JSP, Java, Hibernate, Java, JSP, Java, Spring, TCS, Wipro]

[JSP, Java, TCS, Wipro, Hibernate, Java, JSP, Java, Spring, TCS, Wipro]
```

LinkedList

- It is a data structure where each element consist of three things. First one is the reference to previous element, second one is the actual value of the element and last one is the reference to next element.
- Linkedlist implements both List and Deque so it can be used as a Queue also.
- LinkedList class in Java is not of type Random Access**.** 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.
- 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

Insert, delete and get in LinkedList

```
/*LinkedList as a List*/
            /*Insertion*/
            LinkedList<String> list=new LinkedList<>();
            list.addFirst("TCS");//Adding at first
            list.addLast("Infosys");//adding at last
            list.add(1,"Wipro");//adding at middle using index. method inherited
from Collection Interface.
```

```
list.add(2,"TechMahindra");
list.add(3,"Deloitte");
list.add(4,"UHG");
list.add(5,"MindTree");
for(String str : list)
{
	System.out.print("["+str+"]"+" ");
}
System.out.println();
/*Removing elements*/
list.removeFirst();
list.remove(3);
list.removeLast();
System.out.println("After Removal");
for(String str : list)
{
	System.out.print("["+str+"]"+" ");
}
System.out.println();
/* Retrieving */
System.out.println(list.getFirst());
System.out.println(list.get(2));
System.out.println(list.getLast());
```

Output

[TCS] [Wipro] [TechMahindra] [Deloitte] [UHG] [MindTree] [Infosys]
After Removal
[Wipro] [TechMahindra] [Deloitte] [MindTree]
Wipro
Deloitte
MindTree

Traversing a LinkedList in backward direction(From tail to head)

LinkedList class has a method called descendingIterator

```
LinkedList<String> list=new LinkedList<>();
		list.add("JSP");
		list.add("Java");
		list.add("Hibernate");
		list.add("Java");
		list.add("JSP");
		list.add("Java");
		list.add("Spring");
		Iterator<String> itr=list.descendingIterator();
		while(itr.hasNext())
		{
			System.out.print(itr.next()+" ");
		}

list.removeFirstOccurrence("Java");
list.removeLastOccurrence("JSP");
```

```
Output-

Spring Java JSP Java Hibernate Java JSP
```

Adding an ArraList at the end of LinkedList

```
LinkedList<String> linkedList=new LinkedList<>();
```

```
ArrayList<String>arrayList=new ArrayList<>();
linkedList.addAll(arrayList);
```

Write a Java program which implements LinkedList as a Queue (FIFO)?

```
LinkedList<Integer> queue = new LinkedList<Integer>();

                    //adding the elements into the queue

                    queue.offer(10);

                    queue.offer(20);

                    queue.offer(30);

                    queue.offer(40);

                    //Printing the elements of queue

                    System.out.println(queue);     //Output : [10, 20, 30, 40]

                    //Removing the elements from the queue

                    System.out.println(queue.poll());   //Output : 10

                    System.out.println(queue.poll());   //Output : 20
```

How do you use LinkedList as Stack (LIFO)?

LinkedList has pop() and push() methods which make LinkedList to function as a Stack.

```
LinkedList<Integer> stack = new LinkedList<Integer>();
              //pushing the elements into the stack
         stack.push(10);
         stack.push(20);
         stack.push(30);
         stack.push(40);
         //Printing the elements of stack
        System.out.println(stack);      //Output : [40, 30, 20, 10]
         //Poping out the elements from the stack
          System.out.println(stack.pop());    //Output : 40
         System.out.println(stack.pop());    //Output : 30
```

ArrayList VS LinkedList

	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.
Retrieval (Searching or getting an element)	Insertion and removal operations in ArrayList are of order O(n). Retrieval of elements in the ArrayList is faster than the LinkedList . Because all elements in ArrayList are index based.	Insertion and removal in LinkedList are of order O(1). 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.
Random Access	Retrieval operation in ArrayList is of order of O(1). ArrayList is of type Random Access. i.e elements can be accessed randomly.	Retrieval operation in LinkedList is of order of O(n). 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 ArravList.	If your application does more insertions and deletions than the retrieval, then use LinkedList.

Vector

Vectors also implement dynamic array. It is similar to arraylist but with two differences-

- Vector is synchronized.
- It is thread safe.
- All methods of Vector class is synchronized.
- It contains many legacy methods which are not part of Collection Framework.
- Default size of vector is 10.

Vector is a Legacy class in Java. Earlier version of Java did not include Collection Framework. At that time Legacy class and interface was used to store and manipulate objects.
Legacy Class Examples : Vector, Stack, Properties, HashTable, Dictionary
Legacy Interface Examples : Enumeration
Note : All legacy classes are synchronized.

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.
It has some legacy methods like firstElement() and lastElement() which is not present in ArrayList or LinkedList.

Why not to use Vector in your code?
Answer : The major advantage of Vector is thread safety. As per design when we create a Vector we get re-sizable and synchronized array which will reduce performance.
We can make ArrayList as synchronized list when we wish using synchronizedList() method of Collections class.
Syntax :

```
ArrayList<String> list=new ArrayList<>();
                  Collections.synchronizedList(list);
```

ArrayList VS Vector

1) ArrayList is not synchronized. Vector is synchronized.

2) ArrayList increments 50% of current array size if number of element exceeds from its capacity. Vector increments 100% means doubles the array size if total number of element exceeds than its capacity.

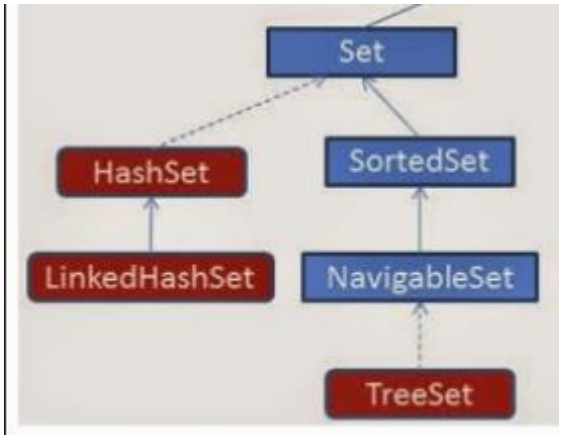
3) ArrayList is not a legacy class, it is introduced in JDK 1.2. Vector is a legacy class.

4) ArrayList is fast because iSt is non-synchronized. Vector is slow because it is synchronized i.e. in multithreading environment, it will hold the other threads in runnable or non-runnable state until current thread releases the lock of object.

5) ArrayList uses Iterator interface to traverse the elements. Vector uses Enumeration interface to traverse the elements. But it can use Iterator also.

The Set Interface

The set interface defines a set which is a linear collection of objects with no duplicates. Set interface does not have its own methods. It's all methods are inherited from Collection Interface.



In Red-Class In Blue-Interface

Facts about Set

- **<u>One set can contain only one null element</u>**. If we add more than one null element and print all the set elements null be printed only once. No any exception is thrown.
- Duplicates are not allowed in Set. If you store duplicates and try to print complete set the element will be printed only once. No exception is thrown.
- Order of elements in Set depends on its implementation.
    - 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)
    - LinkedHashSet maintains insertion order.

- Two set instances, irrespective of their implementation types, are said to be equal if they contain same elements.

The Sorted 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 Sorted Interface

- As SortedSet is a set, duplicate elements are not allowed.
- SortedSet cannot have null elements. If you try to insert null element, it gives NullPointerException at run time.
- 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.

Example
If you don't mention any Comparator while creating a SortedSet, elements will be placed in ascending order. (Natural Ordering).
Printing comparator gives null.

```
SortedSet<String> companies=new TreeSet<>();
		companies.add("Wipro");
		companies.add("Infosys");
		companies.add("TCS");
		companies.add("IBM");
		companies.add("Infosys");
		companies.add("CTS");
		System.out.println("Total Number of Companies : "+companies.size());
		System.out.println("Set of Companies : "+companies);
		System.out.println("Comparator used :"+companies.comparator());
```

Output-
Total Number of Companies : 5
Set of Companies : [CTS, IBM, Infosys, TCS, Wipro]
Comparator used :null

Java Sorted Set Methods

1. **comparator**()
Returns the comparator used to order the elements in this set, or null if this set uses the natural ordering of its elements.
2. **first**()
Returns the first (lowest) element currently in this set.
3. **headSet**(E toElement)
Returns a view of the portion of this set whose elements are strictly less than toElement.
4. **last**()
Returns the last (highest) element currently in this set.
5. **subSet**(E fromElement, E toElement)
Returns a view of the portion of this set whose elements range from fromElement, inclusive, to toElement, exclusive.
6. **tailSet**(E fromElement)
Returns a view of the portion of this set whose elements are greater than or equal to fromElement.

Important Note :
All elements inserted into a sorted set must implement the Comparable interface.
and If you try to sort a set of elements which do not implement Comparable or not has a specific Comparator, a ClassCastException will be thrown

Java Sorted Set Example

Employee.java

```
package collection.set;

public class Employee implements Comparable<Employee> {
	private int id;
	private String name;
	private double salary;

	public Employee(int id, String name, double salary) {
		this.id=id;
		this.name=name;
		this.salary=salary;


	}
```

```
	public int getId() {
		return id;
	}

	public void setId(int id) {
		this.id = id;
	}

	public String getName() {
		return name;
	}

	public void setName(String name) {
		this.name = name;
	}

	public double getSalary() {
		return salary;
	}

	public void setSalary(double salary) {
		this.salary = salary;
	}

	@Override
	public int compareTo(Employee o) {

		return this.id-o.getId();//ascending order
		//return o.getId()-this.id;->Descending order
	}

}
```

SortedSetDemo.javas

```
package collection.set;

import java.util.Iterator;
import java.util.SortedSet;
import java.util.TreeSet;

public class SortedSetDemo {

	public static void main(String[] args) {

		Employee employee2=new Employee(43, "Rama", 3000);
		Employee employee3=new Employee(565, "Uuvraj", 13987);
		Employee employee1=new Employee(10, "Sanjay", 12000);
		Employee employee4=new Employee(109, "Satya", 50000);
		Employee employee5=new Employee(104, "Deepayan", 4778);
		SortedSet<Employee> empSet=new TreeSet<>();
		empSet.add(employee1);
		empSet.add(employee2);
		empSet.add(employee3);
		empSet.add(employee4);
		empSet.add(employee5);
		System.out.println(empSet.size());
```

```
		System.out.println("List of  All Employees");
		System.out.println("-----------------------------------------------------------");
		Iterator<Employee> itr=empSet.iterator();
		while(itr.hasNext())
		{
			Employee emp=(Employee)itr.next();
			System.out.println("Emp Id : "+emp.getId()+" Emp Name :
"+emp.getName()+" Emp Salary :"+emp.getSalary());
		}
		System.out.println("-----------------------------------------------------------");
		System.out.println("Employee Set Comparator used :"+empSet.comparator());
		System.out.println("-----------------------------------------------------------");
		System.out.println("First Employee :"+empSet.first().getId());
		System.out.println("-----------------------------------------------------------");
		System.out.println("Last Employee :"+empSet.last().getId());
		System.out.println("-----------------------------------------------------------");
		System.out.println("Head Set Result");
		System.out.println("-----------------------------------------------------------");
		SortedSet<Employee>headSet=empSet.headSet(employee4);
		//Here it will give set of employees whose id is less than 109 empID of employee4 is
109
		Iterator<Employee> headSetIterator=headSet.iterator();
		while(headSetIterator.hasNext())
		{
			Employee emp=(Employee)headSetIterator.next();
			System.out.println("Id :"+emp.getId()+" "+" Name : "+emp.getName()+" "+"
Salary "+emp.getSalary());
		}

		SortedSet<Employee> subSet=empSet.subSet(employee1, employee4);
		/*in argument if you are passing (employee4, employee1) illegal argument will be
thrown*/
		Iterator<Employee> subSetIterator=subSet.iterator();
		System.out.println("-----------------------------------------------------------");
		System.out.println("Subset Result");
		while(subSetIterator.hasNext())
		{
			Employee emp=(Employee)subSetIterator.next();
			System.out.println("Id :"+emp.getId()+" "+" Name : "+emp.getName()+" "+"
Salary "+emp.getSalary());
		}
		System.out.println("-----------------------------------------------------------");
		System.out.println("Tail Set Result");
		SortedSet<Employee>tailSet=empSet.tailSet(employee4);
		//Here it will give set of employees whose id is greater than equal 109 empID of
employee4 is 109
		Iterator<Employee> tailSetIterator=tailSet.iterator();
		while(tailSetIterator.hasNext())
		{
			Employee emp=(Employee)tailSetIterator.next();
```

```
                        System.out.println("Id :"+emp.getId()+" "+" Name : "+emp.getName()+" "+"
Salary "+emp.getSalary());
                }

        }

}
```

Output-

5

```
List of  All Employees
-----------------------------------------------------------
Emp Id : 10 Emp Name : Sanjay Emp Salary :12000.0
Emp Id : 43 Emp Name : Rama Emp Salary :3000.0
Emp Id : 104 Emp Name : Deepayan Emp Salary :4778.0
Emp Id : 109 Emp Name : Satya Emp Salary :50000.0
Emp Id : 565 Emp Name : Uuvraj Emp Salary :13987.0
-----------------------------------------------------------
Employee Set Comparator used :null
-----------------------------------------------------------
First Employee :10
-----------------------------------------------------------
Last Employee :565
-----------------------------------------------------------
Head Set Result
-----------------------------------------------------------
Id :10  Name : Sanjay  Salary 12000.0
Id :43  Name : Rama  Salary 3000.0
Id :104  Name : Deepayan  Salary 4778.0
-----------------------------------------------------------
Subset Result
Id :10  Name : Sanjay  Salary 12000.0
Id :43  Name : Rama  Salary 3000.0
Id :104  Name : Deepayan  Salary 4778.0
-----------------------------------------------------------
Tail Set Result
Id :109  Name : Satya  Salary 50000.0
Id :565  Name : Uuvraj  Salary 13987.0
```

Note : If you not modify return in compateTo() like return 0 only first object will be returned.

The NavigableSet Interface

The Navigable Set is a sorted set with navigation facilities.

Important methods in Navigable Interface.

1-5,

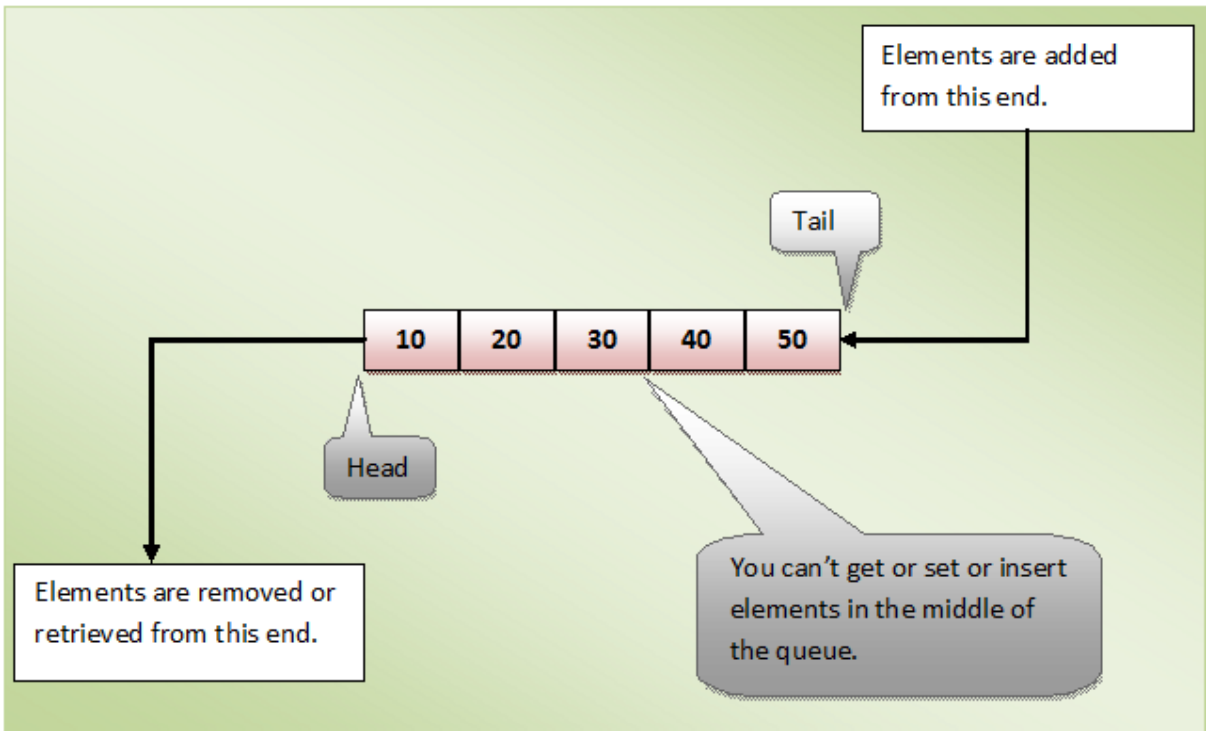
http://javaconceptoftheday.com/java-collection-framework-hashset-class/
http://javaconceptoftheday.com/how-hashset-works-internally-in-java/
http://javaconceptoftheday.com/java-hashset-example/
http://javaconceptoftheday.com/java-collection-framework-linkedhashset-class/
http://javaconceptoftheday.com/how-linkedhashset-works-internally-in-java/
http://javaconceptoftheday.com/java-linkedhashset-example/
http://javaconceptoftheday.com/java-collection-framework-treeset-class/
http://javaconceptoftheday.com/java-treeset-example/
http://javaconceptoftheday.com/hashset-vs-linkedhashset-vs-treeset-in-java/
https://examples.javacodegeeks.com/core-java/util/comparator/java-comparable-and-comparator-example-to-sort-objects

Collection Framework – The Queue Interface

The Queue Interface extends Collection interface.

Queue is a data structure where elements are added from one end called **tail** of the queue and elements are removed from another end called **head** of the queue. Queue is also **first-in-first-out** type of data structure (except priority queue).

How Typical Queue Works?



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.

Queue uses two packages

1. Java.util
2. Java.util.concurrent

Queue in java.util package

In this package there are four classes that implements Queue interface

1. AbstractQueue<E>
2. ArrayDeque<E>
3. LinkedList<E>
4. PriorityQueue<E>

Queue interface API structure

Basically, `Queue` provides three primary types of operations which differentiate a queue from others:

- Insert: adds an element to the tail of the queue.
- Remove: removes the element at the head of the queue.
- Examine: returns, but does not remove, the element at the head of the queue.

And for each type of operation, there are two versions:

- The first version throws an exception if the operation fails, e.g. could not add element when the queue is full.
- The second version returns a special value (either null or false, depending on the operation).

The following table summarizes the main operations of the `Queue` interface:

Type of operation	*Throws exception*	*Returns special value*
Insert	`add(e)`	`offer(e)`
Delete	`remove()`	`poll()`
Examine	`element()`	`peek()`

Insert Code Example :

```
public static void insertinLinkedlistQueue()
	{

queue1.add("one");
		queue1.offer("Two");
		/*
		 null can be added in linkedlist implementing queue as list allows
null to be added.
		 */
		queue1.add(null);
		queue1.add("Three");




	}

	public static void insertInPriorityQueue()
	{

		queue2.add("one");
		queue2.offer("Two");
		/*
		 Adding null will throw null pointer exception as it is
priorityQueue
		 The same applies to ArrayQueue and AbstractQueue
		 */
		queue2.add(null);
		queue2.add("Three");


	   }
```

Delete Code Example

Poll and remove method is used to delete. These methods obtains the object and then removes

```
public static void deleteFromQueue()
	{
		System.out.println(queue1.poll());
		System.out.println(queue1.remove());
		queue1.clear();
		System.out.println("Deleting using poll() method :"+queue1.poll());
	System.out.println("Deleting using remove() method :"+queue1.remove());

	   }
```

Output

one

Two

Deleting using poll() method :null

Exception in thread "main" java.util.NoSuchElementException

Examine Code Example

This is used to get head element without removing it.

```
public static void obtainWithoutRemoving()
{
		System.out.println("Head Element "+queue1.peek());
		System.out.println("Had Element "+queue1.element());
		queue1.clear();
		System.out.println("Head Element :"+queue1.peek());
		System.out.println("Head Element : "+queue1.element());


	}
```

Output

```
Head Element one

Had Element one

Head Element :null

Exception in thread "main" java.util.NoSuchElementException
```

Difference Between add() and offer()

Difference between both the methods comes in picture only when queue is bounded.

In unbounded queue there is no difference.

Both methods are used to insert in queue.
Add() comes from collection : It throws an exception if the operation fails.
Offer() comes from queue : It returns a false if the operation fails.

Example :

```
public static void QueueAddOperation()
	{
		BlockingQueue<Integer> bqueue=new ArrayBlockingQueue<>(2);
		/* Size of blocking queue is 2*/
		bqueue.add(1);
		bqueue.offer(2);
		/*Try to add third element and get the difference
		 * between add and offer as queue will be full now*/

		System.out.println("Queue is Full : Adding more objet using
offer:"+bqueue.offer(3));
		System.out.println("Queue is Full : Adding more objet using
add:"+bqueue.add(3));
```

```
            }
```

```
Output
Queue is Full : Adding more objet using offer:false
 java.lang.IllegalStateException: Queue full
```

Overall Summary

1. Exception Case. In addition it is “java.lang.IllegalStateException: Queue full” Exception while in delete or examine it is “java.util.NoSuchElementException”
2. Special Return value case : while addition false and while deletion or examine null.

Classification of Queue

In Java, we can find many Queue implementations. We can broadly categorize them into the following two types

- Bounded Queues
- Unbounded Queues

Bounded Queues are queues which are bounded by capacity that means we need to provide the max size of the queue at the time of creation.

Unbounded Queues are queues which are NOT bounded by capacity that means we should not provide the size of the queue.
All Queues which are available in java.util package are Unbounded Queues and Queues which are available in java.util.concurrent package are Bounded Queues.