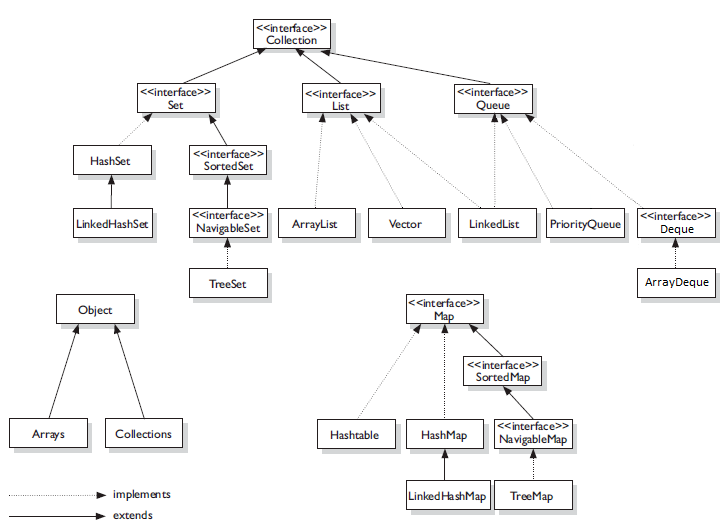
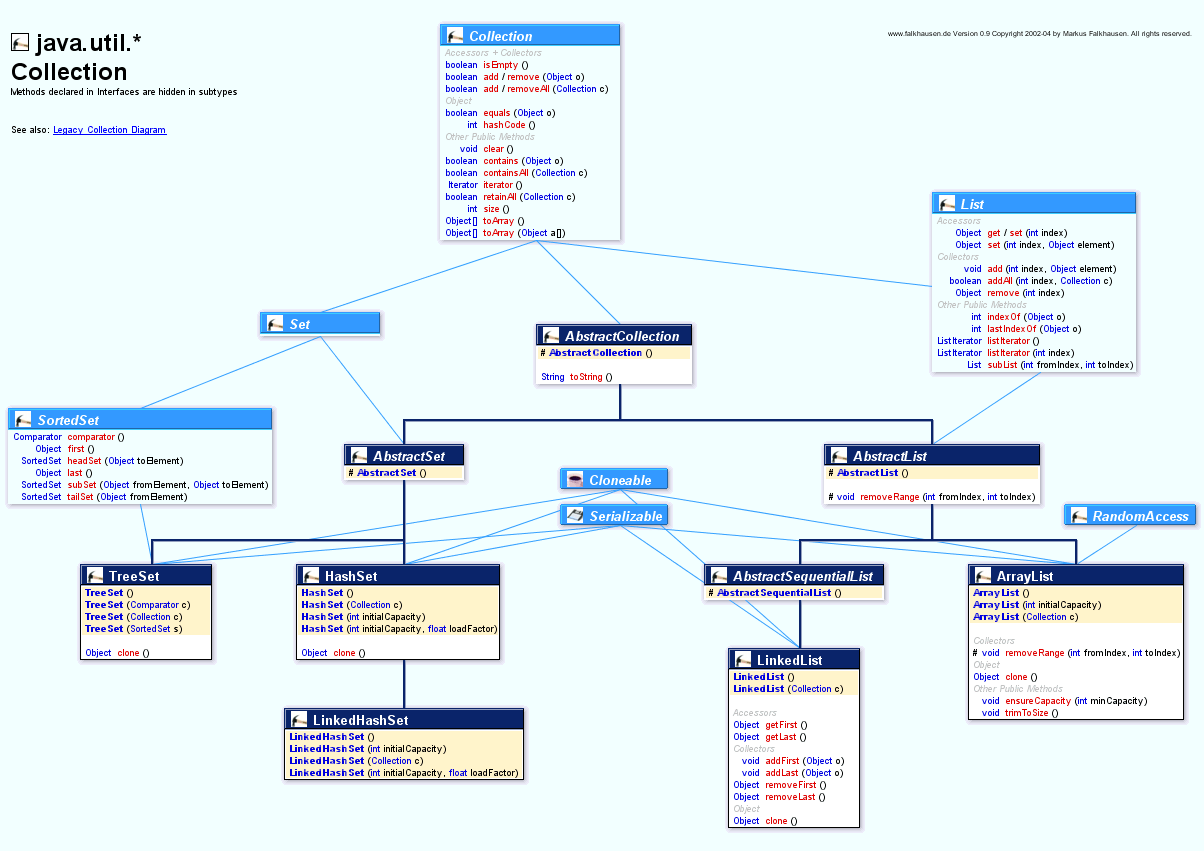
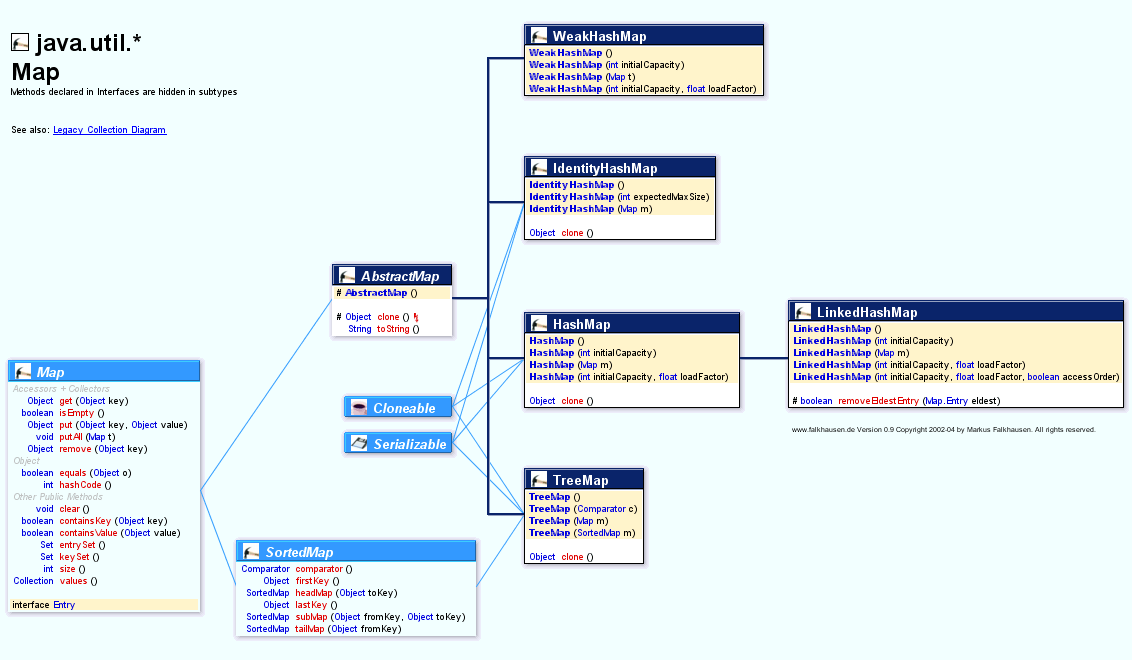
**Collection-**

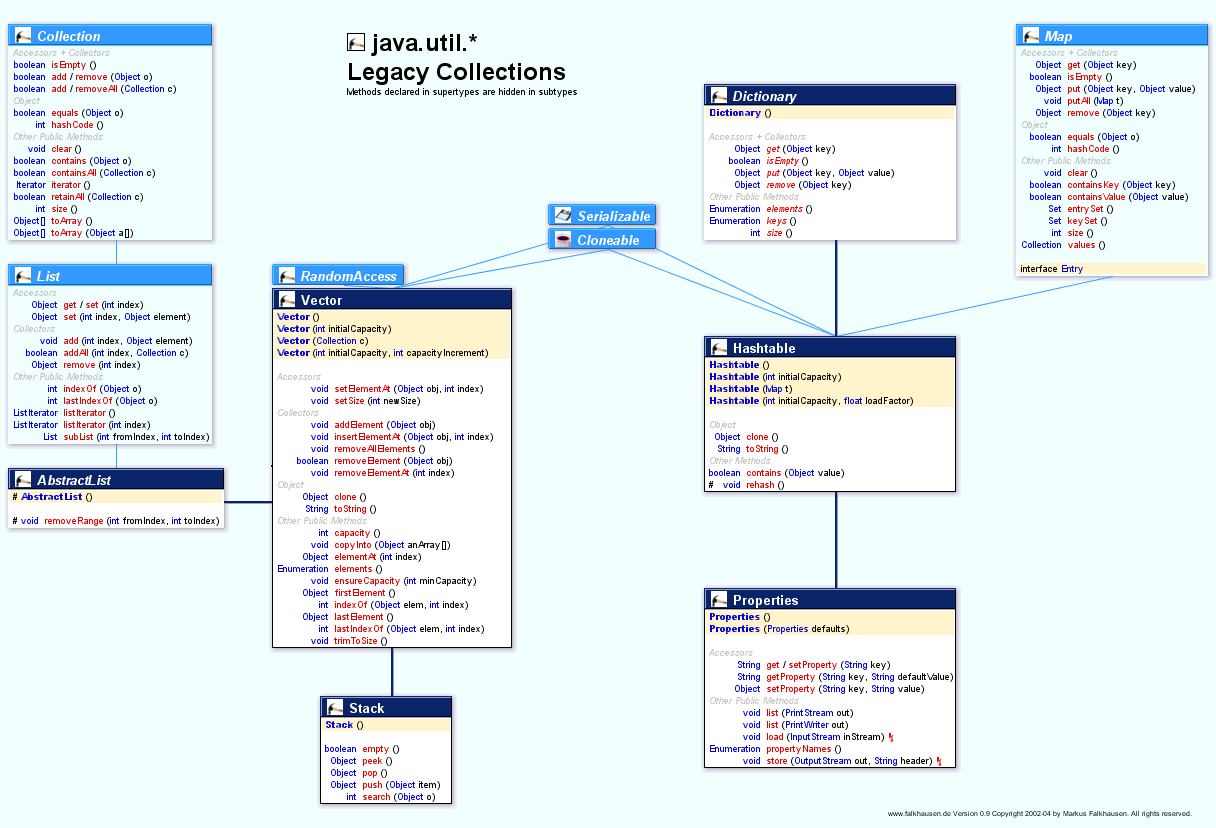
****

****

**Map-**

****

**Legacy Collection-**

****

**List Interface-**

* Elements can be inserted or accessed by their position in the list, using a zero-based index. Hence List is ordered collection.
* A list may contain duplicate elements.
* List has 3 implementions –
  + ArrayList
  + LinkedList
  + Vector(Legacy)

**ArrayList<E> class-**

ArrayList is implemented using dynamic arrays that can grow and shrink as needed.

Standard Java arrays are of a fixed length. After arrays are created, they cannot grow or shrink, which means that you must know in advance how many elements an array will hold.

Array lists are created with an initial size. When this size is exceeded, the collection is automatically enlarged. When objects are removed, the array may be shrunk.

Default capacity: 10;

When ArrayList is become full and additional element(s) needs to be inserted, A new array ( size = initialCapacity + initialCapacity/2 ) is created and all the data resides in this newly created array. So it concludes that ArrayList increases it’s capacity by 50%, when required.

ArrayList is *NOT* synchronized by default, But it can be used in synchronized manner as follows –

*List list = new ArrayList();*

*list = Collections.synchronizedList();*

ArrayList has following 3 constructors-

*ArrayList()*

*ArrayList(Collection c)*

*ArrayList(int capacity)*

**Vector class-**

Vector is same as ArrayList.And similar to ArrayList, Vector is also implemented using dynamic arrays that can grow as needed.

The difference between ArrayList & Vector is that Vector is by default thread-safe/synchronized (Hence slower than ArrayList). And Vector provides an extra constructor (look at 4th constructor as listed below) to define incrementOffset.

Vector has following 4 constructors-

Vector()

Vector(Collection c)

Vector(int Capacity)

Vector(int capacity, int incrementOffset) // Vector capacity will be increased by *incrementOffset. E.g.* newCapacity = initialCapacity *+* incrementOffset;

**LinkedList<E> class-**

LinkedList offers similar functionality as ArrayList & Vector. But unlike ArrayList & Vector which are implemented by using dynamic arrays, LinkedList is implemented by using doubly-linked-list. As the elements are doubly-linked, It provides additional methods for adding/remove elements from beginning/end, which makes it best choice to implement stack and queue.

LinkedList also implements java.util.Queue interface hence contains some additional methods for performing queue operations.

LinkedList is *NOT* synchronized by default, But it can be used in synchronized manner as follows –

*List list = new LinkedList();*

*list = Collections.synchronizedList();*

LinkedList has following 2 constructors-

LinkedList()

LinkedList(Collection c)

**ArrayList vs Vector-**

|  |  |
| --- | --- |
| ArrayList | Vector |
| ArrayList is Not synchronized by default. But can be used in synchronized manner. | Vector is by default synchronized. |
| Elements can be retrieved using Iterator only. | Elements can be retrieved using Iterator & Enumeration both. |
| Capacity of ArrayList is increased by 50% when required. And incrementOffset is not customizable. | In Vector, incrementOffset can be supplied through constructor. |

**ArrayList vs LinkedList-**

|  |  |
| --- | --- |
| ArrayList | LinkedList |
| ArrayList is implemented using dynamic array which grows and shrinks. | LinkedList is implemented using doubly-linked-list. |
| Manipulation with ArrayList is ****slow**** because it internally uses array. If any element is removed from the array, all the bits are shifted in memory. | Manipulation with LinkedList is **faster** than ArrayList because it uses doubly linked list so no bit shifting is required in memory. |
| ArrayList can acts as List only as it implements List interface. | LinkedList implements List & Queue interfaces hence it can acts as List & queue. |

**Set Interface-**

Set interface ensures that duplicate elements are not stored. Which means that only one null value can be added to set. Set does not define any new method, It inherits common methods from Collection interface. Set is implemented by using hashing technique.

Following 3 classes implements Set interface –

1. HashSet<E>
2. LinkedHashSet<E>
3. TreeSet<E> (Child of SortedSet interface)

**HashSet<E> class-**

Only unique elements are stored, And no order is maintained.

A HashSet relies on the implementation of hashCode() & equals() methods of it’s elements. If you want 2 elements(objects) to be treated as equal to each other then their hash codes must be equal. And equals() should return true. For example if 2 objects ob1 and ob2 are equal then following 2 conditions must hold true-

1. ob1.hasCode() == ob2.hashCode();
2. ob1 and ob2 are equal according to equals() method.

It has following contructors-

HashSet();

HashSet(Collection c);

HashSet(int initialCapacity);

HashSet(int initialCapacity, float loadFactor);

**LinkedHashSet<E> class-**

LinkedHashSet extends HashSet and works similar to HashSet. But there is one major difference between the two, LinkedHashSet is ordered. And Elements are retrieved in insertion order in case of LinkedHashSet. Like HashSet, LinkedHashSet also doesn’t allow duplicate elements.

It has constructor analogous to the ones in HashSet class.

Before moving on to SortedSet, We must go through Comparable(natural ordering) & Comparator(total ordering) first.

**The Comparable<E> Interface-**

The *natural ordering* of objects/elements is specified by implementing Comparable interface.

A *total ordering* is specified by implementing Comparator interface.

Comparable interface has only one method-

int compareTo(E elt);

It returns negative, Zero or a positive integer if the current object is less than, equal to, or greater than the specified object, based on the natural ordering.

Objects implementing Comparable can be used as-

1. Elements in a sorted set.
2. Keys in sorted map.
3. Elements in the list that are sorted manually using the Collections.sort() method.

Many of the standard classes in Java API, such as the primitive wrapper classes, String, Date, File implements Comparable.

Natural ordering for String objects and Character objects is lexicographical ordering. i.e., their comparison is based on the Unicode Value of each character in the strings. Objects of String and Character classes will be lexicographically maintained as elements in a sorted set, or as keys in a sorted map that uses their natural ordering.

The natural ordering for objects of numerical wrapper class is in ascending order of the values of corresponding numerical primitive type. As elements in a sorted set or as keys in sorted map that uses their natural ordering, the objects will be maintained in ascending order.

According to natural ordering for the objects of the Boolean class, A Boolean object representing the value *false* is less than a Boolean object representing the value *true.*

The compareTo() method must be consistent with equals() method. i.e. (obj1.compareTo(obj2)==0) == (obj1.equals(obj2)). This is recommended if elements has to be maintained inside sorted set or sorted maps.

The functionality of equals() is a subset of functionality of compareTo() method. The equals() method implementation can call compareTo(). Example-

*public boolean equals(Object other){*

*……….*

*return compareTo(other)==0;*

*}*

**Example-(Natural ordering)**

**import** lombok.Getter;

**import** lombok.Setter;

@Getter @Setter

**public** **class** Employee **implements** Comparable<Employee> {

Integer empId;

String empName;

**public** Employee(Integer empId, String empName){

**this**.empId = empId;

**this**.empName = empName;

}

@Override

**public** **int** compareTo(Employee otherEmp) {

**return** **this**.getEmpId().compareTo(otherEmp.getEmpId());

}

@Override

**public** **boolean** equals(Object obj) {

Employee e2 = (Employee) obj;

//System.out.println(this.empId + " "+ e2.empId);

**if**(**this**.empId==e2.empId){

**return** **true**;

}

**return** **false**;

}

@Override

**public** **int** hashCode() {

**return** **this**.empId\*100;

}

@Override

**public** String toString() {

**return** **this**.getEmpId()+"-"+**this**.getEmpName();

}

}

**import** java.util.TreeSet;

**public** **class** NaturalOrderingMain {

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

System.***out***.println("Sorting according to ascending empId.");

TreeSet<Employee> t = **new** TreeSet<Employee>();

t.add(**new** Employee(1,"Ramya"));

t.add(**new** Employee(5,"Lanka"));

t.add(**new** Employee(7,"Soni"));

t.add(**new** Employee(3,"Sachin"));

t.add(**new** Employee(2,"Suyash"));

t.add(**new** Employee(6,"Virat"));

t.add(**new** Employee(4,"Akshar"));

System.***out***.println("Sorted Treeset : "+t);

}

}

/\*OUTPUT

Sorting according to ascending empId.

Treeset : [1-Ramya, 2-Suyash, 3-Sachin, 4-Akshar, 5-Lanka, 6-Virat, 7-Soni]

\*/

**Comparator<E> Interface-**

Comparator is used to define the total ordering of elements. Comparator interface has following single method-

public int compare(E ob1, E ob2);

compare() method returns negative, zero, or a positive integer if the first object is less than, equal to, or greater than the second object. i.e. it’s contract is equivalent to compareTo() method of Comparable interface.

It’s strongly recommended that it’s implementation does not contradict the semantics of equals() method.

An alternative ordering to the default natural ordering can be specified by passing a Comparator to the constructor when the sorted set or map is created. The Collections & Arrays classes provide utility methods for sorting which take Comparator.

**Example-(Total ordering)**

**import** lombok.Getter;

**import** lombok.Setter;

@Getter @Setter

**public** **class** Employee **implements** Comparable<Employee> {

Integer empId;

String empName;

Integer empAge;

**public** Employee(Integer empId, String empName, Integer empAge){

**this**.empId = empId;

**this**.empName = empName;

**this**.empAge = empAge;

}

@Override

**public** **int** compareTo(Employee otherEmp) {

**return** **this**.getEmpId().compareTo(otherEmp.getEmpId());

}

@Override

**public** **boolean** equals(Object obj) {

Employee e2 = (Employee) obj;

//System.out.println(this.empId + " "+ e2.empId);

**if**(**this**.empId==e2.empId){

**return** **true**;

}

**return** **false**;

}

@Override

**public** **int** hashCode() {

**return** **this**.empId\*100;

}

@Override

**public** String toString() {

**return** **this**.getEmpId()+"-"+**this**.getEmpName()+"-"+**this**.getEmpAge();

}

}

**import** java.util.Comparator;

**import** java.util.TreeSet;

**public** **class** TotalOrderingMain {

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

TreeSet<Employee> t = **new** TreeSet<Employee>(

**new** Comparator<Employee>() {

@Override

**public** **int** compare(Employee e1, Employee e2) {

**return** e1.getEmpAge().compareTo(e2.getEmpAge());

}

}

);

System.***out***.println("Sorting according to ascending empAge.");

t.add(**new** Employee(1,"Ramya", 32));

t.add(**new** Employee(5,"Lanka", 64));

t.add(**new** Employee(7,"Soni", 44));

t.add(**new** Employee(3,"Sachin", 42));

t.add(**new** Employee(2,"Suyash", 22));

t.add(**new** Employee(6,"Virat", 26));

t.add(**new** Employee(4,"Akshar", 21));

System.***out***.println("Sorted Treeset : "+t);

}

}

/\*OUTPUT

Sorting according to ascending empAge.

Sorted Treeset : [4-Akshar-21, 2-Suyash-22, 6-Virat-26, 1-Ramya-32, 3-Sachin-42, 7-Soni-44, 5-Lanka-64]

\*/

**SortedSet<E> Interface-**

The SortedSet interface extends the Set interface to provide the functionality for handling sorted sets. Since the elements are sorted, traversing the set either using the for(:) loop or an iterator will access the elements according to the ordering used by the set.

Following methods are used to retrieve elements-

E first();

E last();

The first() method returns the first element currently in this sorted set, and the last() method returns the last element currently in this sorted set. The elements are chosen based on the ordering used by the sorted set.

Following methods are used to get the subset of underlying sorted set-

SortedSet<E> headSet(<E> toElement);

SortedSet<E> tailSet(<E> fromElement);

SortedSet<E> subSet(<E> fromElement, <E> toElement);

The headSet() method returns a *view* of a portion of this sorted set, whose elements are strictly less than the specified element.

Similarly, the tailSet() method returns a view of the portion of this sorted set, whose elements are greater than or equal to the specified element.

The subSet() method returns a view of the portion of this sorted set, whose elements range from fromElement, inclusive, to toElement, exclusive (also called *half-open interval*).

Note that the views present the elements sorted in the same order as the underlying sorted set. Note that changes made through views are also reflected in the underlying sorted set, and vice versa.

// Comparator access

Comparator<? super E> comparator()

This method returns the comparator associated with this sorted set, or null if it uses the natural ordering of its elements. This comparator, if defined, is used by default when a sorted set is constructed and also used when copying elements into new sorted sets.

**NavigableSet<E> Interface-**

The NavigableSet interface extends the SortedSet interface with navigation methods to find the closest matches for specific search targets. By navigation, we mean operations that require searching for elements in the navigable set.

The NavigableSet interface replaces the SortedSet interface and is the preferred choice when a sorted set is required. In addition to the methods of the SortedSet interface, the NavigableSet interface adds the following *new* methods:

Methods for removing elements-

E pollFirst();

E pollLast();

The pollFirst() method removes and returns the first element and the pollLast() method removes and returns the last element currently in this navigable set.

Methods for obtaining subsets of underlying set-

NavigableSet<E> headSet(<E> toElement, boolean inclusive)

NavigableSet<E> tailSet(<E> fromElement, boolean inclusive)

NavigableSet<E> subSet(<E> fromElement, boolean fromInclusive, E> toElement, boolean toInclusive)

These operations are analogous to the ones in the SortedSet interface, returning different views of the underlying navigable set, depending on the bound elements. However, the bound elements can be *excluded or included* by the operation, depending on the value of the boolean argument inclusive.

Methods for finding Closest-matches-

E ceiling(E e);

E floor(E e);

E higher(E e);

E lower(E e);

The method ceiling() returns the least element in the navigable set greater than or equal to argument e.

The method floor() returns the greatest element in the navigable set less than or equal to argument e.

The method higher() returns the least element in the navigable set strictly greater than argument e.

The method lower() returns the greatest element in the navigable set strictly less than argument e.

All methods return null if the required element is not found.

Reverse order

Iterator<E> descendingIterator();

NavigableSet<E> descendingSet();

The first method returns a reverse-order view of the elements in the navigable set. The second method returns a reverse-order iterator for the navigable set.

**TreeSet<E> class-**

The TreeSet class implements the NavigableSet interface and thereby the SortedSet interface. By default, operations on a sorted set rely on the natural ordering(by default ascending order) of the elements. However, a total ordering can be specified by passing a customized comparator to the constructor.

The TreeSet implementation uses balanced trees, which deliver excellent (logarithmic) performance for all operations. However, searching in a HashSet can be faster than in a TreeSet because hashing algorithms usually offer better performance than the search algorithms for balanced trees.

The TreeSet class is preferred if elements are to be maintained in sorted order and fast insertion and retrieval of individual elements is desired.

It has 4 constructors-

TreeSet();

TreeSet(Comparable<? extends E> comp);

TreeSet(Collection<? extends E> col);

TreeSet(SortedSet<E> set);

**Queue<E> Interface–**

The Queue interface basically orders the element in FIFO (First In First Out) manner.

Elements are maintained in processing order. The order in which elements can be retrieved for processing is dictated by either ‘Natural Processing’ or ‘Ordering of elements by Comparator’.

A *head* position in the queue specifies where the next element for processing can be obtained. Queue interface contains following methods –

Methods for ‘insertion’ –

1. boolean add(E object)
2. boolean offer(E object)

Note- add() throws *IllegalStateException* if queue is full, but offer() does not.

Methods for ‘removal’ –

1. E remove()
2. E poll()

Note – remove() throws *NoSuchElementException* If queue is empty, But poll() returns *null* if queue is empty.

Methods for ‘retrieval/examine’ –

1. E element()
2. E peek()

Note – Both methods retrieve the head element, But don’t remove it from the queue. If queue is empty then element() throws *NoSuchElementException*. While peek() returns *null* if queue is empty.

Both *PriorityQueue* and *LinkedList* implements Queue interface. Unless bi-directional traversal is necessary, Other queue implementations should be considered, and not the *LinkedList*

**PriorityQueue<E> -**

PriorityQueue class is used to implement queue data structure with priority ordering. The implementation is based on *priority heap,* a tree like structure that yields an element at the head of queue according to priority ordering. Priority ordering can be defined either by using “Natural Ordering” (i.e. as described by Comparable#compareto() method in underlying class) or by “Using Comparator”. If more than one elements have same priority then one of them is chosen randomly.

Elements of PriorityQueue are NOT sorted. The queue only gurantees that elements can be *removed* (using poll() or remove()) in priority order. And any traversal using an iterator does not guarantee to maintain in priority order. Hence textual representation of PriorityQueue is generated by print method running an iterator over the queue. And hence textual representation does not guarantee to print the elements in priority order.

It has following constructors –

1. PriorityQueue();

Create new empty PriorityQueue with default initial capacity natural ordering.

1. PriorityQueue(Collection<? extends E> c);

Creates a PriorityQueue with elements containing inside collection with natural ordering unless specified collection is a SortedSet or PriorityQueue, in which case, the collection’s ordering will be used.

1. PriorityQueue(int initialCapacity);

Creates new empty PriorityQueue with specified initial capacity and natural ordering.

1. PriorityQueue(int initialCapacity, Comparator<? Super E> comp);

Creates a new empty PriorityQueue with specified initialCapacity and the ordering defined by specified Comparator.

1. PriorityQueue(PriorityQueue<? extends E> pq);

Creates a new PriorityQueue with elements and ordering from specified PriorityQueue.

1. PriorityQueue(SortedSet<? extends E> set);

Creates a new PriorityQueue with elements and ordering from specified SortedSet.

**Example**-

**import** lombok.Getter;

**import** lombok.Setter;

@Getter @Setter

**public** **class** Employee **implements** Comparable<Employee> {

**private** Integer empId;

**private** String empName;

**private** Integer empAge;

**public** Employee(**int** empId, String empName, **int** empAge){

**this**.empId = empId;

**this**.empName = empName;

**this**.empAge = empAge;

}

@Override

**public** **boolean** equals(Object obj) {

Employee e2 = (Employee) obj;

//System.out.println(this.empId + " "+ e2.empId);

**if**(**this**.empId==e2.empId){

**return** **true**;

}

**return** **false**;

}

@Override

**public** **int** hashCode() {

**return** **this**.empId\*100;

}

@Override

**public** String toString() {

**return** **this**.getEmpId()+"-"+**this**.getEmpName()+"-"+**this**.getEmpAge();

}

@Override

**public** **int** compareTo(Employee emp) {

// Natural ordering based on empId.

// As objects are sorted in ascending order of empId. So Employee object with smaller empId will have higher priority.

**return** **this**.getEmpId().compareTo(emp.getEmpId());

}

}

**import** java.util.Comparator;

**import** java.util.PriorityQueue;

**public** **class** PriorityQueueMain {

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

Employee empArr[] = {**new** Employee(2, "Suyash", 22), **new** Employee(4, "Ramya", 32), **new** Employee(1, "Soni", 44), **new** Employee(3, "Lanka", 64)};

*naturalOrdering*(empArr);

*orderingUsingComparator*(empArr);

}

/\*\*

\* <p>Elements(instances of <code>Employee</code>) will be ordered and processed

\* naturally (i.e. In accordance with the sorting logic provided using <code>Comparable#compareTo()</code> )<p>

\* In this case elements will be processed in the ascending order of <code>empId</code>. Hence <Code>Employee</code> object

\* with <b>lower</b> <code>empId</code> will have <b>higher</b> priority.

\*

\* **@param** empArr - Array containing <code>Employee</code> instances.

\*/

**public** **static** **void** naturalOrdering(Employee[] empArr) {

System.***out***.println("\n----------------------- NATURAL ORDERING --------------------------\n");

// Creating PriorityQueue with natural ordering.

PriorityQueue<Employee> pQueue = **new** PriorityQueue<Employee>();

**for**(Employee e : empArr){

pQueue.add(e); // Inserting all the employees in PriorityQueue.

}

// Textual representation of priority queue does NOT show it's elements in priority order, So here employees will NOT be printed in priority order.

System.***out***.println("Priority queue before processing : "+pQueue);

// Will print the employee with highest priority(i.e. employee present at the HEAD.);

System.***out***.println("Head : "+pQueue.peek());

// Processing

// Task will be processed in natural ordering(i.e. Ascending order of empId)

System.***out***.println("Processing..........................................");

**while**(!pQueue.isEmpty()){

Employee emp = pQueue.poll();

System.***out***.print(emp+" ");

}

System.***out***.println("\n\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

}

/\*\*

\* <p>Elements(instances of <code>Employee</code>) will be ordered and processed in a order provided by using

\* <code>Comparator#compare()</code> <p>

\* In this case elements will be processed in the ascending order of <code>empAge</code>. Hence <Code>Employee</code> object

\* with <b>lower</b> <code>empAge</code> will have <b>higher</b> priority.

\*

\* **@param** empArr - Array containing <code>Employee</code> instances.

\*/

**public** **static** **void** orderingUsingComparator(Employee[] empArr) {

System.***out***.println("\n----------------------- ORDERING USING COMPARATOR --------------------------\n");

// Creating PriorityQueue with ordering provided using Comparator.

PriorityQueue<Employee> pQueue = **new** PriorityQueue<Employee>(

**new** Comparator<Employee>(){

@Override

**public** **int** compare(Employee e1, Employee e2) {

**return** e1.getEmpAge().compareTo(e2.getEmpAge());

}

}

);

**for**(Employee e : empArr){

pQueue.add(e); // Inserting all the employees in PriorityQueue.

}

// Textual representation of priority queue does NOT show it's elements in priority order, So here employees will NOT be printed in priority order.

System.***out***.println("Priority queue before processing : "+pQueue);

// Will print the employee with highest priority(i.e. employee present at the HEAD.);

System.***out***.println("Head : "+pQueue.peek());

// Processing

// Task will be processed in natural ordering(i.e. Ascending order of empId)

System.***out***.println("Processing..........................................");

**while**(!pQueue.isEmpty()){

Employee emp = pQueue.poll();

System.***out***.print(emp+" ");

}

System.***out***.println("\n\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

}

}

/\* OUTPUT -

\* ----------------------- NATURAL ORDERING --------------------------

Priority queue before processing : [1-Soni-44, 3-Lanka-64, 2-Suyash-22, 4-Ramya-32]

Head : 1-Soni-44

Processing..........................................

1-Soni-44 2-Suyash-22 3-Lanka-64 4-Ramya-32

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

----------------------- ORDERING USING COMPARATOR --------------------------

Priority queue before processing : [2-Suyash-22, 4-Ramya-32, 1-Soni-44, 3-Lanka-64]

Head : 2-Suyash-22

Processing..........................................

2-Suyash-22 4-Ramya-32 1-Soni-44 3-Lanka-64

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*/

**Deque<E> Interface -**

Deque interface extends Queue interface to allow *doubly-ended* queues. Such queues are called *deque*. Where operations are allowed from both the ends(head & tail). Hence elements can be inserted and remove from the either end.

A deque can be used as FIFO as well as LIFO. Some common methods present in this interface as given below-

Methods to insert-

boolean offerFirst(E element)

boolean offerLast(E element) // Queue equivalent : offer()

void push(E element) // Synonym: addFirst()

void addFirst(E element)

void addLast(E element) // Queue equivalent : add()

Methods to remove-

E pollFirst() // Queue equivalent : poll()

EpollLast()

E pop() // Synonym: removeFirst()

E removeFirst() // Queue equivalent : remove()

E removeLast()

boolean removeFirstOccurence(Object ob)

boolean removeLastOccurence(Object ob)

Methods to examine/retrieval-

E peekFirst() // Queue equivalent : peek()

E peekLast()

E getFirst() // Queue equivalent : element()

E getLast()

ArrayDeque and LinkedList implents Deque interface. ArrayDeque provides much better performace as comapare to LinkedList for implemting FIFO(e.g. queue) and LIFO(e.g. stack).

**ArrayDeque<E>**

ArrayDeque facilitates traversal in both the direction. descendingIterator() method is used to perform traversal in reverse direction.

It has following constructors-

ArrayDeque();

ArrayDeque(Collection<? extends E> col);

ArrayDeque(int numElements);

**Example –**

**import** java.util.ArrayDeque;

**import** java.util.Arrays;

**public** **class** ArrayDequeMain {

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

String[] empArr = {"Suyash", "Ramya", "Soni", "Lanka"};

*arrayDequeAsStackLIFO*(empArr);

*arrayDequeAsQueueFIFO*(empArr);

}

// Stack : LIFO ; Insert and remove from tail.

**public** **static** **void** arrayDequeAsStackLIFO(String [] empArr){

System.***out***.println("----------------- STACK : LIFO -----------------");

System.***out***.println("Array elements : "+Arrays.*toString*(empArr));

ArrayDeque<String> stack = **new** ArrayDeque<String>();

**for**(String empStr : empArr){

stack.push(empStr);

}

System.***out***.println("Stack: TOP->"+stack);

System.***out***.print("Popping stack : ");

**while**(!stack.isEmpty()){

String poppedElt = stack.pop();

System.***out***.print(poppedElt+" ");

}

System.***out***.println("\n\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

}

// Queue : FIFO ; Insert at tail, remove from head

**public** **static** **void** arrayDequeAsQueueFIFO(String [] empArr){

System.***out***.println("----------------- QUEUE : FIFO -----------------");

System.***out***.println("Array elements : "+Arrays.*toString*(empArr));

ArrayDeque<String> queue = **new** ArrayDeque<String>();

**for**(String empStr : empArr){

queue.offerLast(empStr);

}

System.***out***.println("Queue: HEAD->"+queue+"<-TAIL");

System.***out***.print("Polling queue : ");

**while**(!queue.isEmpty()){

String polledElt = queue.pollFirst();

System.***out***.print(polledElt+" ");

}

System.***out***.println("\n\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

}

}

/\*

\* OUTPUT-

----------------- STACK : LIFO -----------------

Array elements : [Suyash, Ramya, Soni, Lanka]

Stack: TOP->[Lanka, Soni, Ramya, Suyash]

Popping stack : Lanka Soni Ramya Suyash

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

----------------- QUEUE : FIFO -----------------

Array elements : [Suyash, Ramya, Soni, Lanka]

Queue: HEAD->[Suyash, Ramya, Soni, Lanka]<-TAIL

Polling queue : Suyash Ramya Soni Lanka

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\*/

**Map<K, V> Interface-**

A Map defines *mappings* from keys to values. The <*key*, *value*> pair is called an *entry*.A map does not allow duplicate keys.

Both the keys and the values must be objects, with primitive values being wrapped in their respective primitive wrapper objects when they are put in a map.

A map is not a collection and the Map interface does not extend the Collection interface.

**Collection Views**

Views allow information in a map to be represented as collections. Following methods provide different views of a map-

Set<K> keySet(); // return a set view of keys,

Collection<V> values(); // a collection view of values,

Set<Map, Entry<K, V>> entrySet(); // a set view of <*key*, *value*> entries

These methods provide different views of a map. Changes in the map are reflected in the view, and vice versa.

Each <*key*, *value*> in the entry set view is represented by an object implementing the nested *Map.Entry* interface. An entry in the entry set view can be manipulated by methods defined in this interface, which are self-explanatory:

*interface Entry<K, V> {*

*K getKey();*

*V getValue();*

*V setValue(V value);*

*}*

**Map Implementations-**

**The HashMap<K,V>, LinkedHashMap<K,V>, and Hashtable<K,V> Classes-**

HashTable is legacy class hence it’s synchronized(thread-safe). Permits *non-null* keys and values.

HashMap, LinkedHashMap are not thread-safe. But can be used in thread-safe manner. Permits only one *null* key.

LinkedHashMap is ordered map while HashMap & HashTable are unordered map.

These map implementations are based on a hashing algorithm. Operations on a map thus rely on the hashCode() and equals() methods of the key objects.

The LinkedHashMap implementation is a subclass of the HashMap class. The relationship between the map classes LinkedHashMap and HashMap is analogous to the relationship between their counterpart set classes LinkedHashSet and HashSet. Elements of a HashMap (and a HashSet) are unordered.

The elements of a LinkedHashMap (and a LinkedHashSet) are ordered. By default, the entries of a LinkedHashMap are in *key insertion order*, that is, the order in which the keys are inserted in the map. However, a LinkedHashMap can also maintain its elements in (element) *access* *order*, that is, the order in which its entries are accessed, from least-recently accessed to most-recently accessed entries. This *ordering mode* can be specified in one of the constructors of the LinkedHashMap class.

For HashMap, Operations, such as adding, removing, or finding an entry based on a key are in constant time, as these hash the key. Operations such as finding the entry with a particular value are in linear time, as these involve searching through the entries.

For LinkedHashMap, Adding, removing, and finding entries can be slightly slower than in a HashMap, as an ordered doubly-linked list has to be maintained. Traversal of a map is through one of its collection-views. For an underlying LinkedHashMap, the traversal time is proportional to the size of the map—regardless of its capacity.

However, for an underlying HashMap, it is proportional to the capacity of the map.

The concrete map implementations override the toString() method. The standard textual representation generated by the toString() method for a map is

{*key1*=*value1*, *key2*=*value2*, ..., *keyn*=*valuen*}

HashMap has following contructors-

HashMap()

HashMap(int initialCapacity)

HashMap(int initialCapacity, float loadFactor)

HashMap(Map<? extends K,? extends V> otherMap)

The LinkedHashMap and Hashtable classes have constructors analogous to the four constructors for the HashMap class. In addition, the LinkedHashMap class provides a constructor where the ordering mode can also be specified:

LinkedHashMap(int initialCapacity, float loadFactor, boolean accessOrder);

Constructs a new, empty LinkedHashMap with the specified initial capacity, the specified load factor, and the specified ordering mode. The ordering mode is true for *access order* and false for *key insertion order*.

**SortedMap<K,V> and NavigableMap<K,V> Interfaces-**

The SortedMap and NavigableMap interfaces are the analogous of the SortedSet and the NavigableSet interfaces, respectively.

**SortedMap<K,V> Interface-**

The SortedMap interface extends the Map interface to provide the functionality for implementing maps with *sorted keys*. Its operations are analogous to those of the SortedSet interface, applied to maps and keys rather than to sets and elements.

Methods to obtain(first(smallest) key and last(greatest) key)-

K firstKey() Sorted set: first() // SortedSet equivalent : first()

K lastKey() Sorted set: last() // SortedSet equivalent : last()

Methods for Range-view operations-

SortedMap<K,V> headMap(K toKey) // SortedSet equivalent : headSet()

SortedMap<K,V> tailMap(K fromKey) // SortedSet equivalent : tailSet()

SortedMap<K,V> subMap(K fromKey, K toKey) // SortedSet equivalent : subSet()

Return set views analogous to that of a SortedSet. The set views include the fromKey if it is present in the map, but the toKey is excluded.

Method to obtaing Comparator associated-

Comparator<? super K> comparator();

Returns the key comparator, if the map has one. Otherwise returns null.

**NavigableMap<K,V> Interface-**

Analogous to the NavigableSet interface extending the SortedSet interface, the NavigableMap interface extends the SortedMap interface with navigation methods to find the closest matches for specific search targets. The NavigableMap interface replaces the SortedMap interface and is the preferred choice when a sorted map is needed.

In addition to the methods of the SortedMap interface, the NavigableMap interface adds the *new* methods shown below, where the analogous methods from the NavigableSet interface are also identified. Note that where a NavigableMap method returns a Map.Entry object representing a mapping, the corresponding NavigableSet method returns an element of the set.

// First-last elements

Map.Entry<K, V> pollFirstEntry() // Navigable set: pollFirst()

Map.Entry<K, V> pollLastEntry() // Navigable set: pollLast()

Map.Entry<K, V> firstEntry()

Map.Entry<K, V> lastEntry()

The pollFirstEntry() method removes and returns the first *entry*, and the pollLastEntry() method removes and returns the last *entry* currently in this navigable map.

The entry is determined according to the ordering policy employed by the map—e.g., natural ordering. Both return null if the navigable set is empty.

firstEntry() & lastEntry() methods only retrieve, and do not remove, the value that is returned.

Methods for Range-view operations-

NavigableMap<K, V> headMap(K toElement, boolean inclusive) // Navigable set: headSet()

NavigableMap<K, V> tailMap(K fromElement, boolean inclusive) // Navigable set: tailSet()

NavigableMap<K, V> subMap(K fromElement, boolean fromInclusive, K toElement, boolean toInclusive) // Navigable set: subSet()

Methods for obtaining Closest-matches-

Map.Entry<K, V> ceilingEntry(K key); // Navigable set: ceiling()

K ceilingKey(K key);

Map.Entry<K, V> floorEntry(K key); // Navigable set: floor()

K floorKey(K key);

Map.Entry<K, V> higherEntry(K key); // Navigable set: higher()

K higherKey(K key);

Map.Entry<K, V> lowerEntry(K key); // Navigable set: lower()

K lowerKey(K key);

The ceiling methods return the least entry (or key) in the navigable map >= to the argument key.

The floor methods return the greatest entry (or key) in the navigable map <= to the argument key.

The higher methods return the least entry (or key) in the navigable map > the argument key.

The lower methods return the greatest entry (or key) in the navigable map < the argument key. All methods return null if there is no such key.

Methods for Navigation-

NavigableMap<K, V> descendingMap() // Navigable set: descendingSet()

NavigableSet<K> descendingKeySet()

NavigableSet<K> navigableKeySet()

The first method returns a reverse-order view of the entries in the navigable map.

The second method returns a reverse-order key set for the entries in the navigable map.

The last method returns a forward-order key set for the entries in the navigable map.

**TreeMap<K,V> Class-**

The TreeMap class is the analogous of the TreeSet class, but in this case for maps. It provides an implementation that sorts its entries in a specific order.

The TreeMap class implements the NavigableMap interface, and thereby the SortedMap interface. By default, operations on sorted maps rely on the natural ordering of the keys. However, a total ordering can be specified by passing a customized comparator to the constructor.

The TreeMap implementation uses balanced trees, which deliver excellent performance for all operations. However, searching in a HashMap can be faster than in a TreeMap, as hashing algorithms usually offer better performance than the search algorithms for balanced trees.

The TreeMap class provides four constructors, analogous to the ones in the TreeSet class:

TreeMap();

TreeMap(Comparator<? super K> c)

TreeMap(Map<? extends K, ? extends V> m)

TreeMap(SortedMap<K, ? extends V> m)

**Common usage of *Java.util.Collections* class-**

1. To get a synchronized(thread-safe) copy of collection.
2. To get a read-only(unmodifiable) copy of collection.

|  |  |
| --- | --- |
| static <T> boolean | [**addAll**](http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#addAll(java.util.Collection,%20T...))([**Collection**](http://docs.oracle.com/javase/7/docs/api/java/util/Collection.html)<? super T> c, T... elements)  Adds all of the specified elements to the specified collection. |
| static <T> [**Queue**](http://docs.oracle.com/javase/7/docs/api/java/util/Queue.html)<T> | [**asLifoQueue**](http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#asLifoQueue(java.util.Deque))([**Deque**](http://docs.oracle.com/javase/7/docs/api/java/util/Deque.html)<T> deque)  Returns a view of a [**Deque**](http://docs.oracle.com/javase/7/docs/api/java/util/Deque.html) as a Last-in-first-out (Lifo) [**Queue**](http://docs.oracle.com/javase/7/docs/api/java/util/Queue.html). |
| static <T> int | [**binarySearch**](http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#binarySearch(java.util.List,%20T))([**List**](http://docs.oracle.com/javase/7/docs/api/java/util/List.html)<? extends [**Comparable**](http://docs.oracle.com/javase/7/docs/api/java/lang/Comparable.html)<? super T>> list, T key)  Searches the specified list for the specified object using the binary search algorithm. |
| static <T> int | [**binarySearch**](http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#binarySearch(java.util.List,%20T,%20java.util.Comparator))([**List**](http://docs.oracle.com/javase/7/docs/api/java/util/List.html)<? extends T> list, T key, [**Comparator**](http://docs.oracle.com/javase/7/docs/api/java/util/Comparator.html)<? super T> c)  Searches the specified list for the specified object using the binary search algorithm. |
| static <E> [**Collection**](http://docs.oracle.com/javase/7/docs/api/java/util/Collection.html)<E> | [**checkedCollection**](http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#checkedCollection(java.util.Collection,%20java.lang.Class))([**Collection**](http://docs.oracle.com/javase/7/docs/api/java/util/Collection.html)<E> c, [**Class**](http://docs.oracle.com/javase/7/docs/api/java/lang/Class.html)<E> type)  Returns a dynamically typesafe view of the specified collection. |
| static <E> [**List**](http://docs.oracle.com/javase/7/docs/api/java/util/List.html)<E> | [**checkedList**](http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#checkedList(java.util.List,%20java.lang.Class))([**List**](http://docs.oracle.com/javase/7/docs/api/java/util/List.html)<E> list, [**Class**](http://docs.oracle.com/javase/7/docs/api/java/lang/Class.html)<E> type)  Returns a dynamically typesafe view of the specified list. |
| static <K,V> [**Map**](http://docs.oracle.com/javase/7/docs/api/java/util/Map.html)<K,V> | [**checkedMap**](http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#checkedMap(java.util.Map,%20java.lang.Class,%20java.lang.Class))([**Map**](http://docs.oracle.com/javase/7/docs/api/java/util/Map.html)<K,V> m, [**Class**](http://docs.oracle.com/javase/7/docs/api/java/lang/Class.html)<K> keyType, [**Class**](http://docs.oracle.com/javase/7/docs/api/java/lang/Class.html)<V> valueType)  Returns a dynamically typesafe view of the specified map. |
| static <E> [**Set**](http://docs.oracle.com/javase/7/docs/api/java/util/Set.html)<E> | [**checkedSet**](http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#checkedSet(java.util.Set,%20java.lang.Class))([**Set**](http://docs.oracle.com/javase/7/docs/api/java/util/Set.html)<E> s, [**Class**](http://docs.oracle.com/javase/7/docs/api/java/lang/Class.html)<E> type)  Returns a dynamically typesafe view of the specified set. |
| static <K,V> [**SortedMap**](http://docs.oracle.com/javase/7/docs/api/java/util/SortedMap.html)<K,V> | [**checkedSortedMap**](http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#checkedSortedMap(java.util.SortedMap,%20java.lang.Class,%20java.lang.Class))([**SortedMap**](http://docs.oracle.com/javase/7/docs/api/java/util/SortedMap.html)<K,V> m, [**Class**](http://docs.oracle.com/javase/7/docs/api/java/lang/Class.html)<K> keyType, [**Class**](http://docs.oracle.com/javase/7/docs/api/java/lang/Class.html)<V> valueType)  Returns a dynamically typesafe view of the specified sorted map. |
| static <E> [**SortedSet**](http://docs.oracle.com/javase/7/docs/api/java/util/SortedSet.html)<E> | [**checkedSortedSet**](http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#checkedSortedSet(java.util.SortedSet,%20java.lang.Class))([**SortedSet**](http://docs.oracle.com/javase/7/docs/api/java/util/SortedSet.html)<E> s, [**Class**](http://docs.oracle.com/javase/7/docs/api/java/lang/Class.html)<E> type)  Returns a dynamically typesafe view of the specified sorted set. |
| static <T> void | [**copy**](http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#copy(java.util.List,%20java.util.List))([**List**](http://docs.oracle.com/javase/7/docs/api/java/util/List.html)<? super T> dest, [**List**](http://docs.oracle.com/javase/7/docs/api/java/util/List.html)<? extends T> src)  Copies all of the elements from one list into another. |
| static boolean | [**disjoint**](http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#disjoint(java.util.Collection,%20java.util.Collection))([**Collection**](http://docs.oracle.com/javase/7/docs/api/java/util/Collection.html)<?> c1, [**Collection**](http://docs.oracle.com/javase/7/docs/api/java/util/Collection.html)<?> c2)  Returns true if the two specified collections have no elements in common. |
| static <T> [**Enumeration**](http://docs.oracle.com/javase/7/docs/api/java/util/Enumeration.html)<T> | [**emptyEnumeration**](http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#emptyEnumeration())()  Returns an enumeration that has no elements. |
| static <T> [**Iterator**](http://docs.oracle.com/javase/7/docs/api/java/util/Iterator.html)<T> | [**emptyIterator**](http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#emptyIterator())()  Returns an iterator that has no elements. |
| static <T> [**List**](http://docs.oracle.com/javase/7/docs/api/java/util/List.html)<T> | [**emptyList**](http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#emptyList())()  Returns the empty list (immutable). |
| static <T> [**ListIterator**](http://docs.oracle.com/javase/7/docs/api/java/util/ListIterator.html)<T> | [**emptyListIterator**](http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#emptyListIterator())()  Returns a list iterator that has no elements. |
| static <K,V> [**Map**](http://docs.oracle.com/javase/7/docs/api/java/util/Map.html)<K,V> | [**emptyMap**](http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#emptyMap())()  Returns the empty map (immutable). |
| static <T> [**Set**](http://docs.oracle.com/javase/7/docs/api/java/util/Set.html)<T> | [**emptySet**](http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#emptySet())()  Returns the empty set (immutable). |
| static <T> [**Enumeration**](http://docs.oracle.com/javase/7/docs/api/java/util/Enumeration.html)<T> | [**enumeration**](http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#enumeration(java.util.Collection))([**Collection**](http://docs.oracle.com/javase/7/docs/api/java/util/Collection.html)<T> c)  Returns an enumeration over the specified collection. |
| static <T> void | [**fill**](http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#fill(java.util.List,%20T))([**List**](http://docs.oracle.com/javase/7/docs/api/java/util/List.html)<? super T> list, T obj)  Replaces all of the elements of the specified list with the specified element. |
| static int | [**frequency**](http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#frequency(java.util.Collection,%20java.lang.Object))([**Collection**](http://docs.oracle.com/javase/7/docs/api/java/util/Collection.html)<?> c, [**Object**](http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html) o)  Returns the number of elements in the specified collection equal to the specified object. |
| static int | [**indexOfSubList**](http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#indexOfSubList(java.util.List,%20java.util.List))([**List**](http://docs.oracle.com/javase/7/docs/api/java/util/List.html)<?> source, [**List**](http://docs.oracle.com/javase/7/docs/api/java/util/List.html)<?> target)  Returns the starting position of the first occurrence of the specified target list within the specified source list, or -1 if there is no such occurrence. |
| static int | [**lastIndexOfSubList**](http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#lastIndexOfSubList(java.util.List,%20java.util.List))([**List**](http://docs.oracle.com/javase/7/docs/api/java/util/List.html)<?> source, [**List**](http://docs.oracle.com/javase/7/docs/api/java/util/List.html)<?> target)  Returns the starting position of the last occurrence of the specified target list within the specified source list, or -1 if there is no such occurrence. |
| static <T> [**ArrayList**](http://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html)<T> | [**list**](http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#list(java.util.Enumeration))([**Enumeration**](http://docs.oracle.com/javase/7/docs/api/java/util/Enumeration.html)<T> e)  Returns an array list containing the elements returned by the specified enumeration in the order they are returned by the enumeration. |
| static <T extends [**Object**](http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html) & [**Comparable**](http://docs.oracle.com/javase/7/docs/api/java/lang/Comparable.html)<? super T>>  T | [**max**](http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#max(java.util.Collection))([**Collection**](http://docs.oracle.com/javase/7/docs/api/java/util/Collection.html)<? extends T> coll)  Returns the maximum element of the given collection, according to the *natural ordering* of its elements. |
| static <T> T | [**max**](http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#max(java.util.Collection,%20java.util.Comparator))([**Collection**](http://docs.oracle.com/javase/7/docs/api/java/util/Collection.html)<? extends T> coll, [**Comparator**](http://docs.oracle.com/javase/7/docs/api/java/util/Comparator.html)<? super T> comp)  Returns the maximum element of the given collection, according to the order induced by the specified comparator. |
| static <T extends [**Object**](http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html) & [**Comparable**](http://docs.oracle.com/javase/7/docs/api/java/lang/Comparable.html)<? super T>>  T | [**min**](http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#min(java.util.Collection))([**Collection**](http://docs.oracle.com/javase/7/docs/api/java/util/Collection.html)<? extends T> coll)  Returns the minimum element of the given collection, according to the *natural ordering* of its elements. |
| static <T> T | [**min**](http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#min(java.util.Collection,%20java.util.Comparator))([**Collection**](http://docs.oracle.com/javase/7/docs/api/java/util/Collection.html)<? extends T> coll, [**Comparator**](http://docs.oracle.com/javase/7/docs/api/java/util/Comparator.html)<? super T> comp)  Returns the minimum element of the given collection, according to the order induced by the specified comparator. |
| static <T> [**List**](http://docs.oracle.com/javase/7/docs/api/java/util/List.html)<T> | [**nCopies**](http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#nCopies(int,%20T))(int n, T o)  Returns an immutable list consisting of n copies of the specified object. |
| static <E> [**Set**](http://docs.oracle.com/javase/7/docs/api/java/util/Set.html)<E> | [**newSetFromMap**](http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#newSetFromMap(java.util.Map))([**Map**](http://docs.oracle.com/javase/7/docs/api/java/util/Map.html)<E,[**Boolean**](http://docs.oracle.com/javase/7/docs/api/java/lang/Boolean.html)> map)  Returns a set backed by the specified map. |
| static <T> boolean | [**replaceAll**](http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#replaceAll(java.util.List,%20T,%20T))([**List**](http://docs.oracle.com/javase/7/docs/api/java/util/List.html)<T> list, T oldVal, T newVal)  Replaces all occurrences of one specified value in a list with another. |
| static void | [**reverse**](http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#reverse(java.util.List))([**List**](http://docs.oracle.com/javase/7/docs/api/java/util/List.html)<?> list)  Reverses the order of the elements in the specified list. |
| static <T> [**Comparator**](http://docs.oracle.com/javase/7/docs/api/java/util/Comparator.html)<T> | [**reverseOrder**](http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#reverseOrder())()  Returns a comparator that imposes the reverse of the *natural ordering* on a collection of objects that implement the Comparable interface. |
| static <T> [**Comparator**](http://docs.oracle.com/javase/7/docs/api/java/util/Comparator.html)<T> | [**reverseOrder**](http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#reverseOrder(java.util.Comparator))([**Comparator**](http://docs.oracle.com/javase/7/docs/api/java/util/Comparator.html)<T> cmp)  Returns a comparator that imposes the reverse ordering of the specified comparator. |
| static void | [**rotate**](http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#rotate(java.util.List,%20int))([**List**](http://docs.oracle.com/javase/7/docs/api/java/util/List.html)<?> list, int distance)  Rotates the elements in the specified list by the specified distance. |
| static void | [**shuffle**](http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#shuffle(java.util.List))([**List**](http://docs.oracle.com/javase/7/docs/api/java/util/List.html)<?> list)  Randomly permutes the specified list using a default source of randomness. |
| static void | [**shuffle**](http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#shuffle(java.util.List,%20java.util.Random))([**List**](http://docs.oracle.com/javase/7/docs/api/java/util/List.html)<?> list, [**Random**](http://docs.oracle.com/javase/7/docs/api/java/util/Random.html) rnd)  Randomly permute the specified list using the specified source of randomness. |
| static <T> [**Set**](http://docs.oracle.com/javase/7/docs/api/java/util/Set.html)<T> | [**singleton**](http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#singleton(T))(T o)  Returns an immutable set containing only the specified object. |
| static <T> [**List**](http://docs.oracle.com/javase/7/docs/api/java/util/List.html)<T> | [**singletonList**](http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#singletonList(T))(T o)  Returns an immutable list containing only the specified object. |
| static <K,V> [**Map**](http://docs.oracle.com/javase/7/docs/api/java/util/Map.html)<K,V> | [**singletonMap**](http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#singletonMap(K,%20V))(K key, V value)  Returns an immutable map, mapping only the specified key to the specified value. |
| static <T extends [**Comparable**](http://docs.oracle.com/javase/7/docs/api/java/lang/Comparable.html)<? super T>>  void | [**sort**](http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#sort(java.util.List))([**List**](http://docs.oracle.com/javase/7/docs/api/java/util/List.html)<T> list)  Sorts the specified list into ascending order, according to the [**natural ordering**](http://docs.oracle.com/javase/7/docs/api/java/lang/Comparable.html) of its elements. |
| static <T> void | [**sort**](http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#sort(java.util.List,%20java.util.Comparator))([**List**](http://docs.oracle.com/javase/7/docs/api/java/util/List.html)<T> list, [**Comparator**](http://docs.oracle.com/javase/7/docs/api/java/util/Comparator.html)<? super T> c)  Sorts the specified list according to the order induced by the specified comparator. |
| static void | [**swap**](http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#swap(java.util.List,%20int,%20int))([**List**](http://docs.oracle.com/javase/7/docs/api/java/util/List.html)<?> list, int i, int j)  Swaps the elements at the specified positions in the specified list. |
| static <T> [**Collection**](http://docs.oracle.com/javase/7/docs/api/java/util/Collection.html)<T> | [**synchronizedCollection**](http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#synchronizedCollection(java.util.Collection))([**Collection**](http://docs.oracle.com/javase/7/docs/api/java/util/Collection.html)<T> c)  Returns a synchronized (thread-safe) collection backed by the specified collection. |
| static <T> [**List**](http://docs.oracle.com/javase/7/docs/api/java/util/List.html)<T> | [**synchronizedList**](http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#synchronizedList(java.util.List))([**List**](http://docs.oracle.com/javase/7/docs/api/java/util/List.html)<T> list)  Returns a synchronized (thread-safe) list backed by the specified list. |
| static <K,V> [**Map**](http://docs.oracle.com/javase/7/docs/api/java/util/Map.html)<K,V> | [**synchronizedMap**](http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#synchronizedMap(java.util.Map))([**Map**](http://docs.oracle.com/javase/7/docs/api/java/util/Map.html)<K,V> m)  Returns a synchronized (thread-safe) map backed by the specified map. |
| static <T> [**Set**](http://docs.oracle.com/javase/7/docs/api/java/util/Set.html)<T> | [**synchronizedSet**](http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#synchronizedSet(java.util.Set))([**Set**](http://docs.oracle.com/javase/7/docs/api/java/util/Set.html)<T> s)  Returns a synchronized (thread-safe) set backed by the specified set. |
| static <K,V> [**SortedMap**](http://docs.oracle.com/javase/7/docs/api/java/util/SortedMap.html)<K,V> | [**synchronizedSortedMap**](http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#synchronizedSortedMap(java.util.SortedMap))([**SortedMap**](http://docs.oracle.com/javase/7/docs/api/java/util/SortedMap.html)<K,V> m)  Returns a synchronized (thread-safe) sorted map backed by the specified sorted map. |
| static <T> [**SortedSet**](http://docs.oracle.com/javase/7/docs/api/java/util/SortedSet.html)<T> | [**synchronizedSortedSet**](http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#synchronizedSortedSet(java.util.SortedSet))([**SortedSet**](http://docs.oracle.com/javase/7/docs/api/java/util/SortedSet.html)<T> s)  Returns a synchronized (thread-safe) sorted set backed by the specified sorted set. |
| static <T> [**Collection**](http://docs.oracle.com/javase/7/docs/api/java/util/Collection.html)<T> | [**unmodifiableCollection**](http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#unmodifiableCollection(java.util.Collection))([**Collection**](http://docs.oracle.com/javase/7/docs/api/java/util/Collection.html)<? extends T> c)  Returns an unmodifiable view of the specified collection. |
| static <T> [**List**](http://docs.oracle.com/javase/7/docs/api/java/util/List.html)<T> | [**unmodifiableList**](http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#unmodifiableList(java.util.List))([**List**](http://docs.oracle.com/javase/7/docs/api/java/util/List.html)<? extends T> list)  Returns an unmodifiable view of the specified list. |
| static <K,V> [**Map**](http://docs.oracle.com/javase/7/docs/api/java/util/Map.html)<K,V> | [**unmodifiableMap**](http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#unmodifiableMap(java.util.Map))([**Map**](http://docs.oracle.com/javase/7/docs/api/java/util/Map.html)<? extends K,? extends V> m)  Returns an unmodifiable view of the specified map. |
| static <T> [**Set**](http://docs.oracle.com/javase/7/docs/api/java/util/Set.html)<T> | [**unmodifiableSet**](http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#unmodifiableSet(java.util.Set))([**Set**](http://docs.oracle.com/javase/7/docs/api/java/util/Set.html)<? extends T> s)  Returns an unmodifiable view of the specified set. |
| static <K,V> [**SortedMap**](http://docs.oracle.com/javase/7/docs/api/java/util/SortedMap.html)<K,V> | [**unmodifiableSortedMap**](http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#unmodifiableSortedMap(java.util.SortedMap))([**SortedMap**](http://docs.oracle.com/javase/7/docs/api/java/util/SortedMap.html)<K,? extends V> m)  Returns an unmodifiable view of the specified sorted map. |
| static <T> [**SortedSet**](http://docs.oracle.com/javase/7/docs/api/java/util/SortedSet.html)<T> | [**unmodifiableSortedSet**](http://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#unmodifiableSortedSet(java.util.SortedSet))([**SortedSet**](http://docs.oracle.com/javase/7/docs/api/java/util/SortedSet.html)<T> s)  Returns an unmodifiable view of the specified sorted set. |

**Q) When to use ArrayList or LinkedList ?**

**Ans)**

1. Adding new elements is pretty fast for either type of list. Inserting element to nth location in arraylist and to first location in linkedlist takes O(1).
2. For the ArrayList, doing random lookup using "get" is faster O(1), but for LinkedList O(n), it's slow. It's slow because there's no efficient way to index into the middle of a linked list. Linkedlist lookup always start from 1st location.
3. When removing elements, using ArrayList is slow. This is because all remaining elements in the underlying array of Object instances must be shifted down for each remove operation. But LinkedList is fast, because deletion can be done simply by changing a couple of links.

So an ArrayList works best for cases where you're doing random access on the list and a LinkedList works better if you're doing a lot of editing in the middle of the list.

**Q) If an Employee class is present and its objects are added in an ArrayList. Now I want the list to be sorted on the basis of the employeeID of Employee class. What are the steps?**

**Q) What do you understand by iterator fail-fast property?**

**Ans)**

Iterator fail-fast property checks for any modification in the structure of the underlying collection every time we try to get the next element. If there are any modifications found, it throws ConcurrentModificationException. All the implementations of Iterator in Collection classes are fail-fast by design except the concurrent collection classes like ConcurrentHashMap and CopyOnWriteArrayList.

**Q) What are concurrent Collection Classes?**

**Ans)**

Java 1.5 Concurrent package (java.util.concurrent) contains thread-safe collection classes that allow collections to be modified while iterating. By design Iterator implementation in java.util packages are fail-fast and throws ConcurrentModificationException. But Iterator implementation injava.util.concurrent packages are fail-safe and we can modify the collection while iterating. Some of these classes are CopyOnWriteArrayList, ConcurrentHashMap, CopyOnWriteArraySet.

**Q) What is difference between fail-fast and fail-safe?**

**Ans)**

Iterator fail-safe property work with the clone of underlying collection, hence it’s not affected by any modification in the collection. By design, all the collection classes in java.util package are fail-fast whereas collection classes in java.util.concurrent are fail-safe.  
Fail-fast iterators throw ConcurrentModificationException whereas fail-safe iterator never throws ConcurrentModificationException.

Fail-fast Iterators throws ConcurrentModificationException when one thread is iterating over collection object and other thread structurally modify Collection either by adding, removing or modifying objects on underlying collection. They are called fail-fast because they try to immediately throw Exception when they encounter failure. On the other hand fail-safe Iterators works on copy of collection instead of original collection.

**Q) What is the difference between Iterator and Enumeration?**

**Ans)**

Difference between Enumeration and iterator is Iterator has a remove() method while Enumeration doesn't. Enumeration acts as Read-only interface, because it has the methods only to traverse and fetch the objects, whereas by using Iterator we can manipulate the objects like adding and removing the objects from collection e.g. Arraylist.  
Also Iterator is more secure and safe as compared to Enumeration because it does not allow other thread to modify the collection object while some thread is iterating over it and throws ConcurrentModificationException.

**Enumeration**

hasMoreElement()

nextElement()

N/A

**Iterator**

hasNext()

next()

remove()