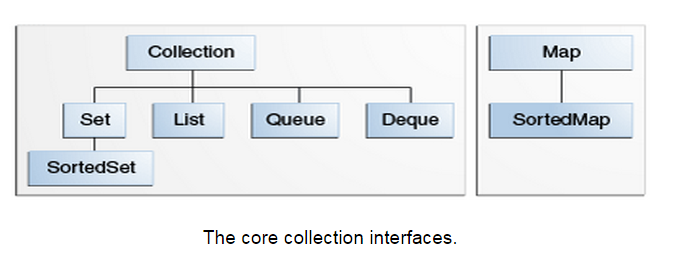
**JAVA**

**Collections**



The following list describes the core collection interfaces:

* Collection — the root of the collection hierarchy. A collection represents a group of objects known as its*elements*. The Collection interface is the least common denominator that all collections implement and is used to pass collections around and to manipulate them when maximum generality is desired. Some types of collections allow duplicate elements, and others do not. Some are ordered and others are unordered. The Java platform doesn't provide any direct implementations of this interface but provides implementations of more specific subinterfaces, such as Set and List. Also see [The Collection Interface](https://docs.oracle.com/javase/tutorial/collections/interfaces/collection.html) section.
* Set — a collection that cannot contain duplicate elements. This interface models the mathematical set abstraction and is used to represent sets, such as the cards comprising a poker hand, the courses making up a student's schedule, or the processes running on a machine. See also [The Set Interface](https://docs.oracle.com/javase/tutorial/collections/interfaces/set.html) section.
* List — an ordered collection (sometimes called a *sequence*). Lists can contain duplicate elements. The user of a List generally has precise control over where in the list each element is inserted and can access elements by their integer index (position). If you've used Vector, you're familiar with the general flavor ofList. Also see [The List Interface](https://docs.oracle.com/javase/tutorial/collections/interfaces/list.html) section.
* Queue — a collection used to hold multiple elements prior to processing. Besides basic Collectionoperations, a Queue provides additional insertion, extraction, and inspection operations.

Queues typically, but do not necessarily, order elements in a FIFO (first-in, first-out) manner. Among the exceptions are priority queues, which order elements according to a supplied comparator or the elements' natural ordering. Whatever the ordering used, the head of the queue is the element that would be removed by a call to remove or poll. In a FIFO queue, all new elements are inserted at the tail of the queue. Other kinds of queues may use different placement rules. Every Queue implementation must specify its ordering properties. Also see [The Queue Interface](https://docs.oracle.com/javase/tutorial/collections/interfaces/queue.html) section.

* Deque — a collection used to hold multiple elements prior to processing. Besides basic Collectionoperations, a Deque provides additional insertion, extraction, and inspection operations.  The name *deque* is short for "double ended queue"

Deques can be used both as FIFO (first-in, first-out) and LIFO (last-in, first-out). In a deque all new elements can be inserted, retrieved and removed at both ends. Also see [The Deque Interface](https://docs.oracle.com/javase/tutorial/collections/interfaces/deque.html) section.

[**peek**](http://docs.oracle.com/javase/7/docs/api/java/util/Deque.html#peek())()

Retrieves, but does not remove, the head of the queue represented by this deque (in other words, the first element of this deque), or returns null if this deque is empty.

[**poll**](http://docs.oracle.com/javase/7/docs/api/java/util/Deque.html#poll())()

Retrieves and removes the head of the queue represented by this deque (in other words, the first element of this deque), or returns null if this deque is empty.

[**offer**](http://docs.oracle.com/javase/7/docs/api/java/util/Deque.html#offer(E))([**E**](http://docs.oracle.com/javase/7/docs/api/java/util/Deque.html) e)

Inserts the specified element into the queue represented by this deque (in other words, at the tail of this deque) if it is possible to do so immediately without violating capacity restrictions, returning true upon success and falseif no space is currently available.

* Map — an object that maps keys to values. A Map cannot contain duplicate keys; each key can map to at most one value. If you've used Hashtable, you're already familiar with the basics of Map. Also see [The Map Interface](https://docs.oracle.com/javase/tutorial/collections/interfaces/map.html) section.

**All Known Implementing Classes:**

[AbstractMap](http://docs.oracle.com/javase/7/docs/api/java/util/AbstractMap.html), [Attributes](http://docs.oracle.com/javase/7/docs/api/java/util/jar/Attributes.html), [AuthProvider](http://docs.oracle.com/javase/7/docs/api/java/security/AuthProvider.html), [ConcurrentHashMap](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentHashMap.html), [ConcurrentSkipListMap](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentSkipListMap.html), [EnumMap](http://docs.oracle.com/javase/7/docs/api/java/util/EnumMap.html), [HashMap](http://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html), [Hashtable](http://docs.oracle.com/javase/7/docs/api/java/util/Hashtable.html), [IdentityHashMap](http://docs.oracle.com/javase/7/docs/api/java/util/IdentityHashMap.html), [LinkedHashMap](http://docs.oracle.com/javase/7/docs/api/java/util/LinkedHashMap.html),[PrinterStateReasons](http://docs.oracle.com/javase/7/docs/api/javax/print/attribute/standard/PrinterStateReasons.html), [Properties](http://docs.oracle.com/javase/7/docs/api/java/util/Properties.html), [Provider](http://docs.oracle.com/javase/7/docs/api/java/security/Provider.html), [RenderingHints](http://docs.oracle.com/javase/7/docs/api/java/awt/RenderingHints.html), [SimpleBindings](http://docs.oracle.com/javase/7/docs/api/javax/script/SimpleBindings.html), [TabularDataSupport](http://docs.oracle.com/javase/7/docs/api/javax/management/openmbean/TabularDataSupport.html), [TreeMap](http://docs.oracle.com/javase/7/docs/api/java/util/TreeMap.html), [UIDefaults](http://docs.oracle.com/javase/7/docs/api/javax/swing/UIDefaults.html), [WeakHashMap](http://docs.oracle.com/javase/7/docs/api/java/util/WeakHashMap.html)

The last two core collection interfaces are merely sorted versions of Set and Map:

* SortedSet — a Set that maintains its elements in ascending order. Several additional operations are provided to take advantage of the ordering. Sorted sets are used for naturally ordered sets, such as word lists and membership rolls. Also see [The SortedSet Interface](https://docs.oracle.com/javase/tutorial/collections/interfaces/sorted-set.html) section.
* SortedMap — a Map that maintains its mappings in ascending key order. This is the Map analog ofSortedSet. Sorted maps are used for naturally ordered collections of key/value pairs, such as dictionaries and telephone directories. Also see [The SortedMap Interface](https://docs.oracle.com/javase/tutorial/collections/interfaces/sorted-map.html) section.

**ArrayList**

Creating the arraylist without initial capacity constructs an empty list with the default capacity of 10.

**Difference between Array and ArrayList**

First and Major difference between Array and ArrayList in Java is that Array is a **fixed length data structure** while ArrayList is a variable length [Collection class](http://java67.blogspot.sg/2012/09/java-collection-interview-questions.html). You can not change length of Array once created in Java but ArrayList re-size itself when gets full depending upon capacity and load factor  
  
.Since ArrayList is internally backed by Array in Java, any resize operation in ArrayList will slow down performance as it involves creating new Array and [copying content](http://java67.blogspot.sg/2012/07/copy-elements-from-list-to-set-in-java-collection-example.html) from old array to new array.

Another difference between Array and ArrayList in Java is that you can not use [Generics](http://javarevisited.blogspot.ca/2011/09/generics-java-example-tutorial.html) along with Array  
  
One more major difference between ArrayList and Array is that, **you can not store primitives in ArrayList**, it can only contain Objects. While Array can contain both primitives and Objects in Java.

## One more difference on Array vs ArrayList is that you can create instance of ArrayList without specifying size, Java will create Array List with default size but its mandatory to provide size of Array while creating either directly or indirectly by initializing. Traversing Collections

There are three ways to traverse collections: (1) using aggregate operations (2) with the for-each construct and (3) by using Iterators.

(1) using aggregate operations

A sequence of elements supporting sequential and parallel aggregate operations. The following example illustrates an aggregate operation using [Stream](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html) and [IntStream](https://docs.oracle.com/javase/8/docs/api/java/util/stream/IntStream.html" \o "interface in java.util.stream):

int sum = widgets.stream()

.filter(w -> w.getColor() == RED)

.mapToInt(w -> w.getWeight())

.sum();

In this example, widgets is a Collection<Widget>. We create a stream of Widget objects via [Collection.stream()](https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html" \l "stream--), filter it to produce a stream containing only the red widgets, and then transform it into a stream of int values representing the weight of each red widget. Then this stream is summed to produce a total weight.

Likewise, you could easily request a parallel stream, which might make sense if the collection is large enough and your computer has enough cores:

myShapesCollection.parallelStream()

.filter(e -> e.getColor() == Color.RED)

.forEach(e -> System.out.println(e.getName()));

There are many different ways to collect data with this API. For example, you might want to convert the elements of a Collection to String objects, then join them, separated by commas:

String joined = elements.stream()

.map(Object::toString)

.collect(Collectors.joining(", "));

Or perhaps sum the salaries of all employees:

int total = employees.stream()

.collect(Collectors.summingInt(Employee::getSalary)));

 The key difference between the new aggregate operations and the existing bulk operations (containsAll, addAll, etc.) is that the old versions are all mutative, meaning that they all modify the underlying collection. In contrast, the new aggregate operations do not modify the underlying collection. When using the new aggregate operations and lambda expressions, you must take care to avoid mutation so as not to introduce problems in the future, should your code be run later from a parallel stream.

### or-each Construct

The for-each construct allows you to concisely traverse a collection or array using a for loop — see [The for Statement](https://docs.oracle.com/javase/tutorial/java/nutsandbolts/for.html). The following code uses the for-each construct to print out each element of a collection on a separate line.

for (Object o : collection)

System.out.println(o);

### Iterators

An [Iterator](https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html) is an object that enables you to traverse through a collection and to remove elements from the collection selectively, if desired. You get an Iterator for a collection by calling its iterator method. The following is the Iterator interface.

public interface Iterator<E> {

boolean hasNext();

E next();

void remove(); //optional

}

The hasNext method returns true if the iteration has more elements, and the next method returns the next element in the iteration. The remove method removes the last element that was returned by next from the underlying Collection. The remove method may be called only once per call to next and throws an exception if this rule is violated.

Note that Iterator.remove is the *only* safe way to modify a collection during iteration; the behavior is unspecified if the underlying collection is modified in any other way while the iteration is in progress.

Use Iterator instead of the for-each construct when you need to:

* Remove the current element. The for-each construct hides the iterator, so you cannot call remove. Therefore, the for-each construct is not usable for filtering.
* Iterate over multiple collections in parallel.

Use Iterator instead of the for-each construct when you need to:

* Remove the current element. The for-each construct hides the iterator, so you cannot call remove. Therefore, the for-each construct is not usable for filtering.
* Iterate over multiple collections in parallel.

As a simple example of the power of bulk operations, consider the following idiom to remove *all* instances of a specified element, e, from a Collection, c.

c.removeAll(Collections.singleton(e));

More specifically, suppose you want to remove all of the null elements from a Collection.

c.removeAll(Collections.singleton(null));

This idiom uses Collections.singleton, which is a static factory method that returns an immutable Setcontaining only the specified element.

# The Set Interface

A [Set](https://docs.oracle.com/javase/8/docs/api/java/util/Set.html) is a [Collection](https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html) that cannot contain duplicate elements. It models the mathematical set abstraction. TheSet interface contains *only* methods inherited from Collection and adds the restriction that duplicate elements are prohibited. Set also adds a stronger contract on the behavior of the equals and hashCode operations, allowing Set instances to be compared meaningfully even if their implementation types differ. Two Set instances are equal if they contain the same elements.

The Java platform contains three general-purpose Set implementations: HashSet, TreeSet, andLinkedHashSet. [HashSet](https://docs.oracle.com/javase/8/docs/api/java/util/HashSet.html" \t "_blank), which stores its elements in a hash table, is the best-performing implementation; however it makes no guarantees concerning the order of iteration. [TreeSet](https://docs.oracle.com/javase/8/docs/api/java/util/TreeSet.html" \t "_blank), which stores its elements in a red-black tree, orders its elements based on their values; it is substantially slower than HashSet. [LinkedHashSet](https://docs.oracle.com/javase/8/docs/api/java/util/LinkedHashSet.html" \t "_blank), which is implemented as a hash table with a linked list running through it, orders its elements based on the order in which they were inserted into the set (insertion-order). LinkedHashSet spares its clients from the unspecified, generally chaotic ordering provided by HashSet at a cost that is only slightly higher.

Here's a simple but useful Set idiom. Suppose you have a Collection, c, and you want to create anotherCollection containing the same elements but with all duplicates eliminated. The following one-liner does the trick.

Collection<Type> noDups = new HashSet<Type>(c);

It works by creating a Set (which, by definition, cannot contain duplicates), initially containing all the elements in c. It uses the standard conversion constructor described in the [The Collection Interface](https://docs.oracle.com/javase/tutorial/collections/interfaces/collection.html" \t "_top) section.

Or, if using JDK 8 or later, you could easily collect into a Set using aggregate operations:

c.stream()

.collect(Collectors.toSet()); // no duplicates

Here's a slightly longer example that accumulates a Collection of names into a TreeSet:

Set<String> set = people.stream()

.map(Person::getName)

.collect(Collectors.toCollection(TreeSet::new));

And the following is a minor variant of the first idiom that preserves the order of the original collection while removing duplicate elements:

Collection<Type> noDups = new LinkedHashSet<Type>(c);

The following is a generic method that encapsulates the preceding idiom, returning a Set of the same generic type as the one passed.

public static <E> Set<E> removeDups(Collection<E> c) {

return new LinkedHashSet<E>(c);

}

**Difference between HashSet and TreeSet**   
  
**1. Ordering :** HashSet stores the object in random order . There is no guarantee that the element we  inserted first in the HashSet  will be printed first in the output .

Elements are sorted according to the natural ordering of its elements in TreeSet. If the objects can not   
be sorted in natural order than use [compareTo() method to sort the elements](http://javahungry.blogspot.com/2013/08/difference-between-comparable-and.html" \t "_blank) of TreeSet object.

**2. Null value :**   HashSet can store null object while TreeSet does not allow null object. If one try to store null object in TreeSet object , it will throw Null Pointer Exception.

**4. Speed :** HashSet is much faster than TreeSet,as performance time of HashSet is constant against the log time of TreeSet for most operations (add,remove ,contains and size) . Iteration performance of HashSet mainly depends on the load factor and initial capacity parameters.

**7. Comparision :** HashSet uses equals() method for comparison in java while TreeSet uses compareTo() method for maintaining ordering.

**Similarities Between HashSet and TreeSet**  
  
**1. Unique Elements :**   Since HashSet and TreeSet both implements Set interface . Both are allowed to store only unique elements in their objects. Thus there can never be any duplicate elements inside the HashSet and TreeSet objects.  
  
**2.** **Not Thread Safe :** HashSet and TreeSet both are not synchronized or not thread safe.HashSet and TreeSet, both implementations are not synchronized. If multiple threads access a hash set/ tree set concurrently, and at least one of the threads modifies the set, it must be synchronized externally.  
  
**3. Clone() method copy technique:**  Both HashSet and TreeSet uses shallow copy technique to create a clone of  their objects .  
  
**4. Fail-fast Iterators :**  The iterators returned by this class's  method are fail-fast: if the set is modified at any time after the iterator is  created, in any way except through the iterator's own  remove method, the iterator will throw a  ConcurrentModificationException.  Thus, in the face of concurrent modification, the iterator fails quickly and cleanly, rather than risking arbitrary, non-deterministic behavior at   an undetermined time in the future.

# **The List Interface**

A [List](https://docs.oracle.com/javase/8/docs/api/java/util/List.html) is an ordered [Collection](https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html) (sometimes called a *sequence*). Lists may contain duplicate elements. In addition to the operations inherited from Collection, the List interface includes operations for the following:

* Positional access — manipulates elements based on their numerical position in the list. This includes methods such as get, set, add, addAll, and remove.
* Search — searches for a specified object in the list and returns its numerical position. Search methods include indexOf and lastIndexOf.
* Iteration — extends Iterator semantics to take advantage of the list's sequential nature. ThelistIterator methods provide this behavior.
* Range-view — The sublist method performs arbitrary *range operations* on the list.

List : a. ArrayList

b. LinkedList:

An iterator for lists that allows the programmer to traverse the list in either direction, modify the list during iteration, and obtain the iterator's current position in the list. A ListIterator has no current element; its *cursor position* always lies between the element that would be returned by a call to previous() and the element that would be returned by a call to next(). An iterator for a list of length n has n+1 possible cursor positions, as illustrated by the carets (^) below:

Element(0) Element(1) Element(2) ... Element(n-1)

cursor positions: ^

And here's an example (JDK 8 and later) that aggregates some names into a List:

List<String> list = people.stream()

.map(Person::getName)

.collect(Collectors.toList());

## Iterators

As you'd expect, the Iterator returned by List's iterator operation returns the elements of the list in proper sequence. List also provides a richer iterator, called a ListIterator, which allows you to traverse the list in either direction, modify the list during iteration, and obtain the current position of the iterator.

|  |  |
| --- | --- |
| **ArrayList** | **LinkedList** |
| 1) ArrayList internally uses **dynamic array** to store the elements. | LinkedList internally uses **doubly linked list** to store the elements. |
| 2) 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. |
| 3) ArrayList class can **act as a list** only because it implements List only. | LinkedList class can **act as a list and queue** both because it implements List and Deque interfaces. |
| 4) ArrayList is **better for storing and accessing** data. | LinkedList is **better for manipulating** data. |

## ArrayList Vs LinkedList

1) **Search**: ArrayList search operation is pretty fast compared to the LinkedList search operation. get(int index) in ArrayList gives the performance of O(1) while LinkedList performance is O(n).

Reason: ArrayList maintains index based system for its elements as it uses array data structure implicitly which makes it faster for searching an element in the list. On the other side LinkedList implements **doubly linked list** which requires the traversal through all the elements for searching an element.

2) **Deletion**: LinkedList remove operation gives O(1) performance while ArrayList gives variable performance: O(n) in worst case (while removing first element) and O(1) in best case (While removing last element).

Conclusion: LinkedList element deletion is faster compared to ArrayList.

Reason: LinkedList’s each element maintains two pointers (addresses) which points to the both neighbor elements in the list. Hence removal only requires change in the pointer location in the two neighbor nodes (elements) of the node which is going to be removed. While In ArrayList all the elements need to be shifted to fill out the space created by removed element.

3) **Inserts Performance**: LinkedList add method gives O(1) performance while ArrayList gives O(n) in worst case. Reason is same as explained for remove.

4) **Memory Overhead**: ArrayList maintains indexes and element data while LinkedList maintains element data and two pointers for neighbor nodes hence the memory consumption is high in LinkedList comparatively.

There are few **similarities between** these classes which are as follows:

1. Both ArrayList and LinkedList are implementation of List interface.
2. They both maintain the elements insertion order which means while displaying ArrayList and LinkedList elements the result set would be having the same order in which the elements got inserted into the List.
3. Both these classes are non-synchronized and can be made synchronized explicitly by using **[Collections.synchronizedList](http://docs.oracle.com/javase/6/docs/api/java/util/Collections.html" \l "synchronizedList(java.util.List)" \t "_blank)** method.
4. The iterator and listIterator returned by these classes are fail-fast (if list is structurally modified at any time after the iterator is created, in any way except through the iterator’s own remove or add methods, the iterator will throw a **[ConcurrentModificationException](http://docs.oracle.com/javase/6/docs/api/java/util/ConcurrentModificationException.html" \t "_blank)**).

### When to use LinkedList and when to use ArrayList?

1) As explained above the insert and remove operations give good performance (O(1)) in LinkedList compared to ArrayList(O(n)). Hence if there is a requirement of frequent addition and deletion in application then LinkedList is a best choice.

2) Search (get method) operations are fast in Arraylist (O(1)) but not in LinkedList (O(n)) so If there are less add and remove operations and more search operations requirement, ArrayList would be your best bet.

# **The Queue Interface**

A [Queue](https://docs.oracle.com/javase/8/docs/api/java/util/Queue.html) is a collection for holding elements prior to processing. Besides basic Collection operations, queues provide additional insertion, removal, and inspection operations. The Queue interface follows.

**All Known Implementing Classes:**

[AbstractQueue](https://docs.oracle.com/javase/8/docs/api/java/util/AbstractQueue.html), [ArrayBlockingQueue](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ArrayBlockingQueue.html), [ArrayDeque](https://docs.oracle.com/javase/8/docs/api/java/util/ArrayDeque.html), [ConcurrentLinkedDeque](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentLinkedDeque.html), [ConcurrentLinkedQueue](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentLinkedQueue.html),[DelayQueue](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/DelayQueue.html), [LinkedBlockingDeque](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/LinkedBlockingDeque.html), [LinkedBlockingQueue](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/LinkedBlockingQueue.html), [LinkedList](https://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html), [LinkedTransferQueue](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/LinkedTransferQueue.html),[PriorityBlockingQueue](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/PriorityBlockingQueue.html), [PriorityQueue](https://docs.oracle.com/javase/8/docs/api/java/util/PriorityQueue.html), [SynchronousQueue](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/SynchronousQueue.html)

public interface Queue<E> extends Collection<E> {

E element();

boolean offer(E e);

E peek();

E poll();

E remove();

}

|  |  |  |
| --- | --- | --- |
| **Queue Interface Structure** | | |
| **Type of Operation** | **Throws exception** | **Returns special value** |
| Insert | add(e) | offer(e) |
| Remove | remove() | poll() |
| Examine | element() | peek() |

Queues typically, but not necessarily, order elements in a FIFO (first-in-first-out) manner. Among the exceptions are priority queues, which order elements according to their values — see the [Object Ordering](https://docs.oracle.com/javase/tutorial/collections/interfaces/order.html) section for details). Whatever ordering is used, the head of the queue is the element that would be removed by a call to remove orpoll. In a FIFO queue, all new elements are inserted at the tail of the queue. Other kinds of queues may use different placement rules. Every Queue implementation must specify its ordering properties.

t is possible for a Queue implementation to restrict the number of elements that it holds; such queues are known as *bounded*. Some Queue implementations in java.util.concurrent are bounded, but the implementations injava.util are not.

The add method, which Queue inherits from Collection, inserts an element unless it would violate the queue's capacity restrictions, in which case it throws IllegalStateException. The offer method, which is intended solely for use on bounded queues, differs from add only in that it indicates failure to insert an element by returningfalse.

The remove and poll methods both remove and return the head of the queue. Exactly which element gets removed is a function of the queue's ordering policy. The remove and poll methods differ in their behavior only when the queue is empty. Under these circumstances, remove throws NoSuchElementException, while pollreturns null.

The element and peek methods return, but do not remove, the head of the queue. They differ from one another in precisely the same fashion as remove and poll: If the queue is empty, element throwsNoSuchElementException, while peek returns null.

Queue implementations generally do not allow insertion of null elements. The LinkedList implementation, which was retrofitted to implement Queue, is an exception. For historical reasons, it permits null elements, but you should refrain from taking advantage of this, because null is used as a special return value by the poll andpeek methods.

LinkedList is one of the implementation of the queue.

BlockingQue:

A [Queue](https://docs.oracle.com/javase/8/docs/api/java/util/Queue.html) that additionally supports operations that wait for the queue to become non-empty when retrieving an element, and wait for space to become available in the queue when storing an element.

BlockingQueue methods come in four forms, with different ways of handling operations that cannot be satisfied immediately, but may be satisfied at some point in the future: one throws an exception, the second returns a special value (either null or false, depending on the operation), the third blocks the current thread indefinitely until the operation can succeed, and the fourth blocks for only a given maximum time limit before giving up. These methods are summarized in the following table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Summary of BlockingQueue methods | | | | |
|  | *Throws exception* | *Special value* | *Blocks* | *Times out* |
| **Insert** | [add(e)](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingQueue.html#add-E-) | [offer(e)](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingQueue.html#offer-E-) | [put(e)](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingQueue.html#put-E-) | [offer(e, time, unit)](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingQueue.html#offer-E-long-java.util.concurrent.TimeUnit-) |
| **Remove** | [remove()](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingQueue.html#remove-java.lang.Object-) | [poll()](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingQueue.html#poll-long-java.util.concurrent.TimeUnit-) | [take()](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingQueue.html#take--) | [poll(time, unit)](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingQueue.html#poll-long-java.util.concurrent.TimeUnit-) |
| **Examine** | [element()](https://docs.oracle.com/javase/8/docs/api/java/util/Queue.html#element--) | [peek()](https://docs.oracle.com/javase/8/docs/api/java/util/Queue.html#peek--) | *not applicable* | *not applicable* |

BlockingQueue implementations are thread-safe

# The Deque Interface

Usually pronounced as deck, a deque is a double-ended-queue. A double-ended-queue is a linear collection of elements that supports the insertion and removal of elements at both end points.

|  |  |  |
| --- | --- | --- |
| **Deque Methods** | | |
| **Type of Operation** | **First Element (Beginning of the Dequeinstance)** | **Last Element (End of the Dequeinstance)** |
| **Insert** | addFirst(e) offerFirst(e) | addLast(e) offerLast(e) |
| **Remove** | removeFirst() pollFirst() | removeLast() pollLast() |
| **Examine** | getFirst() peekFirst() | getLast() peekLast() |

**Map interface**

The Java platform contains three general-purpose Map implementations: [HashMap](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html" \t "_blank), [TreeMap](https://docs.oracle.com/javase/8/docs/api/java/util/TreeMap.html" \t "_blank), and[LinkedHashMap](https://docs.oracle.com/javase/8/docs/api/java/util/LinkedHashMap.html)

The remainder of this page discusses the Map interface in detail. But first, here are some more examples of collecting to Maps using JDK 8 aggregate operations. Modeling real-world objects is a common task in object-oriented programming, so it is reasonable to think that some programs might, for example, group employees by department:

// Group employees by department

Map<Department, List<Employee>> byDept = employees.stream()

.collect(Collectors.groupingBy(Employee::getDepartment));

Or compute the sum of all salaries by department:

// Compute sum of salaries by department

Map<Department, Integer> totalByDept = employees.stream()

.collect(Collectors.groupingBy(Employee::getDepartment,

Collectors.summingInt(Employee::getSalary)));

Or perhaps group students by passing or failing grades:

// Partition students into passing and failing

Map<Boolean, List<Student>> passingFailing = students.stream()

.collect(Collectors.partitioningBy(s -> s.getGrade()>= PASS\_THRESHOLD));

You could also group people by city:

// Classify Person objects by city

Map<String, List<Person>> peopleByCity

= personStream.collect(Collectors.groupingBy(Person::getCity));

Or even cascade two collectors to classify people by state and city:

// Cascade Collectors

Map<String, Map<String, List<Person>>> peopleByStateAndCity

= personStream.collect(Collectors.groupingBy(Person::getState,

Collectors.groupingBy(Person::getCity)))

**HashTable**

This class implements a hash table, which maps keys to values. Any non-null object can be used as a key or as a value. To successfully store and retrieve objects from a hashtable, the objects used as keys must implement the hashCode method and the equals method.

Hashtable is synchronized. If a thread-safe implementation is not needed, it is recommended to use [HashMap](http://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html" \o "class in java.util) in place ofHashtable. If a thread-safe highly-concurrent implementation is desired, then it is recommended to use [ConcurrentHashMap](http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ConcurrentHashMap.html" \o "class in java.util.concurrent) in place of Hashtable

**Difference between HashMap and HashTable**

* 1. **Synchronization or Thread Safe :**  This is the most important difference between two . HashMap is non synchronized and not thread safe.On the other hand, HashTable is thread safe and synchronized.  
     When to use HashMap ?  answer is if your application do not require any multi-threading task, in other words hashmap is better for non-threading applications. HashTable should be used in multithreading applications.   
       
     **2. Null keys and null values :**  Hashmap allows one null key and any number of null values, while Hashtable do not allow null keys and null values in the HashTable object.  
       
       
       
     **3. Iterating the values:**  Hashmap object values are iterated by using iterator .HashTable is the only class other than vector which uses enumerator to iterate the values of HashTable object.

**4.  Fail-fast iterator**  : The iterator in Hashmap is fail-fast iterator while the enumerator for Hashtable is not.  
According to [Oracle Docs](http://docs.oracle.com/javase/7/docs/api/java/util/Hashtable.html),  if the Hashtable is structurally modified at any time after the iterator is created in any way except the iterator's own remove method , then the iterator will throw ConcurrentModification Exception.  
Structural modification means adding or removing elements from the Collection object (here hashmap or hashtable) . Thus the enumerations returned by the Hashtable keys and elements methods are not fail fast.We have already explained the[difference between iterator and enumeration](http://javahungry.blogspot.com/2013/06/difference-between-iterator-and-enumeration-collections-java-interview-question-with-example.html).  
  
  
**5. Performance :**  Hashmap is much faster and uses less memory than Hashtable as former is unsynchronized . Unsynchronized objects are often much better in performance in compare to synchronized  object like Hashtable in single threaded environment.  
  
**6. Superclass and Legacy :**  Hashtable is a subclass of Dictionary class which is now obsolete in Jdk 1.7 ,so ,it is not used anymore. It is better off externally synchronizing a HashMap or using a ConcurrentMap implementation (e.g ConcurrentHashMap).HashMap is the subclass of the AbstractMap class. Although Hashtable and HashMap has different superclasses but they both are implementations of the *"Map"*  abstract data type.

### **ConcurrentHashMap**

* You should use ConcurrentHashMap when you need very high concurrency in your project.
* It is thread safe without synchronizing the whole map.
* Reads can happen very fast while write is done with a lock.
* There is no locking at the object level.
* The locking is at a much finer granularity at a hashmap bucket level.
* ConcurrentHashMap doesn’t throw a ConcurrentModificationException if one thread tries to modify it while another is iterating over it.
* ConcurrentHashMap uses multitude of locks.

### **SynchronizedHashMap**

A hashmap can be made synchronized in following way,

Map<String,String> syncMap = Collections.synchronizedMap(map);

* Synchronization at Object level.
* Every read/write operation needs to acquire lock.
* Locking the entire collection is a performance overhead.
* This essentially gives access to only one thread to the entire map & blocks all the other threads.
* It may cause contention.
* SynchronizedHashMap returns Iterator, which fails-fast on concurrent modification.

**LinkedHashMap**

This implementation differs from HashMap in that it maintains a doubly-linked list running through all of its entries. This linked list defines the iteration ordering, which is normally the order in which keys were inserted into the map (*insertion-order*)

**Collection Views**

The Collection view methods allow a Map to be viewed as a Collection in these three ways:

* keySet — the Set of keys contained in the Map.
* values — The Collection of values contained in the Map. This Collection is not a Set, because multiple keys can map to the same value.
* entrySet — the Set of key-value pairs contained in the Map. The Map interface provides a small nested interface called Map.Entry, the type of the elements in this Set.
* The Collection views provide the *only* means to iterate over a Map. This example illustrates the standard idiom for iterating over the keys in a Map with a for-each construct:
* for (KeyType key : m.keySet())
* System.out.println(key);
* and with an iterator:
* // Filter a map based on some
* // property of its keys.
* for (Iterator<Type> it = m.keySet().iterator(); it.hasNext(); )
* if (it.next().isBogus())
* it.remove();

With the entrySet view, it is also possible to change the value associated with a key by calling a Map.Entry'ssetValue method during iteration (again, assuming the Map supports value modification to begin with). Note that these are the *only* safe ways to modify a Map during iteration; the behavior is unspecified if the underlying Map is modified in any other way while the iteration is in progress.

If you try to sort a list, the elements of which do not implement Comparable, Collections.sort(list) will throw a [ClassCastException](https://docs.oracle.com/javase/8/docs/api/java/lang/ClassCastException.html" \t "_blank). Similarly, Collections.sort(list, comparator) will throw aClassCastException if you try to sort a list whose elements cannot be compared to one another using thecomparator. Elements that can be compared to one another are called *mutually comparable*. Although elements of different types may be mutually comparable, none of the classes listed here permit interclass comparison.

The [following class representing a person's name](https://docs.oracle.com/javase/tutorial/collections/interfaces/examples/Name.java) implements Comparable.

import java.util.\*;

public class Name implements Comparable<Name> {

private final String firstName, lastName;

public Name(String firstName, String lastName) {

if (firstName == null || lastName == null)

throw new NullPointerException();

this.firstName = firstName;

this.lastName = lastName;

}

public String firstName() { return firstName; }

public String lastName() { return lastName; }

public boolean equals(Object o) {

if (!(o instanceof Name))

return false;

Name n = (Name) o;

return n.firstName.equals(firstName) && n.lastName.equals(lastName);

}

public int hashCode() {

return 31\*firstName.hashCode() + lastName.hashCode();

}

public String toString() {

return firstName + " " + lastName;

}

public int compareTo(Name n) {

int lastCmp = lastName.compareTo(n.lastName);

return (lastCmp != 0 ? lastCmp : firstName.compareTo(n.firstName));

}

}

**Difference between Comparable and Comparator**

Comparable and Comparator are two interfaces provided by Java Core API. From their names, we can tell they may be used for comparing stuff in some way. But what exactly are they and what is the difference between them? The following are two examples for answering this question. The simple examples compare two HDTV's size. How to use Comparable vs. Comparator is obvious after reading the code.

1. Comparable

Comparable is implemented by a class in order to be able to comparing object of itself with some other objects. The class itself must implement the interface in order to be able to compare its instance(s). The method required for implementation is compareTo(). Here is an example:

|  |
| --- |
| **class** HDTV **implements** Comparable<HDTV> {  **private** **int** size;  **private** String brand;    **public** HDTV(**int** size, String brand) {  **this**.size = size;  **this**.brand = brand;  }    **public** **int** getSize() {  **return** size;  }    **public** **void** setSize(**int** size) {  **this**.size = size;  }    **public** String getBrand() {  **return** brand;  }    **public** **void** setBrand(String brand) {  **this**.brand = brand;  }    @Override  **public** **int** compareTo(HDTV tv) {    **if** (**this**.getSize() > tv.getSize())  **return** 1;  **else** **if** (**this**.getSize() < tv.getSize())  **return** -1;  **else**  **return** 0;  }  }    **public** **class** Main {  **public** **static** **void** main(String[] args) {  HDTV tv1 = **new** HDTV(55, "Samsung");  HDTV tv2 = **new** HDTV(60, "Sony");    **if** (tv1.compareTo(tv2) > 0) {  System.out.println(tv1.getBrand() + " is better.");  } **else** {  System.out.println(tv2.getBrand() + " is better.");  }  }  } |

Sony is better.

2. Comparator

In some situations, you may not want to change a class and make it comparable. In such cases,Comparator can be used if you want to compare objects based on certain attributes/fields. For example, 2 persons can be compared based on `height` or `age` etc. (this can not be done using comparable.)

The method required to implement is *compare()*. Now let's use another way to compare those TV by size. One common use of Comparator is sorting. Both Collections and Arrays classes provide a sort method which use a Comparator.

|  |
| --- |
| **import** java.util.ArrayList;  **import** java.util.Collections;  **import** java.util.Comparator;    **class** HDTV {  **private** **int** size;  **private** String brand;    **public** HDTV(**int** size, String brand) {  **this**.size = size;  **this**.brand = brand;  }    **public** **int** getSize() {  **return** size;  }    **public** **void** setSize(**int** size) {  **this**.size = size;  }    **public** String getBrand() {  **return** brand;  }    **public** **void** setBrand(String brand) {  **this**.brand = brand;  }  }    **class** SizeComparator **implements** Comparator<HDTV> {  @Override  **public** **int** compare(HDTV tv1, HDTV tv2) {  **int** tv1Size = tv1.getSize();  **int** tv2Size = tv2.getSize();    **if** (tv1Size > tv2Size) {  **return** 1;  } **else** **if** (tv1Size < tv2Size) {  **return** -1;  } **else** {  **return** 0;  }  }  }    **public** **class** Main {  **public** **static** **void** main(String[] args) {  HDTV tv1 = **new** HDTV(55, "Samsung");  HDTV tv2 = **new** HDTV(60, "Sony");  HDTV tv3 = **new** HDTV(42, "Panasonic");    ArrayList<HDTV> al = **new** ArrayList<HDTV>();  al.add(tv1);  al.add(tv2);  al.add(tv3);    Collections.sort(al, **new** SizeComparator());  **for** (HDTV a : al) {  System.out.println(a.getBrand());  }  }  } |

# The SortedSet Interface

A [SortedSet](https://docs.oracle.com/javase/8/docs/api/java/util/SortedSet.html" \t "_blank) is a [Set](https://docs.oracle.com/javase/8/docs/api/java/util/Set.html) that maintains its elements in ascending order, sorted according to the elements' natural ordering or according to a Comparator provided at SortedSet creation time. In addition to the normal Setoperations, the SortedSet interface provides operations for the following:

* Range view — allows arbitrary range operations on the sorted set
* Endpoints — returns the first or last element in the sorted set
* Comparator access — returns the Comparator, if any, used to sort the set

**Set Operations**

The operations that SortedSet inherits from Set behave identically on sorted sets and normal sets with two exceptions:

* The Iterator returned by the iterator operation traverses the sorted set in order.
* The array returned by toArray contains the sorted set's elements in order.

Although the interface doesn't guarantee it, the toString method of the Java platform's SortedSetimplementations returns a string containing all the elements of the sorted set, in order.

A range-view of a sorted set is really just a window onto whatever portion of the set lies in the designated part of the element space. Changes to the range-view write back to the backing sorted set and vice versa. Thus, it's okay to use range-views on sorted sets for long periods of time, unlike range-views on lists.

The SortedSet interface contains two more range-view operations — headSet and tailSet, both of which take a single Object argument. The former returns a view of the initial portion of the backing SortedSet, up to but not including the specified object. The latter returns a view of the final portion of the backing SortedSet, beginning with the specified object and continuing to the end of the backing SortedSet. Thus, the following code allows you to view the dictionary as two disjoint volumes (a-m and n-z).

SortedSet<String> volume1 = dictionary.headSet("n");

SortedSet<String> volume2 = dictionary.tailSet("n");

# The SortedMap Interface

A [SortedMap](https://docs.oracle.com/javase/8/docs/api/java/util/SortedMap.html" \t "_blank) is a [Map](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html) that maintains its entries in ascending order, sorted according to the keys' natural ordering, or according to a Comparator provided at the time of the SortedMap creation. Natural ordering andComparators are discussed in the [Object Ordering](https://docs.oracle.com/javase/tutorial/collections/interfaces/order.html) section. The SortedMap interface provides operations for normal Map operations and for the following:

* Range view — performs arbitrary range operations on the sorted map
* Endpoints — returns the first or the last key in the sorted map
* Comparator access — returns the Comparator, if any, used to sort the map

**Reference**

<https://docs.oracle.com/javase/tutorial/collections/interfaces/collection.html>

<http://javahungry.blogspot.com/2013/08/hashing-how-hash-map-works-in-java-or.html>