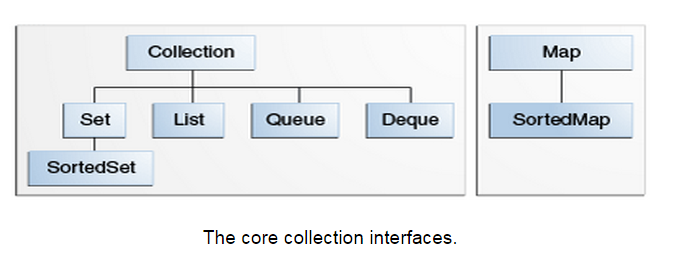
**JAVA**

**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.

**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

**Reference**

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