**<https://docs.oracle.com/javase/tutorial/collections/implementations/queue.html>**

# The Collection Interface

A [Collection](https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html) represents a group of objects known as its elements. The Collection interface is used to pass around collections of objects where maximum generality is desired. For example, by convention all general-purpose collection implementations have a constructor that takes a Collection argument. This constructor, known as a *conversion constructor*, initializes the new collection to contain all of the elements in the specified collection, whatever the given collection's subinterface or implementation type. In other words, it allows you to *convert* the collection's type.

Suppose, for example, that you have a Collection<String> c, which may be a List, a Set, or another kind of Collection. This idiom creates a new ArrayList (an implementation of the List interface), initially containing all the elements in c.

List<String> list = new ArrayList<String>(c);

Or — if you are using JDK 7 or later — you can use the diamond operator:

List<String> list = new ArrayList<>(c);

The Collection interface contains methods that perform basic operations, such as int size(), boolean isEmpty(), boolean contains(Object element), boolean add(E element), boolean remove(Object element), and Iterator<E> iterator().

It also contains methods that operate on entire collections, such as boolean containsAll(Collection<?> c), boolean addAll(Collection<? extends E> c), boolean removeAll(Collection<?> c), boolean retainAll(Collection<?> c), and void clear().

Additional methods for array operations (such as Object[] toArray() and <T> T[] toArray(T[] a) exist as well.

In JDK 8 and later, the Collection interface also exposes methods Stream<E> stream() and Stream<E> parallelStream(), for obtaining sequential or parallel streams from the underlying collection. (See the lesson entitled [Aggregate Operations](https://docs.oracle.com/javase/tutorial/collections/streams/index.html) for more information about using streams.)

The Collection interface does about what you'd expect given that a Collection represents a group of objects. It has methods that tell you how many elements are in the collection (size, isEmpty), methods that check whether a given object is in the collection (contains), methods that add and remove an element from the collection (add, remove), and methods that provide an iterator over the collection (iterator).

The add method is defined generally enough so that it makes sense for collections that allow duplicates as well as those that don't. It guarantees that the Collection will contain the specified element after the call completes, and returns true if the Collection changes as a result of the call. Similarly, the remove method is designed to remove a single instance of the specified element from the Collection, assuming that it contains the element to start with, and to return true if the Collection was modified as a result.

## Traversing Collections

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

### Aggregate Operations

In JDK 8 and later, the preferred method of iterating over a collection is to obtain a stream and perform aggregate operations on it. Aggregate operations are often used in conjunction with lambda expressions to make programming more expressive, using less lines of code. The following code sequentially iterates through a collection of shapes and prints out the red objects:

myShapesCollection.stream()

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

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

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)));

These are but a few examples of what you can do with streams and aggregate operations. For more information and examples, see the lesson entitled [Aggregate Operations](https://docs.oracle.com/javase/tutorial/collections/streams/index.html).

The Collections framework has always provided a number of so-called "bulk operations" as part of its API. These include methods that operate on entire collections, such as containsAll,addAll, removeAll, etc. Do not confuse those methods with the aggregate operations that were introduced in JDK 8. 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.

### for-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.

The following method shows you how to use an Iterator to filter an arbitrary Collection — that is, traverse the collection removing specific elements.

static void filter(Collection<?> c) {

for (Iterator<?> it = c.iterator(); it.hasNext(); )

if (!cond(it.next()))

it.remove();

}

This simple piece of code is polymorphic, which means that it works for *any* Collection regardless of implementation. This example demonstrates how easy it is to write a polymorphic algorithm using the Java Collections Framework.

## Collection Interface Bulk Operations

*Bulk operations* perform an operation on an entire Collection. You could implement these shorthand operations using the basic operations, though in most cases such implementations would be less efficient. The following are the bulk operations:

* containsAll — returns true if the target Collection contains all of the elements in the specified Collection.
* addAll — adds all of the elements in the specified Collection to the target Collection.
* removeAll — removes from the target Collection all of its elements that are also contained in the specified Collection.
* retainAll — removes from the target Collection all its elements that are *not* also contained in the specified Collection. That is, it retains only those elements in the targetCollection that are also contained in the specified Collection.
* clear — removes all elements from the Collection.

The addAll, removeAll, and retainAll methods all return true if the target Collection was modified in the process of executing the operation.

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 Set containing only the specified element.

## Collection Interface Array Operations

The toArray methods are provided as a bridge between collections and older APIs that expect arrays on input. The array operations allow the contents of a Collection to be translated into an array. The simple form with no arguments creates a new array of Object. The more complex form allows the caller to provide an array or to choose the runtime type of the output array.

For example, suppose that c is a Collection. The following snippet dumps the contents of c into a newly allocated array of Object whose length is identical to the number of elements inc.

Object[] a = c.toArray();

Suppose that c is known to contain only strings (perhaps because c is of type Collection<String>). The following snippet dumps the contents of c into a newly allocated array ofString whose length is identical to the number of elements in c.

String[] a = c.toArray(new String[0]);

[« Previous](https://docs.oracle.com/javase/tutorial/collections/interfaces/index.html) • [Trail](https://docs.oracle.com/javase/tutorial/collections/TOC.html) • [Next »](https://docs.oracle.com/javase/tutorial/collections/interfaces/set.html)

# 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. The Set 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, and LinkedHashSet. [HashSet](https://docs.oracle.com/javase/8/docs/api/java/util/HashSet.html), 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), 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), 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 another Collection 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) 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);

}

## Set Interface Basic Operations

The size operation returns the number of elements in the Set (its *cardinality*). The isEmpty method does exactly what you think it would. The add method adds the specified element to theSet if it is not already present and returns a boolean indicating whether the element was added. Similarly, the remove method removes the specified element from the Set if it is present and returns a boolean indicating whether the element was present. The iterator method returns an Iterator over the Set.

The following [program](https://docs.oracle.com/javase/tutorial/collections/interfaces/examples/FindDups.java) prints out all distinct words in its argument list. Two versions of this program are provided. The first uses JDK 8 aggregate operations. The second uses the for-each construct.

Using JDK 8 Aggregate Operations:

import java.util.\*;

import java.util.stream.\*;

public class FindDups {

public static void main(String[] args) {

Set<String> distinctWords = Arrays.asList(args).stream()

.collect(Collectors.toSet());

System.out.println(distinctWords.size()+

" distinct words: " +

distinctWords);

}

}

Using the for-each Construct:

import java.util.\*;

public class FindDups {

public static void main(String[] args) {

Set<String> s = new HashSet<String>();

for (String a : args)

s.add(a);

System.out.println(s.size() + " distinct words: " + s);

}

}

Now run either version of the program.

java FindDups i came i saw i left

The following output is produced:

4 distinct words: [left, came, saw, i]

Note that the code always refers to the Collection by its interface type (Set) rather than by its implementation type. This is a *strongly* recommended programming practice because it gives you the flexibility to change implementations merely by changing the constructor. If either of the variables used to store a collection or the parameters used to pass it around are declared to be of the Collection's implementation type rather than its interface type, *all* such variables and parameters must be changed in order to change its implementation type.

Furthermore, there's no guarantee that the resulting program will work. If the program uses any nonstandard operations present in the original implementation type but not in the new one, the program will fail. Referring to collections only by their interface prevents you from using any nonstandard operations.

The implementation type of the Set in the preceding example is HashSet, which makes no guarantees as to the order of the elements in the Set. If you want the program to print the word list in alphabetical order, merely change the Set's implementation type from HashSet to TreeSet. Making this trivial one-line change causes the command line in the previous example to generate the following output.

java FindDups i came i saw i left

4 distinct words: [came, i, left, saw]

## Set Interface Bulk Operations

Bulk operations are particularly well suited to Sets; when applied, they perform standard set-algebraic operations. Suppose s1 and s2 are sets. Here's what bulk operations do:

* s1.containsAll(s2) — returns true if s2 is a **subset** of s1. (s2 is a subset of s1 if set s1 contains all of the elements in s2.)
* s1.addAll(s2) — transforms s1 into the **union** of s1 and s2. (The union of two sets is the set containing all of the elements contained in either set.)
* s1.retainAll(s2) — transforms s1 into the intersection of s1 and s2. (The intersection of two sets is the set containing only the elements common to both sets.)
* s1.removeAll(s2) — transforms s1 into the (asymmetric) set difference of s1 and s2. (For example, the set difference of s1 minus s2 is the set containing all of the elements found in s1 but not in s2.)

To calculate the union, intersection, or set difference of two sets *nondestructively* (without modifying either set), the caller must copy one set before calling the appropriate bulk operation. The following are the resulting idioms.

Set<Type> union = new HashSet<Type>(s1);

union.addAll(s2);

Set<Type> intersection = new HashSet<Type>(s1);

intersection.retainAll(s2);

Set<Type> difference = new HashSet<Type>(s1);

difference.removeAll(s2);

The implementation type of the result Set in the preceding idioms is HashSet, which is, as already mentioned, the best all-around Set implementation in the Java platform. However, any general-purpose Set implementation could be substituted.

Let's revisit the FindDups program. Suppose you want to know which words in the argument list occur only once and which occur more than once, but you do not want any duplicates printed out repeatedly. This effect can be achieved by generating two sets — one containing every word in the argument list and the other containing only the duplicates. The words that occur only once are the set difference of these two sets, which we know how to compute. Here's how [the resulting program](https://docs.oracle.com/javase/tutorial/collections/interfaces/examples/FindDups2.java) looks.

import java.util.\*;

public class FindDups2 {

public static void main(String[] args) {

Set<String> uniques = new HashSet<String>();

Set<String> dups = new HashSet<String>();

for (String a : args)

if (!uniques.add(a))

dups.add(a);

// Destructive set-difference

uniques.removeAll(dups);

System.out.println("Unique words: " + uniques);

System.out.println("Duplicate words: " + dups);

}

}

When run with the same argument list used earlier (i came i saw i left), the program yields the following output.

Unique words: [left, saw, came]

Duplicate words: [i]

A less common set-algebraic operation is the *symmetric set difference* — the set of elements contained in either of two specified sets but not in both. The following code calculates the symmetric set difference of two sets nondestructively.

Set<Type> symmetricDiff = new HashSet<Type>(s1);

symmetricDiff.addAll(s2);

Set<Type> tmp = new HashSet<Type>(s1);

tmp.retainAll(s2);

symmetricDiff.removeAll(tmp);

## Set Interface Array Operations

The array operations don't do anything special for Sets beyond what they do for any other Collection. These operations are described in [The Collection Interface](https://docs.oracle.com/javase/tutorial/collections/interfaces/collection.html) section.

[« Previous](https://docs.oracle.com/javase/tutorial/collections/interfaces/collection.html) • [Trail](https://docs.oracle.com/javase/tutorial/collections/TOC.html) • [Next »](https://docs.oracle.com/javase/tutorial/collections/interfaces/list.html)

# 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. The listIterator methods provide this behavior.
* Range-view — The sublist method performs arbitrary *range operations* on the list.

The Java platform contains two general-purpose List implementations. [ArrayList](https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html), which is usually the better-performing implementation, and [LinkedList](https://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html) which offers better performance under certain circumstances.

## Collection Operations

The operations inherited from Collection all do about what you'd expect them to do, assuming you're already familiar with them. If you're not familiar with them from Collection, now would be a good time to read [The Collection Interface](https://docs.oracle.com/javase/tutorial/collections/interfaces/collection.html) section. The remove operation always removes *the first* occurrence of the specified element from the list. The add and addAlloperations always append the new element(s) to the *end* of the list. Thus, the following idiom concatenates one list to another.

list1.addAll(list2);

Here's a nondestructive form of this idiom, which produces a third List consisting of the second list appended to the first.

List<Type> list3 = new ArrayList<Type>(list1);

list3.addAll(list2);

Note that the idiom, in its nondestructive form, takes advantage of ArrayList's standard conversion constructor.

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());

Like the [Set](https://docs.oracle.com/javase/8/docs/api/java/util/Set.html) interface, List strengthens the requirements on the equals and hashCode methods so that two List objects can be compared for logical equality without regard to their implementation classes. Two List objects are equal if they contain the same elements in the same order.

## Positional Access and Search Operations

The basic positional access operations are get, set, add and remove. (The set and remove operations return the old value that is being overwritten or removed.) Other operations (indexOf and lastIndexOf) return the first or last index of the specified element in the list.

The addAll operation inserts all the elements of the specified Collection starting at the specified position. The elements are inserted in the order they are returned by the specifiedCollection's iterator. This call is the positional access analog of Collection's addAll operation.

Here's a little method to swap two indexed values in a List.

public static <E> void swap(List<E> a, int i, int j) {

E tmp = a.get(i);

a.set(i, a.get(j));

a.set(j, tmp);

}

Of course, there's one big difference. This is a polymorphic algorithm: It swaps two elements in any List, regardless of its implementation type. Here's another polymorphic algorithm that uses the preceding swap method.

public static void shuffle(List<?> list, Random rnd) {

for (int i = list.size(); i > 1; i--)

swap(list, i - 1, rnd.nextInt(i));

}

This algorithm, which is included in the Java platform's [Collections](https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html) class, randomly permutes the specified list using the specified source of randomness. It's a bit subtle: It runs up the list from the bottom, repeatedly swapping a randomly selected element into the current position. Unlike most naive attempts at shuffling, it's *fair* (all permutations occur with equal likelihood, assuming an unbiased source of randomness) and *fast* (requiring exactly list.size()-1 swaps). The following program uses this algorithm to print the words in its argument list in random order.

import java.util.\*;

public class Shuffle {

public static void main(String[] args) {

List<String> list = new ArrayList<String>();

for (String a : args)

list.add(a);

Collections.shuffle(list, new Random());

System.out.println(list);

}

}

In fact, this program can be made even shorter and faster. The [Arrays](https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html) class has a static factory method called asList, which allows an array to be viewed as a List. This method does not copy the array. Changes in the List write through to the array and vice versa. The resulting List is not a general-purpose List implementation, because it doesn't implement the (optional)add and remove operations: Arrays are not resizable. Taking advantage of Arrays.asList and calling the library version of shuffle, which uses a default source of randomness, you get the following [tiny program](https://docs.oracle.com/javase/tutorial/collections/interfaces/examples/Shuffle.java) whose behavior is identical to the previous program.

import java.util.\*;

public class Shuffle {

public static void main(String[] args) {

List<String> list = Arrays.asList(args);

Collections.shuffle(list);

System.out.println(list);

}

}

## 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.

The three methods that ListIterator inherits from Iterator (hasNext, next, and remove) do exactly the same thing in both interfaces. The hasPrevious and the previousoperations are exact analogues of hasNext and next. The former operations refer to the element before the (implicit) cursor, whereas the latter refer to the element after the cursor. Theprevious operation moves the cursor backward, whereas next moves it forward.

Here's the standard idiom for iterating backward through a list.

for (ListIterator<Type> it = list.listIterator(list.size()); it.hasPrevious(); ) {

Type t = it.previous();

...

}

Note the argument to listIterator in the preceding idiom. The List interface has two forms of the listIterator method. The form with no arguments returns a ListIteratorpositioned at the beginning of the list; the form with an int argument returns a ListIterator positioned at the specified index. The index refers to the element that would be returned by an initial call to next. An initial call to previous would return the element whose index was index-1. In a list of length n, there are n+1 valid values for index, from 0 to n, inclusive.

Intuitively speaking, the cursor is always between two elements — the one that would be returned by a call to previous and the one that would be returned by a call to next. The n+1 validindex values correspond to the n+1 gaps between elements, from the gap before the first element to the gap after the last one. The following figure shows the five possible cursor positions in a list containing four elements.

The five possible cursor positions.

Calls to next and previous can be intermixed, but you have to be a bit careful. The first call to previous returns the same element as the last call to next. Similarly, the first call to nextafter a sequence of calls to previous returns the same element as the last call to previous.

It should come as no surprise that the nextIndex method returns the index of the element that would be returned by a subsequent call to next, and previousIndex returns the index of the element that would be returned by a subsequent call to previous. These calls are typically used either to report the position where something was found or to record the position of theListIterator so that another ListIterator with identical position can be created.

It should also come as no surprise that the number returned by nextIndex is always one greater than the number returned by previousIndex. This implies the behavior of the two boundary cases: (1) a call to previousIndex when the cursor is before the initial element returns -1 and (2) a call to nextIndex when the cursor is after the final element returnslist.size(). To make all this concrete, the following is a possible implementation of List.indexOf.

public int indexOf(E e) {

for (ListIterator<E> it = listIterator(); it.hasNext(); )

if (e == null ? it.next() == null : e.equals(it.next()))

return it.previousIndex();

// Element not found

return -1;

}

Note that the indexOf method returns it.previousIndex() even though it is traversing the list in the forward direction. The reason is that it.nextIndex() would return the index of the element we are about to examine, and we want to return the index of the element we just examined.

The Iterator interface provides the remove operation to remove the last element returned by next from the Collection. For ListIterator, this operation removes the last element returned by next or previous. The ListIterator interface provides two additional operations to modify the list — set and add. The set method overwrites the last element returned bynext or previous with the specified element. The following polymorphic algorithm uses set to replace all occurrences of one specified value with another.

public static <E> void replace(List<E> list, E val, E newVal) {

for (ListIterator<E> it = list.listIterator(); it.hasNext(); )

if (val == null ? it.next() == null : val.equals(it.next()))

it.set(newVal);

}

The only bit of trickiness in this example is the equality test between val and it.next. You need to special-case a val value of null to prevent a NullPointerException.

The add method inserts a new element into the list immediately before the current cursor position. This method is illustrated in the following polymorphic algorithm to replace all occurrences of a specified value with the sequence of values contained in the specified list.

public static <E>

void replace(List<E> list, E val, List<? extends E> newVals) {

for (ListIterator<E> it = list.listIterator(); it.hasNext(); ){

if (val == null ? it.next() == null : val.equals(it.next())) {

it.remove();

for (E e : newVals)

it.add(e);

}

}

}

## Range-View Operation

The range-view operation, subList(int fromIndex, int toIndex), returns a List view of the portion of this list whose indices range from fromIndex, inclusive, to toIndex, exclusive. This *half-open range* mirrors the typical for loop.

for (int i = fromIndex; i < toIndex; i++) {

...

}

As the term *view* implies, the returned List is backed up by the List on which subList was called, so changes in the former are reflected in the latter.

This method eliminates the need for explicit range operations (of the sort that commonly exist for arrays). Any operation that expects a List can be used as a range operation by passing asubList view instead of a whole List. For example, the following idiom removes a range of elements from a List.

list.subList(fromIndex, toIndex).clear();

Similar idioms can be constructed to search for an element in a range.

int i = list.subList(fromIndex, toIndex).indexOf(o);

int j = list.subList(fromIndex, toIndex).lastIndexOf(o);

Note that the preceding idioms return the index of the found element in the subList, not the index in the backing List.

Any polymorphic algorithm that operates on a List, such as the replace and shuffle examples, works with the List returned by subList.

Here's a polymorphic algorithm whose implementation uses subList to deal a hand from a deck. That is, it returns a new List (the "hand") containing the specified number of elements taken from the end of the specified List (the "deck"). The elements returned in the hand are removed from the deck.

public static <E> List<E> dealHand(List<E> deck, int n) {

int deckSize = deck.size();

List<E> handView = deck.subList(deckSize - n, deckSize);

List<E> hand = new ArrayList<E>(handView);

handView.clear();

return hand;

}

Note that this algorithm removes the hand from the *end* of the deck. For many common List implementations, such as ArrayList, the performance of removing elements from the end of the list is substantially better than that of removing elements from the beginning.

The following is [a program](https://docs.oracle.com/javase/tutorial/collections/interfaces/examples/Deal.java) that uses the dealHand method in combination with Collections.shuffle to generate hands from a normal 52-card deck. The program takes two command-line arguments: (1) the number of hands to deal and (2) the number of cards in each hand.

import java.util.\*;

public class Deal {

public static void main(String[] args) {

if (args.length < 2) {

System.out.println("Usage: Deal hands cards");

return;

}

int numHands = Integer.parseInt(args[0]);

int cardsPerHand = Integer.parseInt(args[1]);

// Make a normal 52-card deck.

String[] suit = new String[] {

"spades", "hearts",

"diamonds", "clubs"

};

String[] rank = new String[] {

"ace", "2", "3", "4",

"5", "6", "7", "8", "9", "10",

"jack", "queen", "king"

};

List<String> deck = new ArrayList<String>();

for (int i = 0; i < suit.length; i++)

for (int j = 0; j < rank.length; j++)

deck.add(rank[j] + " of " + suit[i]);

// Shuffle the deck.

Collections.shuffle(deck);

if (numHands \* cardsPerHand > deck.size()) {

System.out.println("Not enough cards.");

return;

}

for (int i = 0; i < numHands; i++)

System.out.println(dealHand(deck, cardsPerHand));

}

public static <E> List<E> dealHand(List<E> deck, int n) {

int deckSize = deck.size();

List<E> handView = deck.subList(deckSize - n, deckSize);

List<E> hand = new ArrayList<E>(handView);

handView.clear();

return hand;

}

}

Running the program produces output like the following.

% java Deal 4 5

[8 of hearts, jack of spades, 3 of spades, 4 of spades,

king of diamonds]

[4 of diamonds, ace of clubs, 6 of clubs, jack of hearts,

queen of hearts]

[7 of spades, 5 of spades, 2 of diamonds, queen of diamonds,

9 of clubs]

[8 of spades, 6 of diamonds, ace of spades, 3 of hearts,

ace of hearts]

Although the subList operation is extremely powerful, some care must be exercised when using it. The semantics of the List returned by subList become undefined if elements are added to or removed from the backing List in any way other than via the returned List. Thus, it's highly recommended that you use the List returned by subList only as a transient object — to perform one or a sequence of range operations on the backing List. The longer you use the subList instance, the greater the probability that you'll compromise it by modifying the backing List directly or through another subList object. Note that it is legal to modify a sublist of a sublist and to continue using the original sublist (though not concurrently).

## List Algorithms

Most polymorphic algorithms in the Collections class apply specifically to List. Having all these algorithms at your disposal makes it very easy to manipulate lists. Here's a summary of these algorithms, which are described in more detail in the [Algorithms](https://docs.oracle.com/javase/tutorial/collections/algorithms/index.html) section.

* sort — sorts a List using a merge sort algorithm, which provides a fast, stable sort. (A *stable sort* is one that does not reorder equal elements.)
* shuffle — randomly permutes the elements in a List.
* reverse — reverses the order of the elements in a List.
* rotate — rotates all the elements in a List by a specified distance.
* swap — swaps the elements at specified positions in a List.
* replaceAll — replaces all occurrences of one specified value with another.
* fill — overwrites every element in a List with the specified value.
* copy — copies the source List into the destination List.
* binarySearch — searches for an element in an ordered List using the binary search algorithm.
* indexOfSubList — returns the index of the first sublist of one List that is equal to another.
* lastIndexOfSubList — returns the index of the last sublist of one List that is equal to another.

[« Previous](https://docs.oracle.com/javase/tutorial/collections/interfaces/set.html) • [Trail](https://docs.oracle.com/javase/tutorial/collections/TOC.html) • [Next »](https://docs.oracle.com/javase/tutorial/collections/interfaces/queue.html)

**Queue Implementations**

The Queue implementations are grouped into general-purpose and concurrent implementations.

**General-Purpose Queue Implementations**

As mentioned in the previous section, LinkedList implements the Queue interface, providing first in, first out (FIFO) queue operations for add, poll, and so on.

The [PriorityQueue](https://docs.oracle.com/javase/8/docs/api/java/util/PriorityQueue.html) class is a priority queue based on the *heap* data structure. This queue orders elements according to the order specified at construction time, which can be the elements' natural ordering or the ordering imposed by an explicit Comparator.

The queue retrieval operations — poll, remove, peek, and element — access the element at the head of the queue. The *head of the queue* is the least element with respect to the specified ordering. If multiple elements are tied for least value, the head is one of those elements; ties are broken arbitrarily.

PriorityQueue and its iterator implement all of the optional methods of the Collection and Iterator interfaces. The iterator provided in method iterator is not guaranteed to traverse the elements of the PriorityQueue in any particular order. For ordered traversal, consider using Arrays.sort(pq.toArray()).

**Concurrent Queue Implementations**

The java.util.concurrent package contains a set of synchronized Queue interfaces and classes. [BlockingQueue](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingQueue.html) extends Queue with operations that wait for the queue to become nonempty when retrieving an element and for space to become available in the queue when storing an element. This interface is implemented by the following classes:

* [LinkedBlockingQueue](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/LinkedBlockingQueue.html) — an optionally bounded FIFO blocking queue backed by linked nodes
* [ArrayBlockingQueue](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ArrayBlockingQueue.html) — a bounded FIFO blocking queue backed by an array
* [PriorityBlockingQueue](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/PriorityBlockingQueue.html) — an unbounded blocking priority queue backed by a heap
* [DelayQueue](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/DelayQueue.html) — a time-based scheduling queue backed by a heap
* [SynchronousQueue](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/SynchronousQueue.html) — a simple rendezvous mechanism that uses the BlockingQueue interface

In JDK 7, [TransferQueue](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/TransferQueue.html) is a specialized BlockingQueue in which code that adds an element to the queue has the option of waiting (blocking) for code in another thread to retrieve the element. TransferQueue has a single implementation:

* [LinkedTransferQueue](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/LinkedTransferQueue.html) — an unbounded TransferQueue based on linked nodes

# 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. The Deque interface is a richer abstract data type than both Stack and Queue because it implements both stacks and queues at the same time. The [Deque](https://docs.oracle.com/javase/8/docs/api/java/util/Deque.html) interface, defines methods to access the elements at both ends of the Deque instance. Methods are provided to insert, remove, and examine the elements. Predefined classes like [ArrayDeque](https://docs.oracle.com/javase/8/docs/api/java/util/ArrayDeque.html) and[LinkedList](https://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html) implement the Deque interface.

Note that the Deque interface can be used both as last-in-first-out stacks and first-in-first-out queues. The methods given in the Deque interface are divided into three parts:

## Insert

The addfirst and offerFirst methods insert elements at the beginning of the Deque instance. The methods addLast and offerLast insert elements at the end of the Dequeinstance. When the capacity of the Deque instance is restricted, the preferred methods are offerFirst and offerLast because addFirst might fail to throw an exception if it is full.

## Remove

The removeFirst and pollFirst methods remove elements from the beginning of the Deque instance. The removeLast and pollLast methods remove elements from the end. The methods pollFirst and pollLast return null if the Deque is empty whereas the methods removeFirst and removeLast throw an exception if the Deque instance is empty.

## Retrieve

The methods getFirst and peekFirst retrieve the first element of the Deque instance. These methods dont remove the value from the Deque instance. Similarly, the methods getLastand peekLast retrieve the last element. The methods getFirst and getLast throw an exception if the deque instance is empty whereas the methods peekFirst and peekLast returnNULL.

The 12 methods for insertion, removal and retieval of Deque elements are summarized in the following table:

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

In addition to these basic methods to insert,remove and examine a Deque instance, the Deque interface also has some more predefined methods. One of these isremoveFirstOccurence, this method removes the first occurence of the specified element if it exists in the Deque instance. If the element does not exist then the Deque instance remains unchanged. Another similar method is removeLastOccurence; this method removes the last occurence of the specified element in the Deque instance. The return type of these methods isboolean, and they return true if the element exists in the Deque instance.

[« Previous](https://docs.oracle.com/javase/tutorial/collections/interfaces/queue.html) • [Trail](https://docs.oracle.com/javase/tutorial/collections/TOC.html) • [Next »](https://docs.oracle.com/javase/tutorial/collections/interfaces/map.html)

# The Map Interface

A [Map](https://docs.oracle.com/javase/8/docs/api/java/util/Map.html) is an object that maps keys to values. A map cannot contain duplicate keys: Each key can map to at most one value. It models the mathematical *function* abstraction. The Mapinterface includes methods for basic operations (such as put, get, remove, containsKey, containsValue, size, and empty), bulk operations (such as putAll and clear), and collection views (such as keySet, entrySet, and values).

The Java platform contains three general-purpose Map implementations: [HashMap](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html), [TreeMap](https://docs.oracle.com/javase/8/docs/api/java/util/TreeMap.html), and [LinkedHashMap](https://docs.oracle.com/javase/8/docs/api/java/util/LinkedHashMap.html). Their behavior and performance are precisely analogous to HashSet,TreeSet, and LinkedHashSet, as described in [The Set Interface](https://docs.oracle.com/javase/tutorial/collections/interfaces/set.html) section.

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)))

Again, these are but a few examples of how to use the new JDK 8 APIs. For in-depth coverage of lambda expressions and aggregate operations see the lesson entitled [Aggregate Operations](https://docs.oracle.com/javase/tutorial/collections/streams/index.html).

## Map Interface Basic Operations

The basic operations of Map (put, get, containsKey, containsValue, size, and isEmpty) behave exactly like their counterparts in Hashtable. The [following program](https://docs.oracle.com/javase/tutorial/collections/interfaces/examples/Freq.java)generates a frequency table of the words found in its argument list. The frequency table maps each word to the number of times it occurs in the argument list.

import java.util.\*;

public class Freq {

public static void main(String[] args) {

Map<String, Integer> m = new HashMap<String, Integer>();

// Initialize frequency table from command line

for (String a : args) {

Integer freq = m.get(a);

m.put(a, (freq == null) ? 1 : freq + 1);

}

System.out.println(m.size() + " distinct words:");

System.out.println(m);

}

}

The only tricky thing about this program is the second argument of the put statement. That argument is a conditional expression that has the effect of setting the frequency to one if the word has never been seen before or one more than its current value if the word has already been seen. Try running this program with the command:

java Freq if it is to be it is up to me to delegate

The program yields the following output.

8 distinct words:

{to=3, delegate=1, be=1, it=2, up=1, if=1, me=1, is=2}

Suppose you'd prefer to see the frequency table in alphabetical order. All you have to do is change the implementation type of the Map from HashMap to TreeMap. Making this four-character change causes the program to generate the following output from the same command line.

8 distinct words:

{be=1, delegate=1, if=1, is=2, it=2, me=1, to=3, up=1}

Similarly, you could make the program print the frequency table in the order the words first appear on the command line simply by changing the implementation type of the map toLinkedHashMap. Doing so results in the following output.

8 distinct words:

{if=1, it=2, is=2, to=3, be=1, up=1, me=1, delegate=1}

This flexibility provides a potent illustration of the power of an interface-based framework.

Like the [Set](https://docs.oracle.com/javase/8/docs/api/java/util/Set.html)and [List](https://docs.oracle.com/javase/8/docs/api/java/util/List.html)interfaces, Map strengthens the requirements on the equals and hashCode methods so that two Map objects can be compared for logical equality without regard to their implementation types. Two Map instances are equal if they represent the same key-value mappings.

By convention, all general-purpose Map implementations provide constructors that take a Map object and initialize the new Map to contain all the key-value mappings in the specified Map. This standard Map conversion constructor is entirely analogous to the standard Collection constructor: It allows the caller to create a Map of a desired implementation type that initially contains all of the mappings in another Map, regardless of the other Map's implementation type. For example, suppose you have a Map, named m. The following one-liner creates a new HashMapinitially containing all of the same key-value mappings as m.

Map<K, V> copy = new HashMap<K, V>(m);

## Map Interface Bulk Operations

The clear operation does exactly what you would think it could do: It removes all the mappings from the Map. The putAll operation is the Map analogue of the Collection interface'saddAll operation. In addition to its obvious use of dumping one Map into another, it has a second, more subtle use. Suppose a Map is used to represent a collection of attribute-value pairs; the putAll operation, in combination with the Map conversion constructor, provides a neat way to implement attribute map creation with default values. The following is a static factory method that demonstrates this technique.

static <K, V> Map<K, V> newAttributeMap(Map<K, V>defaults, Map<K, V> overrides) {

Map<K, V> result = new HashMap<K, V>(defaults);

result.putAll(overrides);

return result;

}

## 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();

The idiom for iterating over values is analogous. Following is the idiom for iterating over key-value pairs.

for (Map.Entry<KeyType, ValType> e : m.entrySet())

System.out.println(e.getKey() + ": " + e.getValue());

At first, many people worry that these idioms may be slow because the Map has to create a new Collection instance each time a Collection view operation is called. Rest easy: There's no reason that a Map cannot always return the same object each time it is asked for a given Collection view. This is precisely what all the Map implementations in java.util do.

With all three Collection views, calling an Iterator's remove operation removes the associated entry from the backing Map, assuming that the backing Map supports element removal to begin with. This is illustrated by the preceding filtering idiom.

With the entrySet view, it is also possible to change the value associated with a key by calling a Map.Entry's setValue 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.

The Collection views support element removal in all its many forms — remove, removeAll, retainAll, and clear operations, as well as the Iterator.remove operation. (Yet again, this assumes that the backing Map supports element removal.)

The Collection views *do not* support element addition under any circumstances. It would make no sense for the keySet and values views, and it's unnecessary for the entrySet view, because the backing Map's put and putAll methods provide the same functionality.

## Fancy Uses of Collection Views: Map Algebra

When applied to the Collection views, bulk operations (containsAll, removeAll, and retainAll) are surprisingly potent tools. For starters, suppose you want to know whether oneMap is a submap of another — that is, whether the first Map contains all the key-value mappings in the second. The following idiom does the trick.

if (m1.entrySet().containsAll(m2.entrySet())) {

...

}

Along similar lines, suppose you want to know whether two Map objects contain mappings for all of the same keys.

if (m1.keySet().equals(m2.keySet())) {

...

}

Suppose you have a Map that represents a collection of attribute-value pairs, and two Sets representing required attributes and permissible attributes. (The permissible attributes include the required attributes.) The following snippet determines whether the attribute map conforms to these constraints and prints a detailed error message if it doesn't.

static <K, V> boolean validate(Map<K, V> attrMap, Set<K> requiredAttrs, Set<K>permittedAttrs) {

boolean valid = true;

Set<K> attrs = attrMap.keySet();

if (! attrs.containsAll(requiredAttrs)) {

Set<K> missing = new HashSet<K>(requiredAttrs);

missing.removeAll(attrs);

System.out.println("Missing attributes: " + missing);

valid = false;

}

if (! permittedAttrs.containsAll(attrs)) {

Set<K> illegal = new HashSet<K>(attrs);

illegal.removeAll(permittedAttrs);

System.out.println("Illegal attributes: " + illegal);

valid = false;

}

return valid;

}

Suppose you want to know all the keys common to two Map objects.

Set<KeyType>commonKeys = new HashSet<KeyType>(m1.keySet());

commonKeys.retainAll(m2.keySet());

A similar idiom gets you the common values.

All the idioms presented thus far have been nondestructive; that is, they don't modify the backing Map. Here are a few that do. Suppose you want to remove all of the key-value pairs that oneMap has in common with another.

m1.entrySet().removeAll(m2.entrySet());

Suppose you want to remove from one Map all of the keys that have mappings in another.

m1.keySet().removeAll(m2.keySet());

What happens when you start mixing keys and values in the same bulk operation? Suppose you have a Map, managers, that maps each employee in a company to the employee's manager. We'll be deliberately vague about the types of the key and the value objects. It doesn't matter, as long as they're the same. Now suppose you want to know who all the "individual contributors" (or nonmanagers) are. The following snippet tells you exactly what you want to know.

Set<Employee> individualContributors = new HashSet<Employee>(managers.keySet());

individualContributors.removeAll(managers.values());

Suppose you want to fire all the employees who report directly to some manager, Simon.

Employee simon = ... ;

managers.values().removeAll(Collections.singleton(simon));

Note that this idiom makes use of Collections.singleton, a static factory method that returns an immutable Set with the single, specified element.

Once you've done this, you may have a bunch of employees whose managers no longer work for the company (if any of Simon's direct-reports were themselves managers). The following code will tell you which employees have managers who no longer works for the company.

Map<Employee, Employee> m = new HashMap<Employee, Employee>(managers);

m.values().removeAll(managers.keySet());

Set<Employee> slackers = m.keySet();

This example is a bit tricky. First, it makes a temporary copy of the Map, and it removes from the temporary copy all entries whose (manager) value is a key in the original Map. Remember that the original Map has an entry for each employee. Thus, the remaining entries in the temporary Map comprise all the entries from the original Map whose (manager) values are no longer employees. The keys in the temporary copy, then, represent precisely the employees that we're looking for.

There are many more idioms like the ones contained in this section, but it would be impractical and tedious to list them all. Once you get the hang of it, it's not that difficult to come up with the right one when you need it.

## Multimaps

A *multimap* is like a Map but it can map each key to multiple values. The Java Collections Framework doesn't include an interface for multimaps because they aren't used all that commonly. It's a fairly simple matter to use a Map whose values are List instances as a multimap. This technique is demonstrated in the next code example, which reads a word list containing one word per line (all lowercase) and prints out all the anagram groups that meet a size criterion. An *anagram group* is a bunch of words, all of which contain exactly the same letters but in a different order. The program takes two arguments on the command line: (1) the name of the dictionary file and (2) the minimum size of anagram group to print out. Anagram groups containing fewer words than the specified minimum are not printed.

There is a standard trick for finding anagram groups: For each word in the dictionary, alphabetize the letters in the word (that is, reorder the word's letters into alphabetical order) and put an entry into a multimap, mapping the alphabetized word to the original word. For example, the word bad causes an entry mapping abd into bad to be put into the multimap. A moment's reflection will show that all the words to which any given key maps form an anagram group. It's a simple matter to iterate over the keys in the multimap, printing out each anagram group that meets the size constraint.

[The following program](https://docs.oracle.com/javase/tutorial/collections/interfaces/examples/Anagrams.java) is a straightforward implementation of this technique.

import java.util.\*;

import java.io.\*;

public class Anagrams {

public static void main(String[] args) {

int minGroupSize = Integer.parseInt(args[1]);

// Read words from file and put into a simulated multimap

Map<String, List<String>> m = new HashMap<String, List<String>>();

try {

Scanner s = new Scanner(new File(args[0]));

while (s.hasNext()) {

String word = s.next();

String alpha = alphabetize(word);

List<String> l = m.get(alpha);

if (l == null)

m.put(alpha, l=new ArrayList<String>());

l.add(word);

}

} catch (IOException e) {

System.err.println(e);

System.exit(1);

}

// Print all permutation groups above size threshold

for (List<String> l : m.values())

if (l.size() >= minGroupSize)

System.out.println(l.size() + ": " + l);

}

private static String alphabetize(String s) {

char[] a = s.toCharArray();

Arrays.sort(a);

return new String(a);

}

}

Running this program on a 173,000-word dictionary file with a minimum anagram group size of eight produces the following output.

9: [estrin, inerts, insert, inters, niters, nitres, sinter,

triens, trines]

8: [lapse, leaps, pales, peals, pleas, salep, sepal, spale]

8: [aspers, parses, passer, prases, repass, spares, sparse,

spears]

10: [least, setal, slate, stale, steal, stela, taels, tales,

teals, tesla]

8: [enters, nester, renest, rentes, resent, tenser, ternes,

treens]

8: [arles, earls, lares, laser, lears, rales, reals, seral]

8: [earings, erasing, gainers, reagins, regains, reginas,

searing, seringa]

8: [peris, piers, pries, prise, ripes, speir, spier, spire]

12: [apers, apres, asper, pares, parse, pears, prase, presa,

rapes, reaps, spare, spear]

11: [alerts, alters, artels, estral, laster, ratels, salter,

slater, staler, stelar, talers]

9: [capers, crapes, escarp, pacers, parsec, recaps, scrape,

secpar, spacer]

9: [palest, palets, pastel, petals, plates, pleats, septal,

staple, tepals]

9: [anestri, antsier, nastier, ratines, retains, retinas,

retsina, stainer, stearin]

8: [ates, east, eats, etas, sate, seat, seta, teas]

8: [carets, cartes, caster, caters, crates, reacts, recast,

traces]

Many of these words seem a bit bogus, but that's not the program's fault; they're in the dictionary file. Here's the [dictionary file](https://docs.oracle.com/javase/tutorial/collections/interfaces/examples/dictionary.txt) we used. It was derived from the Public Domain ENABLE benchmark reference word list.

[« Previous](https://docs.oracle.com/javase/tutorial/collections/interfaces/deque.html) • [Trail](https://docs.oracle.com/javase/tutorial/collections/TOC.html) • [Next »](https://docs.oracle.com/javase/tutorial/collections/interfaces/order.html)

# Object Ordering

A List l may be sorted as follows.

Collections.sort(l);

If the List consists of String elements, it will be sorted into alphabetical order. If it consists of Date elements, it will be sorted into chronological order. How does this happen? String andDate both implement the [Comparable](https://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html) interface. Comparable implementations provide a *natural ordering* for a class, which allows objects of that class to be sorted automatically. Thefollowing table summarizes some of the more important Java platform classes that implement Comparable.

|  |  |
| --- | --- |
| **Classes Implementing Comparable** | |
| **Class** | **Natural Ordering** |
| Byte | Signed numerical |
| Character | Unsigned numerical |
| Long | Signed numerical |
| Integer | Signed numerical |
| Short | Signed numerical |
| Double | Signed numerical |
| Float | Signed numerical |
| BigInteger | Signed numerical |
| BigDecimal | Signed numerical |
| Boolean | Boolean.FALSE < Boolean.TRUE |
| File | System-dependent lexicographic on path name |
| String | Lexicographic |
| Date | Chronological |
| CollationKey | Locale-specific lexicographic |

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). Similarly, Collections.sort(list, comparator) will throw a ClassCastException if you try to sort a list whose elements cannot be compared to one another using the comparator. 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.

This is all you really need to know about the Comparable interface if you just want to sort lists of comparable elements or to create sorted collections of them. The next section will be of interest to you if you want to implement your own Comparable type.

## Writing Your Own Comparable Types

The Comparable interface consists of the following method.

public interface Comparable<T> {

public int compareTo(T o);

}

The compareTo method compares the receiving object with the specified object and returns a negative integer, 0, or a positive integer depending on whether the receiving object is less than, equal to, or greater than the specified object. If the specified object cannot be compared to the receiving object, the method throws a ClassCastException.

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));

}

}

To keep the preceding example short, the class is somewhat limited: It doesn't support middle names, it demands both a first and a last name, and it is not internationalized in any way. Nonetheless, it illustrates the following important points:

* Name objects are *immutable*. All other things being equal, immutable types are the way to go, especially for objects that will be used as elements in Sets or as keys in Maps. These collections will break if you modify their elements or keys while they're in the collection.
* The constructor checks its arguments for null. This ensures that all Name objects are well formed so that none of the other methods will ever throw a NullPointerException.
* The hashCode method is redefined. This is essential for any class that redefines the equals method. (Equal objects must have equal hash codes.)
* The equals method returns false if the specified object is null or of an inappropriate type. The compareTo method throws a runtime exception under these circumstances. Both of these behaviors are required by the general contracts of the respective methods.
* The toString method has been redefined so it prints the Name in human-readable form. This is always a good idea, especially for objects that are going to get put into collections. The various collection types' toString methods depend on the toString methods of their elements, keys, and values.

Since this section is about element ordering, let's talk a bit more about Name's compareTo method. It implements the standard name-ordering algorithm, where last names take precedence over first names. This is exactly what you want in a natural ordering. It would be very confusing indeed if the natural ordering were unnatural!

Take a look at how compareTo is implemented, because it's quite typical. First, you compare the most significant part of the object (in this case, the last name). Often, you can just use the natural ordering of the part's type. In this case, the part is a String and the natural (lexicographic) ordering is exactly what's called for. If the comparison results in anything other than zero, which represents equality, you're done: You just return the result. If the most significant parts are equal, you go on to compare the next most-significant parts. In this case, there are only two parts — first name and last name. If there were more parts, you'd proceed in the obvious fashion, comparing parts until you found two that weren't equal or you were comparing the least-significant parts, at which point you'd return the result of the comparison.

Just to show that it all works, here's [a program that builds a list of names and sorts them](https://docs.oracle.com/javase/tutorial/collections/interfaces/examples/NameSort.java).

import java.util.\*;

public class NameSort {

public static void main(String[] args) {

Name nameArray[] = {

new Name("John", "Smith"),

new Name("Karl", "Ng"),

new Name("Jeff", "Smith"),

new Name("Tom", "Rich")

};

List<Name> names = Arrays.asList(nameArray);

Collections.sort(names);

System.out.println(names);

}

}

If you run this program, here's what it prints.

[Karl Ng, Tom Rich, Jeff Smith, John Smith]

There are four restrictions on the behavior of the compareTo method, which we won't go into now because they're fairly technical and boring and are better left in the API documentation. It's really important that all classes that implement Comparable obey these restrictions, so read the documentation for Comparable if you're writing a class that implements it. Attempting to sort a list of objects that violate the restrictions has undefined behavior. Technically speaking, these restrictions ensure that the natural ordering is a *total order* on the objects of a class that implements it; this is necessary to ensure that sorting is well defined.

## Comparators

What if you want to sort some objects in an order other than their natural ordering? Or what if you want to sort some objects that don't implement Comparable? To do either of these things, you'll need to provide a [Comparator](https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html) — an object that encapsulates an ordering. Like the Comparable interface, the Comparator interface consists of a single method.

public interface Comparator<T> {

int compare(T o1, T o2);

}

The compare method compares its two arguments, returning a negative integer, 0, or a positive integer depending on whether the first argument is less than, equal to, or greater than the second. If either of the arguments has an inappropriate type for the Comparator, the compare method throws a ClassCastException.

Much of what was said about Comparable applies to Comparator as well. Writing a compare method is nearly identical to writing a compareTo method, except that the former gets both objects passed in as arguments. The compare method has to obey the same four technical restrictions as Comparable's compareTo method for the same reason — a Comparator must induce a total order on the objects it compares.

Suppose you have a class called Employee, as follows.

public class Employee implements Comparable<Employee> {

public Name name() { ... }

public int number() { ... }

public Date hireDate() { ... }

...

}

Let's assume that the natural ordering of Employee instances is Name ordering (as defined in the previous example) on employee name. Unfortunately, the boss has asked for a list of employees in order of seniority. This means we have to do some work, but not much. The following program will produce the required list.

import java.util.\*;

public class EmpSort {

static final Comparator<Employee> SENIORITY\_ORDER =

new Comparator<Employee>() {

public int compare(Employee e1, Employee e2) {

return e2.hireDate().compareTo(e1.hireDate());

}

};

// Employee database

static final Collection<Employee> employees = ... ;

public static void main(String[] args) {

List<Employee> e = new ArrayList<Employee>(employees);

Collections.sort(e, SENIORITY\_ORDER);

System.out.println(e);

}

}

The Comparator in the program is reasonably straightforward. It relies on the natural ordering of Date applied to the values returned by the hireDate accessor method. Note that theComparator passes the hire date of its second argument to its first rather than vice versa. The reason is that the employee who was hired most recently is the least senior; sorting in the order of hire date would put the list in reverse seniority order. Another technique people sometimes use to achieve this effect is to maintain the argument order but to negate the result of the comparison.

// Don't do this!!

return -r1.hireDate().compareTo(r2.hireDate());

You should always use the former technique in favor of the latter because the latter is not guaranteed to work. The reason for this is that the compareTo method can return any negative int if its argument is less than the object on which it is invoked. There is one negative int that remains negative when negated, strange as it may seem.

-Integer.MIN\_VALUE == Integer.MIN\_VALUE

The Comparator in the preceding program works fine for sorting a List, but it does have one deficiency: It cannot be used to order a sorted collection, such as TreeSet, because it generates an ordering that is *not compatible with* equals. This means that this Comparator equates objects that the equals method does not. In particular, any two employees who were hired on the same date will compare as equal. When you're sorting a List, this doesn't matter; but when you're using the Comparator to order a sorted collection, it's fatal. If you use thisComparator to insert multiple employees hired on the same date into a TreeSet, only the first one will be added to the set; the second will be seen as a duplicate element and will be ignored.

To fix this problem, simply tweak the Comparator so that it produces an ordering that *is compatible with* equals. In other words, tweak it so that the only elements seen as equal when using compare are those that are also seen as equal when compared using equals. The way to do this is to perform a two-part comparison (as for Name), where the first part is the one we're interested in — in this case, the hire date — and the second part is an attribute that uniquely identifies the object. Here the employee number is the obvious attribute. This is the Comparatorthat results.

static final Comparator<Employee> SENIORITY\_ORDER =

new Comparator<Employee>() {

public int compare(Employee e1, Employee e2) {

int dateCmp = e2.hireDate().compareTo(e1.hireDate());

if (dateCmp != 0)

return dateCmp;

return (e1.number() < e2.number() ? -1 :

(e1.number() == e2.number() ? 0 : 1));

}

};

One last note: You might be tempted to replace the final return statement in the Comparator with the simpler:

return e1.number() - e2.number();

Don't do it unless you're *absolutely sure* no one will ever have a negative employee number! This trick does not work in general because the signed integer type is not big enough to represent the difference of two arbitrary signed integers. If i is a large positive integer and j is a large negative integer, i - j will overflow and will return a negative integer. The resulting comparatorviolates one of the four technical restrictions we keep talking about (transitivity) and produces horrible, subtle bugs. This is not a purely theoretical concern; people get burned by it.

A [SortedSet](https://docs.oracle.com/javase/8/docs/api/java/util/SortedSet.html) 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 Set operations, 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

The code for the SortedSet interface follows.

public interface SortedSet<E> extends Set<E> {

// Range-view

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

SortedSet<E> headSet(E toElement);

SortedSet<E> tailSet(E fromElement);

// Endpoints

E first();

E last();

// Comparator access

Comparator<? super E> comparator();

}

## 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 SortedSet implementations returns a string containing all the elements of the sorted set, in order.

## Standard Constructors

By convention, all general-purpose Collection implementations provide a standard conversion constructor that takes a Collection; SortedSet implementations are no exception. InTreeSet, this constructor creates an instance that sorts its elements according to their natural ordering. This was probably a mistake. It would have been better to check dynamically to see whether the specified collection was a SortedSet instance and, if so, to sort the new TreeSet according to the same criterion (comparator or natural ordering). Because TreeSet took the approach that it did, it also provides a constructor that takes a SortedSet and returns a new TreeSet containing the same elements sorted according to the same criterion. Note that it is the compile-time type of the argument, not its runtime type, that determines which of these two constructors is invoked (and whether the sorting criterion is preserved).

SortedSet implementations also provide, by convention, a constructor that takes a Comparator and returns an empty set sorted according to the specified Comparator. If null is passed to this constructor, it returns a set that sorts its elements according to their natural ordering.

## Range-view Operations

The range-view operations are somewhat analogous to those provided by the List interface, but there is one big difference. Range views of a sorted set remain valid even if the backing sorted set is modified directly. This is feasible because the endpoints of a range view of a sorted set are absolute points in the element space rather than specific elements in the backing collection, as is the case for lists. 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 therange-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.

Sorted sets provide three range-view operations. The first, subSet, takes two endpoints, like subList. Rather than indices, the endpoints are objects and must be comparable to the elements in the sorted set, using the Set's Comparator or the natural ordering of its elements, whichever the Set uses to order itself. Like subList, the range is half open, including its low endpoint but excluding the high one.

Thus, the following line of code tells you how many words between "doorbell" and "pickle", including "doorbell" but excluding "pickle", are contained in a SortedSet of strings called dictionary:

int count = dictionary.subSet("doorbell", "pickle").size();

In like manner, the following one-liner removes all the elements beginning with the letter f.

dictionary.subSet("f", "g").clear();

A similar trick can be used to print a table telling you how many words begin with each letter.

for (char ch = 'a'; ch <= 'z'; ) {

String from = String.valueOf(ch++);

String to = String.valueOf(ch);

System.out.println(from + ": " + dictionary.subSet(from, to).size());

}

Suppose you want to view a *closed interval*, which contains both of its endpoints, instead of an open interval. If the element type allows for the calculation of the successor of a given value in the element space, merely request the subSet from lowEndpoint to successor(highEndpoint). Although it isn't entirely obvious, the successor of a string s in String's natural ordering is s + "\0" — that is, s with a null character appended.

Thus, the following one-liner tells you how many words between "doorbell" and "pickle", including doorbell *and* pickle, are contained in the dictionary.

count = dictionary.subSet("doorbell", "pickle\0").size();

A similar technique can be used to view an *open interval*, which contains neither endpoint. The open-interval view from lowEndpoint to highEndpoint is the half-open interval fromsuccessor(lowEndpoint) to highEndpoint. Use the following to calculate the number of words between "doorbell" and "pickle", excluding both.

count = dictionary.subSet("doorbell\0", "pickle").size();

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");

## Endpoint Operations

The SortedSet interface contains operations to return the first and last elements in the sorted set, not surprisingly called first and last. In addition to their obvious uses, last allows a workaround for a deficiency in the SortedSet interface. One thing you'd like to do with a SortedSet is to go into the interior of the Set and iterate forward or backward. It's easy enough to go forward from the interior: Just get a tailSet and iterate over it. Unfortunately, there's no easy way to go backward.

The following idiom obtains the first element that is less than a specified object o in the element space.

Object predecessor = ss.headSet(o).last();

This is a fine way to go one element backward from a point in the interior of a sorted set. It could be applied repeatedly to iterate backward, but this is very inefficient, requiring a lookup for each element returned.

## Comparator Accessor

The SortedSet interface contains an accessor method called comparator that returns the Comparator used to sort the set, or null if the set is sorted according to the *natural ordering*of its elements. This method is provided so that sorted sets can be copied into new sorted sets with the same ordering. It is used by the SortedSet constructor described [previously](https://docs.oracle.com/javase/tutorial/collections/interfaces/sorted-set.html#constructor).

[« Previous](https://docs.oracle.com/javase/tutorial/collections/interfaces/order.html) • [Trail](https://docs.oracle.com/javase/tutorial/collections/TOC.html) • [Next »](https://docs.oracle.com/javase/tutorial/collections/interfaces/sorted-map.html)

[« Previous](https://docs.oracle.com/javase/tutorial/collections/interfaces/sorted-set.html) • [Trail](https://docs.oracle.com/javase/tutorial/collections/TOC.html) • [Next »](https://docs.oracle.com/javase/tutorial/collections/interfaces/summary.html)

# The SortedMap Interface

A [SortedMap](https://docs.oracle.com/javase/8/docs/api/java/util/SortedMap.html) 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 SortedMapcreation. Natural ordering and Comparators 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

The following interface is the Map analog of [SortedSet](https://docs.oracle.com/javase/8/docs/api/java/util/SortedSet.html).

public interface SortedMap<K, V> extends Map<K, V>{

Comparator<? super K> comparator();

SortedMap<K, V> subMap(K fromKey, K toKey);

SortedMap<K, V> headMap(K toKey);

SortedMap<K, V> tailMap(K fromKey);

K firstKey();

K lastKey();

}

## Map Operations

The operations SortedMap inherits from Map behave identically on sorted maps and normal maps with two exceptions:

* The Iterator returned by the iterator operation on any of the sorted map's Collection views traverse the collections in order.
* The arrays returned by the Collection views' toArray operations contain the keys, values, or entries in order.

Although it isn't guaranteed by the interface, the toString method of the Collection views in all the Java platform's SortedMap implementations returns a string containing all the elements of the view, in order.

## Standard Constructors

By convention, all general-purpose Map implementations provide a standard conversion constructor that takes a Map; SortedMap implementations are no exception. In TreeMap, this constructor creates an instance that orders its entries according to their keys' natural ordering. This was probably a mistake. It would have been better to check dynamically to see whether the specified Map instance was a SortedMap and, if so, to sort the new map according to the same criterion (comparator or natural ordering). Because TreeMap took the approach it did, it also provides a constructor that takes a SortedMap and returns a new TreeMap containing the same mappings as the given SortedMap, sorted according to the same criterion. Note that it is the compile-time type of the argument, not its runtime type, that determines whether the SortedMap constructor is invoked in preference to the ordinary map constructor.

SortedMap implementations also provide, by convention, a constructor that takes a Comparator and returns an empty map sorted according to the specified Comparator. If null is passed to this constructor, it returns a Map that sorts its mappings according to their keys' natural ordering.

## Comparison to SortedSet

Because this interface is a precise Map analog of SortedSet, all the idioms and code examples in [The SortedSet Interface](https://docs.oracle.com/javase/tutorial/collections/interfaces/sorted-set.html) section apply to SortedMap with only trivial modifications.

[« Previous](https://docs.oracle.com/javase/tutorial/collections/interfaces/sorted-set.html) • [Trail](https://docs.oracle.com/javase/tutorial/collections/TOC.html) • [Next »](https://docs.oracle.com/javase/tutorial/collections/interfaces/summary.html)

**Summary of Interfaces**

The core collection interfaces are the foundation of the Java Collections Framework.

The Java Collections Framework hierarchy consists of two distinct interface trees:

* The first tree starts with the Collection interface, which provides for the basic functionality used by all collections, such as add and remove methods. Its subinterfaces — Set,List, and Queue — provide for more specialized collections.
* The Set interface does not allow duplicate elements. This can be useful for storing collections such as a deck of cards or student records. The Set interface has a subinterface,SortedSet, that provides for ordering of elements in the set.
* The List interface provides for an ordered collection, for situations in which you need precise control over where each element is inserted. You can retrieve elements from a List by their exact position.
* The Queue interface enables additional insertion, extraction, and inspection operations. Elements in a Queue are typically ordered in on a FIFO basis.
* The Deque interface enables insertion, deletion, and inspection operations at both the ends. Elements in a Deque can be used in both LIFO and FIFO.
* The second tree starts with the Map interface, which maps keys and values similar to a Hashtable.
* Map's subinterface, SortedMap, maintains its key-value pairs in ascending order or in an order specified by a Comparator.

These interfaces allow collections to be manipulated independently of the details of their representation.

# Questions and Exercises: Interfaces

## Questions

1. At the beginning of this lesson, you learned that the core collection interfaces are organized into two distinct inheritance trees. One interface in particular is not considered to be a trueCollection, and therefore sits at the top of its own tree. What is the name of this interface?
2. Each interface in the collections framework is declared with the <E> syntax, which tells you that it is generic. When you declare a Collection instance, what is the advantage of specifying the type of objects that it will contain?
3. What interface represents a collection that does not allow duplicate elements?
4. What interface forms the root of the collections hierarchy?
5. What interface represents an ordered collection that may contain duplicate elements?
6. What interface represents a collection that holds elements prior to processing?
7. What interface repesents a type that maps keys to values?
8. What interface represents a double-ended queue?
9. Name three different ways to iterate over the elements of a List.
10. True or False: Aggregate operations are mutative operations that modify the underlying collection.

## Exercises

1. Write a program that prints its arguments in random order. Do not make a copy of the argument array. Demonstrate how to print out the elements using both streams and the traditional enhanced for statement.
2. Take the [FindDups](https://docs.oracle.com/javase/tutorial/collections/interfaces/examples/FindDups.java)example and modify it to use a SortedSet instead of a Set. Specify a Comparator so that case is ignored when sorting and identifying set elements.
3. Write a method that takes a List<String> and applies [String.trim](https://docs.oracle.com/javase/8/docs/api/java/lang/String.html#trim--) to each element.
4. Consider the four core interfaces, Set, List, Queue, and Map. For each of the following four assignments, specify which of the four core interfaces is best-suited, and explain how to use it to implement the assignment.
   1. Whimsical Toys Inc (WTI) needs to record the names of all its employees. Every month, an employee will be chosen at random from these records to receive a free toy.
   2. WTI has decided that each new product will be named after an employee but only first names will be used, and each name will be used only once. Prepare a list of unique first names.
   3. WTI decides that it only wants to use the most popular names for its toys. Count up the number of employees who have each first name.
   4. WTI acquires season tickets for the local lacrosse team, to be shared by employees. Create a waiting list for this popular sport.

[Check your answers.](https://docs.oracle.com/javase/tutorial/collections/interfaces/QandE/answers.html)