Previous in the Series		Current Tutorial		Next in the Series
Iterating over the Elements of a Collection	←	**Extending Collection with List**	→	Extending Collection with Set, SortedSet and NavigableSet

Extending Collection with List

Exploring the List Interface

The `List` interface brings two new functionalities to plain collections.

- This order in which you iterate over the elements of a list is always the same, and it respects the order in which the elements have been added to this list.
- The elements of a list have an index.

Choosing your Implementation of the List Interface

In this tutorial

Iterating over the Elements of a List

While the `Collection` interface as no specific implementation in the Collections Framework (it relies on the implementations of its sub-interfaces), the `List` interface has 2: `ArrayList` and `LinkedList`. As you may guess, the first one is built on an internal array, and the second on a doubly-linked list.

Is one of these implementation better than the other? If you are not sure which one to choose, then your best choice is probably `ArrayList`.

What was true for linked lists when computing was invented in the 60's does not hold anymore, and the capacity of linked lists to outperform arrays on insertion and deletion operations is greatly diminished by modern hardware, CPU caches, and pointer chasing. Iterating over the elements of an `ArrayList` is much faster that over the elements of a `LinkedList`, mainly due to pointer chasing and CPU cache misses.

There are still cases where a linked list is faster than an array. A doubly-linked list can access its first and last element faster than an `ArrayList` can. This is the main use case that makes `LinkedList` better than `ArrayList`. So if your application needs a Last In, First Out (LIFO, covered later in this tutorial) stack, or a First In, First Out (FIFO, also covered later) waiting queue, then

choosing a linked list is probably your best choice.

On the other hand, if you plan to iterate through the elements of your list, or to access them randomly by their index, then the `ArrayList` is probably your best bet.

Accessing the Elements Using an Index

The `List` interface brings several methods to the `Collection` interface, that deal with indexes.

Accessing a Single Object

- `add(index, element)`: inserts the given object at the `index`, adjusting the index if there are remaining elements
- `get(index)`: returns the object at the given `index`
- `set(index, element)`: replaces the element at the given index with the new element
- `remove(index)`: removes the element at the given `index`, adjusting the index of the remaining elements.

Calling these methods work only for valid indexes. If the given index is not valid then an `IndexOutOfBoundsException` exception will be thrown.

Finding the Index of an Object

The methods `indexOf(element)` and `lastIndexOf(element)` return the index of the given element in the list, or -1 if the element is not found.

Getting a SubList

The `subList(start, end)` returns a list consisting of the elements between indexes `start` and `end - 1`. If the indexes are invalid then an `IndexOutOfBoundsException` exception will be thrown.

Note that the returned list is a view on the main list. Thus, any modification operation on the sublist is reflected on the main list and vice-versa.

For instance, you can clear a portion of the content of a list with the following pattern:

```
List<String> strings = new ArrayList<
System.out.println(strings);
strings.subList(2, 5).clear();
System.out.println(strings);
```

Running this code gives you the following result:

```
[0, 1, 2, 3, 4, 5]
[0, 1, 5]
```

Inserting a Collection

The last pattern of this list is about inserting a collection at a given indexes: `addAll(int index, Collection collection)`.

Sorting the Elements of a List

A list keeps its elements in a known order. This is the main difference with a plain collection. So it makes sense to sort the elements of a list. This is the reason why a `sort()` method has been added to the `List` interface in JDK 8.

In Java SE 7 and earlier, you could sort the elements of your `List` by calling `Collections.sort()` and pass you list as an argument, along with a comparator if needed.

Starting with Java SE 8 you can call `sort()` directly on your list and pass your comparator as an argument. There is no overload of this method that does not take any argument. Calling it with a null comparator will assume that the elements of your `List` implement `Comparable`, you will get a `ClassCastException` if this is not the case.

If you do not like calling methods will null arguments (and you are right!), you can still call it with `Comparator.naturalOrder()` to achieve the same result.

Iterating over the Elements of a List

The `List` interface gives you one more way to iterate over its elements with the `ListIterator`. You can get such an iterator by calling `listIterator()`. You can call this method with no argument, or pass an integer index to it. In that case, the iteration will start at this index.

The `ListIterator` interface extends the regular `Iterator` that you already know. It adds several methods to it.

- `hasPrevious()` and `previous()`: to iterate in the descending order rather than the ascending order
- `nextIndex()` and `previousIndex()`: to get the index of the element that will be returned by the next `next()` call, or the next `previous()` call
- `set(element)`: to update the last element returned by `next()` or `previous()`. If neither of these methods have been called on this iterator then an `IllegalStateException` is raised.

Let us see this `set()` method in action:

```
List<String> numbers = Arrays.asList(
for (ListIterator<String> iterator =
    String nextElement = iterator.ne
```