

Theory: Iterator and Iterable

🕒 24 minutes

0 / 5 problems solved

Skip this topic

Start practicing

1106 users solved this topic. Latest completion was less than a minute ago.

As you know, there is a **for-each** loop and the `forEach` method to iterate over elements of a collection. Both of them provide a simple and unified way to process different types of collections. In this topic, you will learn more about why they work and how to use them.

§1. Being iterable

The *Java Standard Library* has a special interface called `Iterable`. Implementing this interface allows objects of a class to be targets of the **for-each** loop. If you think that the `Collection` interface extends this to be **iterable**, you are absolutely right.

```
1 public interface Collection<E> extends Iterable<E> { /* methods */ }
```

The `Collection` interface extends `Iterable`, but `Map` does not.

Due to this, any collection class (`List`, `Queue`, `Set`) can be considered as `Iterable`.

```
1 Iterable<String> iterable = List.of("first", "second", "third");
```

The order of elements when iterating is specific to a chosen collection. For lists, the order is the same as the order of its elements.

```
1 List<String> strings = List.of("first", "second", "third");
2
3 // the loop prints "first", "second", and then "third"
4 for (String elem : strings) {
5     System.out.println(elem);
6 }
```

The same is true for the `forEach` method that can take the reference to a method:

```
1 // the loop prints "first", "second", and then "third"
2 strings.forEach(System.out::println);
```

For sets, the situation is different, since ordinary sets are not ordered. As an experiment, you can replace the list with the following set:

```
1 Set<String> strings = Set.of("first", "second", "third");
```

The result may be different each time the program starts.

The `Iterable` interface provides three generic methods. In this topic, we will consider two of them:

- `Iterator<T> iterator()` returns a special object which can iterate over the collection;
- `void forEach(Consumer<T> action)` takes an action and executes it on each element of the collection, it can be used together with lambda expressions and method references.

All collections that inherit the `Collection` interface have these methods.

§2. Using iterators

The `Iterator<T>` is a universal mechanism for iterating over collections regardless of their structure. It takes elements in the order provided by the collection. In some sense, it is like a moveable "pointer" to an element of the collection.

Current topic:

Iterator and Iterable ...

Topic depends on:

✗ List ...

✗ Set ...

✗ Queue and Stack ...

Table of contents:

1 Iterator and Iterable

§1. Being iterable

§2. Using iterators

§3. An iterator for lists

Feedback & Comments

The **iterator** allows you to remove elements from the underlying collection but you cannot do it using a **for-each** loop.

Some methods of the `Iterator<E>` interface:

- `boolean hasNext()` returns `true` if the iteration has more elements, and `false` otherwise;
- `E next()` returns the next element in the iteration;
- `void remove()` removes the last element returned by this iterator from the collection.

The **for-each** loop uses the first two methods under the hood.

It is also possible to directly access and use an iterator of a collection. The typical usage includes three steps:

1. Check the collection has next element.
2. Obtain the next element.
3. Process the obtained element.

For example, let's remove all elements less than 10 from a sorted set.

```
1  Set<Long> set = new TreeSet<>(); // sorted set
2  set.add(10L);
3  set.add(5L);
4  set.add(18L);
5  set.add(14L);
6  set.add(9L);
7
8  System.out.println(set); // [5, 9, 10, 14, 18]
9
10
11  Iterator<Long> iter = set.iterator();
12
13  while (iter.hasNext()) {
14      Long current = iter.next();
15      if (current < 10L) {
16          iter.remove();
17      }
18  }
19
20  System.out.println(set); // [10, 14, 18]
```

In this example, the **iterator** gets elements according to the sorting order and successfully removes some of them.

§3. An iterator for lists

There is a special iterator for lists called `ListIterator` which extends the common `Iterator` interface. It allows the programmer to traverse the list in either direction, modify the list during iteration, and obtain the current position in the list.

In addition to standard `Iterator`'s methods, this iterator provides the following methods:

- `nextIndex()` returns the index of the element that would be returned by invoking `next()`;
- `hasPrevious()` returns `true` if the list has more previous elements;
- `previous()` returns the previous element in the list and moves the cursor position backwards;
- `previousIndex()` returns the index of the element that would be returned by invoking `previous()`;
- `set(E element)` replaces the last element returned by `next()` or `previous()` with the specified element;

- `add(E element)` inserts the specified element into the list immediately before the element that would be returned by `next()`, and after the element that would be returned by `previous()`.

Here is an example of how it works:

```
1 List<Integer> list = List.of(1, 2, 3, 4);
2 ListIterator<Integer> iterator = list.listIterator(); // only for lists!
3
4 // go to the last element
5 while (iterator.hasNext()) { iterator.next(); }
6
7 // print elements in the backward order with their indexes
8 while (iterator.hasPrevious()) {
9     int previousIndex = iterator.previousIndex();
10
11     int element = iterator.previous();
12
13     System.out.println(element + " on " + previousIndex);
14 }
15 }
```

This code prints numbers in the backward order with their indexes.

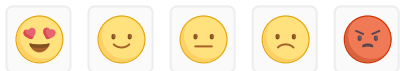
```
1 4 on 3
2 3 on 2
3 2 on 1
4 1 on 0
```

If you invoke `previous()` before `previousIndex()` the result will differ since `previous()` changes the state of the iterator: the current position.

This concludes our consideration of iterators.

 Report a typo

102 users liked this theory. 1 didn't like it. What about you?



Start practicing

[Comments \(1\)](#)

[Hints \(0\)](#)

[Useful links \(0\)](#)

[Show discussion](#)