

# Iterating over the Elements of a Collection

## Using the for-each Pattern

Your simplest choice to iterate over the elements of a collection is to use the for-each pattern.

```
1 | Collection<String> strings = List.of("one", "two", "three");
2 |
3 | for (String element: strings) {
4 |     System.out.println(element);
5 | }
```

Running this code produces the following result:

```
1 | one
2 | two
3 | three
```

This pattern is very efficient, as long as you only need to read the elements of your collection. The [Iterator](#) pattern allows to remove some of the elements of your collection while you are iterating over them. If you need to do that, then you want to use the [Iterator](#) pattern.

## Using an Iterator on a Collection

Iterating over the elements of a collection uses a special object, an instance of the [Iterator](#) interface. You can get an [Iterator](#) object from any extension of the [Collection](#) interface. The [iterator\(\)](#) method is defined on the [Iterable](#) interface, extended by the [Collection](#) interface, and further extended by all the interfaces of the collection hierarchy.

Iterating over the elements of a collection using this object is a two-steps process.

1. First you need to check if there are more elements to be visited with the [hasNext\(\)](#) method
2. Then you can advance to the next element with the [next\(\)](#) method.

If you call the [next\(\)](#) method but there are no more elements in the collection, you will get a [NoSuchElementException](#). Calling [hasNext\(\)](#) is not mandatory, it is there to help you to make sure that there is indeed a next element.

Here is the pattern:

```
1 | Collection<String> strings = List.of("one", "two", "three", "four");
2 | for (Iterator<String> iterator = strings.iterator(); iterator.hasNext();) {
3 |     String element = iterator.next();
4 |     if (element.length() == 3) {
5 |         System.out.println(element);
6 |     }
7 | }
```

This code produces the following result:

```
1 | one
2 | two
```

The [Iterator](#) interface has a third method: [remove\(\)](#). Calling this method removes the current element from the collection. There are cases though where this method is not supported, it will throw an [UnsupportedOperationException](#). Quite obviously, calling [remove\(\)](#) on an immutable collection cannot work, so this is one of the cases. The implementation of [Iterator](#) you get from [ArrayList](#), [LinkedList](#) and [HashSet](#) all support this remove operation.

## Updating a Collection While Iterating over It

If you happen to modify the content of a collection while iterating over it, you may get a [ConcurrentModificationException](#). Getting this exception may be a little confusing, because this exception is also used in concurrent programming. In the context of the Collections Framework, you may get it without touching multithreaded programming.

The following code throws a [ConcurrentModificationException](#).

```
1  Collection<String> strings = new ArrayList<>();
2  strings.add("one");
3  strings.add("two");
4  strings.add("three");
5
6  Iterator<String> iterator = strings.iterator();
7  while (iterator.hasNext()) {
8
9      String element = iterator.next();
10     strings.remove(element);
11 }
```

If what you need is to remove the elements of a collection that satisfy a given criteria, you may use the [removeIf\(.,\)](#) method.

## Implementing the Iterable Interface

Now that you saw what an iterator is in the Collection Framework, you can create a simple implementation of the [Iterable](#) interface.

Suppose you need to create a **Range** class that models a range of integers between two limits. All you need to do is iterate from the first integer to the last one.

You can implement the [Iterable](#) interface with a record, a feature introduced in Java SE 16:

```
1  record Range(int start, int end) implements Iterable<Integer> {
2
3      @Override
4      public Iterator<Integer> iterator() {
5          return new Iterator<>() {
```

```

6         private int index = start;
7
8         @Override
9         public boolean hasNext() {
10             return index < end;
11         }
12
13         @Override
14         public Integer next() {
15             if (index > end) {
16                 throw new NoSuchElementException("'" + index);
17             }
18             int currentIndex = index;
19             index++;
20             return currentIndex;
21         }
22     };
23 }
24 }

```

You can do the same with a plain class, in case your application does not support Java SE 16 yet. Note that the code of the implementation of [Iterator](#) is exactly the same.

```

1  class Range implements Iterable<Integer> {
2
3      private final int start;
4      private final int end;
5
6      public Range(int start, int end) {
7          this.start = start;
8          this.end = end;
9      }
10
11      @Override
12      public Iterator<Integer> iterator() {
13          return new Iterator<>() {
14              private int index = start;
15
16              @Override
17              public boolean hasNext() {
18                  return index < end;
19              }
20
21              @Override
22              public Integer next() {
23                  if (index > end) {
24

```

```

24         throw new NoSuchElementException("'" + index);
25     }
26     int currentIndex = index;
27     index++;
28     return currentIndex;
29 }
30 }
31 };
32 }

```

In both cases, you can use an instance of **Range** in a for-each statement, since it implements [Iterable](#):

```

1  for (int i : new Range1(0, 5)) {
2      System.out.println("i = " + i);
3  }

```

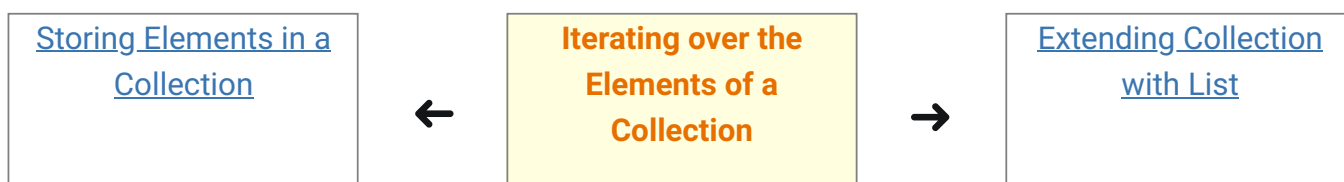
Running this code gives you the following result:

```

1  i = 0
2  i = 1
3  i = 2
4  i = 3
5  i = 4

```

**Last update:** September 14, 2021



[Home](#) > [Tutorials](#) > [The Collections Framework](#) > Iterating over the Elements of a Collection