Storing Elements in a Collection ← **Iterating over the Elements of a Collection** → Extending Collection with List

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.

```java
Collection<String> strings = List.of("one", "two", "three");

for (String element: strings) {
    System.out.println(element);
}
```

Running this code produces the following result:

```
one
two
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:

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

This code produces the following result:

```
one
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`.

```java
Collection<String> strings = new ArrayList<>();
strings.add("one");
strings.add("two");
strings.add("three");

Iterator<String> iterator = strings.iterator();
while (iterator.hasNext()) {

    String element = iterator.next();
    strings.remove(element);
}
```

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:

```java
record Range(int start, int end) implements Iterable<Integer> {

    @Override
    public Iterator<Integer> iterator() {
        return new Iterator<>() {
            private int index = start;

            @Override
            public boolean hasNext() {
                return index < end;
            }

            @Override
            public Integer next() {
                if (index > end) {
                    throw new NoSuchElementException("" + index);
                }
                int currentIndex = index;
                index++;
                return currentIndex;
            }
        };
    }
}
```

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.

```
class Range implements Iterable<Integer> {

    private final int start;
    private final int end;

    public Range(int start, int end) {
        this.start = start;
        this.end = end;
    }

    @Override
    public Iterator<Integer> iterator() {
        return new Iterator<>() {
            private int index = start;

            @Override
            public boolean hasNext() {
                return index < end;
            }

            @Override
            public Integer next() {
                if (index > end) {
                    throw new NoSuchElementException("" + index);
                }
                int currentIndex = index;
                index++;
                return currentIndex;
            }
        };
    }
}
```

In both cases, you can use an instance of `Range` in a for-each statement, since it implements `Iterable`:

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

Running this code gives you the following result:

```
i = 0
i = 1
i = 2
i = 3
i = 4
```

Last update: *September 14, 2021*

Storing Elements in a Collection ← **Iterating over the Elements of a Collection** → Extending Collection with List

Home > Tutorials > The Collections Framework > Iterating over the Elements of a Collection