Avoiding the ConcurrentModificationException in Java

Last updated: January 9, 2024

Written by: baeldung (https://www.baeldung.com/author/baeldung)

Reviewed by: Predrag Marić (https://www.baeldung.com/editor/predrag-author)

Java Concurrency (https://www.baeldung.com/category/java/java-concurrency)

Exception (https://www.baeldung.com/tag/exception)

1. Introduction

In this article, we'll take a look at the *ConcurrentModificationException (https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/ConcurrentModificationException.html)* class.

First, we'll give an explanation how it works, and then prove it by using a test for triggering it.

Finally, we'll try out some workarounds by using practical examples.

2. Triggering a *ConcurrentModificationException*

Essentially, the *ConcurrentModificationException* is used to **fail-fast when something we are iterating on is modified.** Let's prove this with a simple test:

```
@Test(expected = ConcurrentModificationException.class)
public void whilstRemovingDuringIteration_shouldThrowException() throws
InterruptedException {

    List<Integer> integers = newArrayList(1, 2, 3);

    for (Integer integer : integers) {
        integers.remove(1);
    }
}
```

As we can see, before finishing our iteration we are removing an element. That's what triggers the exception.

3. Solutions

Sometimes, we might actually want to remove elements from a collection whilst iterating. If this is the case, then there are some solutions.

3.1. Using an Iterator Directly

A *for-each* loop uses an *Iterator* behind the scenes but is less verbose. However, if we refactored our previous test to use an *Iterator,* we will have access to additional methods, such as *remove().* Let's try using this method to modify our list instead:

```
for (Iterator<Integer> iterator = integers.iterator(); iterator.hasNext();) {
    Integer integer = iterator.next();
    if(integer == 2) {
        iterator.remove();
    }
}
```

Now we will notice that there is no exception. The reason for this is that the *remove()* method does not cause a *ConcurrentModificationException.* It is safe to call while iterating.

3.2. Not Removing During Iteration

If we want to keep our *for-each* loop, then we can. It's just that we need to wait until after iterating before we remove the elements. Let's try this out by adding what we want to remove to a *toRemove* list as we iterate:

```
List<Integer> integers = newArrayList(1, 2, 3);
List<Integer> toRemove = newArrayList();

for (Integer integer : integers) {
    if(integer == 2) {
        toRemove.add(integer);
    }
}
integers.removeAll(toRemove);

assertThat(integers).containsExactly(1, 3);
```

This is another effective way of getting around the problem.

3.3. Using *removeIf()*

Java 8 introduced the *removeIf()* method to the *Collection* interface. This means that if we are working with it, we can use ideas of functional programming to achieve the same results again:

```
List<Integer> integers = newArrayList(1, 2, 3);

integers.removeIf(i -> i == 2);

assertThat(integers).containsExactly(1, 3);
```

This declarative style offers us the least amount of verbosity. However, depending on the use case, we may find other methods more convenient.

3.4. Filtering Using Streams

When diving into the world of functional/declarative programming, we can forget about mutating collections, instead, we can focus on elements that should be actually processed:

```
Collection<Integer> integers = newArrayList(1, 2, 3);

List<String> collected = integers
  .stream()
  .filter(i -> i != 2)
  .map(Object::toString)
  .collect(toList());

assertThat(collected).containsExactly("1", "3");
```

We've done the inverse to our previous example, by providing a predicate for determining elements to include, not exclude. The advantage is that we can chain together other functions alongside the removal. In the example, we use a functional *map()*, but could use even more operations if we want to.

4. Conclusion

In this article we've shown problems that you may encounter if you're removing items from a collection whilst iterating, and also provided some solutions to negate the issue.

The code backing this article is available on GitHub. Once you're **logged in as a Baeldung Pro Member (/members/)**, start learning and coding on the project.

COURSES

ALL COURSES (/COURSES/ALL-COURSES)
BAELDUNG ALL ACCESS (/COURSES/ALL-ACCESS)
BAELDUNG ALL TEAM ACCESS (/COURSES/ALL-ACCESS-TEAM)
LOGIN COURSE PLATFORM (HTTPS://WWW.BAELDUNG.COM/MEMBERS/ACCOUNT)

SERIES

JAVA "BACK TO BASICS" TUTORIAL (/JAVA-TUTORIAL)
LEARN SPRING BOOT SERIES (/SPRING-BOOT)
SPRING TUTORIAL (/SPRING-TUTORIAL)
GET STARTED WITH JAVA (/GET-STARTED-WITH-JAVA-SERIES)
ALL ABOUT STRING IN JAVA (/JAVA-STRING)
SECURITY WITH SPRING (/SECURITY-SPRING)
JAVA COLLECTIONS (/JAVA-COLLECTIONS)

ABOUT