

Theory: Collections and thread-safety

⌚ 5 minutes 0 / 2 problems solved

Skip this topic

Start practicing

630 users solved this topic. Latest completion was about 13 hours ago.

§1. Classic collections and multithreading

As you already know, several threads may access the same data concurrently that often leads to different problems if we do not use some kind of synchronization.

The similar problems occur when multiple threads access a collection:

- most of the classic collections like `ArrayList`, `LinkedList`, `HashMap` and others are non-synchronized and, as a consequence, they do not provide thread-safety;
- there is a set of old collections like `Vector`, `Stack`, and `Hashtable` that are totally synchronized and thread-safe, but they have low performance;
- when one thread is iterating over a standard collection and another thread tries to add a new element to it then we get a runtime exception called `ConcurrentModificationException`.

The following program demonstrates a race condition that appears when two threads add elements to the same collection.

```
1  import java.util.ArrayList;
2  import java.util.List;
3
4  public class NeedOfConcurrentCollectionsDemo {
5
6      public static void main(String[] args) throws InterruptedException {
7          ArrayList<Integer> numbers = new ArrayList<>();
8
9          Thread writer = new Thread(() -> addNumbers(numbers));
10
11          writer.start();
12
13
14
15          addNumbers(numbers); // add number from the main thread
16
17
18
19          writer.join(); // wait for writer thread
20
21
22
23          System.out.println(numbers.size()); // the result can be any
24      }
25
26
27
28
29      private static void addNumbers(ArrayList<Integer> target) {
30
31          for (int i = 0; i < 100_000; i++) {
32
33              target.add(i);
34
35          }
36      }
37  }
```

The expected result is 200000 (100000 + 100000), but in fact, it changes every time you run this code. Some elements of a list are lost.

We started the program three times and get the following results:

Current topic:

[Collections and thread-safety](#) ...

Topic depends on:

✗ [What are collections](#) ...

✗ [Thread synchronization](#) ...

Topic is required for:

[Concurrent queues](#) ...

Table of contents:

[1 Collections and thread-safety](#)

[§1. Classic collections and multithreading](#)

[§2. Concurrent collections](#)

[Feedback & Comments](#)

```
1 | 196797
2 | 154802
3 | 181359
```

You may also try it yourself.

So, it is a bad idea to use standard collections in multithreaded environments without explicit synchronization. Again, though, such synchronization may lead to poor performance and hard-to-find errors in large programs.

§2. Concurrent collections

To avoid all the problems associated with custom synchronization, Java Class Library provides alternative collection implementations that are adapted to be used in multithreaded applications and they are fully thread-safe. You may find them in the `java.util.concurrent` package that includes lists, queues, maps and other collections which make it easier to develop modern Java applications.

These concurrent collections allow you to avoid custom synchronization in many cases as well as they have high performance close to classic collections. Concurrent collections do not use the `synchronized` keyword but rely on more complex synchronization primitives and lock-free algorithm that allows them to be both thread-safe and high-performance. However, if you do not really need multithreading, use classic collections, since they are still more efficient than concurrent ones.

You will learn the different types of concurrent collections later on.

 Report a typo

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



Start practicing

[Comments \(1\)](#)

[Hints \(1\)](#)

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

[Show discussion](#)