Collections and thread-safety

As you already know, several threads may access the same data concurrently.

This often leads to different problems if we do not use some kind of synchronization.

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 non-thread-safe collection and another thread tries to add a new element to it then we get a runtime exception called ConcurrentModificationException;

- when multiple threads write to and read from a non-thread-safe collection, some writes made by one thread may not be visible to the other threads.

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

```
import java.util.ArrayList;

public class NeedOfConcurrentCollectionsDemo {

   public static void main(String[] args) throws InterruptedException {
      ArrayList<Integer> numbers = new ArrayList<>();

      Thread writer = new Thread(() -> addNumbers(numbers));
      writer.start();

      addNumbers(numbers); // add number from the main thread

      writer.join(); // wait for writer thread

      System.out.println(numbers.size()); // the result can be any
   }

   private static void addNumbers(ArrayList<Integer> target) {
      for (int i = 0; i < 100_000; i++) {
         target.add(i);
      }
```

```
    }
}
```

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:

```
196797
154802
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.

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.