CopyOnWriteArrayList

Our guest is the CopyOnWriteArrayList class, a member of
the java.util.concurrent package.

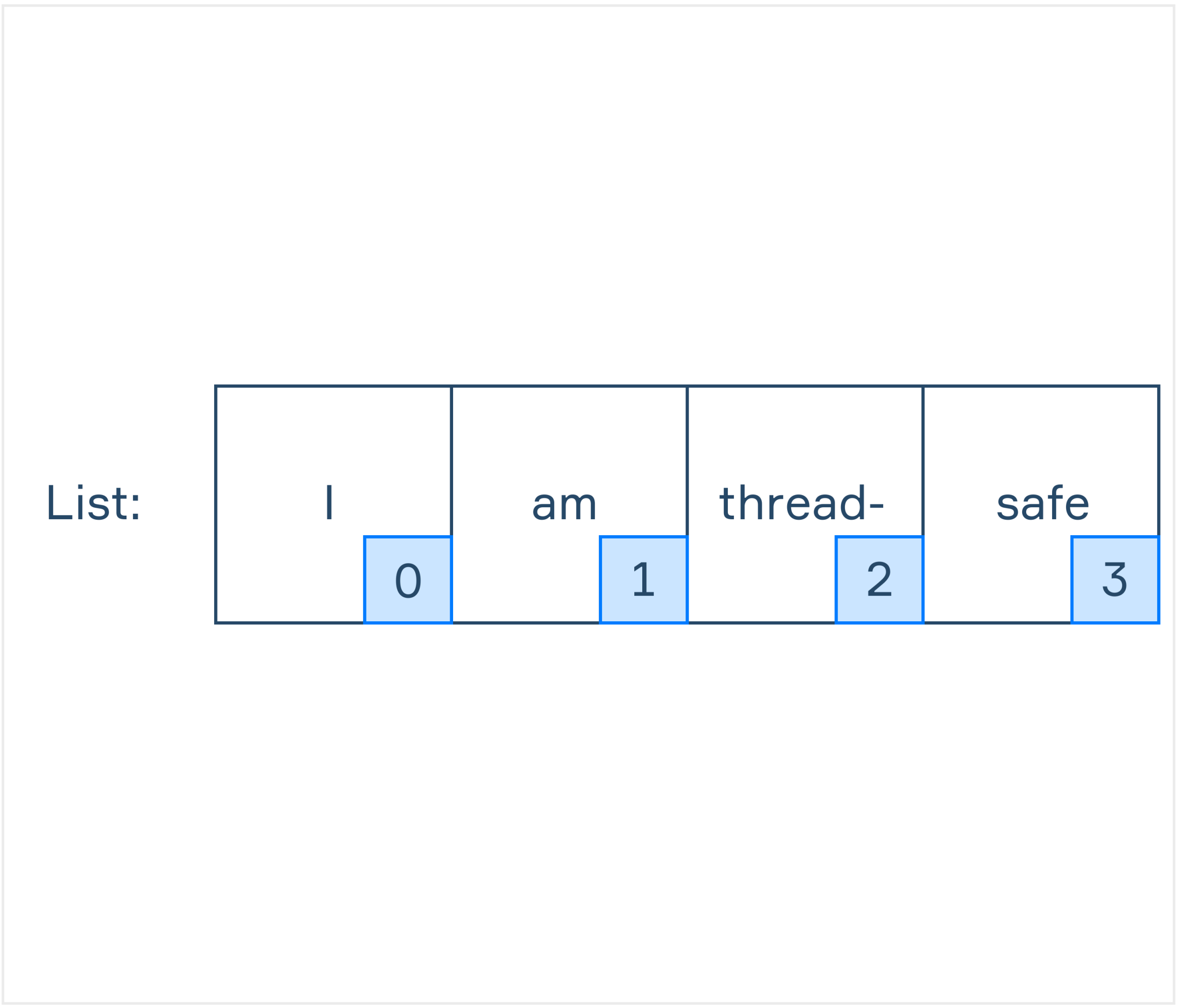
Look closely at the name: ArrayList plus the copy-on-write technique. Together, they give you a thread-safe representation of the ArrayList class.

CopyOnWriteArrayList

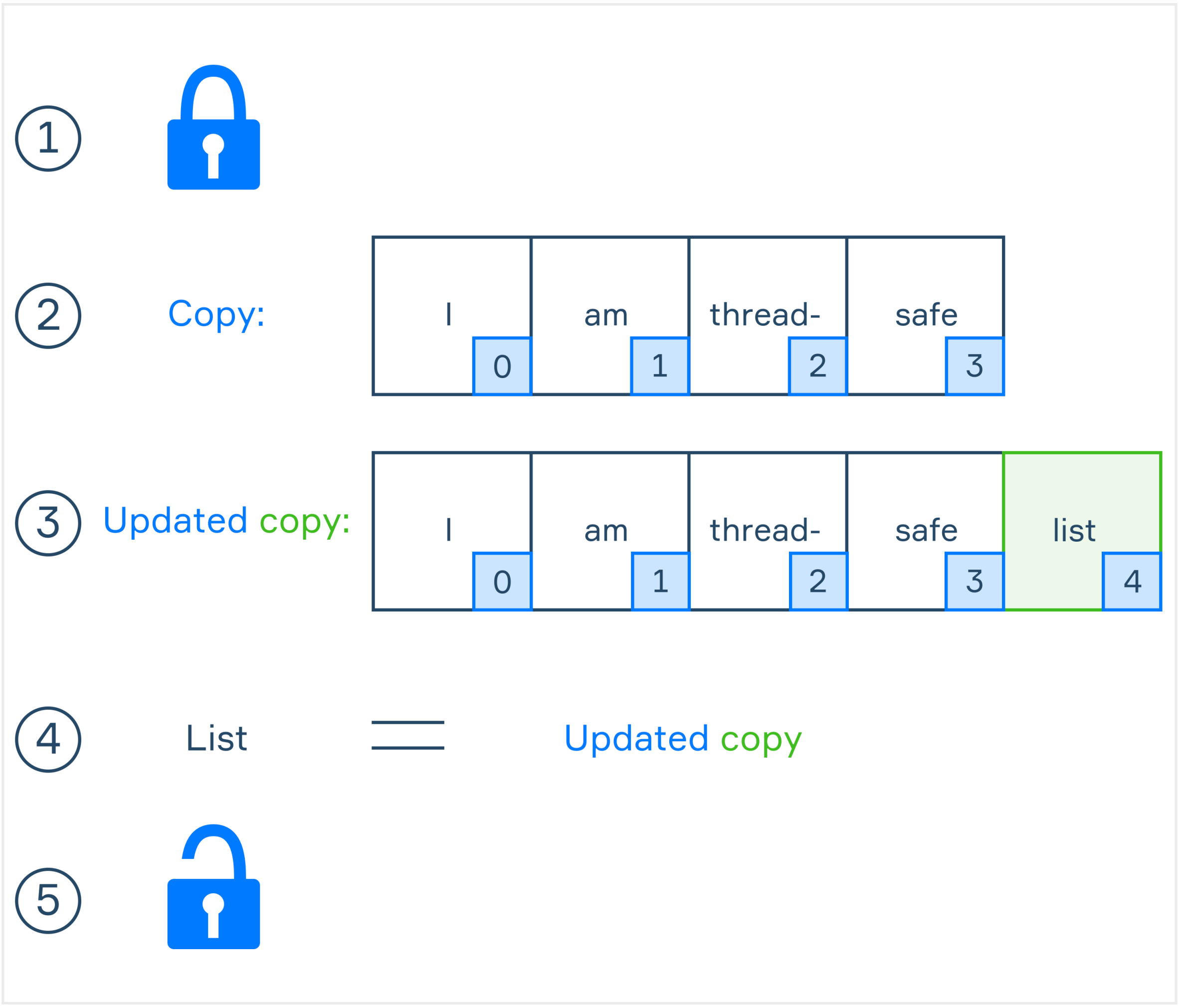
Note that CopyOnWriteArrayList is an "old class" — it has existed since Java 5.

The creation of a CopyOnWriteArrayList looks like the creation of an ArrayList:
List<String> onWriteArrayList = new CopyOnWriteArrayList<>();

Now let's look at how its methods allow thread safety. Imagine that
our CopyOnWriteArrayList looks like this:



We want to add a new element to the end of this list:
onWriteArrayList.add("List!");
These are the steps the add method goes through:



First, a lock is set. Second, a copy of our list will be created.

Third, the copy is updated with a new element. Then, our list is set to the updated copy. The final step is to unlock.

This technique is called **Copy-On-Write** and it ensures thread safety.

All mutative operations (add, set, remove, etc.) use the copy-on-write technique: they create a cloned copy of the original list.
As a consequence, performing many update operations can be very costly.

If you are curious about the details of CopyOnWriteArrayLists, you should know that their underlying structure is an array of Objects.
And now, ladies and gentlemen, the highlight of our program — two threads but only one list.

Two threads, one list
Here is an example with two threads: main and writer. Both of them add numbers to the same CopyOnWriteArrayList.

```
public static void main(String[] args) throws InterruptedException {
    CopyOnWriteArrayList<Integer> onWriteArrayList = new
CopyOnWriteArrayList<>();

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

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

    writer.join(); // wait for the writer thread to finish

    System.out.println(onWriteArrayList.size()); // the result is always the same
}

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

If you try to run this code, the result will always be 200_000.
But what if one thread added numbers while the second thread removed them?

```
public static void main(String[] args) throws InterruptedException {
    CopyOnWriteArrayList<Integer> onWriteArrayList = new
CopyOnWriteArrayList<>();


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

    removeNumbers(onWriteArrayList); // remove numbers from the main thread

    writer.join(); // wait for the writer thread to finish

    System.out.println(onWriteArrayList.size()); // the result is always the same
}

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

private static void removeNumbers(CopyOnWriteArrayList<Integer> list) {
    int index = 0;
    while (index < 100_000) {
        if (!list.isEmpty()) {
            list.remove(0);
            index++;
        }
    }
}
```
In this example, we wanted to add 100_000 numbers and remove 100_000 numbers from the same list. Everything works fine, thanks to the copy-on-write technique. While an element is added, the removal is paused and vice versa. But you can never guarantee that the selected index will exist at any moment. That's why we were removing only the zero-index elements.

If you perform any read operations while updating the list, you will always get the "old" version of the list, that is, how your list looked before the update started.

What about iterators?

Let's consider a simple example:
```
CopyOnWriteArrayList<Integer> onWriteArrayList = new
CopyOnWriteArrayList<>();
onWriteArrayList.add(1);
onWriteArrayList.add(2);
onWriteArrayList.add(3);

Iterator<Integer> iterator = onWriteArrayList.iterator();

onWriteArrayList.add(4);

while(iterator.hasNext()) {
    System.out.print(iterator.next() + " "); // we will see only "1 2 3"
}
```
Do you think we will see three or four elements? The answer is only three. The iterator uses an **immutable snapshot** of the CopyOnWriteArrayList, which is created when the iterator is requested.
Also, because of the immutability, you can't use iterator.remove().

A CopyOnWriteArrayList allows thread-safe iterating over its elements while the underlying list gets modified by other threads.

Conclusion

CopyOnWriteArrayLists come to the rescue when you'd like to use ArrayLists in a multithreading environment.
What to remember:
- A CopyOnWriteArrayList creates a new internal copy for every update operation (with the copy-on-write technique).

- Read operations return the "old" version of the list while an update operation is in progress.
- CopyOnWriteArrayList iterators use an immutable snapshot of the list.
- It's very costly to update CopyOnWriteArrayList often.