The internal workings of `CopyOnWriteArrayList` is a very important topic for Java interviews. In this lesson, we will see how `CopyOnWriteArrayList` provides thread-safety.

> We know that a `CopyOnWriteArrayList` is internally backed by an array, so let's call it **backarray** for the purpose of understanding. Throughout this lesson, wherever we use the term **backarray**, it means the array in which all the elements added to the `CopyOnWriteArrayList` is maintained.

There is a ReentrantLock defined in the `CopyOnWriteArrayList` as shown below:

```
/** The lock protecting all mutators */
final transient ReentrantLock lock = new ReentrantLock();
```

When a new element is added in a `CopyOnWriteArrayList` then the following procedure takes place:

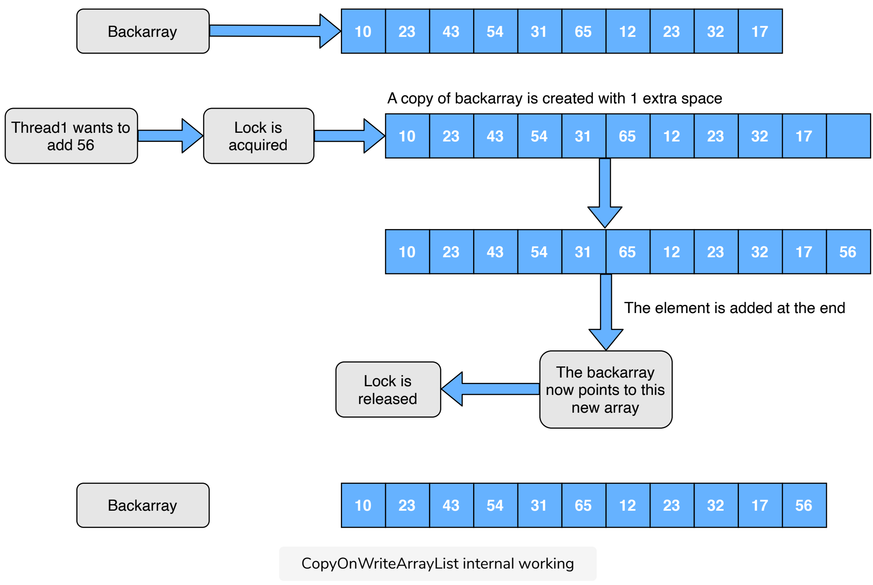
1. The thread that is adding the element acquires a lock on the `lock` object using the `lock.lock()` method. If some other thread tries to add an element to the list, then it will not get access.
2. The thread that has acquired the lock will then make the copy of the **backarray**. So as shown in the below snippet, a new array is created and all the elements from the **backarray** are added to this new array. The size of this new array is one more than the **backarray**.

```
Object[] newElements = Arrays.copyOf(elements, len + 1);
```

3. Now, the element that needs to be added will be added at the end of this newly copied array.

```
newElements[len] = e;
```

3. Finally the **backarray** will now be pointed to this new array and the lock will be released. In this way, a new element is added to the `CopyOnWriteArrayList` in a thread-safe manner.



CopyOnWriteArrayList internal working

You might be wondering what would happen if one thread is trying to add an element to the list and the other is trying to remove an element from the list. When a thread tries to read an element from the list, it will refer to the **backarray**. Multiple threads can read the data from the list without locking. It is only when adding an element that a lock is required, and the **backarray** should be copied. So, `CopyOnWriteArrayList` is ideal for situations where add operations are minimal, and there are mostly read operations.

It is also important to understand why a copy of the **backarray** is made when an element is added and why just acquiring the lock is not sufficient as it will stop other threads from adding at the same time.

The reason for copying the **backarray** is to make traversal synchronized. This is a bit difficult to understand, so let me explain it through an example. But prior to that, let's define one more term to help understand the example. We already know that **backarray** means the array that contains the elements added to the `CopyOnWriteArrayList`. The array that we get by copying the **backarray** will be called **copiedarray**.

Let's say we have created a `CopyOnWriteArrayList` that has five elements. So, the size of the **backarray** is five.

There is a thread, **thread1**, that wants to iterate our list. This thread will create an iterator. The returned iterator provides a snapshot of the state of the list when the iterator was constructed.

The **thread1** is iterating the array, and in the meantime, another thread, **thread2**, comes to add an element to the list. If this thread adds the element to the **backarray**, then `ConcurrentModificationException` will be thrown. To avoid this, **thread2** will create a copy of the **backarray**, and then it will add the new element to this copied array.

The **thread1** will complete its iteration, but it will not be able to see the newly added element. Now, if a new thread, **thread3**, wants to iterate the list, then it will again create the iterator. And this time it will get the snapshot of the **backarray**, which has six elements.

> Please note that when `CopyOnWriteArrayList` creates an iterator, the "snapshot" is a reference to its current array, not a copy of the array.