The **ArrayList** and **LinkedList** data structures that we have discussed until now are not thread-safe. This means that if we are working in an environment where multiple threads are simultaneously adding or removing elements from a list, it may not work as intended. If a thread is iterating over a list and, in the meantime, another thread tries to add an element to the list, then `ConcurrentModificationException` will be thrown.

Now, if we want to use a list in a multi-threaded environment, we have few options. The first option is using a **Vector**. The Vector is a legacy class in which all the methods are synchronized. Since for each operation, such as add or remove, the entire list is locked, it is slow. Hence it is no longer used.

The second option is making our list thread-safe by using the `Collections.synchronizedList()` method, which we have already discussed in this lesson. The problem with this method is that it also locks the entire list for each operation. So, there is no performance benefit.

To overcome these issues **CopyOnWriteArrayList** was introduced. This is a thread-safe list with high performance. We will discuss how it works in the upcoming lesson but in this lesson, we will focus on how it is used.

Creating a CopyOnWriteArrayList#

There are three ways to create a **CopyOnWriteArrayList**:

Using the no-arg constructor#

The default constructor does not take any argument and creates a **CopyOnWriteArrayList** of size zero. **CopyOnWriteArrayList** has an Object array type field named `array`.

```
private transient volatile Object[] array;
```

When we create a list using this constructor, the `array` field is initialized with size zero.

```
public CopyOnWriteArrayList() {
    setArray(new Object[0]);
}

final void setArray(Object[] a) {
   array = a;
}
```

Below is the syntax to create **CopyOnWriteArrayList** using the default constructor.

```
List list = new CopyOnWriteArrayList();
```

Using an existing array#

We can also create a CopyOnWriteArrayList using an existing array. First, a copy of the existing array is made using the `Arrays.copyOf()` method, and then the `array` variable is initialized with this copied array.

```
public CopyOnWriteArrayList(E[] toCopyIn) {
        setArray(Arrays.copyOf(toCopyIn, toCopyIn.length, Object[].class));
    }
```

Using existing Collection#

A **CopyOnWriteArrayList** can also be created using an existing Collection. An array is created using the elements of the passed Collection, and the `array` variable is initialized with this created array.

Inserting elements into a CopyOnWriteArrayList#

Let's discuss some of the ways we can insert elements into a CopyOnWriteArrayList.

Using the `add(E e)` method#

The `add(E e)` method inserts an element at the end of the list. If some other thread is iterating the list while a new element is getting added, then it will not throw `ConcurrentModificationException`. How this happens will be discussed in the internal working lesson.

Using the `add(int index, E element)` method#

We can use this method if we want to add an element to a particular index. The index provided should be greater than zero and less than or equal to the size of the list; otherwise, `IndexOutOfBoundsException` is thrown. When an element is added at an index, the element currently at that position (if any), and any subsequent elements to the right, are shifted to the right.

Using the `addAll(Collection c)` method#

The `addAll(Collection c)` method inserts all the elements present in the provided collection at the end of the list. The elements are inserted in the same order as returned by the iterator of the passed collection.

Using the `addIfAbsent(E e)` method#

The `addIfAbsent()` method adds an element at the end of the list only if the element is not present in the list.

Using the `addAllAbsent(Collection c)` method#

This method appends all of the specified Collection elements that are not already contained in this list, to the end of this list, in the order that the specified Collection's iterator returns them.