

67% completed



Search Course

Java Concurrency
Reference

- Setting-up Threads
- Basic Thread Handling
- Executor Framework
- Executor Implementations
- Thread Pools
- Types of Thread Pools
- An Example: Timer vs ScheduledThreadPool
- ThreadPoolExecutor
- Callable Interface
- Future Interface
- CompletionService Interface
- ThreadLocal
- ThreadLocalRandom
- CountDownLatch
- CyclicBarrier
- Concurrent Collections
- ConcurrentHashMap
- ConcurrentModificationException

Practice Mock Interview →

Java Multithreading for Senior Engineering Interviews / ... / AtomicMarkableReference

AtomicMarkableReference

Learn to use the AtomicMarkableReference class for designing lock-free data structure such as linked-lists.

If you are interviewing, consider buying our **number#1** course for [Java Multithreading Interviews](#).

Overview

The AtomicMarkableReference class is similar to the AtomicStampedReference class in that it holds a reference to an object but instead of an integer stamp it takes in a boolean value, called the mark. Both these fields can be updated atomically either together or individually. One could argue that AtomicStampedReference class would behave similarly to AtomicMarkableReference class if it accepted only two possible values for the integer stamp argument.

The code widget below demonstrates some of the basic operations when working with the AtomicMarkableReference class.

```
Java
main.java
1 • import java.util.concurrent.atomic.AtomicMarkableReference;
2
3 • class Demonstration {
4
5 •     public static void main( String args[] ) {
6
7         Long myLong = new Long(5);
8         Long anotherLong = new Long(7);
9         AtomicMarkableReference<Long> atomicMarkableReference = new AtomicMarkableReference<
10
11         // attempt to change the Long value with the wrong expected mark
12         boolean wasSuccess = atomicMarkableReference.compareAndSet(myLong, anotherLong, true);
```

Example

AtomicMarkableReference can be put to good use when designing lock-free lists, such as a set representation backed by a linked-list. In this lesson, we'll not go through the implementation of a lock-free list but discuss parts of the implementation where the class AtomicMarkableReference is applicable.

Consider the SimpleNode class which represents a node in the linked list. The class uses an AtomicReference for the next node in the list, so that it can be updated atomically. The goal is to keep the list lock-free and update the next of the node correctly when a node is deleted. We'll create a set-up that creates a race condition and causes the list to reach an inconsistent state.

Say we have three nodes linked as follows:

NodeA → NodeB → NodeC → null

Consider the SimpleNode class which represents a node in the linked list. The class uses an AtomicReference for the next node in the list, so that it can be updated atomically. The goal is to keep the list lock-free and update the next of the node correctly when a node is deleted. We'll create a set-up that creates a race condition and causes the list to reach an inconsistent state.

Say we have three nodes linked as follows:

NodeA → NodeB → NodeC → null

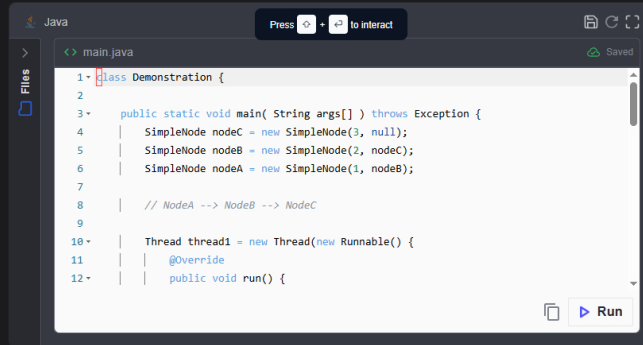
Say we have two threads thread1 and thread2, each one trying to delete NodeA and NodeB respectively. The following sequence of events can cause the NodeC to not be deleted:

- thread1 stores the next of NodeA, which points to NodeB and the next of NodeB which points to NodeC in the variables expected and newNext respectively. These variables serve as the expected and the new value for the compareAndSet() method of the AtomicReference.
- At this point thread1 is suspended. We simulate that by sleeping the thread.
- thread2 starts and attempts to delete NodeC. thread1 computes the expected value for the next of NodeB, which is NodeC itself and the new next value which should be null, since NodeC is the last node of the list.
- thread2 continues and successfully deletes NodeC.
- thread1 wakes up and proceeds to execute the compareAndSet() call, which should

succeed because NodeA still points to NodeB (the expected value) but the new `next` points to NodeC which has been deleted.

6. The program ends in a state where NodeC is not deleted and remains part of the list.

The above scenario is depicted in the code widget below:

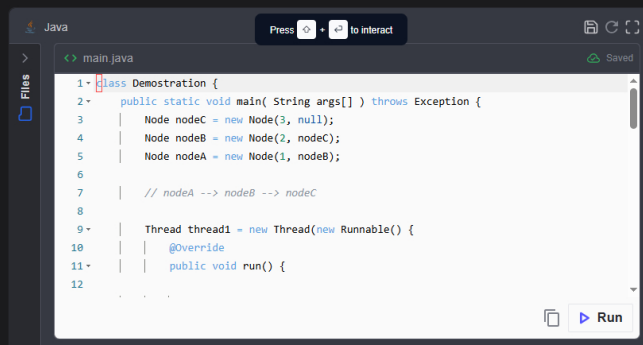


```
1- class Demonstration {
2-
3-     public static void main( String args[] ) throws Exception {
4-         SimpleNode nodeC = new SimpleNode(3, null);
5-         SimpleNode nodeB = new SimpleNode(2, nodeC);
6-         SimpleNode nodeA = new SimpleNode(1, nodeB);
7-
8-         // NodeA --> NodeB --> NodeC
9-
10-        Thread thread1 = new Thread(new Runnable() {
11-            @Override
12-            public void run() {
```

The reason we end up in an inconsistent state is because we have no way of tracking the state of a given node if it has already been deleted or not. In our example, NodeC was deleted but `thread1` didn't realize that NodeC was not part of the list anymore. One possible solution is to store a boolean alongside each node that indicates the deletion status of the node. If set to true, the node is considered deleted but may not have been removed from the list yet.

This functionality is provided by the `AtomicMarkableReference` class. We can use `AtomicMarkableReference` to store the `next` for each node. Whenever, updating the `next` field of a node using the `compareAndSet()` method, we'll pass in two expected values instead of one. The first expected value will be the reference of the object and the second a boolean value. When we update the `next` field of a node, we provide the new reference and `false` for the boolean value, implying that we only update the reference, if the node itself hasn't been marked for deletion. In case, the delete operation finds that the predecessor of the node being deleted is already marked for deletion then the current delete operation fails physically, in the sense that the reference in the `next` field of the predecessor isn't updated but the node being deleted has its mark set to true to indicate a logical delete.

If we re-run the same example with our modified algorithm, `thread1` will succeed to update the reference after resumption from sleep, however, `thread2` will not be able to delete nodeC. The list would still comprise of NodeA and NodeC but not be in an inconsistent state because the mark for NodeC will be true, indicating that the node has been logically deleted. The operation to delete NodeC can be either re-tried immediately or lazily deleted at a later time when another operation traverses the list and removes all the nodes marked true.



```
1- class Demonstration {
2-     public static void main( String args[] ) throws Exception {
3-         Node nodeC = new Node(3, null);
4-         Node nodeB = new Node(2, nodeC);
5-         Node nodeA = new Node(1, nodeB);
6-
7-         // nodeA --> nodeB --> nodeC
8-
9-        Thread thread1 = new Thread(new Runnable() {
10-            @Override
11-            public void run() {
12-                . . .
```

This example captures the gist of the utility of the `AtomicMarkableReference` class and additional state about a data structure can be maintained.

