56% completed

Search Course

- CompletionService Interface
- ThreadLocal
- ThreadLocalRandom
- CountDownLatch
- CyclicBarrier
- Concurrent Collections
- ConcurrentHashMap
- ConcurrentModificationException

Practice Mock Interview →

AtomicIntegerArray

Comprehensive guide to working with AtomicIntegerArray

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

Overview

The class `AtomicIntegerArray` represents an array of type `int` (integers) that can be updated atomically. An instance of the `AtomicIntegerArray` can be constructed either by passing an existing array of `int` or by specifying the desired size to the constructors of `AtomicIntegerArray`. The `int` data type is a 32-bit signed two's complement integer, which has a minimum value of -2^{31} and a maximum value of 2^{31}-1. In Java SE 8 and later, the `Integer` class can be used to represent an unsigned 32-bit integer, which has a minimum value of 0 and a maximum value of 2^{32}-1.

One notable difference between an ordinary array of `int`-s and an `AtomicIntegerArray` is that the latter provides volatile access semantics for its array elements, which isn't supported for ordinary arrays.

Example

The code widget below demonstrates constructing an instance of `AtomicIntegerArray` and the various operations possible on it. Comments have been added to explain the various value of 0 and a maximum value of 2^{32}-1.

One notable difference between an ordinary array of `int`-s and an `AtomicIntegerArray` is that the latter provides volatile access semantics for its array elements, which isn't supported for ordinary arrays.

Example

The code widget below demonstrates constructing an instance of `AtomicIntegerArray` and the various operations possible on it. Comments have been added to explain the various operations.

Java

main.java

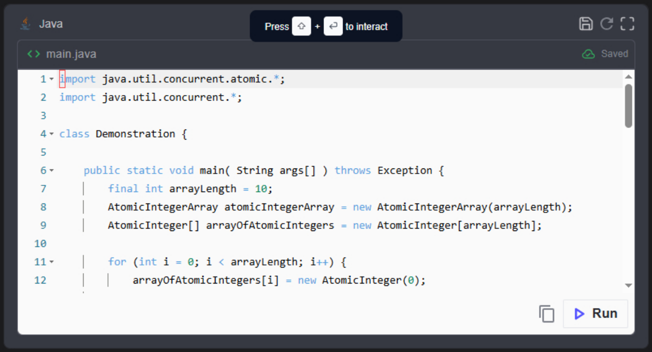
```java
import java.util.concurrent.atomic.*;
import java.util.concurrent.*;

class Demonstration {
    public static void main( String args[] ) {

        int[] inputArray = new int[]{11, 17, 19, 23, 31};

        // use an array of ints to initialize an instance of AtomicIntegerArray
        AtomicIntegerArray atomicIntegerArray = new AtomicIntegerArray(inputArray);

        // index we'll manipulate
```

Run

Difference between `AtomicIntegerArray` and and an array of `AtomicInteger`-s

We can also create an array of `AtomicInteger`s instead of creating an `AtomicIntegerArray` but there are subtle differences between the two. These are: Creating an array of `AtomicInteger`s requires instantiating an instance of `AtomicInteger` for every index of the array, whereas in case of `AtomicIntegerArray`, we only instantiate an object of the `AtomicIntegerArray` class. In other words, using an array of `AtomicInteger`s requires an object per element whereas `AtomicIntegerArray` requires an object of the class and an array object… Both classes provide for updating the integer values present at the indexes atomically, however, in case of array of `AtomicInteger`s updating the object present at the index itself isn't thread-safe. A thread can potentially overwrite the `AtomicInteger` object at say index 0 with a new object. Such a situation isn't possible with `AtomicIntegerArray` since the class only allows `int` values to be passed-in through the public methods for updating the integer values the array holds. `AtomicInteger []` is an array of thread-safe integers, whereas `AtomicIntegerArray` is a thread-safe array of integers.

Both classes are thread-safe when multiple threads update integer values at various indexes.The following widget demonstrates ten threads randomly pick an index using `ThreadLocalRandom` and then add one to the integer value at the chosen index of an instance of `AtomicIntegerArray` and an array of `AtomicInteger`s at the same index. At the end we should observe the same counts for all the indexes for both classes since the operations should be thread-safe.



```java
import java.util.concurrent.atomic.*;
import java.util.concurrent.*;

class Demonstration {

    public static void main( String args[] ) throws Exception {
        final int arrayLength = 10;
        AtomicIntegerArray atomicIntegerArray = new AtomicIntegerArray(arrayLength);
        AtomicInteger[] arrayOfAtomicIntegers = new AtomicInteger[arrayLength];

        for (int i = 0; i < arrayLength; i++) {
            arrayOfAtomicIntegers[i] = new AtomicInteger(0);
```

Note that we have done the initialization of the array of `AtomicInteger`s in the main thread. The array initialization isn't thread-safe and in general the reference of the `AtomicInteger` object can be updated in a thread unsafe manner, something the `AtomicIntegerArray` doesn't suffer from.

Back lesson
AtomicInteger

Mark As Completed

Next
AtomicIntegerFieldUpdater