AtomicLongArray

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

Overview

The `AtomicLongArray` is equivalent of the `AtomicIntegerArray` for the long type. You'll observe several similarities between the two classes.

The class `AtomicLongArray` represents an array of type `long` that can be updated atomically. We can use long when the range of values we want to represent falls outside the range of values for an integer. An instance of the `AtomicLongArray` can be constructed either by passing an existing array of `long` or by specifying the desired size to the constructors of `AtomicLongArray`. The `long` data type is a 64-bit signed two's complement integer, which has a minimum value of -2^{63} and a maximum value of $2^{63}-1$. In Java SE 8 and later, `Long` class can be used to represent an unsigned 64-bit long, which has a minimum value of 0 and a maximum value of $2^{64}-1$.

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

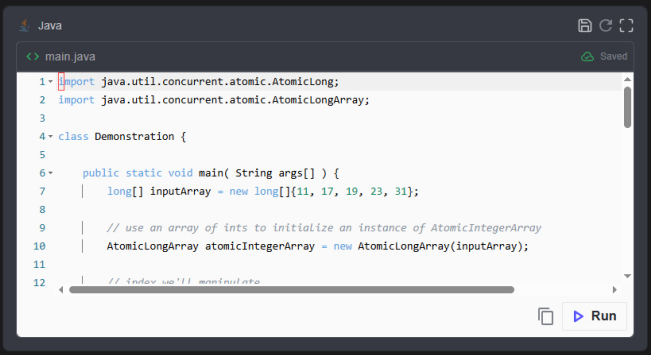
Example

`AtomicLongArray`. The `long` data type is a 64-bit signed two's complement integer, which has a minimum value of -2^{63} and a maximum value of $2^{63}-1$. In Java SE 8 and later, `Long` class can be used to represent an unsigned 64-bit long, which has a minimum value of 0 and a maximum value of $2^{64}-1$.

One notable difference between an ordinary array of `long`-s and an `AtomicLongArray` 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 `AtomicLongArray` and the various operations possible on it. Comments have been added to explain the various operations.



Difference between `AtomicLongArray` and and an array of `AtomicLong`-s

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

Both classes are thread-safe when multiple threads update long values at various indexes.The following widget demonstrates ten threads randomly pick an index using `ThreadLocalRandom` and then add one to the long value at the chosen index of an instance of

`AtomicLongArray` and an array of `AtomicLong`-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.

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

← Back lesson

AtomicLong

Mark As Completed Next →

AtomicLongFieldUpdater