

58% 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

CompletionService Interface

ThreadLocal

ThreadLocalRandom

CountDownLatch

CyclicBarrier

Concurrent Collections

ConcurrentHashMap

ConcurrentModificationException

Practice Mock Interview

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

AtomicLong

Comprehensive guide to working with AtomicLong.

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

Overview

AtomicLong is the equivalent class for long type in the java.util.concurrent.atomic package as is AtomicInteger for int type. The AtomicLong class represents a long value that can be updated atomically, i.e. the read-modify-write operation can be executed atomically upon an instance of AtomicLong. The class extends Number.

Like the AtomicInteger class the AtomicLong makes for great counters, sequence numbers etc as it uses the compare-and-swap (CAS) instruction under the hood. The CAS instruction doesn't penalize competing threads for access to shared data/state with suspension as locks do. In general, suspension and resumption of threads involves significant overhead and under low to moderate contention non-blocking algorithms that use CAS outperform lock-based alternatives.

Performance

To demonstrate the performance of AtomicLong we can construct a crude test, where a counter is incremented a million times by ten threads to reach a total of ten million. We'll time the run for an AtomicLong counter and an ordinary long counter. The widget below outputs the results:

Performance

To demonstrate the performance of AtomicLong we can construct a crude test, where a counter is incremented a million times by ten threads to reach a total of ten million. We'll time the run for an AtomicLong counter and an ordinary long counter. The widget below outputs the results:

```
1 import java.util.concurrent.ExecutorService;
2 import java.util.concurrent.Executors;
3 import java.util.concurrent.TimeUnit;
4 import java.util.concurrent.atomic.*;
5
6 class Demonstration {
7
8     static long simpleCounter;
9     static AtomicLong atomicCounter;
10
11     public static void main( String args[] ) throws Exception {
12         // test/finish
```

### Difference with long

Remember that AtomicLong isn't a drop-in replacement for long. Specifically, just like the AtomicInteger class, the AtomicLong doesn't override equals() or hashCode() and each instance is distinct. The widget below demonstrates that the same long value for two different AtomicLong instances doesn't hash to the same bucket.

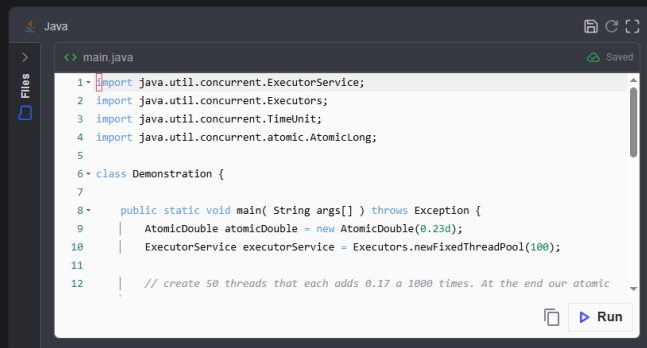
```
1 import java.util.concurrent.atomic.*;
2 import java.util.HashMap;
3
4 class Demonstration {
5
6     public static void main( String args[] ) {
7         // create map
8         HashMap<AtomicLong, String> mapAtomic = new HashMap<>();
9         HashMap<Long, String> mapLong = new HashMap<>();
10
11         // create two instances with the same long value 5
12         AtomicLong fiveAtomic = new AtomicLong(5);
```

### Using AtomicLong to simulate atomic double

## Using AtomicLong to simulate atomic double

Previously, we demonstrated how the `AtomicInteger` class can be used to create atomic byte and atomic float types. Similarly, an equivalent atomic class for `double` primitive type doesn't exist, however, we can simulate one using `AtomicLong` as we did for the primitive type `byte`. The `double` data type is a double-precision 64-bit IEEE 754 floating point. Its range of values is beyond the scope of this discussion. This data type should never be used for precise values, such as currency. For that, you will need to use the `java.math.BigDecimal` class instead.

To create our custom `AtomicDouble` type, we extend from the class `Number` and override several of its functions. Under the hood, we save the floating point's bit representation in an instance of `AtomicLong`. The class listing appears below with comments for explanation:

A screenshot of an IDE window titled 'Java' showing a file named 'main.java'. The code defines a class 'AtomicDouble' that extends 'Number' and implements 'Comparable<AtomicDouble>'. It uses 'AtomicLong' internally to store the double value. The code includes imports for 'java.util.concurrent.\*' and 'java.math.BigDecimal'. A 'main' method is shown with a comment indicating it creates 50 threads that add 0.17 a 1000 times. The code is as follows:

```
1 import java.util.concurrent.ExecutorService;
2 import java.util.concurrent.Executors;
3 import java.util.concurrent.TimeUnit;
4 import java.util.concurrent.atomic.AtomicLong;
5
6 class AtomicDouble {
7
8     public static void main( String args[] ) throws Exception {
9         AtomicDouble atomicDouble = new AtomicDouble(0.23d);
10        ExecutorService executorService = Executors.newFixedThreadPool(100);
11
12        // create 50 threads that each adds 0.17 a 1000 times. At the end our atomic
```

The `AtomicDouble` class in this lesson is shown for instructional purposes only. If you need to use this primitive type atomically, you can find well-written and well-tested libraries on the internet. For instance, Google's Guava library offers an `AtomicDouble` type.