Atomic Assignments

Not all assignments operations in Java are atomic. Learn what types can be atomically assigned according to the Java specification.

Overview

Multithreading revolves around the ability to execute actions atomically, that is without interference from other threads. We use various language provided constructs to ensure that critical sections of code are executed atomically. However, this also begs the questions, what operations are performed atomically by the language. For instance we already know that incrementing an integer counter as follows is never thread-safe:

```
counter++;
```

The above expression breaks down into several lower-level instructions that may or may not be performed atomically and therefore we can't guarantee that the execution of this expression is thread-safe. However, what about simple operations such as an assignment? Consider the following:

```
void someMethod(int passedInt) {
   counter = passedInt; // counter is an instance variable
}
```

If several threads were to invoke `someMethod` and each passing a different value for the integer can we assume the assignment will be thread-safe? To make the example concrete, say we have two threads one invoking `someMethod(5)` and the other invoking ``someMethod(7). There are three outcomes for the counter`

`counter` is assigned 5 `counter` is assigned 7 `counter` has an arbitrary value because some bits are written by the first thread and some by the second thread.

In our example the variable `counter` is of primitive type `int` which is 32 bits in Java. The astute reader would question what guarantees an assignment operation as atomic?

Java specification

The question if assignment is atomic or not is a valid one and the confusion is addressed by the Java specification itself, which states:

- Assignments and reads for primitive data types except for `double` and `long` are always atomic. If two threads are invoking `someMethod()` and passing in 5 and 7 for the integer `counter` variable then the variable will hold either 5 or 7 and not any other value. There will be no partial writes of bits from either thread.
- The reads and writes to `double` and `long` primitive types aren't atomic. The JVM specification allows implementations to break-up the write of the 64 bit `double` or `long` primitive type into two writes, one for each 32 bit half. This can result in a situation where two threads are simultaneously writing a `double` or `long` value and a third thread observes the first 32 bits from the write by the first thread and the next 32 bits from the write by the second thread. As a result the third thread reads a value that has neither been written by either of the two threads or is a garbage value. In order to make reads and writes to `double` or `long` primitive types atomic, we must mark them as `volatile`. The specification guarantees writes and reads to volatile `double` and `long` primitive types as atomic. Note that some JVM implementations may make the writes and reads to `double` and `long` types as atomic but this isn't universally guaranteed across all the JVM implementations. We'll cover `volatile` in a later lesson, for now, just remember that primitive `double` and `long` types must be marked as `volatile` for atomic assignments.
- All reference assignments are atomic. By reference we mean a variable holding a memory location address, where an object has been allocated by the JVM. For instance, consider the snippet `Thread currentThread = Thread.currentThread();` The variable `currentThread` holds the address for the current thread's object. If several threads execute the above snippet, the variable `currentThread` will hold one valid memory location address and not a garbage value. It can't happen that the variable `currentThread` holds some bytes from the assignment operation of one thread and other bytes from the assignment operation of an another thread. Whatever reference the variable holds will reflect an assignment from one of the threads. Reference reads and writes are always atomic whether the reference (memory address) itself consists of 32 or 64 bits.

Bear in mind that atomic assignments promised by the Java platform don't imply thread-safety! Our example method `someMethod()` is not thread-safe. Consider more complex situations such as reading and then assigning a variable in a method (e.g. initializing a

singleton), such scenarios need to be made thread-safe using locks or `synchronized`.

← Back lesson
Moore's Law

Completed Next →
Thready Safety & Synchronized