

Java Tip 0002 — Primitive Types and Atomicity

© 2009 Rex Jaeschke. All rights reserved.

Issue: I'm moving some of my applications to a 64-bit system. I've heard that concurrent access of values of certain primitive types might be problematic.

Response: The fundamental issue here is "Which primitive type accesses are atomic?" This is not just a 64-bit issue; it can also be a concern on 32-bit word-addressable systems (such as some popular RISC machines).

Consider the following class:

```
public class Atomic
{
    private byte b1;           // synchronization is guaranteed
    private byte b2;           //      "      "
    private short s;           //      "      "
    private float f;           //      "      "
    private short[] as;        //      "      "
    private long l1;           // synchronization is implementation specific
    private double d1;         //      "      "
    private volatile long l2;   // synchronization is guaranteed
    private volatile double d2; //      "      "
    // ...
}
```

When an object of this type is packed into memory on a 32-bit system, `b1`, `b2`, and `s` might be packed into the same 32-bit word. And when an object of this type is packed into memory on a 64-bit system, `b1`, `b2`, `s`, and `f` might be packed into the same 64-bit word. Some systems cannot directly read and write sequences of one or more bytes within a word; instead, they read the word in which the byte sequence is packed, extract the sequence, update it, and rewrite the whole word. If multiple threads are accessing separate byte sequences in the same word without synchronization, they can get in each other's way. A similar situation can occur when the byte sequence representing a value spans multiple machine words. (See the JLS, 3e, §17.6, "Word Tearing", p. 578, for more details.)

Java requires that fields and array elements of type `char`, `byte`, `short`, `int`, and `float` be accessed atomically, even if synchronization has to be used by the implementation behind the scenes. However, this is *not* guaranteed for fields and array elements of type `long` and `double`, which, in the absence of the `volatile` modifier, might or might not have synchronized access, depending on the implementation. According to the JLS, 3e, §17.7, "17.7 Non-atomic Treatment of `double` and `long` Word Tearing", p. 579, "For the purposes of the Java programming language memory model, a single write to a non-volatile `long` or `double` value is treated as two separate writes: one to each 32-bit half."

Note that atomicity does *not* come into play when dealing with local variables; when a method is being executed by multiple threads at the same time each thread gets its own local variable set. In fact, a local variable *cannot* have a `volatile` modifier.