Synchronization in Java, Part 3: Atomic operations and deadlocks

Cay Horstmann | March 2, 2022

This third article in a series on thread synchronization describes volatile fields, final variables, atomic operations, deadlocks, the deprecated stop and suspend methods, and on-demand initializations.

[This is the final article of a three-part series on thread synchronization adapted from Core Java, Volume I: Fundamentals, 12th Edition, by Cay S. Horstmann. You can read the first article here and read the second article here. —Ed.]

The first article in this series on thread synchronization covered the fundamentals of race conditions, lock objects, condition objects, and the `await`, `signal`, and `signalAll` methods. The second article addressed intrinsic locks, the `synchronized` keyword, synchronized blocks, ad hoc locks, and the concept of monitors.

This series conclusion describes volatile fields, final variables, atomic operations, deadlocks, the deprecated `stop` and `suspend` methods, and on-demand initializations.

Volatile fields
Sometimes, it seems excessive to pay the cost of synchronization just to read or write an instance field or two. After all, what can go wrong? Unfortunately, with modern processors and compilers, there is plenty of room for error.

- Computers with multiple processors can temporarily hold memory values in registers or local memory caches. As a consequence, threads running in different processors may see different values for the same memory location!
- Compilers can reorder instructions for maximum throughput. Compilers won’t choose an ordering that changes the meaning of the code, but they assume that memory values are changed only when there are explicit instructions in the code. However, a memory value can be changed by another thread!

If you use locks to protect code that can be accessed by multiple threads, you won’t have these problems. Compilers are required to respect locks by flushing local caches as necessary and not inappropriately reordering instructions. The details are explained in the Java Memory Model and Thread Specification developed by JSR 133. Much of the specification is highly complex and technical, but the document also contains a number of clearly explained examples. (“JSR 133 (Java Memory Model) FAQ” is a more accessible overview article by Jeremy Manson and Brian Goetz.)

Oracle Java Language Architect Brian Goetz coined the following synchronization motto: “If you write a variable which may next be read by another thread, or you read a variable which may have last been written by another thread, you must use synchronization.”

The volatile keyword offers a lock-free mechanism for synchronizing access to an instance field. If you declare a field as volatile, the compiler and the virtual machine take into account that the field may be concurrently updated by another thread.

For example, suppose an object has a Boolean flag done that is set by one thread and queried by another thread. As already discussed, you can use a lock, as follows:

```java
private boolean done;
public synchronized boolean isDone() { return done; }
public synchronized void setDone() { done = true; }
```

Perhaps it is not a good idea to use the intrinsic object lock. The isDone and setDone methods can block if another thread has locked the object. If that is a concern, you can use a separate lock just for this variable. But this is getting to be a lot of trouble.

In this case, it is reasonable to declare the field as volatile.

```java
private volatile boolean done;
public boolean isDone() { return done; }
```
The compiler will insert the appropriate code to ensure that a change to the `done` variable in one thread is visible from any other thread that reads the variable.

**Warning:** Variables that are `volatile` do not provide any atomicity. For example, the method

```java
public void flipDone() { done = !done; } // not atomic
```

is not guaranteed to flip the value of the field. There is no guarantee that the reading, flipping, and writing will be uninterrupted.

**Final variables**

As you saw in the preceding section, you cannot safely read a field from multiple threads unless you use locks or the `volatile` modifier.

There is one other situation in which it is safe to access a shared field: when it is declared `final`. Consider

```java
final var accounts = new HashMap<String, Double>();
```

Other threads get to see the `accounts` variable after the constructor has finished.

Without using `final`, you would have no guarantee that other threads would see the updated value of `accounts`—they might all see null, not the constructed `HashMap`.

Of course, the operations on the map are not thread-safe. If multiple threads mutate and read the map, you still need synchronization.

**Atomics**
You can declare shared variables as `volatile` provided you perform no operations other than assignment.

There are several classes in the `java.util.concurrent.atomic` package that use efficient machine-level instructions to guarantee the atomicity of other operations without using locks. For example, the `AtomicInteger` class has methods `incrementAndGet` and `decrementAndGet` that atomically increment or decrement an integer. For example, you can safely generate a sequence of numbers like this:

```java
public static AtomicLong nextNumber = new AtomicLong(); // in some thread.
long id = nextNumber.incrementAndGet();
```

The `incrementAndGet` method atomically increments the `AtomicLong` and returns the postincrement value. That is, the operations of getting the value, adding 1, setting it, and producing the new value cannot be interrupted. It is guaranteed that the correct value is computed and returned, even if multiple threads access the same instance concurrently.

There are methods for atomically setting, adding, and subtracting values, but if you want to make a more complex update, you have to use the `compareAndSet` method. For example, suppose you want to keep track of the largest value that is observed by different threads. The following won’t work:

```java
public static AtomicLong largest = new AtomicLong();
// in some thread.
largest.set(Math.max(largest.get(), observed)); // ERROR--race condition!
```

This update is not atomic. Instead, provide a lambda expression for updating the variable, and the update is done for you. In the example, you can call

```java
largest.updateAndGet(x -> Math.max(x, observed));
```

or
The `accumulateAndGet` method takes a binary operator that is used to combine the atomic value and the supplied argument. There are also methods `getAndUpdate` and `getAndAccumulate` that return the old value.

These methods are also provided for the classes `AtomicInteger`, `AtomicIntegerArray`, `AtomicIntegerFieldUpdater`, `AtomicLongArray`, `AtomicLongFieldUpdater`, `AtomicReference`, `AtomicReferenceArray`, and `AtomicReferenceFieldUpdater`.

When you have a very large number of threads accessing the same atomic values, performance suffers because the optimistic updates require too many retries. The `LongAdder` and `LongAccumulator` classes solve this problem. A `LongAdder` is composed of multiple variables whose collective sum is the current value. Multiple threads can update different summands, and new summands are automatically provided when the number of threads increases. This is efficient in the common situation where the value of the sum is not needed until after all work has been done. The performance improvement can be substantial.

If you anticipate high contention, you should simply use a `LongAdder` instead of an `AtomicLong`. The method names are slightly different. Call `increment` to increment a counter or `add` to add a quantity, and call `sum` to retrieve the total.

```java
var adder = new LongAdder();
or (. . .)
    pool.submit(() ->
        {  // ...}
            while (. . .)
            {  // ...}
                if (. . .) adder.increment();
            }
    });
...  // ...
long total = adder.sum(); long total = adder.sum();
```

Of course, the `increment` method does not return the old value. Doing that would undo the efficiency gain of splitting the sum into multiple summands.
The LongAccumulator generalizes this idea to an arbitrary accumulation operation. In the constructor, you provide the operation as well as its neutral element. To incorporate new values, call `accumulate`. Call `get` to obtain the current value. The following has the same effect as a `LongAdder`:

```java
var adder = new LongAccumulator(Long::sum, 0);
// in some thread...
adder.accumulate(value);
```

Internally, the accumulator has variables \(a_1, a_2, \ldots a_n\). Each variable is initialized with the neutral element (0 in this example).

When `accumulate` is called with value \(v\), one of them is atomically updated as \(a_i = a_i \text{ op } v\), where \(\text{op}\) is the accumulation operation written in infix form. In this example, a call to `accumulate` computes \(a_i = a_i + v\) for some \(i\).

The result of `get` is \(a_1 \text{ op } a_2 \text{ op } \ldots \text{ op } a_n\). In this example, that is the sum of the accumulators \(a_1 + a_2 + \ldots + a_n\).

If you choose a different operation, you can compute maximum or minimum. In general, the operation must be associative and commutative. That means that the final result must be independent of the order in which the intermediate values were combined.

There are also `DoubleAdder` and `DoubleAccumulator` that work in the same way, except with double values.

**Deadlocks**

Locks and conditions cannot solve all problems that might arise in multithreading. Consider the following situation in a banking application:

- Account 1: $200.
- Account 2: $300.
- Thread 1: Transfer $300 from Account 1 to Account 2.
- Thread 2: Transfer $400 from Account 2 to Account 1.

As Figure 1 indicates, Thread 1 and Thread 2 are clearly blocked. Neither can proceed because the balances in Account 1 and Account 2 are insufficient.
**Figure 1.** A deadlock situation

It is possible that all threads get blocked because each is waiting for more money. Such a situation is called a *deadlock*, and unless there are handlers to detect the situation, the program will hang.

If the program hangs, press `Ctrl+\`. You will get a thread dump that lists all threads. Each thread has a stack trace, telling you where it is currently blocked. Alternatively, run `jconsole` and consult the Threads panel (see **Figure 2**).
Figure 2. The Threads panel in jconsole

Consider the following sample scenario of a developing deadlock:

- Account 1: $1,990.
- All other accounts: $990 each.
- Thread 1: Transfer $995 from Account 1 to Account 2.
- All other threads: Transfer $995 from their account to another account.

Clearly, all threads but Thread 1 are blocked, because there isn’t enough money in their accounts. Thread 1 proceeds, and afterward, you have the following situation:

- Account 1: $995
- Account 2: $1,985
- All other accounts: $990 each

Then, Thread 1 calls signal. The signal method picks a thread at random to unblock. Suppose it picks Thread 3. That thread is awakened, finds that there isn’t enough money in its account, and calls await again. But Thread 1 is still running. A new random transaction is generated, say,

- Thread 1: Transfer $997 from Account 1 to Account 2.

Now, Thread 1 also calls await, and all threads are blocked. The system has deadlocked.

The culprit here is the call to signal, which unblocks only one thread, and it may not pick the thread that is essential to make progress. (In this scenario, Thread 2 must proceed to take money out of Account 2.)

Unfortunately, there is nothing in the Java programming language to avoid or break these deadlocks. You must design your program’s logic to ensure that a deadlock situation cannot occur.

Why the stop and suspend methods are deprecated

The initial release of Java defined a stop method that simply terminates a thread and a suspend method that blocks a thread until another thread calls resume. The stop and suspend methods have something in common: Both methods attempt to control the behavior of a given thread without the thread’s cooperation.

The stop, suspend, and resume methods have been deprecated. The stop method is inherently...
The stop, suspend, and resume methods have been deprecated. The stop method is inherently unsafe, and experience has shown that the suspend method frequently leads to deadlocks. Why are these methods problematic? What can you do to avoid problems?

**The stop method.** This method terminates all pending methods, including the run method. When a thread is stopped, it immediately gives up the locks on all objects that it has locked. This can leave objects in an inconsistent state.

For example, suppose a TransferRunnable is stopped in the middle of moving money from one bank account to another, after the withdrawal and before the deposit. Now the bank object is damaged. Since the lock has been relinquished, the damage is observable from the other threads that have not been stopped.

When one thread wants to stop another thread, it has no way of knowing when the stop method is safe and when it leads to damaged objects. Therefore, the stop method has been deprecated. You should interrupt a thread when you want it to stop. The interrupted thread can then stop when it knows it is safe to do so.

By the way, some authors claim that the stop method has been deprecated because it can cause objects to be permanently locked by a stopped thread. However, that claim is not valid. A stopped thread exits all synchronized methods it has called—technically, by throwing a ThreadDeath exception. As a consequence, the thread relinquishes the intrinsic object locks that it holds.

**The suspend method.** Unlike stop, suspend won’t damage objects. However, if you suspend a thread that owns a lock, the lock is unavailable until the thread is resumed. If the thread that calls the suspend method tries to acquire the same lock, the program deadlocks: The suspended thread waits to be resumed, and the suspending thread waits for the lock.

This situation occurs frequently in graphical user interfaces. Suppose you have a graphical simulation of a bank. A button labeled Pause suspends the transfer threads, and a button labeled Resume resumes them.

```java
pauseButton.addActionListener(event ->
{
    for (int i = 0; i < threads.length; i++)
        threads[i].suspend(); // don't do this
});

resumeButton.addActionListener(event ->
{
    for (int i = 0; i < threads.length; i++)
        threads[i].resume();
});
```

Suppose a paintComponent method paints a chart of each account, calling a getBalances method to
get an array of balances. Both the button actions and the repainting occur in the same thread, the event dispatch thread. Consider the following scenario:

- One of the transfer threads acquires the lock of the bank object.
- The user clicks the Pause button.
- All transfer threads are suspended, but one of them still holds the lock on the bank object.
- For some reason, the account chart needs to be repainted.
- The paintComponent method calls the getBalances method.
- That method tries to acquire the lock of the bank object.

Now the program is frozen. The event dispatch thread can’t proceed because the lock is owned by one of the suspended threads. Thus, the user can’t click the Resume button, and the threads won’t ever resume.

If you want to safely suspend a thread, introduce a suspendRequested variable and test it in a safe place of your run method—in a place where your thread doesn’t lock objects that other threads need. When your thread finds that the suspendRequested variable has been set, it should keep waiting until it becomes available again.

**On-demand initialization**

Sometimes you have a data structure that you want to initialize only when it is first needed, and you want to ensure that initialization happens exactly once. Instead of designing your own mechanism, make use of the fact that the JVM executes a static initializer exactly once when the class is first used. The JVM ensures this with a lock, so you don’t have to program your own.

```java
public class OnDemandData
{
    // private constructor to ensure only one object is constructed
    private OnDemandData()
    {
        ...
    }

    public static OnDemandData getInstance()
    {
        return Holder.INSTANCE;
    }

    // only initialized on first use, i.e. in the first call to getInstance
    private static Holder
    {
        // VM guarantees that this happens at most once
        static final OnDemandData INSTANCE = new OnDemandData();
    }
}
```
By the way, to use this idiom, you must ensure that the constructor doesn’t throw any exceptions. The JVM will not make a second attempt to initialize the Holder class.

**Conclusion**

The first article in this three-part series on thread synchronization covered the fundamentals of race conditions, lock objects, condition objects, and the await, signal, and signalAll methods.

The second article addressed intrinsic locks, the synchronized keyword, synchronized blocks, ad hoc locks, and the concept of monitors.

This series conclusion described volatile fields, final variables, atomic operations, deadlocks, the deprecated stop and suspend methods, and on-demand initializations.

**Dig deeper**

- *The Java Tutorials: Atomic access*
- *The Java Tutorials: Deadlock*
- Deprecated threads methods

---

Cay Horstmann

Cay S. Horstmann (@cayhorstmann) has written many books on C++, Java, and object-oriented development, and is a frequent speaker at computer industry conferences. For four years, Horstmann was VP and CTO of an internet startup that went from three people in a tiny office to a public company. He is now a computer science professor at San Jose State University.
<table>
<thead>
<tr>
<th>Resources for</th>
<th>Why Oracle</th>
<th>Learn</th>
<th>What's New</th>
<th>Contact Us</th>
</tr>
</thead>
<tbody>
<tr>
<td>About</td>
<td>Analyst Reports</td>
<td>What is Customer Service?</td>
<td>Try Oracle Cloud Free Tier</td>
<td>US Sales</td>
</tr>
<tr>
<td></td>
<td>Best CRM</td>
<td>What is ERP?</td>
<td>Oracle Sustainability</td>
<td>1.800.633.0738</td>
</tr>
<tr>
<td></td>
<td>Cloud Economics</td>
<td>What is Marketing Automation?</td>
<td>Oracle COVID-19 Response</td>
<td>How can we help?</td>
</tr>
<tr>
<td></td>
<td>Corporate Responsibility</td>
<td>What is Procurement?</td>
<td>Oracle and SailGP</td>
<td>Subscribe to Oracle Content</td>
</tr>
<tr>
<td></td>
<td>Diversity and Inclusion</td>
<td>What is Talent Management?</td>
<td>Oracle and Premier League</td>
<td>Try Oracle Cloud Free Tier</td>
</tr>
<tr>
<td></td>
<td>Security Practices</td>
<td>What is VM?</td>
<td>Oracle and Red Bull Racing Honda</td>
<td>Events</td>
</tr>
</tbody>
</table>

© 2022 Oracle | Site Map | Privacy / Do Not Sell My Info | Cookie Preferences | Ad Choices | Careers