What Cases Require Synchronized Method Access in Java?

Asked 16 years, 1 month ago Modified 14 years ago Viewed 17k times

20

In what cases is it necessary to synchronize access to instance members? I understand that access to static members of a class always needs to be synchronized- because they are shared across all object instances of the class.

My question is when would I be incorrect if I do not synchronize instance members?

for example if my class is

```
public class MyClass {
    private int instanceVar = 0;

    public setInstanceVar()
    {
        instanceVar++;
    }

    public getInstanceVar()
    {
        return instanceVar;
    }
}
```

in what cases (of usage of the class `MyClass`) would I *need* to have methods: `public synchronized setInstanceVar()` and `public synchronized getInstanceVar()` ?

Thanks in advance for your answers.

java concurrency synchronization methods non-static

Share
Improve this question
Follow

edited Nov 21, 2008 at 18:08

Daniel Spiewak
55.1k ● 14 ● 111 ● 120

asked Nov 21, 2008 at 17:59

user39732
205 ● 1 ● 4 ● 7

6 Answers

Sorted by: Highest score (default)

The `synchronized` modifier is really a *bad* idea and should be avoided at all costs. I think it is commendable that Sun tried to make locking a little easier to acheive, but

40

`synchronized` just causes more trouble than it is worth.

The issue is that a `synchronized` method is actually just syntax sugar for getting the lock on `this` and holding it for the duration of the method. Thus, `public synchronized void setInstanceVar()` would be equivalent to something like this:

```
public void setInstanceVar() {
    synchronized(this) {
        instanceVar++;
    }
}
```

This is bad for two reasons:

- All `synchronized` methods within the same class use the exact same lock, which reduces throughput
- Anyone can get access to the lock, including members of other classes.

There is nothing to prevent me from doing something like this in another class:

```
MyClass c = new MyClass();
synchronized(c) {
    ...
}
```

Within that `synchronized` block, I am holding the lock which is required by all `synchronized` methods within `MyClass`. This further reduces throughput and *dramatically* increases the chances of a deadlock.

A better approach is to have a dedicated `lock` object and to use the `synchronized(...)` block directly:

```
public class MyClass {
    private int instanceVar;
    private final Object lock = new Object();     // must be final!

    public void setInstanceVar() {
        synchronized(lock) {
            instanceVar++;
        }
    }
}
```

Alternatively, you can use the `java.util.concurrent.Lock` interface and the `java.util.concurrent.locks.ReentrantLock` implementation to achieve basically the same result (in fact, it is the same on Java 6).

Share Improve this answer Follow

answered Nov 21, 2008 at 18:18

Daniel Spiewak

55.1k ● 14 ● 111 ● 120

1 @Daniel I liked your explanation a lot but shouldn't it 'must' be declared 'static' as well especially to have Thread-safe public classes ? As you also want to guaranty that all threads for attaining lock will use same (shared) object that doesn't get modified. `private static final Object lock = new Object(); // must be static final` – sactiw Feb 18, 2011 at 15:45

3 There is no need for `lock` to be `static` since the data it is synchronizing (`instanceVar`) is itself not-static. Static locks are almost always a sign of very, very serious trouble and you should avoid them whenever possible. – Daniel Spiewak Feb 23, 2011 at 22:38

@Daniel hmmm... I guess I missed the point that here 'MyClass' doesn't extends Thread so it won't be having it's own threads. So here the purpose of making the class thread safe is that when some external threads are sharing same object (instance) of 'MyClass' then they should access [update] the instanceVar in a synchronized manner. In other words the lock should have been declared 'static' if MyClass was declared extending Thread. Right? – sactiw Feb 26, 2011 at 9:08

Extending `Thread` is really just an implementation detail. It's bad practice, btw, but still doable. If you do that, then you will have *at least* two threads running against the same instance. Those threads would use the exact same lock object. However, for threads running against different instances of `MyClass`, there's no point in locking on the same object because they're not accessing the same data! So, we could make it static, but there's absolutely no benefit and a lot of pain if we do. – Daniel Spiewak Feb 26, 2011 at 15:36

1 Nice explanation but can you explain why "must be final!"? – Marian Paździoch Nov 4, 2015 at 8:36

20

It depends on whether you want your class to be thread-safe. Most classes shouldn't be thread-safe (for simplicity) in which case you don't need synchronization. If you need it to be thread-safe, you should synchronize access *or* make the variable volatile. (It avoids other threads getting "stale" data.)

Share Improve this answer Follow

answered Nov 21, 2008 at 18:01

Jon Skeet

1.5m ● 889 ● 9.3k ● 9.3k

3

If you want to make this class thread safe I would declare `instanceVar` as `volatile` to make sure you get always the most updated value from memory and also I would make the `setInstanceVar()` `synchronized` because in the JVM an increment is not an atomic operation.

```
private volatile int instanceVar =0;

public synchronized setInstanceVar() { instanceVar++;

}
```

Share Improve this answer Follow

answered Nov 21, 2008 at 18:08

bruno conde

48.2k ● 15 ● 102 ● 118

3 No need to make the field volatile, but setInstanceVar() does need to be synchronized. – Daniel Spiewak Nov 21, 2008 at 18:13

1 Or you can use a java.util.concurrent.atomic.AtomicInteger. – InverseFalcon Nov 21, 2008 at 18:27

2 Daniel: If the getInstanceVar is not synchronized then the volatile makes sure that the value returned is fresh. But that's probably going to be lost on most programmers. – Tom Hawtin - tackline Nov 25, 2008 at 17:48

InverseFalcon: AtomicInteger may cause problems if there are large numbers of these objects. AtomicIntegerFieldUpdater may be better, but tends to give significantly worse performance when you have to use it. – Tom Hawtin - tackline Nov 25, 2008 at 17:49

@Tom, wouldn't you want your get method to be synchronized as well? – James McMahon Jun 24, 2009 at 13:17

1

. Roughly, the answer is "it depends". Synchronizing your setter and getter here would only have the intended purpose of guaranteeing that multiple threads couldn't read variables between each others increment operations:

```
synchronized increment()
{
      i++
}

synchronized get()
{
  return i;
 }
```

but that wouldn't really even work here, because to insure that your caller thread got the same value it incremented, you'd have to guarantee that you're atomically incrementing and then retrieving, which you're not doing here - i.e you'd have to do something like

```
 synchronized int {
   increment
```

```
    return get()
  }
```

Basically, synchronization is usefull for defining which operations need to be guaranteed to run threadsafe (inotherwords, you can't create a situation where a separate thread undermines your operation and makes your class behave illogically, or undermines what you expect the state of the data to be). It's actually a bigger topic than can be addressed here.

This book Java Concurrency in Practice is excellent, and certainly much more reliable than me.

Share Improve this answer Follow

answered Nov 21, 2008 at 18:15

Steve B.

57.2k ● 12 ● 97 ● 134

1

To simply put it, you use synchronized when you have mutliple threads accessing the same method of the same instance which will change the state of the object/or application.

It is meant as a simple way to prevent race conditions between threads, and really you should only use it when you are planning on having concurrent threads accessing the same instance, such as a global object.

Now when you are reading the state of an instance of a object with concurrent threads, you may want to look into the the java.util.concurrent.locks.ReentrantReadWriteLock -- which in theory allows many threads to read at a time, but only one thread is allowed to write. So in the getter and setting method example that everyone seems to be giving, you could do the following:

```
public class MyClass{
    private ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();
    private int myValue = 0;

    public void setValue(){
        rwl.writeLock().lock();
        myValue++;
       rwl.writeLock().unlock();
    }

    public int getValue(){
       rwl.readLock.lock();
       int result = myValue;
       rwl.readLock.unlock();
       return result;
    }
}
```

Share Improve this answer Follow

answered Nov 21, 2008 at 18:46

Charlie Lavers

The problem with ReadWriteLock is it imposes some extra overhead. This overhead is only worthwhile if you have *long* read operations, otherwise you're better off with just a single ReentrantLock. – Daniel Spiewak Nov 21, 2008 at 18:56

-3

In Java, operations on ints are atomic so no, in this case you don't need to synchronize if all you're doing is 1 write and 1 read at a time.

If these were longs or doubles, you do need to synchronize because it's possible for part of the long/double to be updated, then have another thread read, then finally the other part of the long/double updated.

Share Improve this answer Follow

answered Nov 21, 2008 at 18:04

Tim Frey

9,931 ● 9 ● 45 ● 61

1 Though be careful of the ++ and -- operators, they're not atomic. To be on the safe side, try using classes from java.util.concurrent.atomic. – InverseFalcon Nov 21, 2008 at 18:09

Operations on integers aren't atomic in Java. – Piyush Mattoo Feb 21, 2013 at 3:09

You can modify an int from different threads, get localized caching in the threads, have the ++ operation read and write out of sync. Just because longs and doubles read and write as two op operations (load upper and load lower to get all 64 bits) doesn't mean that integers suddenly become atomic because they suffer from 1 fewer non-threadsafe issues. – Tatarize Jun 2, 2013 at 6:05