What is the reason why “synchronized” is not allowed in Java 8 interface methods?

Ask Question

In Java 8, I can easily write:

```
interface Interface1 {
    default void method1() {
        synchronized (this) {
            // Something
        }
    }

    static void method2() {
        synchronized (Interface1.class) {
            // Something
        }
    }
}
```

I will get the full synchronisation semantics that I can use also in classes. I cannot, however, use the `synchronized` modifier on method declarations:

```
interface Interface2 {
    default synchronized void method1() {
        //  ^^^^^^^^^^^^ Modifier 'synchronized' not allowed here
    }

    static synchronized void method2() {
        // ^^^^^^^^^^^^ Modifier 'synchronized' not allowed here
    }
}
```

Now, one can argue that the two interfaces behave the same way except that `Interface2` establishes a *contract* on `method1()` and on `method2()` , which is a bit stronger than what `Interface1` does. Of course, we might also argue that `default` implementations should not make any assumptions about concrete implementation state, or that such a keyword simply wouldn't pull its weight.

Question:

What is the reason why the JSR-335 expert group decided not to support `synchronized` on interface methods?

java java-8 synchronized default-method jsr335

edited Feb 9 at 21:17
Tim Castelijns
27.9k 12 77 100

asked May 4 '14 at 7:06
Lukas Eder
121k 68 398 837

Synchronized is an implementation behavior and it changes final byte code result made by compiler so it can be used beside a code. It has no sense in method declaration. It should be confusing what has compiler produce if synchronized is on abstraction layer. – Martin Strejc May 4 '14 at 7:18

@MartinStrejc: That *might* be an explanation for omitting `default synchronized` , yet not necessarily for `static synchronized` , although I would accept that the latter might've been omitted for consistency reasons. – Lukas Eder May 4 '14 at 7:31

1 I'm not sure if this question adds any value as the `synchronized` modifier may be overriden in subclasses, hence it would only matter if there was something as final default methods. (Your other question) – skiwi May 4 '14 at 12:40

@skiwi: The overriding argument is not sufficient. Subclasses may override methods that are declared `synchronized` in super classes, effectively removing synchronization. I wouldn't be surprised that not supporting `synchronized` and not supporting `final` is related, though, maybe because of multiple inheritance (e.g. inheriting `void x()` *and* `synchronized void x()` , etc.). But that's speculation. I'm curious about an authoritative reason, if there is one. – Lukas Eder May 4 '14 at 13:32

2 >>"Subclasses may override methods that are declared synchronized in super classes, effectively removing synchronization"... only if they don't call `super` which requires a full re-implementation and possible access to private members. Btw, there is a reason those methods are called "defenders" - they are present to allow easier adding new methods. – bestsss May 10 '14 at 6:10

1 Answer

This was a deliberate decision, rather than an omission (as has been suggested elsewhere.) While at first it might seem obvious that one would want to support the `synchronized` modifier on default methods, it turns out that doing so would be dangerous, and so was prohibited.

Synchronized methods are a shorthand for a method which behaves as if the entire body is enclosed in a `synchronized` block whose lock object is the receiver. It might seem sensible to extend this semantics to default methods as well; after all, they are instance methods with a receiver too. (Note that `synchronized` methods are entirely a syntactic optimization; they're not needed, they're just more compact than the corresponding `synchronized` block. There's a reasonable argument to be made that this was a premature syntactic optimization in the first place, and that synchronized methods cause more problems than they solve, but that ship sailed a long time ago.)

So, why are they dangerous? Synchronization is about locking. Locking is about coordinating shared access to mutable state. Each object should have a synchronization policy that determines which locks guard which state variables. (See Java Concurrency in Practice, section 2.4.)

Many objects use as their synchronization policy the *Java Monitor Pattern* (JCiP 4.1), in which an object's state is guarded by its intrinsic lock. There is nothing magic or special about this pattern, but it is convenient, and the use of the `synchronized` keyword on methods implicitly assumes this pattern.

It is the class that owns the state that gets to determine that object's synchronization policy. But interfaces do not own the state of the objects into which they are mixed in. So using a synchronized method in an interface assumes a particular synchronization policy, but one which you have no reasonable basis for assuming, so it might well be the case that the use of synchronization provides no additional thread safety whatsoever (you might be synchronizing on the wrong lock). This would give you the false sense of confidence that you have done something about thread safety, and no error message tells you that you're assuming the wrong synchronization policy.

It is already hard enough to consistently maintain a synchronization policy for a single source file; it is even harder to ensure that a subclass correctly adhere to the synchronization policy defined by its superclass. Trying to do so between such loosely coupled classes (an interface and the possibly many classes which implement it) would be nearly impossible and highly error-prone.

Given all those arguments against, what would be the argument for? It seems they're mostly about making interfaces behave more like traits. While this is an understandable desire, the design center for default methods is interface evolution, not "Traits--". Where the two could be consistently achieved, we strove to do so, but where one is in conflict with the other, we had to choose in favor of the primary design goal.

edited Feb 26 '15 at 20:09

answered May 5 '14 at 0:50

Brian Goetz
51k 12 90 111

23 Note also that in JDK 1.1, the `synchronized` method modifier appeared in the javadoc output, misleading people into thinking that it was part of the specification. This was fixed in JDK 1.2. Even if it appears on a public method, the `synchronized` modifier is part of the implementation, not the contract. (Similar reasoning and treatment occurred for the `native` modifier.) – Stuart Marks May 5 '14 at 4:33

13 A common mistake in early Java programs was to sprinkle enough `synchronized` and thread safe components around and you had an almost thread safe program. The problem was this usually worked ok but it broken in surprising and brittle ways. I agree that understanding how your locking works is a key to robust applications. – Peter Lawrey May 5 '14 at 10:27

8 @BrianGoetz Very good reason. But why is `synchronized(this) {...}` allowed in a `default` method? (As shown in Lukas's question.) Doesn't that allow the default method to own the state of the implementation class too? Don't we want to prevent that too? Will we need a FindBugs rule to find the cases for which uninformed developers do that? – Geoffrey De Smet May 5 '14 at 10:28

16 @Geoffrey: No, there's no reason to restrict this (though it should always be used with care.) The sync block requires the author to explicitly select a lock object; this allows them to participate in the synchronization policy of some other object, if they know what that policy is. The dangerous part is assuming that synchronizing on 'this' (which is what sync methods do) is actually meaningful; this needs to be a more explicit decision. That said, I expect sync blocks in interface methods to be pretty rare. – Brian Goetz May 5 '14 at 14:08

6 @GeoffreyDeSmet: For the same reason you can do e.g. `synchronized(vector)` . It you want to be safe, you should never use a public object (such as `this` itself) for locking. – Yogu May 5 '14 at 21:26