Why is “final” not allowed in Java 8 interface methods?

Ask Question

One of the most useful features of Java 8 are the new `default` methods on interfaces. There are essentially two reasons (there may be others) why they have been introduced:

- Providing actual default implementations. Example: `Iterator.remove()`
- Allowing for JDK API evolution. Example: `Iterable.forEach()`

From an API designer's perspective, I would have liked to be able to use other modifiers on interface methods, e.g. `final` . This would be useful when adding convenience methods, preventing "accidental" overrides in implementing classes:

```
interface Sender {

    // Convenience method to send an empty message
    default final void send() {
        send(null);
    }

    // Implementations should only implement this method
    void send(String message);
}
```

The above is already common practice if `Sender` were a class:

```
abstract class Sender {

    // Convenience method to send an empty message
    final void send() {
        send(null);
    }

    // Implementations should only implement this method
    abstract void send(String message);
}
```

Now, `default` and `final` are obviously contradicting keywords, but the default keyword itself would not have been strictly required, so I'm assuming that this contradiction is deliberate, to reflect the subtle differences between *"class methods with body"* (just methods) and *"interface methods with body"* (default methods), i.e. differences which I have not yet understood.

At some point of time, support for modifiers like `static` and `final` on interface methods was not yet fully explored, citing Brian Goetz:

> The other part is how far we're going to go to support class-building tools in interfaces, such as final methods, private methods, protected methods, static methods, etc. The answer is: we don't know yet

Since that time in late 2011, obviously, support for `static` methods in interfaces was added. Clearly, this added a lot of value to the JDK libraries themselves, such as with `Comparator.comparing()` .

Question:

What is the reason `final` (and also `static final`) never made it to Java 8 interfaces?

java language-design java-8 default-method jsr335

edited Jun 11 '15 at 13:25
Robert Harvey ♦
143k 31 254 390

asked May 4 '14 at 6:25
Lukas Eder
121k 68 398 837

8 I'm sorry to be a wet blanket, but the only way the question expressed in the title is going to get answered within the terms of SO is via a quotation from Brian Goetz or the JSR Expert group. I understand that BG has requested a public discussion, but that is exactly what contravenes the terms of SO, as it is 'primarily opinion-based'. It seems to me that the responsibilities are being ducked here. It's the Expert Group's job, and the wider Java Community Process's, to stimulate discussion and come up with rationales. Not SO's. So I am voting to close as 'primarily opinion-based'. – EJP May 4 '14 at 8:27

2 As we all know, `final` prevents a method from being overridden, and seeing how you MUST override methods inherited from interfaces, I dont see why it would make sense to make it final. Unless it was to signify that the method is final AFTER overriding it once.. In that case, maybe there qas difficulties? If im not understanding this right, please let me kmow. Seems interesting – Vince Emigh May 4 '14 at 8:38

19 @EJP: *"If I can do that, so can you, and answer your own question, which therefore wouldn't have needed asking."* That would apply to pretty much all questions on this forum, right? I could always Google some topic for 5 hours myself and then learn something that everyone else has to learn the equally hard way. Or we wait for another couple of minutes for someone to give an even better answer that everyone in the future (including the 12 upvotes and 8 stars so far) can profit from, because SO is so well referenced on Google. So, yes. This question *can* perfectly fit into SO's Q&A form. – Lukas Eder May 4 '14 at 9:47

4 @VinceEmigh "*...seeing how you MUST override methods inherited from interfaces...*" That isn't true in Java 8. Java 8 allows you to implement methods in interfaces, which means you don't have to implement them in implementing classes. Here, `final` would have a use in preventing implementing classes from overriding the default implementation of an interface method. – awksp May 4 '14 at 9:53

24 @EJP you never know: Brian Goetz may reply! – assylias May 5 '14 at 8:39

|

4 Answers

This question is, to some degree, related to What is the reason why “synchronized” is not allowed in Java 8 interface methods?

The key thing to understand about default methods is that the primary design goal is *interface evolution*, not "turn interfaces into (mediocre) traits". While there's some overlap between the two, and we tried to be accomodating to the latter where it didn't get in the way of the former, these questions are best understood when viewed in this light. (Note too that class methods *are* going to be different from interface methods, no matter what the intent, by virtue of the fact that interface methods can be multiply inherited.)

The basic idea of a default method is: it is an interface method with a default implementation, and a derived class can provide a more specific implementation. And because the design center was interface evolution, it was a critical design goal that default methods be able to be added to interfaces *after the fact* in a source-compatible and binary-compatible manner.

The too-simple answer to "why not final default methods" is that then the body would then not simply be the default implementation, it would be the only implementation. While that's a little too simple an answer, it gives us a clue that the question is already heading in a questionable direction.

Another reason why final interface methods are questionable is that they create impossible problems for implementors. For example, supposing you have:

```
interface A {
    default void foo() { ... }
}

interface B {
}

class C implements A, B {
}
```

Here, everything is good; C inherits foo() from A. Now supposing B is changed to have a foo method, with a default:

```
interface B {
    default void foo() { ... }
}
```

Now, when we go to recompile C, the compiler will tell us that it doesn't know what behavior to inherit for `foo()` , so C has to override it (and could choose to delegate to `A.super.foo()` if it wanted to retain the same behavior.) But what if B had made its default final, and A is not under the control of the author of C? Now C is irretrievably broken; it can't compile without overriding `foo()` , but it can't override `foo()` if it was final in `B` .

This is just one example, but the point is that final methods are really a tool that makes more sense in the world of single-inheritance classes (generally which couple state to behavior), than to interfaces which merely contribute behavior and can be multiply inherited. It's too hard to reason about "what other interfaces might be mixed into the eventual implementor", and allowing an interface method to be final would likely cause these problems (and they would blow up not on the person who wrote the interface, but on the poor user who tries to implement it.)

Another reason to disallow them is that they wouldn't mean what you think they mean. A default implementation is only considered if the class (or its superclasses) don't provide a declaration (concrete or abstract) of the method. If a default method were final, but a superclass already implemented the method, the default would be ignored, which is probably not what the default author was expecting when declaring it final. (This inheritance behavior is a reflection of the design center for default methods -- interface evolution. It should be possible to add a default method (or a default implementation to an existing interface method) to existing interfaces that already have implementations, without changing the behavior of existing classes that implement the interface, guaranteeing that classes that already worked before default methods were added will work the same way in the presence of default methods.)

edited May 23 '17 at 11:47
Community ♦
1 1

answered May 5 '14 at 16:10
Brian Goetz
51k 12 90 111

79 Fantastic to see you answering questions about the new language features! It's very helpful getting clarification on the intent and details of the design when figuring out *how* we should be using the new features. Are other people who were involved with the design contributing to SO - or are you just doing this yourself? I'll be following your answers under the java-8 tag - I'd like to know if there are other folks doing the same so I can follow them too. – Shorn May 7 '14 at 1:46

9 @Shorn Stuart Marks has been active in the java-8 tag. Jeremy Manson has posted in the past. I also remember seeing messages from Joshua Bloch but can't find them right now. – assylias May 7 '14 at 9:05

The multiple interface scenario is interesting. What happens when you have two conflicting default methods? – Lorenzo Boccaccia May 29 '14 at 13:22

Congratulations for coming up with default interface methods as a much more elegant way of accomplishing that which C# does with its ill-conceived and rather uglily implemented extension methods. As for this question, the answer about the impossibility of resolving a name conflict kind of settles the issue, but the rest of the reasons provided were unconvincing philology. (If I want an interface method to be final, then you ought to presume that I must have my own pretty damn good reasons for wanting to disallow anyone from providing any implementation different from mine.) – Mike Nakis Jun 20 '14 at 0:15

8 @Trying Abstract classes are still the only way to introduce state, or implement the core Object methods. Default methods are for *pure behavior*; abstract classes are for behavior coupled with state. – Brian Goetz Dec 27 '15 at 20:05

|

In the lambda mailing list there are plenty of discussions about it. One of those that seems to contain a lot of discussion about all that stuff is the following: On Varied interface method visibility (was Final defenders).

In this discussion, Talden, the author of the original question asks something very similar to your question:

> The decision to make all interface members public was indeed an unfortunate decision. That any use of interface in internal design exposes implementation private details is a big one.
>
> It's a tough one to fix without adding some obscure or compatibility breaking nuances to the language. A compatibility break of that magnitude and potential subtlety would seen unconscionable so a solution has to exist that doesn't break existing code.
>
> Could reintroducing the 'package' keyword as an access-specifier be viable. It's absence of a specifier in an interface would imply public-access and the absence of a specifier in a class implies package-access. Which specifiers make sense in an interface is unclear - especially if, to minimise the knowledge burden on developers, we have to ensure that access-specifiers mean the same thing in both class and interface if they're present.
>
> In the absence of default methods I'd have speculated that the specifier of a member in an interface has to be at least as visible as the interface itself (so the interface can actually be implemented in all visible contexts) - with default methods that's not so certain.
>
> Has there been any clear communication as to whether this is even a possible in-scope discussion? If not, should it be held elsewhere.

Eventually Brian Goetz's answer was:

> Yes, this is already being explored.
>
> However, let me set some realistic expectations -- language / VM features have a long lead time, even trivial-seeming ones like this. The time for proposing new language feature ideas for Java SE 8 has pretty much passed.

So, most likely it was never implemented because it was never part of the scope. It was never proposed in time to be considered.

In another heated discussion about final defender methods on the subject, Brian said again:

> And you have gotten exactly what you wished for. That's exactly what this feature adds -- multiple inheritance of behavior. Of course we understand that people will use them as traits. And we've worked hard to ensure that the the model of inheritance they offer is simple and clean enough that people can get good results doing so in a broad variety of situations. We have, at the same time, chosen not to push them beyond the boundary of what works simply and cleanly, and that leads to "aw, you didn't go far enough" reactions in some case. But really, most of this thread seems to be grumbling that the glass is merely 98% full. I'll take that 98% and get on with it!

So this reinforces my theory that it simply was not part of the scope or part of their design. What they did was to provide enough functionality to deal with the issues of API evolution.

edited Sep 8 '14 at 15:40

answered May 4 '14 at 15:46
Edwin Dalorzo
49.7k 18 105 154

4 I see that I should have included the old name, "defender methods", in my googling odyssey this morning. +1 for digging this up. – Marco13 May 4 '14 at 17:00

1 Nice digging up of historic facts. Your conclusions align well with the official answer – Lukas Eder May 5 '14 at 16:34

It will be hard to find and identify "THE" answer, for the resons mentioned in the comments from @EJP : There are roughly 2 (+/- 2) people in the world who can give the definite answer *at all*. And in doubt, the answer might just be something like "Supporting final default methods did not seem to be worth the effort of restructuring the internal call resolution mechanisms". This is speculation, of course, but it is at least backed by subtle evidences, like this Statement (by one of the two persons) in the OpenJDK mailing list:

> "I suppose if "final default" methods were allowed, they might need rewriting from internal invokespecial to user-visible invokeinterface."

and trivial facts like that a method is simply not *considered* to be a (really) final method when it is a `default` method, as currently implemented in the Method::is_final_method method in the OpenJDK.

Further really "authorative" information is indeed hard to find, even with excessive websearches and by reading commit logs. I thought that it might be related to potential ambiguities during the resolution of interface method calls with the `invokeinterface` instruction and and class method calls, corresponding to the `invokevirtual` instruction: For the `invokevirtual` instruction, there may be a simple **vtable** lookup, because the method must either be inherited from a superclass, or implemented by the class directly. In contrast to that, an `invokeinterface` call must examine the respective call site to find out *which* interface this call actually refers to (this is explained in more detail in the InterfaceCalls page of the HotSpot Wiki). However, `final` methods do either not get inserted into the **vtable** at all, or replace existing entries in the **vtable** (see klassVtable.cpp. Line 333), and similarly, default methods are replacing existing entries in the **vtable** (see klassVtable.cpp, Line 202). So the *actual* reason (and thus, the answer) must be hidden deeper inside the (rather complex) method call resolution mechanisms, but maybe these references will nevertheless be considered as being helpful, be it only for others that manage to derive the actual answer from that.

answered May 4 '14 at 12:12

Marco13
39.2k 8 52 92

Thanks for the interesting insight. The piece by John Rose is an interesting trace. I still don't agree with @EJP, though. As a counter-example, check out my answer to a very interesting, very similar-style question by Peter Lawrey. It *is* possible to dig out historic facts, and I'm always glad to find them here on Stack Overflow (where else?). Of course, your answer is still speculative, and I'm not 100% convinced that JVM implementation details would be the final reason (pun intended) for the JLS to be written down in one or another way... – Lukas Eder May 4 '14 at 13:57

@LukasEder Sure, these kinds of questions **are** interesting and IMHO fit into the Q&A pattern. I think there are two crucial points that caused the disputation here: The first is that you asked for "the reason". This may in many cases just not *officially* be documented. E.g. there is no "reason" mentioned in the JLS why there are no unsigned `int` s, yet see stackoverflow.com/questions/430346 ... ... – Marco13 May 4 '14 at 14:15

... ... The second is that you asked *only* for "authorative citations", which reduces the number of people who dare to write an answer from "a few dozen" to ... "approximalely zero". Apart from that, I'm not sure about how the development of the JVM and the writing of the JLS are interweaved, i.e. in how far the development influences what's written into the JLS, but... I'll avoid any speculation here ;-) – Marco13 May 4 '14 at 14:15

1 I still rest my case. See who's answered my *other question* :-) It will now forever be clear with an authoritative answer, here on Stack Overflow *why* it was decided not to support `synchronized` on `default` methods. – Lukas Eder May 5 '14 at 7:51

3 @LukasEder I see, the same here. Who could have expected that? The reasons are rather convincing, particularly for this `final` question, and it's somewhat humbling that nobody else seemed to think of similar examples (or, maybe, some thought about these examples, but did not feel authorative enough to answer). So now the (sorry, I have to do this:) *final* word is spoken. – Marco13 May 5 '14 at 19:57

I wouldn't think it is neccessary to specify `final` on a convienience interface method, I can agree though that it *may* be helpful, but seemingly the costs have outweight the benefits.

What you are supposed to do, either way, is to write proper javadoc for the default method, showing exactly what the method is and is not allowed to do. In that way the classes implementing the interface "are not allowed" to change the implementation, though there are no guarantees.

Anyone could write a `Collection` that adheres to the interface and then does things in the methods that are absolutely counter intuitive, there is no way to shield yourself from that, other than writing extensive unit tests.

answered May 4 '14 at 12:36

skiwi
34.1k 20 92 157

2 Javadoc contracts are a valid workaround to the *concrete* example I've listed in my question, but the question is really not about the convenience interface method use-case. The question is about an authoritative reason why

`final` has been decided not to be allowed on Java 8 `interface` methods. Insufficient cost / benefit ratio is a good candidate, but so far, that's speculation. – Lukas Eder May 4 '14 at 13:44