Speaker 1: ... Is good, and that's recording.

Speaker 2: Great.

Speaker 1: All right. There are two ways we could go about this. I know that you've done quite a lot of work with the Java JDK.

Speaker 2: Yes.

Speaker 1: Are you pointing me towards the JSR that you got?

Speaker 2: Mm-hmm (affirmative).

Speaker 1: It was about the about the Optional as well, and you worked with Stuart Marks, I think, or not worked, but he had interviewed ...

Speaker 2: No. I was a spec lead for JSR 308, which was type annotations, and I was on the expert committee for JSR 305, which was trying to define annotations and their semantics. For Optional, I'm not involved. I wrote this critical blog posting that says that Optional is a terrible idea, and Stuart goes around giving a talk saying ... Okay. There are two ways to interpret that talk. The way I interpret it is, he says, "well, it's a terrible idea. Only use it on return values," but the way he views the talk is, "Optional is a great idea, here's how you should use it, by the way, only on return values."

Speaker 1: All right. Yeah. Nonetheless, you work quite a lot with the Java JDK.

Speaker 2: Yes.

Speaker 1: What we've done up until now is interview API developers.

Speaker 2: Okay.

Speaker 1: There would be JUnit developer, Spring, Mockito, developers from industries, such companies as [blinded]. We interviewed them, we got certain results, we asked them about the deprecation mechanism, how they perceive it because I'm sure you know JEPT-77, which is an advanced deprecation, led by Stuart Marks whose doctored deprecator.

Speaker 2: Right.

Speaker 1: It's all about announcing the current deprecation mechanism with two new features, and they have took motivations behind it.

Speaker 2: To actually start using it in anger?

Speaker 1: Yeah. It's all about the job, but it was all based on observations made in the Java AC-API.

Speaker 2: Right.

Speaker 1: They're basing everything on what they start in Java. What we decided to do was look at this from the third-party API developer perspective which would be different, or it could be different. Even necessarily be different.

Speaker 2: Right.

Speaker 1: We also wanted to understand whether or not these enhancements really met all the needs perceived both by third-party API developers and API consumers. For that, we set up this whole interview process and the server process and all of that. It was a more quilted study. However, those were API developers, so we spoke to them. Now, we could treat you more as a Java expert, so that would be a different way to do it, right? Because I don't think you really fall under the category of API developer.

Speaker 2: That's true. I've built libraries and put them on GitHub, and people use them, but that's not the main thing that I do.

Speaker 1: No. Would you prefer to talk as more of a Java expert, or would you prefer to talk as an API developer? I'm giving you the choice.

Speaker 2: It sounds like you're asking ... Java expert, I'm not sure how expert I am. API consumer or API developer?

Speaker 1: Not consumer even. I'm really talking about it from a Java language designer perspective.

Speaker 2: Oh, I see. Sure, I'm happy to do that. It sounds fine.

Speaker 1: Okay. If that's a perspective, I think we would like that perspective, right?

Speaker 2: Great.

Speaker 1: Okay.

Speaker 2: I have one small perspective on it, but I'll share that.

Speaker 1: Absolutely. No. We'd also like to present our results to you then because if it was the other way, I wouldn't have shown you the results. I wouldn't advise you, I'd just ask you straight up certain questions.

Speaker 2: Okay.

Speaker 1: All right. Let's do this with the perspective, you as part of the Java language design group. What is your current thought about the deprecation mechanism? What do you think about it?

Speaker 2: The current one or the proposed one?

Speaker 1: Yeah. The current one.

Speaker 2: When I'm building my own software that's mostly going to be used by me, I use @deprecated. Usually I use @deprecated for libraries that I'm supplying to the world where the Java language is finally caught up, like I might have built something that does joining of arrays with separations, and then the Strings class now does that. It didn't before. Now, I tell people, "Okay, you've been using my library for a while, now you should be using Java or you should be using the real JD, something that's in the real JDK." I'm a little sloppy in that, I'll leave it deprecated for a while. At some point when I'm feeling like it, I'll remove it. Usually, that's in January or July. That's when I think about six-month periods. The fact that I'm sloppy that way, and the fact that my APIs are used by small numbers of people, rather than millions of people means that I found it okay. I think it's ugly that there is an @deprecated Javadoc tag and an @deprecated annotation.

Speaker 1: Yeah, but you know why that is, right?

Speaker 2: Well, they're processed by two totally different tools.

Speaker 1: They still have the exact same implications underneath, so despite not being the Java language specification, the annotation tag is actually perceived as a compartment warning until they introduce annotations into Java 1.5 which was a bit of a ridiculous thing. If you have a non-Oracle JVM and a non-San JVM, you don't actually kept those features. You actually think it's ridiculous to have a separated Javadoc annotation and a separate source code annotation?

Speaker 2: Yeah. We should have one.

Speaker 1: Which one?

Speaker 2: I don't care. I guess the Javadoc one is nice because if you write an annotation, you can't write a paragraph, or you could have an annotation that takes the String as an argument, but it's going to get really ugly to write a paragraph. You perhaps shouldn't be writing a whole paragraph, but the one in the Javadoc lets you do so. I guess I would lean toward having one that's in the code. The reason that I would prefer the annotation is that I feel like that's visible in a totally standard way to any tool that's processing the AST whereas I don't want every tool to have to parse Javadoc comments. Some tools don't even have access to the comments, where all tools have access to that. I'm always in favor of making informal specifications more formal, and of making things that are more english, or harder to parse into things that are easier to parse. I always write both. It's a code smell. Every time I do it, it's a code smell.

Speaker 1: One question I do have there is ...You know that this deprecation mechanism that they have is the same as the source code mechanism, right?

Speaker 2: Mm-hmm (affirmative). Which is the right way to do it.

Speaker 1: Is the right way to do it, I'm sure. The motivation that they used was two-fold. One, it's open to misuse, the current deprecation mechanism. You could use it for a multitude of reasons. One of the interesting cases that we saw from JUnit, for instance, was that when a feature was beta or experimental, they marked it as deprecated. Now, that's in misuse of deprecation because you're not exactly explicitly saying that this is an obsolete feature.

Speaker 2: Right. They were using deprecated as "Don't use this."

Speaker 1: Yeah, because it's beta experimental, but use it at your own will or caution, right?

Speaker 2: Yeah. Right.

Speaker 1: Do you think that's a misuse?

Speaker 2: Yes and no. I would support them in using it. I think they should write it that way because otherwise, how are they going to convey it to programmers? You always want something that's checked. Writing in the Javadoc, "Don't use this" is nothing compared to something that's actually going to be checked by the compiler. Yes, I think it's the right thing to do and that they should do it. I think it's unfortunate that the name of it is deprecated, which is a bad name. It's a misleading name. I would come down on the side of not misuse.

Speaker 1: Not a misuse?

Speaker 2: It's a non-standard use.

Speaker 1: Right.

Speaker 2: If someone asked me, "Should I do this, or not?" I would say yes.

Speaker 1: What would be an example of misuse. This is something that the Java line designers, themselves say. Yes? Hello.

Speaker 3: Hello. Hello.

Speaker 2: Hello.

Speaker 3: Sorry for being late. I was in [inaudible 00:09:03].

Speaker 2: Do you want a chair?

Speaker 3: Thank you.

Speaker 1: We were talking about misuse of the Java deprecated annotations. According to Speaker 2: here, usage of deprecating because of beta is not a misuse because that's a good way to communicate.

Speaker 2: Right.

Speaker 1: If I would spoil your argument.

Speaker 2: Maybe if something is inefficient in certain circumstances and you deprecated it to slow people down, to caution them that they maybe shouldn't use it, but there's no alternative that can be available.

Speaker 1: There's no alternative. Exactly. This does leave the deprecated annotation open to misuse, right?

Speaker 2: Sure.

Speaker 1: Would you then support a warning mechanism, something that's [inaudible 00:09:48] from the deprecated, so that you don't have a perversion of the deprecated word? As you just said, deprecated is a bad term for that. Would you then have a ...

Speaker 2: In some sense, you're asking whether deprecated should be a special sub-case.

Speaker 1: Of warning, yes.

Speaker 2: ... Of this other thing. Okay. I don't support a new feature unless I see many use cases for it. I'm also not sure whether that belongs in the language. Deprecated is part of the JDK, so there are many annotations that you could write. Well, you could write non-null and nullable and your compiler, with the right plug-in, your compiler is going to start issuing warnings about those. Do we need all of those built into the language, or are those better things that can be done via plug-ins? I guess I would lean toward trying to keep that language a little bit leaner, not adding too much complexity to it until we really have a lot of evidence that this is getting ... There are these three use cases and all three of them are really common and really important. We need to standardize the names and provide them to everyone.

Speaker 1: Even if it would address the issue misuse of the deprecation annotation? Which apparently cited as a case for people not taking deprecation seriously.

Speaker 2: Did they have concrete evidence of that?

Speaker 1: Well, they claimed it straight up. They claim it based on the Java SCA, but there's no evidence cited over there.

Speaker 2: Yeah. I don't think you can build a mechanism that can't be misused.

Speaker 1: Okay.

Speaker 2: Any library, any language feature can be misused. The fact that it can be misused doesn't mean that you shouldn't put it there. It doesn't mean you should always have training wheels on every feature. The argument for Optional on return types is yes, people could look at the nullable annotation on it. Then, they could be careful not to reference it or they could look at the documentation that says this might be null or they could run a tool, but you know what? Programmers are too lazy and they're not going to do that, so we should add all this ugly syntax and overhead, so that they can't possibly forget. I think that's a bad design. I think we should permit people to write and use tools that will prevent these bad uses. If we attempted to eliminate every type of misuse, then we're only going to open the opportunity for more types of misuse, and we're going to make it harder for people who are going to use it in a sensible way or in an imaginative way.

Speaker 1: Okay.

Speaker 2: If there's good evidence that this is being systematically misused by a lot of people, then yes, we should do this. Maybe they have that evidence. I don't know if they have it with respect to confidential code bases by their partners or evidence by programmers within Oracle. I guess when I look at the deprecation mechanism, I say, "Okay. Great. You're doing that. It doesn't really affect me because I find the current one adequate and because I'm not doing these misuses."

Speaker 3: Sorry. What is the setting here? Are you discussing with them about the ...

Speaker 1: As a Java language designer, or from the Java language.

Speaker 3: You discuss the results or ...

Speaker 1: Not yet. I'm getting that.

Speaker 3: Okay. Okay.

Speaker 2: I don't yet know what they are.

Speaker 3: Okay. Good.

Speaker 1: The other motivation that was sited by them was the improper removal patterns. What you just said is all about your own libraries APIs, but you decide every six months once in a while to remove deprecated features, right?

Speaker 2: Right.

Speaker 1: You don't exactly have like a clean up protocol, or a regular clean up protocol.

Speaker 2: Right.

Speaker 1: According to them, this led to a lot of confusion. No one really knows what the feature of a deprecated feature is going to be. For instance, Java's deprecated data is 1.1, right?

Speaker 2: Right.

Speaker 1: We're still what? 10 years, 15 years down the line.

Speaker 2: More than that.

Speaker 1: Nothing has happened to it.

Speaker 2: 1.1 was in the 90s.

Speaker 1: '96 actually. We're 21 years out.

Speaker 2: 1.0 was '96.

Speaker 1: Then, 1.1 was also '96 [crosstalk 00:14:30].

Speaker 2: Was it also?

Speaker 1: Yeah.

Speaker 2: Oh, okay. Fine.

Speaker 1: John Rose's data deprecation was in 1.1.

Speaker 2: Okay, great.

Speaker 1: Okay. It's been 21 years. They haven't removed it.

Speaker 2: I actually like that. I like the fact that there's this predictable time in which they'll be removed.

Speaker 1: In this case, there is no time.

Speaker 2: No. In the new proposal, doesn't it give away to indicate when things will be dep ...?

Speaker 1: No.

Speaker 2: Oh, it doesn't?

Speaker 1: The new proposal, all it has is, it says accent. You can tell when it was deprecated, which version, and a for removal [inaudible 00:15:06] which says whether it's going to be removed or not.

Speaker 2: It doesn't tell you when ... okay. I remember the date and a for removal. Thank you for

correcting me.

Speaker 1: It has absolutely no data on when it's going to be removed.

Speaker 2: Right. I guess people would interpret that as if this was deprecated in release a.b.c, I should expect it to be removed in a.b+1 or a.b+2.

Speaker 1: Okay. Was it me? Sorry.

Speaker 2: Sorry. Suppose that something was deprecated in version 4.6.8, if I saw that this had been deprecated since version 4.6.8, I would expect that it's going to be removed in either version 4.6.9 or more likely in version 4.7.0. I guess I would need to read the documentation to figure that out.

Speaker 3: There's no clear indication for when it's going to be removed?

Speaker 2: Correct. It would be nice to have that predictability.

Speaker 3: Yeah.

Speaker 1: Do you think it a addresses the issue? The current enhancement or proposed enhancements?

Speaker 2: Okay. I think it starts to address it, but I don't think it addresses it completely.

Speaker 1: Okay.

Speaker 2: If the documentation for the package says, "This is what we're going to do with deprecated routines," then the annotation plus that policy addresses the issue, but you can't get that just by looking at the one line of code, the one annotation. I'm not sure that you would need to make the annotation that much more complicated. You could imagine having for removal being a String or something like that.

Speaker 1: With the version or ...

Speaker 2: Or date or something.

Speaker 1: Or date or something like that?

Speaker 2: Right. I think that would give clients a lot more predictability.

Speaker 1: Yeah. That's something our server response actually agree with. All right. We asked them this question. What do you think about these proposed enhancements? They actually ranked this one as the most favorable one.

Speaker 2: You mean of all the parts of deprecated, or of all the ...?

Speaker 1: Yeah. If you were to enhance it, if you were to specify the version and whether or not it's going to be removed, for them, that was the one they wanted the most or they're most in favor of. Then we also asked them other question based of C#'s implementation or what our interviewees told us, or any other instance that we actually had or even other proposals for actually JEP 277 which failed. There's actually a list of all turned proposals which they didn't implement because of complexity, and because of time and because they'd be delaying this for a while now. For instance, they would really like an automated tool to replace.

Speaker 2: Sure. Absolutely.

Speaker 1: Right. Developers would like that, but it doesn't really come as a surprise. The third one which is have different stengths for warnings. Now, this stems from ...

Speaker 2: It's interesting. That's not one I would've ranked high. The different strengths would be depending on whether it's for removal or not for removal. Are those the only two different strengths you're talking about?

Speaker 1: No, I could go more in detail on that one. Where does this really stems from is when we asked the API consumers what really motivated them to react to deprecation? It's the reason behind deprecation that really motivated them. Certain cases are important. You have things which is there's an actual functional issue with the feature and then, it was deprecated because of that. Those are things that actually motivated them to actually react. Other things, just trivial upgraded to something don't really matter.

Speaker 2: Okay.

Speaker 1: There's a different class or different hierarchy of reasons behind deprecation. They all cannot be treated the same. We asked API developers, "Well, why do you deprecate? In which cases, do you really think that it's absolutely pivotal that the client reacts?" After them, again it was the harder cases where you have functional and non-functional issues which they said, "No. You should really react to those."

Speaker 2: Yeah. I guess that makes sense then. Yeah.

Speaker 1: Then, we asked them, "Okay. What if you could actually indicate this? What if you could actually instigate a reaction by telling the API consumer that A, this is actually very important to react to?" You have different levels of severity, a higher level.

Speaker 2: Would this severity just be error versus warning, or something even more fine grained like logger levels?

Speaker 1: I think it could be a lot of different ... It could be fine grained. It could be implemented as a logger. It could be also error versus warning. C# is there versus warning for instance.

Speaker 2: That makes sense to me.

Speaker 1: That makes sense as well. That gives the API developer more control. An API consumers actually seemed to have agree with, or Java developers in general, or you have thoughts about that?

Speaker 2: Sorry. Thoughts about?

Speaker 1: About implementing a severity level.

Speaker 2: Yeah. I would think error versus warning. Maybe just based on whether it's for removal or not.

Speaker 1: That's it?

Speaker 2: I don't know. It's hard for us to imagine all the possible reasons or all the possible motivations. If we enshrine one particular list, then people are going to start misusing it because there's going to be some reason that we didn't know about, and so they'll just use other.

Speaker 1: The reason is not the entire thing, just if it's for removal and say, "Hey, this is ..."

Speaker 2: I guess maybe for removal is different than error versus warning, but if I could only have on of the two, then I would have for removal and use that as error versus warning.

Speaker 1: Okay.

Speaker 2: I don't have an objection to having them both. I wonder if people would always feel the need to write them both and the annotation would start to get really long and ugly.

Speaker 1: You still are in favor of severity then?

Speaker 2: Yeah.

Speaker 1: Irrespective of how it's implemented?

Speaker 2: Yeah.

Speaker 1: Severity would be something definitely variable to you?

Speaker 2: Yes.

Speaker 1: That's also a good thing, but that's not part of current enhancement also.

Speaker 2: Oh, absolutely. Right.

Speaker 1: Which is a bit of a problem, I guess.

Speaker 2: Well, you can add new annotation fields in the future, I believe. I'd have to double check whether that's true, but I think you can.

Speaker 3: The Java language or ... The question is if the Java permits you to put more fields in annotation or are you talking about if the deprecation guys would be open to extend this?

Speaker 2: Oh, I don't know what their plan is.

Speaker 3: I think in the Java language, you can because you can find some default, optional values for new attributes.

Speaker 2: The question is whether you have binary compatibility, I can't remember the answer.

Speaker 1: Okay. Another one of the reasons, something that the JP also specify is that you have an email that specifies the reason behind it being removed, so obsolete or dangerous. There were five examples over there. What are your thoughts about that? Would you want that? No? Yes?

Speaker 2: I guess I'm indifferent. I don't see a strong need for that and partly because I'm worried that the eNum alone would not convey that information.

Speaker 1: Okay.

Speaker 2: I'm worried that it would not be rich enough. If something is deprecated, you're going to end up reading its Javadoc anyway and you can convey that same kind of information in the Javadoc. I guess that's how I feel about the replacement features of String. That doesn't seem that compelling to me, when I can just look at the Javadoc. The Javadoc is going to give me an example before, and hopefully give me an example before and after. It may not just be one ... Maybe I need to make two calls.

Speaker 1: Yeah.

Speaker 2: What would that String look like if I had to make two calls because this functional had been split up?

Speaker 1: Right. You prefer the Javadoc in those cases?

Speaker 2: Right. I just feels like it's richer.

Speaker 1: All right. That's good to know. Yeah. These are the ones that they actually proposed. We're also in favor of the severity, for instance and that's actually one of the key points from our findings is that the motivation is something that's very important and hard to convey. Because right now, deprecation is a one size fits all solution. Do you think the current deprecation mechanism as it is actually encourages consumers to react to deprecation?

Speaker 2: Well, people get a warning when they run the compiler. In that sense, I think it encourages them. Yes, it doesn't force them but it encourages them.

Speaker 1: With levels of severity and all that, do you think it would act as more of an incentive?

Speaker 2: Yeah. I think so, especially for the ones marked as error.

Speaker 1: Okay.

Speaker 3: What do you think about this one, issuing a run-time warning when used? Does it make sense to you?

Speaker 2: No, I don't think that's a good idea. There are too many legacy programs out there where we don't want to break them. I guess your argument is that ... This run-time warning, is that just to standard error or is that throwing an exception or not?

Speaker 3: No, it's [crosstalk 00:25:12].

Speaker 1: No, no, no. It's a standard error. Yeah.

Speaker 2: Now, you're going to clutter standard error. That can break a lot of programs. I guess your point is, "Hey, your program going to break in the future, but don't break it yet."

Speaker 1: The counter argument to that is that a lot of these Java binaries are not rebuilt over time. When they're being compiled once on an older version JBM, the developers don't really re-compile it. This is something that the Java developers themselves have actually specified as motivation.

Speaker 2: Right.

Speaker 1: The reason why they would want run-time warnings, or they would at the very least wants standard checking, I think they actually [inaudible 00:25:52] checker, by the way, for deprecation usage now.

Speaker 2: Sure. That'd be easy to do. You just look at the class files.

Speaker 1: Yeah. They actually want that, so in the old code, that's the code that's not being re-compiled, the very least developers know whether or not the feature that is being used over there is deprecated or not and whether or not it's going to be compatible with the future version of Java.

Speaker 2: Whenever you upgrade to a new version of JBM, you're running your systems on both versions for a month to look for problems and you'll discover them then.

Speaker 1: Yeah. This is their plan. I think the standard checker is already out there by the way.

Speaker 2: That's great.

Speaker 1: It's part of now JDK 9 or it's supposed to be part of JDK 9. There's a tool. That would be the counter argument is why you would want run-time?

Speaker 2: Right. Yeah. Your code is going to break you to this version or next version. The question is whether you should break it now or break it later? I'd be inclined to break it later because it's not broken yet and maybe during the couple years before the next JDK is released, maybe you'll happen to re-compile it. If you don't, it won't be any harder to re-compile it then than it is now.

Speaker 3: Okay.

Speaker 1: That's true. One final question at least from me is are you in favor of Java actually removing the deprecated features that it has? The date, for instance?

Speaker 2: As an academic, very much so. As a practitioner, not date. Date it used too much.

Speaker 1: This is despite the day time API coming up.

Speaker 2: Right. There is the day time API, and that's good. We now have 20 years of use of date. That one feels very deeply embedded. I think for future deprecations, it makes a lot of sense to actually say, "We're going to really remove this thing."

Speaker 1: You would say not to date, or not for legacy stuff in 1.1, 1.2?

Speaker 2: Maybe some things that are more obscure.

Speaker 1: Yeah. I know they already removed some of them actually. Some of the internal SAN API calls.

Speaker 2: Yeah. From a cleanliness view point of view, how it bugs me so much when there's a better day time API, that people might still use the old one.

Speaker 1: Yeah.

Speaker 2: That's easy for me to say, because I'm only maintaining a million lines of code or something like that. It's not like I'm maintaining really vast amounts of code that nobody understands or that might be difficult to rewrite. As a point to the academic, of course we should all do that. We should clean code all the time. We should make sure there are no null pointers in any of it either but from a practical point of view, I think if they did that, they would lose customers.

Speaker 1: All right. You think they would lose people? All right. They are planning on removing stuff by the way. Finally.

Speaker 2: Yeah. I think that's great.

Speaker 1: There's no specification about what's going to be removed, but they are going to remove stuff.

Speaker 2: Right.

Speaker 1: That's it.

Speaker 2: Finalization. Certain types of finalization are going to be removed.

Speaker 1: Okay. I don't think the day time API is going anywhere, or the date API. Sorry.

Speaker 2: I don't think the date API is going anywhere. Right.

Speaker 1: All right. Thank you. Any questions from you guys?

Speaker 2: This is cool stuff.

Speaker 3: Yeah. Maybe I arrived a little late. You discussed about having a more generic warning down on deprecation, so I jump in in that moment for me to understand. You said that you would not be in favor of such more generic warning. I wanted to ...

Speaker 2: I don't want to have something that's more generic until we have a bunch of use cases that say exactly how it's going to be used. It just seems like a good idea to do that's not compelling to me. Also, it's not clear how many of these things need to be in the compiler and in the JDK and how many of them can be done as external tools? If you already have a tool like find bugs or the checker framework or something like that, it can actually issue a lot of those warnings.

Speaker 3: Absolutely. Yeah. The thing is that we are seeing is that developers are using deprecation as a way of communicating because it appears to be the only way in which they can reliably communicate across any type of IDs. They use that as token, so they're sure they're something regardless of the ID. If they use another type of annotation, it really depends on whether ...

Speaker 2: They're using the tool.

Speaker 3: Exactly. From this, we thought well maybe you just want to have a generic warning which is a sort of communication between you and the client.

Speaker 2: The warning says there's some issue or other with this method that you need to be aware of.

Speaker 3: Exactly.

Speaker 2: Maybe it's really inefficient or maybe it makes a change to the file system, or maybe it is deprecated.

Speaker 3: Yeah. Deprecation would be a common specialization of this type of work.

Speaker 2: Yeah. I guess I would prefer to experiment with more specific warnings before trying to generalize them. Another type of annotation that's been proposed is you should always use the result of this routine. Never throw it away or like you have an immutable routine, if you have an immutable class, add is going to return a new instance of it. People need to use to saying mylist.add. If anyone ever does mylist.add or myimmutablelist.add, and then they don't use the return value, then they should be warned about it. That feels like something that should be incorporated to a tool that checks those uses as opposed to a general thing that's always going to pop up. If you put @warning on that, and then it popped up no matter what the context was, maybe this is in a test or maybe it's in a mock or somewhere else, or maybe that's not a good example but you might imagine some other example.

I'd rather have an analysis, and the nice thing about having an analysis as opposed to putting the language is you can put in the language, then you have to specify what are the checking rules? If someone else comes up with a more precise checker next year, or more precise out rhythm for checking it, you can't use that. If someone comes up with the whole program analysis, you can't use that.

Speaker 3: Yeah. Interesting. With the same argument, we can say that maybe you can even remove deprecation? Couldn't we move deprecation from the language to an analysis tool?

Speaker 2: Well, you've had to deprecate it before you did that because [crosstalk 00:33:00].

Speaker 3: Of course.

Speaker 2: Yes. I think here though ... Okay. If there is any kind of interesting analysis going on, then I feel like an external tool is better because the external tool can be updated at anytime and the external tool can be a whole program analysis whereas anything in the language has to be modular. This is such a trivial analysis essentially. If you're using this, it's definitely a problem.

Speaker 3: Maybe.

Speaker 2: I guess it depends. If something is deprecated, but you know you're compiling for a javafied KDM, well that's a place where knowing when it's going to be eliminated would be nice because you could have a command line argument that says, "I'm compiling this for a javafied KDM," and you'd only get warnings about things that were deprecated for removal before or after that, whichever is the right one. I don't know. Did that answer your question? Maybe partially?

Speaker 3: Partially, yes. You would not think it would be a good idea to deprecate deprecation?

Speaker 2: For me, it seems like the analysis is so simple that it's reasonable to build that algorithm into the java language and then, be stuck with it forever but if there's any worry about

being stuck with an algorithm forever, then there's something we shouldn't build into the java language until we have a plenty of experience with it as an external tool, for example.

Speaker 3: Okay. From the tool, then we bring it into the language because everybody is using it?

Speaker 2: Maybe eventually or maybe it's just a module that people customarily use.

Speaker 3: Okay. Thank you.

Speaker 2: Sure.

Speaker 1: All right. Thank you.

Speaker 2: Sure. No problem.