EPISODE 962

[INTRODUCTION]

[00:00:00] JM: Java has been popular since the 90s when it started to be used a programming language for enterprises. Today, Java is still widely deployed, but the infrastructure environment is dramatically different. Java is often deployed to containers in the cloud. If those containers can share resources, then those containers can share the same underlying Java infrastructure. Java 13 is the most recent public release of Java. The new features in Java 13 reflect the changing demands of modern application developers.

Georges Saab is an engineer with Oracle who has been working on Java for more than a decade. He joins the show to discuss how Java development patterns are changing and how the language is evolving to accommodate those changes, including discussions of garbage collection, dynamic application, class data sharing and other technical subjects.

I want to announce, we're hiring a content writer. We're also hiring an operations lead. Both of these are part-time positions working closely with myself and Erica. If you are somebody interested in writing content about software engineer or if you're interested in helping us with operations, both of these roles are fairly technical. You will learn more about software engineering. You do not need to be a software engineer. You do not need to have a degree in anything related to software engineer, but we would love to talk to people who are interested in the podcast, who like the podcast. Send me an email, jeff@softwareengineeringdaily.com.

Also, we're going to be at AWS Reinvent Las Vegas this week, and we're planning a meet-up at Reinvent. That meet-up is going to be Wednesday, December 4th, which is in contrast to the date we previously announced, and you can see the announcement in the show notes for this episode. You can see the link and you can grab an RSVP. I would love to see you there.

[SPONSOR MESSAGE]

[00:02:09] JM: Today's sponsor is Datadog, a scalable monitoring and analytics platform that unifies metrics, logs, traces and more. Use Datadog's advanced features to monitor and

manage SLO performance in real-time. Visualize all your SLOs in one place. Easily search, filter and sort SLOs, and share key information with detailed intuitive dashboards, plus Datadog automatically calculates and displays your error budget so that you can see your progress at a glance.

Sign up for a free 14-day trial and Datadog will send you a complementary t-shirt. Go to softwareengineeringdaily.com/Datadog to sign up, try out Datadog and get a wonderful t-shirt. Your SLOs will be even more on-track when you're in your Datadog t-shirt. Actually, I can't guarantee that, but I recommend going to softwareengineeringdaily.com/datadog to get the free trial and a t-shirt.

[INTERVIEW]

[00:03:09] JM: Georges Saab, welcome to Software Engineering Daily.

[00:03:11] GS: Thanks, Jeff. It's great to be here.

[00:03:12] JM: Today we're talking about Java, and specifically Java 13, and I want to start with some contextual discussion. So we have cloud infrastructure these days, and cloud infrastructures had a lot of downstream effects on software development. How has the cloud impacted programming language choices from your point of view?

[00:03:33] GS: Well, I think that one of the things that we see whenever there's kind of a new environment or a situation where people want to use a programming language, it can drive the kind of requirements that you have. So when we look at what people are doing in the cloud, in many ways, the cloud itself is not that different, but a lot of the ways that people use the cloud tend to make different things important.

As an example, the way that people want to use containers as a way of deploying things to the cloud means that we need to make sure that our runtime and language support running in a container really well. Sometimes that can be changes to language itself. More often or usually the first thing that you tend to see is that it will lead to different things in the implementation.

Just as an example, the way that Docker works with Linux in order to let a runtime know what resources are available to it is slightly different from the sort of traditional way that that was done if you were just running Linux directly on your desktop. So that's the kind of thing where the implementation of runtime needs to be aware of these changes and make sure that it's doing things in the right way.

[00:04:46] JM: You mentioned containerization there. Can you drill in to that a little bit further? How has the popularity of containerization affected the usage of programming languages? Maybe if you could address Java specifically and how containerization has affected Java.

[00:05:00] GS: Well, just as an example, the way that people tend to use Java a decade ago was they'll get a big server and then they would run an application server on that, and typically the assumption was that your application server had full control of the server you're running on and it was probably the only thing running there.

So that would tend to lead the runtime to want to be fairly aggressive with its use of the resources on the underlying harbor and operating system. Basically, making the assumption that nothing else is there. It didn't need to worry as much about releasing memory quickly to the operating system, or taking advantage of all of the course that were available on the hardware or other things like that.

Whereas running in Docker containers, basically, if you can continued to use the same Linux system calls, you would not be aware of the fact that you're not actually sharing that underlying hardware with a bunch of other containers that are out there. So you need to be a much better citizen and not take more resources than you're currently making use of. The way that that translates is to different underlying system calls in order to sort of ask what's available to you.

[00:06:13] JM: I think what you're describing is the noisy neighbor problem effectively, where in a multi-tenant system, you have all the different tenants, all the different applications that are running on a single host. They are competing for resources. There are ways that they could share resources or make use of shared infrastructure.

But if they're not really aware of each other or if the host is not really trying to make use of potential commonalities between those applications, then it's just going to be a battle for resources among these different applications. That can just create a different infrastructure environment than perhaps a host with a single application running on it, or even a VM with a single application running on it. Whereas today, you can now have like a VM with multiple containers or a host with just a ton of containers.

[00:07:14] GS: Yeah, that's certainly true, and it's also sort of a question of the scale at which that's done or at what level in the architecture that's done. So as an example with Java EE app servers a decade ago, that problem still occurred, but it was occurring within the JVM itself. Whereas with Docker containers and so on now, what people tend to be doing is having a different level of isolation and having each one of those containers probably have a more sort of single purpose and using the container technology in order to be able to kind of split things up between different services.

[00:07:51] JM: You've worked on Java for almost a decade. How was the Java ecosystem different when you started?

[00:07:58] GS: I've actually been working on Java since the mid-90s. I was on the Java team at Sun in the very early days.

[00:08:06] JM: Okay. So more than two decades.

[00:08:08] GS: Yeah, it's kind of crazy how long I am and actually many of the team members are sort of long-term folks that have been working on Java for quite a while. I think to answer your question specifically in the last decade is a real speed up and how quickly things are happening and how quickly we need to be able to put new features and capabilities in people's hands.

So we basically took a look at this 10 years ago and said, "Hey! We want to make sure that Java is poised to be able to delivery things to people more quickly." The pace at which we deliver new versions of Java 25 years ago was kind of appropriate for the way that people did software development then. But more and more were seeing that people are picking up new

technologies quickly. They're adapting the style of programming and architecture that they're using and they want to see that the tools and languages that they're using are able to keep pace with that rate of change.

So a decade ago, we decided we were going to look at a sort of longer term effort to move to having new releases that happened more frequently than the sort of two to three to four-year pace that Java had been on traditionally. It took a little while to get that setup. It wasn't something that we could change overnight, but basically we were able to successfully move to that model from the release of Java 9 a couple of years ago. Since then we've been keeping pace of a new release with Java every six months.

[00:09:44] JM: So we'll get into that increased release frequency and some more context around that and as well as the specific improvements in Java 13. I want to continue this discussion of the broader ecosystem. So if you've been tracking programming languages for the last 20 years, you've seen the rise of JavaScript. You've seen the rise of Ruby on the backend. You've seen some people using Go and perhaps Rust.

Java continues to truck-on, and there plenty of gigantic successful applications that use Java. Is there a specific type of application that Java caters to today that Java is the best for, or is it just something about the duration of the ecosystem, the maturity of the ecosystem? What is it specifically about Java that makes certain applications a good fit for Java?

[00:10:46] GS: Yeah. It's an interesting question and certainly something that I've looked at and asked people in the ecosystem what their thoughts are on it. I think that in many ways, the fact that Java exist in so many places and is just a really good and sold tool for many different kinds of applications has led to this sort of virtuous circle where the fact that it's good for so many things means it's great for people to invest in and learn about.

The fact that they have invested and learned about has meant that they've created just a wonderful set of libraries and frameworks and other things that give people a lot of choice. It means that employers have a lot of people to choose from and can find a lot of talent, and the other way around that people who want to learn some technology that's going to help them get hired and become key for their employer, it's a pretty good choice.

So I think that when I look at the kinds of applications people are running, I think many people that I speak with come me specifically through the area that they're involved in. They come and say, "Well, I'm working on server side applications on Java on traditional Java EE servers." So they'll have one set of perspectives.

But I also get people who are coming and doing embedded Java, who are running Java on small devices, or people who are working with Java card. There's just a very, very wide range of things out there. One thing that's common is most of these people are very passionate about their language choices and they all have suggestions of what we need to do to make Java better.

Generally, one of the things that they tell me is that our dedication over the years to making sure that the evolution that we're doing in Java is not just haphazard, but is very thoughtful and methodical and that while we're doing evolution, we also take care to make sure that we're bringing everyone along and making sure that the upward compatibility story is really good is something that's been important to them.

They'll say, "Hey! I have this application that I originally wrote for JDK 11, and I've ran the other day on 13 and it's running great, except a lot faster." I think it's actually really neat to see.

[SPONSOR MESSAGE]

[00:13:06] JM: If you are selling enterprise software, you want to be able to deliver that software to every kind of customer. Some enterprises are hosted on-prem. Some enterprises are on AWS. Some enterprises are on cloud providers that you've never heard of, and every cloud provider works differently.

Gravity is a product for delivering software to any kind of potential environment or data center that your customers want to run applications in. Think of Gravity as something you can use to copy-paste entire production environments across clouds and data centers. Gravity is made by Gravitational, and Gravity works with on-premise data centers and on different cloud providers.

Gravity can get software to your biggest customers without the pain of developing individualized deployment systems for every single customer. Gravity puts a bubble of consistency around your application so that you can write it once and deploy it anywhere, and Gravity is open source so you can look into the code and understand how it works.

You can also listen to the episode I recorded with Gravitational CEO, Ev Kontsevoy. Gravity is built to solve the problem of software delivery. Gravity ensures compliance and lowers the cost of development. You don't have to write your code to support every platform. It is as easy as copying and pasting your deployment each time.

Gravity is from Gravitational and it's trusted by leading companies including MuleSoft and Anaconda. Go to gravitational.com/sedaily to try Gravity Enterprise free for 60 days. Gravity uses Kubernetes under the hood and the Gravitational team knows Kubernetes well. If you go to gravitational.com/sedaily, you can sign up for a free consulting session about cross-cloud Kubernetes security. This is in addition to the 60-day free Gravity enterprise trial.

If you feel like you need to get a better understanding of Kubernetes security, check out gravitational.com/sedaily for this offer of a 30-minute free Kubernetes consultation along with a 60-day free Gravity Enterprise trial.

Gravity is a system of securely delivering your applications into any environment, and you can try it free by going to gravitational.com/sedaily. Gravity Community Edition is also available on GitHub and it's free to play with. If you are curious about how Kubernetes will change software deployments, I recommend checking out the Gravity repository, and thanks to Gravitational for being a sponsor of Software Engineering Daily.

[INTERVIEW CONTINUED]

[00:16:07] JM: Programming language historically – In many cases, they age out. So I assume there was a time when small talk was more widely used in industry. I know COBOL was widely used. FORTRAN was widely used. I mean, these applications are still around, but the programming language mindshare of those languages has decreased overtime, and that's just a nature of technology.

Java, I feel, is in the process of crossing the chasm from one age to another, and it seems like it is doing so successfully. We've talked about the past architectural pressures for what you would want out of your Java infrastructure and how that is sort of changed where you maybe want smaller application components. You want to deploy them to containers, and that just creates different desires out of maybe what you want out of a garbage collector or what you want out of a framework. Maybe you see Spring needing to change. Going to Spring boot, and Spring boot being an easier way to spin up some cheap and smaller or lighter weight boilerplate applications.

So as we get into talking about kind of the incremental changes into Java 13, before we get into the incremental changes, give me a long range view for where Java needs to go in order to remain a durable widely used programming language.

[00:17:44] GS: Yeah. So I'd say that I agree with you in terms of there being sort of changes that happened in the industry and it requiring attention from programming language developers to make sure that we're sort of doing the things that are going to help the language make the transition and help the users of language make the transition to new styles of creating applications and new architectures that you'd want to run.

For us, interestingly, move to the cloud has really taken a few things that have been themes that we've thought about for quite a number of years and really brought them on very, very quickly. For example, for many years, the focus in Java design and implementation was around thinking about how to make very long-running workloads with very, very large datasets or very large heaps run really, really well and really quickly.

For the cloud, we're seeing actually two very interesting trends, and these are in some ways somewhat a conflict or divergent, but in some ways not. So to be very concrete, many of the things that we did years ago for long-running large memory kinds of workloads are actually still around today for things like machine learning, right? When you have giant datasets that you want to do reasoning over, it's important that you'd be able to represent that memory compactly, that it'd be convenient to deal with and that you'd be able to write rules that are sort of looking at that data and doing interesting analysis of it quickly.

Many of the capabilities that Java has there are standing in a good stat in those kind of environments. Now, having said that, there are more things we can and are doing. What we tend to do is sort of look at it and understand how some of the underlying changes, for instance, in hardware, make it so that the choices that were made when designing Java 25, 30 years ago maybe need to be rethought a little bit. So we have a long-term project called Valhalla, which is basically been about looking at how the Java runtime decides to lay out data in memory and finding ways of representing it that are really relevant for today's datasets, for instance, in machine learning, so that it can become even more compact and even more performant and really take advantage of the developments and underlying hardware such as vectorization and so on.

On the other hand, we have almost the other extreme where people want to adapt a more microservice style of architecture where basically they want something that can come up quickly, do something really fast and then just kind of go away. Of course, the techniques that we've developed for optimizing code that has been running for many, many hours on a server don't really come into play there, because this thing should have come up and done something and disappeared before you even get to the point where that might happen.

So in order to support those kinds of things, we have other projects that are looking at improving start-up time, doing things like ahead of time compilation. Having different styles of garbage collection that actually work well, or in some cases get out of the way entirely for these kinds of workloads.

[00:21:10] JM: And I think one way in which Java is updating itself to remain highly relevant is this increased release frequency. So back when I was a software engineer, when I actually like wrote applications instead of just podcasting about it, I think, literally, every company that I worked at, which was a Java shop, which was 4 out of 5 companies that I worked at in the duration of my career, maybe 5 out of 6. Each of them had like I think it was Java 7, Java 8 and Java 9 or something, just like different tiers of the architecture that were on different versions of Java, and just getting it updated was a huge struggle. Getting a uniform homogeneous Java, it was always an issue in GitHub, just like an issue with an endless amount of comments, like, "Okay. Update to Java 7." It's just really long. Nobody can ever finish it.

I think part of the reason for that is because the release cycle was so long and there were so much in each Java release. Am I painting an accurate picture of the state of enterprise Java and the motivations for getting to a more frequent release cycle?

[00:22:28] GS: Yeah. I mean, I think you're absolutely right, that part of the problem with those long release cycles is they would happen so infrequently that everyone knew it would be a big effort, and so you would tend to sort of put it off and then it kind of snowballs. Then the same thing is true for the contents of the releases, right? Because the releases were so long in between, there is sort of a great desire. I mean, everyone who's working on the JDK wants to get good features in and make things better for developers.

But what ends up happening is you get lots and lots and lots of features baked in. So the cumulative effect of that is that when that version comes out or when it came out, it would be a very disruptive event. So people would race around trying to get their libraries updated and all of their tools, like their IDEs and others and other things updated. That often was a process that took quite a while.

In addition at that time, I would say even looking back towards the sort of early 2000's, much of that work was being done in closed source and without many sort of early access builds if at all. So it was the kind of thing where it was tough for people to know what to anticipate or to try it out and give feedback, and it was tough for folks like my team working on the JDK to know what kinds of issues people were running into. So that would all kind of snowball.

I'm really happy to say that where we are now, these new releases of Java we're doing that are coming somewhat more frequently, have a lot less in them. They're intentionally very incremental. So the idea there is that there's just not as much that's going to get in your way and cause issues and cause delays and people being able to update, especially because we continue with the practice that we've had of running hundreds of thousands of compatibility tests and now having the sort of added transparency that we get of doing all of our development in open source and providing usually about weekly early access builds of the next version that is under development. It just really gives people a way to try things out that are works in progress

and give us feedback so that we can make sure that they're just really, really sound and stable and don't cause issues when they come out.

I think one other thing that I'd sort of add there is that the module system that we added in JDK 9, really, one of the main intents for that was to make it easier both for us in making the JDK, but also of people making libraries, whether they're ones that they're distributing and encouraging people whose across the industry or whether they're even just kind of local in scope, a library that you're using within your enterprise to be able to distinguish between the public API of your library that you guarantee you're going to keep consistent and private implementation details, which you need to have the flexibility to change and improve as you're going forward.

So in the past, it was sometimes kind of unclear or difficult for people to understand when they had inadvertently built in a dependency on something that they really shouldn't be dependent on. So then when that thing changes, "Oh, wow! I have an update problem." I think that as people are starting to embrace the module system and use it, it just makes it much, much easier for developers of those libraries to be able to evolve and then improve them and makes the people who are using them much more resilient towards these kinds of changes.

[00:26:00] JM: Are there programming languages that are on like a continuous delivery or a continuous release process, or is there something about the nature of programming languages that makes this infeasible?

[00:26:12] GS: I'm not aware of any that update much more rapidly than we are doing. I think that there are quite a number that at least have an ambition of doing releases on about a six-month cadence. The reality is, I mean, we do it more frequently than that. We have releases that are done in between. Basically, we have security stability and performance updates that we do basically four times a year. So you combine that with the six-month feature releases, and that means that we're actually doing at least six updates a year.

Now, those are slightly different in terms of their nature. The main difference is that for intermediate releases are ones that are done without any changes to the language and APIs. But I think that it does end up at a point where you're doing it basically just about every other month.

Then in terms of other languages, I have seen a number that are doing every six months. I haven't really seen any doing more frequently than that. I think part of that is that people do want stability. They want the ability to know that they're writing code that's not going to immediately be no good and have to require a lot of change, which is one of the reasons why we care as much as we do about compatibility and making sure that the changes that we introduced in these feature releases tend for the most part to be additive, right?

We add new capabilities to the language. We add new APIs. We're very, very careful about cases where we might want to renew something and tend to try to make sure that people are aware of that far in advance. So we'll do something like deprecation. I think we still have things in the JDK that have been deprecated since like 1997, but we've actually tried to get better about deprecating things, then marking them as deprecated for removal, which indicates when they're going to go away, and then actually making good on that, because at some level you do have to make sure that you are pruning the garden a little bit so that you can have new healthy growth, right? We try to be careful of that.

There are also other interesting aspects that you get. So as an example, when you're evolving the language itself, if you're doing something like adding a new keyword, for example, you have to be careful for collisions, right? If you all of a sudden add a new keyword that are maybe existing code out there that actually uses that keyword as something else, like a method name, or something.

One of the things that we tend to do there is try to look at some of the great resources that are out there in terms of code and do analysis of, "If we did this change, what would happen?" We'll sort of test it against all the code on Maven central or something like that just to give you an idea in order to make sure that we're doing as little harm as possible in the efforts of trying to improve things for people.

[00:29:02] JM: It can be kind of fun to go back through things that have been deprecated, like things that were deprecated long ago and just get a little bit of an assessment of what didn't work. I remember doing this a little bit in college, like reading comments about hot new ideas in the programming language world, like particularly the Java world and it's like, "Oh! Well –" and

then it didn't work out. It's just like products, like Google sunsets products all the time and it turns out that programming languages do the same thing.

[00:29:33] GS: Yeah, that's true. I think some of that also comes with style, right? If you sort of looked 15 years ago, like you want to go long on XML, and maybe today not so much. Although, there are still tons of these.

[00:29:48] JM: Oh, yeah. Yeah, that's right. I mean, this podcast is going to be distributed over RSS, which is XML.

[00:29:52] GS: There you go. Yeah. Yeah, I think this is true. There are definitely trends that occur in programming language circles, and what we tried to do is not lurch too much back and forth to the new hot thing, but really sort of try to understand where there are things that are going to be long-term persistent that actually bring value to people and then take a step back and go, "Okay. Well, what can we do to support that need?" Sometimes it's something like putting in directly a particular library of support. Sometimes it's actually finding something underlying in the JVM implementation that can support other people making those kinds of libraries.

So we try to be pretty conscious of that. I think Java, when it originally came along, was attractive for many people because it did have a large and extensive standard library. On the other hand, we've always try to make sure that we weren't trying to make the library so broad that it gave people in the ecosystem no room for innovation and producing things themselves, which is a hard balance to achieve, right?

[SPONSOR MESSAGE]

[00:31:08] JM: Cruise is a San Francisco based company building a fully electric, self-driving car service. Building self-driving cars is complex involving problems up and down the stack. From hardware to software, from navigation to computer vision. We are at the beginning of the self-driving car industry and Cruise is a leading company in the space.

Join the team at Cruise by going to getcruise.com/careers. That's G-E-T-C-R-U-I-S-E.com/careers. Cruise is a place where you can build on your existing skills while developing new skills and experiences that are pioneering the future of industry. There are opportunities for backend engineers, frontend developers, machine learning programmers and many more positions.

At Cruise you will be surrounded by talented, driven engineers all while helping make cities safer and cleaner. Apply to work at Cruise by going to getcruise.com/careers. That's getcruise.com/careers.

Thank you to Cruise for being a sponsor of Software Engineering Daily.

[INTERVIEW CONTINUED]

[00:32:28] JM: All right. Well, let's get into some of the newer features of Java 13, because I think we've given a pretty broad perspective for Java and how the development process works. Java 13 has a feature called dynamic application class data sharing, and I think that's a good example of how performance can be improved of the Java runtime overtime. So in order to explain what dynamic application class data sharing is, we first need to explain what application class data sharing is. Can you explain what application class data sharing is?

[00:33:05] GS: Yeah. Basically, the idea is that the way that Java traditionally works is the Java runtime startup. It goes and it loads your application and all of its dependencies. Sometimes it does sort of just on time. It doesn't necessarily go out and exhaustively do it right away. But basically it will tend to do a fairly quick pass that's doing class loading, starting to run, doing a little bit of compilation, and then over time, monitoring the code paths that are hot and basically doing better and better at code generation in order to make this run really, really quickly.

So that is one of the things that has meant that you have something that starts up in a reasonable amount of time, but then over time the performance improves as you're actually running. Okay? Basically the next step then is to say, "Okay. Well, that's great, but I'm always running this application in the same environment. Maybe there's something that I can do that changes the balance a little bit so that I don't need to do all of that work on startup every single

time, but instead I can start up a little bit more quickly and actually get to the warmed up state more quickly."

So the first step that we had there was class data sharing of the things that are in the JDK itself, and then sort of the next level is doing that including your application as well, right? Basically what this ends up doing is taking a bunch of the work once your code paths have gotten hot and a bunch of optimization has been done and figuring out how to take a lot of that data, that metadata, and basically creating a shared library that can just be reloaded and memory mapped in when you're starting up rather than having to generate it each time.

[00:34:50] JM: Can you give an example of what kinds of metadata sharing between classes can occur?

[00:34:58] GS: There's just all kinds of stuff that can be in there. Basically, if you sort of think that a class in and of itself doesn't really know that much about what's going on, it's in the context where it's being called that you can do things like in lining and other things, creating the – Basically, your Java code goes through a stage when you compile it where platform-independent object code is created, right? The class file. But then even from the class file, when you're in runtime, it's at that point that it knows, "I'm on this operating system. I'm on this hardware. I'm on this particular generation of this hardware that it can create the native code and produce the internal representation that's used to quickly be able to map to native code." Those are the kinds of things that you don't want to have to generate each time.

[00:35:53] JM: And so the development of dynamic application class data sharing archives, of all the things that you could've invested time into, why was that – And how did you know that that was a valuable enough endeavor to engineer? Because this sounds like something that was not easy to engineer, so I'm looking to get a window into the development process. How you prioritize dynamic CDS archives.

[00:36:24] GS: So this is exactly one of these features that works and is actually more important when you are running in the cloud, right? So where you're running as an example, something that you expect to come up quickly and be at a warmed up speed quickly, because it may not be around that long. It's sort of extra important. Then the sort of additional notion of creating

something as a shared library that even could be used across instances in the cloud means that you're able to reduce footprint so that you are sort of consuming as few resources as possible and ultimately hopefully using that saved money.

[00:37:07] JM: Are you able to share data between containers?

[00:37:12] GS: Yeah, you can actually set it up so that that works.

[00:37:16] JM: That sounds hard. I mean, that sounds hard to engineer. Can you tell me a little bit about the development process of getting that working?

[00:37:24] GS: Well, it's kind of what you would expect, right? We sort of look at and analyze what underlying capabilities are there, and then we make sure that we are using them. So in this case, it's really sort of understanding the underlying capabilities. Making sure that Java is not doing something weird or making different assumptions. Then the other sort of key piece is trying to figure out how to take data make it as easy to use as possible and not sort of require you to jump through a lot of hoops, and that's where we get to sort of the dynamically generated part, right?

[00:37:57] JM: The ability to leverage commonalities between containers, this to me is an interesting subject especially when you start talk about like serverless functionality, because with serverless, I don't know if you've looked in this very much, but like there's this cold start problem where if I want to have – If I want to deploy an AWS Lambda or Google cloud function or whatever your serverless plan. But they all have this issue where if I want to execute my application on-demand, I need to load all the context, including any programming languages that this thing is going to need to execute.

Then if you have infrastructure that is basically warmed up in the sense that you have like a JVM running or a JavaScript V8 running and it's already loaded on the node, then you save a lot of the startup time. So all you have to do is throw your custom application on to the node that's already been pre-warmed with the right virtual machine or infrastructure or whatever and it's going to execute a lot faster. So that can improve the functionality of the functions as a service things.

On the other hand, then you start to lose some of the isolation benefits, right? So any reflections on that tradeoff?

[00:39:16] GS: So I think it's hard to know when you're making a programming language exactly which style people are going to want to use. So what we try to do is sort of reduce the friction for being able to make that choice, because which one is the right one to choose? Maybe different depending on what you're doing.

Ideally, you're not having to make that choice purely because of limitations in the tool chain that you're using, right? As you say, doing something like having a hot start based on recycling a JVM that's been there and maybe even having applications that have been loaded. So you're basically making additional request to something that's already up and running is one kind of architectural style. That would be one that would favor certain aspects, but may be can be challenges on others, like isolation.

So the change that – The new capability that we're bringing here makes it so that you aren't sort of restricted to choosing one style of architecture purely because the tools you're using aren't good at the other one. So in this case what we're trying to do is make it so that if you want to do something completely isolated in its own container with its own JVM and be an instance that's just kind of coming up quickly and doing something going way, that style of architecture is one where you can absolutely choose Java.

[00:40:34] JM: Let's talk about another element of Java 13, which is garbage collection. Garbage collection is of course an element of every Java iteration, but as I was preparing for this show, I was getting a little bit caught up on the Java ecosystem and I learned that there is a newer Java garbage collector called ZGC. Can you shed some light on ZGC?

[00:40:59] GS: Yeah, absolutely. So ZGC is basically a garbage collector that is made for being able to make very, very low latency pause time targets and guarantees. Traditionally, one of the challenges that people have when they choose a language like Java that does automated memory management is that you're resending some amount of control of when and how things

are allocated, but more importantly when all of the potential garbage that you've created is freed up.

So there sort of different kinds of concerns that you can have depending on the particular workload and the situation you're in. Sometimes what people care a lot about is throughput. Sometimes what they care a lot about is latency, right? You want to know that requests that are coming in are going to be serviced within a certain amount of time, a certain threshold. More importantly, you're almost never going to have a transaction that is just sitting there hanging for long period of time from the perspective of whatever client has made that request.

So the idea with ZGC is that you can set a plus time target and the system will respect that. So basically it will sort of try to do its best at keeping with that very low latency target, and if it finds that for some reason it's starting to take longer than that, it will kind of back off and give you the opportunity to continue running.

The key here is that – And by the way, this is something that a number of folks have done in the past even in some of our own products, like JRocket. JVM did that a decade ago. The difference is that in this day and age, the heap sizes are growing tremendously. So what ZGC has as its design center is providing those kind of guarantees while scaling to terabyte heaps.

[00:42:52] JM: The heap, if I remember correctly, heap is memory that is allocated for objects as supposed to memory that's allocated for stack frames or something?

[00:43:03] GS: Yeah. Basically, any of the things that your application is creating is probably doing much of allocations of memory in the heap. The Java memory system generally manages that for you, right? So it basically worries about where are all of the objects? Where are all the data? If garbage collection is done, it typically will not only collect objects that can no longer be reached. It also may move them around.

From one indication of the garbage collection to the next, the objects that your application has created may have ended up in completely different places, hopefully being compacted so that underlying facilities for handling things like paging and so on works smoothly. As well as if you're lucky, being able to take advantage of things like prefetching if you're working on a collection of

data and making sure that the data that your CPU needs to operate on is actually at the CPU that it needs to be at the time you're trying to do an operation on it.

[00:44:02] JM: One more question about garbage collection. How do you roll out a new garbage collector? Because that sounds like so hard. I mean, if you change the garbage collector, you may like totally change application performance for like millions of users, right?

[00:44:19] GS: Absolutely. So it tends to be something that we're very, very careful about. So, interestingly, people who use Java come in all flavors, right? There are people who really, really care about the performance of their garbage collector and will spend months tuning, and there are those who are like, "Look, I just want it work. Let it do what it does behind the scenes."

Again, it also can depend on the type of application that you are writing. If you're writing a trading system that may be different from if you are writing an IDE. So those are also aspects that tend to be important.

When creating a totally new garbage collector, it takes quite a while to get to the point where you have something that's working at all. But then what you tend to want to do is try it on what you hope are a bunch of representative applications and sort of make measurements about how it's working, and then overtime increase the circle of applications that you're trying it with. Typically, you want to try it with the kind of application that whom is going to display better behavior with your garbage collection algorithm, but you may want to do the reverse as well, right? Try to figure out the things that you think is not going to be as good at and try it on those and maybe try to minimize how bad it is on the things that it's sort of okay, but it'd be bad on.

I think another in a big concern that tends to come in is how tunable do you make it versus how good is it out of the box. So we tend to do all of those things, test all of those things. We will also test how it works overtime. So doing something that will do runs that go on. Throwing all kinds of stuff at it for weeks and months to try to make sure that the performance keeps up and doesn't degrade overtime and you don't have lates and all that kind of great stuff.

Then, ultimately, the real test is we'll download lots of applications from across the Internet. We'll run it on a bunch of internal applications that we have. Oracle has tons of things running in

Java. So there are lots of great applications we can try. But then putting it as, first, generally as a project outside of the main line of Java and delivering those directly that people can try and give us feedback on.

Ultimately, getting it into the JDK, but as an experimental garbage collector, each of these steps give us sort of incrementally more people trying it and using it and giving us feedback and ultimately getting to the state where it becomes a supported garbage collector. Then overtime, it may actually be on a path to becoming a new default collector, or it might not be, right? It might be a collector that remains in there as a supported collector for years and years. But because it's more special-purpose than general purpose, it's sort of might remain in that state for quite a while.

Then sort of to round out the lifecycle, we are actually at kind of a state where we may look at some of these things and say, "Does this garbage collector still make sense? Is it something we should continue to maintain or is it perhaps something whose original purpose and need to have been obviated by later developments?"

We're actually coming up on a stage where that may occur for at least one of the garbage collectors that's been in Java for quite a while. So that'll be interesting to see because, of course, the more things you're carrying around in backpack, the slower you tend to go. We actively look at places where new things and new developments can kind of replace older things and continue to make progress for everyone using Java.

[00:47:45] JM: Georges, thank you for coming on the show. It's been a pleasure talking to you.

[00:47:47] GS: Thank you, Jeff. I really appreciate it, and I look forward to talking with you again.

[END OF INTERVIEW]

[00:48:01] JM: If you want to extract value from your data, it can be difficult especially for nontechnical, non-analyst users. As software builders, you have this unique opportunity to unlock the value of your data to users through your product or your service.

Jaspersoft offers embeddable reports, dashboards and data visualizations that developers love. Give your users intuitive access to data in the ideal place for them to take action within your application. To check out a sample application with embedded analytics, go to softwareengineeringdaily.com/jaspersoft. You can find out how easy it is to embed reporting and analytics into your application. Jaspersoft is great for admin dashboards or for helping your customers make data-driven decisions within your product, because it's not just your company that wants analytics. It's also your customers.

In an upcoming episode of Software Engineering Daily, we will talk to TIBCO about visualizing data inside apps based on modern frontend libraries like React, Angular, and VueJS. In the meantime, check out Jaspersoft for yourself at softwareengineering.com/jaspersoft.

Thanks to TIBCO for being a sponsor of Software Engineering Daily.

[END]