EPISODE 684

[INTRODUCTION]

[0:00:00] JM: Kotlin is a statically typed programming language that started as a JVM language. It gained popularity because it reduces the amount of boiler plate code required for a typically Java project. Many of the early adopters of Kotlin were building Android apps or Java applications, but it's grown to have a variety of use cases including at companies like Uber, Pinterest and Atalassian.

Andrey Breslav is the lead language designer of Kotlin at JetBrains. He joins the show to describe the original goals of Kotlin, the compilation path of the language and how it has moved beyond its days of only running on the JVM.

Before we start, I want to mention that we are looking for a couple of roles including writers and podcasters. We have these roles mentioned on softwareengineeringdaily.com/jobs. So whether you're interested in operations or editorial, we'd love to have you apply.

[SPONSOR MESSAGE]

[00:01:07] JM: Accenture is hiring software engineers and architects skilled in modern cloud native tech. If you're looking for a job, check out open opportunities at accenture.com/cloud-native-careers. That's accenture.com/cloud-native-careers.

Working with over 90% of the Fortune 100 companies, Accenture is creating innovative, cutting-edge applications for the cloud, and they are the number one integrator for Amazon Web Services, Microsoft Azure, Google Cloud Platform and more. Accenture innovators come from diverse backgrounds and cultures and they work together to solve client's most challenging problems.

Accenture is committed to developing talent, empowering you with leading edge technology and providing exceptional support to shape your future and work with a global collective that's shaping the future of technology.

Accenture's technology academy, established with MIT, is just one example of how they will equip you with the latest tech skills. That's why they've been recognized on Fortune 100's best companies to work for list for 10 consecutive years.

Grow your career while continuously learning, creating and applying new cloud solutions now. Apply for a job at Accenture today by going to accenture.com/cloud-native-careers. That's accenture.com/cloud-native-careers.

[INTERVIEW]

[00:02:45] JM: Andrey Breslav, you are the lead language designer of Kotlin at JetBrains. Welcome to Software Engineering Daily.

[00:02:51] AB: Hello. Thanks for having me.

[00:02:52] JM: Yeah. I'd like to start off by talking a bit about what you do. So I haven't met many people who are professional language designers. How did you get involved as a language designer?

[00:03:06] AB: Well, it's I think mostly an accident as it usually happens. I was doing some language related research for my PhD, which I never finished, and just basically friends and former coworkers invited me over to JetBrains where they were discussing possibility of creating a language. It was interesting. I was very skeptical actually in the beginning about creating a progress language from scratch. But that was a very insightful conversation. So I got really convinced that we should do it and that we can pull it off. Basically, that's how it started. So I was never preparing myself to be a language designer.

[00:03:43] JM: Had you been studying language design at all?

[00:03:45] AB: I don't think there is such a thing as studying a language design. Really, I mean, there is no textbook that I'm aware of.

[00:03:52] JM: Well, there's programming languages courses.

[00:03:54] AB: Yeah, there are. Well, I had such a course in the university. I never taught one, by the way, which I may do at some point. I did some theoretical computer science that involves types systems and other things also at the university. So I was kind of prepared. So I knew some of the theory and could understand the language in a textbook, let's say.

[00:04:16] JM: What was it about your background that made your friends reach out to you and say, "Hey, maybe you should come over and talk to us about whether we should be designing this brand new language."

[00:04:27] AB: For my PhD, I was working at domain-specific languages, which is also a kind of language design, but not general purpose languages just like for small languages. It was my academic interest. I was reading papers in this topic, especially about composability and the extensibility of languages. Yeah, I think this was the main reason as far as I remember.

[00:04:47] JM: I guess we could start getting into Kotlin. I would like to revisit that domain-specific language stuff a little bit later, but to get to Kotlin. Kotlin is a JVM language. It's 100% interoperable with Java. Why was Kotlin originally created?

[00:05:03] AB: Well, Kotlin is now somewhat more than a JVM language, but it was conceived as one. It was back in 2010, and the situation of the Java ecosystem was such that Java was the most popular language as it is now still on the JVM, but it wasn't progressing much. In any case, there was a strong feeling that many people could benefit from a new language for this platform in terms of productivity, in terms of really modern programming experience. People at JetBrains wanted such a language for themselves as much as for their users.

So before I joined, they evaluated existing alternatives on the JVM, at that time it was Groovy and Scala. There were concerns about either of them. So they were seriously thinking about creating a new one that would be 100% interoperable and would enable the tooling very well, because JetBrains is a tooling company. So we heavily rely on tooling in our everyday life, and our users rely on that too. These were main driving constrains, to be pragmatic and to be toolable. Yeah, this is why it was created.

[00:06:18] JM: As you said, it was originally a JVM language. Can you talk about the advantages of building a language on the JVM platform?

[00:06:28] AB: There are so many. There was a huge ecosystem. So JVM as a platform brings so many libraries, so much user expertise, framework, tools, everything, and an excellent virtual machine first of all actually. So it's a very rich ecosystem with a lot accumulated by the communities. Kotlin can access all of that. Every library that's available for the Java language that's been written over the last 20 years is available to Kotlin users. So you don't start from scratch. You don't have to rewrite every library out there.

Also, all the other tools, like profilers, debuggers, everything, and the virtual machine is excellent. It's very famous for being very robust and very efficient. So it's a very good starting point. Also, the virtual environment takes care of so many things. You don't have to care about as a language designer, like your memory model is just taken care of. You just rely on that.

It is also like nothing limits the dreams of a programmer as a compiler, right? So the same for a language designer and the runtime. So if I designed my language for JVM, it does a lot for me, but it also constrains me, but I think the benefits outweigh all the possible limitations by far.

[00:07:44] JM: What are some of those constrains.

[00:07:46] AB: Well, there is basically no direct memory manipulation. For example, if anything has to be passed around, it's either a primitive, which is a very limited set of representations, or an object, which has an overhead of another header. This is something that the Java team is now working on in Project Valhalla, but still in the production, JVM today, if I want to pass two values together around my program, I have to wrap them in an object and it's memory overhead. That's just one limitation. Also, of course, it's a mature ecosystem. It has accumulated a lot of legacy, like we have a monitor in every subject, for example, our arrays, our covariant, so on and so forth.

Kotlin is trying to kind of fix it on the surface. So in the Kotlin language, there was no monitor in any object. So you can't just synchronize randomly on objects. You cannot use erase as

covariant. They are invariant. So there are no erased or exceptions in Kotlin programs that do not abuse the underlying Java layer and many other things like this. So we're trying to reface as much as we can on the language level.

[00:08:56] JM: When Kotlin was designed, if I recall, this was around the time when, I guess, Scala and Groovy, those languages had been designed on the JVM and they were great for doing functional programming, but Kotlin was more taking the approach of, "We're going to stick with just non-functional –" What's the other – Imperative? No, it's not imperative.

[00:09:20] AB: Well. It's confusing at least.

[00:09:23] JM: Confusing, okay. But Kotlin was more about the productivity side of things. It's like, "Let's make Java look more like Ruby."

[00:09:29] AB: Well, I wouldn't say that. I never said that. So let's try to sort out the terminology here. There is like the purely functional paradigm, like Haskell is, for example, a purely functional language. Kotlin is nothing like that, and Scala is nothing like that either. Actually, mainstream languages are not pure paradigm anymore. So modern popular language is actually something a single paradigm, like a functional language or an object-oriented language.

We mix and match different ideas from different traditions, and Kotlin as well as many other languages combines object-oriented, functional, structural and just gets some kind of balance. So it's very much a question of how you balance different ideas in the same design. Kotlin had functional features from the very beginning, high order functions, function types, lambdas, were all in the design from day one.

So in this respect – Of course, generics. Generic classes are definitely feature motivated by functional background. So all these is from the functional side of things, but we also have classes and interfaces, which is more from the object-oriented side of things. So it's just what we believe to be a pragmatic balance of these ideas.

[00:10:47] JM: What's the path to a Kotlin program being interpreted and compiled down to Java bytecode?

[00:10:56] AB: Just run the compiler.

[00:10:57] JM: What are some of the things that will get converted into – I mean, is it just directly from Kotlin code to Java bytecode, or is there an intermediate representation before going to Java bytecode?

[00:11:09] AB: Oh, yeah. There was a number of things inside a compiler. So the compiler that we use now for the JVM, it builds a representation of the program matching the source very closely, because we also use it in the ID. Then we generate bytecode from that with the aid of some static analysis type checking, so on and so forth. We're working on a newer version of the compiler that will have more layers of intermediate representations. Currently, it's just this concrete syntax tree in the middle.

[00:11:38] JM: What were some of the issues with Java that Kotlin makes alternative decisions on in its language design? What are the other issues that you're continuing to improve on?

[00:11:49] AB: Well, there is quite a number of things. One this that we try to mitigate was the ceremony that Java is famous for. We're trying to get rid of the things that the user doesn't really need to indicate to be understood. I think it's ideal when the compiler understands everything that a human can understand. Of course, we can't there, like computers are a lot dumber than humans. But we are trying to get there.

An important part here is that if a human struggles understanding something, even if the compiler can figure it out, the language should not allow this, or at least should resist this. So we're trying to make a language where the program is as close to human thought as possible. Yeah, there we were trying to remove a lot of hoops in between the two. So you don't have to have a class to declare a function, for example, in Kotlin for one thing. We have extension methods that let you minimize the core APIs. There are many more things we use, for example, in line as a language feature, and we use declarations like variances [inaudible 00:12:55]

variance, which we also support, but it's like most classes are just variants on the declaration side.

So there is quite a number of things. Actually, we beautify the collection interfaces. We have read only and mutable collection interfaces separated so that you can express the [inaudible 00:13:13] in the type system. You can express nullability in the time system, which is one of the big plagues in the Java world and beyond the Java world actually, which is notoriously called a billion dollar mistake. Yeah, very many exceptions people collect from Java programs or just null pointer exceptions. Kotlin helps with that a lot.

Basically, we're just treating null as a proper citizen in the language as a proper value that can be expressed in the type system, which makes it usable, because otherwise you're probably advised to avoid nulls everywhere across the Java ecosystem, or C++, or whatever other code style and your business logic you're supposed to avoid nulls. This is difficult to say the least. But In Kotlin, you don't have to. It just works, because the type system captures your intent and controls everything around dots. So this is just a few things off the top of my head and there is many, many more.

[SPONSOR MESSAGE]

[00:14:17] JM: Stop wasting engineering time and cycles on fixing security holes way too late in the software development lifecycle. Start with a secure foundation before coding starts. ActiveState gives your engineers a way to bake security into your language's runtime. Ensure security and compliance at runtime.

A snapshot of information about your application is sent to the active state platform. Package names, aversions and licenses and the snapshot is sent each time the application is run or a new package is loaded so that you identify security vulnerabilities and out of date packages and restrictive licenses such as the GPL or the LPGL license and you identify those things before it becomes a problem.

You can get more information at activestate.com/sedaily. You want to make sure that your application is secure and compliant, and ActiveState goes a long way at helping prevent those

kinds of troublesome issues from emerging too late in the software development process. So check it out at activestate.com/sedaily if you think you might be having issues with security or compliance.

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

[INTERVIEW CONTINUED]

[00:15:47] JM: When thinking about some of those things you mentioned, one principle that comes to mind is Kotlin seems like it was designed to produce less code with the same expressivity. Maybe you could tell me if that's right or you could refute it, and perhaps tell me some of the other language design principles of Kotlin.

[00:16:07] AB: I would say the actual guiding principle that is related to a number of lines of a file is that we think that reading code is much more important than writing. So we're trying to remove as much noise in the code as we can so that it's as readable as possible. It also causes the reduction of the code size. Our own measurements and other people reports tell us that it's about 20% to 30% reduction in like an average programming in the number of lines. Yeah, the design principle there is that we are reading a lot more code that we're writing. So we have to optimize for readability. That's one important thing.

There is a lot in the language design that relate to interop all source, like Java interop on the JVM or JavaScript interop, the JS platform, or interop with [inaudible 00:16:58] in the Kotlin native backend. So we're doing a lot of work from the language design standpoint to facilitate seamless interoperability with a platform.

There was actually many more things. We're trying to make things explicit, where they're not obvious, and this is why Kotlin has fewer pleasant conversations between types and some other languages, for example, and there are many more reasons to do this as well.

Yeah, these are, I guess, the driving principles. Yeah, another one which is – Well, you can call it design principle, but it's actually like a major constraint for us. We have to make the language toolable, because I can come up with a wonderful language that's very hard to develop tooling

for. We know many examples of this throughout the history, C++ and Scala being very well-known examples. So we examine our design choices in the view of how the ID will treat this. How it will increment a compilation will work with this, so on and so forth. Tooling is very important in our process.

[00:17:58] JM: You mentioned a term there, the interoperability. Can you go into more detail what you're talking about there with interoperability?

[00:18:05] AB: Yeah. On a high level, the idea is that one should be able to call into API's existing on the platform. For example, if I'm on the JVM, every Java library should be available to me as a Kotlin programmer. Also, my Kotlin code should expose APIs to the Java world. So if you write a library in Kotlin, you can very easily expose the API to the Java client so that it can be used equally well from Java and Kotlin. This is the main idea of the interop. Technically, there are many interesting questions there, like how do objects behave when they cross the boundary between Java and Kotlin? For the JVM, it's trivial. They just don't change as the same objects.

For other platforms, there can be differences. For example, In JavaScript, Kotlin objects are not exactly the same as JS objects, but they are largely interchangeable, so on and so forth. There can be tradeoffs there, but on the JVM, the interop is just transparent. It's the same objects. It's the same types. Basically, you can just call across the boundary without any overhead and without any ceremony.

[00:19:10] JM: Talk about that in a little more detail. So if I have a JavaScript program and a Kotlin program and I want to have them interact with an object, are you saying that I might have to serialize the Kotlin object so the JavaScript program can read it?

[00:19:25] AB: Oh, no. No. So when we compile Kotlin to JavaScript, we can product objects, make them Kotlin classes. I have a Kotlin class, I instantiate it, I get a JavaScript object. The question is –

[00:19:35] JM: Sorry. You said when you compile Kotlin to JavaScript.

[00:19:38] AB: To JavaScript, right. Yeah, just for background. Kotlin can compile not only to JVM. JVM is one of the targets, and then you can compile to JavaScript or to different native platforms. Not all the same libraries are available across different platform. JDK is not available on Linux, for example. I mean, when you compile it to a native Linux binary, you don't have access to JDK APIs, but languages are the same and the standard libraries are the same.

So when you compile to JavaScript, Kotlin objects created from Kotlin classes won't be exactly the same as just native JavaScript objects. They will have more metadata. If I see an object just deserialize with normal JSON deserializer, it will not have any Kotlin class information. So it will be just a raw JavaScript object for me. So it's not exactly the same, but for most purposes, Kotlin can use native JavaScript objects and JavaScript can use Kotlin objects more or less transparently.

[00:20:34] JM: Since you mentioned it, let's go into more detail in the process of being able to compile Kotlin to JavaScript or compile to other languages. What was the motivation for that for the ability to compile to languages other than the JVM platform?

[00:20:49] AB: I think the biggest motivation there is the ability to share code, because modern applications drawn across many platforms. So when I have like a more or less normal project nowadays, I have a server that's running possibly JVM, for example, and I have a web client that's running JavaScript, and a mobile client running Android, and another client running iOS, for example, and they all have a lot of overlap in terms of code that's running there. But they're all different platforms. So it's not easy to share that code. If you write everything in JavaScript, for example, you can do it, but then you have to do JavaScript as a runtime instead of native runtime.

Kotlin compiles to native runtimes. In the browser, Kotlin runs in the JavaScript VM. On Android, Kotlin runs on the Android VM. On iOS, it's native binary. On the JVM, it's basically a java application. But still you can have a common set of source files that's compiled to all those platforms at the same time or two of them or whatever pattern you want to share. So this code can be reused without rewriting it in a different language, or with different dependencies. It can be just shared unchanged across the platforms. This includes things like business logic, for

example. It's a use case we're working hard on now sharing business logic between mobile applications, writing code for Android and iOS with the same business logic written Kotlin once.

[00:22:17] JM: What are the challenges of implementing that?

[00:22:20] AB: Oh, there are so many. Basically, we have to provide the same semantics on all platforms, which is I don't think it is really 100% achievable in a pragmatic case. So if you want to make it performant and really usable and interoperable on all platforms, there are slight changes in the semantics, but it has to be similar enough on all platforms for the code to be actually runnable. Basically, we designed a system where you can have an API exposed to all platforms. Then behind this API, have different implementations that can call into platform libraries that are on different platforms.

Then there was lots of issues with how you compile this, how you distribute this, how you deploy this, so on and so forth. So it's quite an undertaking. We're not done there. I mean, we have experimental support for multiplatform projects, and we're still working on it, but there is a lot news every month about it.

[00:23:14] JM: Kotlin has some different types, such as data classes and companion objects and some other types. Can you talk about some of the different types that Kotlin provides and why new types were introduced in Kotlin?

[00:23:28] AB: First of all, they're not types. I think Kotlin only has one kind of types, which is classes, or interfaces, or objects. They're all the same. We call them classifiers in Kotlin. But there are different keywords that can proceed your class declaration, for example. There is data classes, which are compilers, normal classes actually, but there was this keyword in front at tells the compiler to generate a bunch of useful things for you as equals and hashcode and to string and some other convenience methods generated from one liner.

Basically, this is automating very common use case of having a class that holds data and is manipulated as an aggregate without any fancy logic in it. So that's for data classes. There's, for example, enum classes. It's the same concept as enums in Java or other languages, the types of enum. So what else do we have? Companion objects. It's a different story. We have first class

[inaudible 00:24:28] language. We call them objects, the same as Scala has. An object can be attached to a class so that its members are accessible in the class main. This is done to get rid of the idea of static members. Basically, every member is Kotlin is an actual instance member, not static. Then if you want to call something on a class, it's an instance member of a companion object. The benefit is that the companion object can implement interfaces or extend classes. So there is a lot more code reuse.

Yeah, so this is the companion modifier, and there is I think something – Yeah, there are CO classes. CO classes are somewhat an extension to the concept of a enum. It's a closed higher key where you can have only a given number of subclasses. A CO class always knows all of its subclasses. So a compiler can do nice checks at the call side when you're matching, doing instant soft checks against this higher key. These are just different modifiers that make classes behave slightly differently for common use cases.

[00:25:30] JM: Kotlin does not have static typing for some of the basic types. Can you go into that in a little more detail and explain the decision making around dynamic typing versus static typing?

[00:25:43] AB: So I'm not sure what you refer to as basic type. So Kotlin on the JavaScript platform has a language extension that's called dynamic types. So you can have a dynamic type. It's a single type in the type system. It's available only for JavaScript and is needed for interoperability or the JavaScript world, because the JavaScript world is not expressible in the rigid object-oriented type system. Basically, we just have a type that says, "I don't know what this is. I can call any member on this and we'll just compile through to a plain JavaScript call and [inaudible 00:26:17] at runtime. If it works, great. If it doesn't, then it fails as normal JavaScript fails.

[00:26:22] JM: Okay. Does Kotlin do anything unusual around type inference? Whether we're talking about JavaScript or just regular Kotlin?

[00:26:29] AB: Well, talking about type of inference, I don't know what to call unusual, because there is nothing usual. I mean, type inference is a interesting kind of language feature, because there is a lot of science around type inference, but it's all largely irrelevant to languages like

Java or Kotlin, because basically we have a different kind of type system from those papers that are written about type inference. There is just the distinction between nominal type systems and structural type systems and most papers are written about structural and we're nominal.

In any case, we're doing something along the same lines as Java or, in some sense, Scala or C#. It's a number of heuristics that help us infer types at the call site. It's a pretty complicated algorithm. So there is a lot of different things interacting there. It's basically witchcraft in terms of engineering.

In general, there is nothing super unusual. There is nothing so different in Kotlin from other languages. There are tradeoffs. We took these choices. Some others took other choices, but it's all more or less known.

[00:27:34] JM: Okay. Let's talk a little bit about the tooling. Can you describe the tooling landscape around Kotlin?

[00:27:39] AB: Yeah, there is quite a bit tooling. So IDE-wise, there is IDE support for IntelliJ family of products. There is an open source IntelliJ community edition that supports Kotlin JVM> There is a commercial IntelliJ ultimate that supports JVM and JavaScript. There's also Android Studio, which supports Kotlin JVM, and this is how you can write Kotlin for Android. So these are IDEs that run on the JetBrains platform, IntelliJ. Then there is Eclipse plugin that we also invest some effort in, has some users. So Kotlin is available in Eclipse.

We used to have a NetBeans plugin as well as a student project. It's currently not actively maintained, but it's open source. So anyone can contribute there. That's it about IDEs. There's also a set of smaller plugins for other lightweights IDEs, like Sublime and I think there's a theme integration and so on and so forth.

This is, I think, the most complicated tooling that we have. There's also integration with those systems. There is Gradle and Maven plugins, and they are quite powerful. So we have support for incremental compilation in both Maven and Gradle at this point if I'm not mistaken. Gradle for sure. Maven, I'm almost sure we do support this already.

Gradle currently has slightly more users in Kotlin, because Android is using Gradle by default and we have quite a lot of work done to speed things up and so on and so forth. These are the main tools besides the compiler. Otherwise, it's mostly relying on existing tools for the respective platforms. We can reuse all the profiling tools, debuggers, so on and so forth, to work with Kotlin programs as if they were JVM programs or JS programs or whatever.

[00:29:28] JM: The project, you've been involved in since the beginning, and I've only heard like massive growth of Kotlin users. Occasionally, I'll send out a message on Twitter or in Slack and I'll say, "Hey, what kinds of shows do you want to hear about?" I swear, it's like every time people want to hear about Kotlin.

Kotlin is clearly growing in popularity. What's been the biggest challenge that you faced since the beginning of the project?

[00:29:57] AB: Well, in terms of popularity, really the challenge is – I guess, it's just staying in touch with the users, because we are paying a lot of attention to the feedback that users provide. Well, when the number of users doubles every three months, it's quite an effort to keep up with all the reports that we have. Our Slack channel has grown tremendously over the last year. Still, we're trying to listen to everybody as much as we can. Of course, the team cannot answer everybody's questions anymore. But the community is growing, so people are answering each other's questions very well, and I'm very grateful to people being supporting and helpful on the public channels.

I think keeping in the feedback loop was the biggest challenge related to growth. Then from a technical standpoint, there is a huge number of challenges, but they are not really related with the number of users we have. There is also, I think, the issue of diversity, because we now have users with very different backgrounds, users doing server side, mobile clients, web clients, so on and so forth. They all have different needs, and we have to balance all those needs and consider use cases from very different ecosystems. So we don't have all the expertise in the room anymore. We have to consult people outside the design team to make sure that what we come up with works for everybody. That's now the challenge.

[SPONSOR MESSAGE]

[00:31:24] JM: NNobody becomes a developer to solve bugs. We like to develop software because we like to be creative. We like to build new things, but debugging is an unavoidable part of most developers' lives. So you might as well do it as best as you can. You might as well debug as efficiently as you can. Now you can drastically cut the time that it takes you to debug.

Rookout rapid production debugging allows developers to track down issues in production without any additional coding. Any redeployment, you don't have to restart your app. Classic debuggers can be difficult to set up, and with the debugger, you often aren't testing the code in a production environment. You're testing it on your own machine or in a staging server.

Rookout lets you debug issues as they are occurring in production. Rookout is modern debugging. You can insert Rookout non-breaking breakpoints to immediately collect any piece of data from your live code and pipeline it anywhere. Even if you never thought about it before or you didn't create instrumentation to collect it, you can insert these nonbreaking breakpoints on the fly.

Go to rook out.com/sedaily to start a free trial and see how Rookout works. See how much debugging time you can save with this futuristic debugging tool. Rookout integrates with modern tools like Slack, Datadog, Sentry and New Relic.

Try the debugger of the future, try Rookout at @rookout.com/sedaily. That's R-O-O-K-O-U-T.com/sedaily. Thanks to Rookout for being a new sponsor of Software Engineering Daily.

[INTERVIEW CONTINUED]

[00:33:29] JM: I did a show recently with somebody about Kubernetes, and one of the things he was saying was that in order to reduce the scope, because Kubernetes has such a big surface area of different people that want to use it for radically different applications, and it sounds like with Kotlin, it's kind of the same thing. Do you have to scope your concerns to something narrow that you can actually accomplish? For example, in the early days of Kubernetes, when it was starting to catch on and people really liked it, there were all these different use cases and they were like, "Okay, we need to focus." So we're going to just focus on scenarios with three nodes,

three or more nodes. Let's just get that to baseline stability and then maybe we'll do something like multiple regions and we'll make sure that we can support everything within multiple regions. While we're focused on those things, maybe we won't p ay as much attention to all of the other use cases that have plenty of constituents, but they're just a little bit lower order bits." Do you have to make tradeoff in the surface area of concerns that you're focused on?

[00:34:35] AB: Well, time-wise, we do. We write, because we can't work on all issues at the same time. So we have to pick what we're working on today and work on something else tomorrow. In terms of fixing bugs, we can actually do a round robin, like, "Oh, after the bugs in this area and that area and work in stability issues across the entire language during a release cycle."

In terms of language features, you can't cater to everybody in a single release. So we focus most of our efforts on a single thing that seems to be helping as many people as we can. Then next time we focus on something else.

[00:35:13] JM: Are there any features that you wish were in Kotlin today but are currently not? Things that are perhaps on the roadmap?

[00:35:20] AB: I think everything I wish we had is on the roadmap, but some things are quite distant on the roadmap. For example, I would really love to have immutable data in Kotlin as first class describable in the type system, but currently we were not close to that. Immutable data is very important. Well, for philosophical reasons that also is a functional feature in a sense.

For practical reasons, it's very important for multi-threaded scenarios, because immutable objects can be shared without any precautions. It's basically safe to access them from different threads. It would be a great compliment to our core team story. Kotlin introduced core teams about two years ago, and we actually have a very good story there. Asynchronous programing is getting a lot easier with this feature. But it's still running over a multi-threaded environment.

Well-behaved programs can avoid any issues with sharing of mutable state, because [inaudible 00:36:24] make sure that safe publication happens on every passing from core team to another.

But still, you can have a global or something like this shared inadvertently and have issues there. I would really love to have immutable data. We're making interesting steps in this direction in Kotlin native. So it will be our experiment with the memory model. We'll see how it works out.

Another thing would be to support data science and big data use cases much better than we do now. There we have some actual plans. So it will be hopefully addressed in the next couple of years. Yeah, we have pretty many plans and these are just things that are easily explainable in a minute, then there are slightly more involved things about the type system that probably not be able to explain well at a blackboard.

[00:37:14] JM: Let's go through some of the ones you did mentioned. Immutable data, I thought Java objects were immutable by default. What do you mean by immutable data?

[00:37:23] AB: Java objects are not immutable by default. Java objects can have mutable fields. So if you have a normal field in an object, anywhere in Java, you can mutate it at any point and the access by default not synchronized in any way. You have to take care of synchronization and publication of such changes from one thread to another. On one hand, it enables a lot of use cases with high performance applications. On the other hand, it requires a lot of skill, and it's very difficult to debug and profile and so on and so forth.

So multithreaded programming in Java is a very difficult art. There are techniques to make it easier. One of them is deliberately make all your data immutable, which is not always feasible and not always easy, because the language itself neither Java nor Kotlin supports it fully as a first class citizen as Kotlin promotes immutability. Many thing in Kotlin are immutable just because this is the convention to make them. But we don't have any features in the language that help you track what's mutable and what's not. Once we design all these and make immutability a feature of the language, it will be a lot easier to make a safe multithreaded programs works smoothly without any issues.

[00:38:41] JM: The data science topic as well I thought was interesting. You mentioned that you'd like to make Kotlin support data science workloads better. Why doesn't it support data science workloads today?

[00:38:52] AB: In terms of workloads, I think it support them. But question is how can convenient the language is and how familiar the APIs would be for data science scenarios. There is quite a lot of existing libraries that are used by data scientists, and they're using mathematical and other language constructs. They're not available in Kotlin at the moment. Some example are creating collections in certain ways using slicing syntax for arrays and lists, so on and so forth.

There are seemingly small things in the language that can make the data science experience much better. So you can now do data science in Kotlin, and some people do it, and there are libraries and so on and so forth. To attract many people from this field, we have to make it a lot more accessible, a lot more a familiar. It's largely a question of familiarity and largely a question of the power of the DSL story in Kotlin. So basically want to enable the domain-specific language making in Kotlin for data science like scenarios.

[00:39:55] JM: You touched on Kotlin native. That's something we have not covered. Can you explain what Kotlin native is?

[00:40:01] AB: Kotlin native is one of our compiler backends, basically. It's a part of the Kotlin project that allows you to compile Kotlin sources to native binaries on different platforms, and it supports Linux, MacOS, Windows, iPhone, so on and so forth. Basically, it's a pretty small language runtime and the compiler is using the LLVM toolchain that's used by modern C and C++ compilers to produce a native binary that's reasonably small and runs fast.

[00:40:31] JM: How does the performance of that binary compare to when it runs on – I guess it's just totally a different story than the JVM, because in JVM you've got all these overhead of garbage collection and like hotspot runtime management, and I guess you just don't have that if you do the web assembly route.

[00:40:50] AB: Yeah. If you just go over the native binary, you don't have – First of all, you don't have the startup cost of the JVM. So JVM applications can be very, very fast when they have warmed up. But at the very beginning, they have to startup. The entire hotspot infrastructure

have to startup and this makes them really slow in the very beginning. A native binary just runs instantly. That's one thing.

In the native binary, you have less of the memory overhead theoretically. It also depends on the workload very much. Like in server workloads, there are many scenarios where dynamic optimizations of a hotspot-like VM can be a lot more efficient than any ahead of time compile program. But other scenarios ahead of time will be faster.

So it's largely a tradeoff, but the important part is that a native binary is runnable, where the JVM is not available. In an iPhone, for example, there's just no JVM at all that would perform reasonably well for a real application to run.

I would say the main thing about Kotlin native is availability on different platforms. The startup time is definitely better than a heavy VM. Then the runtime performance, currently we have an experimental compiler that is not doing many optimizations that it will do one day. Currently, I think we are slower than many other compilers. The code that we generate is slower than many other production compiler generate. But this is just a matter of work. Overtime, it will get a lot faster than it is now.

[00:42:23] JM: WE did a show a while ago about a topic called GraalVM. Have you heard of GraalVM?

[00:42:29] AB: Oh, yeah. I did.

[00:42:30] JM: I think GraalVM is some additions to the – If I recall, it's additions to the JVM that make it run more effectively by doing some improvements around object management, and I don't recall many other details right now. But have you looked seriously at it? Do you know how it might impact Kotlin?

[00:42:51] AB: I haven't looked too closely at it, but I think I have a general idea of what it is. So Graal can run a lot for Java programs and JVM programs. So it can run a lot of Kotlin programs as well. Some cases involve making them faster, which is great. It's also good at compiling things ahead of time. So if people care about startup time, it's another way of getting native

binary from a Kotlin program. It's somewhat different from how Kotlin native does that. But still it's available there. So I think it's a great addition to the JVM ecosystem, and there are many interesting use cases for it.

[00:43:25] JM: As we begin to wrap up, I guess it'd like to get a perspective on the development of Kotlin. How big is the dedicated Kotlin team at JetBrains, or is there a team?

[00:43:37] AB: Yeah, there is.

[00:43:37] JM: There is a team. Okay. How big is that team? How does it interact with the open source community?

[00:43:42] AB: We have about 50 people working at Kotlin at the moment. It's fulltime developers and QA people and marketing, so on and so forth, but mostly developers. We have an open source on GitHub. We get quite a few contributions every month. Every release we publish has some of the poll requests integrated. The community is helping and we get a lot of feedback from our users through our issue tracker or other public channels. I think our interaction with the open source community has been very, very helpful over this time.

[00:44:16] JM: Does JetBrains see a business opportunity in the Kotlin investment, or is it unclear at this point?

[00:44:24] AB: I would say it's pretty clear. I mean, some of the opportunities are pretty clear and already working out. So Kotlin was initially thought of as – First of all, of course, a boost to our own productivity. Many projects in JetBrains are now written in Kotlin. Some of them even written in Kotlin from scratch. This is definitely an enabler for many things that we're doing now, which is definitely a business benefit.

Also, Kotlin is a huge boost to the brand awareness. So many more people know JetBrains now because of Kotlin, and it's also working great for selling our commercial products to people in many interfaces. These are like the obvious things that we indirectly monetize Kotlin through. Then we'll be working on commercial tooling for Kotlin, and Kotlin native already has some early access versions of commercial tooling for it.

So, yeah, it's our business to help programmer be more productive through great IDs, and Kotlin is just another market that we basically created ourselves. Now we have this market to cater to and we can sell our software there. That's it. The language itself is open source and the basic tooling for it is open source, and we'll always be open source. Then there are so many added value, like things that we can provide in the market and make money on it.

[00:45:41] JM: Okay. Well, Andrey, are there any future plans and development for Kotlin that you want to mention?

[00:45:47] AB: There are many plans. I have mentioned some already. We'll be releasing 1.3 soon enough with core teams and the new version of multiplatform project support and it will be quite an exciting release. Just stay tuned for it. Then we'll be working on many more things, including making the ecosystem more uniform with the better compiler API. So we want different platforms to have same access to the Kotlin compiler and leverage things like transforming Kotlin code for tooling purposes. There will be quite a few things happening over the next year or two.

[00:46:20] JM: Sounds good. Well, Andrey Breslav, thank you for coming on Software Engineering Daily. It's been great talking to you.

[00:46:24] AB: Thank you very much. Great questions.

[END Of INTERVIEW]

resources for your application. There are also CPU optimized droplets, perfect for highly active frontend servers or CICD workloads, and running on the cloud can get expensive, which is why DigitalOcean makes it easy to choose the right size instance. The prices on standard instances have gone down too. You can check out all their new deals by going to do.co/sedaily, and as a bonus to our listeners, you will get $100 in credit to use over 60 days. That's a lot of money to experiment with. You can make a hundred dollars go pretty far on DigitalOcean. You can use the credit for hosting, or infrastructure, and that includes load balancers, object storage. DigitalOcean Spaces is a great new product that provides object storage, of course, computation.

Get your free $100 credit at do.co/sedaily, and thanks to DigitalOcean for being a sponsor. The cofounder of DigitalOcean, Moisey Uretsky, was one of the first people I interviewed, and his interview was really inspirational for me. So I've always thought of DigitalOcean as a pretty inspirational company. So thank you, DigitalOcean.

[END]