Editor: **Robert Blumen**
SalesForce
robert@robertblumen.com

Nicolai Parlog on Java 9 Modules

Nate Black

From the Editor

In Episode 316 of Software Engineering Radio, host Nate Black talks with guest Nicolai Parlog (author of the forthcoming book *The Java Module System*) about Java's evolution, emphasizing the latest release, version 9. The largest chunk covers the most significant new feature: modules. Nate and Nicolai also review the major changes from earlier versions of Java and speculate about Java's future. The excerpt here covers the why and how of the module system. To hear the full interview, visit www.se-radio.net or access our archives via RSS at feeds.feedburner.com/se-radio. —*Robert Blumen*

Nate Black: The big change in Java 9 is the introduction of modules. How do you explain modules to people who know Java but are new to the concept?

Nicolai Parlog: When I think about code, I have a huge graph in my head. I have a class. What does this class do? It usually calls other classes. In my head this class is a bubble. Other classes are also bubbles, and then there are arrows between them where they call each other. In computer science you would call [this a "graph"], the classes "nodes," and the edges "arrows."

But [this is] not only for classes. You can go lower. You can say, "I have the same thing for methods, because methods call each other." You can go the other way as well—to packages and JARs [Java Archives]. [A JAR is a ZIP file that contains the class files that are the combined bytecode of the Java classes.]

The graph also looks different at compile time and at runtime. But it is an idea that many people have: How does the code relate? How does one thing call another thing?

Each thing in the graph has properties, like a name. Methods have a name; classes have a class name; JARs have a JAR name; packages have a package name. They have dependencies. For example, methods make other method calls. But they all use each other.

And there is a third property. Each of these bubbles has something that I need. I wouldn't call another method just for the fun of it. Usually, it does something that I need. Beyond name and dependency, it also has something that I want to use—let's call it an API. These three things exist on all these levels. But let's stick to methods, classes, and JARs.

For methods and classes, the JVM [Java virtual machine] shares our understanding. The JVM says, "Yeah, it's a class, it has a class name, and it has an API, which are the public methods. And it has dependencies." You can scan the bytecode to find what other classes it uses. On the level of classes and methods, the JVM sees things like we do.

But on the JAR level, that's not the case anymore. You cannot say,

0740-7459/18/$33.00 © 2018 IEEE

SOFTWARE ENGINEERING RADIO

Visit www.se-radio.net to listen to these and other insightful hour-long podcasts.

RECENT EPISODES

- 318—Host Felienne interviews Veronika Cheplygina on image recognition.
- 317—Host Kishore Bhatia talks with Travis Kimmel on measuring engineering productivity.
- 314—Scott Piper and host Kim Carter discuss cloud security.

UPCOMING EPISODES

- Nicole Hubbard reports on a large migration from virtual-machine scale sets to Kubernetes.
- Nate Tagart discusses serverless tooling and operations.
- Maria Colgan talks about cost-based database query optimization.

"This JAR depends on that other JAR, and I only want to launch the program if that other JAR is there." JARs don't have names. For example, when you track down complicated runtime errors, you sometimes see a stack trace and wonder, "Which JAR is this class in again?" It was just loaded from a JAR. But you don't know which one.

It's easy to see why JARs are so useless: because a JAR is just a container. It has no identity. This causes all kinds of problems that we've gotten used to. We use Maven or Gradle to provide the dependencies because we have no other way to find out at runtime whether everything is there. But for public APIs, we don't have a good solution.

At runtime, the JVM collects all of the JARs and puts them into one big ball of mud. Whatever we had in our mind about one JAR using another JAR's API, that's all fiction. [At runtime, a Java program is] just a bunch of classes running in the same environment. Everything that is public is fair game. Even if, as a library developer, I said, "This package is internal," the JVM doesn't care. It can call whatever it wants. That's the situation we're in. Java 9 takes JARs and says, "Look, you now have an identity that the JVM understands."

There are a few terms floating around. We have JPMS (Java Platform Module System), Java modules, and Project Jigsaw. Could you please briefly explain?

Java is developed in projects. When Mark Reinhold back in 2008 said he wanted to have modules, he created Project Jigsaw. Whenever JDK [Java SE Development Kit] teams work on something new, they create a project. Project Jigsaw had the goal to provide a specification and an implementation of a module system that was geared toward the JDK but also usable by user code. That is Project Jigsaw. It produced the Java Platform Module System, although almost nobody calls it that because it's a mouthful. Usually people just say "modules."

For programmers, what are modules' benefits?

[In Java development], there's something known as "JAR hell." We have gotten used to it, but it means that the JVM doesn't understand dependencies, particularly transitive dependencies, so [you] have to hunt them down manually. Of course, we built great tools to solve that, but still, it's a shortcoming of the JVM. Something could be missing at runtime, for example, and you wouldn't find out until it's too late.

You can also have version conflicts. It can happen that you have two versions of the same library that you absolutely have to use because of transitive dependencies. One of your immediate dependencies uses, let's say, Guava 19, and the other one uses Guava 14, and there is no way they can both run on the same version [of Guava]. That's the problem.

The other issue is that we had no encapsulation across JARs. As I said earlier, every public type is free to be used by everyone. The JDK itself contains some security-relevant code, but not everybody should be calling that. They put in the security manager, which you have to activate. If you do so, then the security manager is on critical code paths and checks whether this access is allowed.

The problem with that is that it's a manual process. You cannot automatically put in all the places because if [the security manager is] in a hot loop, you couldn't make that check every time. It has to be put into the right places in order to not impact performance too much. Even

then, anecdotes seem to suggest that turning the security manager on means 10 to 15 percent less performance. You must put it in the right places so as not to make that 15 percent into 50 percent. But that means it's a manual process.

When Java 8 was delayed due to security problems, two of the five major security breaches that Java had were missing security manager calls. The reason for that is that Java does not understand that this code is not supposed to be called by the user—that this code is only meant to be called by other parts of the JDK. Because the separation on the level of libraries does not exist in the JDK, every call is indistinguishable to the JDK. You would have had to put in manual checks to distinguish whether or not a call is coming from code that's allowed to make that call. This level of manual security was a problem.

Last but not least, Java is a monolith. [Before Java 9], you had just one Java runtime: it's all or nothing. And it was rather big. In the meantime, memory got so much larger, but with Docker images and other virtualization, the idea of a smaller runtime containing only the stuff that I need has come back. Why would I want an 80-Mbyte runtime, half of which is probably [GUI and windowing libraries] that I never use because I am writing back end? Why have that in the default runtime? Why not have a runtime that can be split apart?

Class path hell, no encapsulation, the security problem, and the rigid Java runtime: those were the big problems that the module system could tackle.

I'd like to understand how modules and Java 9 address some of those problems. But first, let's talk at a high level about how modules work.

There's a one-to-one relationship between JARs and modules. What is this module's name? What other modules does it need? And what is its API? (The API part is not that important at the moment because we've largely talked about dependencies.)

The JDK itself got split up into about 100 modules. Around 20 or 30 of them are publicly supported and standardized platform modules.

The module system can make sure that all transitive dependencies are present. It will not let you launch otherwise. It understands the dependency graph of JARs. If you are missing a dependency, even though it's not a direct dependency, it tells you which one. That solves the first part of the problem.

I invented a new term: "launch time." Technically it's during runtime, but it's at the beginning of runtime. Even if the first time some service runs is an hour into the program run, it won't take an hour to realize that something is missing and then crash.

In Java 9 there is a module path, which is like the class path, but for modules. For example, when you have Guava 14 and 19 on the module path, then it will not launch. It will say, "You have the same thing twice."

The module system does not understand versions and does not help you with the version conflict thing. It enables you to find out at launch time that you have a problem, but it does not provide a solution.

Another problem occurs when there is a fork in a package. One says, "I'm Guava," and the other says, "I'm whatever the other fork is called," so they're not the same modules. But they still contain the same packages. This is a so-called split-package problem. The module system ensures that each package is only contained in one module. You cannot have Guava and its fork on the module path as long as they contain the same packages. You get an error then as well.

What does it look like to use the module system? Does it change how people write code? Is it a change in tooling? What does it look like at the implementation level to use modules?

A module is just a JAR with a module descriptor. It's a regular JAR. You can even compile and run it on Java 8. The additional file is called module-info.class. It's compiled from module-info.java, which contains the module declaration. The compiler sees it and thinks "Aha, we're doing a module here." Then it expects the module things to be in place. [It's the] same at runtime. If you have a JAR with that module descriptor, and you put it onto the module path, then the JVM ensures that you want to have the other modules in place.

The developer creates the module descriptor. The file contains `module . . . <your module name>`, which must be a regular Java identifier. It is recommended that the module name is the same as the package. Then [come] curly braces, and then come two blocks: `requires` and `exports`. `Requires` is a list of the modules that your module needs.

And then come the `exports`. You export the packages that contain your API. You list only the packages that you intend to support. The other packages that are internal—the module system guards these parts. By not exporting, you are making a statement that "this is not a supported API." And you are telling the JVM, "Don't let people use this."

ABOUT THE AUTHOR

NATE BLACK is a software engineer at Sleeperbot. Contact him at nathanael.black@gmail.com.

And then you're done. Then you've done all you needed to create your module.

To summarize, there's no change to the Java code that I'm writing. There's an additional small file called module-info.java that contains a list of all my dependencies and my exported API. I'm explicitly saying what my supported API is.

Yes. [In a project] you may have the problem that you create a JAR and you have a small subproject with a dozen packages, where you created two or three of them to be publicly used, but the others, you didn't consider them to be the API.

When you are doing a big project with dozens of modules, and you wrote one of them a couple of years ago, before Java 9, other developers would use that package—they have no incentive not to. There was no step in the process to question whether or not to do that. The IDE asks, "Do you want to auto-import that?" and I'm, like, "Sure, I want to"—and it's done. In my personal experience, nobody really looks at import clauses during code reviews. There are so many of them that nobody really bothers going through them.

[With Java 9], if I start using your old JAR, I have to add an export. And then, at code review, somebody can say "Are you sure?" This means that we're going to think much more about public APIs.

Is this a change in thinking that's on a par with generics and lambdas in terms of its effect on how people write code?

I think no. In day-to-day programming, it will show up much less. But that doesn't mean it has less of a long-term effect. Many of the problems that slow down projects and eventually cause them to fail are not that the classes got too wordy, which lambda fixes. The long-term problems are often that the development speed got so slow because everybody was doing everything that was allowed to be done. You end up with a big ball of mud of JAR references, with no oversight over dependencies or APIs.

And the module system is a tool that helps you to keep in mind that you don't want to do that.

IEEE Software (ISSN 0740-7459) is published bimonthly by the IEEE Computer Society. IEEE headquarters: Three Park Ave., 17th Floor, New York, NY 10016-5997. IEEE Computer Society Publications Office: 10662 Los Vaqueros Cir., Los Alamitos, CA 90720; +1 714 821 8380; fax +1 714 821 4010. IEEE Computer Society headquarters: 2001 L St., Ste. 700, Washington, DC 20036. Subscribe to *IEEE Software* by visiting www.computer.org/software.

Postmaster: Send undelivered copies and address changes to *IEEE Software*, Membership Processing Dept., IEEE Service Center, 445 Hoes Lane, Piscataway, NJ 08854-4141. Periodicals Postage Paid at New York, NY, and at additional mailing offices. Canadian GST #125634188. Canada Post Publications Mail Agreement Number 40013885. Return undeliverable Canadian addresses to PO Box 122, Niagara Falls, ON L2E 6S8, Canada. Printed in the USA.