Java 9, 10 and Beyond:

The Ultimate Guide to the Future of the Java Platform

By Tali Soroker

Table of Contents

The Java Platform is on the Brink of a Big Change

The last few Java upgrades brought us features that the community has been arguing over, denouncing and requesting for years. Java 8 introduced us to the world of Lambdas, bringing Java to the forefront of functional programming, and Java 9 was the first modular version of the JDK with the completion and introduction of Project Jigsaw.

With the release of Java 9, Reinhold and the rest of the Oracle team made another big announcement. This time, it wasn't about the "next big feature" coming to Java, it was about WHEN the next big feature will be coming to Java. More to the point, they announced a new 6-month release cycle for the Java platform.

This means that before the dust had even settled from Java 9's highly-anticipated release, Java 10 was already on its way to its release in March (join the countdown to Java 11 here). Before we get into the future features of Java, we'll cover some of the major changes and "under the hood" API updates we saw in Java 9 and quickly touch on the current state of modern software reliability.

The Elephant in the JDK:

Changes to Future of Java That We Can't Ignore

Project Jigsaw

Project Jigsaw aims to make Java modular and break the JRE to interoperable components. This means that you'll be able to create a scaled down runtime Jar (rt.jar) customized to the components a project actually needs.

This project aims to make Java scalable to small computing devices, improve security and performance, and mainly make it easier for developers to construct and maintain libraries.

Project Amber

Project Amber was first introduced last January when Brian Goetz, Java Language Architect, proposed creating a project for exploring and incubating smaller, productivity-oriented Java language features.

The main prerequisite for features that'll be a part of this project: they have been accepted as candidate JEPs, also known as JDK Enhancement Proposal. It's the process in which Oracle collects proposals for enhancements to the Java

Development Kit and OpenJDK. Approved proposals continue down the road to become actual features in Java.

On March 16th, 2017, Goetz welcomed Project Amber into the Java community, along with the first three Java Enhancement Proposals adopted by it: **Enhanced Enums** (JEP 301), **Lambda Leftovers** (JEP 302) and **Local Variable Type Inference** (JEP 286), which after almost exactly one year as a part of Project Amber, will be introduced as a part of Java 10 in March 2018 (more on this in Part 4!).

Incubator Modules (HTTP 2.0)

A new JDK Enhancement Proposal (aka JEP 11) introduces playground incubator modules that target experimental features to either be adopted and standardized in the next version or Java, or to otherwise be removed quietly from the JDK. The first feature to be introduced as a part of this project is Java's HTTP/2 Client.

While originally slated for complete integration in Java 9, the HTTP/2 Client API was introduced as the first incubator module. The three main implications of the HTTP/2 Client being implemented as an incubator module are:

1. It won't be included in the SE Platform
2. It will be in the jdk.incubator namespace rather than java.net
3. It won't resolve by default at runtime (this is true for all incubator modules)

Noteworthy API Updates and Changes to the Java Platform

CompletableFuture & Stack-Walking API

The first thing we want to mention here are the Java 9 concurrency updates with CompletableFuture and java.util.concurrent.Flow. Flow is the Java implementation of the Reactive Streams API, and we're pretty excited that it's now a part of Java. Reactive Streams solve the pain of back-pressure, the build-up of data that happens when the incoming tasks rate is higher than the application's ability to process them. This results in a buffer of unhandled data.

As part of the concurrency updates, CompletableFuture will also get an update that will resolve complaints that came in after their introduction in Java 8. This will include support for delays and timeouts, better support for subclassing, and a few utility methods.

The second thing we wanted to mention here is the Stack-Walking API. That's right, Java 9 will change the way you traverse stack traces. This is basically an official Java way to process stack traces, rather than simply treating them as plain text.

Process API

So far there has been a limited ability for controlling and managing operating system processes with Java. For example, in order to do something as simple as get your process PID in earlier versions of Java, you would need to either access native code or use some sort of a magical workaround. Moreover, it would require a different implementation for each platform to guarantee you're getting the right result.

Before Java 9, the code for retrieving Linux PIDs, looked like this:

```
1  public static void main(String[] args) throws Exception {
2      Process proc = Runtime.getRuntime().exec(new String[] {
3        "/bin/sh",
4         "-c",
5         "echo $PPID"
6      });
7
8      if (proc.waitFor() == 0) {
9        InputStream in = proc.getInputStream();
10       int available = in.available();
11       byte[] outputBytes = new byte[available];
12
13       in.read(outputBytes);
14       String pid = new String(outputBytes);
15
16       System.out.println("Your pid is " + pid);
17     }
18 }
```

Now, it looks like:

```
1  System.out.println("Your pid is " + Process.getCurrentPid());
```

The update will extend Java's ability to to interact with the operating system: New direct methods to handle PIDs, process names and states, and ability to enumerate

JVMs and processes and more. This might not be too exciting for some, but we at OverOps will make an extensive use of it so this is why we chose to highlight it among the other features.

Unsafe

sun.misc.Unsafe has been one of the key APIs in the JVM since Java 6, but it was intended to be only used by core Java classes, and not the developers themselves. The library itself is a collection of methods for performing low-level, unsafe operations. We know, the title gives it away.

To make a long story short, numerous libraries started using Unsafe, and it became a key library for many projects. Because it was never intended for developer use and it's, well, unsafe, Oracle planned on removing it altogether. As you can imagine, the Java community didn't take it so well and protested against this decision.

It took a while, but eventually the community won – and Unsafe was encapsulated in Java 9. To make it safe, the functionality of many of the methods in this class will be available via variable handles.

G1 GC Default

Java 9 comes with Garbage First Garbage Collector (G1 GC) as its default garbage collector (although we can still switch it if we really want to).

G1 is a server-style garbage collector, designed for multi-processor machines with large memories. With G1, the heap is partitioned into a set of equal-sized heap regions, each a contiguous range of virtual memory. It supports heaps larger than 4GB and is a parallel, concurrent, and incrementally compacting low-pause garbage collector.

Part 3

Modern Software Reliability

While not specific to the new version of Java, we've seen that over the past few years, nearly every organization has undergone a change in the way they deliver software to production, just like Oracle has with their decision to switch to a 6-month release cycle. The introduction of new platforms and tools has improved efficiency, allowing us to innovate faster, collaborate more easily and most importantly remove human error and obstacles… all through automation.

This change has happened because organizations are required to innovate at an accelerated pace. We need to be more agile, but we also risk introducing more error prone code into production. The challenge here is trying to balance speed and reliability, which are often diametrically opposed. So, how do we balance this more effectively?

While the rest of the software delivery and supply chain has been reinvented, companies still rely on age old technology to debug and troubleshoot our applications - log files. The fact is, though, log files simply won't show you the root of the issue. They might not even be pointing you in the right direction. With Automated Root Cause (ARC) analysis, OverOps captures all known and unknown errors, 100% of exceptions, whether they were caught, uncaught, logged or not, and shows the complete source code and variable state that caused them.

To see how it works in production and pre-production environments, try it out for yourself. It only takes a few minutes to install and analyze your first exception.

The Next Big Thing

Java 10's big claim to fame is Local Variable Type Inference. It extends type inference to declarations of local variables and initializers, and Java 10 now includes the identifier var that allows for defining and initializing local variables.

In other words, you can now declare variables without having to specify the associated type. A declaration such as:

```
1  List<string> list = new ArrayList<String>();
2  Stream<tring> stream = getStream();
```

Will be replaced with this new, simplified syntax:

```
1  var list = new ArrayList<string>();
2  var stream = list.stream();
```

Use of the feature is restricted to:

• Local variables with initializers

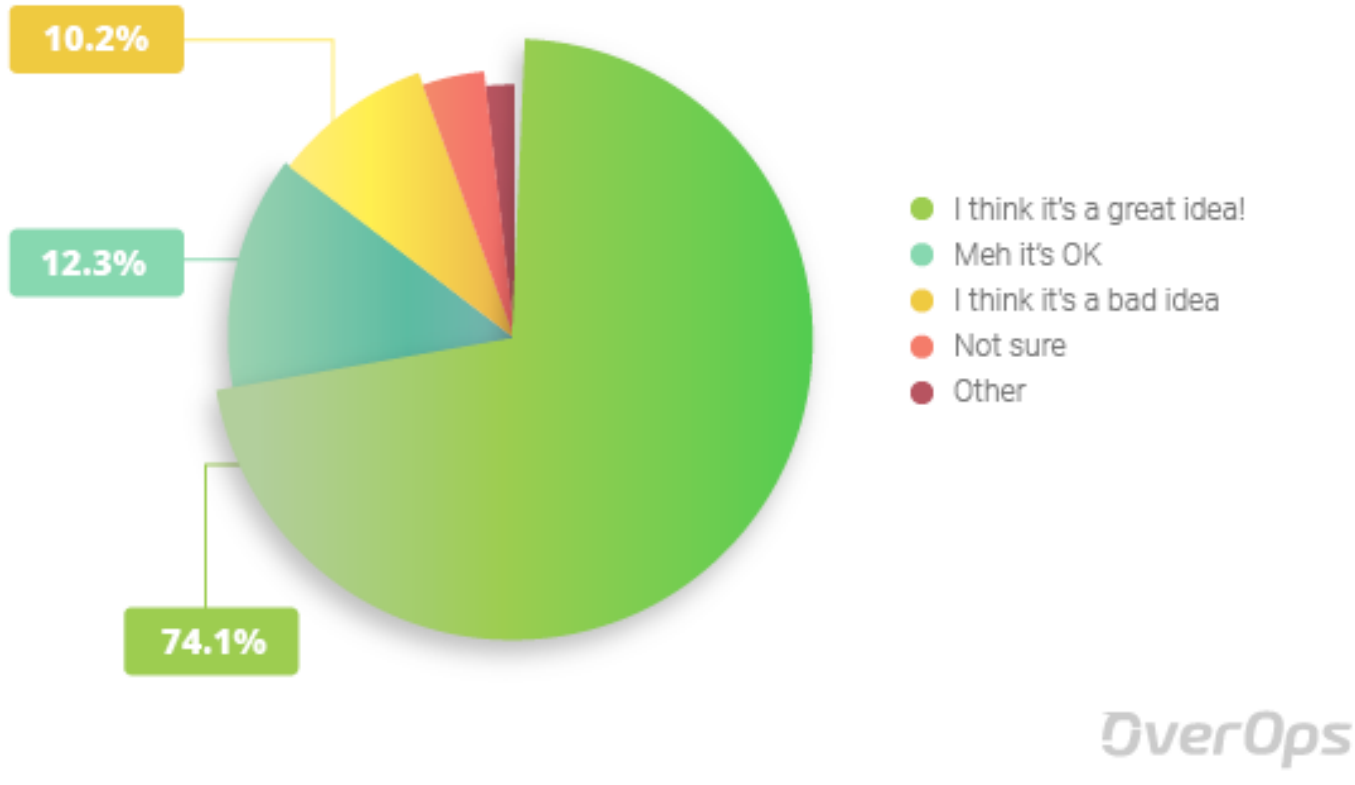
    • Indexes in the enhanced for-loop

    • Locals declared in a traditional for-loop

Plus, it is not be available for:

- Method parameters
- Constructor parameters
- Method return types
- Fields
- Catch formals (or any other kind of variable declaration)

Results from a questionnaire that was published during the planning stage for the feature showed strong support, with 74.1% of the close to 2,500 respondents answering that they “think it’s a great idea!” Initially, the plan was to include both var and val commands, but in the end the team decided on var only.



Meanwhile, 4 JEPs have already been slated for the release of JDK 11 in September 2018. This release will be one to pay attention to, as it will mark the start of a new 3-year cycle of Long-Term Support releases. This release and others like it will have available updates for at least 3 years and possibly longer, compared to releases like JDK 9 and 10 which will only see 2 quarterly updates for security and bugs.

After introducing Local Variable Type Inference, the Oracle Team will release JEP 323 ("Local-Variable Syntax for Lambda Parameters") in JDK 11 to "allow var to be used when declaring the formal parameters of implicitly typed lambda expressions." The Team will also be working to remove the Java EE and CORBA Modules (JEP 320) from the Java SE Platform and the JDK after they were deprecated in Java SE 9 with plans for future removal.

Plus, one more upcoming feature we can expect in JDK 11 includes JEP 318 ("Epsilon: An Arbitrarily Low-Overhead Garbage Collector") which will "provide a completely passive GC implementation with a bounded allocation limit and the lowest latency overhead possible, at the expense of memory footprint and memory throughput."

“Java Must Move Forward Faster”

There’s an obvious difference between the changes that we saw to the platform with Java 9 and what we can expect from Java 10. And it’s not difficult to figure out that it’s, at least in part, due to the fact that Java 9 launched 3 and a half years after Java 8, while Java 10 will be here just 6 months after Java 9’s debut.

The move towards a time-driven model is modernizing the platform, allowing Java to evolve faster and with more flexibility. Development of complicated platform features doesn’t always go as planned and, in the past, has resulted in delays for the entire platform’s release. By changing the release cycle, developers can get their hands on smaller APIs and JVM features or updates regardless of the state of a project like Project Jigsaw.

Mark Reinhold explained, “for Java to remain competitive it must not just continue to move forward — **it must move forward faster**.”

And that’s what this new approach is all about – helping Java move faster, stay competitive in the programming landscape and still maintain its compatibility and reliability.

Just like Oracle, more companies today are moving towards accelerated workflows and more frequent releases. As development speeds up, though, it’s important to address the risks that this can pose to application reliability. OverOps’ Continuous Reliability platform helps teams identify and investigate errors throughout the development lifecycle and integrates with the full application monitoring ecosystem. Build new features faster without compromising on application performance and reliability. Try it out.