CODING

The role of preview features in Java 14, Java 15, Java 16, and beyond

How Oracle gathers feedback on new JDK functionality with preview, experimental, and incubating features

by David Delabassee

June 8, 2020

With millions of people relying on Java in production for critical workloads, the reach of Java is global. Given its depth and breadth, new features have to not only be designed in a clear and complete manner but also implemented in a reliable and maintainable manner. All the new features given to Java developers to use in production have to attain the highest possible quality. It is, hence, critical to provide developers with preliminary access to new features to encourage feedback: feedback that will help refine those features and reach the expected quality level for their final and permanent form.

There are several categories of new, nonfinal features:

1. Preview, for new Java platform features fully specified and implemented but yet subject to adjustments
2. Experimental, mainly for new features in the HotSpot JVM
3. Incubating (also known as incubator modules), for potentially new APIs and JDK tools

In addition, there are other nonfinal features that don't fit in any of those three categories. I'll discuss them later.

This article discusses how and where Oracle leverages preview features, experimental features, and incubator modules to gather feedback from the community before new features are made permanent features of the Java platform. **Figure 1** shows examples of the progression of some new features.

Figure 1. The evolution of three recent features

New features almost always start out as a JDK Enhancement Proposal (JEP), a well-defined mechanism to manage nontrivial JDK enhancements such as the following:

- A **Java language feature** for the *Java Language Specification*, such as text blocks or records.
- A **Java SE API feature** in the core Java platform, such as `java.lang.Object`, `java.lang.String`, or `java.io.File`. Such an API feature will reside in a module whose name starts with `java`.
- A **JDK API feature** with additional JDK-specific features, such as the JDK Flight Recorder. Such an API feature will reside in a module whose name starts with `jdk`.
- A **JDK tool feature** such as `jshell` or `jlink`.
- A **feature specific to HotSpot JVM**, the OpenJDK implementation of the Java Virtual Machine. Two such features are Application Class-Data Sharing and the Z Garbage Collector (ZGC).

You can see the relationships in **Figure 2**.

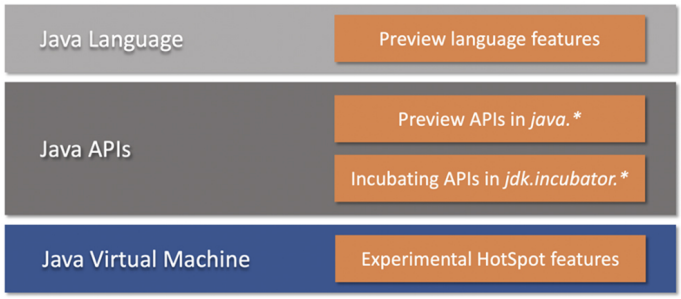

Figure 2. Where preview, experimental, and incubating features live

By the way, in the context of the JDK, the term *Java APIs* is often used to describe both Java SE APIs and JDK APIs.

What about importance? An enhancement is considered nontrivial if it is highly demanded, if it impacts the JDK or the processes and infrastructure by which the JDK itself is developed, or simply because it requires quite a bit of engineering investment. The JEP process is also used for deprecating features and improving existing features.

Most larger features are introduced using a two-phased approach that uses JEPs, starting with a preliminary access phase followed by an activation phase. There might be one or multiple iterations of the preliminary access phase during which developers are given access to new nonfinal features. During the preliminary access phase (or phases), developers are

encouraged to actively use and gain experience with nonfinal features in order to provide feedback.

If the provided feedback highlights some room for improvements, those can be addressed in the next iteration. This feedback might also be used to improve documentation such as a programmer's guide that often accompanies new language features, the Javadoc examples for a new API, or an FAQ.

Finally, when a new feature is deemed ready, a final phase will transition that new feature into a permanent one in the Java platform.

It is important to set expectations by qualifying what type of feedback the engineers working on new features expect to get. Those engineers are primarily interested in how developers are using a new feature, for example, whether there are technical or compatibility issues when using it, whether it integrates coherently with existing features, and whether there are any impediments to refactor code to use it.

Note: The engineers working on the new feature are not really looking for alternative ideas or potential solutions to semi-related issues. They are also not looking for someone's wish list tied to that feature. Why? Most of the time, such suggestions tend to propose limited short-term benefits while ignoring the global Java platform vision. (That's especially true when it comes to feedback from people who haven't even used the feature but want to offer objections or suggestions anyway, which is not helpful.)

Who should offer feedback? Well, the engineers welcome feedback from Java developers (such as about a new API) or tool vendors (such as about a new JDK tool). At the end, all constructive and actionable feedback is welcome.

By the way, it's important to use the requested channel for submitting feedback. Comments posted on social networks have little chance of being considered. That's why each JEP clearly designates a mailing list to collect feedback. For example, JEP 359 asks for feedback using the amber-dev mailing list.

In essence, the Java engineers are looking for real-world experience and actionable feedback on new features by making them accessible early in a nonfinal form and doing any necessary adjustments before those features are made final and permanent parts of the Java platform.

Preview features

Java language features and Java SE API features have a lot of exposure, and any mistake in their design can have negative consequences. To avoid such a risk, a specific JEP (JEP 12) offers the ability to preview new Java language and Java SE API features. A preview feature is one that is believed to be fully specified and implemented, but may still change before it is

included in the Java platform on a final and permanent basis. The gathered feedback will be evaluated and used to make eventual adjustments before a feature becomes permanent.

For example, Project Amber is an OpenJDK project whose goal is to improve developer productivity through evolution of the Java language. Amber is leveraging the preview feature mechanism to gradually deliver standard permanent features into the Java platform. We can observe that two preview rounds seem adequate to collect actionable feedback on new Amber features before making them permanent. (See **Figure 3**.)

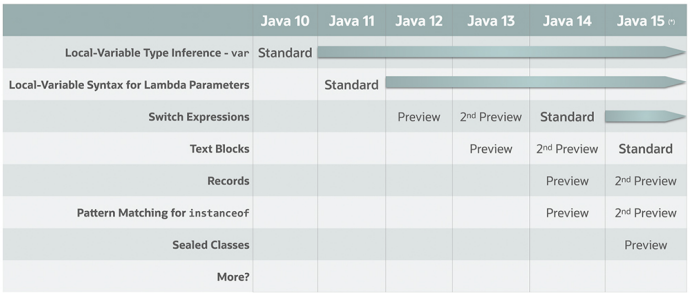

Figure 3. Project Amber features delivered as of Java 15

To dig deeper, consider the `switch` expression, which developed under the umbrella of Project Amber which was previewed in Java 12 (JEP 325) and Java 13 (JEP 354) before being turned into a standard language feature in Java 14 (JEP 361).

In Java 12, the `break` keyword was used to produce a value for a `switch` expression, for example, `break 42;`, but feedback suggested that this use of `break` was confusing. In response, the `yield` keyword was introduced in Java 13 to accomplish the same task, for example, `yield 42;`.

The final `switch` expression in Java 14 kept the `yield` approach previewed in Java 13. While a preview feature is intended to be very close to final, Oracle reserves the right to make changes between preview versions, such as changing from `break 42;` in Java 12's preview feature to `yield 42;` in Java 13's preview feature. Only the final version of the switch expression in Java 14, which kept `yield` from Java 13, is subject to long-term compatibility rules.

Experimental features

An experimental feature is a test-bed mechanism used to gather feedback on nontrivial HotSpot enhancements. Unlike JEP 12 for preview features, there is no JEP governing experimental features; the process for experimental features is more an established HotSpot convention than a formal process.

Let's take the Z Garbage Collector as an example. ZGC offers a low-latency garbage collection pause time—below 10 ms but typically more around 2 ms—regardless of the heap size, even if

the heap is as small as a few megabytes, or as large as multiple terabytes.

The ZGC team leveraged the experimental feature mechanism several times, with ZGC initially introduced in JDK 11 as an experimental feature limited to Linux x64 (JEP 333). Since then, additional improvements were added to ZGC (for example, concurrent class unloading, memory uncommit, and additional platforms) while other ZGC capabilities were ironed out.

The overall feedback and experience collected during those iterations enabled ZGC to be gradually solidified to a point where it now has the high quality expected for a HotSpot feature. Consequently, JEP 377 is now proposing to formally turn ZGC into a regular HotSpot production feature in JDK 15.

Incubator modules

JEP 11 introduces the notion of incubation to enable the inclusion of JDK APIs and JDK tools that might one day, after improvements and stabilizations, be included and supported in the Java SE platform or in the JDK. For example, the HTTP/2 Client API has been incubating—as a JDK-specific API in JDK 9 and JDK 10 via JEP 110—to finally leave that incubating phase and be included as a standard Java SE API in Java 11 (JEP 321).

Using nonfinal features

Important safeguards prevent developers from using nonfinal features accidentally. This is necessary because a nonfinal feature may well be different when it becomes final and permanent in a later Java feature release. Moreover, only final, permanent features are subject to Java's stringent backward-compatibility rules.

Therefore, to avoid unintentional use, preview and experimental features are disabled by default, and the JDK documentation unequivocally warns developers about the nonfinal nature of these features and any of their associated APIs.

Preview features are specific to a given Java SE feature release and require the use of special flags at compile time as well as at runtime. In a given Java SE platform release (for example, Java 14), `javac --enable-preview --release 14 …` will enable the Java compiler to generate class files that use preview features. The compiler will also issue a warning to inform developers that preview features are being used. And similarly, `java --enable-preview …` will allow those classes to run on a matching JVM (version 14, in this case), while `jshell --enable-preview` will enable the use of preview features on the corresponding `jshell` version.

Most IDEs support the use of preview features, which not only allows developers to use preview features in their favorite IDEs

but also help those IDEs to support those features shortly after they become permanent and final. For example, JDK 14 preview features can be enabled in IntelliJ IDEA 2020.1 by simply switching the “Project language level” from “14” to “14 (Preview) – Records, patterns, text blocks” in the “Project” and “Modules” settings.

By the way, artifacts that require nonfinal features shouldn’t be distributed. It would, for example, be a bad idea to distribute on Maven Central an artifact that leverages a preview feature because the artifact will run only on a specific Java feature release!

Experimental features are JVM features and are disabled by default; the `-XX:+UnlockExperimentalVMOptions` flag instructs HotSpot to allow experimental features. The actual experimental feature can then be enabled via specific flags, for example, `-XX:+UseZGC` for ZGC.

Finally, incubator modules are also shielded from accidental use because incubating can be done only in the `jdk.incubator` namespace. Therefore, an application on the classpath must use the `--add-modules` command-line option to explicitly request resolution for an incubating feature. Alternatively, a modular application must specify `requires` or `requires transitive` dependencies upon an incubating feature directly.

Psst: There are also early access builds

Loom, Panama, and Valhalla are examples of OpenJDK long-term projects. The goal of those projects is to conduct fundamental investigations in particular areas to drastically improve (or completely revamp) certain aspects of the Java platform. For example, Loom’s goal is to bring greater concurrency to the Java platform by making threads more lightweight and easier to use.

Given their ambitious scope, those projects will iteratively deliver, over the course of several Java feature releases, multiple features that together address the tackled area. To achieve this, various investigations will be conducted, different prototypes will be developed to experiment with potential solutions, and some approaches might be abandoned or reimagined.

As you would expect, this work takes time and requires considerable engineering efforts. Novel features developed under the auspices of those projects are unable to leverage the regular feedback mechanisms; because they are simply unfinished, they don’t have the expected stability. That does not mean that usage feedback would not be valuable. On the contrary, early feedback can potentially inform some of the design discussions and validate early prototypes.

To gather such feedback early on, sometimes there are specific early access builds for novel features while they are being designed and developed. The unique goal of such occasional feature-specific early access (EA) JDK builds is solely to allow expert users to test specific novel features early.

Given the highly skilled but limited target audience of EA builds, project leads have the ability to relax some rules (for example, in terms of compatibility) or impose constraints (for example, to allow some aspects of a novel feature to be partly missing). For instance, the first Loom EA build appeared in July 2019; the second EA build came six months later. This second build was, in the words of the project lead, “a drastic departure from the API in the first EA build,” demonstrating that early access features are not subject to any compatibility rule. This also reaffirms that an EA build should be used only by expert users for testing a novel feature within the scope of that particular EA build.

EA JDK builds are available from jdk.java.net. In addition to scoping an EA build, and documenting limitations and known issues, the download page also designates the appropriate mailing list for providing feedback. Over time, the feedback and experience gathered for novel features contributes to reshaping and refining them. Once a given feature has reached the expected stability and quality level, it is then able to leverage the regular mechanisms, such as JEPs with or without feedback mechanisms, with the ultimate goal of being made a permanent feature of the Java platform.

Conclusion

To avoid any costly design flaws for new features in the Java platform, in the JDK, or in HotSpot, Oracle uses multiple mechanisms—preview features, experimental features, and incubator modules—to give developers preliminary access to nonfinal features.

The feedback loop for nonfinal features, along with the biannual release cadence and the help of the Java community, contribute to reaching the high quality expected for any feature that is put in the hands of millions of Java developers to be used in production.

David Delabassee

David Delabassée is a developer advocate in the Java Platform Group at Oracle. Prior to that, he was involved in Oracle Serverless initiatives. He has also been heavily involved in Java EE 8 and its transition to the Eclipse Foundation as part of the Jakarta EE initiative.
Over the years, Delabassée has championed Java extensively throughout

the world, by presenting at conferences and user groups, large and small. He blogs at https://delabassee.com and has authored many technical articles for various publications.
Delabassée lives in Belgium. Follow him on Twitter @delabassee.

Share this Page