The Best of the JDK Face-Off

Twenty-five years of Java development saw many innovations in languages, libraries, tooling, and runtimes. Tell us what you think matters most.

by Sharat Chander

May 18, 2020

Happy 25th birthday, Java! It’s sometimes hard to grasp that it’s been a quarter century since the first release of Java was delivered on May 23, 1995.

In the early days, excitement was around the language (and the concepts behind the language), which was first envisioned with the names Oak, then Green, and finally Java. Yet here we are because of the amazing engineering talent whose collective vision was simple: acknowledging that the world was moving from a hardware-first to a software-first paradigm with a network delivery paradigm. That change opened an opportunity to create a programming language and development platform that would offer increased ease of use, reliability, security, and platform independence with each new release.

But it wasn’t just great engineering expertise that advanced Java innovation. In those early days, users were instrumental in helping continuously design Java so it would meet the demands of an ever-evolving technology landscape. “Focus on the developer!” was the battle cry. That may seem obvious now in the modern development world, but back in the 1990s, putting developers first was not the norm.

Today, Java’s technology and Java’s ecosystem offer a unique symbiotic relationship that drives innovation forward in an ongoing journey that treats both with care and focus on equal footing. And the result is continual productivity for you, the developer.

As the Java community celebrates this important Java milestone, the Java team and Java Magazine invite you to join us for some fun in the Best of the JDK Face-Off competition.

Obviously, Java has a rich history of delivering memorable and meaningful innovation. Everyone saw tremendous developer productivity in Java 8, and then an amazing collection of innovation in the releases from Java 9 all the way up through the current Java 14 release. But which are the most important advances? That’s the question of the day.

The Java team has aggregated a list of some of the most talked-about features (both small and large) over the last few years. Please vote on your favorites.
Don’t treat this lightly: Please do some soul-searching and think about which features really excited you, moved you to greater productivity, and advanced your Java programming expertise.

In the coming weeks, via the Java team’s Twitter @Java account, you will see matchups of key features laid out in four different brackets:

- Languages
- Libraries
- Tooling
- Runtime

You’ll have the opportunity to cast your vote via a Twitter poll that will run 24 hours for each posted matchup. Each progressive day will offer a new matchup until there are only two finalists to face off against each other. This competition will run socially over @Java through the next several weeks until there’s one feature left standing!

To get the wheels turning, Figure 1 shows the bracket board. As you’ll see, this isn’t easy.

![Figure 1. The JDK Face-Off bracket board](View full-size image)

Here are some questions to ponder as you think about the bracket board:

- Which feature do I pick?
- Which one matters more to me?
- Which one drives me to learn more?
- How will I decide which to vote for?

Are you ready? Let’s dig into the four brackets, to make sure everyone is on the same page. If you aren’t familiar with a particular feature (some are new for Java 14), there are recommended resources to bring you up to speed.

**Languages Bracket**

- Default methods: A default method is a method that is declared in an interface with the modifier default. Its body provides a default implementation for any class that implements the interface without overriding the method. This allows new functionality to be added to existing (and perhaps already widely distributed) interfaces. More generally, it provides a mechanism for multiple inheritance of behavior.

- Pattern matching for instance of: Pattern matching allows common logic in a program, namely the conditional extraction of components from objects, to be expressed more concisely and safely. This is a preview language feature in JDK 14. Learn more.

- Text blocks: A text block is a multiline string literal that avoids the need for most escape sequences, automatically formats a string in a predictable way, and gives the developer control over the format.
• Annotations on Java types: JEP 104 extends the set of annotatable locations in the syntax of the Java programming language to include names that indicate the use of a type as well as the declaration of a type.

• Local-variable type inference (var): JEP 286 enhances the Java language to extend type inference to declarations of local variables with initializers.

• Records: JEP 359 records provide a compact syntax for declaring classes that are transparent holders for shallowly immutable data. This is a preview language feature in JDK 14.

• Switch expressions: JEP 361 extends switch so it can be used as either a statement or an expression, and so that both forms can use either traditional case ... : labels (with fall-through) or new case ... -> labels (with no fall-through), with a further new statement for yielding a value from a switch expression. These changes simplify everyday coding and prepare the way for the use of pattern matching in switch.

• Method references: Method references use lambda expressions to create anonymous methods. Sometimes, however, a lambda expression does nothing but call an existing method. In those cases, it's often clearer to refer to the existing method by name. Method references enable you to do this; they are compact, easy-to-read lambda expressions for methods that already have a name.

Libraries Bracket

• Multi-resolution images: JEP 251 defines a multi-resolution image API so images with resolution variants can easily be manipulated and displayed.

• JVM constants: JEP 334 introduces an API to model nominal descriptions of key class-file and runtime artifacts, in particular constants that are loadable from the constant pool.

• Non-volatile byte buffers: JEP 352 adds new JDK-specific file mapping modes so the FileChannel API can be used to create MappedByteBuffer instances that refer to non-volatile memory.

• HTTP/2 client: JEP 110 defines a new HTTP client API that implements HTTP/2 and WebSocket and can replace the legacy HttpURLConnection API.

• Stack-walking API: JEP 259 defines an efficient standard API for stack walking that allows easy filtering of, and lazy access to, the information in stack traces.

• TLS 1.3: JEP 332 implements version 1.3 of the Transport Layer Security (TLS) protocol.

• Convenience factory methods: JEP 269 defines library APIs that make it convenient to create instances of collections and maps with small numbers of elements, so as to ease the pain of not having collection literals in the Java programming language.

• Enhanced deprecation: JEP 277 revamps the @Deprecated annotation and provides tools to strengthen the API lifecycle.

Tooling Bracket

• JDK Mission Control: JDK Mission Control (JMC) is a suite of tools for OpenJDK and Oracle JDK focusing on production time profiling and diagnostics.

• jlink: JEP 282 creates a tool that can assemble and optimize a set of modules and their dependencies into a custom runtime image.

• jpackage: JEP 343 creates a tool for packaging self-contained Java applications.

• jdeprscan: This static analysis tool scans a JAR file (or some other aggregation of class files) for uses of deprecated API elements.
• Launcher for single-file source-code programs: JEP 330 enhances the java launcher to run a program supplied as a single file of Java source code, including usage from within a script by means of "shebang" files and related techniques.

• Javadoc search: JEP 225 adds a search box that appears in the header of all pages of API documentation generated by the standard doclet. This search box can be used to search for program elements and tagged words and phrases within the documentation.

• jshell: JEP 222 provides an interactive tool to evaluate declarations, statements, and expressions of the Java programming language. It also provides an API so that other applications can leverage this functionality.

• Multi-release JAR files: JEP 238 extends the JAR file format to allow multiple Java release-specific versions of class files to coexist in a single archive.

Runtime Bracket

• Application class-data sharing: To improve startup and footprint size, JEP 310 extends the existing class-data sharing feature to allow application classes to be placed in the shared archive.

• Compact strings: JEP 254 adopts a more space-efficient internal representation for strings.

• ZGC: The Z Garbage Collector, also known as ZGC, is a scalable low-latency garbage collector designed to meet several goals: Pause times do not exceed 10 ms; pause times do not increase with the heap or live-set size; and it handles heaps ranging from 8 MB to 16 TB in size.

• Graal JIT: JEP 317 enables the Java-based JIT compiler, Graal, to be used as an experimental JIT compiler on the Linux/x64 platform.

• Modular JDK: Use the Java Platform Module System, specified by JSR 376 and implemented by JEP 261, to modularize the JDK. Learn more in JEP 200.

• Thread-local handshakes: JEP 312 introduces a way to execute a callback on threads without performing a global VM safepoint, which makes it both possible and cheap to stop individual threads, not just all threads or none.

• JDK Flight Recorder: JEP 328 provides a low-overhead data collection framework for troubleshooting Java applications and the HotSpot JVM.

• Helpful NullPointerException: JEP 358 improves the usability of NullPointerExceptions generated by the JVM by describing precisely which variable was null.

Conclusion

Put your thinking caps on, and as you vote during each Twitter poll, please post your comments on the thread to explain why you voted a certain way. Please also share your opinions on the matchup. If you use the hashtag #MovedByJava, the broader community can also take part in the conversation.

Finally, this face-off competition isn’t meant to be rooted in agreement. Everyone would like to hear, understand, and accept differences in opinion. Spirited debate is healthy, and let’s always remember to be polite, professional, open-minded, and sincere. That’s the hallmark ethos of the Java ecosystem.

Happy birthday, Java! Oh, one more thing. Please join me for a panel discussion celebrating this milestone at 9:00 a.m PT on Tuesday, May 20, 2020.
Sharat Chander

Sharat Chander has worked in the IT industry for 20 years, for firms such as Bell Atlantic, Verizon, Sun Microsystems, and Oracle. His background and technical specialty is in Java development tools, graphics design, and product/community management. Chander has been actively involved in the Java Community for 15 years, helping drive greater Java awareness, acceptance, adoption, and advocacy. At Oracle, as the director of Java developer relations, Chander serves as the JavaOne conference content chairperson, a role he’s filled for 7 years, where he drives the technical content strategy and Java community involvement in the conference. He is a frequent keynote speaker and participant in developer programs worldwide. Chander holds a BS in corporate finance from the University of Maryland and an MBA in international business from Loyola College, Maryland. You can find Chander at multiple global developer events and Java community engagements. When not growing visibility for Java, he follows his other passion for baseball and fanatically following his hometown Baltimore Orioles.

Twitter handle: @Sharat_Chander