

Contributing to OpenJDK: Participating in stewardship for the long-term

Joseph D. Darcy ([OpenJDK darcy](#),  [jddarcy](#),  [@jddarcy](#),  [@jddarcy](#))

OpenJDK Compatibility & Specification Review (CSR) Group Lead,

JCP Spec Lead JSRs 269 (annotation processing) and 334 (Project Coin)

Inaugural lead engineer for OpenJDK 6

Java Platform Group, Oracle

JCP Executive Committee, April 12, 2023

Who am I?

- Long-time JDK engineer; multiple projects including:
 - [Compatibility & Specification Review \(CSR\) Group](#) Lead: review interface changes in JDK feature and update releases, ≈400 issues reviewed per year
 - Spec lead for:
 - [JSR 269](#) (annotation processing) in Java SE 6 and continuing maintenance lead
 - [JSR 334](#) in Java SE 7 ([Project Coin](#) – small language changes)
 - Over 1,000 commits in OpenJDK mainline

Outline

- OpenJDK Release Model
- Background
 - JCP Model
 - Compatibility Policies
 - Possible API/feature lifecycles
- Logistics of Contributing to OpenJDK
 - Suggestions on contributed to a JEP
 - Other kinds of contributions
 - General Contributions
- Questions and Discussion

A lookahead: ways to contribute

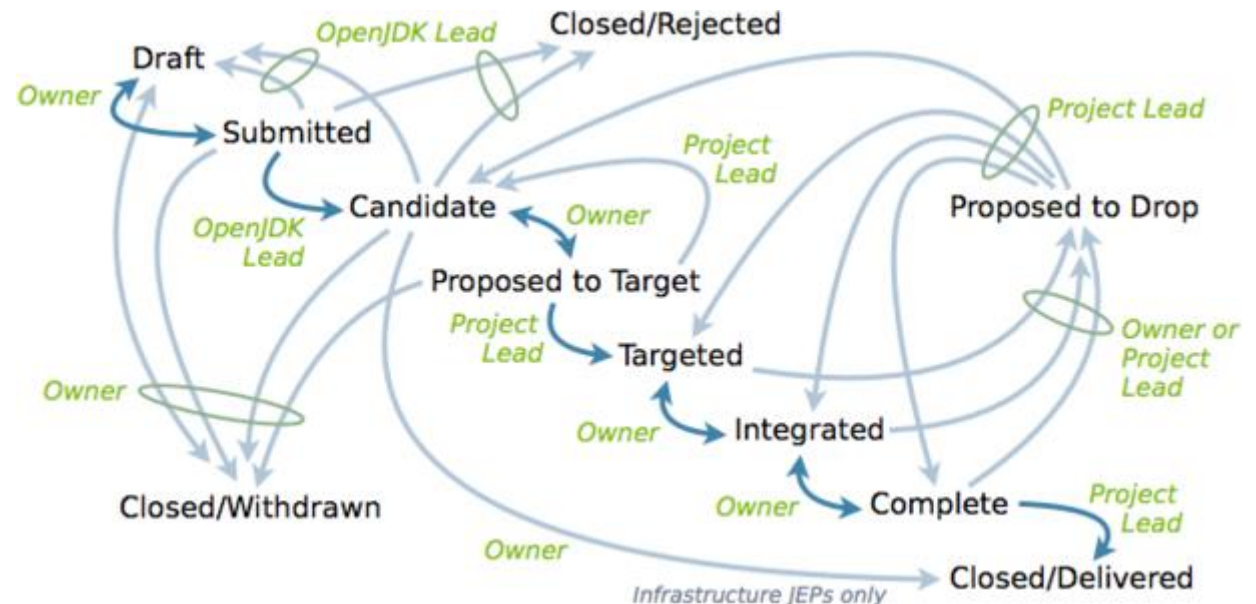
- Comment on GitHub PR's or correspond on mailing lists
 - Perhaps even initiate a mailing list thread or PR yourself 😊
- Send in feedback on JEPs (JDK Enhancement Proposals)
- *Try out new features and write up your experience.*
- *See how the early access builds work in your CI system.*

- Will discuss context of JDK development in much of the rest of the talk to contribute more effectively.

OpenJDK Release Model

What is a JEP?

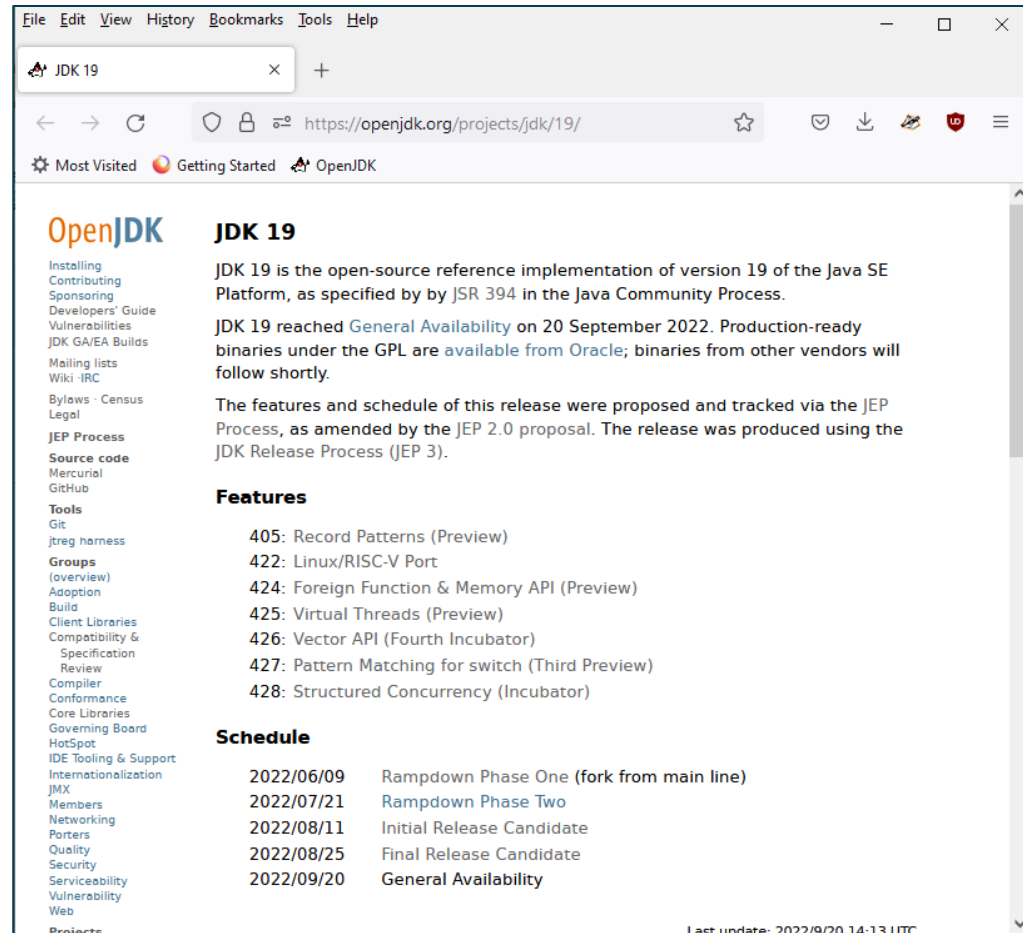
- JDK Enhancement Proposal; open to committers, used for:
 - Policy documents (e.g. JEP 12: Preview Features)
 - Project management of features (e.g. JEP 444: Virtual Threads) *and so much more!*
 - Rational of the feature, design alternatives
 - Sample usages
 - “One stop shopping.”
 - Submitted JEPs are requested to be added to the technical roadmap
 - Candidate JEPs are on the technical roadmap
 - Proposed to Target requests to bind a JEP to a particular release



JEPs I've worked on

- [JEP 306](#): Restore Always-Strict Floating-Point Semantics (JDK 17)
- [JEP 369](#): Migrate to GitHub (JDK 16)
- [JEP 357](#): Migrate from Mercurial to Git (JDK 16)
- [JEP 296](#): Consolidate the JDK Forest into a Single Repository (JDK 10)
- ...

JEPs in JDK Feature Releases



The screenshot shows the OpenJDK 19 project page. The browser address bar displays `https://openjdk.org/projects/jdk/19/`. The page features a sidebar with navigation links such as "Installing", "Contributing", "Sponsoring", "Developers' Guide", "Vulnerabilities", "JDK GA/EA Builds", "Mailing lists", "Wiki - IRC", "Bylaws - Census", "Legal", "JEP Process", "Source code", "Mercurial", "GitHub", "Tools", "Groups", "Adoption", "Build", "Client Libraries", "Compatibility & Specification", "Review", "Compiler", "Conformance", "Core Libraries", "Governing Board", "HotSpot", "IDE Tooling & Support", "Internationalization", "JMX", "Members", "Networking", "Porters", "Quality", "Security", "Serviceability", "Vulnerability", and "Web". The main content area includes the "OpenJDK" logo, the title "JDK 19", and a description: "JDK 19 is the open-source reference implementation of version 19 of the Java SE Platform, as specified by JSR 394 in the Java Community Process." It also states that "JDK 19 reached General Availability on 20 September 2022. Production-ready binaries under the GPL are available from Oracle; binaries from other vendors will follow shortly." A "Features" section lists JEPs 405 through 428, and a "Schedule" table shows the release timeline from 2022/06/09 to 2022/09/20. The page footer indicates "Last update: 2022/9/20 14:13 UTC".

OpenJDK **JDK 19**

JDK 19 is the open-source reference implementation of version 19 of the Java SE Platform, as specified by JSR 394 in the Java Community Process.

JDK 19 reached [General Availability](#) on 20 September 2022. Production-ready binaries under the GPL are available from Oracle; binaries from other vendors will follow shortly.

The features and schedule of this release were proposed and tracked via the JEP Process, as amended by the JEP 2.0 proposal. The release was produced using the JDK Release Process (JEP 3).

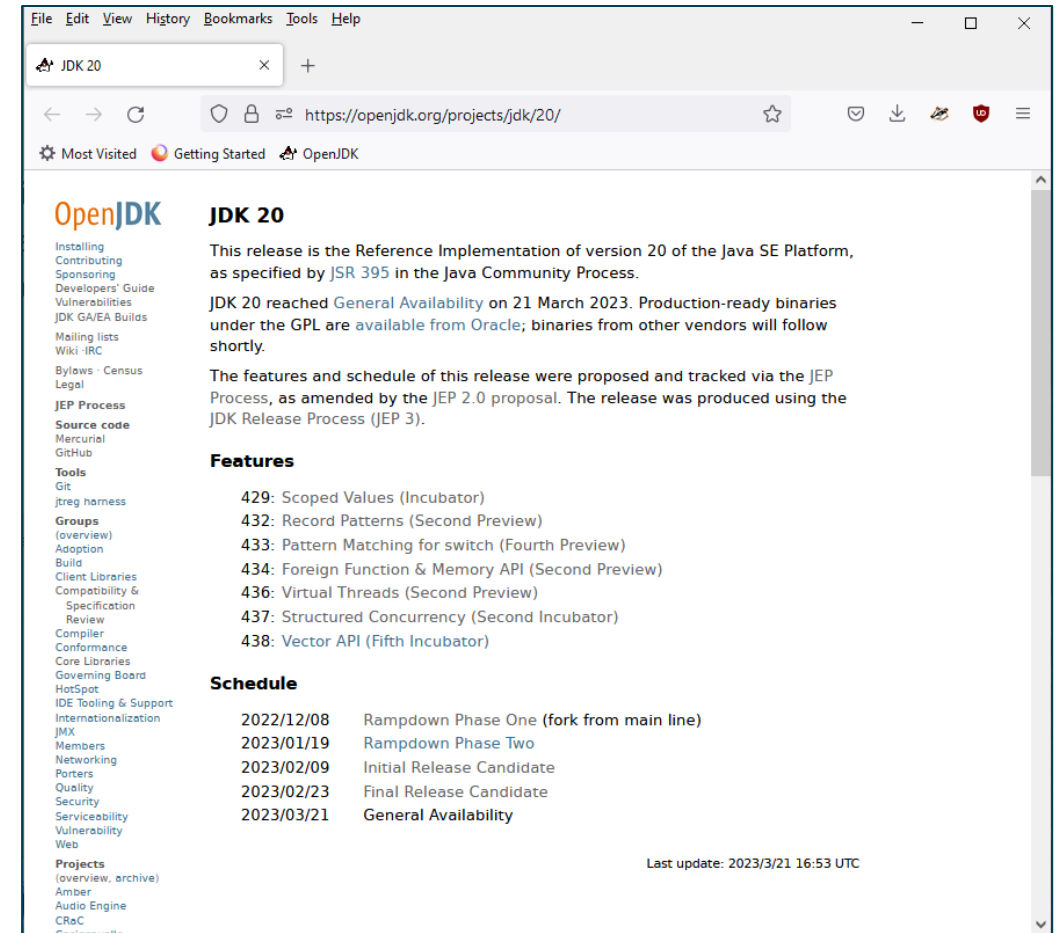
Features

- 405: Record Patterns (Preview)
- 422: Linux/RISC-V Port
- 424: Foreign Function & Memory API (Preview)
- 425: Virtual Threads (Preview)
- 426: Vector API (Fourth Incubator)
- 427: Pattern Matching for switch (Third Preview)
- 428: Structured Concurrency (Incubator)

Schedule

| | |
|------------|--|
| 2022/06/09 | Rampdown Phase One (fork from main line) |
| 2022/07/21 | Rampdown Phase Two |
| 2022/08/11 | Initial Release Candidate |
| 2022/08/25 | Final Release Candidate |
| 2022/09/20 | General Availability |

Last update: 2022/9/20 14:13 UTC



The screenshot shows the OpenJDK 20 project page. The browser address bar displays `https://openjdk.org/projects/jdk/20/`. The page features a sidebar with navigation links such as "Installing", "Contributing", "Sponsoring", "Developers' Guide", "Vulnerabilities", "JDK GA/EA Builds", "Mailing lists", "Wiki - IRC", "Bylaws - Census", "Legal", "JEP Process", "Source code", "Mercurial", "GitHub", "Tools", "Groups", "Adoption", "Build", "Client Libraries", "Compatibility & Specification", "Review", "Compiler", "Conformance", "Core Libraries", "Governing Board", "HotSpot", "IDE Tooling & Support", "Internationalization", "JMX", "Members", "Networking", "Porters", "Quality", "Security", "Serviceability", "Vulnerability", and "Web". The main content area includes the "OpenJDK" logo, the title "JDK 20", and a description: "This release is the Reference Implementation of version 20 of the Java SE Platform, as specified by JSR 395 in the Java Community Process." It also states that "JDK 20 reached General Availability on 21 March 2023. Production-ready binaries under the GPL are available from Oracle; binaries from other vendors will follow shortly." A "Features" section lists JEPs 429 through 438, and a "Schedule" table shows the release timeline from 2022/12/08 to 2023/03/21. The page footer indicates "Last update: 2023/3/21 16:53 UTC".

OpenJDK **JDK 20**

This release is the Reference Implementation of version 20 of the Java SE Platform, as specified by JSR 395 in the Java Community Process.

JDK 20 reached [General Availability](#) on 21 March 2023. Production-ready binaries under the GPL are available from Oracle; binaries from other vendors will follow shortly.

The features and schedule of this release were proposed and tracked via the JEP Process, as amended by the JEP 2.0 proposal. The release was produced using the JDK Release Process (JEP 3).

Features

- 429: Scoped Values (Incubator)
- 432: Record Patterns (Second Preview)
- 433: Pattern Matching for switch (Fourth Preview)
- 434: Foreign Function & Memory API (Second Preview)
- 436: Virtual Threads (Second Preview)
- 437: Structured Concurrency (Second Incubator)
- 438: Vector API (Fifth Incubator)

Schedule

| | |
|------------|--|
| 2022/12/08 | Rampdown Phase One (fork from main line) |
| 2023/01/19 | Rampdown Phase Two |
| 2023/02/09 | Initial Release Candidate |
| 2023/02/23 | Final Release Candidate |
| 2023/03/21 | General Availability |

Last update: 2023/3/21 16:53 UTC

What are “Preview” and “Incubator” features?

Will be discussed later...

Overview of JDK Release Process

JEP 3: JDK Release Process

- Previously, multi-year releases
- Since 2018 starting with JDK 10, new feature release every six months, March and September. Each release developed over about 9 months.
- Four phases:
 - Least-restricted development (\approx six months)
 - Rampdown Phase One (RDP 1)
 - Rampdown Phase Two (RDP 2)
 - Release-Candidate Phase (RC)
- The last RC build is then released as the GA build.

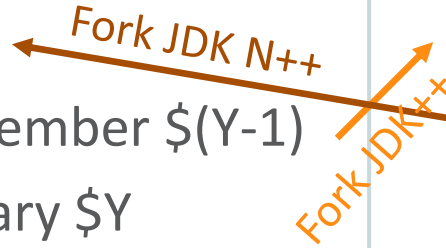
Sample JDK release schedules

March release year \$Y

- Dev starts: early June \$(Y-1)
- Rampdown 1 start: early December \$(Y-1)
- Rampdown 2 start: mid January \$Y
- Initial release candidate: early February \$Y
- GA: late March \$Y

September release year \$Y

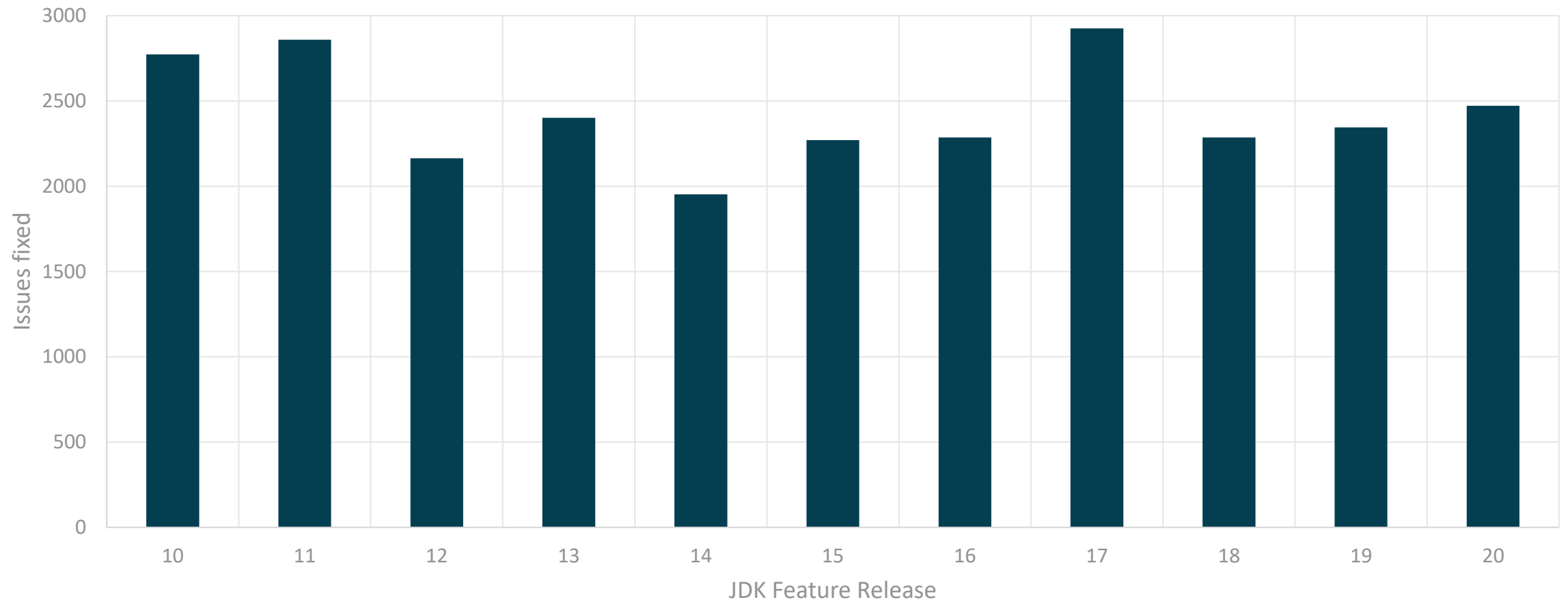
- Dev starts: early December \$(Y-1)
- Rampdown 1 start: early June \$Y
- Rampdown 2 start: mid July \$Y
- Initial release candidate: early August \$Y
- GA: late September \$Y



Current JDK Release Process

- Predictable schedule
- Always an open-for-business repo for developers to push to
- Skipping details of:
 - Handling of update releases
 - LTS (long-term support) vs non-LTS – however; note that non-LTS releases are still “real” releases worth using and testing on, etc.
 - (Such details are JDK-vendor specific)

Fixes per JDK Feature Release

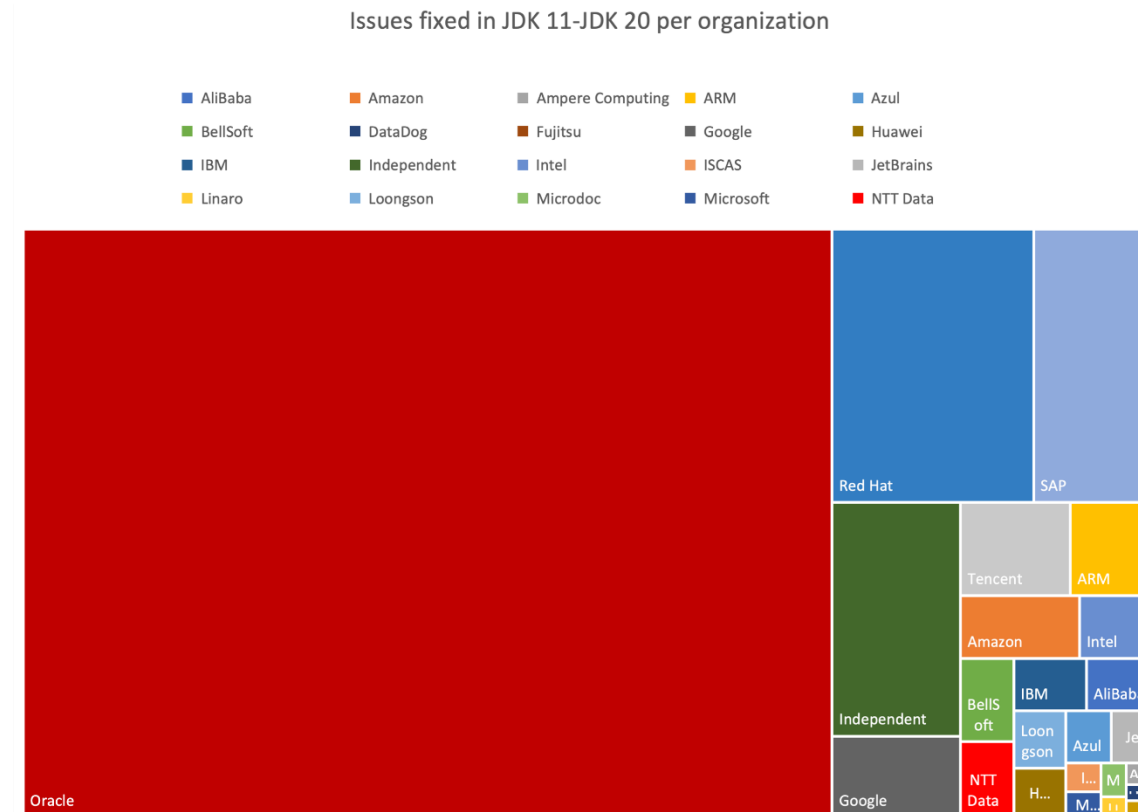


Bug fix observations

- About 2,400 fixes per JDK feature release
- Lots of changes to potentially comment on (or contribute!) beside large features and JEPs

Who has done what

The Arrival of Java 20! by Sharat Chander



Background

JCP and Open Source

- Since JCP 2.5 in 2002, the JCP has allowed and embraced open source development of the technologies standardized through the process
- For Java SE, the *reference implementation* is built from sources in OpenJDK repositories

What about the **OpenJDK** community?

- Good faith down payment by publication of HotSpot and javac sources in 2006; rest of JDK in 2007 (longer effort to remove all binary plugs, etc.)
- OpenJDK is licensed under open source licenses, predominantly GPLv2, (with the ClassPath Exception for the libraries)
- Initial OpenJDK sources populated from the in-progress JDK 7.
- A “backward branch” from JDK 7 used to make OpenJDK 6.
 - Red Hat’s IcedTea project first to get a OpenJDK 6 binary passing the Java SE 6 TCK.

JCP model

The Java Community Process, starting circa 1998

JCP Triad for the Java SE 20 Umbrella JSR 395

Specification

JLS
JVMS
java.*
javax.*
...

Reference
Implementation
(RI)

Build of OpenJDK 20
On Linux and Windows
<https://jdk.java.net/java-se-ri/20>

Technology
Compatibility Kit
(TCK)

JCK 20

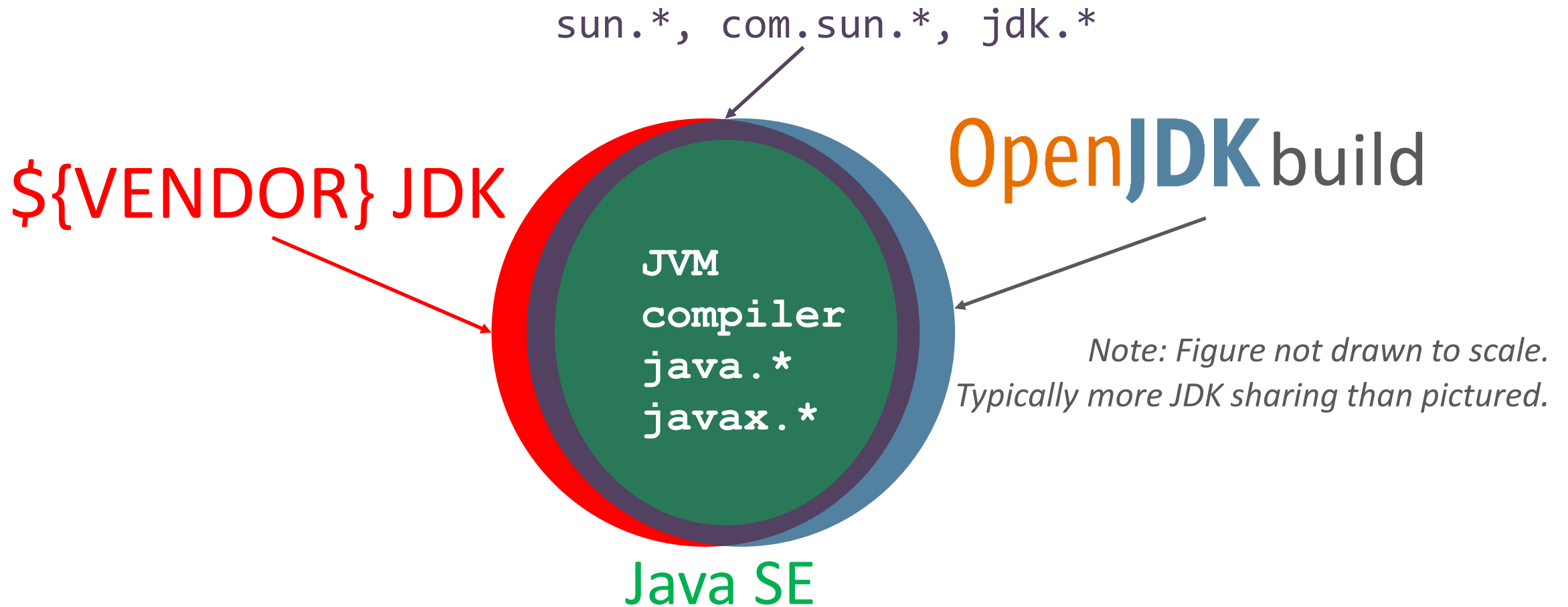
Logistics

- A JSR is run to cover Java SE changes in each JDK *feature* release; e.g: [JSR 393 for Java SE 18](#), [JSR 394 for Java SE 19](#), [JSR 395: Java SE 20](#), etc.
- Without benefit of a *MR* (maintenance review) of the Java SE platform specification, cannot make normative changes to the Java SE APIs in an *update* release.
 - This constraint precludes the additional of technically innocuous methods, often caught by the *signature test* (checks for supersetting, subsetting).
- MRs of the platform are rarely done, but are done with sufficient cause, such as JSR 337 MR 3 for adding TLS 1.3 to Java SE 8.

Different categories of interfaces and maintenance domains

A JDK doesn't just contain Java SE!

`\${VENDOR}` JDK & OpenJDK builds, Java SE interfaces



Implementations of many Java SE technologies are included in the OpenJDK sources

- Java Virtual Machine (HotSpot)
- Java Language Specification (javac)
- Class libraries in the `java.*` and `javax.*` namespace
 - (Java EE classes can also be in `javax.*` etc.)
- Most Java SE-related work now done under the umbrella JSR

Not all (Open)JDK APIs are defined in the JCP

Public exported APIs of the JDK but *not* Java SE:

- Tools in various modules:
 - `jdk.compiler` (javac)
 - `jdk.jpackage`
 - `jdk.jshell`
- Tree API (for abstract syntax trees, ASTs):
`jdk.compiler/com.sun.source.*`

JDK Internal APIs

- JDK Internal APIs should *not* be used outside of the JDK itself, including but not limited to:
 - `sun.*`
 - `jdk.internal.*`
 - non-public methods/fields in `java.*`

Compatibility Policies

The constraints of success

- The Java platform's success in part stems from strong compatibility policies
- Those policies constrain the kinds of changes that can be made in subsequent releases
- Therefore, important to get an API/interface “right” in the first release it ships in
 - At least right enough in the factors that are impractical to change afterward; taking YAGNI into account [preview and incubator](#) options, etc.
- Counterpoint: avoid issues like those in Python community with Python 3: [*Mercurial's Journey to and Reflections on Python 3*](#)

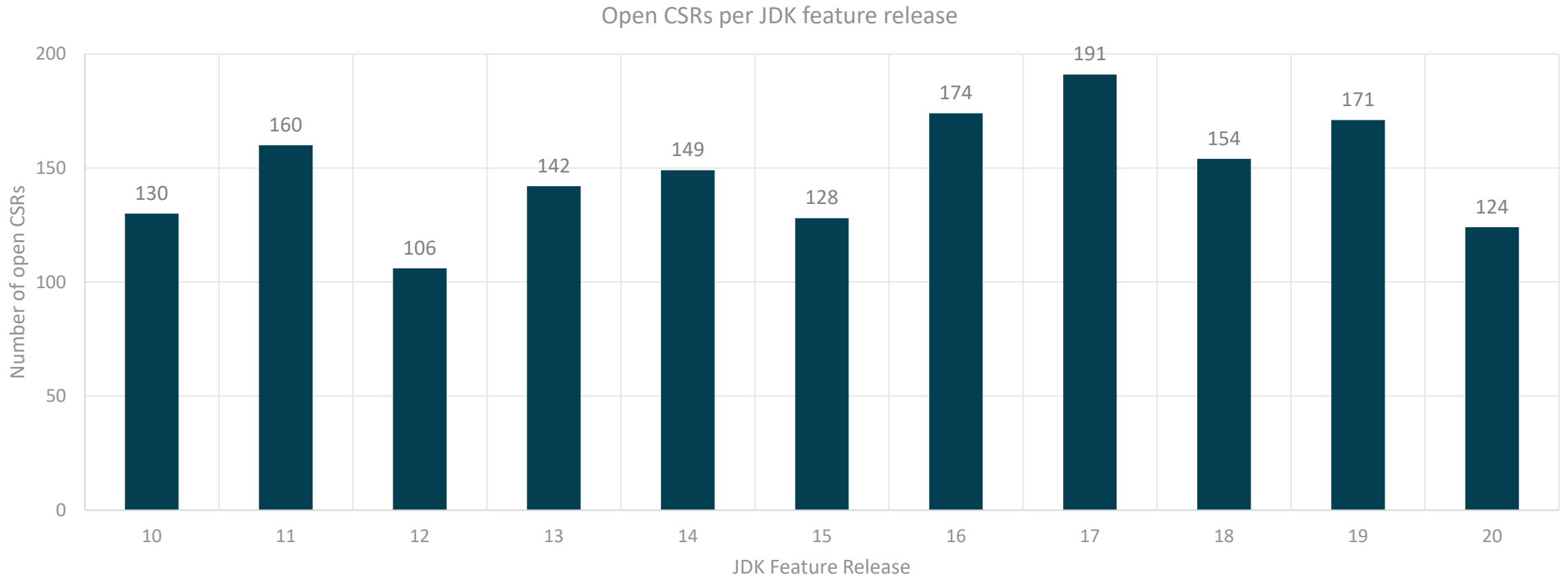
What is the CSR?

Process to review interfaces changes and an [OpenJDK group](#)

- [Compatibility and Specification Review](#)
 - Covers various kinds of exported interfaces of the JDK
 - Looks after source, binary, and behavioral compatibility
 - Reviews changes in the Java language, core libraries, as well as HotSpot
 - On average $\approx 6\%$ of fixes in a JDK feature release also go through CSR review
- CSRs used in preparation of JCP material for a feature release, HT Iris Clark:
 - [Java SE 21 CSR dashboard](#)
 - [Java SE CSRs in Java SE 20 \(JSR 395\)](#)

Open CSRs per JDK feature release

Average ≈ 150 /release under the six-month release model.



Background: JDK General Evolution Policy

From the [OpenJDK CSR wiki page](#)

“The general compatibility policy for exported APIs implemented in the JDK is:

- 1. Don't break binary compatibility (as defined in the [Java Language Specification](#)) without sufficient cause.*
- 2. Avoid introducing source incompatibilities.*
- 3. Manage behavioral compatibility changes.”*

- Extends to language evolution too
 - Continue to recognize old class files
 - Limit cases where currently legal code stops compiling
 - Avoid changes in code generation introducing behavioral change
- Goal: balance between stability and progress

Short compatibility definitions

- Binary compatibility: do programs still link?
- Source compatibility: do programs still compile and still compile to equivalent class files?
- Behavioral compatibility: do programs still operate the same way at runtime?

Binary Compatibility

- Specific definition: the continued ability to *link*; see [JLS Chapter 13 Binary Compatibility](#)
- Broken by, for example:
 - removing types
 - changing `public` methods to be `private`
 - ...

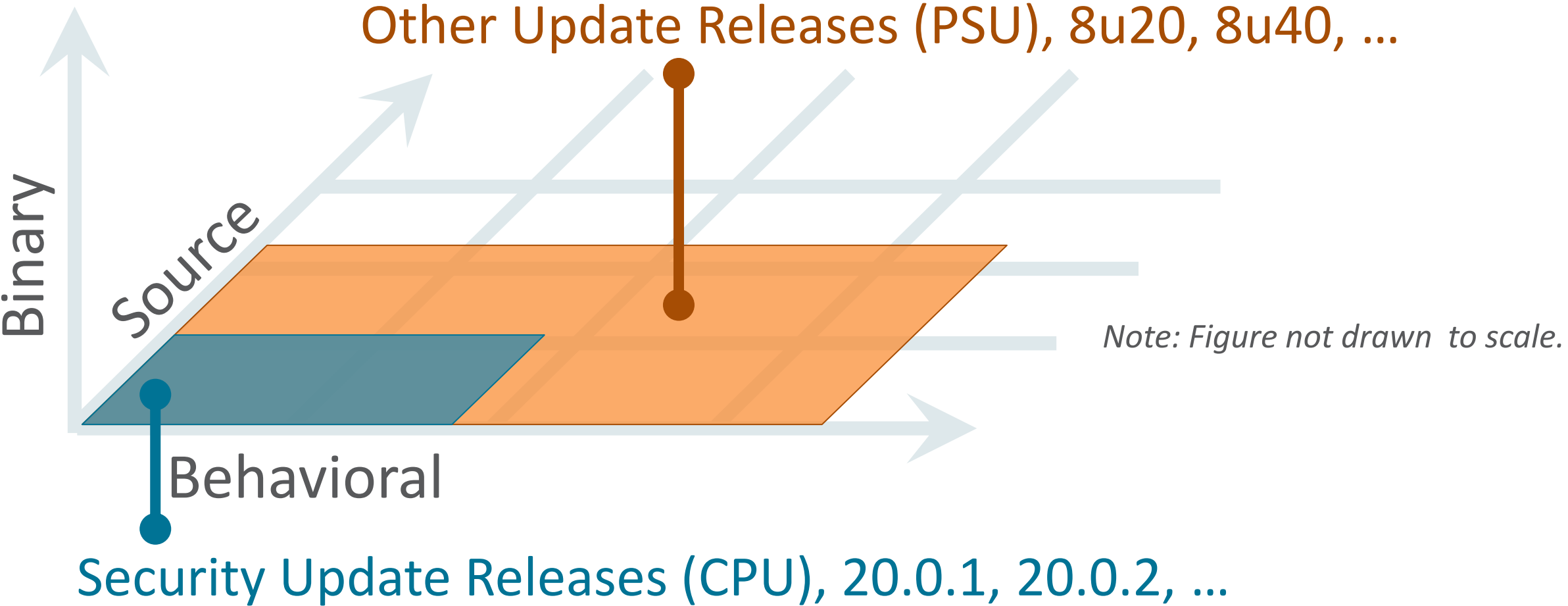
Source Compatibility

- A surprisingly subtle topic!
- Degrees of source compatibility ([CSR](#), [Dev. Guide](#)):
 - Does the client code still compile (or not compile)?
 - If the client code still compiles, do all the names resolve to the same binary names in the class file?
 - If the client code still compiles and the names do *not* all resolve to the same binary names, does a *behaviorally equivalent* class file result?
- For example, adding overloaded methods/constructors to a class can change how source code using that class is compiled; e.g. if a class has a constructor taking a `double`, add a constructor taking a `long`.

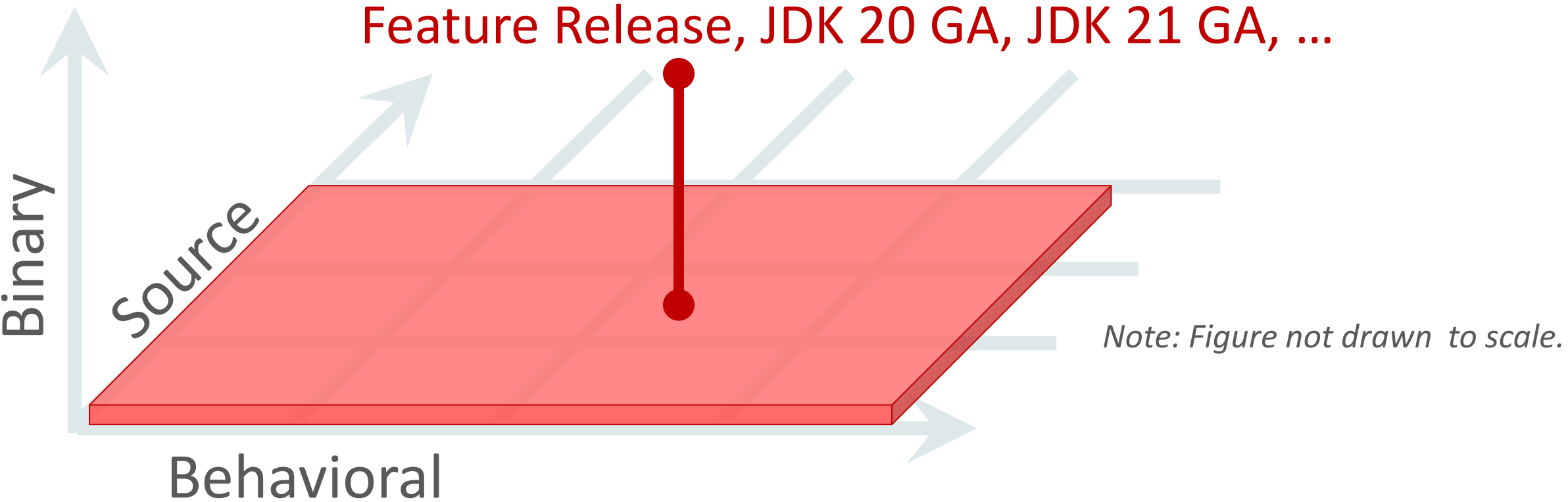
Behavioral Compatibility

- Intuitively, before and after a change to the JDK, do “the same” or “equivalent” inputs produce “the same” or “equivalent” results?
- Difficult to define equivalence in all cases:
 - Reflective/introspective operations
 - Side-channels (relative performance, etc.)
- (Also includes serialization compatibility)

Release Compatibility Regions



Feature Release Compatibility Region



Example: changing the iteration order of HashSet

- Specification of [HashSet.iterator\(\)](#):
“Returns an iterator over the elements in this set. The elements are returned in no particular order.”
- Changing iteration order is allowed by the specification and:
 - Binary compatible (same set of methods)
 - Source compatible (compilation of clients independent of `iterator` method body)
 - But a change in behavioral compatibility;
people can and do implicitly (and accidentally) rely on iteration order
For comparison see [JEP 269: Convenience Factory Methods for Collections](#)
- Therefore, this kind of change *generally* wouldn't be made in either kind of update release, but would be (and has been) made in a platform release.

Looking ahead

How to better accommodate behavioral changes

- What you can do: try out EA builds in your CI systems!
- [quality-discuss](#) outreach to open-source projects through OpenJDK

Vacuously true, but aspirational, statement

Except for timing dependencies or other non-determinisms and given sufficient time and sufficient memory space, a program written in the Java programming language should compute the same result on all machines and in all implementations.

[Preface to the JLS, first edition](#)

Interpretation of and limits to compatibility

Old feature request from a customer circa 2006 for JDK 7 planning:

“What we're [at customer] talking about is compatibility between consecutive feature releases. To give an example, binary compatibility [customer means behavioral compatibility –Ed.] is extremely important to us:

We strongly believe that any Java binary application that is known to run without any problems on Java SE platform version N should be able to run without any problems on Java SE platforms of later versions as well (> N). There shouldn't be a need to recompile, rebuild, retest, etc., Java applications upon upgrading the underlying Java SE platform layer to a higher version. Similarly, we believe source compatibility is extremely important and needs to be(come) a Java SE platform feature by design as well.”

Why this request is infeasible *as stated*

Adversarial programs

- Program tests for the Java version and fails if it isn't equal to a particular value
⇒ couldn't update Java version number
- Program measures the relative performance of two methods. If the relative performance isn't within epsilon of the expected ratio, the program fails
⇒ couldn't add new intrinsics
- Program calls javac and expects a compilation failure on a given text
⇒ couldn't evolve the Java programming language
- Program intentionally causes a NPE and fails if the detail message doesn't match
⇒ couldn't add [helpful NullPointerExceptions](#)
- Cloud vendor wants customer program *P* to use **at least** *X* CPU and *Y* memory
⇒ can't improve performance properties for end users
- ...

And *really* adversarial programs...

- Program tests for the presence of security vulnerabilities
⇒ couldn't fix security vulnerabilities

Aside on source compatibility

- Customer request for greater source compatibility is likely a reaction to the addition of “assert” as a keyword in Java SE 1.4 ([JSR 41](#)).
- Current [keyword management policies](#) mitigate the impact of such changes today – contextual keywords ([JLS §3.9](#)), hyphenated-keywords, etc.

Actual examples

- Eclipse IDE “broke” when the JDK vendor name was changed from “Sun” to “Oracle” (checked vendor to set command-line flags of the JVM)
- Many behavioral incompatibilities in JDK 9 are not due to the module system, but from changing the version numbering from “1.8.x” to “9.0.y”.
- Judgement needed to balance stability with progress
- Reasonable behavioral compatibility is a shared responsibility of the users and platform provider
 - Platform provider should produce a good specification
 - If users code to the specification, should have fewer problems updating

Evolution of behavioral compatibility expectations

JDK 1.0/1.1 era examples

- In the beginning, the consensus was that details of `toString` and `hashCode` needed to be specified to satisfy WORA properties; examples:
 - `Short.hashCode()`:
“Returns a hash code for this Short; equal to the result of invoking `intValue()`.”
[So `Short.hashCode()` is constrained to only use half the bits of the 32-bit hash.
Alternative: “All distinct short values have distinct hash codes...”]
 - `Method.toString()`:
“The string is formatted as the method access modifiers, if any, followed by the method return type, followed by a space, followed by the class declaring the method, followed by a period, followed by the method name, followed by a parenthesized, comma-separated list of the method's formal parameter types.” ...
[Is it really necessary or helpful to give a parseable grammar of the `toString` output?]

Newer convention: less-specific specifications

JDK 5.0 and 6

- [Annotation.toString\(\)](#):

“Returns a string representation of this annotation. The details of the representation are implementation-dependent, but the following may be regarded as typical:

```
@com.example.Name(first="Duke", middle="of", last="Java") ”
```

[Exact behavior of `Annotation.toString()` has changed several times to be more faithful to annotations as used in source code.]

- [javax.lang.model.Element.hashCode\(\)](#):

“Obeys the general contract of `Object.hashCode()`.”

[Explicitly indicates there is nothing else to say.]

Set.of() iteration order

JEP 269: Convenience Factory Methods for Collections, JDK 9

- Static factories to create unmodifiable sets; for such sets:
 - “The iteration order of set elements is unspecified and is subject to change.”
 - Implementation of the iteration order of these sets is randomized per JVM-invocation.

Q: What about cases where those compatibility policies are too restrictive or premature?

A: Preview Features and Incubator Modules

Motivation: back during Project Coin in JDK 7...

- Working on various language changes including try-with-resources.
- Put out a call to [try out try-with-resources](#):
 - Implementation in promoted JDK 7 build
 - [Fully specified](#) with a null-handling policy to throw NPE on a null resource
 - Included [library support](#) and regression testing, etc.
- Time passed ... and about six months later Rémi Forax [sent in feedback](#) based on experience that the null-handling policy should be changed to ignore a null resource.
- After due consideration, the [null-handling policy was changed](#).

Handling the policy change

- Changing the `null`-handling policy as suggested would be outside the scope of what would be considered an acceptable language change after the feature was included in a Java SE release.
- Since JDK 7 was a multi-year release, updating the `null`-handling policy could be made (update specification, implementation, and tests) before GA.
- How could this kind of situation be accommodated in a six-month release?

Preview Features and Incubator Modules

[JEP 12: Preview Features](#) & [JEP 11: Incubator Modules](#)

Preview Features

“A *preview feature* is a new feature of the Java language, Java Virtual Machine, or Java SE API that is fully specified, fully implemented, and yet impermanent. It is available in a JDK feature release to provoke developer feedback based on real world use; this may lead to it becoming permanent in a future Java SE Platform.”

Preview Features, cont.

“A preview feature is:

- a new feature of the Java language ("preview language feature"), or
- a new feature of the JVM ("preview VM feature"), or
- a new module, package, class, interface, method, constructor, field, or enum constant in the `java.*` or `javax.*` namespace ("preview API")

whose design, specification, and implementation are complete, but which would benefit from a period of broad exposure and evaluation before either achieving final and permanent status in the Java SE Platform or else being refined or removed.”

Preview Features, cont.

“The key properties of a preview feature are:

- 1. High quality.* A preview feature must display the same level of technical excellence and finesse as a final and permanent feature of the Java SE Platform. For example, a preview language feature must respect traditional Java principles such as readability and compatibility, and it must receive appropriate treatment in the reflective and debugging APIs of the Java SE Platform.
- 2. Not experimental.* A preview feature must not be experimental, risky, incomplete, or unstable. ...
- 3. Universally available.* The Umbrella JSR for the Java SE \$N Platform enumerates the preview features of the platform. ...”

Preview: discussion

- Analogy with `try-with-resources`: as a preview feature it would definitely have had an explicit policy of `null` handling, but would have reserved the right to change that policy before `try-with-resources` became a non-preview part of the platform.
- Preview features *are* part of Java SE, *but* have a different cross-release compatibility policy; they can be arbitrarily changed or even removed.
- Example language feature of pattern matching:
 - [JEP 406: Pattern Matching for switch \(Preview\)](#) in JDK 17
 - [JEP 420: Pattern Matching for switch \(Second Preview\)](#) in JDK 18
 - [JEP 427: Pattern Matching for switch \(Third Preview\)](#) in JDK 19
 - [JEP 433: Pattern Matching for switch \(Fourth Preview\)](#) in JDK 20
 - [JEP 441: Pattern Matching for switch](#), currently Candidate, (in JDK 21?)

Preview: discussion, cont.

- Use of Preview features is opt-in at compile-time and runtime
- See [JEP 12](#) for detailed discussion of different use cases, preview language features vs. preview APIs, reflective APIs, etc.

Incubator Modules

JEP 11: Incubator Modules

- tl;dr APIs live in `jdk.incubator.*`; *not* part of Java SE
- Opt-in for usage; can change or be dropped between releases.
- Example: Foreign-Memory Access API started incubating in JDK 14 ([JEP 370](#)); iteration of the API was a preview in JDK 19 ([JEP 424](#)).

Possible API/feature lifecycles

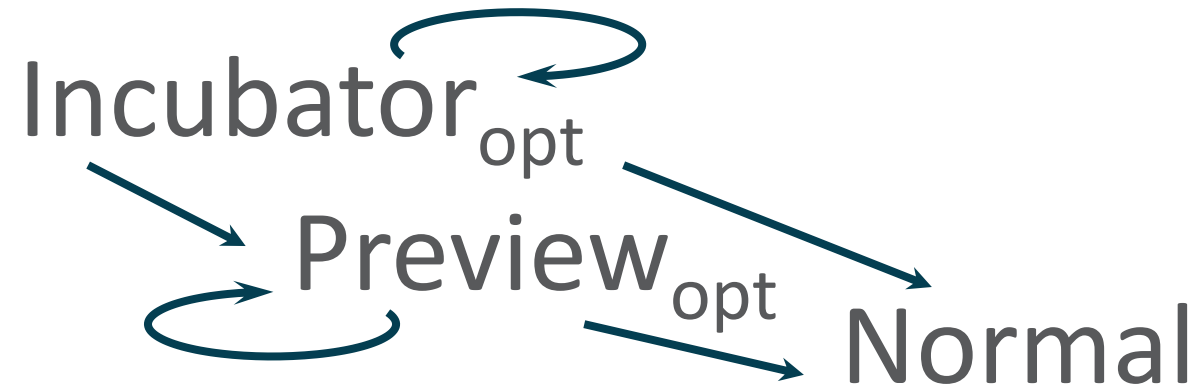
API/feature lifecycles

Common case

Normal

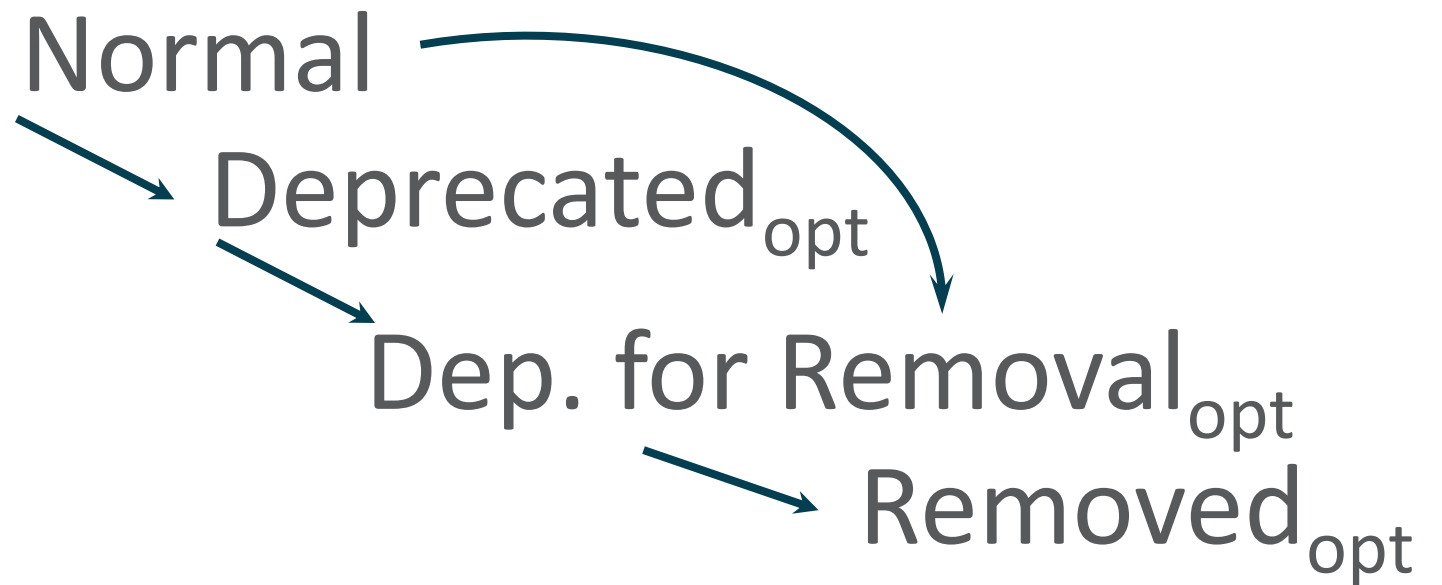
API/feature lifecycles

Incubator and/or Preview



API/feature lifecycles

Deprecation; see [JEP 277: Enhanced Deprecation](#)



Logistics of Contributing to OpenJDK

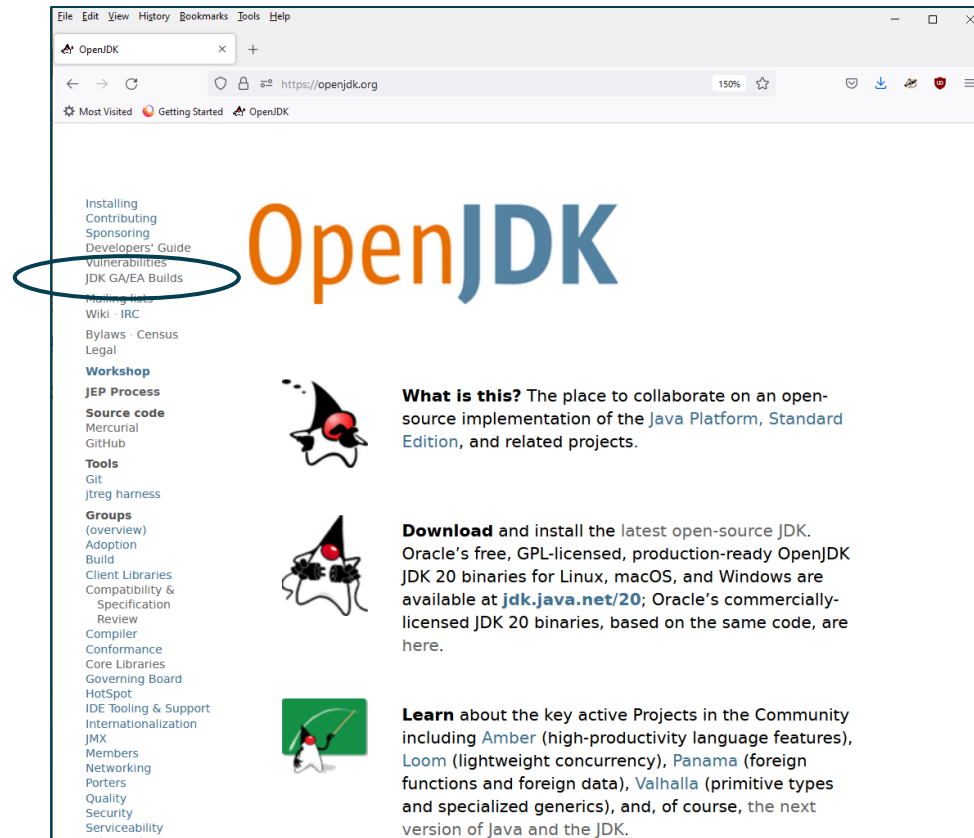
[How to contribute](#)
[OpenJDK Developers' Guide](#)

Security Vulnerabilities

- *If you think you've found a security vulnerability, separate procedures handled by the [OpenJDK Vulnerabilities group](#).*

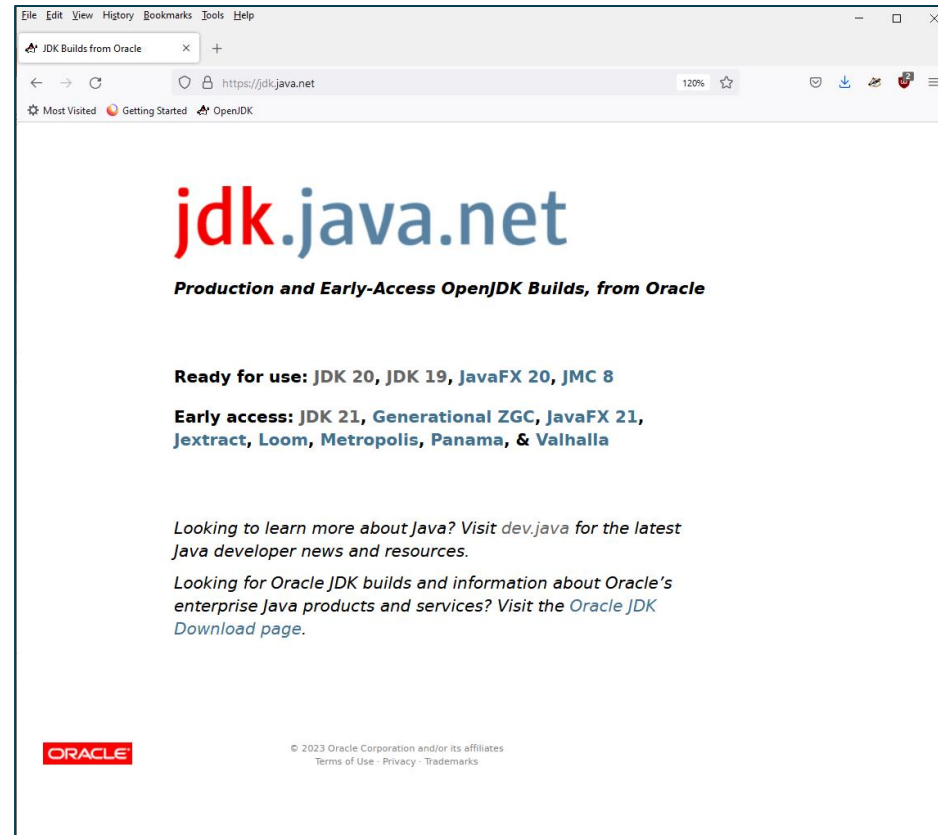
Basics: find a build to get started...

<https://openjdk.org/>

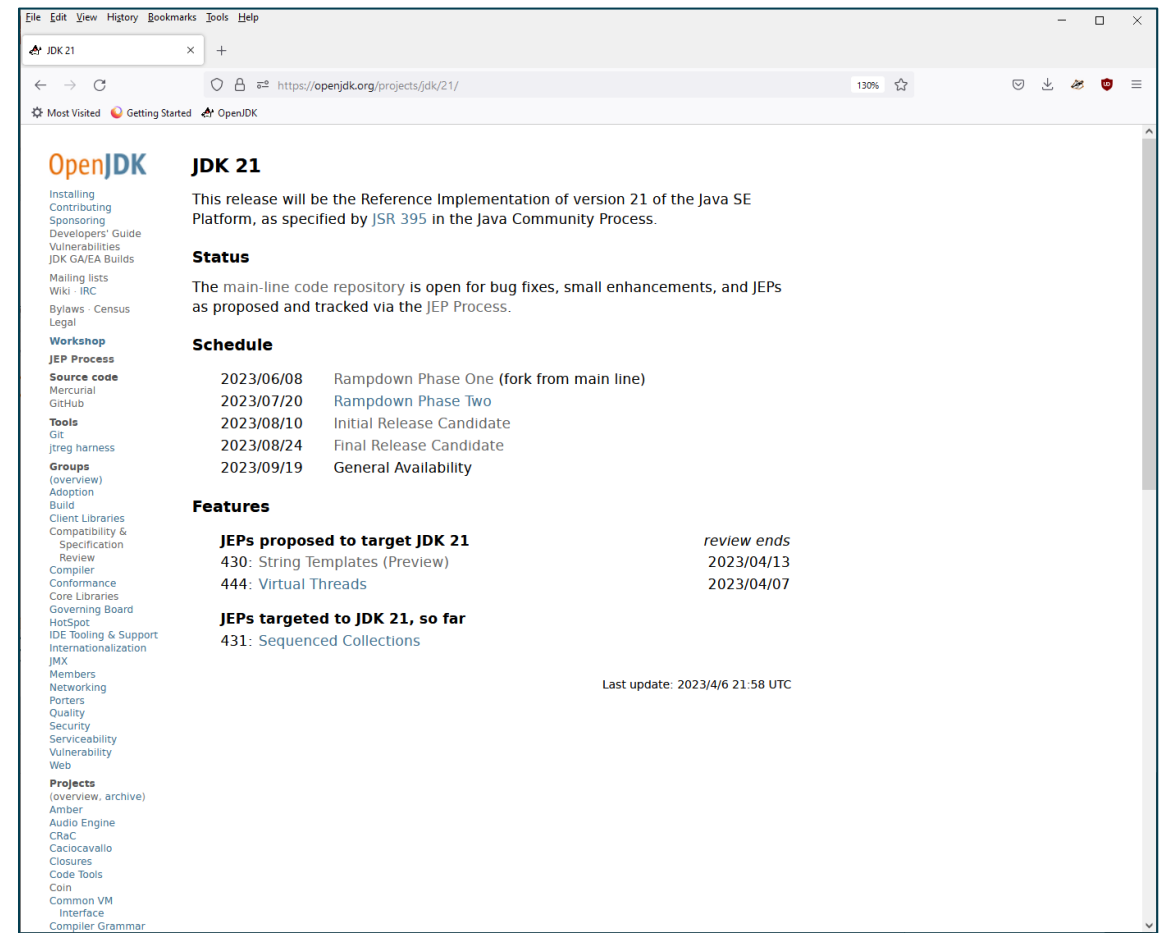
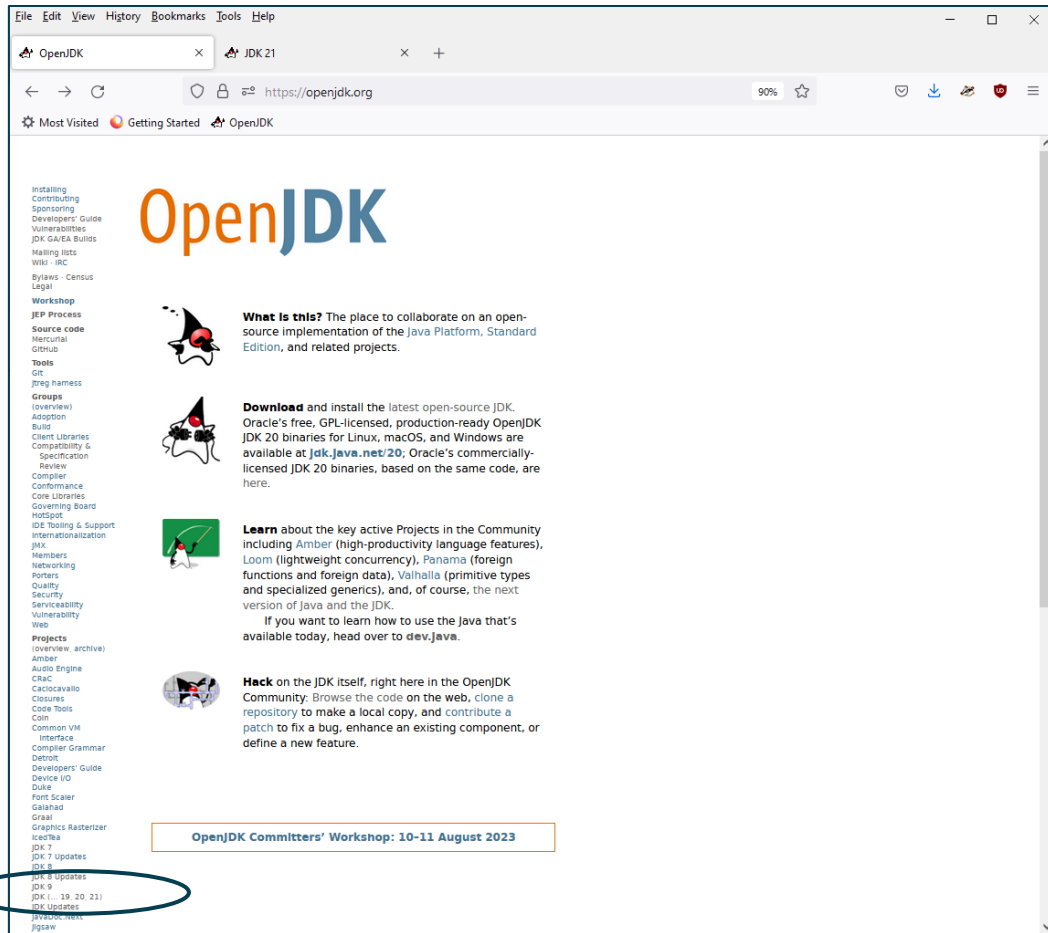


Build landing page

<https://jdk.java.net/>



For a given feature release project like JDK 21...

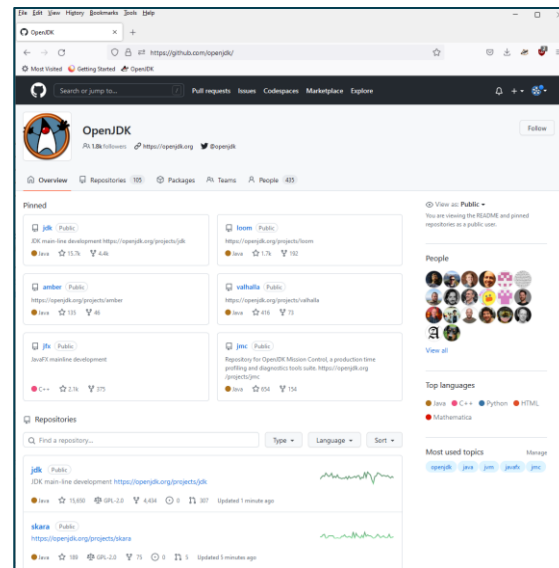


<https://openjdk.org/projects/jdk/21/>

General OpenJDK infrastructure

Including, but not limited to:

- [Mailing lists](#); used for different technology areas ([core-libs](#), [HotSpot](#), [client-libs](#), etc.) and well as various projects ([Amber](#), [Valhalla](#), etc.)
- [JBS – JDK Bug System](#): Jira instance, most bugs of interest are in the “JDK” project
- [OpenJDK Wiki](#)
- [OpenJDK Project on GitHub](#)



Suggestions on contributing to a JEP

Example: JEP 430: String Templates (Preview)

The screenshot shows the OpenJDK website page for JEP 430: String Templates (Preview). The page includes a navigation sidebar on the left and a main content area. The main content area displays the following information:

| | |
|--------------------|--|
| Owner | Jim Laskey |
| Type | Feature |
| Scope | SE |
| Status | Proposed to Target |
| Release | 21 |
| Component | specification / language |
| Discussion | amber dash dev at openjdk dot org |
| Effort | M |
| Duration | M |
| Reviewed by | Alex Buckley, Brian Goetz, Maurizio Cimadamore |
| Endorsed by | Brian Goetz |
| Created | 2021/09/17 13:41 |
| Updated | 2023/04/06 21:41 |
| Issue | 8273943 |

Summary

Enhance the Java programming language with *string templates*. String templates complement Java's existing string literals and text blocks by coupling literal text with embedded expressions and *template processors* to produce specialized results. This is a preview language feature and API.

Goals

- Simplify the writing of Java programs by making it easy to express strings that include values computed at run time.
- Enhance the readability of expressions that mix text and expressions, whether the text fits on a single source line (as with string literals) or spans several source lines (as with text blocks).
- Improve the security of Java programs that compose strings from user-provided values and pass them to other systems (e.g., building queries for databases) by supporting validation and transformation of both the template and the values of its embedded expressions.
- Retain flexibility by allowing Java libraries to define the formatting syntax

Send comments to
amber-dev@openjdk.org

Avoid

“I just read over the first half of the JEP; here is my hot-take on why these proposed changes are *not* Java...”

Recommendations and observations

- JEP text is a distillation of *significant* effort
- Suggestion: *do the homework*; read the whole JEP. For context, more detailed discussion and rationale may be present in mailing list threads
 - If available, try out build with the feature
 - Send comments based on experience/retrofitting

Other kinds of contributions

Download an EA build and...

- Try it out in your CI system
 - Report any issues (performance or functional regressions, etc.) to a mailing list or file a bug (can be done *without* a JBS account).
- Program against preview features or incubator modules and report on experiences
- If you help manage an open source project, consider joining the quality-discuss efforts.
- *If you only try out LTS releases, skipping over thousands and thousands of bug fixes/improvements, possibly with behavioral compatibility impact.*
- Can use `javac --release $OLD` to reliably compile to supported older releases.

Standing suggestions

- Reduce reliance on JDK internals!
 - Both in your libraries own and your dependencies!
 - Question every `--add-exports` or `--add-opens` that you see
- Move away from deprecated functionality; see `$JDK/bin/jdeps`
- Feedback sent before a JDK's rampdown 1 starts is easier to act upon in that release.

General Contributions

You can send in a large unsolicited GitHub PR *as long as...*

- ...you don't have any attachment to it.
 - A PR can be much more expensive to maintain than to write. (“Free puppy!”)
 - A PR can even be much more expensive to review than to write.
- For example, for language changes, the cost of the change includes its interactions with all existing language features, and *all future language features*.

Recommendations

- Solicit input on large changes on appropriate mailing list *before* implementing them.
 - This includes refactoring changes that touch many files; likely better to decompose into one PR per review domain.

Questions and Discussion

Thank you!

Hope to see you in the OpenJDK community.

Safe Harbor Statement

The preceding is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.