Java™ magazine

By and for the Java community

NOVEMBER/DECEMBER 2018

LARGEST JAVA SURVEY EVER

RESULTS PAGE 15

Java Present and Future

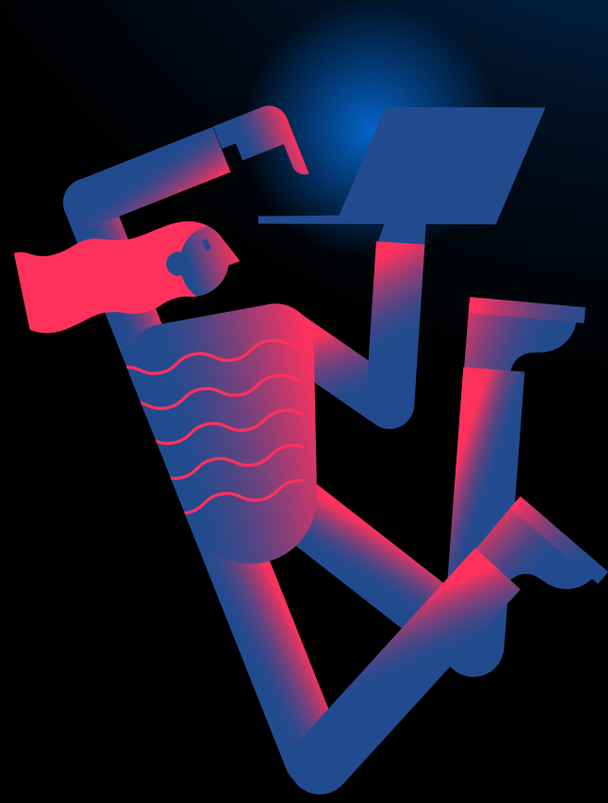
ORACLE.COM/JAVAMAGAZINE

ORACLE®

IntelliJ IDEA

Capable and Ergonomic IDE for JVM

Indepth coding assistance
Cross-language refactorings
Clever error analysis
and much more.



Download at jetbrains.com/idea

THE DRIVE
TO DEVELOP

JET
BRAINS

//table of contents/

COVER FEATURES

OTHER FEATURES

DEPARTMENTS

COVER ART BY WES ROWELL

EDITORIAL

Editor in Chief
Andrew Binstock

Senior Managing Editor
Leslie Steere

Copy Editors
Lea Anne Bantsari, Karen Perkins

Contributing Editors
Deirdre Blake, Simon Roberts, Mikalai Zaikin

Technical Reviewer
Stephen Chin

DESIGN

Senior Creative Director
Francisco G Delgadillo

Design Director
Richard Merchán

Senior Designer
Arianna Pucherelli

Designer
Jaime Ferrand

Senior Publication Designer
Sheila Brennan

Production Designer
Kathy Cygnarowicz

PUBLISHING

Group Publisher
Karin Kinnear

Audience Development Manager
Jennifer Kurtz

ADVERTISING SALES

Tom Cometa

Mailing-List Rentals
Contact your sales representative.

RESOURCES

Oracle Products
+1.800.367.8674 (US/Canada)

Oracle Services
+1.888.283.0591 (US)

ARTICLE SUBMISSION

If you are interested in submitting an article, please email the editors.

SUBSCRIPTION INFORMATION

Subscriptions are complimentary for qualified individuals who complete the subscription form.

MAGAZINE CUSTOMER SERVICE

java@omeda.com

PRIVACY

Oracle Publishing allows sharing of its mailing list with selected third parties. If you prefer that your mailing address or email address not be included in this program, contact Customer Service.

Java Magazine is published bimonthly and made available at no cost to qualified subscribers by Oracle, 500 Oracle Parkway, MS OPL–3A, Redwood City, CA 94065-1600.

Unexpected Results from the Java Developer Survey

Large surveys surface results that belie the common wisdom.

The largest survey ever done of Java developers (with some 10,500 responses) contains numerous data points worthy of thoughtful consideration. To be sure, many of the data points confirm suppositions we have had about the industry in general, but a few surprises stand out. Here are my thoughts on them.

JVM Languages. This question inquired about the principal JVM language used by respondents. As you'd expect, Java came in first by a wide margin. The big surprise is that Clojure, the functional-style JVM language, came in second, beating out Kotlin and Groovy by a whisker and ahead of Scala by a somewhat larger margin. Several key points add some context to these numbers. Kotlin's greatest adoption is on the Android platform, where it is now an officially supported language. Android developers are not likely to be well represented in Java user groups or in *Java Magazine*—the two principal sources of respondents to the survey. Likewise, it's not clear that Scala developers are well represented. I expect that if we could tap the entire universe of JVM and Android developers, the order of popularity would be Java, Kotlin, Scala, and then Clojure. Regardless, Clojure is an unexpected surprise and shows that functional programming has many adherents; its popularity also suggests that the Java team's focus on adding functional-style elements to Java (notably, lambdas and streams) is in tune with developer preferences.

Static Analysis. Currently, 36% of developers don't use static checking tools at all. This is hard to explain, because studies have repeatedly

PHOTOGRAPH BY BOB ADLER/STUDIO AT GETTY IMAGES FOR ORACLE

shown that static analysis of code is one of the most effective ways of finding defects.

I discussed static analysis in my September/October 2015 editorial and gave these statistics: "Relying on the work of Capers Jones, one of the most widely experienced analysts of software-engineering data, static analysis reduces defects in projects by 44% (from an average of 5.0 defects per unit of functionality to 2.8 defects). Among standard quality assurance techniques, only formal inspections have a higher effectiveness (60%). By comparison, test-driven development (TDD) comes in at 37%." That's a powerful argument for using static analysis.

If you're one of the developers who doesn't routinely use a static analysis tool, consider adding one of those listed on the related graph to your continuous integration process. In our survey, SonarQube was the preferred tool. But if you want to check your own code simply, either (or better yet, both) of the next two popular solutions are especially easy to add to your process: FindBugs and Checkstyle.

Code hosting. For most developers accustomed to poking around in open source projects, GitHub is often the principal hosting site they check. The other well-known code hosts, Bitbucket and GitLab, are viewed as distinctly secondary (even though they both offer private repositories for free, which GitHub does not do).

However, when we asked where the developers' *principal* project is hosted, the three major hosts are much closer: GitHub and Bitbucket are tied at 25%, with GitLab at 20%—indicating that businesses view the offerings through a different lens and find meritorious features that are invisible to open source projects. (We didn't survey developers on what those features are, but I expect they include pricing and the ability to integrate with the businesses' existing development pipelines. The latter factor probably most favors Bitbucket, which is owned by Atlassian, whose JIRA defect-tracking tool is used at many sites.)

Release Frequency. I am gratified to see that the mania for constant releasing that gripped our industry a few years ago has settled into something more sensible. Only 10% of developers work on projects that release once a day or more, whereas nearly half (46%) ship updates once a month or less frequently. To all of us who endured the feverish rate of revisions in our apps, this pace will come as good news. Surely you recall opening apps to find the UI had changed again, or that features you used regularly were now moved elsewhere or renamed while features you never employed were suddenly front and center. Constant, rushed updates was one of the least user-friendly IT practices, and I've noticed that, consistent with this chart, the number of presentations at conferences in which experts crow about their ability to push changes multiple times a day has slowly abated.

There are many other data points in this wide-ranging survey that merit thoughtful consideration.

Let me know what stands out for you.

Andrew Binstock, Editor in Chief
javamag_us@oracle.com
@platypusguy

SEPTEMBER/OCTOBER 2018

Comments as Design Elements

I'm so totally on board with your suggestions in your editorial "Using Comments to Design Classes," in the September/October issue—particularly your example of using a comment to describe what you're planning to do. I've been pseudocoding since 1985, when I graduated from Troy State University. To everyone who is willing to listen here on my development team, I say "write it down in English." Listen to what you wrote and rewrite it, if necessary, before you write a line of code. You will be amazed how easy your software development life will become!

I just want to say thank you for your editorial piece. I am printing it now, and it is going up on my cube wall.

—Peter Pahules
Scottsdale, Arizona

Step-by-Step Guide for Using Jakarta EE

I built a step-by-step guide for implementing the project described by Josh Juneau in his article "Jakarta EE: Building Microservices with Java EE's Successor," in the September/October issue. I've made it available for your readers to access.

—Mikalai Zaikin
Minsk, Belarus

Editor Andrew Binstock responds: This excellent tutorial contains a step-by-step setup of the IDE, the server, and finally of the implementation of the project, complete with numerous screenshots and hands-on tips. The guide is available in the Java Magazine *download area or directly as a PDF file here. Zaikin is a coauthor of the "Fix This" quiz column that appears in every issue.*

Covering Containers—and More

There is so much happening with Java these days that it's a relief to get the straight dope from *Java Magazine*. However, one topic I'd like to see more coverage of is developing container-based applications. Can you wrangle some content about that?

—Will Rubin
Bend, Oregon

Editor Andrew Binstock responds: Thanks for the kind words. We are indeed planning coverage of containers. In fact, here are the topics scheduled for 2019:

- *January/February: Lightweight frameworks*
- *March/April: Big data*
- *May/June: Containers (and coverage of Java 12)*
- *July/August: Libraries (our annual issue)*
- *September/October: Java 13 and a preview of Java 14*
- *November/December: Testing*

As always, editorial calendars are subject to change.

Corrigenda

In the September/October 2018 issue, we were less than clear about where the unpublished listings are located for Josh Juneau's article "Jakarta EE: Building Microservices with Java EE's Successor." They are located here. Note a small update to one of the files has been made since publication of the article.

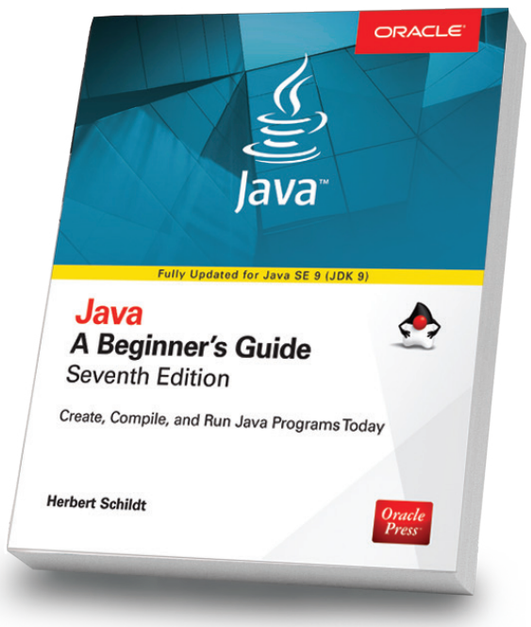
Contact Us

We welcome comments, suggestions, grumbles, kudos, article proposals, and chocolate chip cookies. All but the last two might be edited for publication. If your note is private, please indicate this in your message. Write to us at javamag_us@oracle.com. For other ways to reach us, see the last page of this issue.

Oracle Press™

Your Destination for Oracle and Java Expertise

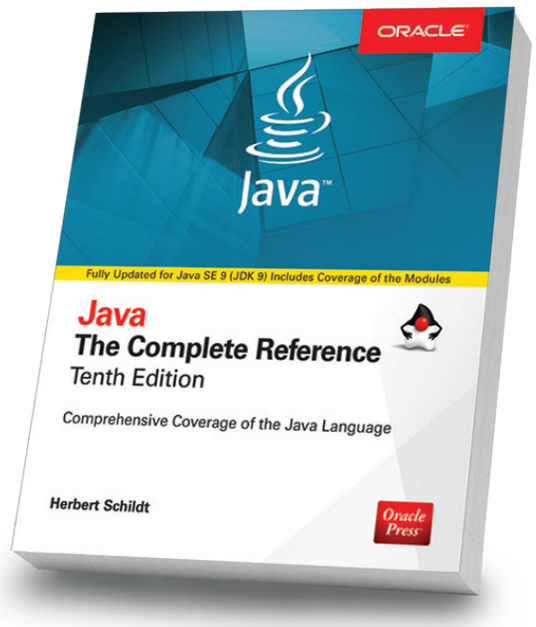
Written by leading experts in Java, Oracle Press books offer the most definitive, complete, and up-to-date coverage of Java available.



Java: A Beginner's Guide, 7th Edition

Herb Schildt

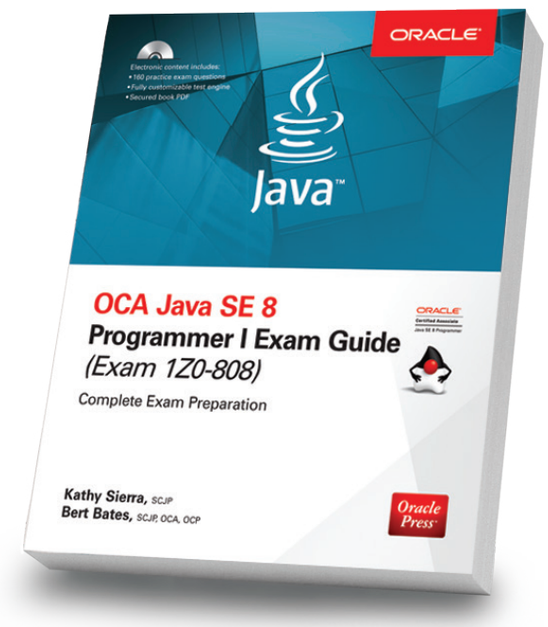
Revised to cover Java SE 9, this book gets you started programming in Java right away. Free online supplement covering key new features in JDK 10 available for download on the book's page on OraclePressBooks.com



Java: The Complete Reference, 10th Edition

Herb Schildt

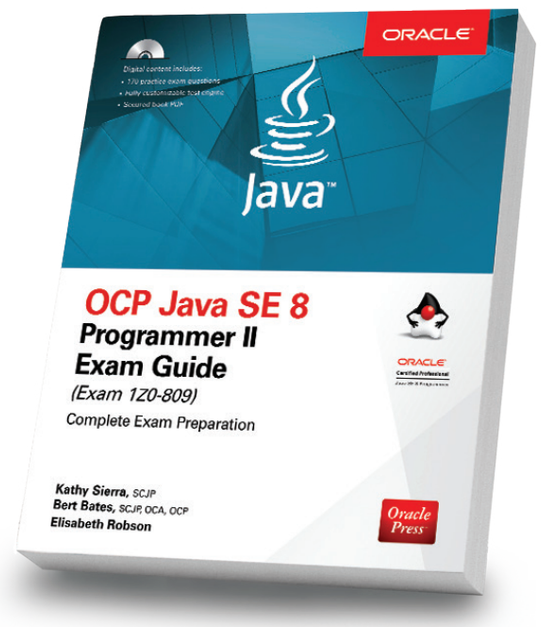
Updated for Java SE 9, this book shows how to develop, compile, debug, and run Java programs. Visit the book's page on OraclePressBooks.com to download free supplements on JDK's key new features.



OCA Java SE 8 Programmer I Exam Guide (Exam 1Z0-808)

Kathy Sierra, Bert Bates

Get complete coverage of all objectives for Exam 1Z0-808. Electronic practice exams include more than 200 questions that help you prepare for this challenging test.



OCP Java SE 8 Programmer II Exam Guide (Exam 1Z0-809)

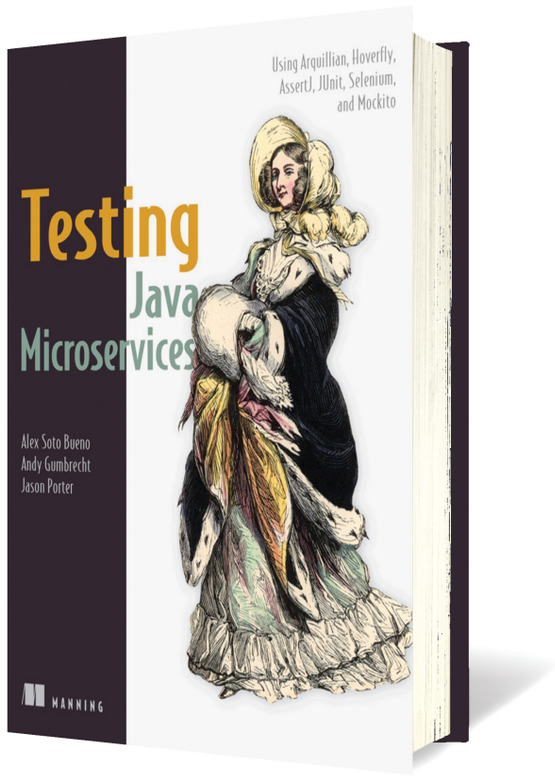
Kathy Sierra, Bert Bates, Elisabeth Robson

Prepare for the OCP Exam 1Z0-809 with this comprehensive guide which offers every subject appearing on the exam. Includes more than 350 practice questions.

Available in print and eBook formats.

Oracle Press™

www.OraclePressBooks.com • @OraclePress



TESTING JAVA MICROSERVICES

By Alex Bueno, Andy Gumbrecht, and Jason Porter

Microservices continue to gain ground in enterprises due to the benefits for which they have been most promoted: reduced size, loose coupling, and the composability they enable. IT managers have begun to understand that service-based computing is not the same as the service-oriented architecture (SOA) popular at the turn of the century. As microservices gain traction, however, there has arisen the nettlesome problem of testing them.

Testing Java Microservices aims to provide a solution to this problem by examining a small ecosystem of tools based on Arquillian. The latter is a container-agnostic testing framework designed to test services as a producer (does it work correctly internally?), as a consumer (how does it function when used as a REST service?), and as a client (by driving browser activity). The distinguishing element of Arquillian is the ability to run your microservices and then execute tests against them. It can drive unit tests and integration tests, and it understands containers, virtualization, and fundamental problems of a testing environment, such as using a separate classpath. (Note that Arquillian was originally a framework for testing apps running in a Java EE container. Its most recent versions have added Spring support and the ability to work with Docker containers.)

The authors, who are strongly connected to Arquillian, are at pains to include coverage of tools other than Arquillian. And they introduce some notable tools—in particular Pact, which is a family of test frameworks for verifying consumer-side contract testing. The authors also integrate testing with familiar products, such as JUnit and Mockito. In this sense, the book serves as a reasonable overview of testing microservices. However, everything is still fundamentally tied to Arquillian and its way of doing things. As a result, other useful tools such as mountebank, which provides over-the-wire test doubles, are not mentioned at all, even though mountebank would be an important resource to know about. (The authors do cover a similar resource, Hoverfly, which is briefly introduced and explained in the book.)

A final chapter covers the crucial topic of setting up the microservice tests to run in a continuous-delivery pipeline. This step is easy to overlook because of its complexity, leading the final delivered service to be tested only in isolation.

Overall, *Testing Java Microservices* is a good introduction to the issues of testing microservices that run on the JVM (they need not be written in Java as the title would imply). However, its greatest value is to organizations that are evaluating Arquillian or are already committed to its use as their primary microservices testing framework. *—Andrew Binstock*

//events/

CodeMash 2019
JANUARY 8–11, 2019
SANDUSKY, OHIO
CodeMash is an event that educates developers on current practices, methodologies, and technology trends in a variety of platforms and development languages including Java, .NET, Ruby, Python, and PHP. The Java track features presentations from many Java Champions.

Codemotion Berlin
NOVEMBER 20–21
BERLIN, GERMANY
Codemotion conferences are devoted to developers sharing the latest tech information and best practices among the tech community worldwide. Confirmed speakers at this event include Picnic CTO Daniel Gebler, Apache Software Foundation member Kanchana Welagedara, and Microsoft Senior Program Manager Christian Heilmann. The event is open to all languages and technologies and features coding lectures and workshops.

Topconf Tallinn 2018
NOVEMBER 20–22
TALLINN, ESTONIA
Topconf Tallinn is an international software conference covering Java, open source, agile development, architecture, and new languages.

Voxxed Days Cluj-Napoca
NOVEMBER 21–22
CLUJ-NAPOCA, ROMANIA
This conference brings together well-known speakers, core developers of popular open source technologies, and professionals willing to share their knowledge and experience. Scheduled speakers include Agile expert Venkat Subramaniam and Java Champion Vlad Mihalcea.

JVM-Con
NOVEMBER 27–28
COLOGNE, GERMANY
Among the topics slated for this German conference devoted to JVM languages are the JRE, concurrency, Java EE, mobile, and cloud-native development. (Website in German.)

Codemotion Milan
NOVEMBER 29–30
MILAN, ITALY
Codemotion conferences are devoted to developers sharing the latest tech information and best practices among the tech community worldwide. Confirmed speakers at this event include Rogue Wave Senior Software Engineer Enrico Zimuel, ThoughtWorks Quality Analyst Wamika Singh, and Accenture Manager Maurizio Mangione. The event is open to all languages and technologies and features coding lectures and workshops.

PHOTOGRAPH BY DOUGLAS SACHA/GETTY IMAGES

Clojure/conj 2018

NOVEMBER 29–DECEMBER 1
DURHAM, NORTH CAROLINA
This event is expected to draw more than 400 Clojure developers from around the world for three days of cutting-edge Clojure, ClojureScript, and functional programming talks.

DevTernity

NOVEMBER 30–DECEMBER 1
RIGA, LATVIA
The DevTernity forum covers the latest developments in coding, architecture, operations, security, leadership, and many other IT topics. Venkat Subramaniam, author of *Programming Concurrency on the JVM* and *Functional Programming in Java*, is slated to be one of the featured speakers.

The Lead Developer Austin

DECEMBER 6
AUSTIN, TEXAS
This conference for tech leads, engineering managers, and CTOs promises a day of inspiring talks and practical takeaways to help you become a better team leader. Topics planned for discussion include refactoring, code review, and GraphQL.

ArchConf 2018

DECEMBER 10–13
CLEARWATER, FLORIDA
ArchConf is an educational event for software architects, technical leaders, and senior developers presented by the No Fluff Just Stuff software symposium. Among scheduled sessions are talks on applying design patterns, building serverless applications, machine learning, and scalable microservices.

KubeCon + CloudNativeCon

DECEMBER 10, COMMUNITY EVENTS AND LIGHTNING TALKS
DECEMBER 11–13, CONFERENCE
SEATTLE, WASHINGTON
The Cloud Native Computing Foundation's flagship North American conference gathers adopters and technologists from leading open source and cloud native communities to further the education and advancement of cloud native computing.

EmTech Asia

JANUARY 22–23, 2019
SINGAPORE
EmTech Asia is an international conference hosted in collaboration with *MIT Technology Review*. Talks and presentations focus on emerging technologies.

SnowCamp 2019

JANUARY 23, 2019, WORKSHOPS
JANUARY 24–25, 2019, CONFERENCE
JANUARY 26, 2019, UNCONFERENCE
GRENOBLE, FRANCE
SnowCamp is a developer conference held in the French Alps that focuses on innovation, exchange, and research exploring web, cloud, DevOps, and software architecture topics. The event features a mix of sessions in French and English. The last day, dubbed "unconference," offers a unique opportunity to socialize with peers and speakers on the ski slopes.

DevConf.cz

JANUARY 25–27, 2019
BRNO, CZECH REPUBLIC
DevConf.cz is an open source developer and DevOps conference. There is no admission or ticket charge for DevConf.cz events, but free registration is required. All talks, presentations, and workshops will be conducted in English. Topics under consideration this year include blockchain, middleware, machine learning, and immutable operating systems.

PHOTOGRAPH BY AIMINTANG/GETTY IMAGES

PHOTOGRAPH BY SONSE/FLICKR

jSpirit
JANUARY 25–27, 2019
HAUSHAM, GERMANY
jSpirit is a new community-organized, nonprofit Java "unconference" in the Bavarian Alps featuring two days of sessions at the Lantenhammer Erlebnisdestillerie Hausham followed by two days of skiing. A mini-conference for kids, jSpirit4Kids, is slated for January 28.

Domain-Driven Design Europe
JANUARY 28–30, 2019, WORKSHOPS
JANUARY 31–FEBRUARY 1, 2019, CONFERENCE
AMSTERDAM, THE NETHERLANDS
This software development and engineering event spans analysis, modeling and design, systems thinking and complexity theory, architecture, testing and refactoring, visualization, and collaboration.

O'Reilly Software Architecture Conference
FEBRUARY 3–4, 2019, TRAINING
FEBRUARY 4–6, 2019, TUTORIALS AND CONFERENCE
NEW YORK, NEW YORK
This event consists of four days of in-depth professional software architecture training on topics ranging from domain-driven design and event-driven microservices to database architecture.

Jfokus
FEBRUARY 4–6, 2019
STOCKHOLM, SWEDEN
Sweden's largest developer conference will cover Java and JVM languages, as well as best practices and emerging technologies. On February 4, the conference will hold the Jfokus VM Tech Summit, which is an open technical collaboration among language designers, compiler writers, tool builders, runtime engineers, and VM architects.

JSConf Hawaii
FEBRUARY 7–8, 2019
HONOLULU, HAWAII
The inaugural Hawaiian JSConf for JavaScript developers promises a fun and welcoming gathering where diversity thrives and attendees can discuss JavaScript as a technology and grow it as a community.

DeveloperWeek SF Bay Area
FEBRUARY 20–24, 2019
OAKLAND, CALIFORNIA
The DeveloperWeek expo and conference series gathers 8,000 participants for a weeklong, technology-neutral programming conference and associated area events. Tracks include artificial intelligence, blockchain development, popular programming languages, microservices, and mobile.

AgentConf 2019
FEBRUARY 21–24, 2019
DORNBIRN AND LECH, AUSTRIA
AgentConf is an international event that combines software development and skiing and features talks by world-class engineers envisioning the future of mobile and web technologies. Speaker sessions are hosted in Dornbirn; skiing and networking take place in Lech.

Embedded World 2019
FEBRUARY 26–28, 2019
NUREMBERG, GERMANY
Now celebrating its 17th year, this annual embedded systems developer event will focus on the state of embedded intelligence, which is shaping systems ranging from autonomous vehicles to image recognition and embedded vision systems to preventive and demand-driven maintenance in small edge computers as well as high-performance cloud servers.

//events/

QCon London
MARCH 4–6, 2019, CONFERENCE
MARCH 7–8, 2019, WORKSHOPS
LONDON, ENGLAND
Although the content has not yet been announced, past QCon conferences have offered several Java tracks along with tracks related to web development, DevOps, cloud computing, and more. Proposed session topics include machine learning, software ethics, and DSLs.

Devnexus
MARCH 6–8, 2019
ATLANTA, GEORGIA
Devnexus is an international open source developer conference. Its stated goal is to connect developers from all over the world, provide affordable education, and promote open source values. Past presenters have included Java experts Donald Raab and Venkat Subramaniam.

ConFoo Montreal
MARCH 13–15, 2019
MONTREAL, CANADA
This multitechnology conference for web developers promises 155 presentations and typically features sessions tailored for Java and JVM developers.

JavaLand 2019
MARCH 19–21, 2019
BRÜHL, GERMANY
This annual conference features lectures on subjects such as core Java and JVM languages, microservices architecture, front-end development, and much more.

Voxxed Days Zürich
MARCH, 2019 (DATES NOT FINALIZED)
ZÜRICH, SWITZERLAND
Voxxed Days Zürich shares the Devoxx philosophy that content comes first and draws internationally renowned and local speakers discussing topics such as cloud development, containers, machine learning, and programming languages. The schedule of speakers had not yet been announced at press time.

Are you hosting an upcoming Java conference that you would like to see included in this calendar? Please send us a link and a description of your event at least 90 days in advance at javamag_us@oracle.com. Other ways to reach us appear on the last page of this issue.

//user groups/

CENTRAL OHIO JUG

The Central Ohio Java Users Group, COJUG, was founded in 1997 and has been going strong ever since. Hosted in Columbus, Ohio, COJUG holds two meetings the second Tuesday of every month—one during the lunch hour and one in the evening. These meetings usually present the same speaker and content, but offering two meeting times helps to accommodate attendees' schedules. This format has also enabled the group to more than double regular attendance, averaging more than 70 attendees each month from a mailing list of more than 650 members.

Eleven years ago, COJUG, along with several other local user groups including the .NET user group, started a conference called CodeMash that combined the different technology interests and communities. The conference is held each January in Sandusky, Ohio. Northern Ohio in January might seem like a strange location and time, but holding the conference at the Kalahari indoor water park with an associated free kids' conference called KidzMash makes this a warm and family-friendly experience. The four-day event attracts more than 2,200 developers and architects. Over the years, CodeMash has hosted many Java Champions and regional JUG leaders as presenters. The next event takes place January 8 to 11, 2019.

If you are interested in getting involved in COJUG as an attendee, speaker, or sponsor, please visit the COJUG website or email the user group at cojug@cojug.org for details.

DEVELOPER EVENTS FROM THE DEVOXX & VOXXED FAMILY
COMING IN 2019

DEVOXX™

DEVOXX.COM

FRANCE 17-19 APRIL

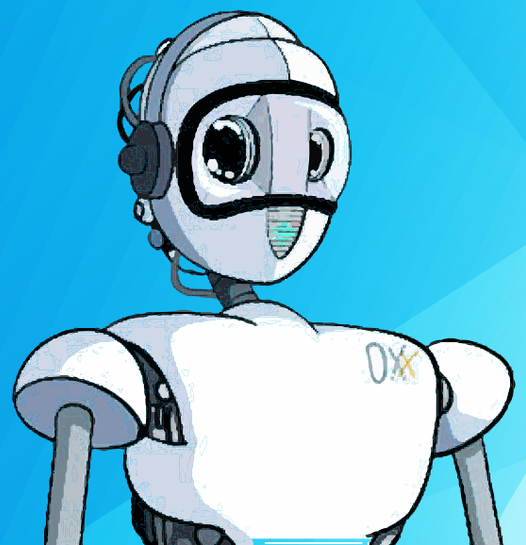
UK 8-10 MAY

POLAND 24-27 JUNE

UKRAINE DATE TBA

BELGIUM DATE TBA

MOROCCO DATE TBA



BUCHAREST 20-22 MARCH

ZURICH 21 MARCH

MILAN 13 APRIL

CERN 1 MAY

MELBOURNE 14 MAY

FRONTEND, BUCHAREST 22 MAY

MINSK 24-25 MAY

SINGAPORE 30-31 MAY

ATHENS 31 MAY-1 JUNE

LUXEMBOURG DATE TBA

TICINO DATE TBA

MICROSERVICES, PARIS DATE TBA

THESSALONIKI DATE TBA

CLUJ-NAPOCA DATE TBA

VOXXEDDAYS

VOXXEDDAYS.COM

Java—Where We Are and What's Coming

There is a lot happening in Java, and in this issue we do our best to make the state of Java as clear as possible. We begin with the largest survey ever (page 15) of Java developers: more than 10,500 responses were received to the questionnaire we ran with the security startup Snyk. The data is as interesting as it is substantial. The survey covers JDK, tools in use, processes, and finally a profile of Java developers.

We follow that up with a look at Java 11 (page 39), which was released in September: what's in this release that you need to know about—such as changes to lambda syntax, a new HTTP client, and the updated WebSocket interface. The Java 11 release was the first in several years not to include JavaFX, which has been spun out from the JDK. This means that JavaFX can evolve on its own timeline, which is discussed (page 62) by Johan Vos, one of its principal developers.

ART BY WES ROWELL

We also examine upcoming technology from the Valhalla project (page 56), which promises to make it far easier and faster to access primitive data types. This performance enhancement will be particularly welcome when accessing objects in arrays, as Ben Evans explains in his deep look inside the JVM.

Finally, the product management team for Java explains recent changes (page 52) to the licensing model. In addition, we have a deep dive into the decorator design pattern (page 67), our quiz (page 78), and our book review (page 8), as well as our calendar of upcoming developer conferences and events (page 9).

SIMON MAPLE

ANDREW BINSTOCK

The Largest Survey Ever of Java Developers

What 10,500 Java developers tell us about their projects, their tools, and themselves

The data presented in the following report was taken from more than 10,500 questionnaires. If you were one of those survey-takers, *many* thanks for putting aside the time to share your experience for the benefit of others. Snyk and *Java Magazine* conducted the survey by publishing its availability to the Java community at large, to Java user groups (JUGs), and to subscribers of *Java Magazine*. As an inducement to complete the survey, Snyk and *Java Magazine* promised a contribution to Devoxx4kids.

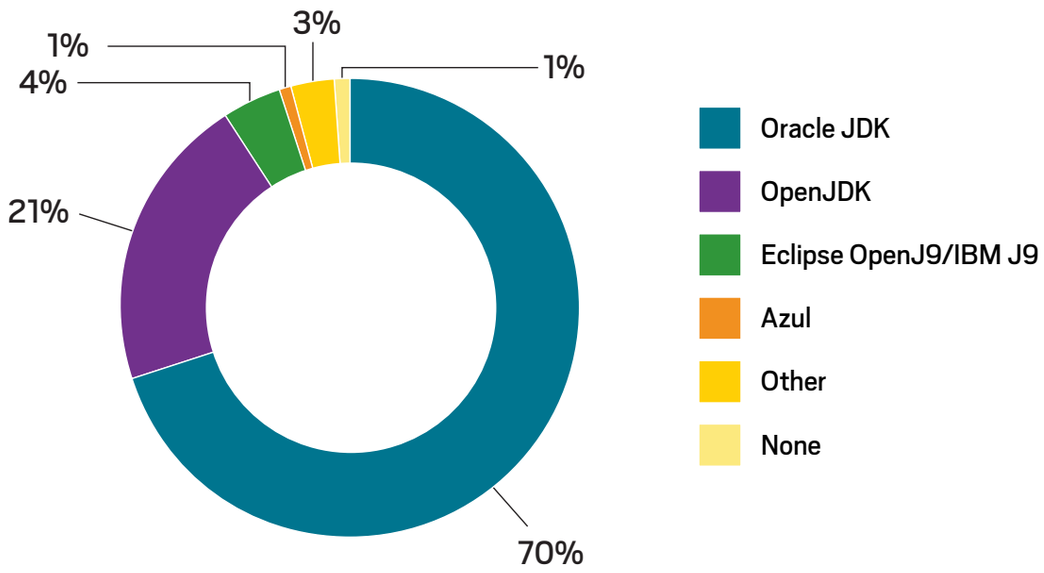
Information regarding respondents, including geographical location, company size of their employer (if any), age, and experience with Java are presented at the end of this survey.

About Your JDK

1. Which Java vendor's JDK do you use in production for your main applications?

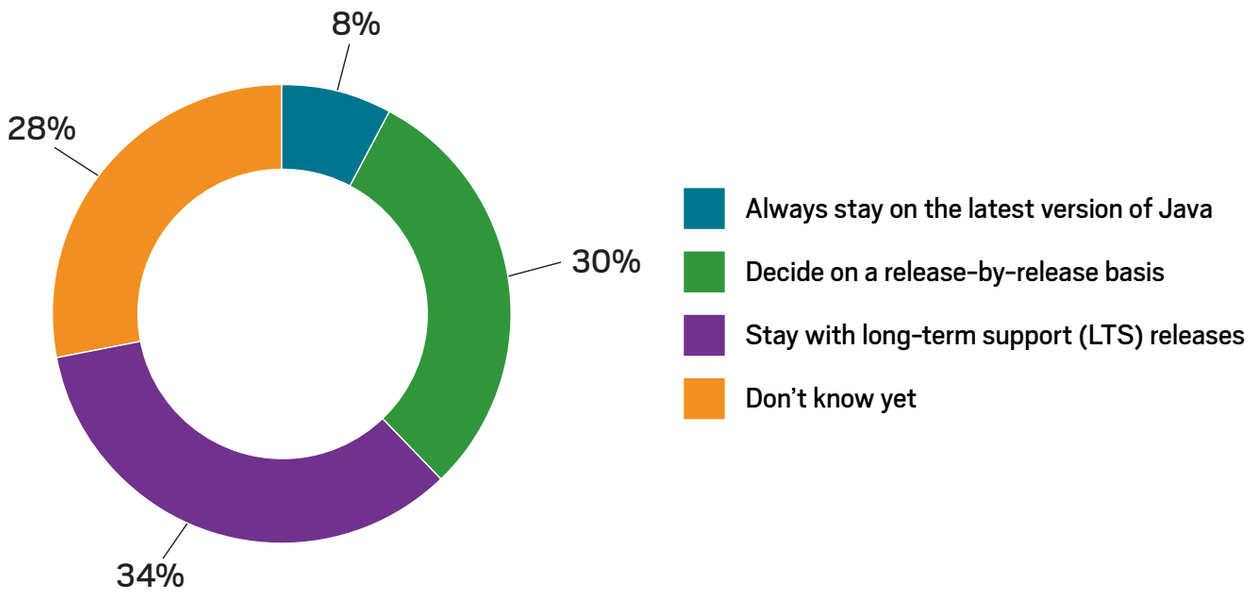
We start the report with a core question. With so many vendors providing their own JDK implementations, which offerings are developers using in production for their applications? We can see the dominance that Oracle JDK and OpenJDK have over everyone else. With 7 in 10 developers opting to use the Oracle JDK and a further 2 in 10 opting for the OpenJDK, there isn't much competition. However, future licensing and support changes might cause these numbers to differ in the future.

BINSTOCK PHOTOGRAPH BY BOB ADLER/STUDIO AT GETTY IMAGES FOR ORACLE



2. How do you plan to respond to Java's new release cycle?

While the Java 9 release brought with it some major architectural changes, it also introduced a new release cadence in which Java SE versions ship every six months. Every two to three years, a long-term support (LTS) release offers longer-term support, such as security updates. This question asks how development teams will respond to the new release cadence. The responses

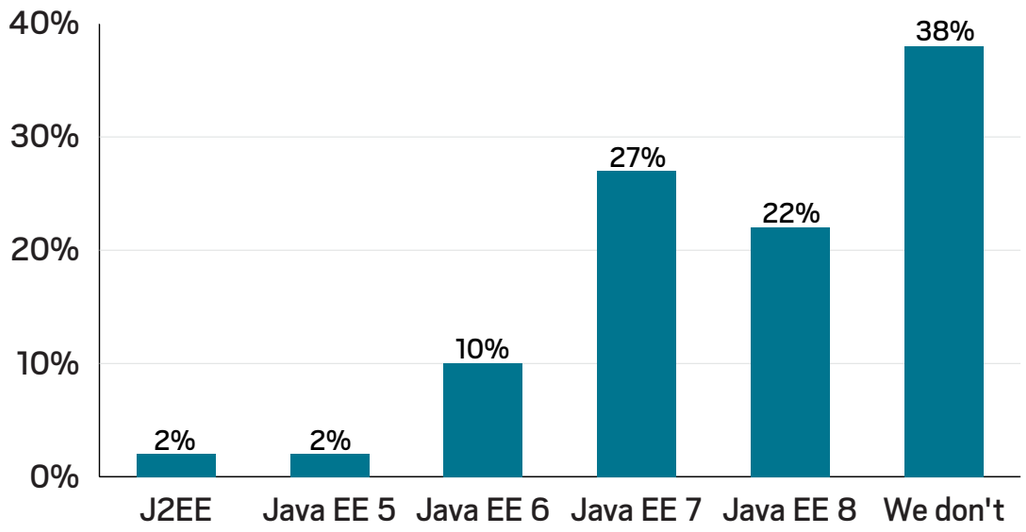


were varied, suggesting there is still some uncertainty about how to proceed. In fact, almost 1 in 3 developers don't yet know how they will respond to the new release cycle.

We expect that in the forthcoming year, best practices will emerge, and companies will settle into a preferred migration cycle, which likely will vary considerably by industry. As a result, we expect that the "Don't know yet" figure will drop, but we don't know which of the other buckets will see increases.

3. What Java EE version do you use for your main applications?

Almost 4 in 10 respondents do not use enterprise Java. Of those who do, the majority are on version 7. Java EE 8 was released in September 2017, so it's promising to see that within less than a year of release it is almost the most popular version. Unlike Java SE, a release of Java EE takes longer to be adopted, because it takes longer for implementations to become available. In addition, app server vendors require time to adopt and implement the specifications, so to see such user adoption after less than nine months is a promising sign.



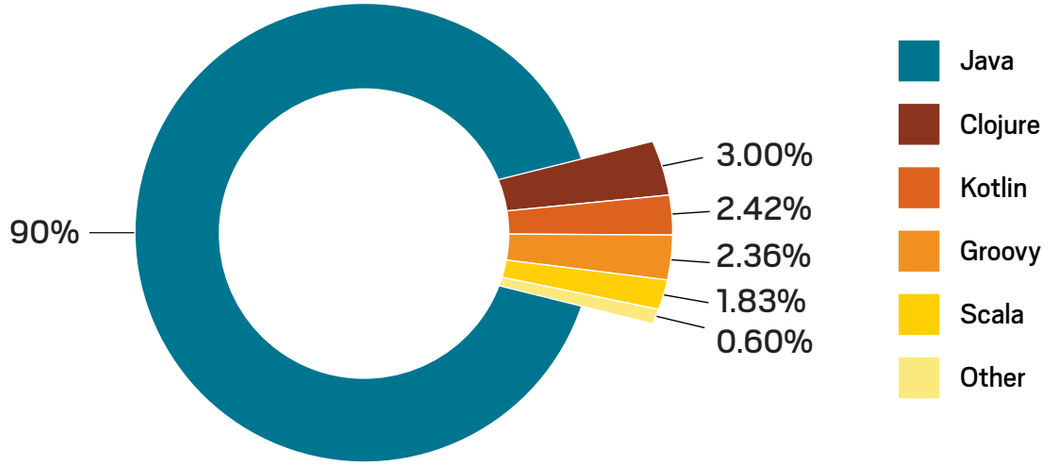
We'll be keen to see how these numbers shift as Jakarta EE 8 begins its rollout and adoption. We should spare a thought for the 2% who are struggling on J2EE—a version whose last major release was in 2003.

4. What is the principal JVM language you use for your main applications?

Exactly 9 in 10 JVM users are using Java for their main applications. While many projects today are defined as multilanguage or polyglot, the part on the JVM primarily runs Java.

Despite this strong preference for Java, JVM-based developers have consistently shown great interest in other JVM languages, as supported by the popularity of articles about them in *Java Magazine* and on major programming sites. For the last few years, the emerging and "hot" JVM language has been Kotlin, from JetBrains, which continues to make steady progress. It is now a supported language for development on Android, and it is the second major language for writing scripts for the build tool Gradle (behind Groovy). All this adoption has helped move Kotlin past Scala and just past Groovy in our survey.

The 3.0% figure for Clojure is remarkably high and signals—to us, at least—the continued interest in functional programming. The functional orientation of many Java 8 features shows the imprint of functional programming on Java itself.

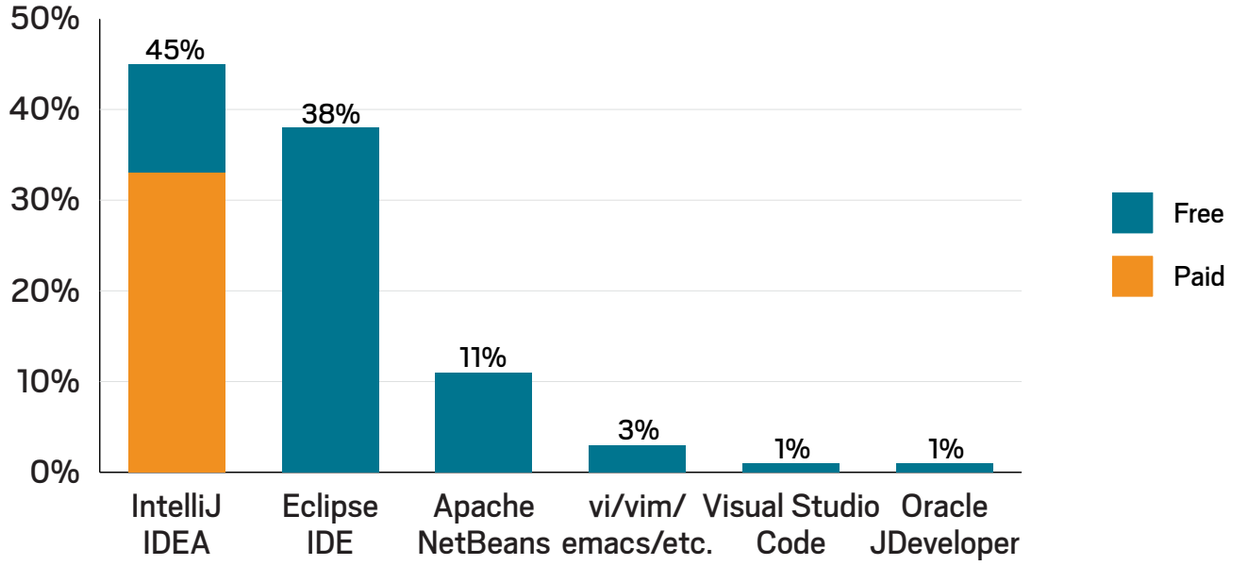


About Your Tools

5. Which IDE do you develop with?

The graph on the next page is consistent with other recent surveys: IntelliJ passing Eclipse in the last one to two years, and Apache NetBeans staying at around 10% of the market. The 45% total votes for IntelliJ consist of 32% IntelliJ IDEA Ultimate Edition (the paid version), 11% IntelliJ Community Edition (the free version), and 2% Android Studio users. The Eclipse category includes Eclipse STS, JBoss tools, Rational Application Developer, and other Eclipse-based tools.

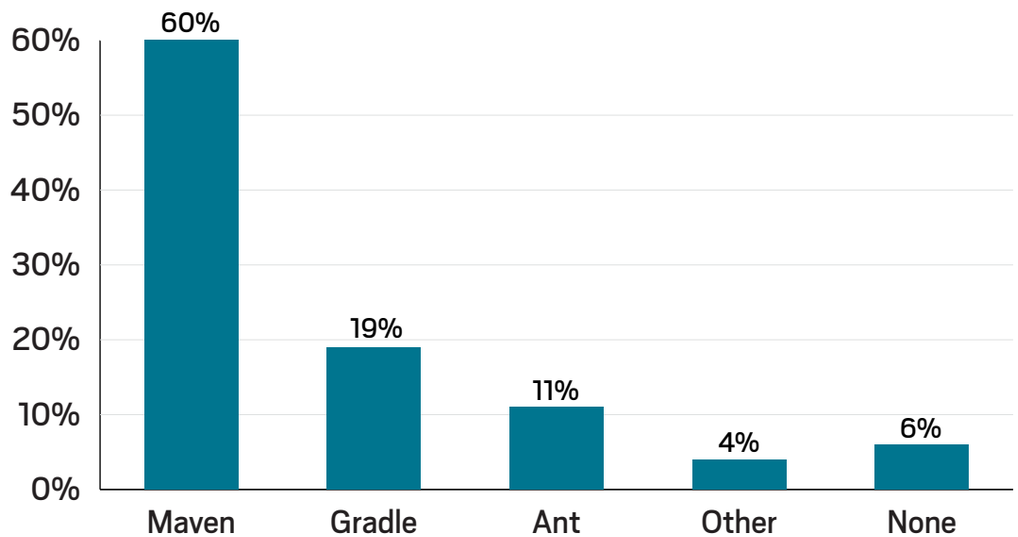
The numbers of Apache NetBeans users haven't changed much, which suggests that the migration from Oracle to the Apache Software Foundation hasn't affected its user base. It's also worth mentioning that Visual Studio Code has made an appearance, which, although it is at only 1%, shows it's starting to make its mark in the Java community. Also, a tip of the hat to the "vi/vim/emacs/etc." group, who are probably reading this report on a tablet (carved out of stone).



6. Which build tool do you use for your main project?

In examining the numbers, it's important to note that we asked for the *principal* build tool used. We want to know which build tool you rely on for your main project. It can also be interesting to know which build tools teams use across all their projects, but the plethora of

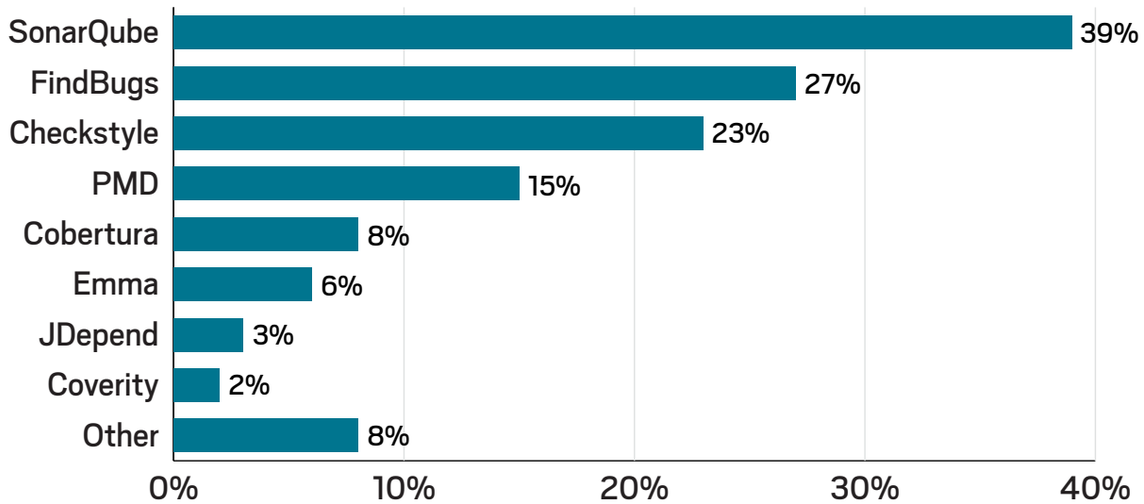
answers tends to dilute the usefulness of the responses. As we can see here, Maven clearly dominates with a 3:1 ratio over its closest rival, Gradle. Ant is still in use by 1 in 10 developers, whereas 1 in 20 use nothing!



In a 2016 version of a similar survey by RebelLabs (2,000 respondents), Maven stood at 68% and Gradle at 16%. Gradle's improved adoption rate is due perhaps to the addition of support for Kotlin as the scripting language. Gradle is also the default build engine for Android projects. However, its progress against Maven has been slow.

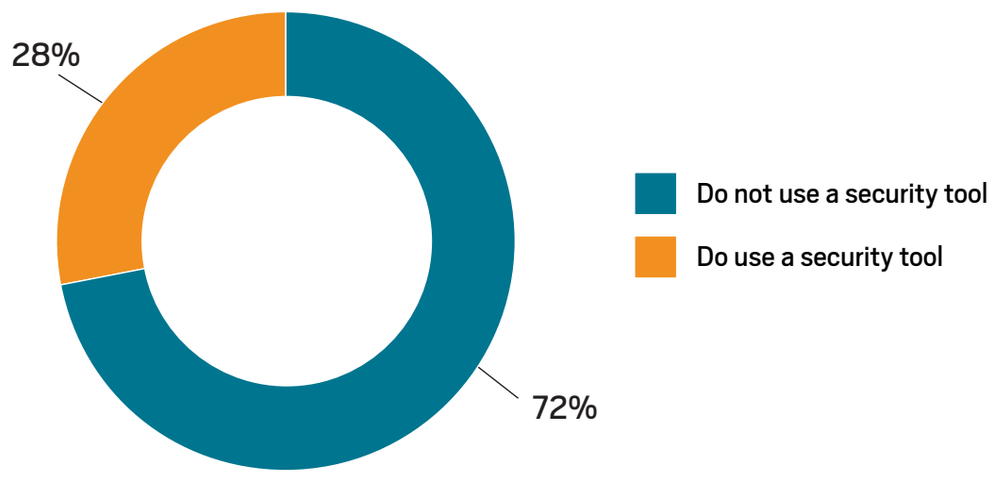
7. Which static quality tools do you use?

Before you send that email or tweet trying to fix the internet, note that this is a multiple-choice question, so numbers here do not add up to 100%. One takeaway from this data is that there are only a handful of static quality tools in popular use, with no real surprises at the top: SonarQube, FindBugs, and Checkstyle dominate. We were surprised to see that 36% of respondents don't use any static quality tool whatsoever. We assumed that the use of these tools was the norm.



8. Which static security tools do you use?

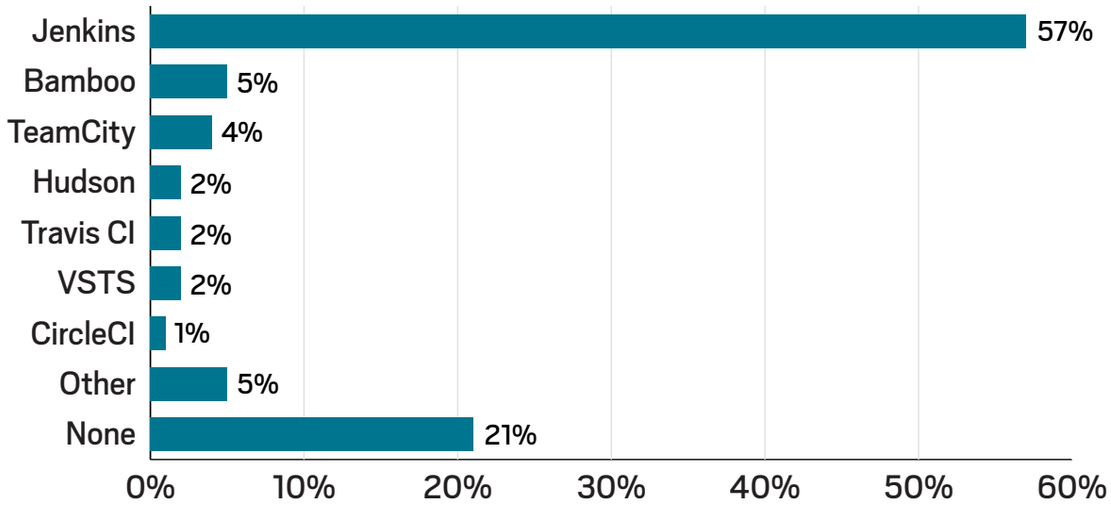
Most sites don't use static security tools. It's a surprise to see how low overall adoption is, considering the widely publicized costs of security issues. For sites that do use them, the older tools dominate: Sonatype and Fortify lead the market. Snyk makes its first appearance on our surveys and provides full automated remediation across many ecosystems. We hope a wider adoption of security tools will appear in future surveys.



9. Which CI server do you use?

As most developers would expect, Jenkins wins the CI server race with a whopping 57% market share. Its closest competitor is "None" at 21% of the vote, which almost matches the rest of the competition combined (at 22%). The remaining CI servers have less than 5% of the market share each, with Hudson, the elderly relative of Jenkins, weighing in at 2%. It's worth mentioning VSTS (Microsoft Visual Studio Team Server), which is not usually thought of in the Java/JVM space, clocks in at 2%.

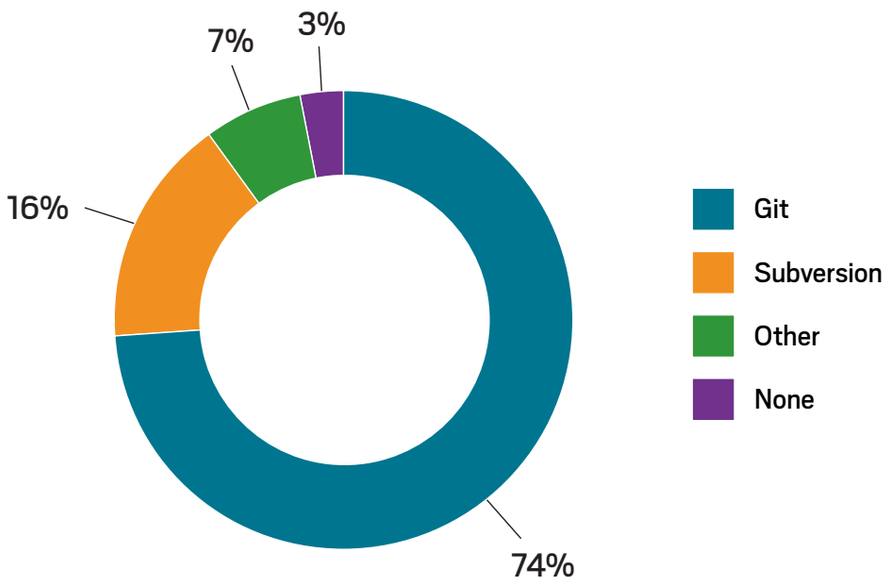
Most developers expect that nearly all sites today use continuous integration. So, it's startling to see that 1 in 5 applications rely on none at all. Even personal projects today use CI (such as Travis CI and CircleCI), which are made available on public project-hosting sites such as Bitbucket and GitHub. If you're one of the 21% who don't use CI on your projects, we'd love to hear why.



10. Which source code management platform does your team use for your main project?

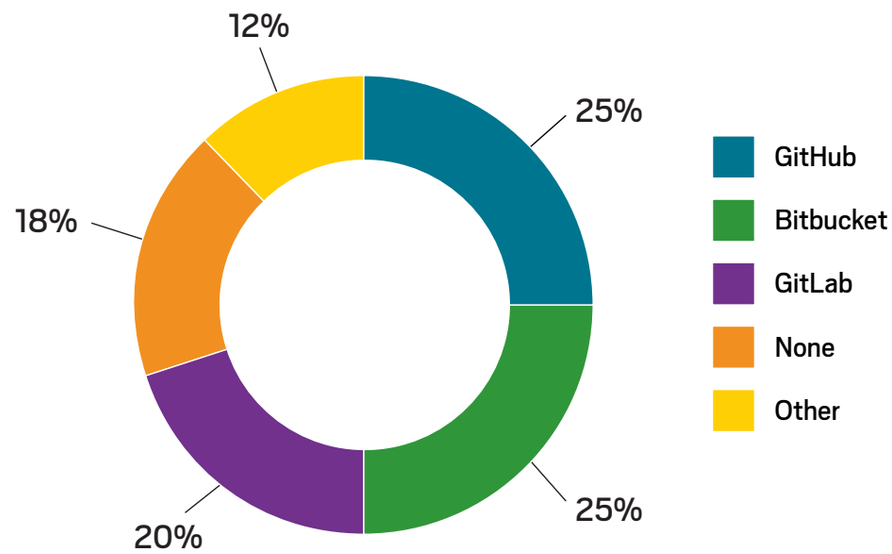
As expected, Git has convincingly won the horse race in source code management. If you were in any doubt as to the extent of its dominance, almost 3 in 4 of our respondents work in teams that use Git to manage their codebases. Subversion now covers the majority of the remaining

respondents; and somehow in 2018, 3% of people still don't use source code management whatsoever. Sometimes, there are no words.



11. Which code repository do you use for your main project?

With code repositories, the story is quite different from source code management: much more spread out, with GitHib and Bitbucket neck and neck at 25% each and GitLab close behind at 20%. We could call those the "big three" of project hosting. Note that this question is not just

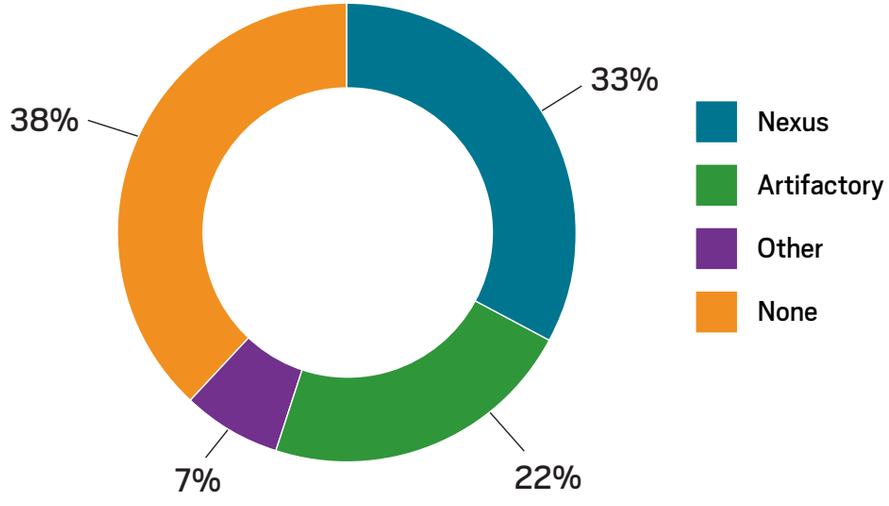


for public projects (in which we expect GitHub would have a more significant lead), but also for public and private project hosting.

Microsoft's recent acquisition of GitHub might affect its future adoption rate, and we'll know more in future surveys. Of the 25% share that GitHub has, just over half (52%) of those respondents are using the public version, whereas the remainder (48%) are using the private GitHub Enterprise on-premises offering. VSTS makes up part of the "Other" bucket with 2%.

12. Which private binary/artifact repository do you use?

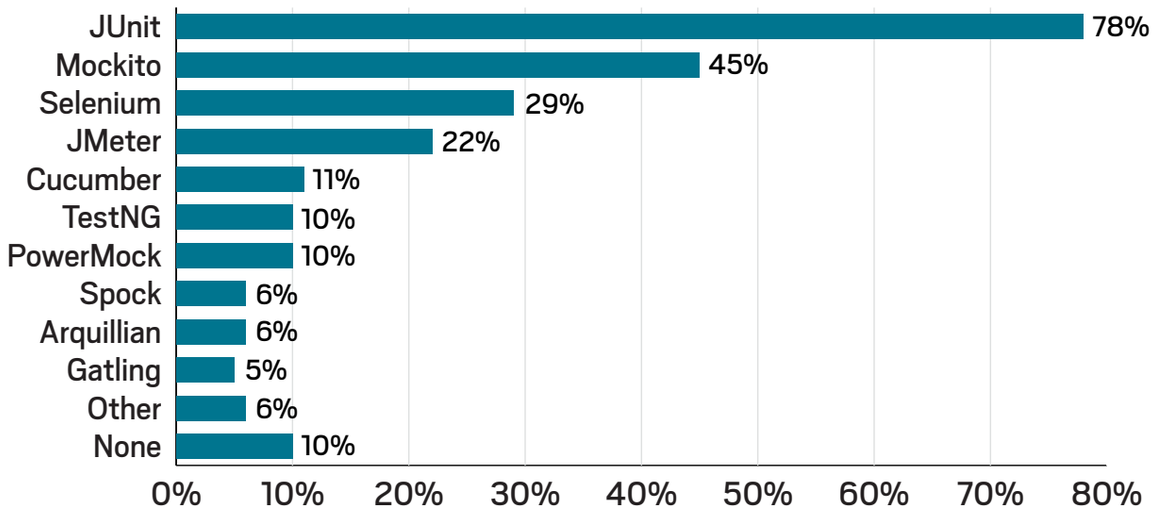
Most sites don't use a packaged artifact repository—in theory because they don't have the need. Those that like the convenience it offers choose the well-established Nexus, which is heavily focused on the JVM ecosystem, followed by JFrog's Artifactory, which is somewhat more popular across polyglot ecosystems.



13. Which testing technologies do you use?

With an amazing (almost) 4 in 5 people using JUnit and TestNG used by 10% more, it's clear that unit testing is by far the most dominant testing practice in the JVM ecosystem. (Respondents

could choose multiple answers, so totals exceed 100%.) As to mocking, it's now clear that Mockito has emerged as the preferred mocking framework.

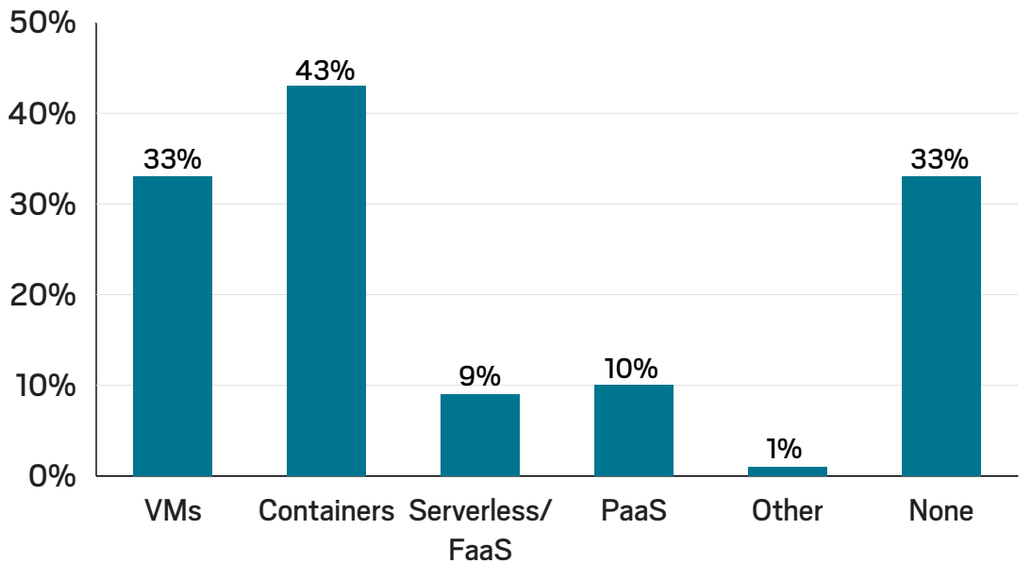


With JMeter being used by almost 1 in 4 respondents and Gatling by 5%, we can see that the need for performance testing is becoming better appreciated. Selenium takes an impressive 29%. Unlike the results for static quality tools, only 10% of respondents say they don't use any testing tool whatsoever. Hang on…1 in 10 people don't use a testing tool? We should move on before we lose faith in humanity.

About Your Cloud

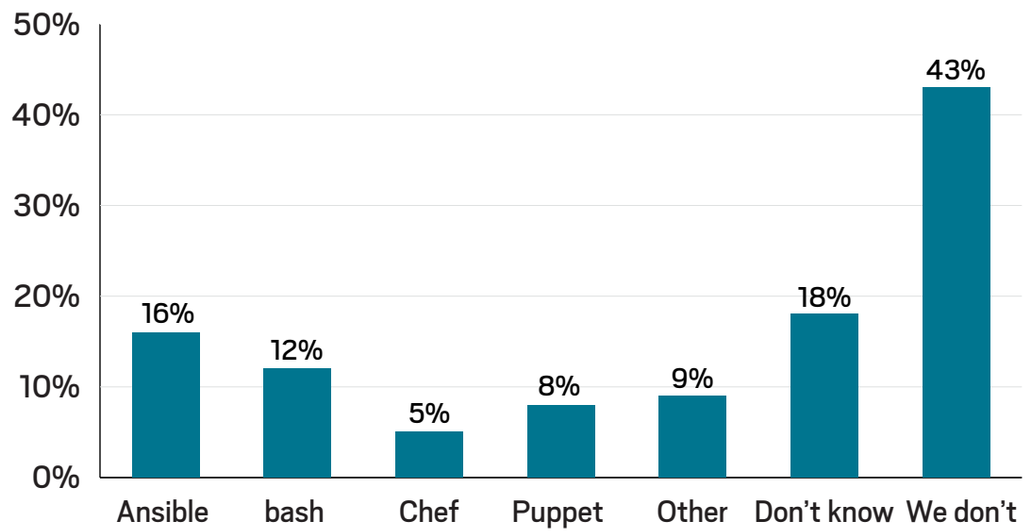
14. Which cloud approaches do you use?

Containers lead the way at 43%, while VMs stay in the game at 33%. As a relatively new technology, serverless/FaaS comes in strongly, with almost 1 in 10 respondents adopting this approach. PaaS, which has been around a lot longer, sits on a similar split at 10%. 1 in 3 respondents don't use any cloud approach at all.



15. Which continuous deployment or release automation tools do you use?

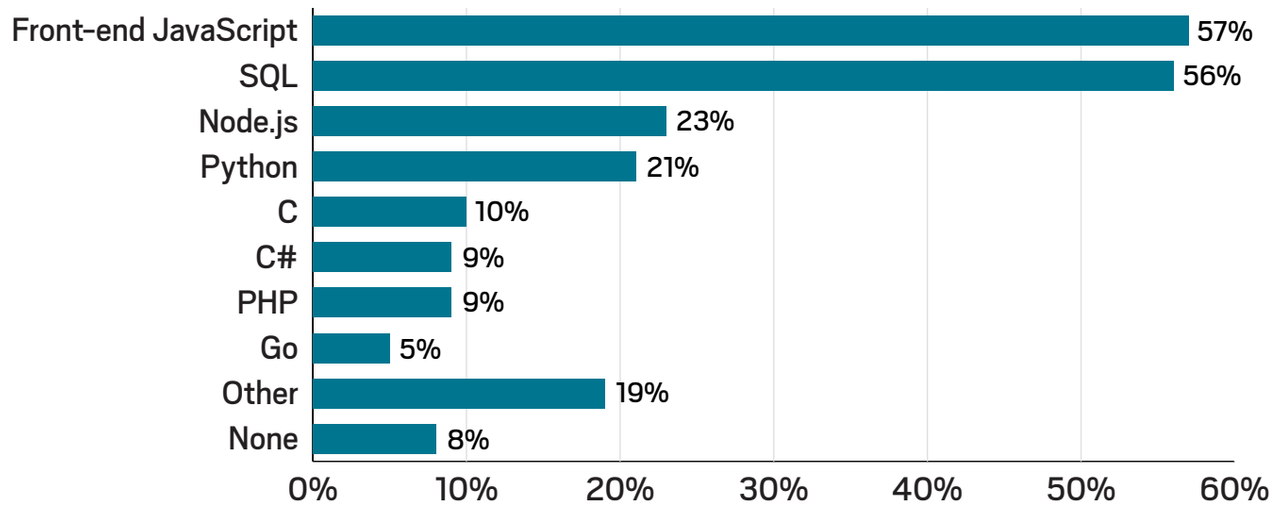
Nearly half the respondents don't use any continuous deployment or release automation tools whatsoever. This might even be higher, because almost 1 in 5 don't have any idea which tools are used in CD or release automation. It's somewhat surprising to see bash more popular than Chef and Puppet. Actually, who are we kidding? Everyone loves bash! Ansible is the leading CD tool at 16%.



About Your Application

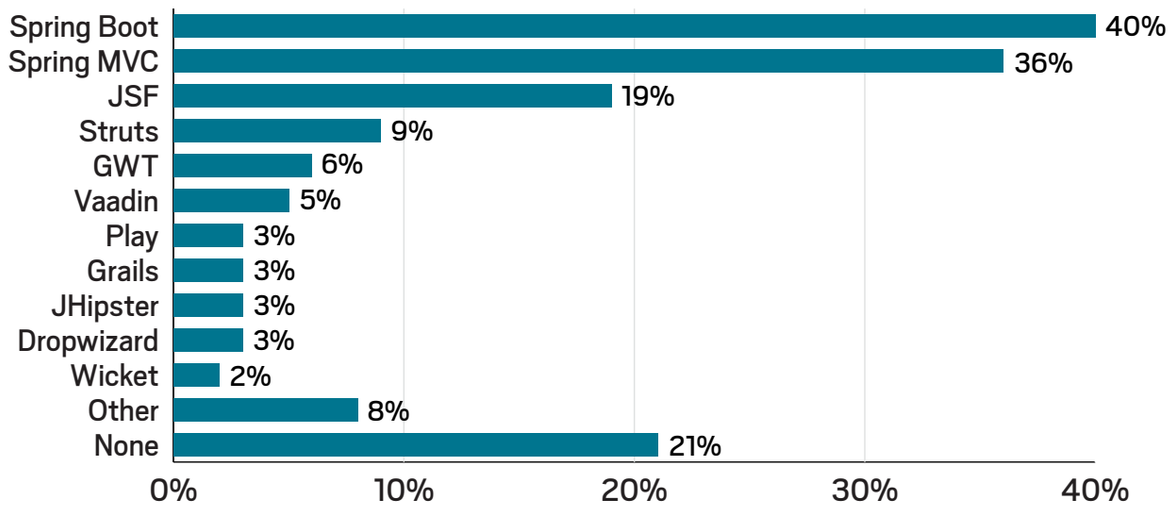
16. Which other (non-JVM) languages does your application use?

In today's polyglot world, it would be naive to assume that JVM languages are the only languages used in JVM apps. In fact, more than half of JVM applications use front-end JavaScript, 1 in 5 use Python, and almost 1 in 4 use Node.js. As you'd expect, many projects use SQL as well.



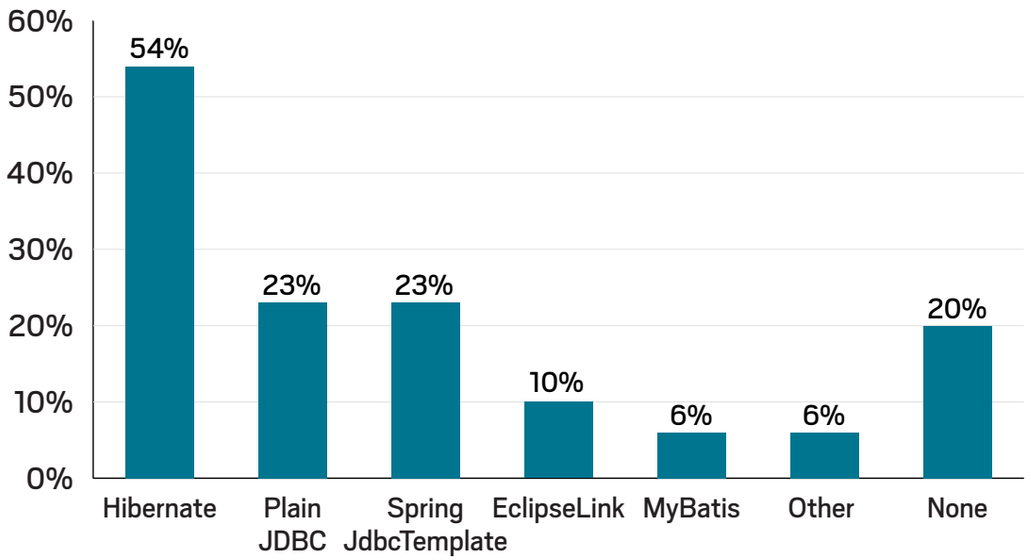
17. Which web frameworks do you use?

Few words can better express the Spring domination in the Java ecosystem than this graph. With 4 in 10 developers using Spring Boot in their applications, it's interesting to see it has overtaken the Spring MVC framework for the first time. JSF is the closest entrant with a respectable 19% and Struts, despite a constant stream of remote code-execution vulnerabilities in the news, is a strong fourth with almost 1 in 10 developers adopting it. More than 1 in 5 developers likely boast about how small their applications are, not needing a web framework at all.



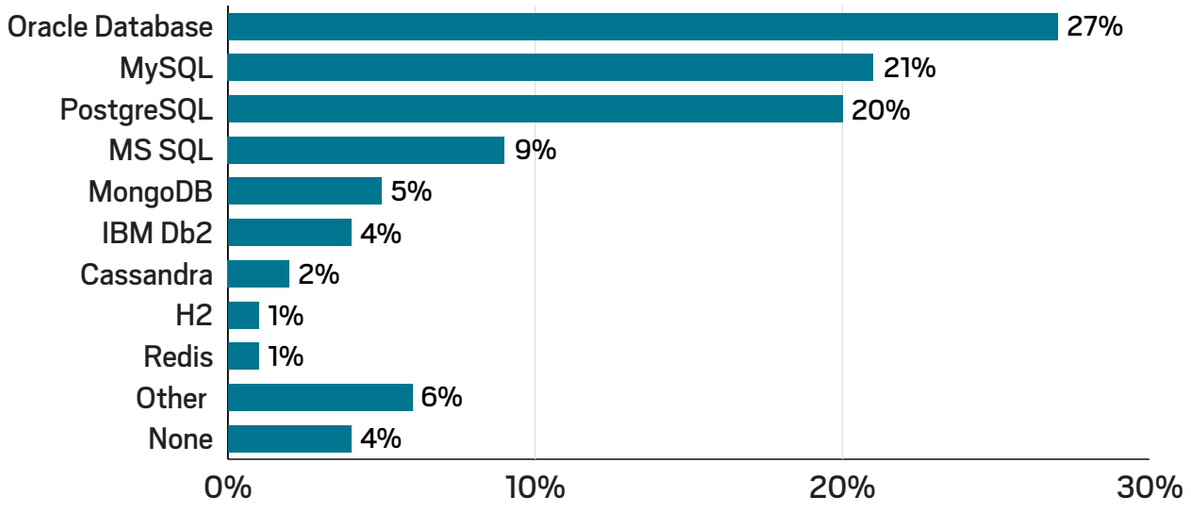
18. Which ORM frameworks do you use?

More than 1 in 2 developers use Hibernate in their applications. Almost 1 in 4 developers are happy with plain old JDBC, and Spring developers of course have the option of using Spring JdbcTemplate, which is used by 23%. 1 in 5 developers don't use any ORM framework whatsoever to access their data. (Developers could choose more than one answer, so totals do not equal 100%.)



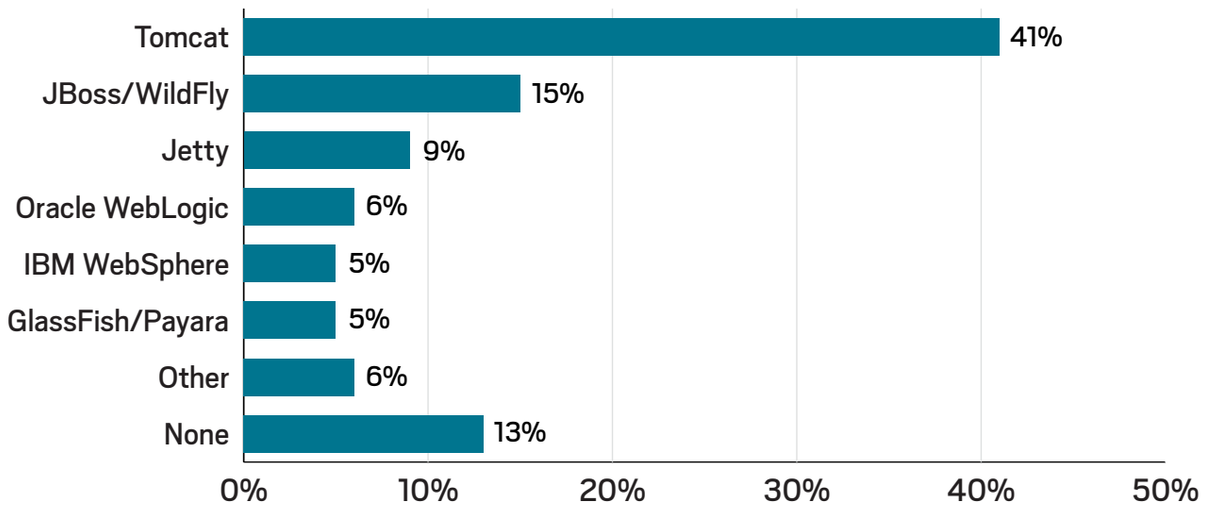
19. Which database do you use in production?

Once again, Oracle Database takes the top spot, with almost 3 in 10 applications using it in production. MySQL and PostgreSQL are strong competitors, taking 21% and 20%, respectively. MongoDB is the highest NoSQL database in use, with 5%.



20. Which application server do you use in production for your main application?

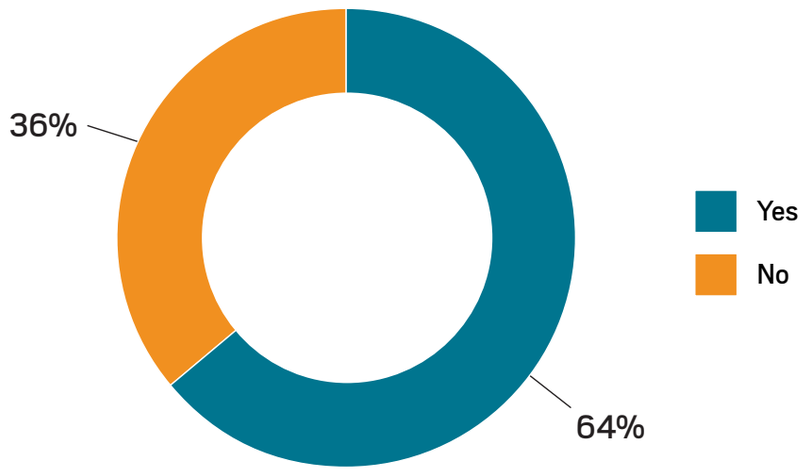
More than 4 in 10 respondents use Apache Tomcat as their application server of choice. The fast, lightweight, open source, community favorite has led the pack for a long time now, and it doesn't look like that's going to change any time soon. JBoss and WildFly are not too far behind



at 15%. In the larger enterprise app server category, Oracle WebLogic has a slight lead over WebSphere. The "Other" category contains Apache TomEE and Liberty Profile at 1% each, which lead that group.

21. Do you develop on the same application server you use in production?

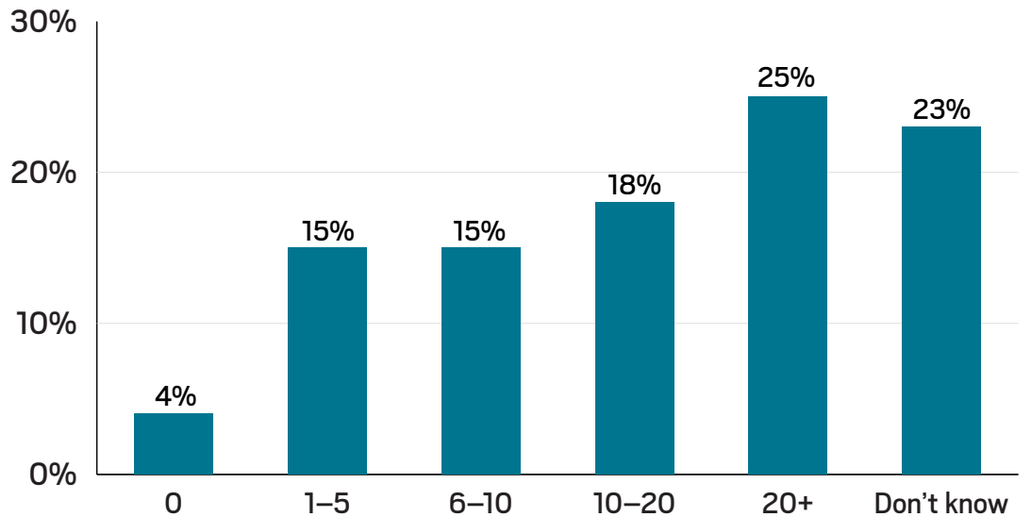
Despite the obvious dangers, more than one-third of respondents develop on a different server from the one they use in production—trading the possible cost of failures for the convenience. Surprisingly, those who state they use different application servers (or none) in development actually have a wide variety of apps and servers in production. We were expecting mostly the larger monolith-suited app servers that could cause developers pain to use locally, but the ratios were comparable.



22. How many open source (direct) dependencies does your main application have?

It would be interesting to know how many people had to check to see how many direct dependencies their application has. We'd bet it was the vast majority of you. It's a good thing we didn't ask for direct and transitive dependencies too! In fact, almost 1 in 4 respondents openly state they don't know how many dependencies they have. This might be because of the way the application is distributed across a more complex build system. We can see from the results that fewer than 1 in 20 respondents don't use any open source dependencies, whereas the over-

whelming 72% do. If we remove those who don't know, we can see that 95%, or 19 of 20 respondents, use open source dependencies in their applications. This shows how far open source adoption has come, as well as the need for us to ensure these third-party libraries provide the security, quality, and availability required from an application.

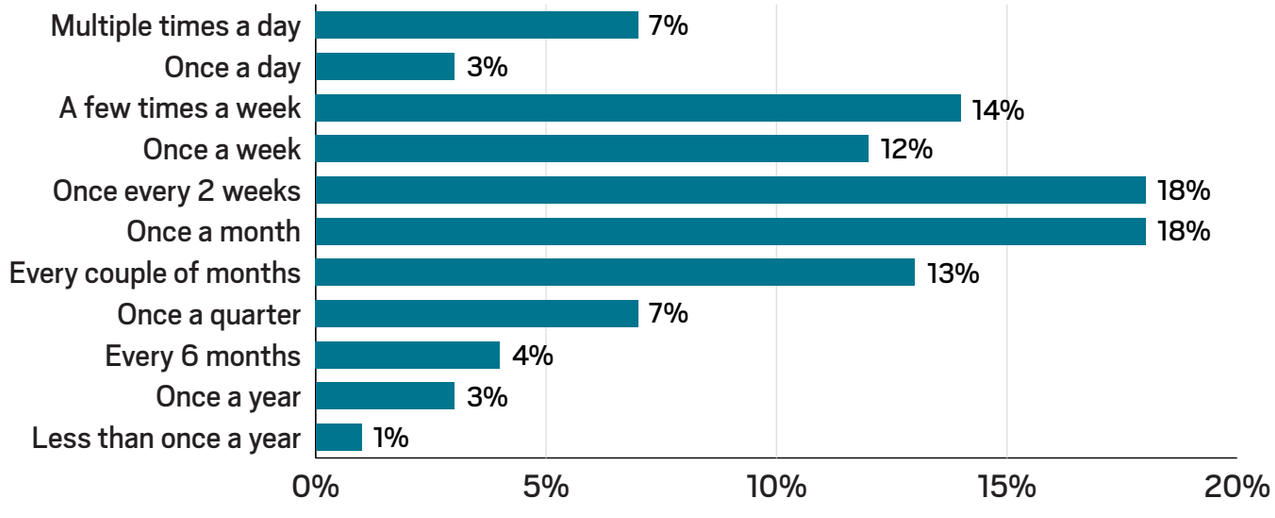


About Your Processes

23. How often do you release new versions of your code?

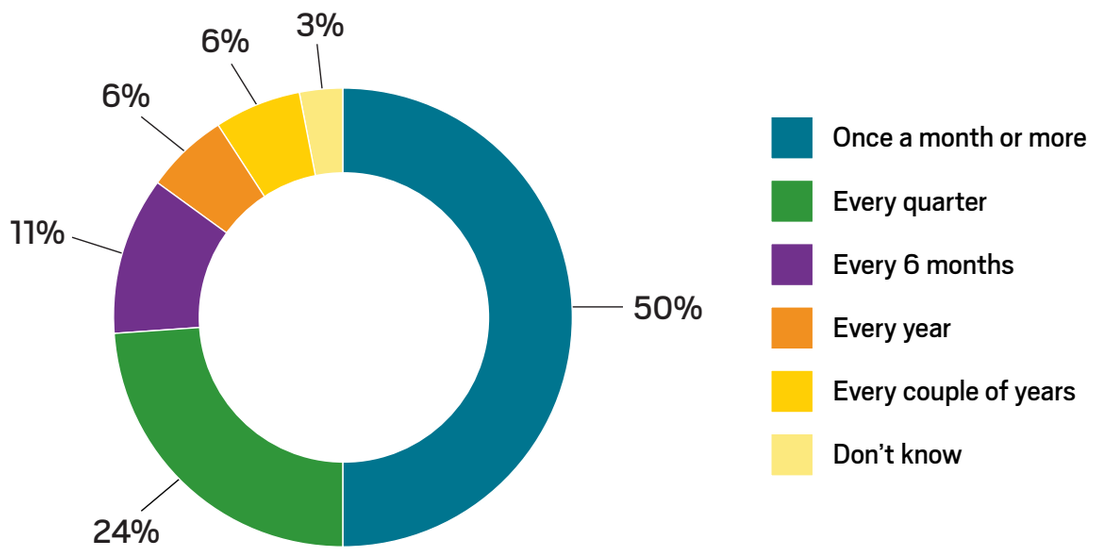
Not shown in this graph is that almost 1 in 4 respondents (24%) don't know how often their code is released. This again might be due to the complexity of the application, and perhaps because different services are being released at various times.

Almost 1 in 10 respondents are brave enough to release multiple times a day, but then again, when you release that often, bravery is replaced with consistency. The majority of respondents release once every couple of weeks to once a month. Fewer than 1 in 10 respondents release once every six months or less.



24. How often do you audit your code?

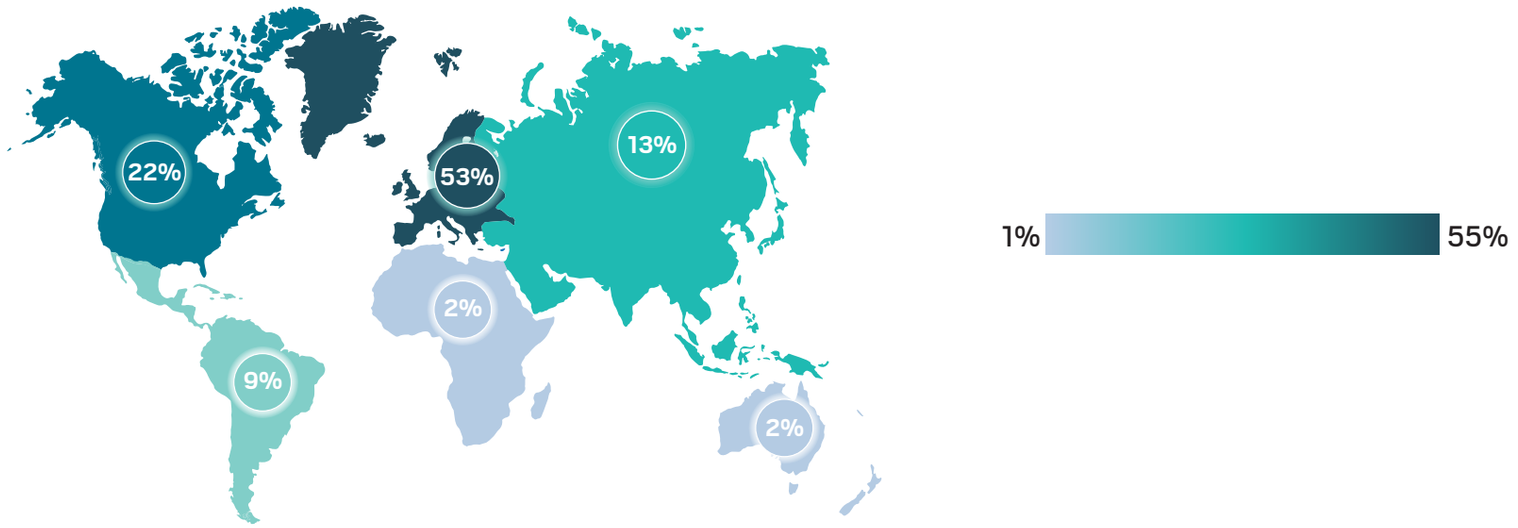
About half the sites audit their code. Only 1 in 4 do so more than once a quarter. Whether the audit is for security, performance, or quality, it's good to have a clear out. Half of the respondents do not audit their code whatsoever. Imagine the gremlins that could exist in those codebases!



About You

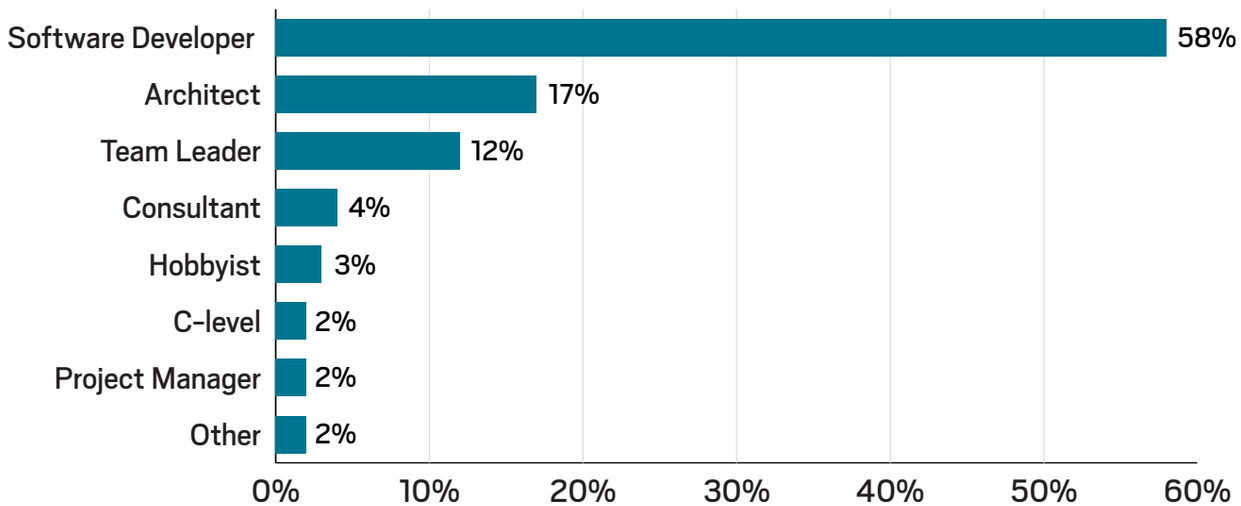
25. Where do you do your development work?

Not much to say here, other than we expected more respondents to come from North America.



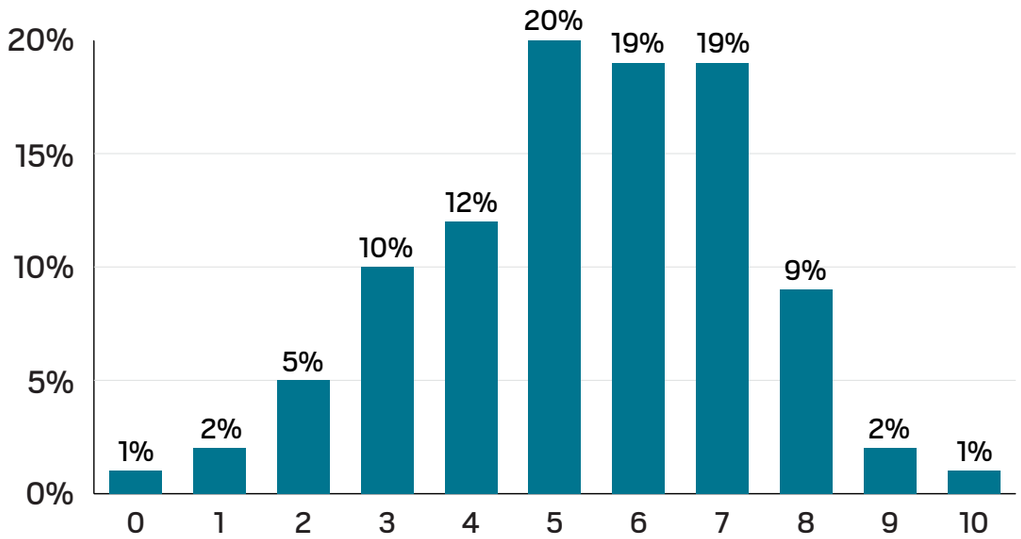
26. How would you describe yourself?

We can see the vast majority of respondents are technical, with 87% being either developers, team leaders, or architects. More than half state they are software developers, and 2% of respondents were C-level who took the time to fill out our survey.



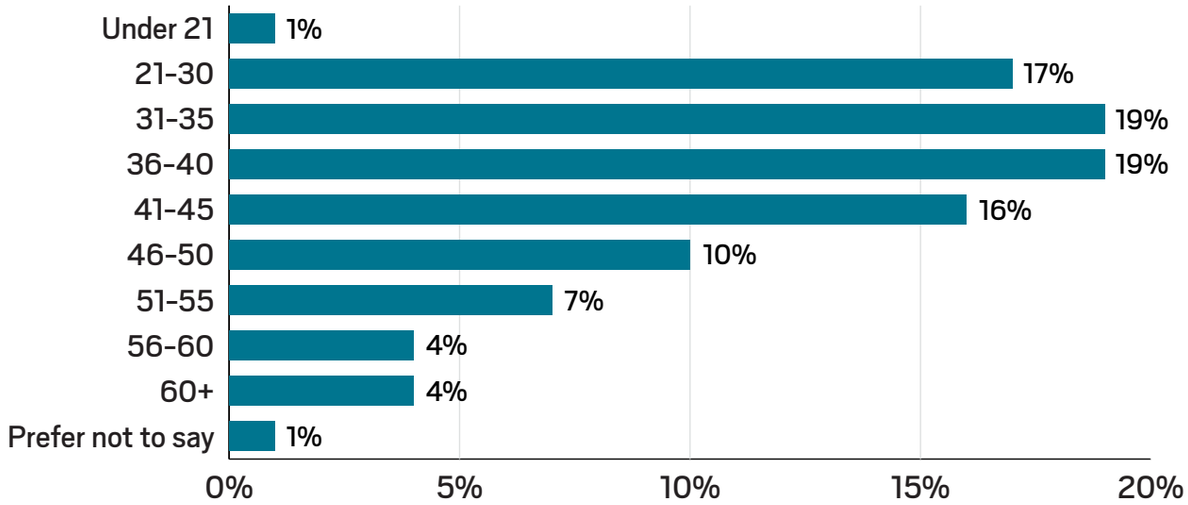
27. How do you rank your security expertise?

Security is often considered one of the dark arts. Many developers learn only "just enough" to get by and deliver their feature work, meaning the real experts are those who have a dedicated security career. As developers are owning more and more application security responsibility, this is becoming a hot topic. In our survey, 1% of respondents state they have zilch, none, zero security knowledge and are just happily writing their applications. The same number state they are true security experts and gave themselves 10 out of 10. The majority sit around the 5 to 7 mark, with 6 in 10 respondents stating they're no expert, but certainly not novices.



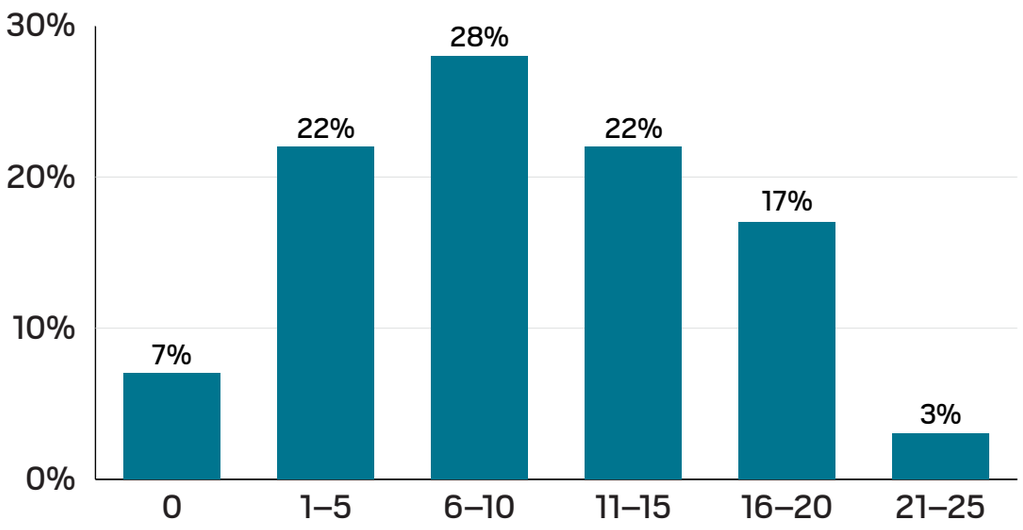
28. How old are you?

Programming remains a profession associated with the young and early middle-aged. 37% of survey respondents are younger than 35, 35% are between 35 and 45, and only 25% are older than that. Survey data shows a correlation between positions of greater responsibility and age, suggesting that the lower numbers after middle age are in part due to programmers moving into management positions.



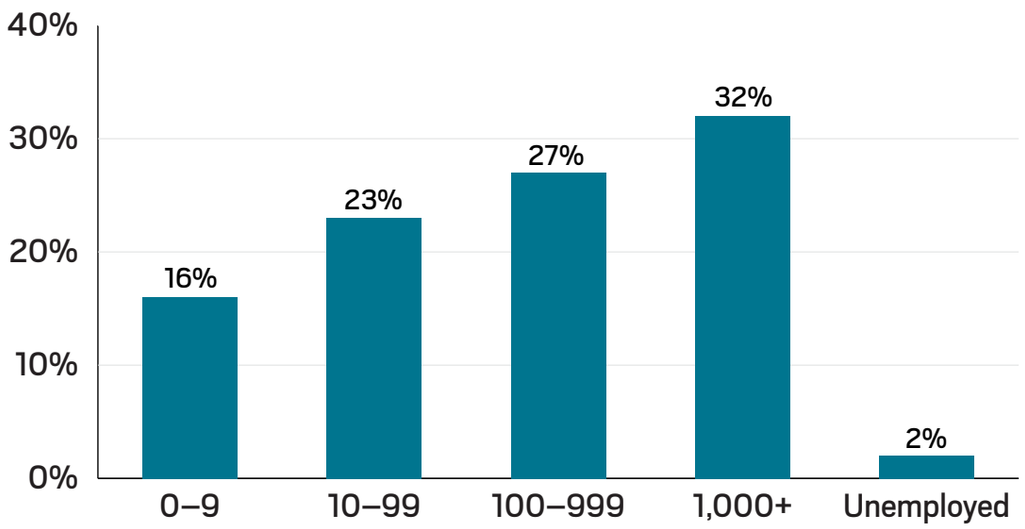
29. How many years of paid professional experience with Java do you have?

There was quite a range of experience among our respondents, as you can see from the graph, but we wanted to also look at the median ages of job roles back from question 26, to see when people typically receive promotions. The results are very interesting, with the median developer having 10 years of experience, team leaders 11 years, architects 14 years, and finally C-levels with 12 years.



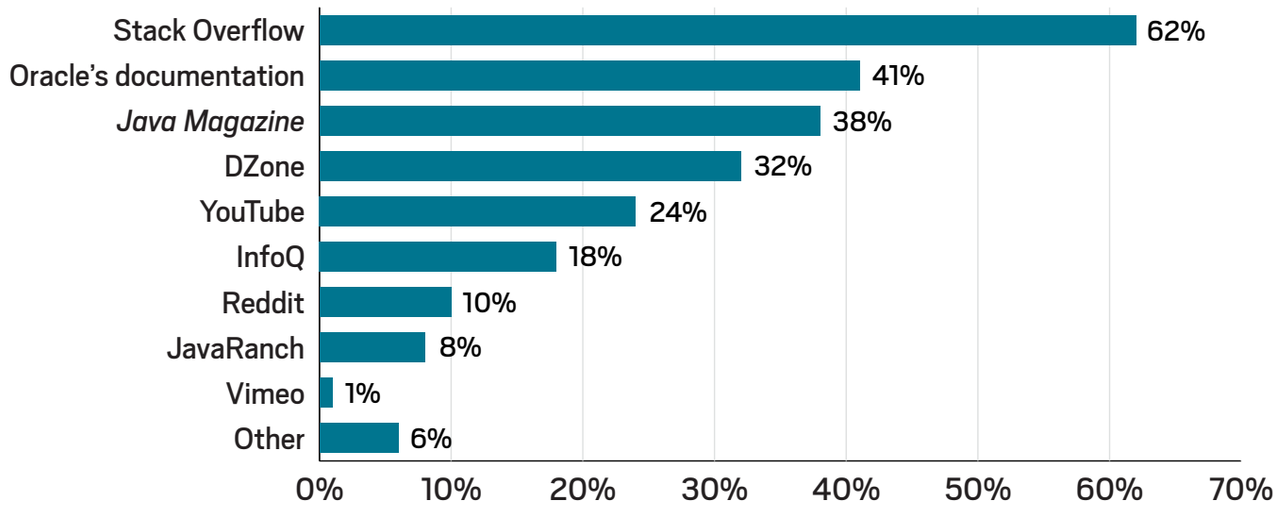
30. What is the size of your company?

With almost 40% of respondents working for companies that have fewer than 100 employees, we see that Java continues to have a significant role in startups and in small-to-medium businesses. This finding is at odds with the perception of Java being the language for enterprise apps. It is that, certainly, but definitely more than that.



31. Where do you principally get information about Java online?

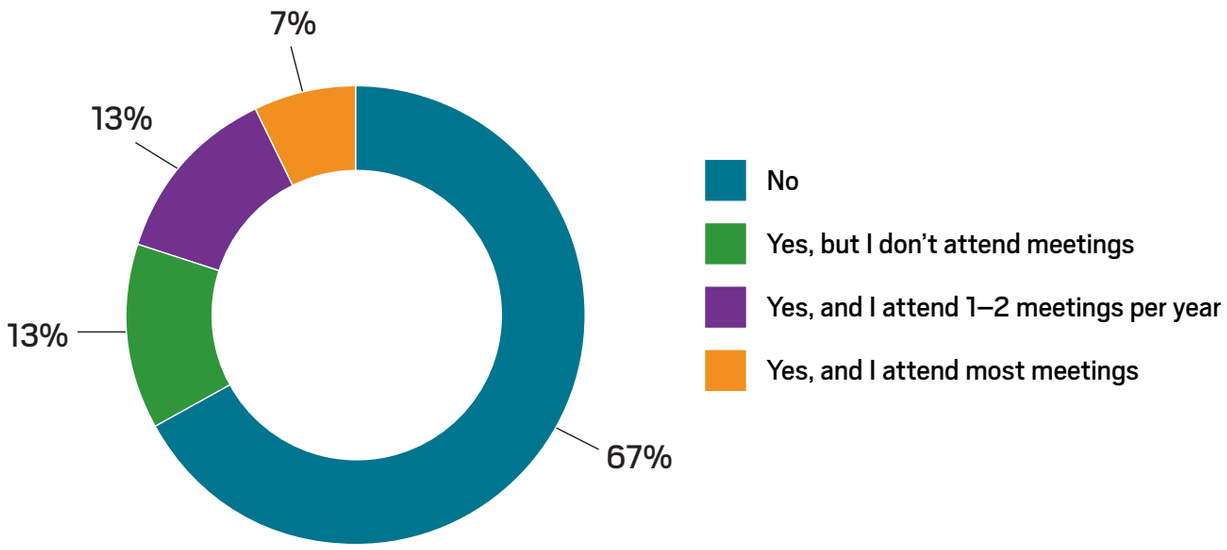
Stack Overflow remains the preferred forum for asking one-off questions and grepping replies to previous answers for useful information. Oracle's excellent documentation is a natural place



for reference. *Java Magazine* presents long-form, in-depth articles for readers wishing to fully understand a topic, and YouTube does the equivalent in video form. DZone and to a lesser extent InfoQ overlap all these areas. (Respondents could choose multiple answers.)

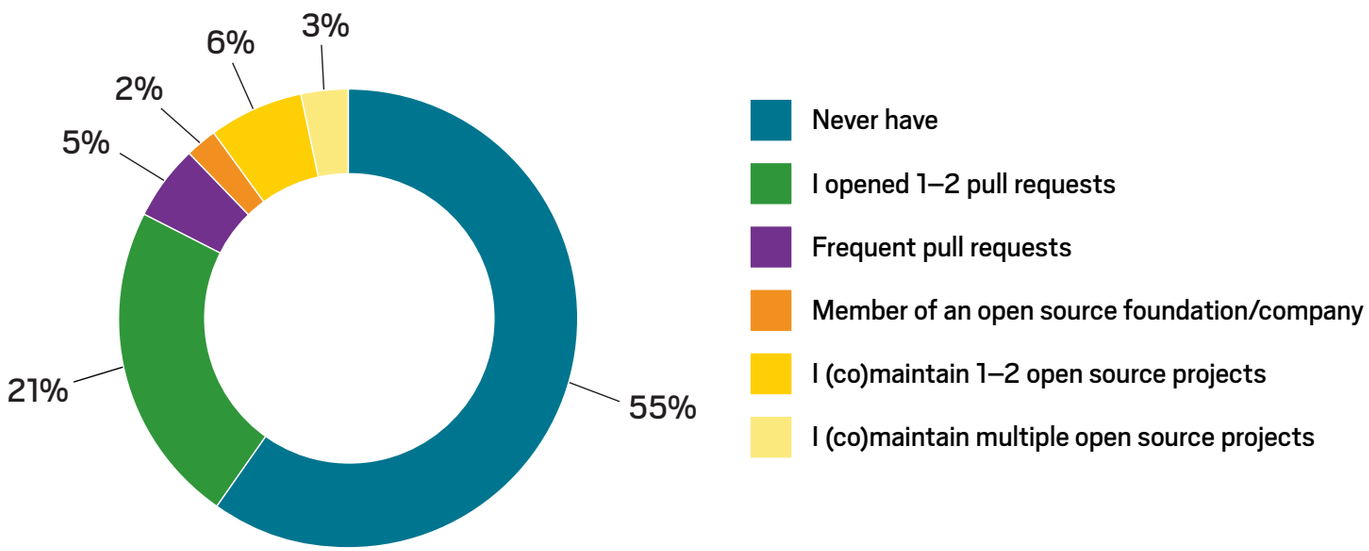
32. Are you a member of a Java user group (JUG)?

JUGs remain an underused resource in the community, with only 1 in 5 developers attending one meeting a year. The overwhelming two-thirds of respondents are not members of any JUG whatsoever. If geography is a factor for those developers, the Virtual Java User Group is an excellent solution.



33. How much do you contribute to open source?

Open source remains what it has primarily been from the start: the domain of a small minority of dedicated developers. The existence of GitHub and other easily accessible code repositories has helped developers to contribute their code, but with more than 1 in 2 developers never having contributed to an open source project, there's still a lot of work to be done. You should



want to contribute back so that rather than just being one of the 19 in 20 developers using open source, you can one day be one of the 19 in 20 developers contributing to open source. </article>

Simon Maple (@sjmaple) is a Java Champion and director of developer advocacy for Snyk, a company whose products find and fix vulnerabilities in project dependencies. He is also the founder of the Virtual Java User Group (vJUG). Earlier, he was director of developer relations at ZeroTurnaround. Previous to that, he held a similar role at IBM.

Andrew Binstock (@platypusguy) is the editor in chief of *Java Magazine*. Previously, he headed *Dr. Dobb's*, the renowned developer magazine and website. Before that, he was the editor in chief of *UNIX Review* magazine, and earlier of the *C Gazette*. He is the author of several books on programming and has contributed to open source projects since before the term existed.

RAOUL-GABRIEL URMA

RICHARD WARBURTON

What's New in JDK 11?

New features facilitate writing lambdas, ease HTTP and WebSocket communications, and help the JDK better handle inner classes.

JDK 11 appeared in late September of this year. It's the third delivered release using Oracle's new six-month cadence. This article covers the short list of principal features that you are likely to encounter in the new release. They are

- The new HTTP Client library
- Local variable syntax for lambda parameters
- Nests
- Removal of deprecated modules

Let's start with HTTP and WebSocket, because they're likely to get a lot of your attention.

New HTTP Client and WebSocket API

Hot topics such as asynchronous programming, functional-style programming, and reactive streams have been discussed in depth in previous *Java Magazine* articles. Their benefits are clear: you can write code that reads closer to the business problem statement. This ability gives you flexibility for requirements changes while also enabling some modern concurrency techniques. Why then is a simple HTTP request to a web service so cumbersome? And is it limited to blocking I/O?

For example, querying a website would be done as follows using the traditional `HttpURLConnection` API:

```
URL url = new URL("http://iteratorlearning.com");
HttpURLConnection httpURLConnection
    = (HttpURLConnection) url.openConnection();
```

```
httpURLConnection.setRequestMethod("GET");

try (BufferedReader in = new BufferedReader(new
InputStreamReader(httpURLConnection.getInputStream()))) {
    String response = in.lines().collect(Collectors.joining());
    System.out.println(response);
}
```

This example is very simple; the Stream API makes it more pleasant to read and work with. However, this API has some deficiencies. It's not straightforward to use for handling common requirements such as managing time-outs, cookies, sessions, and HTTP request parameters or working with proxies.

In addition, the old `HttpURLConnection` API has design problems:

- It is verbose to use, and simple requests require a lot of setup.
- It supports only blocking I/O. In other words, the thread executing the code must wait for the I/O to complete, and that thread cannot be used to run other tasks.
- It doesn't embrace the new functional-style programming that was delivered in Java 8.
- It is based on configurations using "setter" methods that partially change the state of an object. Modern APIs have adopted a chainable builder pattern to configure complex objects, which produces code that is easier to follow.

Most of these limitations have been fixed in JDK 11. In fact, the new HTTP Client library was introduced in JDK 9 as an incubator project. This means that the library got an opportunity to be polished and tested before its official integration in JDK 11.

The new library supports nonblocking I/O and data types such as `CompletableFuture`, `Optional`, and the new Date and Time APIs that were introduced in Java 8.

The API consists of four main classes and interfaces:

- `HttpClient`: The entry point for using the API
- `HttpRequest`: A request to be sent via the `HttpClient`

- `HttpResponse`: The result of an `HttpRequest` call
- `WebSocket`: The entry point for setting up a WebSocket client

Here's how you can rewrite the previous example to make a simple request:

```
var httpClient = HttpClient.newHttpClient();
var request
    = HttpRequest
        .newBuilder(URI.create("http://iteratorlearning.com"))
        .build();
HttpResponse<String> response
    = httpClient.send(request,
                      HttpResponse.BodyHandlers.ofString());
System.out.println(response.body());
```

You have to admit this code is immediately readable. The new HTTP Client API supports non-blocking I/O via the `sendAsync` method, which returns a `CompletableFuture`. You can use it as follows:

```
var httpClient = HttpClient.newHttpClient();
var request
    = HttpRequest
        .newBuilder(URI.create("http://iteratorlearning.com"))
        .build();
httpClient.sendAsync(request, HttpResponse.BodyHandlers.ofString())
        .thenApply(HttpResponse::body)
        .thenAccept(System.out::println);
```

If you want to send a POST request, you can use the `POST` method available in the API. For example, the following code creates a POST request consisting of a JSON payload read from the file `data.json` and sent to the `/post` endpoint of `http://httpbin.org`:

```
var postRequest
    = HttpRequest.newBuilder()
       .uri(URI.create("https://httpbin.org/post"))
       .header("Content-Type", "application/json")
       .POST(HttpRequest.BodyPublishers.ofFile(
            Paths.get("data.json"))
       )
       .build();
```

You've seen a couple of simple examples that showcase the basic use and principles behind the new HTTP Client API. There's a lot more to it, including support for WebSockets as well as adapters to connect with the reactive stream protocol standardized in Java 9.

Local Variable Syntax for Lambda Parameters

Java 10 introduced local variable type inference, which was discussed earlier in this *Java Magazine* article. Local variable type inference enables you to do the following:

```
var url = new URL("http://iteratorlearning.com");
```

Some additional examples of the use of `var` appear in the previous section. Java 11 extends this syntax to the arguments of lambda parameters. Since Java 8, you can declare a lambda expression this way:

```
FilenameFilter xmlTargetFiles
    = (dir, fileName) -> fileName.endsWith(".xml")
                      && "target".equals(dir.getName());
```

Here, the types of the two arguments, `dir` and `fileName`, are inferred. You don't need to explicitly declare them. The types are inferred to be `File` and `String`, respectively.

In Java 11, you can rewrite this lambda expression as follows:

```
FilenameFilter xmlTargetFiles
    = (var dir, var fileName) -> fileName.endsWith(".xml")
                              && "target".equals(dir.getName());
```

What are the benefits of writing a couple of additional new characters?

There are two. First, since you can already use the `var` type inference syntax with local variables as of Java 10, why not use `var` with lambda parameters as well? Doing so provides consistency in the Java language. Second, this syntax enables you to use type annotations such as `@NonNull`. These annotations can be helpful for documentation and also for type checking via the Checker Framework. To find out more about how you can leverage such annotations generally to catch more bugs in your application at compile time, see the Checker Framework's website.

Here's the lambda expression example using `var` and also annotated:

```
FilenameFilter xmlTargetFiles
    = (@NonNull var dir, @NonNull var fileName)
          -> fileName.endsWith(".xml")
          && "target".equals(dir.getName());
```

This syntax is not supported in implicitly typed parameters that don't include `var`. Allowing modifiers on a bare identifier would add significant complexity to the grammar of the Java language, with little advantage.

Note that the introduction of the `var` syntax for lambda parameters does not allow you to mix and match typed parameters and `var` parameters. For example, the following is forbidden and results in a compile error:

```
// compile error
FilenameFilter xmlTargetFiles
      = (var dir, String fileName) -> fileName.endsWith(".xml")
                                   && "target".equals(dir.getName());
```

In fact, allowing this mix would require adapting the type-checking and overload-resolution rules. However, this mix is something that might be considered in a future Java version. The benefit of this restriction for now is that it enforces a consistent style and raises fewer eyebrows when you look at code.

Nests

Nests are filled with inner classes rather than birds. Let's look at what nests are and what problem they solve.

For a long time, Java supported the concept of an inner class—that is, a class that's contained within another class. Inner classes may be anonymous, which means they do not have a name. Inner classes were pretty useful before lambda expressions replaced some of them by defining simple bodies of behavior.

The concept of a nest applies only to classes syntactically nested within each other. It doesn't apply to different classes within the same source file.

Let's take a look at a simple example of an inner class. Suppose you want to loop over the characters in a String in the same way that you can with an array of `char` values. To implement this, you need to write a class, `StringIterable`, that adapts the String class into the `Iterable` interface that the `for each` loop uses, so you can write code like this:

```
for (Character character : new StringIterable("abc")) {
    System.out.println(character);
}
```

Internally, your `StringIterable` class has an `iterator()` method that returns a new `Iterator` object that iterates over the `char` values in the String. You implement this `StringIterator` class as an inner class—for example:

```
public class StringIterable implements Iterable<Character> {

    private final String string;

    public StringIterable(String string) {
        this.string = string;
    }

    @Override
    public Iterator<Character> iterator() {
        return new StringIterator();
    }

    private class StringIterator implements Iterator<Character> {

        private int index = 0;

        @Override
        public boolean hasNext() {
            return index < string.length(); // 1
        }

        @Override
        public Character next() {
            final char c = string.charAt(index); // 2
            index++;
            return c;
        }
    }
}
```

In this code example, your `StringIterator` class is a private class, so only the `StringIterable` class should be able to access it. Also note that `StringIterator` has access to the private `string` field (marked by comments "1" and "2" in the preceding example) of the `StringIterator` class. This combination of classes nested within a single source file (`StringIterable` and `StringIterator`) is called a *nest*. Each of the classes individually is called a *nestmate*. It's worth noting that the concept of a nest applies only to classes syntactically nested within each other. It doesn't apply to different classes within the same source file.

What problem do nests solve? The historical problem with nests in Java is that their implementation damages encapsulation, confuses tooling by misrepresenting the nest, and bloats the size of class files by generating extra methods.

If you decompile the generated bytecode for the previous example by using javap on Java 10 or earlier versions, you can see how nests are implemented historically. The `StringIterator` class file is generated with the name `StringIterable$StringIterator`, indicating that it's within the `StringIterable` class. To access the private `string` field of `StringIterable` that its `next()` and `hasNext()` methods use, `StringIterator` takes a constructor parameter with the `StringIterable` object as a parameter:

```
StringIterable$StringIterator(StringIterable, StringIterable$1);
```

It also has an internal field that holds a reference to the `StringIterable` object:

```
final StringIterable this$0;
```

So far, so good. The problem comes when you want to read from the private `string` field. The code within `StringIterator` can't just refer to `this$0.string`, because `string` is a private field and it lives in another class. To work around this problem, javac generates what is formally called a *synthetic accessor* or *bridge method*, which is a bit of a hack:

```
static java.lang.String access$100(StringIterable);
  Code:
    0: aload_0
    1: getfield #1 // Field string:Ljava/lang/String;
    4: areturn
```

This `access$100` method exists only to expose the private field `string` to the outside world. That breaks encapsulation, because other classes can call this method via reflection, effectively turning a private field into a package-scoped field. An `ACC_SYNTHETIC` flag is set for this method to indicate that it's synthetic, that is, it's generated by javac. This compilation strategy also introduces an asymmetry between source code and reflection code because in source code, you can read out the private `string` field, but you can't do the same with reflection without using the `setAccessible()` method.

From Java 11 onward, the semantics at the bytecode level are much closer to the source code level when nested classes are compiled.

The reason for this bridge method is that `StringIterable` and `StringIterable$StringIterator` were different classes in this compilation approach. The JVM didn't know about the relationship between them and, thus, didn't know what the access rules were between the two classes. As a result, the bridge method was generated to expose the private field from where it needs to be accessed.

If you compiled the same source code with Java 11, this method would disappear. That's because the concept of a nest and nestmates is being explicitly introduced to the JVM, which enables reflection to work, smaller class files, and proper encapsulation. The private `this$0` field remains, however.

There is also arguably a performance improvement in these situations because under the old compilation scheme, every access of a supposedly private field from a nestmate would need to go through these bridge methods. Now, sometimes the JIT compiler may optimize away this

method call and sometimes not. Either way, Java 11 removes these bridge methods and resolves this performance issue.

From Java 11 onward, the semantics at the bytecode level are much closer to the source code level when nested classes are compiled. Within the whole nest, private access for methods and fields is complete and undifferentiated between any nestmates. This is codified in Java 11's updated JVM specification, which requires JVM vendors to support the nest access rules.

It's not just magic, though. To support the access rules, javac must be able to communicate to the JVM which classes are part of what nests. This is done via class file annotations. The top-level class file, `StringIterable` in the example, is called the *nest host*. It has an attribute added to it called *NestMembers* that lists all the other classes in the nest. In this example, the only other class would be `StringIterator`. These other nestmates have an attribute, *NestHost*, that declares the host class of the nest. To expose these properties via the reflection API, the methods `getNestHost()`, `getNestMembers()`, and `isNestmateOf()` have been added to `java.lang.Class`.

If you're interested in more information about nests and nestmates, see JEP 181.

So far, you've seen how nestmates tidied up a rough edge of the Java language and bytecode specification, but they do more than that: they provide a glimpse into future improvements to the JVM. The idea of a nest likely will be used by generic specialization from project Valhalla, replacing the deprecated `Unsafe.defineAnonymousClass()` API and "sealed" classes that can't be extended.

Removing Modules

Java has had a long history of deprecation. It added the `@Deprecated` annotation in Java 5. What *deprecated* means has evolved over time. Originally, the idea of deprecating something in Java meant that it was intended to be removed in a future version. However, deprecated methods have tended to not be removed. For example, several methods from the Abstract Window Toolkit (AWT) were deprecated in Java 1.1 with the intention of removing them in Java 1.2, but they weren't removed in that or subsequent releases. With Java 11, however, deprecation gains some teeth—in fact, several Java modules have been removed.

Since Java 6, the Java EE technologies JAX-WS, JAXB, JavaBeans Activation Framework (JAF), and Common Annotations framework have been included with the JDK. The original goal of this inclusion was to make it easier for developers to get up and running out of the box with a web development stack. In practice, however, these technologies can be easily obtained by developers who are using build systems such as Maven and Gradle that support the automatic downloading of dependencies.

Over time, the maintenance burden of supporting the code in these components began to be a drag on JDK developers. This was especially true because Common Annotations added a dependency on a Java EE container in Java EE 6, which required the JDK developers to include only a subset of its functionality. Furthermore, since the release cycles of these systems were different from that of Java SE, they often became out of date, so developers would include third-party distributions of these Java EE technologies. Problems have occurred when developers wanted to use a newer version of these third-party distributions. An "endorsed standards" override allows these implementations to be replaced; however, often people simply prepended the library to the boot classpath, which resulted in compatibility issues.

The JDK included several other technologies that previously were considered useful but are now a maintenance burden that doesn't offer Java programmers much value. CORBA, for example, used to be a very popular approach for communicating data between servers, but it was replaced by more-modern approaches for communication such as HTTP or message transport systems. Java SE also ships a subset of the Java Transaction API (JTA) and an XA transactions API.

To clear out the deadwood when Java 9 modularized the JDK, the following nine modules were marked as deprecated for removal:

- `java.xml.ws` (JAX-WS, plus the related technologies SAAJ and Web Services Metadata)
- `java.xml.bind` (JAXB)
- `java.activation` (JAF)
- `java.xml.ws.annotation` (Common Annotations)
- `java.corba` (CORBA)
- `java.transaction` (JTA)

- `java.se.ee` (aggregator module for the six modules above)
- `jdk.xml.ws` (tools for JAX-WS)
- `jdk.xml.bind` (tools for JAXB)

These modules have been removed from the JDK, but they continue to be developed by the Java EE/Jakarta EE community, and newer versions can be downloaded from Maven Central. More information can be found in JEP 320.

Conclusion

Since the move to a six-month release cycle, improvements in Java SE releases have been evolutionary rather than revolutionary. Java 11 continues that trend. All the features, however, are useful and solid improvements to the platform that will serve developers in the years to come and help continue to enhance Java's position as the most popular developer ecosystem on the planet. `</article>`

Raoul-Gabriel Urma (@raoulUK) is the CEO and cofounder of Cambridge Spark, a leading learning community for data scientists and developers in the UK. He is also chairman and cofounder of Cambridge Coding Academy, a community of young coders and students. Urma is coauthor of the best-selling programming book *Java 8 in Action* (Manning Publications, 2015). He holds a PhD in computer science from the University of Cambridge.

Richard Warburton (@richardwarburto) is a software engineer, teacher, author, and Java Champion. He is the cofounder of Opsian.com and has a long-standing passion for improving Java performance. He's worked as a developer in HFT, static analysis, compilers, and network protocols. Warburton also is the author of the best-selling *Java 8 Lambdas* (O'Reilly Media, 2014) and helps developers learn via Iteratr Learning and at Pluralsight. He is a regular speaker on the conference circuit and holds a PhD in computer science from the University of Warwick.

Join the World's Largest Developer Community

Oracle Groundbreakers

Download the latest software, tools, and developer templates

Grow your network with the Oracle Ambassador and Oracle ACE Programs

Get exclusive access to hands-on trainings and workshops

Publish your technical articles—and get paid to share your expertise

ORACLE GROUNDBREAKERS developer.oracle.com
Membership Is Free | Follow Us on Social:

@groundbreakers **facebook.com/OracleDevs**

ORACLE®

DONALD SMITH

Understanding Oracle JDK Releases in Transition

Oracle JDK, Oracle OpenJDK, and the end of public updates for Java 8

Starting with Java 11, Oracle will provide JDK releases under the open source GNU General Public License v2—with the Classpath Exception (GPLv2+CPE)—and under a commercial license for those using the Oracle JDK as part of an Oracle product or service or for those who do not wish to use open source software. This combination of using an open source license and a commercial license replaces the historical Binary Code License (BCL), which had a combination of free and paid commercial terms. Different builds will be provided for each license, but these builds are functionally identical aside from some cosmetic and packaging differences.

From the BCL to the GPL

The BCL for Oracle Java SE technologies has been the primary license for Java SE technologies for more than a decade. The BCL permits use without license fees under certain conditions. To address modern application development needs, Oracle started providing open source licensed OpenJDK builds as of Java 9, using the same license model as the Linux platform. If you are accustomed to getting Java SE binaries for free, you can simply continue doing so with the OpenJDK builds from Oracle available at jdk.java.net. If you are accustomed to getting Java SE binaries as part of a commercial product or service from Oracle, then you can continue to get Oracle JDK releases through My Oracle Support and other locations.

Functionally Identical and Interchangeable

Oracle's BCL-licensed JDK historically contained commercial features that were not available in OpenJDK builds. As promised, however, during the past year Oracle has contributed these

features to the OpenJDK community, including Java Flight Recorder, Java Mission Control, application class-data sharing, and the Z garbage collector (ZGC).

From Java 11 forward, therefore, Oracle JDK builds and OpenJDK builds will be essentially identical. There do remain a small number of differences, some intentional and cosmetic, and some simply because more time for discussion with OpenJDK contributors is warranted.

Differences Between Oracle JDK and OpenJDK

Oracle JDK 11 emits a warning when using the `-XX:+UnlockCommercialFeatures` option, whereas in OpenJDK builds this option results in an error. The `-XX:+UnlockCommercialFeatures` option was never part of OpenJDK, and it would not make sense to add it now because there are no commercial features in OpenJDK. This difference exists to make it easier for users of Oracle JDK 10 and earlier releases to migrate to Oracle JDK 11 and later.

Oracle JDK 11 can be configured to provide usage log data to the Advanced Management Console tool, which is a separate commercial Oracle product. Oracle will work with other OpenJDK contributors to discuss how such usage data might be useful in future releases of OpenJDK, if at all. This difference remains primarily to provide a consistent experience to Oracle customers until such decisions are made.

The javac `--release` command behaves differently for the Java 9 and Java 10 targets, because in those releases the Oracle JDK contained some additional modules that were not part of corresponding OpenJDK releases. The additional modules are

- `javafx.base`
- `javafx.controls`
- `javafx.fxml`
- `javafx.graphics`
- `javafx.media`
- `javafx.web`
- `java.jnlp`
- `jdk.jfr`
- `jdk.management.cmm`

- `jdk.management.jfr`
- `jdk.management.resource`
- `jdk.packager.services`
- `jdk.snmp`

This difference remains in place to provide a consistent experience for specific kinds of legacy use. These modules now are either available separately as part of OpenJFX, available in both OpenJDK and the Oracle JDK because they were commercial features that Oracle contributed to OpenJDK (such as Flight Recorder), or removed from Oracle JDK 11 (for example, JNLP).

The output of the `java --version` and `java -fullversion` commands will distinguish Oracle JDK builds from OpenJDK builds, so that support teams can diagnose any issues that may exist. Specifically, running `java --version` with an Oracle JDK 11 build results in

```
java 11 2018-09-25
Java(TM) SE Runtime Environment 18.9 (build 11+28)
Java HotSpot(TM) 64-Bit Server VM 18.9 (build 11+28, mixed mode)
```

And for an OpenJDK 11 build, it results in

```
openjdk version "11" 2018-09-25
OpenJDK Runtime Environment 18.9 (build 11+28)
OpenJDK 64-Bit Server VM 18.9 (build 11+28, mixed mode)
```

The Oracle JDK has always required third-party cryptographic providers to be signed by a known certificate. The cryptography framework in OpenJDK has an open cryptographic interface, meaning it does not restrict which providers can be used. Oracle JDK 11 will continue to require a valid signature, and Oracle's OpenJDK builds will continue to allow the use of either a valid signature or an unsigned third-party crypto provider.

Oracle JDK 11 will continue to include installers, branding, and JRE packaging to provide an experience consistent with legacy desktop uses. Oracle's OpenJDK builds are currently available as zip and tar.gz files; alternative distribution formats are being considered.

What Should They Be Called?

Ideally, you could simply refer to all Oracle JDK builds as the "Oracle JDK," under either the GPL or the commercial license depending on your situation. However, for historical reasons while the small remaining differences exist, Oracle will refer to them separately as Oracle's OpenJDK builds and the Oracle JDK.

Java 8 and the End of Public Updates

Whatever you call it, migrating to Java 11 should be under active consideration, because very shortly (January 2019), Java SE 8 will reach the end of public updates for commercial users. By that time, Oracle will have provided almost five years of continuous, free public updates. For commercial users who need to stay on Java SE 8 for a longer period of time, Oracle offers the Oracle Java SE Subscription so they can continue to benefit from support and regular updates to Java SE 8, including stability, performance, and security patches. Users wishing to continue with free updates and patches should move on to later releases, the most current of which is Java 11, as mentioned above.

However, Oracle will continue to provide free public updates of Oracle Java SE 8 for *personal* desktop use until at least December 2020. During that time, personal users should contact their application providers and encourage them to migrate their applications to the most recent version of Java, or else switch to alternative applications.

Going forward, Oracle intends to provide free public updates to the Oracle JDK and Oracle OpenJDK builds for at least the six-month period until the next release. Commercial users can then get a support subscription for subsequent updates on long-term releases of the JDK or switch to an OpenJDK release.

All these changes—to licensing, to multiple releases, and to support options—are intended to give developers and companies the choice to use Java at no cost or with paid support. `</article>`

Donald Smith (@DonaldOJDK) is the senior director of product management for Java at Oracle.

BEN EVANS

Value Types Are Coming to the JVM

How Project Valhalla will simplify access to many Java objects

In this article, I'll introduce *value types*, a potential future feature for the Java language and the JVM. This change is so deep and far-reaching that it affects every aspect of the platform—from performance to generics, even down to the fundamental way that Java represents data.

To understand what value types are, and how they'll change the way you program, you need to understand a bit about how Java represents data right now—as of Java 11. This will allow you to put value types in context by seeing what this major change involves and the consequences for how you might use Java in the future.

Current Types of Values

Since version 1.0 of the Java platform, the virtual machine has supported only two kinds of values: primitive types and object references. Primitive types (such as `int` and `double`) encode the value that they represent directly as a bit pattern with no additional metadata. Object references, on the other hand, are pointers that represent an address in the Java heap, which is an area of memory that is solely managed by the virtual machine through garbage collection.

In other words, the Java environment deliberately does not provide full low-level control over memory layout. In particular, this means that currently Java has no equivalent to C's structs; any composite data type can be defined and accessed only by the use of a reference.

Note that different Java implementations may have slightly different mechanisms and representations of objects and their references. To keep things clear, I'll talk only about the representation used in the Java HotSpot VM implementation (that is, the virtual machine used by Oracle's JDK and OpenJDK).

PHOTOGRAPH BY JOHN BLYTHE

For more than 20 years, the current memory layout pattern...has had the advantage of simplicity, but there is a performance trade-off.

Java HotSpot VM represents Java objects at runtime as *oops*, which is short for *ordinary object pointers*. Usually the oops that represent instances of a Java class are referred to as *instanceOops*. Java references are genuine pointers in the C sense and can be placed into local variables of reference type. They point from the stack frame of the Java method into the memory area that makes up the Java heap.

Specifically, all Java references point to the start of the *object header* of the object that they refer to. All Java objects are handled via an object reference, so all objects must have an object header. Unlike in C++, it is not possible to have a raw pointer to a type. There is simply no mechanism to handle objects without having the object header.

In Java HotSpot VM, the object header consists of two machine words, and the memory layout of every Java object starts with these two words. The *mark word* is the first of these, and it contains metadata that is specific to this precise instance. Following this is the *klass word*, which is a pointer to metadata that is class-specific. That means an object's *klass metadata* is shared with all other instances of the same class.

Both words of metadata are crucial to understanding how the Java runtime implements certain language features. For example, the klass word is used to look up the bytecode for Java methods.

However, for this discussion of value types, the mark word is especially important, because it is where Java objects store their identity. The mark word also stores the object's monitor (which is what allows threads to lock the object during synchronization).

To see the effect of Java's current simple model for values, let's take a closer look at the memory layout of arrays (which are, of course, objects in Java). In **Figure 1**, you can see an array

Figure 1. Array of ints

of primitive `int`s. Because these values are primitive types and not objects, they are laid out at adjacent memory locations.

To see the difference with object arrays, contrast this with the boxed integer case. An array of `Integer` objects will be an array of references, as shown in **Figure 2**, with each `Integer` having to pay the "header tax" that comes with being a Java object.

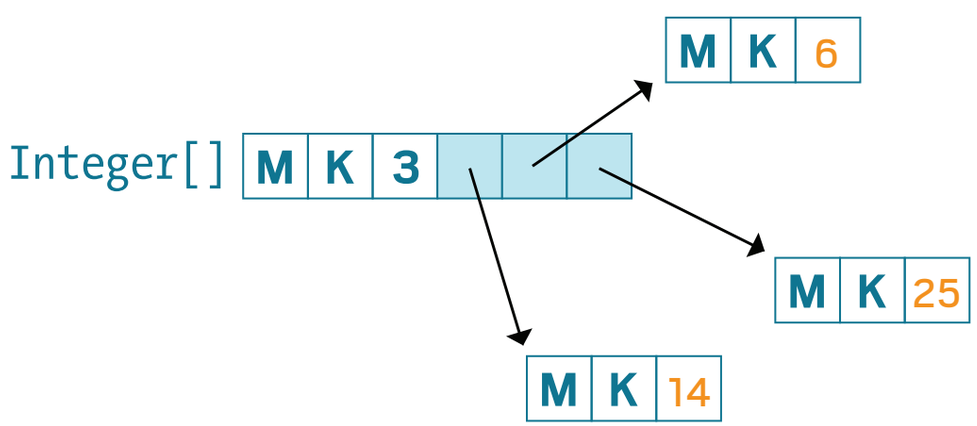


Figure 2. Array of Integer objects

For more than 20 years, the current memory layout has been the way the Java platform has worked. It is relatively simple, but there is a performance trade-off: Working with arrays of objects involves unavoidable pointer indirections and the cost of cache misses.

As an example, consider a class that represents a point in three-dimensional space, a `Point3D` type. It really comprises only three primitive doubles (for the three spatial coordinates) and, as of Java 11, it is represented as a simple object type that has three fields:

```
public final class Point3D {
    private final double x;
    private final double y;
    private final double z;

    public Point3D(double a, double b, double c) {
        x = a;
```

```
        y = b;
        c = z;
    }

    // Additional methods, for example, getters, toString(), and so on
}
```

In the Java Hotspot VM, an array of these point objects is laid out in memory as shown in Figure 3.

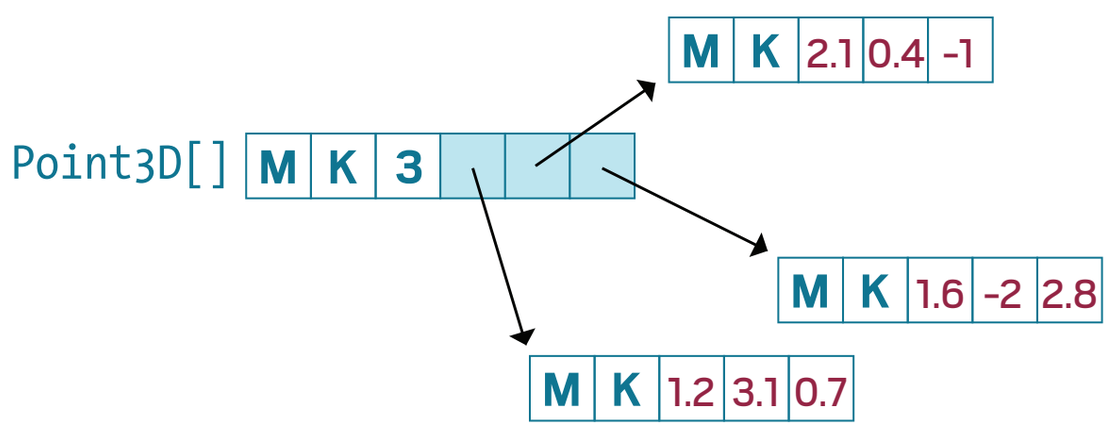


Figure 3. Array of point objects

When processing this array, each element is accessed via an additional pointer indirection to get the object that holds the coordinates of each point. This could cause a cache miss for each point in the array, which in turn might cause a performance issue.

For programmers who care a lot about performance, the ability to define types that can be laid out in memory more effectively would be very useful.

In this particular example, object identity is not likely to be important to the programmer, because the code uses neither mutability nor polymorphism, and clients are unlikely to make use of identity-sensitive features such as locking.

Project Valhalla

Within OpenJDK, a long-term research project known as Project Valhalla has been investigating the possibility of extending Java and the JVM in new directions. Its mission statement is to be "a venue to explore and incubate advanced Java VM and language feature candidates." One of the primary goals of the project is "to align JVM memory layout behavior with the cost model of modern hardware."

From a developer's perspective, one of the main outcomes of Valhalla is intended to be the arrival of a new class of values in the Java ecosystem, referred to as *value types*. These new value types are expected to be small, immutable, identityless types. Brian Goetz, the Java language architect, has used the phrase "codes like a class, works like an int" to describe how a typical user will interact with the finished version of the value types feature.

Example use cases for value types include numeric types, tuples, and more-advanced possibilities (such as algebraic data types). There is also the possibility that some existing types could be retrofitted and evolve to become represented as value types. For example, `Optional`, `LocalDate`, and lambda expressions are all candidates that could become value types in a future release if that proves to be feasible.

Removing the commitment to maintaining object identity enables optimizations such as the following:

- Removing the memory overhead of object headers
- Flattening of value types stored in fields (or arrays) rather than storing by indirection
- Reducing the memory footprint and load on garbage collectors

If this new form of value can be implemented on the JVM for types such as the spatial points discussed previously, a memory layout such as that shown in **Figure 4** would be far more efficient, because it would be *flattened* and many cache misses could be avoided. This arrangement

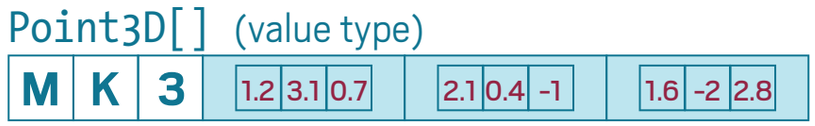


Figure 4. Possible future memory layout

would effectively represent the points as an array of `struct` without risking the full dangers of low-level memory control.

However, while the initial aims of value types seem clear, the feature has some far-reaching consequences. For Project Valhalla to be successful, it must consider the logical conclusions of introducing a third form of value. In particular, a need arises to revisit the subject of generic types.

If Java is to include value types, the question naturally occurs whether value types can be used in generic types—for example, as the value for a type parameter. The high-level design of value types has always assumed that they will be valid as values of type parameters in a new, enhanced form of generics.

This is a large and complex question that is not completely answered by the current stage of the project. Instead, Oracle has produced a very early preview as the current prototype, which is referred to as LWorld 1 (or just LW1). It replaces the earlier Minimal Value Types (MVT) prototype.

The current LW1 prototype is functional, but it is still at a very, very early stage. Its target audience is advanced developers, library authors, and toolmakers.

In my next article, I will explore LW1 and its successor, LW2. I will analyze the bytecode it currently generates and how it might benefit upcoming releases of Java and the JVM. `</article>`

Ben Evans (@kittylyst) is a Java Champion, a tech fellow and founder at jClarity, an organizer for the London Java Community (LJC), and a member of the Java SE/EE Executive Committee. He has written four books on programming, including the recent *Optimizing Java* (O'Reilly).

JEP 343: Packaging Tool for Self-Contained Java Apps

JEP 343, first written up in April 2018, proposes to develop a simple packaging tool for Java apps. Rather than just stopping at existing packaging norms (JAR, WAR, and EAR), this proposal seeks to define a tool that creates an executable containing the runtime environment and the app as a single file: an .msi or .exe file on Microsoft Windows, a pkg or app file on macOS, and deb or rpm on Linux.

In this sense, JEP 343 aims to duplicate the ability of existing packaging tools, such as Inno Setup and Wix.

Interest in this topic is driven by two trends: the greater use of containers, in which an entire app bundled with the necessary runtime is a desirable deployment option, and the advent of Java modules as of Java 9. With modules and the jlink tool, it's fairly straightforward to identify the modules in the JRE needed to run the app—leading to skinny deployables and executables.

The proposed solution, called jpackager, might include some discussed options such as generating executables that will meet the criteria for release through Windows and macOS app stores. Also under consideration is a similar solution that could launch multiple applications simultaneously in the same container or on the same platform.

If you've ever had to step users through the process of downloading and installing a JRE so that they could run your app, you understand how welcome a turnkey solution such as the proposed tool would be.

JOHAN VOS

The Future of JavaFX

JavaFX 11 marks a major turning point in the UI framework's development.

JavaFX is now an open project, separate from the JDK. Starting with Java 11, the JavaFX APIs are not included in the Oracle JDK builds or in the OpenJDK binaries. However, JavaFX is still being developed as a project in the OpenJDK initiative. Instead of being bundled with the binary releases of the core JDK builds, it is now available as a separate SDK, as well as via artifacts in Maven Central. The Maven Central repository enables developers to use JavaFX via build tools such Maven and Gradle in the same way they would use any other Java library.

Although it's a relatively young UI framework, JavaFX already has an interesting history. Initially, the JavaFX compile and runtime environment, which contains Java class files, platform-specific native libraries, and resources (such as CSS and property files) was contained in an SDK that had to be downloaded separately and configured to work with the JDK. Developers working with JavaFX, as well as all users of JavaFX applications, had to install this JavaFX SDK on their systems. Obviously, that requirement made adoption difficult.

Therefore, it was a huge win in 2012 when the JavaFX SDK became part of the JDK. From that moment, JavaFX developers didn't need to install a separate component to develop JavaFX applications. JavaFX users needed only a standard JRE distribution to run JavaFX applications. This approach made it much easier for developers to create JavaFX applications.

Technologies change, and development approaches change as well. For the following reasons, the model in which the JavaFX SDK was bundled with the JDK needed to change again:

- **Libraries are pulled in from build tools.** Today, developers use build tools such as Maven or Gradle to manage software dependencies. They specify which artifacts they want to use and what versions. The build tools make sure that the required artifacts and all their transitive dependencies are installed on the developers' systems. Most developers consider this a much easier approach than manually installing all the libraries and frameworks they depend on

PHOTOGRAPH BY TON HENDRIKS

and manually managing the different versions and dependencies. Starting with JavaFX 11, the JavaFX components are available in Maven Central and can be used with Maven and Gradle.

- **The JDK benefits from a smaller size and has a different release cadence.** The JDK itself is required by the typical build tools. Hence, Java developers still need to install a JDK. However, the lifecycle of the software they want to use does not necessarily match the lifecycle of the libraries they want to use. The fewer modules there are in the core JDK, the more flexibility developers have to decide which modules and components they want and what versions. It also makes the core JDK simpler and easier to maintain. Having modules that are not critical in the core JDK makes it harder to maintain the core: Modules that are in the core JDK require extreme caution to be maintained. JavaFX needs to be able to adapt to changes in the UI landscape and, therefore, benefits from its own release cycle and roadmap.

JavaFX is still being developed in the OpenJFX project, which is a subproject of the OpenJDK umbrella project.

- **End-user preference.** End users don't want to install Java runtimes on their systems anymore, nor do they want to upgrade them whenever there is a new release. They want applications that simply work. Applications should be self-contained and have all the dependencies they require—except for some libraries that are guaranteed to be available on the platforms. Requiring end users to download a JRE instance in order to run JavaFX applications would drastically lower the potential target audience. Instead, applications need to be self-contained, using tools like jlink and the Java Packager.

When these reasons were taken together, it was clear that the right path was to remove the JavaFX modules from the core JDK. Not everything changes, though. JavaFX is still being developed in the OpenJFX project, which is a subproject of the OpenJDK umbrella project. Therefore, OpenJFX follows the same procedures as the OpenJDK project, including different roles for participants—such as committers and reviewers—and it leverages the same project infrastructure. OpenJDK projects can have their own variations on the rules, so if the OpenJFX community decides to do so, it can make those procedures more specific and suitable for OpenJFX.

Because the JavaFX components are not distributed anymore via the JDK, they need to be distributed in another way. Gluon has undertaken doing this; it makes sure builds of the JavaFX modules are uploaded to Maven Central, and JavaFX SDKs are also built that contain all the modules including the native libraries (which is useful, for example, for developers who do not use Maven or Gradle). This last step is especially convenient for developers who want to use jlink to create a JDK runtime that includes the relevant modules from the JDK and the JavaFX modules that their application needs.

While the distribution model had to be changed, the OpenJFX community took the opportunity to make a few more changes at the same time. For example, because many of today's popular software components are developed on GitHub, a mirror of the OpenJFX repository has been created there. This step has brought in more new contributors who are accustomed to the GitHub development model but thought the OpenJDK procedures were too intimidating. As a result, there is now more traffic on the OpenJFX developer mailing list than before this change.

In addition, continuous testing and building is now an integral part of OpenJFX development. Whenever a developer creates a pull request on the GitHub mirror, both AppVeyor and Travis jobs are run to see that all code still builds and the tests pass.

Combined, all these changes, which ultimately led to the JavaFX 11 release, have created the foundation for future development.

Moving Forward

JavaFX follows the same release cadence as OpenJDK, and the benefits are similar: Features that are ready for a release will make it into a release. Features that require more time can be added in later releases. JavaFX 12 is planned for release in March 2019.

There are myriad companies using JavaFX in a wide variety of environments—for example, medical, automotive, scientific, military, and entertainment. Similar to how the core JDK tries to stay small enough to be maintainable, JavaFX does not include vertical market-specific extensions. There is a big ecosystem in the JavaFX world, and different companies and individuals provide (free and commercial) libraries and extensions.

A key point in future development of JavaFX is making sure that third-party frameworks and applications can focus on their specific added value. The JavaFX modules will ensure that application developers are shielded from the integration of new and upcoming low-level, platform-specific graphical interfaces.

An important criterion for JavaFX is performance. JavaFX has a rendering system that allows for multiple hardware-accelerated systems, such as DirectX and OpenGL. New systems are being integrated with operating systems, and work is in progress for evaluating those new systems and eventually supporting them in JavaFX.

Because the OpenJFX project really wants to be open, its community is largely responsible for the roadmap. In a healthy ecosystem, the features that are truly required are also the ones that are implemented. This happens because different users want such features; they discuss them (on the mailing lists, for example); and they spend resources on writing code, tests, and maintenance procedures.

Conclusion

How can you help? If you're interested in a specific UI control, you can join one of the third-party projects (such as ControlsFX). If you are a JavaFX user and you encounter a defect, you can file a bug report. If you want to go one step further, you can discuss the cause of the issue on the mailing list or in the bug report itself. If you want to go even further, you can suggest a fix and create a pull request. The more steps you take, the larger your influence in the OpenJFX project. That is how open source works. `</article>`

Johan Vos is a Java Champion and the cofounder and CTO of Gluon. He started working with Java in 1995 and was a core developer on the team responsible for porting Java to Linux. He is the founder and driving force behind JavaFXPorts, which forms the bottom layer of Gluon Mobile, where he coordinates the ports of JavaFX to mobile platforms.

Java™

developer.oracle.com/java

DEVELOP WITH THE
GLOBAL STANDARD

#1 Developer Choice for the Cloud

12 Million Developers Run Java
21 Billion Cloud-Connected Java Virtual Machines
38 Billion Java Virtual Machines are in the Cloud

ORACLE®

IAN DARWIN

The Decorator Pattern in Depth

Add functionality to a class without modifying it.

Interior decorators are people who come to your house and tell you how to modify the look and feel of your rooms—for example, what furniture and wall or floor covering you should add. Decorators in software are pieces of software that add functionality to code by "wrapping" a target class without modifying it. They exemplify the open/closed principle, which states that classes should be open for extension but closed to modification.

Consider the set of classes needed to manage orders for a small art photography business. The business sells prints, which can be made in various sizes, on matte or glossy paper, with or without a frame, without a mat or with one or more mats that come in many colors. It also sells prints digitally through a variety of stock photo agencies.

Your first thought might be to use inheritance to implement this data model. However, trying to make a different class for every combination of paper finish, paper size, with and without a mat, and with and without a frame would result in a true explosion of classes. And it would all fall apart as soon as market conditions changed and the business tried to add another variable. The opposite approach—trying to make one class to handle all the combinations—would result in a tangled mess of `if` statements that is fragile and difficult to understand and maintain.

One of the best solutions to this problem is the *Decorator* pattern, which allows you to add functionality to an existing component *from outside* the class. To do this, you create a class called `Decorator` (typically abstract), which needs to implement the same interface as the original `Component` (the class to which we're adding functionality); that is, the `Decorator` needs to be able to substitute for the original, using its interface or class definition as appropriate. The `Decorator` will hold an instance of the `Component` class, which it is also extending. The `Decorator` will, thus, be in both a *has-a* and an *is-a* relationship with the `Component`. So you'll often see code like this in the `Decorator`:

```
public abstract class Decorator extends Component {
    protected Component target;
            ...
    }
```

As a result of this inheritance, the `Decorator` or its subclasses can be used where the original was used, and it will directly delegate business methods to the original. You can then make subclasses from this `Decorator`. The subclasses will usually add "before" and/or "after" functionality to some or all of the methods that they delegate to the real target; this is, in fact, their raison d'être.

In the photo business example, the original `Component` is `PhotoImage`. Its constructor takes the artistic name of the image and the name of the file in which the master version of the sellable image is stored. The code that follows can be found online.

The `Decorator` class both extends `PhotoImage` and has a photo. You can then have `Print` and `DigitalImage` decorators for the two main ways of selling. The `Print` constructor represents a physical print, so it has a paper size (width and height, in inches). For `Print`, you can have `Frame`, `Mat`, and other `Decorator`s, with parameters as needed (for example, mats have color).

First, here are a few examples of using this set of classes, from `ImageSales.java`:

```
// Create an undecorated image
final PhotoImage image = new PhotoImage(
    "Sunset at Tres Ríos", "2020/ifd12345.jpg");

// Make a print of that, on US letter-size paper
Print im1 = new Print(11, 8.5, image);
addToPrintOrder(im1);

// Make a 19x11 print of a second image, matted in green, framed
Print im2 =
    new Print(19, 11,
```

```
            new Frame(
                new Mat("Lime Green",
                    new PhotoImage("Goodbye at the Station",
                        "1968/ifd.00042.jpg"))));
        addToPrintOrder(im2);

        // Make a digital print sale
        PhotoImage dig = new DigitalImage(image, StockAgency.Getty, 135.00);
        System.out.println(dig);
```

The `addToPrintOrder()` method takes a `Print` argument, providing a degree of type safety. In this simple demo, this method just prints the argument by calling `toString()`, to show the effect of the delegation methods.

Here is the code for `PhotoImage` (the `Component`):

```
/** A PhotoImage is a picture that I took at some point.
 */
public class PhotoImage {
    /** The human-readable title */
    String title;

    /** Where the actual pixels are */
    String fileName;

    /** How many pixels are there in the image file? */
    int pixWidth, pixHeight;

    public PhotoImage() {
        // Empty; used in Decorators
    }
```

```
public PhotoImage(String title, String fileName) {
    super();
    this.title = title;
    this.fileName = fileName;
}

/** Get a printable description; may be more detailed than toString()
 * but in any case, it's the example delegation method
 */
public String getDescription() {
    return getTitle();
}

/** Default toString() just uses getDescription */
@Override
    public String toString() {
        return getDescription();
    }

    // setters and getters...
}
```

Here is part of the code for `ImageDecorator`, which is the `Decorator` class:

```
public abstract class ImageDecorator extends PhotoImage {
    protected PhotoImage target;

    public ImageDecorator(PhotoImage target) {
        this.target = target;
    }
```

```
    @Override
    public String getDescription() {
        return target.getDescription();
    }

    @Override
    public String toString() {
        return getDescription();
    }
}
```

Here is part of one of the decorators, the `Print` decorator:

```
/** A Print represents a PhotoImage with the physical size at which to print it.
 */
public class Print extends ImageDecorator {
    /** PrintWidth, PrintHeight are in inches for the US audience */
    private double printWidth, printHeight;

    public Print(double printWidth, double printHeight, PhotoImage target) {
        super(target);
        this.printWidth = printWidth;
        this.printHeight = printHeight;
    }

    @Override
    public String getDescription() {
        return target.getDescription() + " " +
            String.format("(%4.1f x %4.1f in)",
                          getPrintWidth(), getPrintHeight());
    }
```

```
        // setters and getters...
}
```

One of the key parts of this code is how the `Print` class's `getDescription()` calls the target's method of the same name and also adds its own functionality to it. This is the "delegate but also add functionality" part of the pattern. And it's why, when you run the main program, you get this output for the `Print` object:

```
Goodbye at the Station, Matted(Lime Green), Framed (19.0 x 11.0 in)
```

It might seem unusual to put the secondary characteristics (paper size, mat color) as the first arguments to the constructors. If you're as compulsive as I am about such things, you'd normally put the principal element—the `Component`—as the first argument. However, in practice, putting the secondary items first facilitates the coding style with which multiple decorators are typically combined. Take a look back at the main program and imagine lining up the brackets and closing arguments when you have that many nested constructor calls. But if you don't like it, that's fine; it's just a style issue. Picking one way and being consistent in your hierarchy will make your life easier.

If the Decorator pattern looks familiar to you, it should. This is the same way that the common java.io Streams and Readers and Writers have worked since the beginning of Java.

The generic view of the class hierarchy is shown in **Figure 1**, where `Component` is the `PhotoImage`, `Print` and `DigitalImage` subclass `Decorator`, and `Mat` and `Frame` are other decorators. The only operation illustrated is `getDescription()`; others would exist in the production code.

There is no special reason that `Print` and `DigitalImage` must be decorators rather than the `ConcreteComponent` shown in **Figure 1**—that is, subclassing `PhotoImage` directly. The reason I made them decorators is so that I can, as in the code sample, create one `PhotoImage` and use it to create both a `Print` and a `DigitalImage`. You might not need to do the equivalent operation in

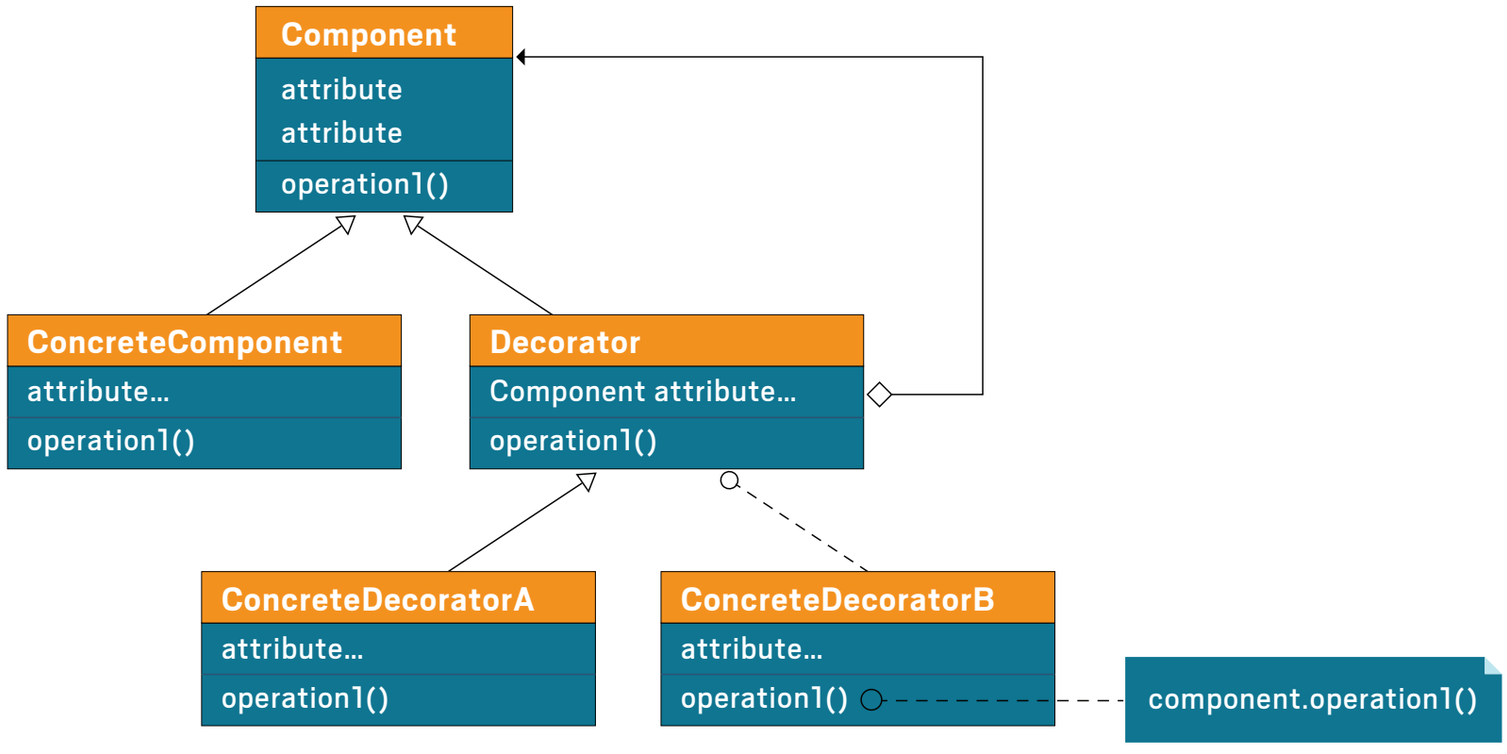


Figure 1. Decorator hierarchy

your class hierarchy; it depends on your use cases as to whether you make classes like that be `ConcreteComponent`s or decorators. Note also that for very simple cases, such as where you will only ever have one decorator class, you could skip the definition of the `Decorator` as a separate abstract class and do the delegation directly in your actual decorator. The separate class is just a convenient place to build a hierarchy when you expect you will have multiple decorators.

All this code is on GitHub, in the subdirectory `src/main/java/structure/decorator`.

Using Frameworks

Various widely used frameworks provide mechanisms for intercepting calls on managed beans. For example, Java EE's interceptors and Spring's aspect-oriented programming (AOP) offer both predefined interceptors and the ability to define your own. These are decorators. For example, both frameworks provide annotation-based transaction management, in which you can say something like this:

```
public class WidgetService {

    @Transactional
    public void saveWidget(Widget w) {
        entityManager.persist(w);
    }
}
```

Behind the scenes, both frameworks provide an interceptor whose pseudocode is probably akin to the following. The real thing will be more complex because of different transaction types (read-only, read-write) and different needs (transaction required, new transaction required, etcetera), different handling of checked versus unchecked exceptions, and so on.

```
public class TransactionInterceptor {

UserTransaction transaction;    // Injected by the container

// PSEUDOCODE

public void interceptTransaction(
    Method method, Object target, Object[] args) throws Exception {

    if (transaction.getStatus() != Status.NoTransaction) {
        transaction.begin();
    }
    try {
        method.invoke(target, args);
        transaction.commit();
    } catch (Exception ex) {
        transaction.rollback();
        throw ex;
```

```
            }
        }
    }
```

You can write your own interceptors using the EJB3 interceptor API (see `@Interceptor`) or Spring's AOP (see `@Around`).

Border Decorator

Historically, many of the earliest uses of the Decorator pattern were graphical add-ons, for example, to add a nice border to a text area or dialog box as part of a layout. Decorators were used this way in many early and influential window systems, including Smalltalk and InterViews. You'd use these as something like the following pseudocode (the `TextField` constructor takes row and column arguments):

```
Component niceTextBox = new BorderDecorator(new TextField(1, 20));
Component niceTextArea =
    new BorderDecorator(new ScrollingDecorator(new TextField(5, 20)));
```

Depending on your application, you might or might not need to forward all delegated methods. An issue with `Decorator` is that if the component being decorated has a large number of methods (it shouldn't, if you remembered to keep things simple), and you need to forward all the methods, the number of delegation methods required can make this pattern cumbersome. Yet if you don't forward a particular inherited method, because of how inheritance works that method will (if called) execute in the context of the `Decorator` alone, not the target, and this can lead to bad results. You can use the IDE to generate all the delegates (for example, in Eclipse, using the `Source → Generate Delegate Methods` option), but that does not lead to maintainable code. The Strategy pattern or the Proxy pattern might be a good alternative way of adding functionality.

The people who wrote Java's Swing UI package were aware of this issue. They wanted to decorate `JComponent` (a subclass of the Abstract Window Toolkit's `Component`), but that dear thing

has around 120 public methods. To allow decoration in the original sense, without making you subclass it just for this purpose, they took a variant approach and provided the Swing `Border` object. This `Border` object isn't used like a traditional decorator but as what you might roughly call a "plugin decorator," using these methods defined in `JComponent`:

```
public void setBorder(Border);
public Border getBorder();
```

This `Border` class is in the `javax.swing.border` package. You get instances of `Border` from `javax.swing.BorderFactory` by calling methods such as `createEtchedBorder()` and `createTitledBorder()`.

I/O Streams

If the Decorator pattern looks familiar to you, it should. This is the same way that the common `java.io` `Streams` and `Readers` and `Writers` have worked since the beginning of Java. You've probably seen code like this a million times:

```
// From IOStreamsDemo.java
BufferedReader is =
    new BufferedReader(new FileReader("some filename here"));
PrintWriter pout =
    new PrintWriter(new FileWriter("output filename here"));
LineNumberReader lrdr =
    new LineNumberReader(new FileReader(foo.getFile()));
```

Here, there is no separate `Decorator` class; the classes involved all just subclass `Reader` or `Writer` (or `InputStream` and `OutputStream` for binary files) directly, and all the classes have constructors that accept an instance of the top-level class (or any subclass, of course) as an argument. But this usage fits in with the basic description of the Decorator pattern: one class adds functionality to another by wrapping it.

Also see the previous articles in this series:

Command Pattern

State Pattern

Visitor Pattern

Decorator Versus Proxy

Proxy is another pattern in which classes expand upon other classes, often using the same interface as the object being proxied. As a result, people sometimes confuse the Proxy pattern with the Decorator pattern. However, Decorator is primarily about adding *functionality* to the target. The Gang of Four definition of Proxy is that it's about *controlling access* to the target. This control could be to provide lazy creation for expensive objects, to enforce permissions or a security-checking point of view (a security proxy), or to hide the target's location (such as a remote access proxy as used in RPC-based networking APIs such as RMI, remote EJB invocation, the JAX-RS client, or the JAX-WS client).

In the lazy-creation situation, a lightweight proxy is created with the same interface as the expensive (heavyweight) object, but none or only a few of the fields are filled in (ID and title, perhaps). This proxy handles creation of the heavyweight object only when and if a method that depends on the full object is called.

In the networking situation, the client code appears to be calling a local object, but it is in fact calling a proxy object that looks after the networking and the translation of objects to and from a transmissible format, all more or less transparently. With the Decorator pattern, the client is usually responsible for creating the decorator, often at the same time that it creates the object being decorated.

Conclusion

Decorators are a convenient way of adding functionality to a target class without having to modify it. Decorators usually provide the same API as the target, and they do additional work before or after forwarding the arguments to the target object. Try using the Decorator pattern the next time you need to add functionality to a small set of classes. `</article>`

Ian Darwin (@Ian_Darwin) has done all kinds of development, from mainframe applications and desktop publishing applications for UNIX and Windows, to a desktop database application in Java, to healthcare apps in Java for Android. He's the author of *Java Cookbook* and *Android Cookbook* (both from O'Reilly). He has also written a few courses and taught many at Learning Tree International.

SIMON ROBERTS

MIKALAI ZAIKIN

Quiz Yourself

More intermediate and advanced test questions

If you're a regular reader of this quiz, you know these questions simulate the level of difficulty of two different certification tests. Those marked "intermediate" correspond to questions from the Oracle Certified Associate exam, which contains questions for a preliminary level of certification. Questions marked "advanced" come from the 1Z0-809 Programmer II exam, which is the certification test for developers who have been certified at a basic level of Java 8 programming knowledge and now are looking to demonstrate more-advanced expertise.

Answer 1 page 80

Question 1 (intermediate). What is true about the enhanced `for` statement? Choose one.

A. The loop cannot iterate directly over `java.util.HashSet`.
B. The loop can iterate directly over `java.util.Map`.
C. The loop does not provide access to the index of the current element.
D. The loop supports deletion of the current element during execution of the loop.
E. Early termination of the loop via a `break` statement is prohibited.

Answer 2 page 83

Question 2 (advanced). Given the following code and assuming the numbers at the left are line numbers, not part of the source file:

```
11: public class Ex2<T extends Runnable, String> {
12:     String s = "Hello";
13:     public void test(T t) {
14:         t.run();
15:     }
16: }
```

Which one of the following is true?

A. Line 11 fails to compile.
B. Line 12 fails to compile.
C. Line 13 fails to compile.
D. Line 14 fails to compile.
E. Compilation succeeds.

Answer 3
page 86

Question 3 (advanced). Given the following:

```
List<String> wordList =
    Arrays.asList("how", "this", "may", "be", "correct");
Set<String> words1 = new HashSet<>(wordList);
Set<String> words2 = new TreeSet<>(wordList);

// CODE OMITTED HERE

System.out.print(words1.stream().findFirst().get());
System.out.println(words2.stream().findFirst().get());
```

Assume that the behavior of the code not shown at the `// CODE OMITTED HERE` line is to add more elements to both sets and then delete all the newly added elements. As a result, the number and values of the data items in the sets after this comment are identical to the contents before this line.

Which best describes the possible output?

A. bebe
B. be<any value from list>
C. <any value from list>be
D. <any value from list><any value from list>

Answer 4
page 88

Question 4 (intermediate). Given the following code fragment:

```
String[] nums = {"One", "Two", "Three", "Four", "Five", "Six", "Sev"};
for(int i = 0; i < nums.length; i++) {
    if (nums[i++].length() % 3 == 0) {
         continue;
    }
    System.out.println(nums[i]);
    break;
}
```

What is the output?

A. Three

B. Four

C. Five

D. Three
Four
Five

E. No output

Question 1
page 78

Answer 1. Option C is correct. The enhanced `for` expression was introduced in Java SE 5, and it provides a higher level of abstraction iterating over the elements of an `Iterable` (for example, a Set or a List) or an array.

Prior to the enhanced `for` loop, developers would describe *how* to iterate—for example, by declaring a loop and a counter variable, incrementing the counter, and comparing it against the

maximum count to control progress through the data and to terminate the iteration at the right point.

Using the enhanced `for` loop, developers simply indicate the intention of processing each item from the data set. The details of how that extraction is performed and how the loop is controlled are handled by the enhanced `for` loop. This focus on "what should be achieved" rather than "how to achieve it" is what is meant by the description "higher level of abstraction."

While Map is part of the Collections API, it doesn't extend any parent interface. That is, it doesn't extend the Iterable interface (either directly or indirectly) and so it cannot be used as the actual parameter of an enhanced for loop.

However, with this higher level of abstraction comes a separation from specific implementations, and with it, a loss of some familiar detail. Specifically, the enhanced `for` loop is able to iterate over arrays and anything that implements the interface `Iterable`. Although it's common to iterate arrays, and perhaps Lists, using an index, an index has no real meaning with a Set or with any other `Iterable` that doesn't keep its contents in an explicitly ordered structure.

Because the enhanced `for` loop must work consistently in all valid cases, including those without an index position, the notion of an index does not form part of the enhanced `for` loop. This means that option C is correct.

Returning to the fact that the enhanced `for` loop can operate on any array or any type that implements the `java.lang.Iterable` interface: `HashSet` implements the `Set` interface, which extends the `Collection` interface, which extends the `Iterable` interface. Basically, all classes that implement `java.util.Iterable` (and, therefore, anything that implements `java.util.Collection`) can be used with the enhanced `for` loop. This means that option A is incorrect.

As a side note, notice that option A says "cannot." Such negatives are a little unfair, and the exam generally tries to avoid them. In this question, however, we allowed ourselves to use "cannot" to simplify the question by requiring only a single answer. Even though the exam tries to avoid using "cannot," it's always important to pay close attention to wording to be sure you don't miss a detail of this type.

Option B discusses iteration over the `java.util.Map`. While `Map` is part of the Collections API, it doesn't extend any parent interface. That is, it doesn't extend the `Iterable` interface (either directly or indirectly) and so it cannot be used as the actual parameter of an enhanced `for` loop. As a result, option B is incorrect.

Notice that option B uses the description "can iterate directly." The `Map` interface provides a method called `entrySet()` that returns a set of the "rows" of the table. These rows are represented in a `Set<Map.Entry<KeyType, ValueType>>` object. Because these rows are a `Set`, which does implement `Iterable`, it's possible to iterate over the contents of a `Map` if you use this method as an intermediate step. The wording's reference to "directly" is intended to clarify that such a solution isn't what's being asked about in this option.

The enhanced `for` loop is built on the behavior of the `Iterable` and `Iterator` interfaces. The latter interface declares a method `remove()`, and the documentation for that method includes the following remarks:

> "The behavior of an iterator is unspecified if the underlying collection is modified while the iteration is in progress in any way other than by calling this method, unless an overriding class has specified a concurrent modification policy."

This excerpt tells you that you cannot safely modify the structure that's being iterated except by using the `remove()` method of the iterator. But, the enhanced `for` loop does not provide access to that iterator and, hence, you cannot safely delete an element from within the enhanced `for` loop. It is possible that deleting an element might work reliably with some particular `Iterable`, but not in the general case. As a result, you can deduce that option D is incorrect.

If you need to safely delete an element while looping, you should use a simple `for` loop or a `while` loop, and manipulate the `Iterator` directly. Of course, the particular `Iterator` implementation must support the `remove()` method, which not all do.

Option E is incorrect: the `break` statement in Java works identically with the enhanced `for` statement and the simple `for` statement (as well as with the `switch` statement).

Question 2
page 78

Answer 2. Option B is correct. Java's generics mechanism provides powerful "consistency checking" during compilation. That is, it allows the programmer to declare ideas such as "I intend to use this generic object with `Fruit`s," and then the compiler verifies that all uses of that object are consistent with the `Fruit` intention. During compilation, a process called *type erasure* removes most of the type information, but compile-time checking is sufficient for a large majority of situations.

In general, a generic class can be declared, along with generic type variables, in this form:

```
public class Pair<T> {
    private T left, right;
```

Notice that after declaring `T` in the angle brackets, `T` may be used as the placeholder for a type in the declaration of the variables `left` and `right`. Multiple type variables may be declared in a comma-separated list, for example:

```
public interface Map<K, V> …
```

However, a naive approach to type erasure can be inadequate if your code needs to use some knowledge about the generic type. Consider this extension of the example:

```
interface Sized { int getSize(); }
public class Pair<T> {
    private T left, right;
    public boolean matched() {
        return left.getSize() == right.getSize(); // line n1
    }
}
```

In the presence of type erasure, the comparison of the sizes in line `n1` will not work, because `left` and `right` are treated as simple `Object` types, which don't have the necessary `getSize()`

methods. You can rectify that by using the following syntax:

```
interface Sized { int getSize(); }
public class Pair<T extends Sized> {
    private T left, right;
    public boolean matched() {
        return left.getSize() == right.getSize(); // line n1
    }
}
```

The syntax `<T extends Sized>` constrains `T` to be something that implements `Sized` and, therefore, you (and the compiler) know that you can invoke the `getSize` method on it. In fact, this syntax can be extended to place multiple constraints on the generic type:

```
interface Sized { int getSize(); }
interface Colored { Color getColor(); }
public class Pair<T extends Sized & Colored> {
    private T left, right;
    public boolean matched() {
        return left.getSize() == right.getSize()
            && left.getColor().equals(right.getColor());
    }
}
```

Notice the syntax this time is `<T extends Sized & Colored>`, which requires `T` to implement both interfaces and, therefore, allows the matched method to perform both tests successfully. Also, one class can be mentioned in this list, but if that happens, the class must be first in the list. It's perhaps obvious, but Java's single-implementation inheritance rule makes it impossible to specify more than a single class, even though multiple interfaces might be relevant.

Given the statement above noting that if a class is mentioned in the list it must come first, does this mean that line 11 fails to compile, because the line mentions `Runnable` first and `String`

second? No, it does not, because that line doesn't use this multiple-constraints syntax. Look closely: it uses a comma.

The API documentation doesn't always reliably tell you if a data structure has an encounter order.

That comma means that two generic variables are being declared, and the declaration is correct, which in turn means that line 11 compiles, and option A is incorrect.

The first of the declared generic type variables is called `T` (by the way, there's an informal convention that `T` is usually short for "type" when you are using generics), and it's constrained to be something that implements `Runnable`. It's correctly declared and constrained, which means that the use of `T` as an argument in line 13 is correct, and option C is incorrect.

Also, because the type variable `T` carries the constraint `extends Runnable`, the invocation of `t.run()` on line 14 is correct, and option D is incorrect.

However, in addition to declaring `T` with a single constraint, the comma separation in line 11 also declares another generic type variable that has a very unwise name: `String`. This is a generic type variable, and it has nothing to do with `java.lang.String`. Because it's not constrained in any way, its base representation is `Object`, but it's not possible to know what type can be assigned to it until someone makes use of the class. Imagine that the following declaration is made:

```
class MyJob implements Runnable { public void run() {} }
Ex2<MyJob, LocalDate> ex2;
```

In this context, the type variable called `String` (remember—it's *not* a `java.lang.String`) is supposed to be `LocalDate`. It should now be clear why you cannot assign the literal `java.lang.String` object `Hello` to it and, therefore, why line 12 fails to compile and option B is the correct answer.

Although the namespace of generic type variables is totally different from the namespace of "real" types (classes, interfaces), ambiguities such as this one with the meaning of `String` caused the adoption of the convention that type variables should generally be single uppercase

letters. With a little luck, the insanity this question presents should convince you of the importance of adhering to this convention.

Of course, because line 12 fails to compile, option E is also incorrect.

Question 3
page 79

Answer 3. Option C is correct. This question investigates the required behaviors of streams and Set objects and the variations that they permit. The `findFirst` method has some flexibility that might cause unpleasant surprises if you're not familiar with the rules.

The name of the `findFirst` method appears to indicate that it specifically returns the *first* element from the stream, hinting that the output perhaps ought to be `bebe`, as in option A. But in fact, there's a significant note in the documentation for `findFirst` that states the following and means that option A is incorrect:

> "If the stream has no encounter order, then any element may be returned."

So, what does *encounter order* mean, and do these collections have it? The documentation continues:

> "Streams may or may not have a defined encounter order. Whether or not a stream has an encounter order depends on the source and the intermediate operations. Certain stream sources (such as `List` or arrays) are intrinsically ordered, whereas others (such as `HashSet`) are not. Some intermediate operations, such as `sorted()`, may impose an encounter order on an otherwise unordered stream, and others may render an ordered stream unordered, such as `BaseStream.unordered()`. Further, some terminal operations may ignore encounter order, such as `forEach()`.
>
> If a stream is ordered, most operations are constrained to operate on the elements in their encounter order; if the source of a stream is a `List` containing `[1, 2, 3]`, then the result of executing `map(x -> x*2)` must be `[2, 4, 6]`. However, if the source has no defined encounter order, then any permutation of the values `[2, 4, 6]` would be a valid result.

> For sequential streams, the presence or absence of an encounter order does not affect performance, only determinism. If a stream is ordered, repeated execution of identical stream pipelines on an identical source will produce an identical result; if it is not ordered, repeated execution might produce different results."

The code in this question draws its stream from two distinct implementations of the `Set` interface. The `TreeSet` is ordered, but as noted in the documentation excerpt above, the `HashSet` is not. Because of this, the stream extracted from the `HashSet` does not have a defined encounter order and, consequently, `findFirst` might return any item from the stream. So *any* of the strings originally added to the set might be printed as the first output item.

However, the second item is printed from the `findFirst` operation executed on the stream drawn from the `TreeSet`. Because `TreeSet` is ordered, and nothing else prevents it, the stream runs with an encounter order. The "first" item in the `TreeSet` must be returned by the `findFirst` operation. Because the default ordering of a `TreeSet` is the so-called natural order of the items it contains, and the natural order of `String` objects is alphabetical, you can infer that the second output item will consistently be `be`.

There's a reason that side effects are generally discouraged in modern programming: they tend to get overlooked, and they make it harder to comprehend code.

Given these observations, options B and D are incorrect and the correct answer must be option C.

A side note on ordering is that the API documentation doesn't always reliably tell you if a data structure has an encounter order. It's usually a fairly safe guess that if ordering is an intrinsic part of the structure, as with lists in general and with `HashSet`, encounter order will be defined. But for any given set, it's hard to tell (and from the codebase, it might be impossible to tell if you know only that you have "something that implements `Set`.") One trick worth knowing is that if you have a stream in a variable called `myStr`, the following expression will be true if the stream is ordered—and false otherwise:

```
((myStr.spliterator().characteristics() & Spliterator.ORDERED) != 0)
```

Alternatively, if you simply want to test whether a particular data structure has an encounter order, you can request the spliterator directly from that structure and test it, without ever going to a stream. For a collection called `myColl`, you could use this code:

```
((myColl.spliterator().characteristics() & Spliterator.ORDERED) != 0)
```

Finally, it's worth mentioning that it's possible to encounter in a real exam question a comment that describes the behavior of a piece of code that's omitted from the body of a description (as with the `// CODE OMITTED HERE` comment in this question). It's not something that happens often, and the complexity of the description in this case is almost certainly more than a real question would allow. However, the guidance notes on the Oracle website include the following text (expand the Review Exam Topics list):

> "**Descriptive comments:** Take descriptive comments, such as 'setter and getters go here,' at face value. Assume that correct code exists, compiles, and runs successfully to create the described effect."

So, don't be afraid to take literally any such comments, and know that the "hidden" code is not an attempt to trick you but instead is simply an effort to allow the code that's presented to focus on what's important for understanding the question.

Question 4
page 80

Answer 4. Option B is correct. This question investigates the behavior of code that uses `continue` and `break` in a loop. These kinds of questions require that you mentally simulate the Java interpreter. Doing that can be a surprisingly effective way to determine if you really understand the language, but it also demands that you pay close attention to all the details in the question. As a result, such a question can be time-consuming and frustrating in an exam, because the time required to notice details can encourage you to rush, and a tiny slip can cause you to miss the

mark for the question. Of course, real-life programming is more successful if you pay close attention to details too, so such questions are not an unreasonable test of your skills.

Let's examine the behavior of this code step by step.

- A regular C-style `for` loop is defined with a variable `i` that will serve as an index into the array of strings. The index value is initialized to zero.
- The "while-like" behavior of the `for` loop tests whether the value of `i` is less than the length of the array. It is, and so execution enters the body of the loop.
- The expression that controls the behavior of the `if` statement fetches the "zero-th" element on the array (which is `One`) and checks whether the length of that string is exactly divisible by 3 (which it is). The expression in the `if` clause also increments the value of the index variable `i`, so `i` now has a value of 1. Because the `if` condition evaluated to true, the body of the `if` is entered, and execution proceeds with the `continue` statement.
- The `continue` statement causes execution to jump immediately to the third part of the `for` loop, which is the expression `i++`. After this, the index variable `i` has a value of 2.
- Next, the `for` loop tests its "while condition" again. The index value (which is 2) is less than the length of the `nums` array, so the test is true, and the next loop iteration starts.
- Now the `if` test determines if `nums[2]` (which is `Three`) has a length that is exactly divisible by 3 and, as a side effect, it increments the index value of `i` again, bringing its value to 3. Meanwhile, because the length of `Three` is not exactly divisible by 3, the `continue` statement is skipped this time.
- Next, the code proceeds to the `println` that follows the closing brace of the `if` clause. That `println` prints the value of `nums[3]`, so the output is `Four`.
- After the print operation, the `break` statement causes the loop to exit and the code fragment's execution has been completed.

Given this explanation, you can see that only one line of output is presented and that line consists of the text `Four`. Consequently, option B is correct, and options A, C, D, and E are incorrect.

Notice that the use of a side effect of `i++` inside the condition expression in the `if` clause deliberately obfuscates this question. There's a reason that side effects are generally discour-

aged in modern programming: they tend to get overlooked, and they make it harder to comprehend code. Misreading this code would likely lead you to select option A, because the modified arithmetic would cause the output to be `Three`.

If you additionally mistook the meaning of the `break` keyword—thinking it performed like `continue` (perhaps that's not very likely in a question that has both `break` and `continue` to remind you that they're different)—the expected output would be `Three`, `Four`, and `Five` on consecutive lines, prompting you to select D.

Part of the point here is that distractors are, as far as possible, selected based on what would be true if a particular, and preferably fairly likely, error is made by test candidates.

Finally, you might think that it's unfair for a question to use a programming style that's known to be tricky, such as the side effect used in this situation. Of course, the harsh reality is that although you (and I) would never do anything so unprofessional, it's not safe to assume that all of your colleagues and the predecessors on your projects were always as thoughtful. Also, in this particular case, there's a persistent belief among some programmers that this kind of code will run faster than it would if the side effect were separated out. That's unlikely to be true with a modern compiler and a JVM, all of which have extensive optimizers built into them. `</article>`

Simon Roberts joined Sun Microsystems in time to teach Sun's first Java classes in the UK. He created the Sun Certified Java Programmer and Sun Certified Java Developer exams. He wrote several Java certification guides and is currently a freelance educator who publishes recorded and live video training through Pearson InformIT (available direct and through the O'Reilly Safari Books Online service). He remains involved with Oracle's Java certification projects.

Mikalai Zaikin is a lead Java developer at IBA IT Park in Minsk, Belarus. During his career, he has helped Oracle with development of Java certification exams, and he has been a technical reviewer of several Java certification books, including three editions of the famous *Sun Certified Programmer for Java* study guides by Kathy Sierra and Bert Bates.

//contact us/

Comments

We welcome your comments, corrections, opinions on topics we've covered, and any other thoughts you feel important to share with us or our readers. Unless you specifically tell us that your correspondence is private, we reserve the right to publish it in our Letters to the Editor section.

Article Proposals

We welcome article proposals on all topics regarding Java and other JVM languages, as well as the JVM itself. We also are interested in proposals for articles on Java utilities (either open source or those bundled with the JDK). Finally, algorithms, unusual but useful programming techniques, and most other topics that hard-core Java programmers would enjoy are of great interest to us, too. Please contact us with your ideas at javamag_us@oracle.com and we'll give you our thoughts on the topic and send you our nifty writer guidelines, which will give you more information on preparing an article.

Customer Service

If you're having trouble with your subscription, please contact the folks at java@omeda.com, who will do whatever they can to help.

Where?

Comments and article proposals should be sent to our editor, **Andrew Binstock**, at javamag_us@oracle.com.

While they will have no influence on our decision whether to publish your article or letter, cookies and edible treats will be gratefully accepted by our staff at *Java Magazine*, Oracle Corporation, 500 Oracle Parkway, MS OPL 3A-3133, Redwood Shores, CA 94065, USA.

- World's shortest subscription form
- Download area for code and other items
- *Java Magazine* in Japanese