THE DZONE GUIDE TO

MODERN JAVA

VOLUME II

BROUGHT TO YOU IN PARTNERSHIP WITH

APPDYNAMICS

ca technologies

IBM

JNbridge

Lightbend

Pivotal

vaadin

Dear Reader,

Why *isn't* Java dead after more than two decades? A few guesses: Java is (still) uniquely portable, readable to fresh eyes, constantly improving its automatic memory management, provides good full-stack support for high-load web services, and enjoys a diverse and enthusiastic community, mature toolchain, and vigorous dependency ecosystem.

Java is growing with us, and we're growing with Java. Java 8 just expanded our programming paradigm horizons (add Church and Curry to Kay and Gosling) and we're still learning how to mix functional and object-oriented code. Early next year Java 9 will add a wealth of bigger-picture upgrades.

But Java remains vibrant for many more reasons than the robustness of the language and the comprehensiveness of the platform. JVM languages keep multiplying (Kotlin went GA this year!), Android keeps increasing market share, and demand for Java developers (measuring by both new job posting frequency and average salary) remains high. The key to the modernization of Java is not a laundry-list of JSRs, but rather the energy of the Java developer community at large.

In fact, the enthusiasm of practicing developers has been Java's strength since the beginning. The Java platform has always been backed, but never limited, by a single entity. Sun first made Java for their own enterprise hardware and software ecosystem, but WORA made everyone else want to use it, too. Microsoft failed to fork Java onto a Windows-flavored JVM because the developer community – spearheaded by Javalobby, DZone in its earliest form – understood the vital importance of a platform-neutral, industrial-strength, object-oriented language. Google is plenty gung-ho to create its own languages (Go, Dart) but chose Java for Android – fatefully, since now Java powers far more feature *and* smart phones than any other language. A few months ago Red Hat, IBM, Tomitribe, Payara, and other groups banded together to sketch out the beginnings of a specification for microservice support in Java – just the most recent of many platform-driving endeavors of the Java community beyond the formal JCP.

One of the more convenient effects of Java's strong commitment to backwards compatibility is that we don't have to change when the language and/or platform do. But those changes aren't determined by corporate or academic experiment. A broad-based community means that the Java ecosystem evolves in response to developers' changing real-world needs. We're steering this ship; let's ride it as fast and as far as it can take us.

So here at DZone we're thrilled to publish our *2016 Guide to Modern Java*. No other language has benefited the world as much as Java, and we're proud to be a part of that ecosystem. We know you are too, and we hope you'll be able to do even more with Java after reading this Guide.

BY JOHN ESPOSITO
SENIOR RESEARCH ANALYST, DZONE
RESEARCH@DZONE.COM

TABLE OF CONTENTS

3 **EXECUTIVE SUMMARY**

4 **KEY RESEARCH FINDINGS**

10 **THE JAVA 8 API DESIGN PRINCIPLES**
BY PER MINBORG

13 **PROJECT JIGSAW IS COMING**
BY NICOLAI PARLOG

18 **REACTIVE MICROSERVICES: DRIVING APPLICATION MODERNIZATION EFFORTS**
BY MARKUS EISELE

21 **CHECKLIST: 7 HABITS OF SUPER PRODUCTIVE JAVA DEVELOPERS**

22 **THE ELEMENTS OF MODERN JAVA STYLE**
BY MICHAEL TOFINETTI

28 **12 FACTORS AND BEYOND IN JAVA**
BY PIETER HUMPHREY AND MARK HECKLER

31 **DIVING DEEPER INTO JAVA DEVELOPMENT**

34 **INFOGRAPHIC: JAVA'S IMPACT ON THE MODERN WORLD**

36 **FROM DESKTOP TO WEB JAVA: A PLANNER'S GUIDE TO MIGRATION**
BY BEN WILSON

40 **HASH TABLES, MUTABILITY, AND IDENTITY: HOW TO IMPLEMENT A BI-DIRECTIONAL HASH TABLE IN JAVA**
BY WAYNE CITRIN

42 **EXECUTIVE INSIGHTS INTO JAVA DEVELOPMENT**
BY TOM SMITH

46 **JAVA SOLUTIONS DIRECTORY**

55 **GLOSSARY**

EDITORIAL

CAITLIN CANDELMO
DIRECTOR OF CONTENT + COMMUNITY

MATT WERNER
CONTENT + COMMUNITY MANAGER

MICHAEL THARRINGTON
CONTENT + COMMUNITY MANAGER

NICOLE WOLFE
CONTENT COORDINATOR

MIKE GATES
CONTENT COORDINATOR

SARAH DAVIS
CONTENT COORDINATOR

INDUSTRY + VENDOR RELATIONS

JOHN ESPOSITO
SENIOR RESEARCH ANALYST

TOM SMITH
RESEARCH ANALYST

BUSINESS

RICK ROSS
CEO

MATT SCHMIDT
PRESIDENT & CTO

JESSE DAVIS
EVP & COO

KELLET ATKINSON
DIRECTOR OF MARKETING

MATT O'BRIAN
SALES@DZONE.COM
DIRECTOR OF BUSINESS DEVELOPMENT

ALEX CRAFTS
DIRECTOR OF MAJOR ACCOUNTS

JIM HOWARD
SR ACCOUNT EXECUTIVE

CHRIS BRUMFIELD
ACCOUNT MANAGER

PRODUCTION

CHRIS SMITH
DIRECTOR OF PRODUCTION

ANDRE POWELL
SENIOR PRODUCTION COORDINATOR

G. RYAN SPAIN
PRODUCTION PUBLICATIONS EDITOR

ART

ASHLEY SLATE
DESIGN DIRECTOR

SPECIAL THANKS to our topic experts, Zone Leaders, trusted DZone Most Valuable Bloggers, and dedicated users for all their help and feedback in making this report a great success.

WANT YOUR SOLUTION TO BE FEATURED IN COMING GUIDES?
Please contact research@dzone.com for submission information.

LIKE TO CONTRIBUTE CONTENT TO COMING GUIDES?
Please contact research@dzone.com for consideration.

INTERESTED IN BECOMING A DZONE RESEARCH PARTNER?
Please contact sales@dzone.com for information.

Executive Summary

The most successful programming language in history owes much of its continuing vitality to the most effective development community in history. More than any other widely used programming language, Java learns from and responds to millions of developers' needs; and the most important perspective on how Java is working well, and what Java needs to do better, comes from experienced software engineers spending hours every day in their Java weeds. Is Java doing its job? We surveyed 2977 developers to find out. And we went deeper, seeking to discover exactly how developers are using Java to solve today's business problems, on modern deployment topologies, using modern release practices, in shortening release cycles, with increasingly modularized application architectures, meeting increasingly stringent performance requirements. Here's what we learned.

JAVA 8 IS NOW DOMINANT

DATA 84% of developers are now using Java 8, showing steady growth from October 2014 (27%) through May 2015 (38%), August 2015 (58%), and March 2016 (64%).

IMPLICATIONS Java 8 is the new normal. You are increasingly (and now extremely) likely to encounter Java 8 code in others' applications (especially applications created in the past year), and you will be increasingly expected to be able to write code in idioms influenced by Java 8.

RECOMMENDATIONS Take advantage of features new to Java 8 and assume that you will encounter Java 8 in other developers' code. For more on modern Java API design, check out Per Minborg's article on page 10.

DEVELOPERS ARE USING NEW-TO-JAVA-8 FEATURES EXTENSIVELY, ESPECIALLY LAMBDAS AND STREAMS

DATA 73% of developers are using lambdas. 70% are using the Stream API. 51% are returning Optionals.

IMPLICATIONS Two of the three most important features new to Java 8 are now used by a large majority of developers. Both (lambdas and streams) facilitate function composition and (for certain kinds of function pipelines) more readable code. Multi-paradigm programming in Java is a reality. Separately: as Optional usage increases, NullPointerExceptions become less likely.

RECOMMENDATIONS To exercise your modern Java muscles, practice refactoring code using lambdas, streams, and Optionals. To understand functional programming from a theoretical point of view, dive deeper into the lambda calculus. Celebrate your shrinking chances of cursing the existence of Null.

MICROSERVICES ADOPTION IS GROWING STEADILY

DATA In August 2015, 10% of developers were using microservices; in January 2016, 24% were using microservices somewhere in their organization; in a report just published by Lightbend, 30% are using microservices in production; and, in our latest survey, 39% of developers are currently using microservices, and another 19% are planning to adopt microservices in the next 12 months.

IMPLICATIONS No matter how old the concept, and no matter how much you appreciate the headaches of locating complexity in the entire component graph rather than individual nodes, developers are already architecting their applications as microservices at significant rates and will doing so even more in the near future.

RECOMMENDATIONS Consider how you might refactor your "monolithic" applications into microservices, even if you decide to retain your "monolith" in production. Become more familiar with distributed computing principles. Practice domain-driven design. Read Markus Eisele's article on page 16 below.

NON-JAVA JVM LANGUAGES ARE PROLIFERATING AND ENJOYING MORE USE, ESPECIALLY SCALA

DATA 45% of developers are using Groovy, up from 39% in 2015. 41% are using Scala in development (18%), in production (10%), and/or just for fun (26%), up from 31% in 2015. Clojure use has doubled over the past year, and Kotlin is now used in significant numbers, although still mostly just for fun.

IMPLICATIONS The JVM continues to gain momentum even among non-Java developers. JVM developers are increasingly playing with programming models beyond object-orientation. For all the valuable readability introduced by Java's verbosity, the syntactic sugar offered by Kotlin and Groovy, for example, is not to be dismissed lightly.

RECOMMENDATIONS Play with other JVM languages: Groovy for high-productivity scripting, Scala for type-safe and multi-paradigm programming, Clojure for Lisp-like macros and code-as-data functional purity, Kotlin for the pleasure of rapid development and beyond-Groovy performance on Android.

THE FUTURE OF JAVA LOOKS BRIGHT

DATA 80% of developers are optimistic about the future of Java (42% "very optimistic," 40% "fairly optimistic").

IMPLICATIONS Java is neither dead nor dying. Java developers have both the highest stakes in the future of Java and also the deepest, most information-rich engagement with the Java community; so developers' aggregate opinions on the future of Java should be an excellent indicator of the health of the ecosystem.

RECOMMENDATIONS Keep writing Java. Contribute to open-source Java projects. Participate actively in the community-driven improvement of the Java platform, whether via the JCP or through any of the growing set of ancillary organizations and communities (MicroProfile.io, Java EE Guardians, local JUGs, etc.).

Key Research Findings

2977 software professionals responded to DZone's 2016 Java survey. Respondent demographics are as follows:

- 60% identify as developers or developer team leads
- 24% identify as architects (no overlap with developers and developer team leads)
- 43% have more than 15 years of experience as IT professionals
- 42% work at companies whose headquarters are located in Europe; 31% in the USA
- 20% work at companies with more than 10,000 employees; 27% with 500-9,999 employees; 20% at companies with 100-499 employees

1. JAVA 8 NOW DOMINATES NEW APPLICATIONS

It is not surprising that Java 8 adoption is increasing. But the numbers are becoming impressive: 84% of our survey respondents are now using Java 8 for old or existing applications. This number represents a steadily growing adoption curve over the past two years. In October 2014, a Typesafe survey showed a Java 8 adoption rate of 27%; in May 2015, a Baeldung survey showed 38%; in August 2015, our annual Java survey showed 58%; in March 2016, another Baeldung survey showed 64%. (Not all of these surveys included explicit demographics, so comparison of results is not rigorous; but given the sources it seems likely that all of these surveys were conducted on a roughly similar enterprise Java developer population.

Java 8 is far more likely to be adopted for new applications than to be introduced into existing applications. While 81% of respondents report using Java 8 for new applications, only 34% report using Java 8 for existing applications. (The overlap is huge, of course: only 59 respondents (3% of total) that use Java 8 in existing applications do not also use Java 8 in new applications.) But Java 8 adoption in legacy code is also up considerably from August 2015, when only 20% of respondents reported using Java 8 for existing apps.

2. HOW IS JAVA 8 CHANGING HOW DEVELOPERS WRITE CODE?

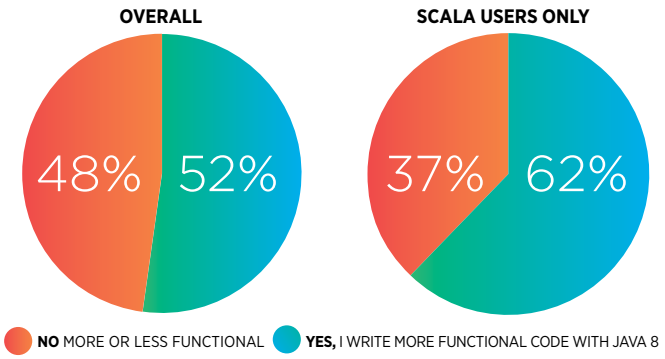
Developer usage of particular OpenJDK versions speaks to developers' attitudes toward Java 8; but usage of language features new to Java 8 says more about how actual Java code is changing over time. To understand how Java 8 has changed how Java developers write code, we asked four questions about usage and attitudes.

QUESTION 1: Overall do you write more 'functional' Java code now vs. before Java 8?

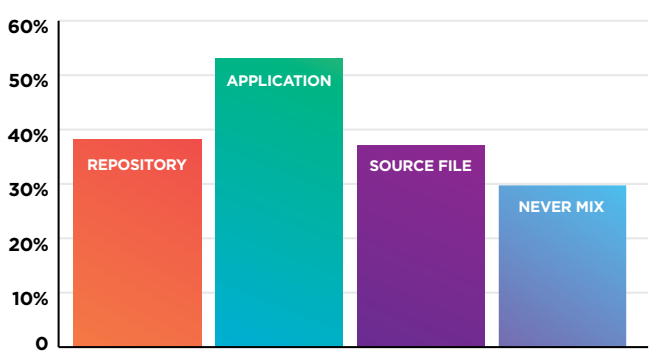
1. Of course any Turing-complete language can compute anything computable; and of course lambdas and streams simply make convenient certain computations that were possible in previous versions of Java. But as a programming model becomes more convenient, it is likely to be used more; and as developers use it more, their minds are likely to grow more accustomed to the newly habitual model. It seems surprising, therefore, that developers do **not** report writing more "functional" code because of Java 8. Only 63 responses out of 1977 separated "yes, I write more functional code because of Java 8" from "no, my code is no more or less functional now vs. before Java 8" — a negligible difference (51.6% vs. 48.4%) and well within the survey's margin of error.

2. Two interpretations seem possible. First, suppose that these respondents' reports are accurate; then Java 8 did not significantly re-habituate Java developers' minds toward a more functional programming model, in spite of newly simplified constructs. This may indicate that the language

HOW FUNCTIONAL IS YOUR CODE NOW VS. BEFORE JAVA 8?



DO YOU MIX "OLD STYLE" (PRE-JAVA 8) AND "NEW STYLE" (LAMBDAS, STREAMS, OPTIONALS)?



conveniences were well behind its users' actual coding work — that Java lambdas and streams were well overdue. But, on the other hand, "more functional" is vague enough that the half/half split seems more likely than random to be significantly affected by the wide range of possible interpretations of the question. We do have more specific data on usage of new Java 8 constructs (discussed below); but to understand developers' habits, rather than just usage at a given moment in time, future surveys providing time-series data on usage of these constructs will be essential.

3. Interestingly, respondents who also use Scala (in development, in production, or just for fun) were significantly more likely to write more functional code because of Java 8 (62%). Because use of Scala probably indicates a more multi-paradigm (including functional) mental programming model, this number may suggest that Java 8 is successfully addressing the needs/preferences of developers who were using other JVM languages for less-object-oriented code.

QUESTION 2: What "new style" Java 8 features are you using?

1. Java 8 offered many new features, but three stood out as most important because of their possible ubiquity and significant effects on program readability, efficiency, and correctness: lambdas (anonymous functions), the Stream API (function pipelines w/automatic parallelization via ForkJoinPool threads), and the Optional (return) type. Of these three, we hypothesized that lambdas were most likely to be used most, because they are less likely to require deeper rethinking of application code.

2. Our results were consistent with this hypothesis, but just barely. Usage of lambdas and streams is nearly equal both overall and in either new or refactored code (3-5% difference, within the survey's margin of error). Moreover, the numbers were high enough (67-71% in new code; 31-33% in refactored code) to cause us to re-interpret developers' responses to the general question of "functional" programming adoption via Java 8. That is, if usage of lambdas and streams (which facilitate chaining and pipelining, which allows for more declarative and functional programming) is so high, then it would seem likely that overall increase in "functional" programming would be significant as well (but responses to that explicit question indicate virtually no change). To explain this, we would have to know both (a) how *frequently* developers use lambdas and streams (as opposed to whether they use them at all) and at what stage in code creation they use them and (b) how their use of lambdas and streams changes their *mental models* of their applications.

3. We also accepted write-in Java 8 feature usage, in addition to the top features described above. The most popular write-in by far was the new `java.time` API, which appeared (under various descriptions) in 24% of write-in responses (45 out of 186). No other new Java 8 feature was written in more than 10 times.

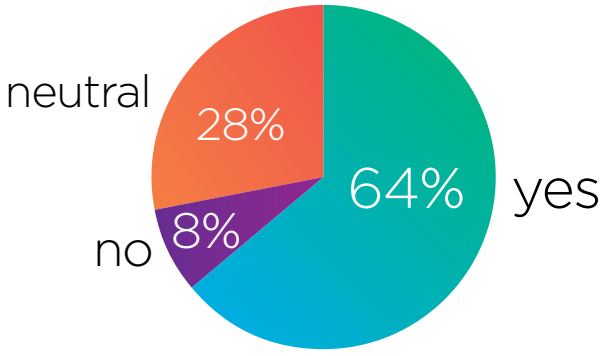
QUESTION 3: Where do you mix "old style" (pre-Java 8) and "new style" (Java 8 e.g. lambdas, streams, Optionals) code?

1. The previous questions address how Java 8 features are affecting developers' entire coding outputs. That is, if a developer is writing multiple new applications but uses Java 8 in only one of them, then that developer will be counted as using Java 8 in new applications. It also seems important to understand the effect of Java 8 orthogonally, from the point of view of the code. So we asked *where* developers were using lambdas, streams, and Optionals, in three types of grouping: in the same repository, the same application, or the same source file. The results were encouraging for the integration of functional and object-orientation: 55% of respondents mix new- and old-style Java in the same application.

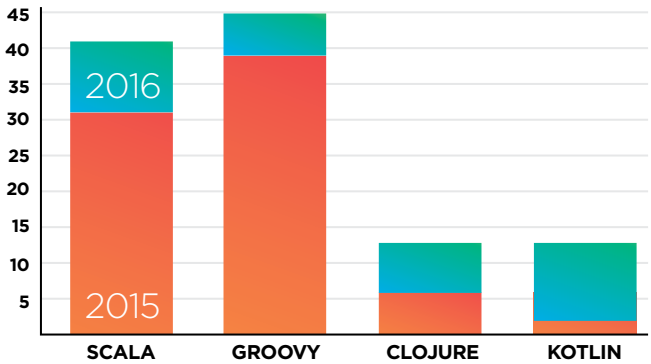
2. Two further details merit comment, however. First, only 39% report mixing "old" and "new" Java in the same repository. The difference between the 55% and 39% may indicate expansive and/or inconsistent understanding of the term "application," which apparently might or might not be split into multiple repositories; further, this seems consistent with the rise of cloud-oriented and microservices-based architectures, which blur the line between "single application" and "distributed system" such that the term "application" is increasingly likely to include code from multiple (separately managed) repositories.

3. Second, significantly fewer respondents mix "old" and "new" Java in the same source file (36%) than in the same application (55%). The division of source files is of course very variably coupled with the domain model, and is sometimes imposed by frameworks rather than chosen freely by the programmer; so it is unclear how much of the difference between "old" and "new" Java style mixing in individual application vs. individual source

HAVE THE NEW JAVA 8 FEATURES MADE IT MORE FUN TO PROGRAM WITH JAVA?



WHAT OTHER JVM LANGUAGES ARE DEVELOPERS USING?



file is a function of developers' perception of the semantics of source-file divisions, on the one hand, and factors exogenous to the developers' deliberate decisions, on the other.

QUESTION 4: Have the "new style" Java 8 features made it more fun to program in Java?

1. How developers feel influences what developers do, but many unpleasant things are worth doing for good technical reasons. Adoption of Java 8 features is therefore a function of both developer experience and also technical utility. To distinguish developers' feelings about Java 8 features from use of these features, we explicitly asked whether the "new style" Java 8 features have made it more fun to program in Java.

2. Overall they have: 64% said yes. Developers who also use Scala, however, skewed these results significantly: 73% of Scala users report that the "new style" Java 8 users make it more fun to program in Java, while only 57% of respondents who do not use Scala report the same. From this we might draw a twofold conclusion: that Java 8 had a relatively small effect on non-Scala Java developers' "fun" levels, and has particularly satisfied developers who were already exploring other programming paradigms on the JVM.

3. WHAT OTHER JVM LANGUAGES ARE DEVELOPERS USING?

As Java 8 usage increases, non-Java JVM language usage increases as well. In 2016 Groovy remains the most popular non-Java JVM language (45% of respondents use it in development, in production, or just for fun), but Scala is now very nearly as popular (41%). Year over year usage increases are significant for four JVM languages: Scala (up from 31% in 2015), Groovy (39% in 2015—note the slower growth vs. Scala), Clojure (13% in 2016 vs. 6% in 2015), and Kotlin (barely used (2%) in 2015, now 12%, since the language went GA this February). Increases in both Scala and Clojure usage presumably reflect JVM developers' generally increasing interest in functional and multi-paradigm programming, as discussed above.

The role of each of these languages varies considerably, however. Groovy's lead in production remains strong: 80% more developers use Groovy in production than use Scala in production, while just over half again as many developers use Scala "just for fun" as use Groovy "just for fun." Kotlin usage is almost entirely "just for fun" (as might be expected from a language barely half a year beyond GA). For Clojure, the difference between "in development" and "just for fun" is negligible, while it is significant (9-10%) for both Groovy and Scala—albeit in reverse order (Groovy is more likely to be used in development than just for fun, while Scala is more likely to be used just for fun than in development).

4. ENTERPRISE JAVA PLATFORMS: JAVA EE VS. SPRING (ROUND N)

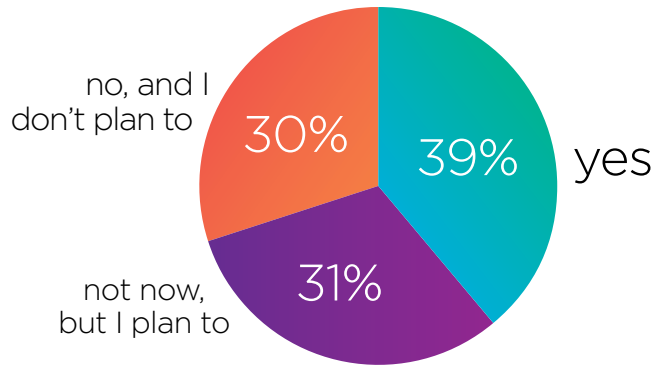
It is currently difficult to predict how Java EE will evolve in the future. This year the extremely slow progress on most Java EE JSRs, among other things, prompted a number of Java developers to found the Java EE Guardians, an independent group dedicated to advancing Java EE with as much community involvement as possible. Until two days ago (as of this writing on 9/20/2016), Oracle had not publicly discussed any substantive roadmap for Java EE 8; and in spite of an enthusiastic presentation at JavaOne 2016, which alleviated some concerns, it seems that Java EE 8 is now significantly delayed (now not scheduled to ship until late 2017). However, Java EE 9 is scheduled to release a year later—a schedule that suggests serious commitment to the platform in general, but with priorities currently unknown.

Nevertheless, usage of Java EE remains high and continues to modernize. Overall usage of Java EE remains virtually unchanged since 2015; but users have shifted from Java EE 5 to Java EE 7, which 41% of respondents are now using.

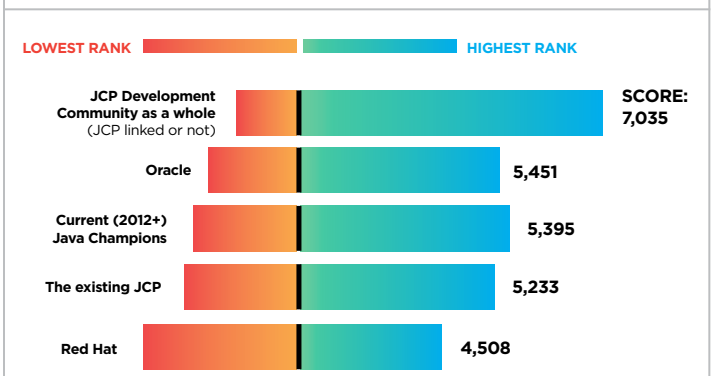
Spring usage, on the other hand, has increased significantly year over year, especially Spring 4.x (49% in 2016 vs. 38% in 2015).

Major caveat: these numbers reflect answers to slightly but meaningfully different questions, which make year-over-year comparison of Spring with Java EE particularly difficult. Last year's question asked "which platforms do *you* use;" this year's asked "which platforms do *you or your organization* use." Because Spring is more modular than Java EE, it is harder in the case of Spring than in the case of Java EE for an average developer to tell whether the "platform" is used by the entire organization, given that the developer is themselves already using Spring, depending on how modularly the developer understands "Spring." That is, any developer who uses any Java EE API will be able to answer that their organization uses Java EE; but a developer who uses

ARE YOU CURRENTLY USING MICROSERVICES?



RANK HOW MUCH YOU TRUST THE FOLLOWING DRIVERS/ STEWARDS OF THE JAVA LANGUAGE AND PLATFORM



Spring Boot to spin up a web app, for example, may or may not consider "my organization uses Spring" to be true in the context of a question that also includes "Java EE" versions as answer options. The current research project aims to discover further breakdown of enterprise Java platform usage by module and API; a breakdown with adequate detail proved too cumbersome to include in this survey, but is planned for a follow-up survey.

5. THE FUTURE OF JAVA

Java is obviously not dead; but is it moribund? Most evidence that bears on this question can be spun both optimistically and pessimistically. For example, perhaps the high rate of Java 8 "new-style" / functional feature adoption indicates that Java is responding well to developers' needs—or perhaps it indicates pent-up demand from a developer community dissatisfied with the glacial pace of the JCP and JEP. Again, maybe the increasing adoption of non-Java JVM languages indicates a thriving ecosystem built on a common bytecode; but whether this means that "Java" is getting stronger because the JVM is spawning more higher-level languages, or that "Java" is getting weaker because its secret WORA sauce is being leveraged by other languages, depends on the level at which you locate your concept of "Java."

Because this sort of evidence is highly polyvalent, and because so much of a language's vitality is felt while actually writing code (rather than answering survey questions), it seemed important to ask developers separately for their feelings about the future of the Java ecosystem. Moreover, individual experiences of a language ecosystem's "vitality"— visible in virtually any comment thread comparing Java and Scala, for example—vary so tremendously that aggregation of these sentiments seems particularly necessary. So we asked four explicit questions aimed at developers' thoughts about the future of Java.

QUESTION 1: How do you feel about the future of Java?

4. Overall respondents are optimistic: less than 6% are "pessimistic" (either "very" or "slightly") about the future of Java, and slightly more are "very optimistic" (43%) than "fairly optimistic" (40%). These results are virtually identical to last year's (as the differences for each answer response fall within both surveys' independent margins of error). This suggests that developers have detected no major new signals about the future of Java—despite the fact that Java 9 was delayed twice this year. Either the release date of Java 9 (at least within its current window) does not impact developers' overall optimism about the future of Java, or the reasons given for the delays (primarily around Jigsaw) in public discussions seem acceptable to most developers.

   Note: the first delay of Java 9 was announced in December 2015, after our 2015 survey closed; the second delay was announced briefly before our 2016 survey closed, but in time to affect about a fifth of responses received *if* those respondents were very up-to-date on the OpenJDK mailing list. So it seems probable that the majority of the effect of these delays on our data does not take into account the second delay announcement; but the exact amount is unknowable.

QUESTION 2: What is the most important new feature of Java 9?

Jigsaw, the higher-level modularity system coming in Java 9 (and cause of much of the delay), remains the most important new feature of Java 9 in developers' eyes, as in last year's survey. HTTP/2 support remains the second-most-important as well. Exact numbers are not comparable (this year we added a "no opinion" option), but the constant order offers quantitative confirmation of the generally accepted opinion that modularity will make the biggest difference to Java developers at large (and not just the developers of OpenJDK itself).

QUESTION 3: Are you currently using microservices?

Java applications are perhaps more likely than most to be affected by the "decentralizing" and "distributed" modern trends in application architecture—precisely because the Java language's strong object-orientation, high performance, and multi-platform capabilities are suited to large-scale systems, and because the Java platform provides so much rich functionality at so many levels and in so many problem domains. The importance of microservices to the Java community is reflected by, for example, the MicroProfile initiative, spearheaded by companies other than Oracle; the rise of Java frameworks suitable for easy spin-up of RESTful services, such as Spring Boot, Lagom, and WildFly Swarm; the growing popularity of shared-little architectures (e.g. '12-factor') and programming models (both functional and actor-oriented); and (as of two days ago) increased support for microservices in Java EE 8.

Developers' interest in microservices has been growing over the past year. In August 2015, 10% of our respondents were using microservices; in January 2016, 24% were using microservices somewhere in their organization; in a report just published by Lightbend, 30% are using microservices in production; and, in our latest survey, 39% of developers are currently using microservices, and another 19% are planning to adopt microservices in the next 12 months.

6. RANK HOW MUCH YOU TRUST THE FOLLOWING DRIVERS/STEWARDS OF THE JAVA LANGUAGE/PLATFORM.

Java more than any other language or platform has tested the relation between a programming language and its users. No other language has formalized a community-driven improvement process as thoroughly as the JCP (whatever its faults); no other platform has exposed APIs whose legal status has been addressed by multi-billion dollar lawsuits. Developers' trust in the stewards and drivers of Java is therefore essential to the health of the community and the technology ecosystem.

In light of recent concerns about Oracle's commitment to Java (some spurred by an ever more sluggish JCP, others by simple radio silence), it is encouraging to see that, after the Java development community as a whole (which is of course the most trusted steward by far), Oracle comes in (a distant) second place, significantly ahead of the JCP, although negligibly ahead of current (2012+) Java Champions. (Answer options are ranked by sum of weighted scores: because the question included 9 answer options, a first-place ranking of option **o** adds 9 points to **o**'s total, second-place ranking adds 8 points to **o**'s total, etc.)

ca technologies

See Your Users as People—Not Numbers

Manage and Maximize their experience with CA Application Performance Management

Behind the pretty face of today's applications can be a complex array of microservices, containers, APIs and back-end services. You need more than just data to deliver exceptional user experience. CA Application Performance Management provides the analytics and insights you need to truly understand and manage user experience – and make your customers happy.

Start your personalized demo today at:
ca.com/java

Make it simple, fast – See only what you need to see with role-based views that streamline complex app maps.

Locate the real problems – Avoid false alerts and see the real issues with built-in analytics.

Understand the impact of change – When problems arise, understand what, when and where your app changed

Five Tips to Effectively Monitor Application Performance and User Experience

When implementing an application performance monitoring strategy it can be tempting to just grab some tools and start using them. This can ultimately lead to choosing one or more disparate tools that are not integrated or holistic in their approach. Too many tools and too much data can actually lead to not enough insight into what is really going on with your apps or your users' experience. Here are five tips for success.

First, understand all of your customers. Monitor apps across mobile, web and wearables and include synthetic monitoring to find and fix problems even at times where you have no users. Leverage passive monitoring when security or other concerns prohibit direct end-user monitoring.

Second, make sure you can follow transactions from front-end to back-end. Transactions can cover a lot of ground from your app to APIs, security layers, middleware all the way to the back-end. Make sure your monitoring covers the same ground.

Third, get continuous feedback across DevOps by integrating monitoring across all parts of the SDLC. This is as much cultural as it is technical. Collaboration across Dev and Ops is critical to delivering great user experiences.

Fourth, understand how changes impact performance. Being able to roll back time to see what changed before an issue helps you find "patient zero" and resolve problems faster.

> Too many tools and too much data can actually lead to not enough insight into what is really going on with your apps.

Finally, simplify the complex! Modern apps can have a lot going on under the covers. Views and perspectives that remove layers of complexity help you see what is important more clearly, without a distracting data deluge.

Consider these tips and you'll be more successful in managing the performance of your applications - and help keep your customers happy.

WRITTEN BY DAVID HARDMAN
DIRECTOR, PRODUCT MARKETING, **CA TECHNOLOGIES**

PARTNER SPOTLIGHT

CA Application Performance Management By CA Technologies

Find and fix problems early in the development lifecycle with patent-pending innovations that speed and simplify triage.

CATEGORY
APM

NEW RELEASES
Quarterly

OPEN SOURCE
No

CASE STUDY

Orange has been offering communication services for more than 20 years. Today it provides mobile and landline telecommunications and broadband services to 244 million retail and business customers around the globe. An excellent customer experience is a strategic priority for Orange. But the performance of some applications on Orange.com was not up to par. CA APM plays a critical role in ensuring the overall quality of Orange's applications. It helps Orange assess the risk associated with an application prior to its release into a given environment. Orange can deliver the excellent online experience expected by today's increasingly connected customers with better reliability, availability and faster response times.

STRENGTHS

Modern application monitoring with broad support for application environments and programming language, based on an E.P.I.C. APM Strategy that is:

- **Easy**—Simplify the triage process through role based views and integrated timeline
- **Proactive**—Analytics that recognize problems as they develop and focus on the most critical issues
- **Intelligent**—Detect and monitor application processes and transactions automatically
- **Collaborative**—Enable better communication between Dev and Ops to resolve problems faster

NOTABLE CUSTOMERS

- Lexmark
- Vodafone
- Itau Unibanco
- U.S. Cellular
- Innovapost
- Produban
- Blue Cross Blue Shield of Tennessee
- National Australia Bank

BLOG bit.ly/catechblog | **TWITTER** @cainc | **WEBSITE** ca.com/apm

The Java 8 API Design Principles

QUICK VIEW

Learn to be a better Java programmer by mastering Java 8 API design and:

- Expose a well designed API and hide the implementation details
- Make sure that client code can use lamdas
- Ensure that the API can evolve in a controlled way
- Get rid of all those nasty NullPointerExceptions

BY **PER MINBORG**
CTO AT **SPEEDMENT, INC.**

Anyone that writes Java code is an API designer! It does not matter if the coders share their code with others or not, the code is still used; either by others, by themselves or both. Thus, it becomes important for all Java developer to know the fundamentals of good API design.

A **good API design** requires careful thinking and a lot of experience. Luckily, we can learn from other clever people like Ference Mihaly, whose blog post inspired me to write this Java 8 API addendum. We relied heavily on his checklist when we designed the Speedment API. (I encourage you all to read his guide.)

Getting it **right from the start** is important because once an API is published, a firm commitment is made to the people who are supposed to use it. As Joshua Block once said: "Public APIs, like diamonds, are forever. You have one chance to get it right, so give it your best." A well designed API combines the best of two worlds, a firm and precise commitment combined with a high degree of implementation flexibility, eventually benefiting both the API designers and the API users.

Why use a checklist? Getting the API right (i.e. defining the visible parts of a collection of Java classes) can be much harder than writing the implementation classes that makes up the actual work behind the API. It is really an art that few people master. Using a checklist allows the reader to avoid the most obvious mistakes, become a better programmer and save a lot of time.

API designers are strongly encouraged to **put themselves in the client code perspective** and to optimize that view in terms of simplicity, ease-of-use, and consistency—rather than thinking about the actual API implementation. At the same time, they should try to hide as many implementation details as possible.

DO NOT RETURN `null` TO INDICATE THE ABSENCE OF A VALUE

Arguably, inconsistent null handling (resulting in the ubiquitous `NullPointerException`) is the single largest source of Java applications' errors historically. Some developers regard the introduction of the `null` concept as one of the worst mistakes ever made in the computer science domain. Luckily, the first step of alleviating Java's `null` handling problem was introduced in Java 8 with the advent of the `Optional` class. **Make sure** a method that can return a no-value returns an `Optional` instead of `null`. This clearly signals to the API users that the method may or may not return a value. Do not fall for the temptation to use null over `Optional` for performance reasons. Java 8's escape analysis will optimize away most `Optional` objects anyway. **Avoid** using Optionals in parameters and fields.

DO THIS:

```
public Optional<String> getComment() {
        return Optional.ofNullable(comment);
    }
```

DON'T DO THIS:

```
public String getComment() {
        return comment; // comment is nullable
    }
```

DO NOT USE ARRAYS TO PASS VALUES TO AND FROM THE API

A significant API mistake was made when the `Enum` concept was introduced in Java 5. We all know that an `Enum` class has a method called `values()` that returns an array of all the Enum's distinct values. Now, because the Java framework must ensure that the client code cannot change the Enum's values (for example, by directly writing to the array), a copy of the internal array must be produced for each call to the `value()` method. This results in poor performance and also poor client code usability. If the `Enum` would have returned an unmodifiable `List`, that `List` could be reused for each call and the client code would have had access to a better and more useful model of the Enum's values. In the general case, **consider** exposing a `Stream`, if the API is to return a collection of elements. This clearly states that the result is read-only (as opposed to a `List` which has a `set()` method). It also allows the client code to easily collect the elements in another data structure or act on them on-the-fly. Furthermore, the API can lazily produce the elements as they become available (e.g. are pulled in from a file, a socket, or from a database). Again, Java 8's improved escape analysis will make sure that a minimum of objects are actually created on the Java heap. **Do not** use arrays as input parameters for methods either, since this—unless a defensive copy of the array is made—makes it possible for another thread to modify the content of the array during method execution.

DO THIS:

```
public Stream<String> comments() {
        return Stream.of(comments);
    }
```

DON'T DO THIS:

```
public String[] comments() {
        return comments; // Exposes the backing array!
    }
```

CONSIDER ADDING STATIC INTERFACE METHODS TO PROVIDE A SINGLE ENTRY POINT FOR OBJECT CREATION

Avoid allowing the client code to directly select an implementation class of an interface. Allowing client code to create implementation classes directly creates a much more direct coupling of the API and the client code. It also makes the API commitment much larger, since now we have to maintain all the implementation classes exactly as they can be observed from outside instead of just committing to the interface as such. **Consider** adding static interface methods, to allow the client code to create (potentially specialized) objects that implement the interface. For example, if we have an interface `Point` with two methods `int x()` and `int y()`, then we can expose a static method `Point.of(int x, int y)` that produces a (hidden) implementation of the interface. So, if `x` and `y` are both zero, we can return a special implementation class `PointOrigoImpl` (with no `x` or `y` fields), or else we return another class `PointImpl` that holds the given `x` and `y` values. **Ensure** that the implementation classes are in another package that are clearly not a part of the API (e.g. put the `Point` interface in `com.company.product.shape` and the implementations in `com.company.product.internal.shape`).

DO THIS:

```
Point point = Point.of(1,2);
```

DON'T DO THIS:

```
Point point = new PointImpl(1,2);
```

FAVOR COMPOSITION WITH FUNCTIONAL INTERFACES AND LAMBDAS OVER INHERITANCE

For good reasons, there can only be one super class for any given Java class. Furthermore, exposing abstract or base classes in your API that are supposed to be inherited by client code is a very big and problematic API commitment. Avoid API inheritance altogether, and instead **consider** providing static interface methods that take one or several lambda parameters and apply those given lambdas to a default internal API implementation class. This also creates a much clearer separation of concerns. For example, instead of inheriting from a public API class `AbstractReader` and overriding `abstract void handleError(IOException ioe)`, it is better to expose a static method or a builder in the Reader interface that takes a `Consumer<IOException>` and applies it to an internal generic `ReaderImpl`.

DO THIS:

```
Reader reader = Reader.builder()
    .withErrorHandler(IOException::printStackTrace)
    .build();
```

DON'T DO THIS:

```
Reader reader = new AbstractReader() {
        @Override
        public void handleError(IOException ioe) {
            ioe. printStackTrace();
        }
    };
```

ENSURE THAT YOU ADD THE `@FunctionalInterface` ANNOTATION TO FUNCTIONAL INTERFACES

Tagging an interface with the `@FunctionalInterface` annotation signals that API users may use lambdas to implement the interface, and it also makes sure the interface remains usable for lambdas over time by preventing abstract methods from accidently being added to the API later on.

DO THIS:

```
@FunctionalInterface
public interface CircleSegmentConstructor {

    CircleSegment apply(Point cntr, Point p, double ang);

        // abstract methods cannot be added
 }
```

CONTINUED

DON'T DO THIS:

```
public interface CircleSegmentConstructor {

    CircleSegment apply(Point cntr, Point p, double ang);

        // abstract methods may be accidently added later

}
```

AVOID OVERLOADING METHODS WITH FUNCTIONAL INTERFACES AS PARAMETERS

If there are two or more functions with the same name that take functional interfaces as parameters, then this would likely create a lambda ambiguity on the client side. For example, if there are two `Point` methods `add(Function<Point, String> renderer)` and `add(Predicate<Point> logCondition)` and we try to `call point.add(p -> p + " lambda")` from the client code, the compiler is unable to determine which method to use and will produce an error. Instead, **consider** naming methods according to their specific use.

DO THIS:

```
    public interface Point {
        addRenderer(Function<Point, String> renderer);
        addLogCondition(Predicate<Point> logCondition);
    }
```

DON'T DO THIS:

```
    public interface Point {
        add(Function<Point, String> renderer);
        add(Predicate<Point> logCondition);
    }
```

AVOID OVERUSING DEFAULT METHODS IN INTERFACES

Default methods can easily be added to interfaces and sometimes it makes sense to do that. For example, a method that is expected to be the same for any implementing class and that is short and "fundamental" in its functionality, is a viable candidate for a default implementation. Also, when an API is expanded, it sometimes makes sense to provide a default interface method for backward compatibility reasons. As we all know, functional interfaces contain exactly one abstract method, so default methods provide an escape hatch when additional methods must be added. However, **avoid** having the API interface evolve to an implementation class by polluting it with unnecessary implementation concerns. If in doubt, **consider** moving the method logic to a separate utility class and/or place it in the implementing classes.

DO THIS:

```
    public interface Line {
        Point start();
        Point end();
        int length();
    }
```

DON'T DO THIS:

```
    public interface Line {
        Point start();
        Point end();
        default int length() {
            int deltaX = start().x() - end().x();
            int deltaY = start().y() - end().y();
            return (int) Math.sqrt(
                deltaX * deltaX + deltaY * deltaY
            );
        }
    }
```

ENSURE THAT THE API METHODS CHECK THE PARAMETER INVARIANTS BEFORE THEY ARE ACTED UPON

Historically, people have been sloppy in making sure to validate method input parameters. So, when a resulting error occurs later on, the real reason becomes obscured and hidden deep down the stack trace. **Ensure** that parameters are checked for nulls and any valid range constrains or preconditions before the parameters are ever used in the implementing classes. **Do not** fall for the temptation to skip parameter checks for performance reasons. The JVM will be able to optimize away redundant checking and produce efficient code. Make use of the `Objects.requireNonNull()` method. Parameter checking is also an important way to enforce the API's contract. If the API was not supposed to accept nulls but did anyhow, users will become confused.

DO THIS:

```
public void addToSegment(Segment segment, Point point) {
        Objects.requireNonNull(segment);
        Objects.requireNonNull(point);
        segment.add(point);
}
```

DON'T DO THIS:

```
public void addToSegment(Segment segment, Point point) {
        segment.add(point);
}
```

PER MINBORG has been a Java Developer since Java 1.0 and runs a popular Java blog and the open-source project Speedment, which is a tool for accelerating development and execution performance of Java database applications. Per is a DZone MVB and has also held numerous presentations on various Java topics, for example at JavaOne in San Francisco and at other larger Java events.

Project Jigsaw Is Coming

BY **NICOLAI PARLOG**
FREELANCER AT **CODEFX**

QUICK VIEW

01 Java 9 brings modularity to the language.

02 Modules are like JARs but come with a descriptor that defines a name, dependencies, and an API.

03 Two basic rules, readability and accessibility, build on that and allow reliable configuration, strong encapsulation, improved performance, security, maintenance, and more.

04 Migration will not be without challenges and should be well prepared.

Java 9 is coming! It's just nine more months to its release, so this is a good time to familiarize ourselves with it. There are a couple of interesting new features, like Java's REPL (a.k.a. JShell), the additions to the Stream and Optional APIs, support for HTTP 2.0, and more. But they are all overshadowed by Java 9's flagship feature: Project Jigsaw, which will bring artifact-level modularity to the language.

Let's have a look at it!

(All unattributed quotes come from the excellent State of the Module System.)

CREATING MODULES

In Java 9 we will have the possibility to create modules. But what exactly is a module? It's much like a regular artifact (most commonly a JAR) but it has three additional properties, which it explicitly expresses:

- a name
- dependencies
- a well-defined API

This information is encoded in a *module descriptor* (in the form of `module-info.class`), which is compiled from a *module declaration* (`module-info.java`). The module descriptor defines the three properties stated above and we'll see in a minute how it does that.

After creating the `module-info.java` we (and in the future, our tools) have to compile it to `module-info.class` and then package it together with the rest of our source files. The result is a *modular JAR*. At runtime, the JVM will read the classes and resources along with `module-info.class` and turn it all into a module.

But how does the module declaration work, and how do the compiler and JVM interpret it?

(By the way, I assume reading "compiler and JVM" is as tiring as writing it, so I will forego some precision and use "Java" instead.)

MODULE DECLARATION

As stated above, a module declaration defines a module's name, dependencies, and API. It is usually defined in `module-info.java` and looks like this:

```
module MODULE_NAME {
    requires OTHER_MODULE_NAME;
    requires YET_ANOTHER_MODULE_NAME;
    exports PACKAGE_NAME;
    exports OTHER_PACKAGE_NAME;
    exports YET_ANOTHER_PACKAGE_NAME;
}
```

Let's examine the three properties one by one.

NAME

A module's name can be arbitrary, but to ensure uniqueness, it is recommended to stick with the inverse-URL naming schema for packages. Guava, for example, will likely be `com.google.guava`, and Apache Commons IO could be either `org.apache.commons.commons_io` or `org.apache.commons.io`.

While this is not necessary, it will often lead to the module name being a prefix of the packages it contains.

DEPENDENCIES

A module lists the other modules it depends on to compile and run by naming them in `requires` clauses. This is true for application and library modules, but also for modules in the JDK itself, which was split up into about 80 of them (have a look at them with `java -listmods`).

API

By default, all types are internal to the module and not visible outside of it. But surely some types *should* be visible to the outside, right? Yes, and this is done by exporting the packages that contain these types with the `exports` clause. Other code, and even reflection, can only access public types in exported packages.

JAR HELL AND OTHER NICETIES

That is all nice and dandy but… why do we need it? Well, there are some deep-seated problems with how Java handles artifacts.

Before Project Jigsaw, Java sees JARs as simple containers for compiled classes without any meaningful representation at compile or run time. It simply rips all of the classes it has to access out of their JARs and rolls them into one big ball of mud. The JARs themselves, left behind on the class path, are meaningless, and it does not matter at all how many there are and how they are structured. For all Java cares, there might just as well only be a single JAR.

This has a couple of negative consequences, and some of them are notable contributors to JAR Hell:

- Dependencies between artifacts can not be expressed, which means that many problems lead to runtime errors and crashing applications (`NoClassDefFoundError` anyone?).
- Loading classes requires linear scans of the class path.
- If there are several classes with the same fully qualified name, the first one found during the scan will be loaded and will shadow the others.
- There is no way to reliably run an application that depends on two versions of the same library (usually as transitive dependencies via other libraries it needs).
- There is no way to have code that is only visible inside a JAR.
- Security-relevant code cannot be made accessible to some JARs but hidden from others.

These problems can cause all kinds of trouble, like bad performance, lacking security, maintenance nightmares, and anything from too-subtle-to-notice misbehavior to havoc-wreaking errors. Hence, lots of tools and mechanisms were devised to tackle some problem or other from our list: build tools, web servers, component systems, Java's very own security manager, fiddling with class loaders, and so on. These generally work, but not without adding their own complexity and potential for errors. Sometimes considerable amounts of it!

It would be so much better if Java itself would have an understanding of artifacts…

MODULARITY WITH PROJECT JIGSAW

Enter Project Jigsaw! It was specifically designed to provide a solution to these problems. At the core of that solution are the modules that we already looked at and three other concepts:

- module graph
- readability
- accessibility

Together they want to achieve the project's goals, most notably among them:

- reliable configuration
- strong encapsulation
- improved security, maintainability, and performance

MODULE GRAPH

The information contained in module descriptors gives Java the ability to actually understand what's going on between modules. So, instead of the big ball of mud it created before, it can now map how they relate to each other.

More precisely, it builds a graph where the modules are nodes and where the dependencies (expressed by the `requires` clauses) are edges. Fittingly, this is called the Module Graph.

READABILITY

Directly based on this graph is the concept of *Readability*:

When one module depends directly upon another in the module graph, then code in the first module will be able to refer to types in the second module. We therefore say that the first module reads the second or, equivalently, that the second module is readable by the first.

RELIABLE CONFIGURATION

Readability is the basis of reliable configuration:

> The readability relationships defined in a module graph are the basis of *reliable configuration:* The module system ensures that every dependence is fulfilled by precisely one other module, that the module graph is acyclic, that every module reads at most one module defining a given package, and that modules defining identically-named packages do not interfere with each other.

So expressing and understanding dependencies means that a lot of the problems that used to crash an application can now be found at launch or even compile time!

IMPROVED PERFORMANCE

Readability also helps to improve performance. The module system now knows for any given class which module is supposed to contain it and can thus forego the repeated linear scans. And with clearer bounds of where code is used, existing byte-code optimization techniques can be used more effectively.

As JSR 376 puts it:

> Many ahead-of-time, whole-program optimization techniques can be more effective when it is known that a class can refer only to classes in a few other specific components rather than to any class loaded at run time.

It might also be possible to index annotated classes so that they can be found without a full class path scan.

ACCESSIBILITY

Together with readability, the `exports` clauses are the basis for *Accessibility*:

CONTINUED

The Java compiler and virtual machine consider the public types in a package in one module to be accessible by code in some other module only when the first module is readable by the second module, in the sense defined above, and the first module exports that package.

STRONG ENCAPSULATION

What happens if code tries to access a type that does not fulfill these requirements?

> A type referenced across module boundaries that is not accessible in this way is unusable in the same way that a private method or field is unusable: Any attempt to use it will cause an error to be reported by the compiler, or an `IllegalAccessError` to be thrown by the Java virtual machine, or an `IllegalAccessException` to be thrown by the reflective run-time APIs. Thus, even when a type is declared public, if its package is not exported in the declaration of its module then it will only be accessible to code in that module.

This means that public is no longer public! It also means that modules will be able to hide their internals and clearly define the parts of their functionality that make up their public API.

Mark Reinhold, spec lead on Project Jigsaw, once wrote about this:

> A class that is private to a module should be private in exactly the same way that a private field is private to a class.

It might take some time to get used to "public" no longer meaning "public for everyone" but "public for everyone in this module." I am convinced that this is worth it, though, as it finally allows us to create a safe zone within a module.

IMPROVED SECURITY AND MAINTAINABILITY

The strong encapsulation of module-internal APIs greatly improves security and maintainability. It helps with security because critical code is now effectively hidden from code which does not require to use it. It makes maintenance easier as a module's public API can more easily be kept small.

BIRTH PAINS

But not all is well…

MIGRATION CHALLENGES

Besides the core features we just discussed, Jigsaw entails a lot of changes under the hood. While almost all of them are backwards compatible in the strict meaning of the word, some interact badly with existing code bases. In the end, whether you modularize your application or not, running on Java 9 may break your code.

A couple of things that could cause trouble:

- Your (or, more likely, your dependencies) could depend on the JDK-internal API, which will soon be inaccessible thanks to strong encapsulation. See JEP 260 for details about which APIs will disappear. Also, have a look at *jdeps* and run `jdeps -jdkinternals <jars>` on your and your dependencies' artifacts.
- The JVM will refuse to launch when two modules contain packages with the same name (known as *split package*). If a module and a regular JAR split a package, the content of the JAR's package would not be visible at all.
- The JRE/JDK layout changed:
  - `rt.jar` and `tools.jar` no longer exist
  - JDK modules are packed as JMODs, a new and deliberately unspecified format
  - there is no longer a folder `jre` in the JDK
  - the URLs for runtime content look different
- Class loaders are no longer always `URLClassLoader`-s, so casts to that type will fail.
- The Endorsed Standards Override Mechanism, Extension Mechanism, and Boot Class Path Override are gone.

More possible problems are listed under *Risks and Assumptions* in JEP 261.

WHAT ABOUT VERSION CONFLICTS?

Unfortunately, the module system has no understanding of versions. It will see two different versions of the same module as a duplication and refuse to compile or launch. The fact that the module system does nothing to ameliorate version conflicts is, frankly, somewhat disappointing, and I believe we might soon be talking about module hell.

CONTESTED TOPICS

These are some of the questions currently being discussed on the Jigsaw mailing list:

- Should strong encapsulation be stronger than reflection?
- Do we need optional dependencies?
- Should modules be able to "stand in" for other modules (by aliasing)?
- Is it helpful that a module can make its dependencies available to other modules?

If these open questions or other details of Jigsaw interest you, make sure to check the mailing list archives or even participate in the discussion.

SUMMARY

The most important takeaway is that Jigsaw introduces modules: JARs with a module descriptor that gives them names, explicit dependencies, and a well-defined API. The module graph, readability, and accessibility build on these descriptors to tackle JAR hell and other existing problems as well as to provide reliable configuration, strong encapsulation, and improved security, maintainability, and performance. But Jigsaw will make some migrations to Java 9 daunting and is also being criticized for a number of shortcomings—perceived or real.

To see for yourself, download JDK 9, play around with it—maybe by following my hands-on guide—and try to judge or even spike your code's migration.

NICOLAI PARLOG is the editor of SitePoint's Java channel, writes a book about Project Jigsaw, blogs about software development on codefx.org, and is a long-tail contributor to several open source projects. You can hire him for all kinds of things.

Modernize your approach with microservices

With a microservices architecture built on IBM® Bluemix®, you can quickly and easily build, test and maintain complex applications that are agile, reliable, scalable, and secure.

Extend Java apps to mobile
Deliver mobile tools and apps fast that meet the continuous demand for new features and services.

Design intelligent systems for the cognitive era
Create innovative, smart Java-based apps that use language, vision, speech and data insight APIs.

Program faster, better, easier.
Learn more about the advantages of moving from a monolithic to microservices architecture.
ibm.com/java

Understand the What, Why and How of Using Microservices

Application architecture patterns are changing, a result of the convergence of factors that has led to the concept of "cloud native" applications. General availability of cloud computing platforms, advancements in virtualization, and the emergence of agile and DevOps practices have helped to streamline and shorten release cycles. Two-week roll outs for new functionality is simply too long for today's on-demand business world. This is the challenge that microservices, the cloud native application architecture, is trying to address.

Microservices empower developers to program faster, better, and more easily. Separating the bits of an application into pieces can be very freeing. The ability to isolate each piece can drastically reduce the overhead involved in making changes as well as the risk involved with trying something new.

But before you embark on a new project using a microservices architecture, there are factors you need to consider. For example, how you will manage your data, ensure your apps are secure, leverage logging and metrics to maintain a healthy infrastructure, and more. Together with WAS Liberty, IBM® has created a library of resources with information on how to get started, best practices, and methodologies.

> Separating the bits of an application into pieces can be very freeing.

Seeing is believing. That's why IBM and WAS Liberty developed an app to give developers a fuller picture of what a microservices application should look like. Game On! is a throwback text-based adventure built to help you explore microservices architectures and related concepts. It demonstrates the key aspects of microservice architectures, and it is infinitely extensible in a creative and fun way.

WRITTEN BY ERIN SCHNABEL
SENIOR SOFTWARE ENGINEER, **IBM**

PARTNER SPOTLIGHT

IBM Bluemix By IBM

IBM Bluemix is a cloud platform, built on open source, and offering flexible support for developers

CATEGORY
Cloud, PaaS, IaaS

NEW RELEASES
Continuous

OPEN SOURCE
Yes

CASE STUDY

The KLM Open, one of the oldest golf tournaments in the European Tour, attracts an average of 45,000 visitors and wanted to provide fans with a new and more interactive mobile application but lacked the infrastructure and expertise needed to develop and deploy such a solution. Turning to IBM's Bluemix platform, which combines the power of IBM SoftLayer, IBM MobileFirst Platform, and IBM WebSphere Application Server Liberty, they were able to deliver a new mobile application that helped improve the live experience for fans with mobile access to real-time tournament information, resulting in 7,500 downloads in three days and a 25% increase in mobile usage rate by fans during the tournament.

STRENGTHS

- Built on Cloud Foundry, supports OpenStack, Node, Docker & more
- Over 150 services across Watson, data, mobile, IoT, and DevOps
- Spans bare metal to serverless programming
- Delivered as public, dedicated, and on-premises deployment model

NOTABLE CUSTOMERS

- PayPal
- WhatsApp
- GitHub
- GameStop
- Apple
- Acccenture
- Tumblr
- Tata

BLOG developer.ibm.com/wasdev/blog

TWITTER @IBMBluemix

WEBSITE ibm.com/bluemix

REACTIVE MICROSERVICES:

Driving Application Modernization Efforts

QUICK VIEW

01 Traditional enterprises are forced to rethink how they build applications because of rapidly changing use-cases and new requirements.

02 Isolation, Single Responsibility, Autonomy, Exclusive State, Asynchronous Message-Passing, and Mobility are required to build out a reactive microservice architecture.

03 Reactive Principles provide the coherent approach to Microservices systems architecture.

BY **MARKUS EISELE**
DEVELOPER ADVOCATE AT **LIGHTBEND, INC.**

Even today, after almost two solid years, microservices are still one of the emerging topics in enterprises. With the challenges coming with a rising number of consumers and changing use cases for customers, microservices provide a more stable and less cost intensive way to build applications. Though this isn't the first time evolving technologies have shocked the well-oiled machine of software to its core, microservices adoption truly forces traditional enterprises to re-think what they've been doing for almost two decades. We've seen design paradigms change over time and project management methodologies evolve. But this time it looks like the influence is far bigger than anything we've seen before. And the interesting part is that microservices aren't new from the core.

A core skill of software architects is understanding modularization and components in software and designing appropriate dependencies between them. We've already learned how to couple services and build them around organizational capabilities, and it's in looking beyond those binary dependencies, that exciting part of microservices-based architectures comes into play — how independent microservices are distributed and connected back together.

Building an individual service is easy with all technologies. Building a system out of many is the real challenge because it introduces us to the problem space of distributed systems. This is a major difference from classical, centralized infrastructures. As a result, there are very few concepts from the old world which still fit into a modern architecture.

MICROSERVICES IN A REACTIVE WORLD

Up to now, the usual way to describe distributed systems has been to use a mix of technical and business buzzwords: asynchronous, non-blocking, real-time, highly-available, loosely coupled, scalable, fault-tolerant, concurrent, message-driven, push instead of pull, distributed, low latency, high throughput, etc. The Reactive Manifesto brought all these characteristics together, and it defines them through four high-level traits: Responsive, Resilient, Elastic, and Message driven.

Even if it looks like this describes an entirely new architectural pattern, the core principles have long been known in industries that require real-time IT systems, such as financial trading. If you think about a systems composed out of individual services, you will quickly realize how closely the Reactive world is related to microservices. Let's explore the four main characteristics of a Reactive microservices system closer.

RESPONSIVE

A responsive application satisfies the consumers expectations in terms of availability and real-time responses. Responsiveness is measured in **latency**, which is the time between request and response. Especially with many small requests by mobile or internet-connected devices, a microservices-based architecture can achieve this using the right design.

RESILIENT

A very high-quality service performs its function without any downtime at all. But failures do happen, and handling and recovering from the failure of an individual service in a gentle way without affecting the complete system is what the term **resiliency** describes. While monolithic applications need to be recovered completely and handling failures mostly comes down to proprietary infrastructure components, individual microservices can easily provide these features, for example by supervision.

ELASTIC

A successful service will need to be scalable both up and down in response to changes in the rate at which the service is used. While monoliths usually are scaled on a per server basis, the flexibility and elasticity that comes with microservices has a much greater potential to respond to changing workloads.

MESSAGE DRIVEN

And basically the only way to fulfill all the above requirements is to have loosely coupled services with explicit protocols communicating over messages. Components can remain inactive until a message arrives, freeing up resources while doing nothing. In order to realize this, non-blocking and asynchronous APIs must be provided that explicitly expose the system's underlying message structure. While traditional frameworks (e.g. Spring and Java EE) have very little to offer here, modern approaches like Akka or Lagom are better equipped to help implement these requirements.

ARCHITECTURAL CONCEPTS OF REACTIVE MICROSERVICE ARCHITECTURES

Becoming a good architect of microservice systems requires time and experience. However, there are attempts to guide your decision-making and experiences with frameworks providing opinionated APIs, features, and defaults. The following examples are built with the Lagom framework. It provides tools and APIs that guide developers to build systems out of microservices. The key to success is good architecture, rooted in a solid understanding of microservices concepts and best practices, and Lagom encapsulates these for you to use in the form of APIs.

ISOLATION

It is not enough to build individual services. These services also need to isolate their failures. Containing and managing them without cascading throughout all the services participating in a complete workflow is a pattern referred to as bulkheading. This includes the ability to heal from failure, which is called resilience. And this highly depends on compartmentalization and containment of failure. Isolation also makes it easier to scale services on demand and also allows for monitoring, debugging, and testing them independently.

AUTONOMY

Acting autonomously as a microservice also means that those services can only promise their behavior by publishing their protocols and APIs. And this gives a great amount of flexibility around service orchestration, workflow management, and collaborative behavior. Through communication over well defined protocols, autonomous services also add to resilience and elasticity. Lagom services are described by an interface, known as a service descriptor. This interface not only defines how the service is invoked and implemented, it also defines the metadata that describes how the interface is mapped down onto an underlying transport protocol.

```
public interface HelloService extends Service {
    ServiceCall<String, String> sayHello();

    default Descriptor descriptor() {
        return named("hello").withCalls(
                call(this::sayHello)
        );
    }
}
```

Services are implemented by providing an implementation of the service descriptor interface, implementing each call specified by that descriptor.

```
public class HelloServiceImpl implements HelloService {

    public ServiceCall<String, String> sayHello() {
        return name -> completedFuture("Hello " + name);
    }
}
```

SINGLE RESPONSIBILITY PRINCIPLE

One of the most pressing questions for microservices has always been about size. What can be considered "micro"? How big in terms of lines of code or jar-file size is the optimal microservice? But these questions really aren't at the heart of the problem. Instead, "micro" should refer to scope of responsibility. One of the best guiding principles here is the Unix philosophy: let it do one thing, and do it well. When you look at refactoring existing systems, it will help to find a verb or noun as an initial description of a microservice. If a service only has one single reason to exist, providing a single composable piece of functionality, then business domains and responsibilities are not tangled. Each service can be made more generally useful, and the system as a whole is easier to scale, make resilient, understand, extend, and maintain.

EXCLUSIVE STATE

Microservices are often called stateful entities: they encapsulate state and behavior, in a similar fashion to an Object, and isolation most certainly applies to state and requires that you treat state and behavior as a single

unit. Just pushing the state down to a shared database and calling it a stateless architecture isn't the solution. Each microservice needs to take sole responsibility for its own state and the persistence thereof. This opens up possibilities for the most adequate persistence technology, ranging from RDBMS to Event-Log driven systems like Kafka. Abstracted techniques such as Event Sourcing and Command Query Responsibility Segregation (CQRS) serve as the persistence technology for reactive microservices.

In Lagom, a `PersistentEntity` has a stable entity identifier, with which it can be accessed from the service implementation or other places. The state of an entity is persistent using Event Sourcing. All state changes are represented as events and those immutable facts are appended to an event log. To recreate the current state of an entity when it is started, all those events get replayed. You interact with a `PersistentEntity` by sending command messages to it. An entity stub can look like this:

```
public class Post
  extends PersistentEntity<BlogCommand, BlogEvent,
BlogState> {

  @Override
  public Behavior initialBehavior(Optional<BlogState>
snapshotState) {
    BehaviorBuilder b = newBehaviorBuilder(
        snapshotState.orElse(BlogState.EMPTY));
    // Command and event handler go here
    return b.build();
  }
}
```

The three type parameters of the extended `PersistentEntity` class define: the Command, the super class/interface of the commands; the Event, the super class/interface of the events; and the State, the class of the state.

ASYNCHRONOUS MESSAGE-PASSING

Communication between microservices needs to be based on asynchronous message-passing.

Besides being the only effective help in isolating the individual services, the non-blocking execution and Input/Output (IO) is most often more effective on resources. Unfortunately, one of the most common implementations of the REST architecture, REST over HTTP, is widely considered the default microservices communication protocol. If you're looking into this, you need to be aware that the implementations most often are synchronous and as such not a suitable fit for microservices as the default protocol. All Lagom APIs use the asynchronous IO capabilities of Akka Stream for asynchronous streaming and the JDK8 CompletionStage API for asynchronous computation. Furthermore, Lagom makes asynchronous communication the default: when communicating between services, streaming is provided as a first-class concept. Developers are encouraged and enabled to use asynchronous messaging via streaming, rather than synchronous request-response communication. Asynchronous messaging is fundamental to a system's resilience and scalability. A streamed message is a message of type Source. Source is an Akka streams API that allows asynchronous streaming and handling of messages. Here's an example streamed service call:

```
ServiceCall<String, Source<String, ?>> tick(int
interval);

default Descriptor descriptor() {
    return named("clock").withCalls(
        pathCall("/tick/:interval", this::tick)
    );
}
```

MOBILITY

Another requirement for microservices is the ability to run independent of the physical location. And asynchronous message-passing provides the needed decoupling, which is also called location transparency: this gives the ability to, at runtime, dynamically scale the microservice—either on multiple cores or on multiple nodes—without changing the code. This kind of service distribution is needed to take full advantage of cloud computing infrastructures with microservices.

A successful microservices services architecture needs to be designed with the core traits of Reactive Microservices in mind. Isolation, Single Responsibility, Autonomy, Exclusive State, Asynchronous Message-Passing, and Mobility are required to build out a reactive microservice architecture. The most interesting, rewarding, and challenging parts take place when microservices collaborate and build a complete system. This is the chance to learn from past failures and successes in distributed systems and microservices-based architectures.

You can learn even more about reactive microservice architectures by reading Jonas Bonér's free O'Reilly book, and you can get a deep-dive into Lagom with my own mini-book about implementing reactive microservices.

MARKUS EISELE (@myfear) is a Java Champion, former Java EE 7 Expert Group member, deputy lead for the Java community at German DOAG, founder of JavaLand, reputed speaker at Java conferences around the world, and a very well known figure in the Enterprise Java world. He is a developer advocate at Lightbend.

7 HABITS OF

Super Productive Java Developers

The core engineering team at Stormpath has a combined 153 years of professional experience in Java, so we surveyed them for their top advice, tips, and tricks. Here's what they had to say:

1. KNOW YOUR TOOLS (AND HAVE THE RIGHT ONES)

Thoroughly research the existing frameworks or libraries that could make your implementation easier before you begin. Also, have the right IDE and customize it for your project. Beyond the Java-specific toolkit, our developers deploy a veritable battalion of apps and services to increase their productivity:

CROSS-PLATFORM, OR PLATFORM-AGNOSTIC

- Private Internet Access: Easy VPN service (great for hotels and public networks)
- Franz: One chat app to rule them all! Franz supports Slack, Hipchat, Facebook Messenger, GChat, Whatsapp, and Telegram (and many others)
- RecordIt: Multimedia recorder that turns quick screencasts into animated gifs
- JWT Inspector: Decode, inspect, and debug JWTs from cookies, local storage, and requests, straight from your browser with this Chrome extension, by Stormpath

FOR MAC

- Bartender: Tame the Mac menu bar
- Karabiner: Keyboard customizer
- Be Focused Pro: Pomodoro technique timer for the menu bar
- Alfred: Enhanced Spotlight functionality (and a lot more)
- Jumpcut: Indispensable clipboard buffering app
- Riverflow: Workflow manager that assigns unique two-finger gestures to actions

2. WRITE TESTS FIRST

It sounds counterintuitive, but by thinking ahead to your testing you'll end up writing testable code.

3. LOVE THAT HTTPIE

Forget curl; httpie is where it's at, so learn to love the command line. This Swiss Army knife for developers is quite possibly the most powerful tool in your arsenal, and the most frequently overlooked.

4. BE PROACTIVE ABOUT PRODUCTIVITY

Productivity hacks abound, and we recommend you to devote some time to trying a few out to find the one that best suits your workflow.

GET UP AND MOVE

Take breaks. For real. Get up and walk away from your computer, think about something other than work. Apps like Pause can help force your brain to disengage from work and slow down.

5. AUTOMATE WHERE POSSIBLE

Scripts are your friend. Take the time to automate your repetitive tasks, even the simple ones. Those saved seconds add up, and can eliminate errors.

6. DON'T STAY STUCK

Start by not being afraid to spike and delete: Try out different approaches and explore not just their impact on your project and interaction with existing code, but also where you get stuck. Then, delete that and write some tests.

USE YOUR TEAM

Don't be afraid to ask questions on Stack Overflow or pair up with a friend or colleague. Getting a second set of eyes on your problem can get you unstuck in a fraction of the time.

OPEN A BOOK!

There are some amazing general and Java-specific reference texts on the Stormpath bookshelves that can, and have, gotten our team unstuck a time or two. These include Effective Java, Simple Java, Clean Code, and Design Patterns.

7. PAY IT FORWARD

Stormpath co-founder Les Hazlewood is also the founder and primary contributor to the open-source Java security framework Apache Shiro. He offers this advice: "Participate (actually code) in some great open source projects. A *lot*. As much as you can. There is simply nothing in the world that I know of that will expose you to the quantity and quality of great code written by senior developers than participating in multiple solid open source projects. You will learn more by looking at clean code and good design patterns than anything you could do on your own or what you would see by working on a few closed-source projects."

The Elements Of Modern Java Style

BY **MICHAEL TOFINETTI**
DEVELOPMENT TEAM LEAD AT **ROGUE WAVE SOFTWARE**

QUICK VIEW

01 Code style is an important aspect of software development.

02 When done consistently and correctly, code style can help improve the quality of software by increasing readability and the ability to discover potential defects earlier in development.

03 Conventions include basic code formatting, how to name objects and variables, as well as following consistent design patterns.

CODE STYLE AND HOW TO USE IT

Compilers and interpreters require the syntax of code to conform to a given programming language's grammar. Humans, however, require additional guides to translate the cold instructions of machine code into something that can be followed and understood. Code style can provide these cues, and can include things like basic text formatting, indentation, and small methods. Style can also apply to the use of design patterns, such as best practices for things like constructor chaining, exception handling, and synchronization.

Well-styled code is easy to read, like well-written prose. And easier-to-read code means it is easier to understand, and this means more robust, error-free software that developers are happy to work with.

Coding style also works best when it is consistent throughout the whole codebase. Sometimes, however, many years of development may exist where no style or minimal style is applied. In these cases, it is best not to rush in and change everything just for the sake of applying style. Apply changes slowly as code changes, starting first with all new files. Then update existing files only in those places that are changing, such as a method due to a bug fix. And once a particular file passes a threshold (such as 75%), the remaining non-styled sections can be updated.

It is also important to note that modern IDEs allow for the configuration of formatting rules; take advantage of this assistance where available.

PRINCIPLES OF MODERN JAVA STYLE

The most important aspect of style is simplicity, since the simpler the style the easier it is to remember and apply. No developer wants to memorize hundreds of rules and dozens of exception in order to write software.

There is an overall guiding principle that is commonly known as "The Boy Scout Rule for Software Development". This simply states that a developer should leave the code in a better state than when they found it, much like Boy Scouts and their creed to maintain camp sites.

HIGHLIGHTS OF CODING AND FORMATTING CONVENTIONS

CODE FORMATTING

Formatting consists of all the typological conventions that give code its basic appearance and includes indentation, the use of comments, and where braces appear.

Some best practices:

Nest code and use 4 spaces for indentation. This gives a good trade-off between emphasizing the text indentation at each level and does not push text too far into the line if multiple indentations levels are required.

Use spaces for indentation, never tab characters. Tabs are visually indistinguishable from spaces and can cause issues if both are mixed, since tab lengths can vary across environments. Using only spaces ensure that indentation is always consistent.

Break long lines and margins of 80, 120, or 132 characters. 80 is better for doing side-by-side comparison or multi-way merges, but will lead to more wrapped lines.

Use white space, blank lines and comments to improve readability.

BRACES

Braces have long since become a point of contention in any discussion of coding style. There are two main schools of thought on braces: Cozied and Aligned. Each has its own pros and cons, which are explained below.

Cozied Braces put the open brace right at the end of the originating statement and the closing brace on its own line. Associated keywords (like "else" to an "if") cozy up to the end brace on the same line.

```
if (condition) {
  statement;
} else {
  statement;
}
```

This style allows for reducing the amount of vertical space used by the code, showing more code on the screen at a time. However, compacting information in this way can reduce readability.

Aligned braces, alternatively, place the braces directly on top of each other.

```
if (condition)
{
  statement;
}
else
{
  statement;
}
```

This style sets off statements from the keywords surrounding it by introducing blank lines since each brace is on its own line, instantly improving readability. It also introduces consistency in the location of braces because they are always in the same place, directly below the first character of the statement that introduced them.

Aligned Braces are better for making the code consistent and symmetrical, and this especially becomes obvious when the statement before the first open brace is longer than a single line.

```
if (long condition with
    keyword and keyword and
    keyword and keyword)
{
    keyword;
}
```

The cozied brace version of the above requires an additional indentation to differentiate the continuation of the condition with the start of the execution block.

When using Aligned Braces, the formatting automatically aligns the text blocks, keeps the condition items together, and does not require additional formatting rules or indentation spaces.

It is for these reasons that Aligned Braces are preferred.

NAMING CONVENTIONS

Naming is an important part of code writing — picking an appropriate name that conveys meaning and is appropriate for the scope and lifespan of the construct.

In general, it is better to be too descriptive than too terse, but always consider the scope that the variable will exist in. Short names are preferable in smaller scopes, while longer names are more appropriate for longer-lived objects.

A short, single character is appropriate for a self-contained loop:

```
for (int i = 0; i < listSize; i++)
{
    if (condition)
    {
        sum += list.getItemAt(i);
    }
}
```

Larger-scoped variables require longer and more descriptive names:

```
private CommandProcessor sequentialCommandProcessor =
    new CommandProcessor();
```

This variable may be used throughout the class in various places, where each of the separate invocations are not simultaneously visible on the editor screen at the same time.

```
sequentialCommandProcessor.init();
...
sequentialCommandProcessor.addCommand(...);
...
sequentialCommandProcessor.execute();
...
sequentialCommandProcessor.cleanUp();
...
```

Having a descriptive name reduces the time spent attempting to figure out the intention of the variable.

As in the example above, longer variable names use Camel Caps to join words, and never exclude vowels for brevity. Acronyms should only have their first letter capitalized, such as `parseXml()`.

To maintain consistency across the code base, use the following naming conventions:

- Capitalize the first letter of ***Classes*** and ***Interfaces***.
- Start with a lower case letter for ***methodNames*** and ***variableNames***.
- Constants are in all ***UPPERCASE_WITH_UNDERSCORES***.
- Use single words in all lowercase for ***package*** names.

HIGHLIGHTS OF PROGRAMMING AND DESIGN CONVENTIONS

Programming conventions cover aspects of implementation, including items like Type Safety, Statements and Expressions, Chaining Constructors, Exception Handling, Assertions, Concurrency, Synchronization, and Efficiency.

Some general conventions:

- Always use braces for block statements, even if they are empty or a single line; this improves readability and prevents issues if those code blocks are changed in the future.
- Use parenthesis to clarify the order of operations.
- Use polymorphism to reduce the need for `switch` statements or the `instanceof` operator. `instanceof` is an

CONTINUED

expensive operation and can lead to performance issues if done repeatedly in a loop structure.

- If using `switch` statements, always use a `default:` case and put `break;` statements at the end of each block, including the default.

CHAINING CONSTRUCTORS

Object construction occurs frequently, and often times with various parameters to help simplify creation. The best practice in this case is to not write duplicate code and instead make each constructor do only the minimum work necessary, passing the remaining processing to the other constructors.

```
public ChainedConstructor()
{
    // Common setup
    ...
}

public ChainedConstructor(ObjectTypeA a)
{
    ChaintedConstrucor();
    this.a = a;
}

public ChainedConstructor(ObjectTypeA a, ObjectTypeB b)
{
    ChainedConstructor(a);
    this.b = b;
}
```

EXCEPTION HANDLING

One of the most important things a developer can do is ensure that the software never crashes, even in the event of unexpected circumstances. At run-time, many things can go wrong, from invalid user input to network interruptions. It is for this reason that all potential exception cases must be handled.

At the very least, run-time exceptions should always be logged. It is very rare that there is an exception that will truly never occur and can be ignored.

```
try
{
    ...
}
catch (IOException e)
{
    // Should never reach here
    logger.debug("Unexpected I/O exception:");
    logger.logStackTrace(e);
}
```

Catch exceptions in as small an exception scope as possible. Do not wrap a try block around a section of code and then only catch `java.lang.Throwable`. Some exception cases are easier to recover from than others; it is best not to lump non-recoverable errors (such as `java.lang.OutOfMemoryError`) with more reasonable expected exceptions (such as `java.lang.NumberFormatException` when converting a String into an Integer).

SYNCHRONIZATION

Synchronization is the enforcement that only a single thread shall have access to a particular portion of code or an object at one moment. The most important rule of maintaining data integrity in a threaded environment is to always allocate and synchronize on an object that is used exclusively for synchronization purposes. Java does provide a mechanism to apply synchronization to classes or methods, but this implicitly or explicitly uses the instance of the class object itself as the synchronization object. That means that all synchronized methods in the class will be blocked if a thread is locked in one of them.

As such, to prevent unintended consequences, always use an Object other than the current object (this) as the synchronization lock token.

```
private String fileLock = "token";

public void writeDataFile(String data)
{
    synchronized(fileLock)
    {
        dataFile.write(data);
    }
}

public String readDataFile(int lineNumber)
{
    String result;
    synchronized(fileLock)
    {
        result = dataFile.read(lineNumber);
    }
    return result;
}
```

Synchronization is an expensive operation that will slow down the execution of a code block, so only apply synchronization where necessary to avoid thread collisions or potential data corruption.

CONCLUSION

Code style is an important aspect of software development. Judiciously and consistently applying a well-defined style will produce code that is simpler to read, understand, and debug, and with fewer defects.

MICHAEL TOFINETTI received an Honours Bachelor of Computer Science from Lakehead University and has worked in software design and development for nearly twenty years, primarily in the telecommunications industry. He has co-authored eight patents (six issued, two pending), and is currently a Development Team Lead for Rogue Wave Software working on the Klocwork static code analysis tool in Ottawa, Canada.

Java or .NET? Be Technology-Agnostic with Cross-Platform Tools

With the proliferation of new platforms and languages, developers are faced with the challenge of being technology-agnostic, whereby they must mix and match whatever technologies are necessary to build the best possible system or to satisfy the end user. Rather than insisting on the purity of using a single platform or technology, the technology-agnostic developer selects parts of a solution from a wide array of platforms and other technologies with the aim of developing a superior or most suitable solution.

Consider the following scenario. A firm has spent years implementing a trading platform in Java. At some point they realize they must incorporate a best-of-breed quant package that's been written in C#. Does the development team resolve the problem by throwing away the C# library and rewriting it in Java?

Enter an interoperability solution. In this case, the firm easily and efficiently integrated the .NET-based quant API into their Java-based trading platform, maintaining the solidity and integrity of both code bases.

In another scenario, customers of a Java-based software provider demanded a .NET-based API. Rather than adding the time and expense to rewrite and maintain another set of code, the software team used an interoperability tool to quickly deliver a solution and make the sale.

> The technology-agnostic developer selects from a wide array of technologies in order to build superior applications and systems.

Being technology-agnostic during the development process allows you to choose the best overall solution regardless of the underlying platform. Being technology-agnostic also allows you to switch, rather than fight, when faced with adapting a solution long after the original technology decisions were made. Using interoperability tools helps you support your evolving technology needs.

WRITTEN BY WAYNE CITRIN
CTO AT **JNBRIDGE**

PARTNER SPOTLIGHT

JNBridgePro By JNBridge

Connect anything Java together with anything .NET.
Reuse your existing C#, VB or Java code, and run your applications anywhere.

CATEGORY
Java & .NET Interoperability

NEW RELEASES
Semi-Annual

OPEN SOURCE
No

CASE STUDY

A major financial institution that has spent years building up their Java-based trading infrastructure needed to integrate a risk engine API written in C#. They faced three choices: creating and using a very complex workaround process to assess the risk in each financial transaction (which would have cost significant processing time), completely rewriting and debugging the entire C# library in Java, or finding a way to make both sides work together seamlessly. After exploring open-source applications that didn't work, they chose JNBridgePro as the solution. Now, the risk-evaluation process takes a fraction of the time it otherwise would have, and the developers can instead focus on other business-critical items on their to-do list.

STRENGTHS

- Access Java classes from .NET as if Java were a .NET language (C#, VB, etc.)
- Access .NET classes (written in C#, VB, F#...) from Java as if they were Java classes
- Gain full access to any API on the other side, whether it's service-enabled or not
- Expose any Java or .NET binary, no source code required
- Deploy anywhere: same process, separate processes, separate devices, across a network

NOTABLE CUSTOMERS

Over 600 global enterprises and software development houses rely on JNBridge products in all kinds of applications, integrating Java with .NET across every layer from the UI to the enterprise server backend. See jnbridge.com/about/customers for details.

BLOG jnbridge.com/blog | **TWITTER** @jnbridge | **WEBSITE** jnbridge.com

vaadin
FRAMEWORK

OPEN SOURCE
Apache 2.0

The Java Web UI Framework for Business

100% JAVA

Write your UI in any language running on the JVM and use industry proven tools and techniques such as refactoring, testing and strong IDE support to build a robust scalable business application.

150 000
DEVELOPERS WORLDWIDE

AUTOMATED SERVER-CLIENT COMMUNICATION

Vaadin Framework automates the communication between the server and the browser. Hook your data to your UI components and have it automatically rendered as HTML5 over the web without tedious DTOs, JSON or JavaScript.

SINGLE PAGE APPS FOR DESKTOP, TABLET AND MOBILE

Run your applications on Mobile and Desktop on any browsers in HTML5. Enjoy your blazing fast development and build features for your customers twice as fast.

USED BY 40% OF THE FORTUNE 500

vaadin.com/framework

Modernizing Applications

Many companies are facing the problem of aging applications that either don't support customers' platforms or don't have a fast enough time to market. Web as a platform has already won the platform wars as a platform everybody supports. Even mobile apps are consolidating towards the web, with emerging browser features and the ease of findability compared to installable apps. Your technology stack however defines your time to market – developers can either solve business issues or use their time on technical issues.

Java as the basis for your technology stack ensures that your team can stay productive. The less technologies your team has to juggle the easier it is to get new people into your project or switch between different teams. Managing competencies gets easier the less technologies you have to master. If you already from before have a Java-stack it makes sense to re-use as much as possible of the UI and back-end logic. When deciding upon your migration path, consider the amount of technologies and languages you need to know. For instance Java over JavaScript has many advantages in this respect. It also makes for an easier migration when as much as possible can be reused.

As important as the language, and thus the tools it provides, is also the longevity of the framework you migrate to. Even though history doesn't always give promises about the future, having a solid back is worth a lot when building for years to come. You want to ensure your framework supports the platforms your future customers will be using.

Check out the framework comparison matrix @ vaadin.com/comparison.

WRITTEN BY FREDRIK RÖNNLUND
VP OF MARKETING, **VAADIN**

PARTNER SPOTLIGHT

Vaadin

vaadin }>

Use a familiar component based approach to build awesome single page web apps faster than with any other UI framework. Forget complex web technologies and just use Java or any other JVM language. No plugins or installations required and it works from desktop to mobile.

CATEGORY
Web UI Framework

NEW RELEASES
Every 2 Weeks

OPEN SOURCE
Yes

CASE STUDY

Simplifying the Development Model at CAS Software
CAS Software GmbH in Germany switched to Vaadin Framework for their main product in order to shave off development costs and get faster time to market.

Selecting the right tool for the company was imperative back in 2014 so they made a thorough comparison. During the evaluation phase several technologies were evaluated, including Eclipse RAP, Vaadin, GWT and Sencha/Ext JS. Vaadin was found to be the most solid technology and has proven to be the right choice.

"Vaadin enabled a highly economic assignment of tasks: the most of the application including the UI can be implemented by Java developers and only a few specialists were needed to implement complex widgets."

- Dr. Markus Bauer, Head of Development Center SmartDesign, CAS Software AG

STRENGTHS

- One language for the whole application
- Extensible with Java and HTML5
- UI logic is right next to your data
- Strong abstraction of web technologies
- Half the lines of code, twice the productivity
- Open source and backed up by a strong company
- Spring and Java EE compatible
- Supported for 15 years, 5 year support guarantee

NOTABLE CUSTOMERS

- TNT
- Nasa
- CGI
- Puma
- Accenture

BLOG vaadin.com/blog

TWITTER @vaadin

WEBSITE vaadin.com

12 Factors and Beyond in Java

BY **PIETER HUMPHREY** PRINCIPAL PRODUCT MARKETING MANAGER AT **PIVOTAL**
AND **MARK HECKLER** PRINCIPAL TECHNOLOGIST/DEVELOPER ADVOCATE AT **PIVOTAL**

QUICK VIEW

01 Be conservative when estimating skills, knowledge of users, and developers.

02 Establish a common understanding between IT and users of what the modernization drivers are so project components can be prioritized.

03 Keep business engaged and momentum up by following an incremental approach.

04 Look for opportunities to reuse the tested, working code in production.

05 Invest adequately in the understanding of the existing codebase.

For many people, "cloud native" and "12 factor applications" are synonymous. A goal of this article is to illustrate that there's a lot more to cloud native than just adhering to the original 12 factors. As with most things, Java is up to the task. In this article we'll examine concepts and code samples, taking a look beyond the standard 12 factors in the process, as Kevin Hoffmann does in his recent O'Reilly book Beyond the 12 Factor App.

1. ONE CODEBASE

While less of a Java-specific concept, this factor generally refers to getting to a single code base managed in source control or a set of repositories from a common root. Getting to a single codebase makes it cleaner to build and push any number of immutable releases across various environments. The best example of violating this is when your app is composed of a dozen or more code repositories. While using one code repository to produce multiple applications can be workable, the goal is a 1:1 relationship between apps and repos. Operating from one codebase can be done but is not without its own challenges. Sometimes one application per repository is the simplest thing that works for a team or organization.

2. DEPENDENCY MANAGEMENT

Most Java (and Groovy) developers can take advantage of facilities like Maven (and Gradle), which provide the means to *declare* the dependencies your app requires for proper build and execution. The idea is to allow developers to declare dependencies and let the tool ensure those dependencies are satisfied and packaged into a single binary deployment artifact. Plugins like Maven Shade or Spring Boot enable you to bundle your application and its dependencies into a single "uberjar" or "fat jar" and thus provide the means to *isolate* those dependencies.

Figure 1 is a portion of an example Spring Boot application Maven build file, **pom.xml**. This shows the dependency declarations as specified by the developer.

FIGURE 1: A PORTION OF POM.XML SHOWING APPLICATION DEPENDENCIES

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>1.3.7.RELEASE</version>
  <relativePath/> <!-- lookup parent from repository
-->
</parent>

<dependencies>
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-config</
artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-eureka</
artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-zipkin</
artifactId>
  </dependency>
```

Figure 2 is a portion of listed dependencies within the same application, showing JARs bundled into the application's uberjar, which isolates those dependencies from variations in the underlying environment. The application will rely upon these dependencies rather than potentially conflicting libraries present in the deployment target.

FIGURE 2: A PORTION OF MVN DEPENDENCY:TREE FOR A SAMPLE APPLICATION

```
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------
------------------
[INFO] Building quote-service 0.0.1-SNAPSHOT
[INFO] ------------------------------------------------------------
------------------
[INFO]
[INFO] --- maven-dependency-plugin:2.10:tree (default-cli) @
quote-service ---
[INFO] com.example:quote-service:jar:0.0.1-SNAPSHOT
[INFO] +- org.springframework.cloud:spring-cloud-starter-
config:jar:1.1.3.RELEASE:compile
[INFO] |  +- org.springframework.cloud:spring-cloud-starter:j
ar:1.1.1.RELEASE:compile
[INFO] |  |  +- org.springframework.cloud:spring-cloud-contex
t:jar:1.1.1.RELEASE:compile
[INFO] |  |  |  \- org.springframework.security:spring-
security-crypto:jar:4.0.4.RELEASE:compile
[INFO] |  |  +- org.springframework.cloud:spring-cloud-common
s:jar:1.1.1.RELEASE:compile
[INFO] |  |  \- org.springframework.security:spring-security-
rsa:jar:1.0.1.RELEASE:compile
[INFO] |  |     \- org.bouncycastle:bcpkix-
jdk15on:jar:1.47:compile
[INFO] |  |        \- org.bouncycastle:bcprov-
jdk15on:jar:1.47:compile
[INFO] |  +- org.springframework.cloud:spring-cloud-config-
client:jar:1.1.2.RELEASE:compile
[INFO] |  |  \- org.springframework.boot:spring-boot-autoconf
igure:jar:1.3.7.RELEASE:compile
[INFO] |  \- com.fasterxml.jackson.core:jackson-
databind:jar:2.6.7:compile
[INFO] |     \- com.fasterxml.jackson.core:jackson-
core:jar:2.6.7:compile
[INFO] +- org.springframework.cloud:spring-cloud-starter-
eureka:jar:1.1.5.RELEASE:compile
[INFO] |  +- org.springframework.cloud:spring-cloud-netflix-
core:jar:1.1.5.RELEASE:compile
[INFO] |  |  \- org.springframework.boot:spring-
boot:jar:1.3.7.RELEASE:compile
```

3. BUILD, RELEASE, RUN

A single codebase is taken through a build process to produce a single artifact; then merged with configuration information external to the app. This is then delivered to cloud environments and run. Never change code at runtime!

The notion of *Build* leads naturally to continuous integration (CI), since those systems provide a single location that assemble artifacts in a repeatable way. Modern Java frameworks can produce uberjars, or the more traditional WAR file, as a single CI-friendly artifact. The *Release* phase merges externalized configuration (see *Configuration* below) with your single app artifact and dependencies like the JDK, OS, and Tomcat. The goal is to produce a release that can be executed, versioned, and rolled back. The cloud platform takes the release and handles the Run phase in a strictly separated manner.

4. CONFIGURATION

This factor is about externalizing the type of configuration that varies between deployment environments (dev, staging, prod). Configuration can be everywhere: littered among an app's code, in property sources like YAML, Java properties, environment variables (env vars), CLI args, system properties, JNDI, etc. There are various solutions—refactor your code to look for environment variables.

For simpler systems, a straightforward solution is to leverage Java's `System.getenv()` to retrieve one or more settings from the environment, or a Map of all keys and values present. Figure 3 is an example of this type of code.

FIGURE 3: A PORTION OF POM.XML SHOWING APPLICATION DEPENDENCIE

```
private String userName = System.getenv("BACKINGSERVICE_UID");
private String password = System.getenv("BACKINGSERVICE_PASSWORD");
```

For more complex systems, Spring Cloud and Spring Boot are popular choices and provide powerful capabilities for source control and externalization of configuration data.

5. LOGS

Logs should be treated as event streams: a time-ordered sequence of events emitted from an application. Since you can't log to a file in a cloud, you log to `stdout/stderr` and let the cloud provider or related tools handle it. For example, Cloud Foundry's loggregator will turn logs into streams so they can be aggregated and managed centrally. `stdout/stderr` logging is simple in Java:

```
Logger log = Logger.getLogger(MyClass.class.getName());
log.setLevel(Level.ALL);

ConsoleHandler handler = new ConsoleHandler();
handler.setFormatter(new SimpleFormatter());

log.addHandler(handler);

handler.setLevel(Level.ALL);
log.fine("This is fine.");
```

6. DISPOSABILITY

If you have processes that takes a while to start up or shut down, they should be separated into a backing service and optimized to accelerate performance. A cloud process is disposable — it can be destroyed and created at any time. Designing for this helps to ensure good uptime and allows you to get the benefit of features like auto scaling.

7. BACKING SERVICES

A backing service is something external your app depends on, like a database or messaging service. The app should

declare that it needs a backing service via an external config, like YAML or even a source-controlled config server. A cloud platform handles binding your app to the service, ideally attaching and reattaching without restarting your app. This loose coupling has many advantages, like allowing you to use the circuit breaker pattern to gracefully handle an outage scenario.

8. ENVIRONMENTAL PARITY

Shared development and QA sandboxes have different scale and reliability profiles from production, but you can't make snowflake environments! Cloud platforms keep multiple app environments consistent and eliminate the pain of debugging environment discrepancies.

9. ADMINISTRATIVE PROCESSES

These are things like timer jobs, one-off scripts, and other things you might have done using a programming shell. Backing Services and other capabilities from cloud platforms can help run these, and while Java doesn't (currently) ship with a shell like Python or Ruby, the ecosystem has lots of options to make it easy to run one off tasks or make a shell interface.

10. PORT BINDING

In the non-cloud world, it's typical to see several apps running in the same container, separating each app by port number and then using DNS to provide a friendly name to access. In the cloud you avoid this micro-management—the cloud provider will manage port assignment along with routing, scaling, etc.

While it is possible to rely upon external mechanisms to provide traffic to your app, these mechanisms vary among containers, machines, and platforms. Port binding provides you full control over how your application receives and responds to requests made of it, regardless of where it is deployed.

11. PROCESS

The original 12-factor definition here says that apps must be stateless. But some state needs to be somewhere, of course. Along these lines, this factor advocates moving any long-running state into an external, logical backing service implemented by a cache or data store.

12.CONCURRENCY

Cloud platforms are built to scale horizontally. There are design considerations here—your app should be disposable, stateless, and use share-nothing processes. Working with the platform's process management model is important for leveraging features like auto-scale, blue-green deployment, and more.

13. BEYOND 12 FACTOR: TELEMETRY, SECURITY, API-FIRST DESIGN

The 12 Factors were authored circa 2012. Let's look at just a few of the many baseline capabilities from modern clouds that make your app more sustainable to run:

- Health alerts, cloud system metrics, logs
- Domain-specific telemetry
- Application performance monitoring (APM)

On Cloud Foundry, Java app logs can simply be directed to `stdout / stderr`, where they are streamed and aggregated for operators. Spring Boot makes JMX a snap, and commercial cloud platforms can provide advanced capabilities like APM.

Security external to your application, applied to application endpoints (URLs) with RBAC, is important on cloud platforms for SSO & OAUTH2 provider integration. Otherwise, security for multiple Java apps becomes unmanageable.

Beyond the 12 Factor App describes the API-first approach as "an extension of the contract-first development pattern, where developers concentrate on building the edges or seams of their application first. With the integration points tested continuously via CI servers, teams can work on their own services and still maintain reasonable assurance that everything will work together properly."

REPLATFORMING

In conclusion, it's important to realize that you don't need all 15 factors just to replatform an existing app to run on the cloud. This cloud native maturity model (expressed by a large financial services organization) illustrates the type of progression used to approach large, complex monolithic apps and "12 factorize" them incrementally.

PIETER HUMPHREY is a Consulting Product Marketing Manager responsible for Java Developer Marketing at Pivotal Software, Inc. Pieter comes from BEA/Oracle with long history of developer tools, Java EE, SOA, EAI, application server and other Java middleware as both a marketing guy and sales engineer since 1998. You can find him on twitter discussing Java, Spring and the Cloud.

MARK HECKLER is a Pivotal Principal Technologist & Developer Advocate, conference speaker, published author, & Java Champion focusing upon developing innovative production-ready software at velocity for the Cloud and IoT applications. Mark is an open source contributor and author/curator of a developer-focused blog and an occasionally interesting Twitter account.

Diving Deeper

INTO JAVA

TOP #JAVA TWITTER FEEDS TO FOLLOW RIGHT AWAY

 @reza_rahman

 @starbuxman

 @trisha_gee

 @lukaseder

 @arungupta

 @myfear

 @mariofusco

 @dblevins

 @omniprof

 @javinpaul

JAVA ZONES LEARN MORE & ENGAGE YOUR PEERS IN OUR JAVA-RELATED TOPIC PORTALS

Java

dzone.com/java

The largest, most active Java developer community on the web. With news and tutorials on Java tools, performance tricks, and new standards and strategies that keep your skills razor-sharp.

Web Dev

dzone.com/webdev

Web professionals make up one of the largest sections of IT audiences; we are collecting content that helps web professionals navigate in a world of quickly changing language protocols, trending frameworks, and new standards for user experience. The Web Dev Zone is devoted to all things web development—and that includes everything from front-end user experience to back-end optimization, JavaScript frameworks, and web design. Popular web technology news and releases will be covered alongside mainstay web languages.

DevOps

dzone.com/devops

DevOps is a cultural movement, supported by exciting new tools, that is aimed at encouraging close cooperation within cross-disciplinary teams of developers and IT operations/system admins. The DevOps Zone is your hot spot for news and resources about Continuous Delivery, Puppet, Chef, Jenkins, and much more.

TOP JAVA REFCARDZ

Learn Microservices in Java

dzone.com/refcardz/learn-microservices-in-java

A practical guide complete with examples for designing Java microservices to support building systems that are tolerant of failure.

Java Containerization

dzone.com/refcardz/java-containerization

Includes suggested configurations and extensive code snippets to get your Java application up and running inside a Docker-deployed Linux container.

Core Java

dzone.com/refcardz/core-java

Gives you an overview of key aspects of the Java language and references on the core library, commonly used tools, and new Java 8 features.

TOP JAVA WEBSITES

JavaEE-Guardians.io

An independent group of people concerned about Oracle's current lack of commitment to Java EE who are doing all they can to preserve the interests of the Java EE community.

ProgramCreek.com

A site dedicated to posting high quality community-submitted Java tutorials and articles.

Java-Source.net

A well-organized directory of open source software focused on Java

TOP JAVA BOOKS

Java 8 Lambdas: Functional Programming for the Masses

If you're a developer with core Java SE skills, this hands-on book takes you through the language changes in Java 8 triggered by the addition of lambda expressions.

Java Concurrency in Practice

"This book is a must-read for anyone who uses threads and cares about performance."

Java SE 8 for the Really Impatient

This short book gives an introduction to the many new features of Java 8 (and a few features of Java 7 that haven't received much attention) for programmers who are already familiar with Java.

Pivotal **Cloud Foundry**®

Cloud Native Java At Your Service

Install, Deploy, Secure & Manage Spring Cloud's Service Discovery, Circuit Breaker Dashboard, and Config Server capabilities automatically as Pivotal Cloud Foundry — managed services today.

Engineered for apps built with Spring Boot

A distributed platform engineered for distributed Spring Cloud Apps

Cloud Native stream and batch processing with Spring Cloud Data Flow

Pivotal®

Microservices and Cloud Native Java

Spring Cloud provides tools for Spring Boot developers to quickly apply some of the common patterns found in distributed systems (e.g. configuration management, service discovery, circuit breakers, intelligent routing, micro-proxy, control bus, and much much more). When moving into production, Cloud Native applications can leverage a unique technology for automation of foundational microservice infrastructure. Spring Cloud Services for Pivotal Cloud Foundry packages the certain server-side components of Spring Cloud, making them available as native managed services inside Pivotal Cloud Foundry (PCF). For DevOps teams, this creates powerful autoscaling & automation possibilities for your services via:

- PCF-managed NetflixOSS Eureka
- PCF-managed NetflixOSS Hystrix Dashboard
- PCF-managed Git based Config Server from Pivotal

> Spring Cloud provides tools for Spring developers to quickly apply some of the common patterns found in distributed systems

When authoring client-side application code to use these services, developers can use native extensions to Spring Boot to wield microservice technology like Eureka, Hystrix, Zuul, Atlas, Consul, Zookeeper, Zikpin, as well as abstractions for common AWS services like RDS, SQS, SNS, Elasticache, S3, and Cloud Formation namespaces. The most recent major addition is Spring Cloud Data Flow, a cloud native programming and operating model for (streaming or batch) data microservices on structured platforms. Taking advantage of these battle-tested microservice patterns, and of the libraries that implement them, can now be as simple as including a starter POM in your application's dependencies and applying the appropriate annotation.

WRITTEN BY PIETER HUMPHREY
PRINCIPAL MARKETING MANAGER, **PIVOTAL**

PARTNER SPOTLIGHT

Spring Cloud Services By Pivotal

Pivotal

Spring Cloud Services for Pivotal Cloud Foundry packages server-side components of certain Spring Cloud projects and makes them available as managed services in Pivotal Cloud Foundry.

CATEGORY
Java Solution for Distributed Computing

NEW RELEASES
As Needed

OPEN SOURCE
No

CASE STUDY

Spring Cloud Netflix provides NetflixOSS integrations for Spring Boot apps through autoconfiguration and binding to the Spring Environment and other Spring programming model idioms. With a few simple annotations you can quickly enable and configure the common patterns inside your application and build large distributed systems with battle-tested Netflix components. The patterns provided include Service Discovery (Eureka), Circuit Breaker (Hystrix), Intelligent Routing (Zuul), and Client Side Load Balancing (Ribbon). Eureka instances can be registered and clients can discover the instances using Spring-managed beans, and an embedded Eureka server can be created with declarative Java configuration.

STRENGTHS

- Automated production operation on Cloud Foundry
- Microservice Security: OAUTH2, HTTPS, PCF UAA integration, RBAC across apps
- Scriptable, automated install & configuration
- Zero Downtime updates and upgrades
- Service-level HA for Config Server & Service Registry (in addition to Pivotal Cloud Foundry HA)
- Automatic provisioning and configuration of MySQL and RabbitMQ dependencies

PROMINENT TEAM MEMBERS

- Chris Sterling
- Ben Klein
- Scott Frederick
- Chris Schaefer
- Craig Walls
- Roy Clarkson
- Will Tran
- Mike Heath
- Chris Frost
- Glyn Normington

BLOG spring.io/blog | **TWITTER** @SpringCloudOSS | **WEBSITE** cloud.spring.io

JAVA'S IMPACT ON THE MODERN WORLD

Java's impact on the world is very far-reaching. It may not be new to the programming language game, but it is still very much present in the modern world. According to GitHub stats, Java has been the most popular programming language since 2005. Since it has been the predominant language for so long, almost every major industry has a major investment in Java. Here, we've illustrated where Java is all around us, how many Java developer jobs are listed in that industry based on a LinkedIn Jobs search, and what it's used for in a wide array of industries.

SPACE

NASA uses Java to simulate physics in space.

Java is cross-platform, and it is easy to integrate NASA's existing legacy systems into new projects.

739 space-related jobs for Java developers

MOBILE TECHNOLOGIES

Android devices are all driven by Java libraries.

Java is the official language for developing Android apps.

1,627 jobs for Java/Android developers.

HOSPITALS

All the management of hospital functions and reporting run on Java.

It is the basis of multi-institutional medical information retrieval systems.

972 hospital jobs for Java developers.

CITY INFRASTRUCTURE

Java is used by Hitachi to monitor and control water and sewage systems in cities across desktop and mobile devices.

Java is used to track energy usage in embedded devices, from single appliances to a city grid, to make developers aware of how much power they're using.

317 Java city infrastructure jobs.

MILITARY

Java is used in military mobile apps transitioning away from Ada for more secure applications.

Any military-controlled devices, such as drones and anti-aircraft defense systems, are using Java.

402 military jobs for Java developers.

GAMING

Java has moved beyond what XCode can do in creating virtual and augmented reality games.

Minecraft was written in Java, which has over 24 million in sales and was acquired by Microsoft for 2.5 billion dollars.

641 Java game developer jobs.

CONSUMER ELECTRONICS & IoT

100% of Blu-Ray players ship with Java.

Java Smart Cards (the same chips on your credit card) are used to deploy Java apps to rugged devices.

3,499 embedded device Java developer jobs and 980 IoT jobs.

BANKS

Use Tomcat - an open source version of Java web service technologies – for their APIs and web apps.

Java is a good alternative to COBOL as it is a modern language and can be documented more easily.

12,160 jobs for Java developers in financial services.

AUTOMOTIVE

Responsible for operating the accelerometer, thermal sensors for tires and breaks, in-car heart rate monitors, and touchscreen controls.

Java and JavaFX are responsible for collecting and visualizing data such as GPS, engine load, fuel pressure, air intake temp.

173 jobs for automotive Java developers.

FROM DESKTOP TO WEB JAVA:

A Planner's Guide to Migration

BY **BEN WILSON**
SENIOR CONSULTANT AT **VAADIN**

QUICK VIEW

01 Be conservative when estimating developer skills and user knowledge

02 Establish a common understanding between IT and users on the modernization drivers and their priorities.

03 Keep business engaged and preserve momentum with incremental improvements.

04 Look for opportunities to reuse the tested, working code in production.

05 Invest adequately in understanding the existing codebase

Companies with desktop applications written in Java are under increased pressure to provide their users the freedom of a responsive web experience. Ideally you could just run a desktop-style application inside a browser, and for a long time applets have made it possible to do just that. However, applets work using a protocol called NPAPI, which has been deprecated in most recent browsers. Today, the only real option to run a web application is to get pure HTML5 into the browser — not easy for most desktop developers given the new set of styling, scripting, and markup languages they have to learn.

There are many benefits of moving applications to the web, but some of the most challenging software projects have always been application migrations. What do successful migration projects look like? How do you eliminate a desktop system that has reliably run a part of the business for years, stay in budget with a migration, and delight users with a new browser application all at once?

AVOID THESE COMMON PITFALLS

#1: OVERESTIMATING TECHNICAL SKILLS

Overestimating technical skills can lead to both delays in producing code and increased defects, since developers do a poorer job of predicting side-effects and remembering new conventions.

Skills are one of the trickiest things to estimate correctly. The DevOps teams maintaining the current applications will be proficient in the current set of technologies, but often forget the years of effort that went into acquiring this proficiency. Coding conventions, naming conventions, namespace organization, software architecture, and testing approaches are just some of the choices that require more than casual experience to establish. However, most will be needed from the very start of any serious developments of the new software.

The problem is illustrated in the four stages of competence model: unless the migration can happen in a way that leverages developer skills, transition for most will be from unconscious competence in the old to unconscious incompetence in the new. This black hole of unconsciousness is a terrible starting point for rational planning and realistic effort estimation of a complex technical exercise.

#2: OVERRELIANCE ON USERS FOR REQUIREMENTS

Relying on users for specifications can result in surprises, especially towards the end of the project when acceptance starts, since many features are invisible to them.

Users typically have many years of experience using the application but may not have comprehensive specifications on all features that exist, those that exist and work, those that work and are still used, and those that are used and used only in specific conditions.

Application migration initiatives often start with lacking or outdated documentation artifacts. For large applications it is also likely that there is no one single person in the organization who knows every feature of the application, but that many people have a small piece of the puzzle. In these cases, requirements can only be elicited through meetings and the occasional groupthink.

#3: RUNNING OUT OF STEAM

Frequent feedback and reinforcement from both users and developers breathe life into long-running projects.

Modernization efforts are prone to being cancelled midway. While the first version of the current information system might have gone into production in one go, business applications undergo all manner of corrective and adaptive maintenance as user needs evolve. Repeated in many theories of software evolution, enterprise applications change — but importantly also grow — over time. Compounded over multiple years, legacy applications can grow to include a large number of features that can make managing testing and acceptance activities a project in itself.

As long as the new system does not fulfill all of the feature requirements, users will see they have little choice but to continue using the system in production. Prolonged periods of not being able to deliver value to business are lethal to software projects, and many modernizations die this way.

TIPS ON MAKING THE MIGRATION A SUCCESS

#1: DO APM FIRST

APM helps organizations focus their modernization efforts where they are needed most.

Application Portfolio Management (APM) is a tool to help organizations identify which applications they are running and where the source code is, so it is a useful starting point for planning a migration anyway. But APM also goes further to help us analyze the value of the applications in our portfolios and allows us to plan modernization efforts strategically. With it we recognize that even if we select a framework as a strategic technical platform for the future, not all applications in our organization need to be rewritten to use this framework with equal urgency, if at all.

APM comes in various flavors, and one of the most widely known is Gartner's TIME APM framework. APM frameworks typically categorize applications according to technical and business criteria, where an application can score either high or low on either. Applications that score high on business criteria (for instance, the application is an enabler of a key differentiator for the company) and low on technical criteria (for instance, high risk or cost of ownership) benefit the most from migration and should be prioritized.

#2: FIND A WAY TO REUSE AT LEAST SOME PARTS OF THE EXISTING DESKTOP JAVA

In migration, the shortest and most agile path is reuse.

An easy way to convince a business that a new application is ready to be accepted is if the old and new applications, with identical data, can execute identical business processes to yield identical results. This is much easier when data structures and services of the production and modernized systems correspond or can at least be matched using a simple conversion.

Another reason why artifact reuse is worthwhile is that migration projects of large applications can easily take over a year to complete. In that time users will still expect new features to be implemented in the existing systems, and other updates may be required for regulatory compliance or to fix critical defects. In this dynamic setup, we want to make sure that things we change in the production system can be demonstrated to also work as intended in the new system. The more that production business logic, services, interfaces, and data structures can be leveraged in the new modernized version, the easier this synchronization becomes.

#3: FIND AN INCREMENTAL APPROACH THAT WORKS FOR YOUR ORGANIZATION

Lower risk and the perception of risk by limiting scope.

It's likely that the business has a long list of wishes, and that developers have a far-reaching plan to deploy a future-proof IT architecture. When considering larger business applications, it is unlikely that an attempt to migrate big-bang style all code, while fulfilling all business wishes and all IT wishes, will succeed.

Migration projects don't happen for their own sake. For IT and the business to agree on an incremental modernization approach that will succeed, it is useful to understand which factors are actually relevant for driving the modernization and where the emphasis lies. Commonly there will be a combination of internal drivers (newer, better supported technology) and external ones (business agility and user freedom).

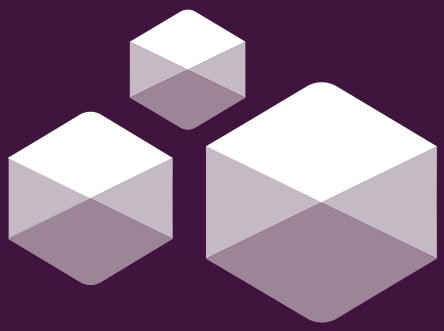
Following this exercise, there are different incremental approaches that could be considered:

- Modernize first the back-end, then the front-end;
- Fully modernize and replace one module or view at a time;
- Perform first a technical migration and then a modernization;
- Embed mini browser components inside the desktop application to make a gradual shift to web technology possible;
- Create an abstraction layer for the UI framework. This is often more expensive than doing a complete UI layer re-write, but allows running and developing the legacy version during the migration phase.

CONCLUSION

Migration projects are daunting, especially when applications are large. Companies used to typical software maintenance scenarios are not used to scoping and executing these kinds of projects and many fail. APM, the appropriate reuse of existing code, and incremental approaches, improve the chances of success significantly. Getting help from software companies with experience in migrations is always a good idea if migrations are unprecedented for the organizations taking the step.

BEN WILSON is a senior software professional with over ten years' specialized experience automating the modernization of large mission-critical enterprise applications, with a focus on UI modernization. Ben has worked on numerous projects in the banking, telecoms, government, automotive, and aerospace sectors, and currently works at Vaadin as a senior consultant.



lagom

Open source.
Highly opinionated.

Build greenfield microservices & decompose your Java EE monolith like a boss.

Get involved at
lagomframework.com

Lightbend

One Microservice Is No Microservice. They Come in Systems.

Individual microservices are comparatively easy to design and implement; however, learning from Carl Hewitt, just one microservice is no microservice at all—they come in systems.

Like humans, microservices-based systems act autonomously and therefore need to communicate and collaborate with others to solve problems—and as with humans, it is in collaboration that both the most interesting opportunities and challenging problems arise.

The challenge of building and deploying a microservices-based architecture boils down to all the surrounding requirements needed to make a production deployment successful. For example:

- Service discovery
- Coordination
- Security
- Replication
- Data consistency
- Deployment orchestration
- Resilience (i.e. failover)
- Integration with other systems

Lagom Framework is a comprehensive and prescriptive microservices platform that does the heavy lifting for distributed systems, and offers essential services and patterns built on a solid foundation of the Reactive principles.

> With Lagom, the Java community may have a powerful new tool in its arsenal for creating, managing, and scaling microservices to meet the rigorous demands of today's applications.
>
> -KIRAN "CK" OLIVER, NEWSTACK.IO

Built using technologies proven in production by some of the most admired brands in the world, Lagom is the culmination of years of enterprise usage and community contributions to Akka and Play Framework. Going far beyond the developer workstation, Lagom combines a familiar, highly iterative code environment using your existing IDE, DI and build tools, with additional features like service orchestration, monitoring, and advanced self-healing to support resilient, scalable production deployments.

WRITTEN BY MARKUS EISELE
DEVELOPER ADVOCATE, **LIGHTBEND, INC.**

PARTNER SPOTLIGHT

Lagom Framework By Lightbend

Lightbend

"Java finally gets microservices tools." -Infoworld.com

CATEGORY
Microservices Framework

NEW RELEASES
Multiple times per year

OPEN SOURCE
Yes

CASE STUDY

Hootsuite is the world's most widely used social media platform with more than 10 million users, and 744 of the Fortune 1000. Amidst incredible growth, Hootsuite was challenged by diminishing returns of engineering pouring time into scaling their legacy PHP and MySQL stack, which was suffering from performance and scalability issues. Hootsuite decomposed their legacy monolith into microservices with Lightbend technologies, creating a faster and leaner platform with asynchronous, message-driven communication among clusters. Hootsuite's new system handles orders of magnitude more requests per second than the previous stack, and is so resource efficient that they were able to reduce Amazon Web Services infrastructure costs by 80%.

STRENGTHS

- Powered by proven tools: Play Framework, Akka Streams, Akka Cluster, and Akka Persistence.
- Instantly visible code updates, with support for Maven and existing dev tools.
- Message-driven and asynchronous, with supervision and streaming capabilities.
- Persistence made simple, with native event-sourcing/CQRS for data management.
- Deploy to prod with a single command, including service discovery and self-healing.

NOTABLE CUSTOMERS

- Walmart
- Hootsuite
- Zalando
- Samsung
- UniCredit Group
- iHeart

BLOG lagomframework.com/blog | **TWITTER** @lagom | **WEBSITE** lagomframework.com

HASH TABLES, MUTABILITY, AND IDENTITY:

How to Implement a Bi-Directional Hash Table in Java

QUICK VIEW

01 Mutability can affect the behavior of hash tables.

02 Misunderstanding mutability can cause objects stored in a hash table to become irretrievable and essentially disappear.

03 It is possible to implement hash tables where any object — no matter how complex or mutable — can be used as a key.

BY **WAYNE CITRIN**
CTO AT **JNBRIDGE**

Data structures like hash tables and hash maps are essential for storing arbitrary objects and efficiently retrieving them. The object, or value, is stored in the table and associated with a key. Using the key, one can later retrieve the associated object. However, there are situations where, in addition to mapping keys to objects, one might also want to retrieve the key associated with a given object. This problem has been encountered time and again, and the solution is much trickier.

For example, we must efficiently keep track of the underlying remote objects corresponding to proxies when developing our JNBridge product. We do that by maintaining a table mapping unique keys to objects. This is straightforward to implement, using hash tables and maps available in many libraries. We also must be able to obtain an object's unique key given the object — provided it has one. (Yes, some libraries now offer *bimaps*, which purport to implement this functionality, but these didn't exist when we first needed this, so we had to implement it on our own. In addition, these bimaps don't provide everything we need. We'll discuss these issues later.)

Following are some helpful hints in implementing these forward/reverse map pairs.

HASHING ALGORITHM REQUIREMENTS

Let's first review the requirements that must be fulfilled by whatever hashing algorithm is used, and how it relates to equality. While a hashing method can be implemented by the developer, it is expected to obey a contract. The contract in general cannot be verified by the underlying platform, but if it is not obeyed, data structures that rely on the hashing method may not behave properly.

In Java, for example, the contract of the `hashCode()` method is:

- For any given object, the `hashCode()` method must return the same value throughout execution of the application (assuming that information used by the object's equality test also remains unchanged).
- If two objects are equal according to the objects' equality test, then they must both have the same hash code.

Connected with the `hashCode()` method is an equality test, which should implement an equivalence relation; that is, it must be reflexive, symmetric, and transitive. It should also be consistent: It should yield the same result throughout execution of the application (again, assuming the information used by the equality test doesn't change).

In addition to these contracts, Java provides guidelines for the use and implementation of hash codes, although these are not binding and may just be advisory. While guidelines for some non-Java frameworks suggest that hash codes for mutable objects should only be based on aspects of the objects that are immutable — so that the hash codes never change — in Java culture, the guidelines for implementing `hashCode()` are less strict, and it is quite likely that an object's hash code can change. For example, the hash code of a hash table object can change as items are added and removed. However, one informal guideline suggests that one should be careful using an object as a key in a hash-based collection when its hash code can change. As you'll see, the potential mutability of hash codes is a crucial consideration when implementing reverse maps.

If an object's hash code changes while it's stored inside a data structure that depends on the hash code (for example, if it's used as a key in a hash table), then the object may never be retrieved, as it will be placed in one hash bucket as it's added to the hash table, but looked for later in another hash bucket when the hash code has changed.

IMPLEMENTATION ASSUMPTIONS

So, what does this mean when forward/reverse map pairs must be implemented? Let's start with some assumptions:

- The *forward map* maps from keys to values. The keys are a particular type that one chooses in advance, and values can be any object. The value objects can be implemented by anyone, and their hash codes and equality tests may not conform to implementation guidelines — they may not even obey the required contract.
- The *reverse map* maps from the user-provided values back to keys.
- For simplicity, the user-defined object cannot be null. (Although, if necessary, this can be accommodated, too, through some additional tests.)
- Keys and user-defined objects should be unique; that is, they should not be used in the tables more than once.

Since the key is under our control, a developer can use objects of any immutable class (for example, `Integer` or `Long`), and avoid the possibility that the key's hash code changes. For the forward map, a simple hash table or hash map can be used.

The reverse map is more of a problem. As discussed above, one cannot rely on the hash code method associated with the user-provided objects that are used as keys. While some classes, particularly Java base classes, may have well-behaved hash code functions, other Java classes might not. Therefore, it's unsafe to trust that user-defined classes will obey these rules, or that the programmers that defined them were even aware of these guidelines.

FIND AN IMMUTABLE ATTRIBUTE

Therefore, one must come up with an attribute that every object has, that is guaranteed to never change, even when the contents of the object do change. The attribute's value should also be well-distributed, and therefore suitable for use in hashing. One such immutable attribute is the object's identity. When a hash table, for example, has items added to it, it's still the same hash table, even though the computed hash code might change. Fortunately, Java provides a hashing method based on object identity: `java.lang.System.identityHashCode()`, which is guaranteed to return the same value on a given object, even if its state has changed, since it is guaranteed to use `java.lang.Object`'s `hashCode()` method, which is identity-based, even if the target class overrides that method.

Java actually provides a class `java.util.IdentityHashMap`, which uses identity hashing. The developer could have used `IdentityHashMap` for his reverse hash table, except that, unlike the Java `Hashtable`, `IdentityHashMap` is not synchronized, and hash tables must be thread-safe.

CREATING IDENTITY-BASED HASHING

In order to write one's own identity-based hash tables, the developer must first ensure that, no matter what object is used as the key, identity-based hashing is always used. Unfortunately, the hash methods for these classes can't be overridden, since they're out of the developer's control. Instead, the key objects must be wrapped in classes that are in the developer's control, and where he can control the way the hash values are computed. In these wrappers, the hash method simply returns the identity-based hash code of the wrapped object. In addition, since identity-based hashing is being used, one must also use reference-based equality, so that two objects are equal if — and only if — they're the same object, rather than simply equivalent objects. In Java, a developer must use the "==" operator, which is guaranteed to be reference equality, rather than `equals()`, which can be, and often is, redefined by the developer.

In Java, our identity-based wrappers look like this:

```
final class IdentityWrapper
{
   private Object theObject;

   public IdentityWrapper(Object wrappedObject)
   {
      theObject = wrappedObject;
   }

   public boolean equals(Object obj2)
   {
      if (obj2 == null) return false;
      if (!(obj2 instanceof IdentityWrapper)) return false;
      return (theObject == ((IdentityWrapper)obj2).theObject);
   }

   public int hashCode()
   {
      return System.identityHashCode(theObject);
   }
}
```

Once these wrappers have been defined, the developer has everything he needs for a reverse hash table that works correctly:

```
Hashtable ht = new Hashtable();
…
ht.put(new IdentityWrapper(mutableUserDefinedObject), value);
…
mutableUserDefinedObject.modify();
…
Value v = (Value) ht.get(new
IdentityWrapper(mutableUserDefinedObject));
// retrieved v is the same as the value that was initially added.
```

If the `IdentityWrapper` classes are not used, the `ht.get()` operation is not guaranteed to retrieve the proper value.

At this point, the developer has all that's needed to implement bi-directional hash tables.

THE TROUBLE WITH OTHER LIBRARIES

What about other existing libraries? In particular, what about Google's Guava library, which implements a `HashBiMap` class, as well as other classes implementing a `BiMap` interface? Why not use that, and avoid reinventing the wheel? Unfortunately, while `HashBiMap` implements a forward/reverse hashmap pair and makes sure that the two are always in sync, it does not use identity hashing, and will not work properly if one of the keys or values is a mutable object. This can be seen by examining the `HashBiMap` source code. So, while `HashBiMap` solves part of the problem of implementing forward/reverse hashmap pairs, it does not address another, arguably more difficult part: the problem of storing mutable objects. The approach described here solves that issue.

IN CONCLUSION

This piece discusses an important, but unfortunately somewhat obscure, issue in the implementation of hash tables: the way in which mutability can affect the behavior of hash tables, and the way in which misunderstanding the issue can cause objects stored in a hash table to become irretrievable and essentially disappear. When these issues are understood, it becomes possible to implement hash tables where any object, no matter how complex or mutable, can be used as a key, and where bi-directional hash tables can be easily created.

WAYNE CITRIN is the CTO of JNBridge, the leading provider of interoperability tools that connect Java and .NET that he cofounded 15 years ago. Previously, Wayne was a leading researcher in programming languages and compilers and was on the Computer Engineering faculty at the University of Colorado, Boulder. He has a PhD from the University of California, Berkeley in Computer Science.

Executive Insights on the State of the Java Ecosystem

BY **TOM SMITH**
RESEARCH ANALYST AT **DZONE**

QUICK VIEW

01 Java continues to be an important element of enterprise IT given the breadth, depth, longevity, and diversity of the ecosystem.

02 The Open Source community is driving more innovation and development of the Java ecosystem than Oracle.

03 Java will continue to be prominent in enterprise development in the near term; however, it will likely be replaced by more specific languages in the long term.

To gather insights on the state of the Java ecosystem today, we spoke with 14 executives who are familiar with the Java ecosystem. Here's who we talked to:

JOEL DEPERNET, E.V.P. Global Research and Development, Axway
SACHA LABOUREY, CEO and Founder, CloudBees
RICK REICH, CEO, Development Heroes
ASAD ALI, Principal Software Developer, Dynatrace
LISA HAMAKER, Marketing Manager, Dynatrace
DAVID PARK, V.P. of Products, HackerRank
CHARLES KENDRICK, Founder and CTO, Isomorphic Software
WAYNE CITRIN, CTO, JNBridge
RAYMOND AUGÉ, Senior Software Architect, Liferay
LAURA KASSOVIC, Founder, MbientLab
CAMERON WILBY, Co-Founder, Origin Code Academy
JAMES FAULKNER, Technology Evangelist, Red Hat
PAUL TROWE, CEO, Replay Games
CALVIN FRENCH-OWEN, CTO and Co-Founder, Segment

KEY FINDINGS

01 The most important components of the Java ecosystem are the **JVM, its versatility, its breadth and depth as a result of its age, and the Open Source movement driving the availability of free online content for all developers**. The JVM platform runs on all machines and the cloud. The single most important part of the Java ecosystem is the vast amount of free online content (answered questions, tutorials, etc.) and free libraries. It is rare to encounter a development task in Java where you cannot find at least a partial solution, or hints at a solution, within a few minutes of searching.

02 The single biggest event in the past year was **Oracle's lawsuit against Google**. The entire industry was waiting to see whether the Java platform, and software development in general, was going to change radically to accommodate the fact that licenses would be required just to create a compatible implementation of an API. The Java platform will be forever diminished by Oracle's stewardship, since they have shown themselves to be fundamentally hostile to Open Source by filing the lawsuit in the first place, while vendors like Red Hat and the Open Source community have been pushing Java more than Oracle.

03 The impact of Java 8 so far is that **Lambda expressions make it easier for developers to build applications**. It's easier to write cleaner code. It's more maintainable with less repetition. Oracle has finally updated the language features to match those of other modern languages. The uptake of Java 8 depends on enterprises' desire and need to do so. Adoption can take years, which can be very frustrating. While there's been a lot of attention on security fixes, enterprises are slow to adopt due to lack of support. There's not a lot of buy-in to get to 8, even though it has better security and garbage collection.

04 There's little to no anticipation for Java 9 at this point, considering the slow adoption of Java 8. Vendors won't use Java 9 until they see some uptake by their enterprise clients. While **Java 9 offers modularity**, some people expressed fear and confusion over how the modularity is being handled. Is there misrepresentation of what will be possible? It's not playing out as Oracle would like due to lack of uptake. A lot of software is already offering modularity. Doing this in a closed environment makes it difficult to close the gap.

05 **Perspectives on the Java Community Process (JCP), and its future, are depressed**. JCP changes have been difficult given the situation with Oracle and their desire to milk Java like an annuity without letting others have too much freedom with the product. There's a perception that Java 8 and 9 will lead to a more closed ecosystem. This perception is driven by the fear of vendor lock-in by what's currently going on with cloud platforms. Some people see the JCP stagnating and becoming non-existent. The future of the JCP lies with stewards like the Apache Software Foundation, the Eclipse Foundation, and the OSGi Alliance. The Java community is strong and will keep the language viable for a while; however, Oracle needs to open it up and let others contribute if Java is going to have a robust future.

06 The **greatest value seen in Java is its diversity**: it's scalable, it's portable, it works on many machines, and it works on many different operating systems. It's not a one trick pony and can't be compared to other languages. It makes the lives of companies much better. Java is one of the top three languages in the world for quality, reliability, ability to deliver on demand, toolchains, and developer ecosystem. It's the only language that can make this claim. It's still the most powerful and comprehensive language. Lastly, unlike other languages, there are plenty of developers.

07 **The most common issue affecting the Java ecosystem is Oracle**. If Oracle would go to an Open Source model, Java would grow faster if everyone took the code and improved it, rather than waiting for Oracle. Oracle is a dubious steward and Java is inappropriate for a number of tasks that it's being used for, and it will ultimately be replaced for specific needs. There's a lack of visibility of the leadership of Java within Oracle. Oracle owns Java but isn't stepping up to push Java forward. There's a lack of innovation within Oracle and therefore a slow uptake of the newer versions of Java by enterprises.

08 **The future of Java continues to be strong in the near term** but diminishing over the long term (10+ years). Even with all of the new languages being developed, Java's not going anywhere for the next five to 10 years since it's the preferred language of enterprises because it scales, as well as saving them time and money. It will remain the predominant language in coding for quite a while since a lot of enterprises have a lot of equity invested in Java. There will be a slow loss of relevance over time until ultimately it's seen as a specialized skill needed only by a few, similar to the way mainframe languages are currently seen. There will be fewer developers using Java in the areas where Java has core strengths and Java will ultimately be phased out in favor of more appropriate languages and tools.

09 When we asked executives what developers need to keep in mind when working with Java, we received a variety of opinions. One set revolved around the philosophy that **companies are generally looking for the best programmers and engineers** with less concern for a particular language. As such, developers should focus on developing their coding skills and knowing the fundamentals of computer science, as well as its real-world applications. Another group feels that having full knowledge of Java is a good basic skill to have, and developers should get to know the common Java libraries and tools. Learn to appreciate the tools that are available and leverage them to the max. Always be learning while having a primary skill to ensure you have stable employment. Look for a secondary skill that will provide you with "developer insurance."

10 When asked about **additional considerations** with regard to the Java ecosystem, executives raised several questions and made some important observations:

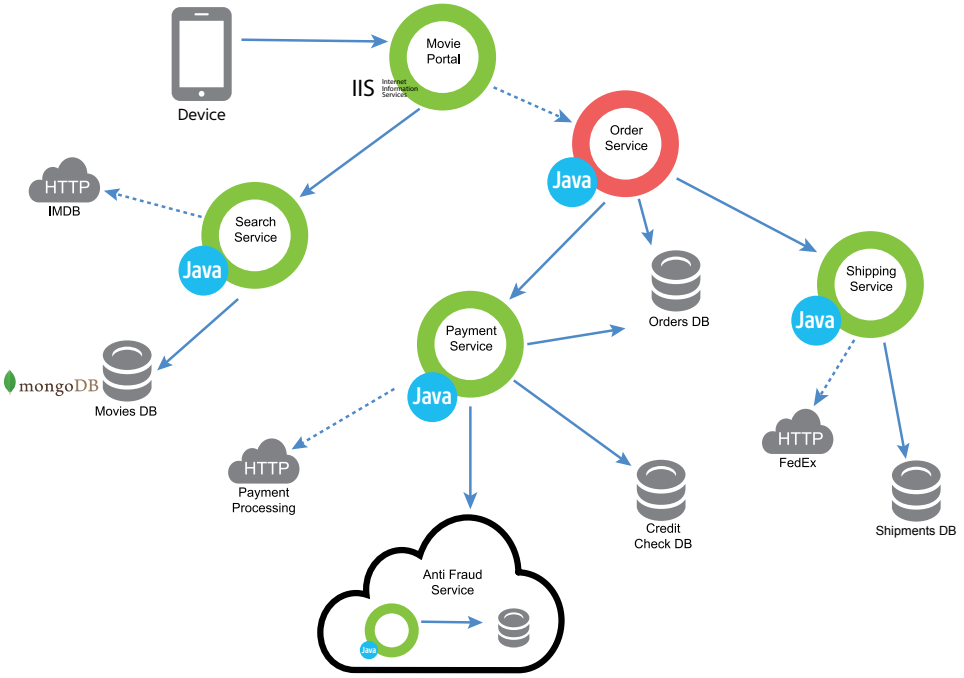
- What could Java be doing better?
- There will be more opportunities with mobile. Will Java keep up the pace?
- As more and more infrastructure moves to the cloud, or cloud-like provisioning, will these services run on Java or JVMs at all?
- Java is notorious for all the zero-day exploits and is only second to Adobe Flash in the number of vulnerabilities and security patches.
- Is Oracle going to make Java programming more flexible?
- Why aren't more companies making contributions to the Java community?

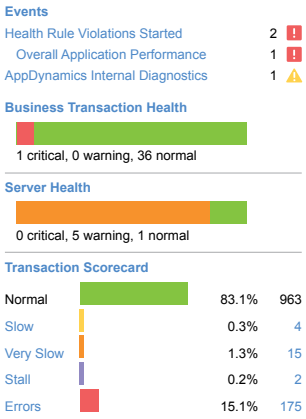
TOM SMITH is a Research Analyst at DZone who excels at gathering insights from analytics—both quantitative and qualitative—to drive business results. His passion is sharing information of value to help people succeed. In his spare time, you can find him either eating at Chipotle or working out at the gym.

APPDYNAMICS

There's nothing about Java that AppDynamics doesn't see.

AppDynamics gives you the visibility to take command of your Java application's performance, no matter how complicated or distributed your environment is.





Start your Free Trial

When your business runs on Java, count on AppDynamics to give you the complete visibility you need to be sure they are delivering the performance and business results you need — no matter how complex, distributed or asynchronous your environment, live 'in production' or during development.

See every line of code. Get a complete view of your environment with deep code diagnostics and auto-discovery. Understand performance trends with dynamic baselining. And drastically reduce time to root cause and remediation.

See why the world's largest Java deployments rely on the AppDynamics Application Intelligence Platform. Sign up for a FREE trial today at www.appdynamics.com/java.

What's Exciting About Java 9 and Application Performance Monitoring

In today's modern computing age, constant enhancements in software innovations are driving us closer to an era of software revolution. Perhaps in the distant future, that may be how the 21st century is remembered best. Among the popular software languages out there, however, Java continues to have the largest industry footprint, running applications around the globe producing combined annual revenue in trillions. That's why keeping up on the JDK is a high priority. Despite having a massive API to improve programming productivity, Java has also grown due to its high performance yet scalable JVM runtime, building among the fastest computing modern applications. As Java's footprint expands, JDK innovations continue to impact billions of lines of code. As AppDynamics continues to grow, our focus towards supporting Java is only furthered by our customer use & industry adoption of the JVM.

WRITTEN BY AAKRIT PRASAD
HEADING CORE & APM PRODUCTS, PRODUCT MANAGEMENT, **APPDYNAMICS**

PARTNER SPOTLIGHT

Application Intelligence Platform By AppDynamics

APPDYNAMICS

If your business runs on apps, Application Intelligence is for you. Real-time insights into application performance, user experience, and business outcomes.

CATEGORY
Application Performance Management

NEW RELEASES
Bi-Yearly

OPEN SOURCE
No

CASE STUDY

"AppDynamics has enabled us to move towards data-driven troubleshooting rather than 'gut-feels.' The solution gives us the application intelligence to know when things aren't functioning optimally."

- Nitin Thakur, technical operations manager, Cisco

STRENGTHS

Application Performance Management is a technology solution that provides end-to-end business transaction-centric management of the most complex and distributed applications. Auto-discovered transactions, dynamic baselining, code-level diagnostics, and Virtual War Room collaboration ensure rapid issue identification and resolution to maintain an ideal user experience.

NOTABLE CUSTOMERS

- NASDAQ
- eHarmony
- DIRECTV
- Cisco
- Citrix
- Hallmark

BLOG blog.appdynamics.com | **TWITTER** @AppDynamics | **WEBSITE** appdynamics.com

Solutions Directory

Java gets even greater when you have the right tools to back you up. This directory contains libraries, frameworks, IDEs, and more to help you with everything from database connection to release automation, from code review to application monitoring, from microservice architectures to memory management. Amp up your Java development with these solutions to make your life easier and your application more powerful.

PRODUCT	COMPANY	TYPE	FREE TRIAL	WEBSITE
Akka	Lightbend	Java implementation of Actor Model	Open Source	akka.io
AngularFaces	-	AngularJS + JSF	Open Source	angularfaces.com
Ansible	Red Hat	Deployment automation + configuration management	Open Source	ansible.com
ANTLR	-	Parser generator (for creating compilers and related tools)	Open Source	antlr3.org
AnyPoint Platform	MuleSoft	Hybrid integration platform	Free Trial	mulesoft.com/platform/enterprise-integration
Apache Ant	Apache Software Foundation	Build automation (process-agnostic: specify targets + tasks)	Open Source	ant.apache.org
Apache Camel	Apache Software Foundation	Java implementation of enteprrise integration patterns	Open Source	camel.apache.org
Apache Commons	Apache Software Foundation	Massive Java package collection	Open Source	commons.apache.org/components.html
Apache Commons DBCP	Apache Software Foundation	Database connection pooling	Open Source	commons.apache.org/proper/commons-dbcp
Apache Commons IO	Apache Software Foundation	Utilities for Java I/O (part of Apache Commons)	Open Source	commons.apache.org/proper/commons-io
Apache CXF	Apache Software Foundation	Java services framework with JAX-WS and JAX-RS support	Open Source	cxf.apache.org
Apache DeltaSpike	Apache Software Foundation	Portable CDI extensions (bean validation, JSF enhancements, invocation controls, transactions contexts, more)	Open Source	deltaspike.apache.org
Apache Ignite	Apache Software Foundation	In-memory Data Grid	Open Source	ignite.apache.org
Apache Ivy	Apache Software Foundation	Dependency management with strong Ant integration)	Open Source	ant.apache.org/ivy
Apache Kafka	Apache Software Foundation	Distributed pub-sub message broker	Open Source	kafka.apache.org
Apache Log4j	Apache Software Foundation	Logging for Java	Open Source	logging.apache.org/log4j/2.x
Apache Lucene	Apache Software Foundation	Search engine in Java	Open Source	lucene.apache.org/core
Apache Maven	Apache Software Foundation	Build automation (opinionated, plugin-happy, higher-level build phases, dependency management/resolution)	Open Source	maven.apache.org
Apache Mesos	Apache Software Foundation	Distributed systems kernel	Open Source	mesos.apache.org

PRODUCT	COMPANY	TYPE	FREE TRIAL	WEBSITE
Apache MyFaces	Apache Software Foundation	JSF + additional UI widgets, extensions, integrations	Open Source	myfaces.apache.org
Apache OpenNLP	Apache Software Foundation	Natural language processing machine learning toolkit	Open Source	opennlp.apache.org
Apache POI	Apache Software Foundation	Microsoft document processing for Java	Open Source	poi.apache.org
Apache Shiro	Apache Software Foundation	Java security framework (authen/author, crypto, session management)	Open Source	shiro.apache.org
Apache Struts	Apache Software Foundation	Web framework (Servlet + MVC)	Open Source	struts.apache.org
Apache Tapestry	Apache Software Foundation	Web framework (pages&components=POJOs, live class reloading, opinionated, light HttpSessions)	Open Source	tapestry.apache.org
Apache Tomcat	Apache Software Foundation	Servlet container + web server (JSP, EL, Websocket)	Open Source	tomcat.apache.org
Apache TomEE	Apache Software Foundation (esp. Tomitribe)	Apache Tomcat + Java EE features (CDI, EJB, JPA, JSF, JSP, more)	Open Source	tomee.apache.org
Apache Wicket	Apache Software Foundation	Simple web app framework (pure Java + HTML with Ajax output)	Open Source	wicket.apache.org
Apache Xerces2	Apache Software Foundation	XML parser for Java	Open Source	xerces.apache.org/xerces2-j
AppDynamics	AppDynamics *	APM with Java agent	Free Tier Available	appdynamics.com
Artifactory	JFrog	Binary/artifact repository manager	Open Source	jfrog.com/artifactory
ASM	OW2 Consortium	Java bytecode manipulation and analysis framework	Open Source	asm.ow2.org
AssertJ	-	Java assertion framework (for verification and debugging)	Open Source	joel-costigliola.github.io/assertj
AutoPilot	Nastel	APM	Freemium	nastel.com
AWS ECS	Amazon Web Services	Elastic container service (with Docker support)	Free Tier Available	aws.amazon.com/ecs
BigMemory Max	Terracotta	In-memory data grid with Ehcache (JCache implementation)	90 Days	terracotta.org/products/bigmemory
Bintray	JFrog	Package hosting and distribution infrastructure	Open Source	bintray.com
Black Duck Platform	Black Duck Software	Security and open-source scanning and management (with container support)	Free Security Scan	blackducksoftware.com
BlueMix	IBM	PaaS with extensive Java support	Free Tier Available	ibm.com/bluemix
BouncyCastle	-	Java and C# cryptography libraries	Open Source	bouncycastle.org
CA Application Monitoring	CA Technologies *	APM with Java agent	30 Days	ca.com
Cask	Cask	Data and application integration platform	Open Source	cask.co/
Catchpoint	Catchpoint	APM with Java agent	Free Trial	catchpoint.com
Censum	jClarity	GC log analysis	7 Days	jclarity.com
CGLIB	Raphael Winterhalter	Byte code generation library	Open Source	github.com/cglib/cglib
CheckStyle	-	Automated check against Java coding standards	Open Source	checkstyle.sourceforge.net

PRODUCT	COMPANY	TYPE	FREE TRIAL	WEBSITE
Chef	Chef Software	Infrastructure automation / configuration management	Open Source	chef.io/chef/
Cloudbees Jenkins Platform	Cloudbees	CI server + verified plugins, build server provisioning, pipeline monitoring, build analytics	2 Weeks	cloudbees.com
Clover	Atlassian	Code coverage analysis tool	30 days	atlassian.com/software/clover/pricing
Codenvy IDE	Codenvy	SaaS IDE with dev workspace isolation	Free Tier Available	codenvy.com
Couchbase	Couchbase	Document-oriented DBMS	Open Source	couchbase.com
Coverity	Synopsys	Security and open-source scanning (static, runtime, fuzz)		
C3P0	-	JDBC connection and statement pooling	Open Source	mchange.com/projects/c3p0
CUBA Platform	Haulmont	Java rapid enterprise app development framework	Free Tier Available	cuba-platform.com
Cucumber	Cucumber	BDD framework with Java version	-	cucumber.io
Dagger	Square	Dependency injector for Android and Java	Open Source	square.github.io/dagger
DataDirect	Progress Software	JDBC connectors (many data sources)	Free Trial	progress.com/jdbc
Derby	Apache Software Foundation	Java SQL database engine	Open Source	db.apache.org/derby
Docker	Docker	Containerization platform	Open Source	docker.com
Dolphin Platform	Canoo	Presentation model framework (multiple views for same MVC group)	Open Source	dolphin-platform.io
DripStat	Chronon Systems	Java+Scala APM with many framework integrations	Free Tier Available	dripstat.com
Drools	Red Hat	Business rules management system	Open Source	drools.org
Dropwizard	-	REST web services framework (opinionated, rapid spinup)	Open Source	dropwizard.io
Dynatrace Application Monitoring	Dynatrace	APM	30 Days	dynatrace.com
Dynatrace SaaS and Managed	Dynatrace (formerly Ruxit)	APM	30 Days	dynatrace.com/platform/offerings/ruxit
EasyMock	-	Unit testing framework (mocks Java objects)	Open Source	easymock.org
Eclipse	Eclipse Foundation	IDE (plugin-happy)	Open Source	eclipse.org
Eclipse Che	Eclipse Foundation	IDE (workspace isolation, cloud hosting)	Open Source	eclipse.org/che
Eclipse Collections	Eclipse Foundation	Java Collections framework	Open Source	eclipse.org/collections
EclipseLink	Eclipse Foundation	JPA+MOXx(JAXB) implementation	Open Source	eclipse.org/eclipselink
EHCache	Terracotta	JCache implementation	Open Source	ehcache.org
ElasticSearch	Elastic	Distributed search and analytics engine	Open Source	elastic.co
ElectricFlow	Electric Cloud	Release automation	Free Version Available	electric-cloud.com
Elide	-	JSON<-JPA web service library	Open Source	elide.io

PRODUCT	COMPANY	TYPE	FREE TRIAL	WEBSITE
Finagle	Twitter	RPC for high-concurrency JVM servers (Java+Scala APIs, uses Futures)		
Finatra	Twitter	Scala HTTP services built on TwitterServer and Finagle	Open Source	twitter.github.io/finatra
FlexJSON	-	JSON serialization	Open Source	flexjson.sourceforge.net
FreeMarker	Apache Software Foundation	Server-side Java web templating (static+dynamic)	Open Source	freemarker.org
FusionReactor	Integral	JVM APM with production debugging and crash protection	Free Trial	fusion-reactor.com
GemFire	Pivotal	Distributed in-memory data grid (using Apache Geode)	Open Source	pivotal.io/big-data/pivotal-gemfire
GlassFish	Oracle	Java application server	Open Source	glassfish.java.net
Go	ThoughtWorks	Continuous delivery server	Open Source	go.cd
Google Web Toolkit (GWT)	Google	Java->Ajax	Open Source	gwtproject.org
Gradle	Gradle	Build automation (Groovy-based scripting of task DAGs)	Open Source	gradle.org
Grails	-	Groovy web framework (like Ruby on Rails)	Open Source	grails.org
GridGain	GridGain Systems	In-memory data grid (Apache Ignite + enterprise management, security, monitoring)	Free Tier Available	gridgain.com
GSON	Google	JSON serialization	Open Source	github.com/google/gson
Guava	Google	Java libraries from Google (collections, caching, concurrency, annotations, I/O, more)	Open Source	github.com/google/guava
Guice	Google	Dependency injection framework	Open Source	github.com/google/guice
H2	-	Java SQL database engine	Open Source	h2database.com
Hazelcast Enterprise Platform	Hazelcast *	Distibuted in-memory data grid (with JCache implementation)	30 Days	hazelcast.com
Heroku Platform	Salesforce *	PaaS	Free Tier Available	heroku.com
Hibernate ORM	-	Java ORM with JPA and native APIs	Open Source	hibernate.org/orm
Hibernate Search	-	Full-text search for objects (indexes domain model with annotations, returns objects from free text queries)	Open Source	hibernate.org/search
Hoplon	-	ClojureScript web framework	Open Source	hoplon.io
HyperForm	HyperGrid (formerly DCHQ)	Container composition platform	Free Tier Available	dchq.io
Hystrix	Netflix	Latency and fault tolerance library	Open Source	github.com/Netflix/Hystrix
IceFaces	IceSoft	JSF framework	Open Source	icesoft.org
Illuminate	jClarity	Java-focused APM with machine learning & autosummarization	14 Days	jclarity.com
Immunio	Immunio	Runtime application self-protection with Java support	30 days	immun.io
Infinispan	Red Hat	Distributed in-memory key/value store (Java embeddable)	Open Source	infinispan.org

PRODUCT	COMPANY	TYPE	FREE TRIAL	WEBSITE
Informatica	Informatica	Data integration and management	-	informatica.com
IntelliJ IDEA	JetBrains *	IDE	Free Tier Available	jetbrains.com/idea
iText	iText Group	PDF manipulation from Java	Open Source	itextpdf.com
ItsNat	Jose Maria Arranz	Web framework (Swing-inspired, Single Page Interface (multiple states=appPages) concept)	Open Source	itsnat.sourceforge.net
Jackson	-	JSON processing	Open Source	wiki.fasterxml.com/JacksonHome
Jahia Platform	Jahia Solutions Group	Enterprise CMS/portal (Jackrabbit compliant)	open-source version available	jahia.com/home.html
Janino	-	Lightweight Java compiler	Open Source	janino-compiler.github.io/janino
JavaFX	Oracle	Java GUI library	Open Source	docs.oracle.com/javase/8/javase-clienttechnologies.htm
JavaServer Faces	Oracle	Java Web Framework	Open Source	oracle.com
JAX-RS	Oracle	REST spec for Java	Open Source	jax-rs-spec.java.net
JBoss Data Grid	Red Hat	In-memory distributed NoSQL data store	Free Tier Available	redhat.com
JBoss EAP	Red Hat	Java EE 7 platform	Open Source	developers.redhat.com/products/eap/overviewith
JD	-	Java decompiler	Open Source	jd.benow.ca
jDBI	-	SQL library for Java	Open Source	jdbi.org
JDeveloper	Oracle	IDE	Freeware	oracle.com
JDOM	-	XMLÊin Java (with DOM and SAX integration)	Open Source	jdom.org
Jelastic	Jelastic	Multi-cloud PaaS (with Java support)	Free Tiers Available	jelastic.com
Jenkins	Cloudbees	CI server	Open Source	jenkins.io
Jersey	Oracle	RESTful web services in Java (JAX-RS with enhancements)	Open Source	jersey.java.net
Jetty	Eclipse Foundation	Servlet engine + http server (with non-http protocols)	Open Source	eclipse.org/jetty
JFreeChart	Object Refinery Limited	Java charting library	Open Source	jfree.org/jfreechart
JGroups	Red Hat	Java multicast messaging library	Open Source	jgroups.org
jHiccup	Azul Systems	Show performance issues caused by JVM (as opposed to app code)	Open Source	azulsystems.com
JMS Adapters for .NET or BizTalk by JNBridge	JNBridge	JMS Integration with .NET or BizTalk	30 Days	jnbridge.com/software/jms-adapter-for-biztalk/overview
JNBridgePro	JNBridge	Java and .NET interoperability	30 Days	jnbridge.com/software/jnbridgepro/overview
Joda	-	Date&time library for Java	Open Source	joda.org/joda-time
jOOQ	Data Geekery	Non-ORM SQL in Java	Open Source	jooq.org
jOOL	Data Geekery	Extension of Java 8 lambda support (tuples, more parameters, sequential and ordered streams)	Open Source	github.com/jOOQ/jOOL

PRODUCT	COMPANY	TYPE	FREE TRIAL	WEBSITE
Joyent	Joyent	Container-native infrastructure with Java images	-	joyent.com
JProfiler	EJ Technologies	Java profiling	Free for Open Source and Nonprofit	ej-technologies.com/products/jprofiler/overview.html
JRebel	ZeroTurnaround *	Class hot-loading (in running JVM)	Free Trial	zeroturnaround.com/software/jrebel
JReport	Jinfonet	Reporting, dashboard, analytics, BI for Java	Free Trial	jinfonet.com
JSF	Oracle	Java spec for server-side component-based UI	Open Source	javaserverfaces.java.net
JSP	Oracle	Server-side Java web templating (static+dynamic)	Open Source	jsp.java.net
JUnit	-	Unit testing framework (mocks Java objects)	Open Source	junit.org
Kubernetes	-	Container orchestration	Open Source	kubernetes.io
Lagom	Lightbend	Reactive microservices framework (Java, Scala)	Open Source	lightbend.com/lagom
LaunchDarkly	Catamorphic	Feature flag platform	30 days	launchdarkly.com
Liferay Digital Experience Platform	Liferay	Enterprise CMS/portal	Open Source Version Available	liferay.com
Lift	-	Scala web framework with ORM, strong view isolation, emphasis on security	Open Source	liftweb.net
Lightbend Reactive Platform	Lightbend	Dev+prod suite for reactive JVM applications (Akka+Play+Lagom+Spark)	Open Source	lightbend.com/platform
Logback	QOS.ch	Java logging framework (Log4j take two)	Open Source	logback.qos.ch
MarkLogic 8	MarkLogic	Multi-model enterprise NoSQLÊdatabase	Free Developer Version	marklogic.com
Mendix Platform	Mendix	Enterprise aPaaS	Free Trial	mendix.com/application-platform-as-a-service/
Mockito	-	Unit testing framework (mocks Java objects)	Open Source	mockito.org
MongoDB	MongoDB	Document-oriented DBMS	Open Source	mongodb.com
MyBatis	-	JDBC persistence framework	Open Source	mybatis.org/mybatis-3
MyEclipse	Genuitec	IDE (Java EE + web)	30 Days	genuitec.com/products/myeclipse
NetBeans	Oracle	IDE	Open Source	netbeans.org
Netty	-	Event-driven, non-blocking JVM framework for protocol clients & servers	Open Source	netty.io
New Relic	New Relic	APM with Java agent	14 Days	newrelic.com
Nexus Repository	Sonatype	Binary/artifact Repository	Open Source	sonatype.org/nexus
NGINX	NGINX	Web server, load balancer, reverse proxy	Open Source	nginx.com
Ninja Framework	-	Full-stack web framework for Java		ninjaframework.org
Nuxeo Platform	Nuxeo	Structured+richcContent management platform	30 days	nuxeo.com

PRODUCT	COMPANY	TYPE	FREE TRIAL	WEBSITE
OmniFaces	-	JSF utility library	Open Source	omnifaces.org
OpenCV	-	Computer vision libraries (with Java interfaces)	Open Source	opencv.org
Oracle Coherence	Oracle	In-memory ditributed data grid	Open Source	oracle.com
Oracle Database 12c	Oracle	Relational DBMS	-	oracle.com/technetwork/database/index.html
OSGi	OSGi Alliance	Dynamic component system spec for Java	-	osgi.org
OutSystems	OutSystems	Rapid application development platform	Free Tier Available	outsystems.com
OverOps	OverOps (formerly Takipi)	JVM agent for production debugging	Free Tier Available	overops.com
Palamida	Palamida	Security and open-source scanning and management	contact for demo	palamida.com
Payara Server	Payara	Java EE application server (enchanced GlassFish)	Open Source	payara.fish/home
Pedestal	-	Clojure Web Framework	Open Source	github.com/pedestal/pedestal
Percona Server	Percona	High-performance drop-in MySQL or MongoDB replacement	Open Source	percona.com
Play	Lightbend	Java + Scala web framework (stateless, async, built on Akka)	Open Source	playframework.com
Plumbr	Plumbr	Memory Leak Detection, GC Analysis, Thread & Query Monitoring	14 days	plumbr.eu
Predix	GE Software	Industrial IoT platform with Java SDK (on Cloud Foundry)	-	ge.com/digital/predix
PrimeFaces	PrimeTek	UI components for JSF	Open Source	primefaces.org
Project Reactor	Pivotal	Non-blocking, async JVM library (based on Reactive Streams spec)	Open Source	projectreactor.io
PubNub	PubNub	Real-timeÊmobile, web, and IoT APIs	free tier available	pubnub.com
Puppet	Puppet Labs	Infrastructure automation / configuration management	Open Source	puppet.com
Push Technology	Push Technology	Real-time messaging (web, mobile, IoT)	contact for more info	pushtechnology.com
Qoppa PDF Studio	Qoppa	PDF manipulation from Java	demos available	qoppa.com
QueryDSL	Mysema	DSL for multiple query targets (JPA, JDO, SQL, Lucene, MongoDB, Java Collections)	Open Source	querydsl.com
Race Catcher	Thinking Software	Dynamic race detection	7 days	thinkingsoftware.com
Redis	Redis Labs	In-memory key-value data structure store (use as database, cache, message broker)	Open Source	redis.io
Rhino	Mozilla	JavaScript implemention in Java (for embedded JS)	Open Source	developer.mozilla.org/en-US/docs/Mozilla/Projects/Rhino
Ribbon	Netflix	RPC library with load balancing	Open Source	github.com/Netflix/ribbon
RichFaces	Red Hat	UI components for JSF	Open Source	richfaces.jboss.org
Ring	-	Clojure Web Framework	Open Source	github.com/ring-clojure/ring

PRODUCT	COMPANY	TYPE	FREE TRIAL	WEBSITE
RxJava	Netflix	Reactive extension for JVM (extends observer pattern)	Open Source	github.com/ReactiveX/RxJava
Salesforce App Cloud	Salesforce	PaaS with app marketplace	free developer instance	developer.salesforce.com
Sauce Labs Automated Testing Platform	Sauce Labs	Browser and mobile test automation (Selenium, Appium) with Java interface	Open Source	saucelabs.com/open-source
SBT (Simple Build Tool)	Lightbend	Build tool for Scala and Java (with build scripting in Scala DSL)		
Scalatra	-	Scala web microframework	Open Source	scalatra.org
Selenide	Codeborne	UI tests in Java (Selenium WebDriver)	Open Source	selenide.org
Selenium	-	Browser automation with Junit and TestNG integration	Open Source	seleniumhq.org
Site24x7	Zoho	Website, server, application performance monitoring	30 days	site24x7.com
Sl4j	QOS.ch	Logging for Java	Open Source	slf4j.org
SmartGWT	Isomorphic Software	Java->Ajax with rapid dev tools, UI components, multi-device	60 Days	smartclient.com
SonarQube	SonarSource	Software quality platform (unit testing, code metrics, architecture and complexity analysis, coding rule checks, more)	Open Source	sonarqube.org
Spark Framework	-	Lightweight Java 8 web app framework	Open Source	sparkjava.com
Split	Split Software	Feature flag platform	Free Trial	split.io
Spock	-	Test and specification framework for Java and Groovy	Open Source	spockframework.org
Spray	Lightbend	REST for Scala/Akka	Open Source	spray.io
Spring Boot	Pivotal	REST web services framework (opinionated, rapid spinup)	Open Source	projects.spring.io/spring-boot
Spring Cloud	Pivotal	Distributed systems framework (declarative, opinionated)	Open Source	cloud.spring.io
Spring Framework	Pivotal	Enterpise Java platform (large family of (convention-over-configuration) services, including dependency injection, MVC, messaging, testing, AOP, data access, distributed computing services, etc.)		projects.spring.io/spring-framework
Spring MVC	Pivotal	Server-side web framework	Open Source	docs.spring.io/spring/docs/current/spring-framework-reference/html/mvc.html
SteelCentral	Riverbed	APM	30-90 days	riverbed.com
Stormpath	Stormpath *	Identity and user management	Free Version Available	stormpath.com
SWT	Eclipse Foundation	Java UI widget toolkit	Open Source	eclipse.org/swt
Sysdig	Draios	Container monitoring	Open Source	sysdig.com
Tasktop Dev	Tasktop	In-IDE ALM tool (commercial version of Eclipse Mylyn)	30 Days	tasktop.com/tasktop-dev
TayzGrid	Alachisoft	In-memory data grid (JCache compliant)	Open Source	tayzgrid.com
Teradata	Teradata	Data warehousing, analytics, lake, SQL on Hadoop and Cassandra, Big Data appliances, R integration, workload management	Free Developer Version	teradata.com

PRODUCT	COMPANY	TYPE	FREE TRIAL	WEBSITE
TestNG	-	Java unit testing framework (JUnit-inspired)	Open Source	testng.org
ThingWorx	PTC	IoT platform with Java SDK	Free Trial	developer.thingworx.com
Thymeleaf	Thymeleaf	Server-side Java web template engine	Open Source	thymeleaf.org
TrueSight Pulse	BMC	Infrastructure monitoring	14 Days	bmc.com/truesightpulse
Twilio	Twilio	Messaging APIs (text, voice, VoIP)	Free Key Available	twilio.com
Upsource	JetBrains *	Code review	Free 10-User Plan	jetbrains.com/upsource
Vaadin	Vaadin	Server-side Java->HTML5	Open Source	vaadin.com
Vert.x	-	Event-driven, non-blocking JVM framework	Open Source	vertx.io
Visual COBOL	Microfocus *	COBOL accessibility from Java (with COBO->Java bytecode compilation)	30 Days	microfocus.com/products/visual-cobol
VisualVM	Oracle	JVM Monitoring	Open Source	visualvm.java.net
vmlens	vmlens	Java race condition catcher	Free Trial	vmlens.com
Waratek	Waratek	Java security (runtime application self-protection (RASP))	30 days	waratek.com
WebLogic	Oracle	Java application server	-	oracle.com/middleware/weblogic/index.html
WebSphere Application Server	IBM	Java application server	-	www-03.ibm.com/software/products/en/appserv-was
WebSphere eXtreme Scale	IBM	In-memory data grid	Free Trial	ibm.com
WildFly	Red Hat	Java application server	Open Source	wildfly.org
WildFly Swarm	Red Hat	Uber JAR builder (with trimmed WildFly app server)	Open Source	wildfly.org/swarm
Wiremock	-	HTTP mocking	Open Source	wiremock.org
WSO2 Application Server	WSO2	Web application server	Open Source	wso2.com/products/application-server/
WSO2 Microservices Framework for Java	WSO2	Microservices framework for Java	Open Source	wso2.com/products/microservices-framework-for-java
XebiaLabs XL	XebiaLabs	Deployment automation + release management	Free Trial	xebialabs.com
XRebel	ZeroTurnaround *	Java web app profiler	14 Days	zeroturnaround.com
Xstream	-	XML serialization	Open Source	x-stream.github.io
YourKit Java Profiler	YourKit	Java CPU & memory profiler	15 Days	yourkit.com
Zing	Azul Systems	JVM with unique pauseless GC	Free Tier Available	azul.com/products/zing
ZK Framework	Zkoss	Enterprise Java web framework	Open Source	zkoss.org
Zulu	Azul Systems	Enterprise-grade OpenJDK build	Open Source	azul.com/products/zulu

GLOSSARY

12-FACTOR APP A set of guidelines for building hosted applications that focus on scalability, automation, and compatibility with cloud platforms.

APPLICATION PROGRAM INTERFACE (API) A set of tools for determining how software components should act within an application.

ARRAY An object that contains a fixed number of variables, the number of which are established when the array is created.

ATTRIBUTE A particular characteristic of an object, which is defined by that object's class.

CLASS A blueprint used to create objects by establishing what attributes are shared between each object in that class.

CLOUD PLATFORM A service offered by a third party that deploys and hosts applications on their hardware.

COMPILER A program that transforms programming language code to machine-readable bytecode so that it can be read and executed by a computer.

CONCURRENCY The ability to run several applications, or several parts of an application, at the same time.

CONSTRUCTOR A subroutine within a class that is called in order to create an object from that class.

CONSTRUCTOR CHAINING The act of calling a constructor to create an object by using another constructor from the same class.

CONTINUOUS INTEGRATION The process of combining and testing changes to an application as often as possible.

DEPENDENCY An instance in a JAR where classes called from packages outside the JAR must be explicitly stated in the code rather than being automatically included.

DESIGN PATTERN A reusable, high-level solution to a common problem in an application or architecture.

EXCEPTION An interruption that occurs when a program is running that alters the way the program is normally run.

EXCEPTION HANDLING A programming practice to ensure that any exceptions do not cause the program to stop running.

HASH FUNCTION A way to create a simplified representation of a large amount of data to make it easily and quickly searchable, such as assigning objects to integers.

IMMUTABLE OBJECT Any object that cannot be modified once it has been created.

JAVA An object-oriented programming language that can be deployed on a variety of platforms, developed by Sun Microsystems and now under the stewardship of Oracle.

JAVA ARCHIVE (JAR) A file used to aggregate several Java classes and resources into one file in order to easily distribute those classes for use on the Java platform.

JAVA DEVELOPMENT KIT (JDK) A free set of tools, including a compiler, provided by Oracle, the owners of Java.

JAVA VIRTUAL MACHINE (JVM) Abstracted software that allows a computer to run a Java program.

LAMBDA EXPRESSION A new feature in Java 8 which allows functions, called anonymous functions, to be written without belonging to a class.

LIBRARY A collection of commonly used pieces of code that can be used in any application that uses that programming language.

LOG A file with information on everything that has happened while an application has been running.

METHOD A named piece of code that can be called at any time by using its name.

MICROSERVICES ARCHITECTURE An architecture for an application that is built with several modular pieces, which are deployed separately and communicate with each other, rather than deploying one single piece of software.

OBJECTS An instance of a class, which has different attribute values than other objects in its class.

POLYMORPHISM A condition in which objects can take on several forms in an application. All objects in Java are considered polymorphic since all objects are all implementations of classes.

REPOSITORY A data structure where directories, files, and metadata can be stored and managed.

SPRING FRAMEWORK An open-source collection of tools for building web applications in Java.

STREAM A sequence of data that is read from a source and then written to a new destination.

SYNCHRONIZATION A condition in which multiple threads that share crucial resources run at the same time.

TOMCAT An open-source web server technology for Java applications.