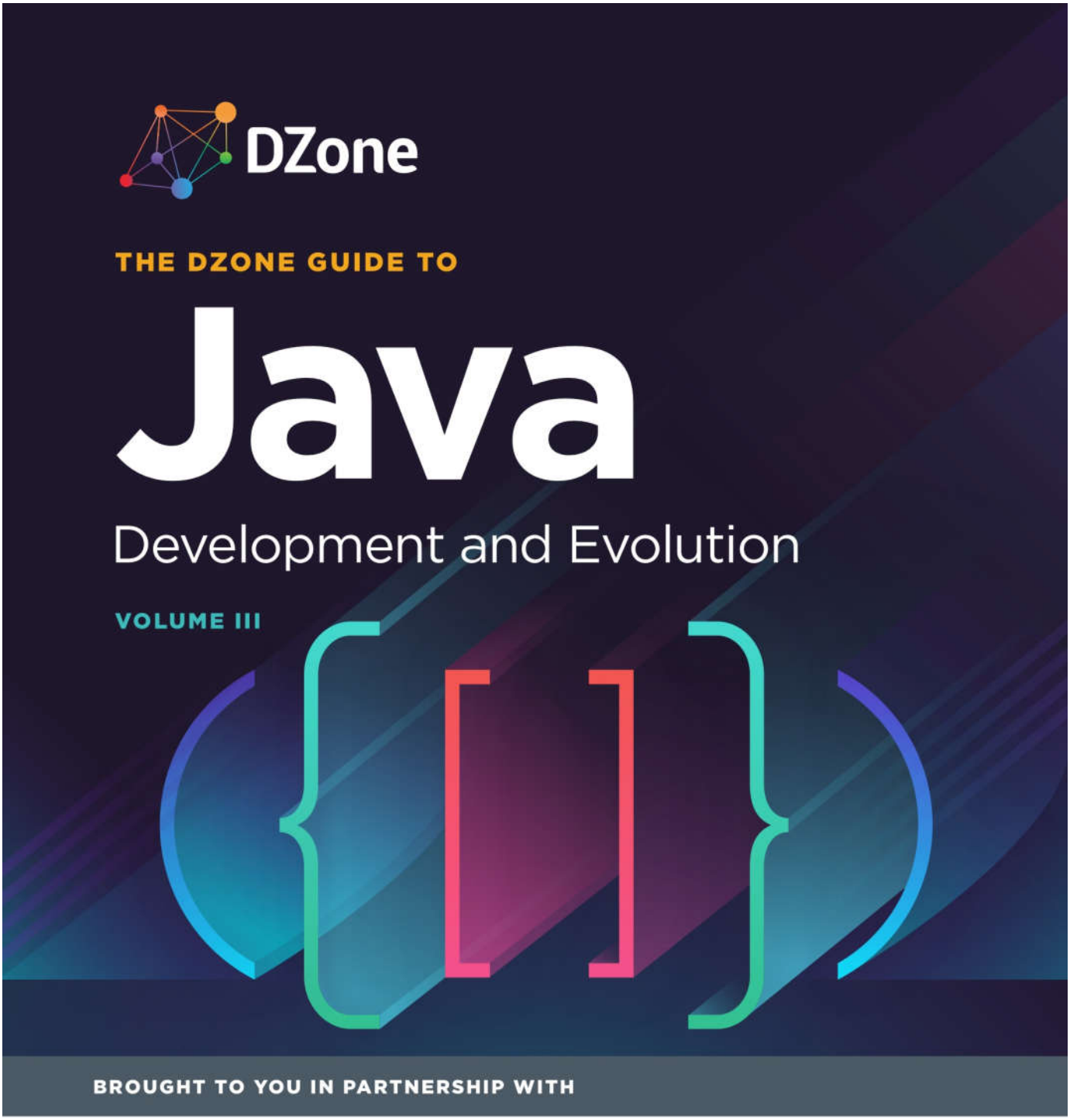

DZone
THE DZONE GUIDE TO
Java
Development and Evolution
VOLUME III
BROUGHT TO YOU IN PARTNERSHIP WITH


CUBA
.platform


fusion
reactor™


ITEXT


Jelastic


Lightbend

DEAR READER,

Every year we see new languages appear that claim to unseat Java from its throne. However, 22 years since it's conception, Java remains the steadiest programming language out there. While other languages fit certain niches, Java developers are among the most employable software professionals out there. However, writing Java doesn't mean that you're dealing with legacy systems or old-fashioned code styles. Java has continued to move along with the times.

Java 8 introduced lambdas, finally giving the power of functional programming to Java developers, resulting in more concise code. Meanwhile, the addition of the Streams API allowed developers to handle larger volumes of data. This summer we'll (hopefully!) see the introduction of new features with Java 9, but the primary focus will be on the new module system: Project Jigsaw. These modules will be a game changer for teams that need to refactor their applications to be truly scalable. JShell will give us an interactive Java REPL meaning that the exploration of language features is more lightweight, and let's face it, fun! There are also improvements to the Streams API and support for HTTP/2 and WebSocket. Other language fans will say they already have all of these capabilities, but on top of the language's rock solid foundations, it reinstates Java as a compelling choice for future applications. This guide provides a tour of Java 9 features for legacy developers, guaranteed to make you feel more comfortable with the upcoming changes.

This guide will also take you through some of the main talking points of modern Java, from designing appropriately for microservices, using Java 8 to address code smells, to concurrency with Java Futures and Kotlin Coroutines. Since Google I/O this year, Kotlin has received a boom in popularity. Whether it becomes the language of choice for Android developers remains to be seen, but it's interoperability with Java ensures that no one will get left behind.

There are some skills, like how to handle memory leaks and debugging that will always be required in the Java developer's toolbox, and these are also discussed in this guide. While it's easy to ignore these fundamentals for the sake of newer features, I urge all readers to take note of these pages.

I would say that Java 9 ushers in the third golden age for Java developers. It all started with the Age of Applets, a game changer in its day. We then had the Age of JavaEE, which brought about Hibernate and Spring, frameworks that became so rooted in Java development that they remain a staple of many technology stacks. With the introduction of real modularity and building on Streams, we have entered a new era, where Java developers no longer need to feel left behind by the advances of modern programming languages. For those who feel like Java is too verbose, Kotlin has emerged as a great alternative on the JVM and Android devices. If you're a new Java developer, there's never been a better time to get on board. However, if you're like me and have grown up with Java throughout your career, it's very difficult not to be excited.

Welcome to the golden age of Java!

BY JAMES SUGRUE
DZONE ZONE LEADER, AND CHIEF ARCHITECT, OVER-C

TABLE OF CONTENTS

PRODUCTION
Chris Smith — DIRECTOR OF PRODUCTION
Andre Powell — SR. PRODUCTION COORDINATOR
G. Ryan Spain — PRODUCTION PUBLICATIONS EDITOR
Ashley Slate — DESIGN DIRECTOR

BUSINESS
Rick Ross — CEO
Matt Schmidt — PRESIDENT AND CTO
Jesse Davis — EVP
Gordon Cervenka — COO

MARKETING
Kellet Atkinson — DIRECTOR OF MARKETING
Lauren Curatola — MARKETING SPECIALIST
Kristen Pagàn — MARKETING SPECIALIST
Natalie Iannello — MARKETING SPECIALIST
Miranda Casey — MARKETING SPECIALIST
Julian Morris — MARKETING SPECIALIST

SALES
Matt O'Brian — DIRECTOR OF BUSINESS DEV.
Alex Crafts — DIRECTOR OF MAJOR ACCOUNTS
Jim Howard — SR ACCOUNT EXECUTIVE
Jim Dyer — ACCOUNT EXECUTIVE
Andrew Barker — ACCOUNT EXECUTIVE
Brian Anderson — ACCOUNT EXECUTIVE
Chris Brumfield — SALES MANAGER
Ana Jones — ACCOUNT MANAGER
Tom Martin — ACCOUNT MANAGER

EDITORIAL
Caitlin Candelmo — DIRECTOR OF CONTENT AND COMMUNITY
Matt Werner — PUBLICATIONS COORDINATOR
Michael Tharrington — CONTENT AND COMMUNITY MANAGER
Mike Gates — SR. CONTENT COORDINATOR
Sarah Davis — CONTENT COORDINATOR
Tom Smith — RESEARCH ANALYST
Jordan Baker — CONTENT COORDINATOR
Anne Marie Glen — CONTENT COORDINATOR

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.

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 guide a great success.

Executive Summary

BY MATT WERNER
PUBLICATIONS COORDINATOR, DZONE

If you believe the number of articles online, you might be inclined to believe that Java is dying, but we all know better. While the release of Java 9 and Java EE 8 have been delayed, the excitement and passion of the Java community remains strong, proven by groups like the Java EE Guardians, who formed to push Oracle to commit to improvements for Java EE. This energy is not just a result of complacent developers failing to leave behind a stagnant language. New JVM technologies like the Kotlin language are making incredible waves in the industry, and Java itself has been encouraging the combination of functional and object-oriented programming with features like lambdas. While the language may be 22 years old, it has not rested on its laurels, and with Project Jigsaw on the horizon, Java and its stewards continue to prove that they are constantly looking to the future.

DZone surveyed 652 tech professionals to learn about how our audience has embraced Java and its related technologies, and collected a series of articles from some of Java's strongest champions to educate our readers about how to use them.

JAVA IS BECOMING MORE FUNCTIONAL

DATA 77% of survey respondents are using lambda functions in 2017, compared to 46% in 2016, while 67% of members feel that they do more functional development than they used to, compared to 52% in 2016.

IMPLICATIONS As developers continue to adopt Java 8 features, they are becoming more comfortable with creating more functional code (51% are either comfortable or very comfortable). Of those who are using functional programming, 80% say that coding in Java is more fun than it was before.

RECOMMENDATIONS One comment that has plagued Java is its perception of being a verbose language, but Java 8's features are starting to catch on and help developers enjoy Java programming, whether they have been long-time fans, are new to the language, or have to use it for their work. If developing Java applications has lost its luster, consider experimenting with Java 8 features like lambdas and the Streams API. You can see a fun comparison of how these features can decrease development time in our infographic on page 23. In addition to making development more enjoyable, you'll be able to easily ensure that the code you develop now will be compatible with new Java 9 features thanks to multi-release JAR files, which you can read about in Wayne Citrin's article on page 6.

KOTLIN IS MAKING A SPLASH

DATA 61% of DZone members use at least one JVM-based language, and 27% use one or more of these in production. Kotlin adoption doubled year-over-year, from 7% to 16%, while the top two languages, Scala (38%) and Groovy (43%), decreased by 3% and 2% over last year, respectively.

IMPLICATIONS JetBrains' relatively new JVM language is slowly turning into a force to be reckoned with. Version 1.0 was released in 2016, and considering its massive growth since last year and interest within DZone's audience, its adoption numbers will likely continue to grow.

RECOMMENDATIONS First, you certainly don't need to use a non-Java JVM language, especially since Java 8 and Java 9 features will provide some overlap with languages like Scala, which supports functional programming in Java. However, given Google's support of Kotlin as an official Android development language and the importance of mobile development in general, it would be a worthwhile investment to learn more about the language. For an excellent article on Kotlin Coroutines, you can refer to Nicolas Fränkel's article on page 12.

JAVA EE AND SPRING'S TUG-OF-WAR

DATA While Spring 4.x adoption stagnated over the past year (47% in 2017 compared to 49% in 2016), Java EE 7 saw a 10% increase in usage, from 41% to 51%, mirroring a Spring 3.x decrease from 37% to 27% in the same amount of time.

IMPLICATIONS Adoption of platforms can normally take a long time, but this year saw a stark increase in the use of Java EE 7 that mirrored a 10% drop in usage of Spring 3.x. Though Spring 4.x was also released 4 years ago, the sudden jump of Java EE7 adoption and abandoning of Spring 3.x suggests that Spring 4.x is missing something for older Spring fans that Java EE 7 may offer.

RECOMMENDATIONS Refactoring and upgrading applications can take an enormous amount of time and money, so if an update is required, you need to carefully weigh your options. Based on how DZone members have responded, Java EE 7 seems to be a sensible choice over newer versions of Spring. However, Spring 5 was previewed in February 2017, and it will include a functional web framework and will be built on Reactive Core. It's scheduled to be released in June. Java EE 8 is scheduled to be released later this year, but there has been confusion around Oracle's commitment to the project, leading to the formation of MicroProfiles and the Java EE Guardians. While we don't have a full idea of how these upgraded platforms will perform, it's a good idea to wait until they are released, and weigh your options unless it's critical that you update now.

Key Research Findings

BY G. RYAN SPAIN
PRODUCTION COORDINATOR, DZONE

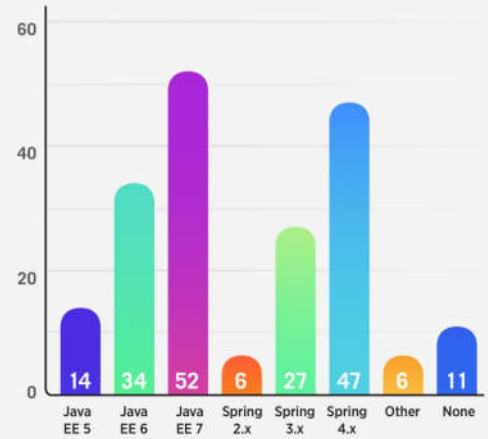
652 software professionals completed DZone's 2017 Java survey. Respondent demographics are as follows:

- 38% of respondents identify as developers or engineers; 23% identify as developer team leads; and 19% identify as architects.
- The average respondent has 14.4 years of experience as an IT professional. 66% of respondents have 10 years of experience or more; 29% have 20 years or more.
- 43% of respondents work at companies headquartered in Europe; 30% work at companies with HQs in North America.
- 20% of respondents work at organizations with more than 10,000 employees; 22% at organizations between 1,000 and 10,000 employees; and 23% at organizations between 100 and 999 employees.
- 85% develop web applications or services; 53% develop enterprise business apps; and 26% develop native mobile applications.

THE NEW NORMAL

It should be no surprise that Java 8 continues to gain popularity in both new and refactored apps, especially considering the delays in Java 9's launch. In this year's Java survey, 89% of respondents say they use Java 8 in new apps (up 8% from last year), while 49% say they use Java 8 in existing applications (up 15% from last year). So as Java 8 continues to be cemented as the go-to Java version for new applications, it is also increasingly turned to for refactoring. Features new to Java 8 are getting more use as well. 77% of respondents to this year's survey said they use lambdas in new code, 75% say they use streams in new code, and 48% say they use optionals in new code. This is a considerable increase from Java 8 feature usage last year, where 46% of respondents said they used lambdas, 43% said they used streams, and 29% said they used optionals in new code.

Which of the following 'enterprise' Java platforms do you or your organization use?



What versions of Java are being used at your organization?

	FOR NEW APPS	FOR EXISTING APPS	NOT USING
Java 5 and below	0%	11%	89%
Java 6	2%	35%	65%
Java 7	17%	63%	32%
Java 8	89%	49%	9%

THE 'FUN' IN 'FUNCTIONAL'

As the use of these Java 8 features increases, so do the number of developers who feel they write more functional code. Last year, 52% of survey respondents said they write more functional code after adoption of Java 8. This year, that number grew to 67%. Mixing "old style" and "new style" code in the same application increased from 55% in 2016 to 62% in 2017. A little over half of the respondents feel either very comfortable (11%) or comfortable (40%) with mixing functional and object-oriented paradigms in their code; 21% have no opinion. Furthermore, of the respondents who said they are now programming more functionally using Java 8's features, 80% said these features have made programming in Java more fun.

THE JVM BEYOND JAVA

61% of survey respondents said they use at least one non-Java, JVM-based language, and 27% use one or more of these languages in production. Groovy and Scala usage had no significant change over the last year—38% of survey respondents said they use Scala, compared to 41% in 2016, and 43% of respondents said they use Groovy, compared to 45% in 2016. Kotlin, on the other hand, more than doubled its adoption rate since last year, with 16% of respondents saying they use the language, as opposed to 7% last year.

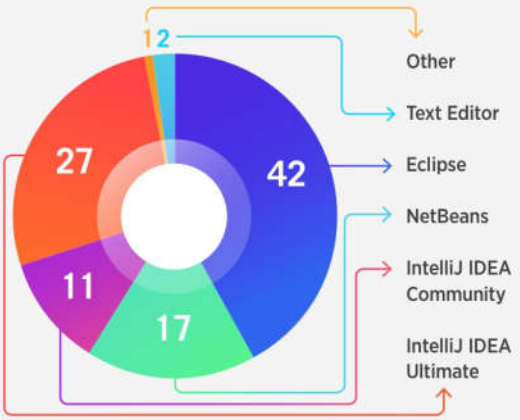
IDE TIME

The Eclipse IDE remains the most popular place for Java developers to primarily write their code, though it saw a 7% decrease from last year's survey results (50% in 2016 vs. 43% in 2017). IntelliJ IDEA Ultimate has a considerable user base as well, with 27% of respondents saying they primarily write their Java code in that IDE. NetBeans usage increased 7% from 2016 (10% in 2016 vs. 17% in 2017), overtaking IntelliJ IDEA Community Edition as the third most popular IDE.

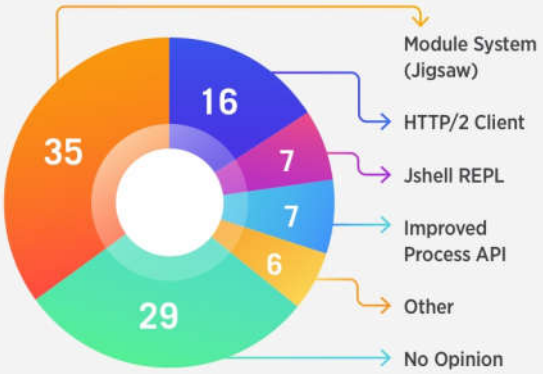
JAVA EE VS. SPRING... AGAIN

Last year we saw a significant increase in Spring 4.x usage from 2015—38% to 49%. This year showed stagnation in Spring 4.x adoption, as 47% of respondents said they or their organization use Spring 4.x. Spring 3.x, on the other hand, had a fairly dramatic drop, from 37% in 2016 to 27% in 2017. This decrease in Spring 3 usage was mirrored by a jump in Java EE 7 usage; 41% of respondents in 2016 said they or their organization used Java EE 7, compared to 52% this year. This makes Java EE 7 the most popular enterprise Java platform again, as it was in 2015, but like in 2015, the gap is fairly narrow between Java EE 7 and Spring 4. 30% of respondents use both Java EE 7 and Spring 4, and only 24% use neither of the two platforms.

Where do you primarily write Java code?



In your opinion, what is the most important new feature in Java 9?



Yes, You Can: Java 9 for the Legacy Developer

BY **WAYNE CITRIN**
CTO, JNBRIDGE

QUICK VIEW

01 Java 9 offers several features that allow developers maintaining older Java apps to use new features while maintaining backwards compatibility.

02 The new multi-release JAR files in Java 9 allow developers to write code using new APIs and language features while making new code invisible when running on older versions of Java.

03 Java 9's new module capability allows for the gradual introduction of modules while allowing modularized and not-yet-modularized components to be mixed and matched.

Whenever a new version of Java is released, there's a lot of excited discussion about new language constructs and APIs, features, and benefits. But the excitement quickly wanes for legacy developers when they remember that they must maintain and enhance existing applications, rather than create new ones. Most applications must be backwards-compatible with earlier versions of Java, which do not support new features presented by shiny new Java releases. So, legacy developers resign themselves to sit on the sidelines and watch.

Fortunately, Java 9's designers have kept this in mind, and have worked out ways to make Java 9's new features accessible for developers who have to worry about their applications supporting older versions of Java. Here we will discuss how new features in Java 9 — multi-release JAR files, Project Jigsaw (the new module system), and the modular JDK and jlink — make Java 9 usable and relevant to legacy Java developers.

MULTI-RELEASE JAR FILES

Until recently, there hasn't been a good way to use the latest Java features while still allowing the application to run on earlier versions of Java that don't support that application. Java 9 finally provides a way to do this for both new APIs *and* for new Java language constructs: multi-release JAR files.

Multi-release JAR files look just like old-fashioned JAR files, with one crucial addition: there's a new nook in the JAR file where you can put classes that use the latest Java 9 features. If you're running Java 9, the JVM recognizes this nook, uses the classes in that nook, and ignores any classes of the same name in the regular part of the JAR file. If you're running Java 8 or earlier, however, the JVM doesn't know about this special nook and will ignore it, and only run the classes in the regular part of the JAR file. In the future, when Java 10 comes out, there'll be another nook specifically for classes using new Java 10 features, and so forth.

The Java 9 JDK will contain a version of the jar.exe tool that supports creating multi-release JAR files. Other non-JDK tools will also provide support.

PROJECT JIGSAW

The Java 9 module system (also known as Project Jigsaw), is undoubtedly the biggest change to Java 9. One goal of modularization is to strengthen Java's encapsulation mechanism so that the developer can specify which APIs are exposed to other components and count on the JVM to enforce the encapsulation. Modularization's encapsulation is stronger than that provided by the public/protected/private access modifiers of classes and class members. The second goal of modularization is to specify which modules are required by which other modules, and to ensure that all necessary modules are present before the application executes. In this sense, modules are stronger than the traditional classpath mechanism, since classpaths are not checked ahead of time, and errors due to missing classes only occur when

the classes are actually needed, which means that an incorrect classpath might be discovered only after an application has been run for a long time, or after it has been run many times.

Java 9 offers both a *classpath* and a *module path*. The classpath works just as before, and you can keep using it. JAR files in the module path are interpreted as modules — they expose APIs and have dependencies that can be checked at compile time. If you want to go the extra mile, you can *modularize* a JAR file by adding information specifying which APIs are exposed and which other modules are required. However, even if you don't, if your JAR file is in the module path, it's considered an *automatic module*, although it lacks some information that can be used to check whether anything is missing. Also, all the JAR files in the classpath are considered part of the *unnamed module*, which means they become part of the module system, too.

This means that it really doesn't matter whether your JAR files are modularized or whether they're still old school. It may even be the case that you can't modularize a JAR file because it doesn't belong to you. All modules play in the module system, whether they're up to date or not, and whether they're in the classpath or the module path. If you modularize your JAR files and put them in the module path, you'll get additional benefits and avoid potential errors. But even if you don't, you get many of the benefits of the module system. It all might sound complicated, but it just works.

With Java 9, it's now possible to create a self-contained environment with your application and anything it needs to run.

HOW TO SUPPLY YOUR OWN JAVA ENVIRONMENT WITH MODULAR JDK AND JLINK

One problem with legacy Java applications is that the end user might not be using the right Java environment, and one way to guarantee that the Java application will run is to supply the Java environment with the application.

Java allows the creation of a *private* or *redistributable JRE*, which may be distributed with the program. The JDK/JRE installation comes with instructions on how to create a private JRE. Typically, you take the JRE file hierarchy that's installed with the JDK, keep the required files, and retain those optional files whose functionality your application will need. The process is a bit of a hassle: You need to maintain the installation file hierarchy, you have to be careful that you don't leave out any files and directories that you might need, and, while it does no harm to do so, you don't want to leave in anything that you don't need, since it will take up unnecessary space. It's easy to make a mistake. So why not let the JDK do the job for you? With Java 9, it's now possible to create a self-contained environment with your application and anything it needs to run. No need to worry that the wrong Java environment is on the user's machine, and no need to worry that you've created the private JRE incorrectly.

The key to creating these self-contained runtime images is the module system. Not only can you modularize your own code (or not), but the Java 9 JDK is itself now modularized. The Java class library is now a collection of modules, as are the tools of the JDK itself. The module system requires you to specify the base class modules that your code requires, and that in turn will specify the parts of the JDK that are needed. To put it all together, we use a new Java 9 tool called jlink. When you run jlink, you'll get a file hierarchy with exactly what you'll need to run your application — no more and no less. It'll be much smaller than the standard JRE.

With jlink, it becomes easy to package up your application and everything it needs to run, without worrying about getting it wrong, and only packaging that part of the runtime that's necessary to run your application. This way, your legacy Java application has an environment on which it's guaranteed to run.

With this newfound knowledge, it's clear that legacy developers need not sit on the sidelines and watch as everyone else gets to play with the new Java 9 features. Using these approaches, anyone can take advantage of all Java 9 has to offer, without breaking compatibility with earlier versions of Java.

Wayne Citrin is CTO at JNBridge, the leading provider of interoperability tools to connect Java and .NET frameworks. The architect of award-winning bridging technology JNBridgePro and JMS Adapters for .NET and BizTalk, Citrin has been solving Java and .NET interoperability issues since .NET's beta days. Citrin has served as a leading researcher in programming languages and compilers, and was on the Computer Engineering faculty at the University of Colorado, Boulder. Visit Citrin's blog at jnbridge.com/jnblog.

Design Patterns in the Age of Microservices and Frameworks

QUICK VIEW

01 We should strive to keep our code as simple as possible.

02 GoF patterns are good for framework and library designers.

03 DDD patterns are useful for business code, but we should be aware of the associated risks.

04 PoEAA pattern usage seems settled, but it's not necessarily a good thing.

BY **GRZEGORZ ZIEMOŃSKI**
JAVA ENGINEER, **ZOOPLUS AG**

We are in the age of web applications, frameworks, and microservices architectures. We can spin a module of our application within a day from zero to production. Any technical concern other than coding up our business rules has already been taken care of for us. We simply write the business rules, put them in a framework, and it works. What does that mean for design patterns? Do we still need them or we can safely put them in a museum?

BACK TO THE ROOTS

To answer the questions posed in the introduction, we will need to go back in time a little and answer other questions first. Why did we need the patterns in the first place? What problems did they solve back then? Do we still face these problems today?

Each of the patterns has, of course, its own separate motivation and reason for existence, yet, in general, the goal is pretty much the same. We want our code to be as simple as possible, while at the same time, meeting all the users' current needs and ensuring future maintainability.

If we are to judge the usefulness of design patterns, we should look towards that general goal. Not flexibility, configurability, or any other fancy characteristic. These either lie within our users' current needs or they don't. These are either necessary for future maintainability or can be added later on in the project and therefore should be kept out for the sake of simplicity.

GANG OF FOUR PATTERNS

The GoF patterns have garnered a bad reputation in the industry. I suppose that's because of the typical developer learning process – we learn something new, overuse it, notice problems when it's too late, and only then use the thing with caution. The same can be said about design patterns. I remember my own feeling of enlightenment when I read the book for the first time. I saw the patterns EVERYWHERE. I was sure I could build entire applications using them almost exclusively.

All that being said, if we are to stay objective, we should forget this bad reputation and any bad memories associated with it. The fact that either we, or someone else, overused the Gang of Four's design patterns in the past has nothing to do with their usefulness. Taking this into account would be like demanding a ban of rope production because some people used ropes to hang themselves. Instead, we'll look towards our general goal from the previous section.

When it comes to coding up business rules and configuring frameworks, especially in a microservices (or even serverless) architecture, most of the GoF patterns are not very useful. Actually, it would be faster to enumerate those that might be.

The strategy pattern is a good fit when there is a group of business algorithms to choose from. When the same conditionals start popping up in different places in the code, using an interface and injecting strategies provides a simpler and more maintainable solution. It's not that common, but it happens.

The façade pattern can be used when implementing an anti-corruption layer, to shield our brand new microservice from the influence of legacy systems. Otherwise, the complexity of the old system might start spilling into the new one, making things harder to understand and maintain.

The adapter pattern can help us decouple our business code from the frameworks and libraries that our application builds upon. One use case for this type of usage of the pattern would be to deal with a library that has a particularly annoying

API. In such a case, we could use the pattern to simplify our business code. In another approach, the pattern can be used to build an application-wide architectural style, commonly known as Hexagonal Architecture or Ports and Adapters.

The decorator pattern allows us to add our own bits to the code provided by those frameworks and libraries. These usually include simple operations like logging or gathering some kind of metrics.

To be honest, I can't think of any other GoF pattern that I have used recently, and I don't use the ones above too often either. They just somehow don't fit with the nature of the business code. Now, does that mean that the GoF patterns are almost entirely useless?

Rubies are red,

Some threads are green,

But only Java has AbstractSingletonProxyFactoryBean.

The poem above might give you a little hint. My answer is obviously negative. The frameworks and libraries, that make our work so pleasant and easy, have to use the GoF patterns themselves to remain flexible and configurable. For a framework, this is an actual "current" user need, not some kind of a developer whim.

Often these patterns are not even under the hood – we as programmers take advantage of them. The most notable examples would be the usage of Proxy patterns by Hibernate and developers implementing template methods to configure Spring.

DOMAIN-DRIVEN DESIGN PATTERNS

The GoF patterns were more suitable for technical problems, like the ones framework designers face. But if a pattern is domain-driven, and I really mean **domain-driven** here, it has to be useful in writing business code, right? Well, the answer is not so simple.

As written in the original Domain-Driven Design book, and various other sources, DDD techniques are not suitable for every single project. The same applies to DDD patterns, both strategic and tactical ones. Let's take at some of them from the perspective of our general goal.

Entities are a great pattern to represent domain concepts that are defined by their identity, rather than their attributes. If they come from a domain model built upon a ubiquitous language, they greatly simplify the understanding and, therefore, the maintainability of the system. At the same time, if there is no clearly defined domain model, it's worth exploring if the same work can be done without having entities at all. I have recently seen such a case in practice. By replacing a bunch of entities and their associated **repositories** with simple SQL and JDBC, we built the exact same thing with much less code and hassle.

Value Objects are a great example of a pattern that can both simplify and complicate your code. Whenever your value field requires certain validation logic or special manipulating operations, putting them together in the same place can easily save you a headache. At the same time, if a value field has no such logic or operations, we're just complicating the system by adding extra wrapper classes, each in an extra source file (at least in Java). Does it improve maintainability or user experience? Not really.

Last in this section but surely not least, let's touch on **aggregates**. When an aggregate scope is reasonably small, it provides an easy mechanism for maintaining consistency and a central place for associated business rules. On the other hand, when an aggregate gets too big, it can literally slow down the entire system, reduce code maintainability, and start turning into a "god class." Therefore, aggregates should be used with caution, especially if they are not used as a part of a full blown DDD process.

PATTERNS OF ENTERPRISE APPLICATION ARCHITECTURE

I expect that some readers might not be familiar with Martin Fowler's great book, but, at the same time, I'd expect the majority of you to have used the patterns described in it. Maybe my picture of the entire Java world is a bit limited, but from what I've seen so far, it seems that almost every Java web application is built upon three patterns described in the book: **Domain Model**, **Service Layer**, and **Repository**.

I don't want to say here that it's necessarily a bad thing. In the end, most web applications are conceptually similar and it's the business rules that differentiate them. What I'd merely like to achieve here is to challenge the status quo a bit.

The same way I described my SQL/JDBC solution as an alternative to DDD patterns, this same solution could be considered as an alternative to the triplet above. I invite you to do the same thing in your projects. Read the book or the pattern descriptions online and see if you could simplify your code by abandoning or switching a pattern or two. Maybe an **active record** or a **transaction script** is just what your project needs right now?

CONCLUSION

Currently, the development world is full of patterns. From tricks to avoid inheritance to big architectural decisions – we have a pattern for everything. The keys to choose the right ones are code simplicity, users' needs, and future maintainability. These keys drive us pretty far from the GoF patterns, at least as long as we're not writing a framework or a library. They make us look cautiously on the DDD patterns, unless we're following a full-blown DDD process. Last, but not least, they make us continuously challenge the architectural status quo, so that we pick the right tool for the job at hand.

Grzegorz Ziemoński is a software craftsman specializing in writing clean, readable code. He shares his skills and ideas both as a blogger and as a Java Zone Leader at DZone. He currently works for Zooplus, but keeps dreaming about his own consultancy. Personally, he's a big fan of self-development and a proud owner of two lovely cats.

The Evolution of Scalable Microservices

From building microliths to designing reactive microsystems

Today's enterprise applications are deployed to everything from mobile devices to cloud-based clusters running thousands of multi-core processors. Users have come to expect millisecond response times and close to 100% uptime. And "user" means both humans and machines. Traditional architectures, tools and products simply won't cut it anymore. To paraphrase Henry Ford's classic quote: We can't make the horse any faster, we need cars for where we are going.

While many organizations move away from the monolith and adopt a microservices-based architecture, they mostly do little more than creating microlith instances communicating synchronously with each other. The problem with a single instance is that it cannot be scalable or available. A single monolithic thing, whatever it might be (a human or a software process), can't be scaled out, and can't stay available if it crashes.

But it is also true that as soon as we exit the boundary of the single service instance we enter a wild ocean of non-determinism—the world of distributed microservice architectures.

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

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

- Verizon
- Samsung
- UniCredit Group
- Walmart
- Hootsuite
- Zalando

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

Concurrency: Java Futures and Kotlin Coroutines

QUICK VIEW

01 Developing concurrent code has seen a lot of changes since its inception, from synchronized blocking code to futures.

02 In Java, using both futures and the streaming API creates a lot of clutter, especially if the underlying code uses checked exceptions.

03 In Kotlin, a new experimental feature called coroutine brings a way to write sequential-looking but concurrent-running code.

BY **NICOLAS FRÄNKEL**
SOFTWARE ARCHITECT/DEVELOPER, **SAP**

A long time ago, one had to manually start new threads to run code concurrently in Java. Not only was this hard to write, it also was easy to introduce bugs that were hard to find. Testing, reading, and maintaining such code was no walk in the park, either. Since that time, and with a little incentive coming from multi-core machines, the Java API has evolved to make developing concurrent code easier. Meanwhile, alternative JVM languages also have their opinion about helping developers write such code. In this post, I'll compare how it's implemented in Java and Kotlin.

To keep the article focused, I deliberately left out performance to write about code readability.

ABOUT THE USE CASE

The use case is not very original. We need to call different web services. The naïve solution would be to call them sequentially, one after the other, and collect the result of each of them. In that case, the overall call time would be the sum of the call time of each service. An easy improvement is to call them in parallel and wait for the last one to finish. Thus, performance improves from linear to constant — or for the more mathematically inclined, from *o(n)* to *o(1)*.

To simulate the calling of a web service with a delay, let's use the following code (in Kotlin, because this is so much less verbose):

```
class DummyService(private val name: String) {
    private val random = SecureRandom()
    val content: ContentDuration
        get() {
            val duration = random.nextInt(5000)
            Thread.sleep(duration.toLong())
            return ContentDuration(name, duration)
        }
}
data class ContentDuration(val content: String, val duration:
Int)
```

THE JAVA FUTURE API

Java offers a whole class hierarchy to handle concurrent calls. It's based on the following classes:

Callable: A `Callable` is a "task that returns a result." From another view point, it's similar to a function that takes no parameter and returns this result.

Future: A `Future` is "the result of an asynchronous computation." Also, "the result can only be retrieved using method `get` when the computation has completed, blocking if necessary until it is ready." In other words, it represents a wrapper around a value, where this value is the outcome of a calculation.

Executor Service: An `ExecutorService` "provides methods to manage termination and methods that can produce a `Future` for tracking progress of one or more asynchronous tasks." It is the entry point into concurrent handling code in Java. Implementations of this interface, as well as more specialized ones, can be obtained through static methods in the `Executors` class.

This is summarized in the diagram here.

Calling our services using the concurrent package is a two-step process.

CREATING A COLLECTION OF CALLABLES

First, there needs to be a collection of `Callable` to pass to the executor service. This is how it might go:

1. Form a stream of service names.
2. For each service name, create a new dummy service initialized with the string.
3. For every service, return the service's `getContent()` method reference as a Callable. This works because the method signature matches `Callable.call()` and `Callable` is a functional interface.

This is the preparation phase. It translates into the following code:

```
List<Callable<ContentDuration>> callables = Stream.
of("Service A", "Service B", "Service C")
      .map(DummyService::new)
      .map(service -> (Callable<ContentDuration>)
service::getContent)
      .collect(Collectors.toList());
```

PROCESSING THE CALLABLES

Once the list has been prepared, it's time for the `ExecutorService` to process it, AKA the "real work."

1. Create a new executor service — any will do.
2. Pass the list of `Callable` to the executor service. and stream the resulting list of `Future`
3. For every future, either return the result or handle the exception.

The following snippet is a possible implementation:

```
ExecutorService executor = Executors.newWorkStealingPool();
List<ContentDuration> results = executor.
invokeAll(callables).stream()
    .map(future -> {
        try { return future.get(); }
        catch (InterruptedException | ExecutionException
e) { throw new RuntimeException(e); }
     }).collect(Collectors.toList());
```

THE FUTURE API, BUT IN KOTLIN

Let's face it: While Java makes it possible to write concurrent code, reading and maintaining it is not that easy, mainly due to:

- Going back and forth between collections and streams.
- Handling checked exceptions in lambdas.
- Casting explicitly.

```
var callables: List<Callable<ContentDuration>> =
arrayOf("Service A", "Service B", "Service C")
    .map { DummyService(it) }
    .map { Callable<ContentDuration> { it.content } }
val executor = Executors.newWorkStealingPool()
val results = executor.invokeAll(callables).map { it.get() }
```

KOTLIN COROUTINES

With version 1.1 of Kotlin comes a new experimental feature called coroutines. From the Kotlin documentation:

"Basically, coroutines are computations that can be suspended without blocking a thread. Blocking threads is often expensive, especially under high load [...]. Coroutine suspension is almost free, on the other hand. No context switch or any other involvement of the OS is required."

The leading design principle behind coroutines is that they must feel like sequential code but run like concurrent code. They are based on the diagram here.

Nothing beats the code itself, though. Let's implement the same as above, but with coroutines in Kotlin instead of Java futures.

As a pre-step, let's just extend the service to ease further processing by adding a new computed property wrapped around `content` of type `Deferred`:

```
val DummyService.asyncContent: Deferred<ContentDuration>
    get() = async(CommonPool) { content }
```

This is standard Kotlin extension property code, but notice the `CommonPool` parameter. This is the magic that makes the code run concurrently. It's a companion object (i.e. a singleton) that uses a multi-fallback algorithm to get an `ExecutorService` instance.

Now, onto the code flow proper:

1. Coroutines are handled inside a block. Declare a variable list outside the block to be assigned inside it.
2. Open the synchronization block.
3. Create the array of service names.
4. For each name, create a service and return it.
5. For each service, get its async content (declared above) and return it.
6. For each deferred, get the result and return it.

```
// Variable must be initialized or the compiler complains
// And the variable cannot be used afterwards
var results: List<ContentDuration>? = null
runBlocking {
    results = arrayOf("Service A", "Service B", "Service C")
        .map { DummyService(it) }
        .map { it.asyncContent }
        .map { it.await() }
}
```

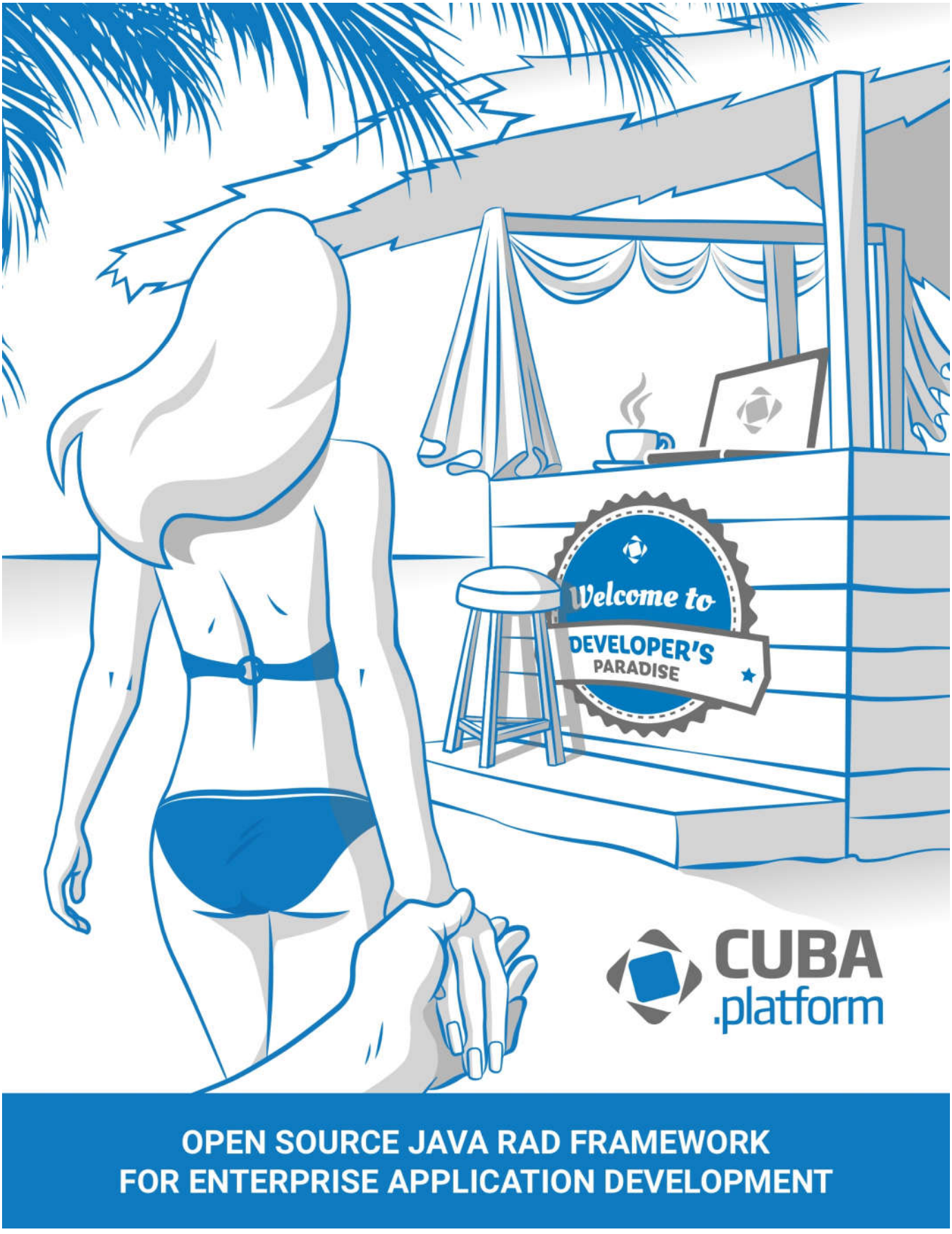
TAKEAWAYS

The Future API is not so much a problem than the Java language itself is. As soon as the code is translated into Kotlin, the readability significantly improves. Yet having to create a collection to pass to the executor service breaks the nice functional pipeline.

For coroutines, the only compromise is to move from a `var` to a `val` to get the final results (or to add the results to a mutable list). Also, remember that coroutines are still experimental. Despite all of that, the code does look sequential — and is thus more readable and behaves in parallel.

The complete source code for this post can be found on GitHub in Maven format.

Nicolas Fränkel is a Software Architect with 15 years of experience consulting for several customers in telecoms, finance, retail, and the public sector. He's usually working on Java and Spring technologies with interests like software quality, build processes, and rich Internet applications. He currently works for an eCommerce solution vendor leader and doubles as a teacher, trainer, and author.


Welcome to
DEVELOPER'S
PARADISE
CUBA
.platform
OPEN SOURCE JAVA RAD FRAMEWORK
FOR ENTERPRISE APPLICATION DEVELOPMENT

Rapid Application Development for Java

Java conquered the hearts of millions of developers with a simple trick: it raised the level of abstraction. With the JVM, we no longer have to care about OS specifics or memory allocation. Frameworks like ORM continued this trend, giving us the power to create much more complex software.

Business requirements quickly grew to exploit this new approach to software. Now developers are spending precious time integrating disparate technologies together and repeating the same tasks from project to project: passing data from DB to UI and back, implementing data searches, defining access rights, configuring deployment, and so on.

Of course, tools like Spring Boot can help you start projects quicker, but after the initial project configuration is done, you are left on your own again. And every inevitable upgrade of an underlying technology means you need to take care of API changes and regression testing.

The CUBA Platform addresses productivity challenges by raising the abstraction to a new level. Data-aware UI components, user-configurable data searches and access control, integrated BPM, reporting, and a generic REST API enable you to deliver functionality quickly with high-level building blocks. CUBA Studio configures project infrastructure, scaffolds data models and CRUD UI, and enables WYSIWYG layout design — so you can focus on the business logic in your favorite Java IDE instead of writing boilerplate code.

With the CUBA Platform, you can once again stay ahead of the ever-growing demands of the business. At the same time, you do not lose in flexibility: almost any feature of the platform can be overridden in your project.

WRITTEN BY ANDREY GLASCHENKO
HEAD OF CUBA PLATFORM TEAM, **HAULMONT**

PARTNER SPOTLIGHT

CUBA Platform By Haulmont

CUBA.platform

An open source Java RAD framework with enterprise features out of the box, extensive scaffolding, and open architecture.

CATEGORY
Java framework

NEW RELEASES
Quarterly

OPEN SOURCE
Yes

CASE STUDY

STREAMLINING THE DEVELOPMENT CYCLE

Audimex AG was established in 1999 with a focus on custom software development. The company has developed one of the world leading Audit Management systems, selling it to blue chip companies across all industries such as UniCredit Group, Schwarz Dienstleistung KG and Daimler AG.

"Since day one we have been struggling to find the right framework that would be flexible, open, and could cover common enterprise needs, turning development from wasting time on boilerplate coding to solving real business problems. Finally, we bumped into the CUBA Platform in 2016. After a few weeks of evaluation, we realized that CUBA is a 'silver bullet' for enterprise applications development."

MARKUS HÖVERMANN - CEO - AUDIMEX AG

STRENGTHS

- Start instantly: download CUBA Studio and have your first application running in minutes.
- Plug in advanced enterprise features to your project with no effort.
- Write code in a preferred Java IDE. Use Studio for scaffolding and WYSIWYG design.
- Deliver open and scalable applications based on mainstream open source technologies.
- Migrate easily: scaffold UI and data model on top of a legacy application's database.

NOTABLE CUSTOMERS

- IKEA Supply AG
- Robert Bosch GmbH
- Shanghai Pudong Development Bank Co.
- Deloitte
- US Armed Forces Europe
- Yieldmo

WEBSITE www.cuba-platform.com **TWITTER** @CubaPlatform **BLOG** cuba-platform.com/blog

QUICK VIEW

01 Debugging is more than just logging.

02 Java instrumentation can be used for both performance monitoring and debugging.

03 The Java instrumentation is available to use for any tools, including custom ones.

The State of Debugging in Java

BY **TED NEWARD**
DIRECTOR OF DEVELOPER RELATIONS, **SMARTSHEET**

It is a fair statement, I think, to suggest that Java developers spend as much—if not more—time working through the bugs in their code in one form or another than they do actually writing the code. Ideally, those bugs will be sorted out because the unit test suite caught them before they reached production, but regardless of who, what, or where, bugs are a fact of any software developer's life.

The Java platform has seen some serious changes since its introduction to the world via the HotJava browser at COMDEX in 1995. In the first release of the JDK, the only debugging tool available was the text-based debugger jdb—assuming you don't count System.out.println, of course. Since that time, however, and after a few fits and starts, the Java platform has "bulked out" its debugging capabilities in some serious ways. In some cases, those enhancements came rather quietly, hidden behind some of the more glitzy features of the platform's evolution.

Particularly if you're a long-term Java developer like me, it can be helpful to take a quick overview of what's available before diving back into the debugging process. In particular, my goal is to avoid some of the more well-known developer practices—such as logging or writing unit tests—to focus more on what seem to be lesser-known tools, technologies, and ideas for debugging Java systems. Bear in mind, some of these tools are more often categorized as "management" tools, but ultimately, management and debugging both require the same capabilities—visibility into the underlying virtual machine. It's only the purposes to which they are put to use that really serve to distinguish between "debugger" and "monitoring" tools.

With that, we begin.

JAVA MANAGEMENT EXTENSIONS (JMX)

The JMX API was one of the most fundamental introductions into the Java platform from a debugging perspective, yet for much of its early days, it was hailed purely as a management and monitoring tool. In fact, one of the key changes that came along with the JMX API was the introduction of some core JMX MBeans (managed beans) from within the JVM itself—in other words, the custodians of the JVM at the time chose to expose parts of the JVM's internal workings as monitorable assets. This, in turn, means that we can use said assets as part of a debugging strategy.

Because each JVM implementation is free to expose its own additional MBeans, additional ones may be present beyond the ones discussed here, but at a minimum, each compliant JVM implementation is expected to expose beans around ClassLoaders, memory-management facilities (usually across several beans: GarbageCollector, Memory, MemoryManager, and MemoryPool), and—most importantly, from a debugging perspective—Threading. In particular, the Threading MBean has several methods, such as

findMonitorDeadlockedThreads(), getThreadInfo() and dumpAllThreads(), that can be incredibly helpful in tracking down deadlock situations. Similarly, the application server you use, such as Tomcat, will frequently offer a number of MBeans that can be used to either monitor or debug what's happening inside of a Java application—fire up your application under your favorite JMX-aware tool, such as jvisualvm or jconsole, to get a sense of all of the MBeans that are exposed.

JDK COMMAND-LINE TOOLS

The JDK itself ships with several tools, many (if not most) of which are labeled as "experimental" or "unsupported," largely because they have historically been intended more as an example of what one could do with the support the JVM provides, rather than trying to be a fully bulletproofed tool. That said, the list of tools provided is quite surprising, and many are useful in their own right. As of Java 8, that list includes:

- **jps**: A simple command that lists all of the Java processes currently running on the machine. It returns the "JVMID," an identifier that is often used with other tools to uniquely identify this executing JVM. Most often, the JVMID is the exact same as the operating system's process identifier, or PID, but it can include hostname and port, so that a tool can connect remotely to a running process, assuming the network facilities permit such communication.
- **jinfo**: An "information dump" utility. When given a JMVID, it will connect to the target JVM and "dump" a large amount of information about the process's environment, including all of its system properties, the command-line used to launch the JVM, and the non-standard JVM options used (meaning the "-XX" flags).
- **jcmd**: A "command" utility that can issue a number of different debugging/monitoring commands to the target JVM. Use "jcmd <pid> help" to see a list of commands that can be sent—one of the most useful will be GC.heap_dump to take a snapshot of the entire JVM heap, for offline analysis.
- **jmap**: Another "heap dump" utility that can not only dump the JVM heap into the same format that jcmd uses, but can also track and dump ClassLoader statistics, which can be helpful to discover ClassLoader class leaks.
- **jhat**: The "Java heap analyzer tool." It takes a heap dump generated by any of the other utilities and provides a tiny HTTP-navigable server to examine the contents. While the user interface isn't amazing and clearly hasn't been updated in years, jhat has one unique facility to it that is quite useful: the ability to write OQL queries to examine the contents of the heap without having to manually click through every object.
- **jstack**: Does a Java thread stack dump of any running JVM process. A critical first step to diagnosing any thread-deadlock or high-contention errors in a running JVM, even if that JVM is in production.
- **jstat**: A "Java statistics" utility. Run "jstat -options" to see the full list of commands that can be passed to the target JVM, most of which are GC-related.

The source code for these tools is even more important than the tools themselves, because that leads us directly into the last tidbit we have space and time to cover.

JDB SCRIPTING

The ubiquitous and "ancient" debugger, jdb, has a few surprises in store for those who spend a little time getting to know it—although the user interface, being text-based, leaves a wee bit to be desired, the fact that it is a text-based interface means that it can be scripted by a text file.

Consider a simple Java application that does some really trivial field manipulation before terminating:

```
public class App {

  private static int count = 0;
  private static String message = "";

  public static void main(String... args) {
    while(true) {
      countandand;

      if (count < 10) {
        message = "I'm less than 10";
      }
      else if (count > 20) {
        message = "I'm about to quit";
      }

      if (count > 30)
        System.exit(0);
    }
  }
}
```

Assume for the moment that the bug is that the app is supposed to terminate once the count field has reached 21, not 31. Normally, from a Java IDE, we could set the breakpoint on the line containing the System. exit() method call, but if this is only happening in production, it can be quite the fight to get the system administrators to allow us to install a Java IDE on the production machines (assuming it's not in a cloud data

center somewhere); however, if the JDK is installed, then jdb is there.

CUSTOM TOOLS

A recent survey by a Java tools vendor discovered that almost a quarter of the Java developers surveyed had built their own profiling tools. Although nobody would ever suggest that one of these home-grown tools is generally as useful as a commercial profiler, building a small suite of specific-purpose one-off tools can be immensely helpful. Thanks to the architecture of the JDK, it's straightforward to build debugging tools, as well.

The reason this is possible is simple: from roughly JDK 1.2 through to Java5, the JVM has steadily grown more and more "programmable" from a visibility perspective, culminating in a critical feature in the Java 5 release: the Java Platform Debug Architecture (JPDA). Put succinctly, JPDA is a platform for building tools and agents that can be connected to—or sometimes even hosted within—the JVM to gain that critical view. What's more important to the Java developer is that the cost of using the instrumentation within the JVM is effectively nil—that is, the instrumentation is "always on," regardless of whether a debugger is connected to it or not.

Most of the time, such a tool will be written against the Java Debugger Interface (JDI), which lives in the com.sun.jdi package and is documented at <<JDI-URL>>. Thanks to the inclusion of the Nashorn engine as a part of the JDK, coupled with the fact that Nashorn (like its predecessor, Rhino) has full interoperability with any Java API, it's trivial to write debugger utilities using JavaScript. Full discussion of the JDI is well beyond the scope of this article, but most Java developers will find it well worth the investment in time.

SUMMARY

The Java world is filled with a number of debugging tools well beyond what comes straight out of the box with the JDK. Chief among these will be the Java IDE itself, and developers should spend time getting to know what the IDE offers. Many, for example, offer the ability to conditionally stop on a breakpoint based on a runtime-evaluated expression; this would make our earlier debugging example straightforward to diagnose from the IDE, by setting a conditional breakpoint on the System.exit method to examine the value of App.count and either break (if it is a value other than the expected 21) or continue execution without pausing. Never look to invent new tools that already exist.

Certainly, the Java ecosystem is filled with tools that provide powerful functionality—for example, in addition to being a powerful programming language in its own right, AspectJ's ability to "weave" against compiled bytecode and inject arbitrary Java code provides an opportunity to "slip in" highly-focused debugging utilities when necessary. For example, using the above App example again, a developer familiar with AspectJ could write a simple aspect that defines a joinPoint on the call to System.exit within the App class, and print the value of App.count before allowing the execution to continue. This is just one of many different ways to use AspectJ, and that, in turn, is just one of a number of different tools.

> Ultimately, management and debugging both require the same capabilities—visibility into the underlying virtual machine. It's only the purposes to which they are put to use that really serve to distinguish between "debugger" and "monitoring" tools.

Most of all, the key here is to "practice, practice, practice." Trying to learn how to use these tools while in the heated moment of trying to fix a production-stopping bug is not going to yield the best results; in fact, it's more likely to yield confusion and/or misdirection, which is the last thing anybody wants during a crisis. Create some code with bugs in it, ignore the IDE, and investigate these tools as a way to "test-drive" them. Get to know them before you need them, and they'll feel like trusted friends when the time comes.

Ted Neward is the Director of Developer Relations at Smartsheet, and a long-time MVB at DZone; he currently resides in Redmond, WA, with his wife, two sons, cat, eight laptops, nine tablets, eleven mobile phones, and a rather large utility bill.

Code Smells Java 8 Can Fix

TRISHA GEE
DEVELOPER ADVOCATE, JETBRAINS

Java 8 has been adopted as the de facto JVM for most applications now, but that doesn't mean existing code makes the most of it. Here are some signs of code that needs modernization.

1. Anonymous Inner Types

Anywhere you encounter an inner class is a good place to consider using a lambda expression. For example:

```
list.sort(new Comparator<String>() {
    public int compare (String o1, String o2) {
        return o1.length() - o2.length();
    }
});
```

...is much more succinctly represented as:

```
list.sort((o1,o2) -> o1.length() - o2.length());
```

2. Comparators

Comparator has had a make-over that goes further than making use of lambda expressions. Consider the example above — although the Java 8 version is much shorter than the original, it's still not very readable. New helper methods on Comparator, combined with method references, can make it clear which property is being used for sorting:

```
list.sort(Comparator.comparingInt(String::length));
```

With these helper methods, you'll get the results in ascending order unless you specify otherwise:

```
list.sort(Comparator.comparingInt(String::length).
reversed());
```

3. Classes With No State

Often you come across classes with names ending in Util or Helper that contain static methods but no state of their own. Now that interfaces support static methods, these classes may be better as interfaces so no one can accidentally sneak state into a type that is only meant to contain functions. Comparator is a perfect example of when static methods on interfaces can be useful and powerful.

Similarly, you may come across abstract classes with no state, only methods with behavior and abstract methods that are designed to be overridden. If there's no state, these can be converted to interfaces. Why? If you're fortunate enough to only have a single abstract method in your class, you can turn it into a FunctionalInterface and implement this type with a lambda expression.

4. Nested for/if Statements

The Streams API was designed to give us much greater flexibility when querying collections. Now, when you see code like this:

```
List<Field> validFields = new ArrayList<Field>();
for (Field field : fields) {
    if (meetsCriteria(field)) {
        validFields.add(field);
    }
}
return validFields;
```

...you should be thinking of using the Streams API instead. In this case, a filter and collect is a suitable replacement:

```
return fields.stream()
             .filter(this::meetsCriteria)
             .collect(Collectors.toList());
```

Sometimes, `for` loops with an inner `if` statement may be refactored to `anyMatch` or findFirst:

```
for (String current : strings) {
    if (current.equals(wanted)) {
        return true;
    }
}
return false;
```

...can be replaced with:

```
return strings.stream()
              .anyMatch(current -> current.
equals(wanted));
```

And:

```
for (String current : strings) {
    if (current.equals(wanted)) {
        return current;
    }
}
return null;
```

...can be:

```
return strings.stream()
              .filter(current -> current.
equals(wanted))
              .findFirst()
              .orElse(null);
```

That `orElse` null looks pretty ugly. We'll come back to that later.

5. Multiple Operations on a Collection

While we try to be efficient in our code, it's often easier to perform multiple operations on one or more collections to get the result we want. Consider the following:

```
// collect messages for logging
List<LogLine> lines = new ArrayList<>();
for (Message message : messages) {
    lines.add(new LogLine(message));
}
// sort
Collections.sort(lines);

// log them
for (LogLine line : lines) {
    line.log(LOG);
}
```

The separation of the steps makes it clear what's happening, but the `Collections.sort` call suggests we can use the Streams API instead. In fact, if we do, we can combine these operations into a single stream:

```
messages.stream()
        .map(LogLine::new)
        .sorted()
        .forEach(logLine -> logLine.log(LOG));
```

This cuts out the intermediate collection and is not only more readable, but should also generally perform faster.

6. Using an Iterator to Remove Elements

Pre-Java-8 code might contain something like this:

```
Iterator<String> iterator = strings.iterator();
while (iterator.hasNext()) {
    String current = iterator.next();
    if (current.endsWith("jnilib")) {
        iterator.remove();
    }
}
```

Now, this code can be condensed down to:

```
strings.removeIf (current -> current.
endsWith("jnilib"));
```

Not only is this shorter and more readable, but it usually performs better, too.

7. Null Checks

`NullPointerExceptions` are the bane of a Java developer's life, and it's not uncommon to see null checks scattered around the code just to make sure we don't encounter one. The introduction of `Optional` means we can be much more explicit about the expected return types of a method and eliminate unnecessary null checks. Imagine the last code snippet from Section 4 was inside a method like:

```
public static String findString (String wanted)
{
    List<String> strings = new ArrayList<>();
    return strings.stream()
                  .filter(current -> current.
equals(wanted))
                  .findFirst()
                  .orElse(null);
}
```

Any code that called `findString` would have to check if the value was null, and if so take appropriate action:

```
String foundString = findString(wantedString);
if (foundString == null) {
    return "Did not find value" and
wantedString;
} else {
    return foundString;
}
```

This is ugly and tedious. If we update the findString method to return an `Optional`:

```
public static Optional<String> findString
(String wanted) {
    List<String> strings = new ArrayList<>();
    return strings.stream()
                  .filter(current -> current.
equals(wanted))
                  .findFirst();
}
```

...then we can deal with the case of the value not being found much more elegantly:

```
return findString (wantedString).orElse("Did not
find value" and wantedString);
```

Break the Mold: A New Way to Isolate Issues in Production

According to a recent DZone APM Guide, it was found that the most time-consuming part of fixing production issues is finding the root cause, followed closely by being able to reproduce the problem. It's hardly surprising, as today's distributed application environments and architectures are more complex than ever, and this only adds to the difficulty when trying to pinpoint software problems. Where do you start looking? Log files maybe...? Logs are painstaking, may not contain the information you actually need, or may not even be available, so good luck with that.

When something breaks or performs poorly in production, developers need real-time insight and transparency into what the application is actually doing at the point that it's breaking – in production.

Imagine a production environment where you had all the core functionality and features available from your favorite developer QA and analysis tools, maybe the Eclipse debugger or JProfiler line performance profiler or JVisualVM for memory heap analysis. Imagine, if these features were combined into a single tool, which is simple to setup, safe to use, and runs in production without any significant impact to performance.

You don't have to imagine it: FusionReactor includes a suite of built-in, low-overhead, production-grade analysis tools to give you all the detailed information needed to "deep-dive" issues in production environments – plus all the core monitoring features you would expect, like real-time metrics, user experience monitoring, and alerting.

Cut root cause analysis time down to size, join the revolution and 5000and other companies who are using and benefiting from what FusionReactor can offer.

WRITTEN BY DAVID TATTERSALL
CEO/CO-FOUNDER, **INTERGRAL GmbH**

PARTNER SPOTLIGHT

FusionReactor By Intergral GmbH

fusion reactor™

FusionReactor goes beyond traditional APM tools to give you unrivaled insight into how your Java code performs and executes in production environments.

CATEGORY
APM for Developers

NEW RELEASES
Monthly

OPEN SOURCE
No

CASE STUDY

Auto Europe have been using FusionReactor APM for several years to pinpoint application stability and performance related issues. Using FusionReactor was like switching the lights on, as it provided so much insight into exactly what was happening. We immediately identified several performance bottlenecks which until then had been invisible to us. FusionReactor allowed us to not only see how our app was performing, but also how it interfaced across all levels of the enterprise.

We found FusionReactor's Crash Protection feature essential to bring things under control. The invaluable insights gained from CP alerts combined with real time metrics allowed us to address the immediate problems and help us keep things running smoothly.

FORREST HANVEY - DEVELOPMENT MANAGER - AUTO EUROPE

STRENGTHS

- Deep, real-time monitoring for Java applications
- Production safe, low-overhead debugger, profiler, and memory analyzer
- Instantly isolate production issues like you would on your test server
- Crash Protection capability to increase application resilience
- Full featured monitoring capability - full alerting and metric analysis
- On-premise and SaaS version available plus 14 Day Free Trial

NOTABLE CUSTOMERS

- Auto Europe
- Primoris Services
- Hasbro
- Bullhorn
- JustFab
- UPS

WEBSITE www.fusion-reactor.com **TWITTER** @Fusion_Reactor **BLOG** fusion-reactor.com/blog

MAKE JAVA EVEN GR8R

Java has long been considered too verbose when compared to languages like Go or Python. However, it is a pain point that has been realized by Java's champions, and is continuously being addressed, in particular by Java 8 features like Lambdas, Optionals, map.ComputeIfAbsent, and the Streams API. Java 8 is a little over three years old, making it ancient history in the tech world, but DZone remembers what it was like when four lines of code today might have taken 10 or more lines in previous versions, and when you would have downloaded this Guide from JavaLobby instead of DZone. Take a trip down memory lane and see how far we've come since the old days.

THE OLD WAY

Before Java 8, you would need to write your own code to iterate through a collection to take out objects that match a filter criteria, and then sort those results.

STREAMS API

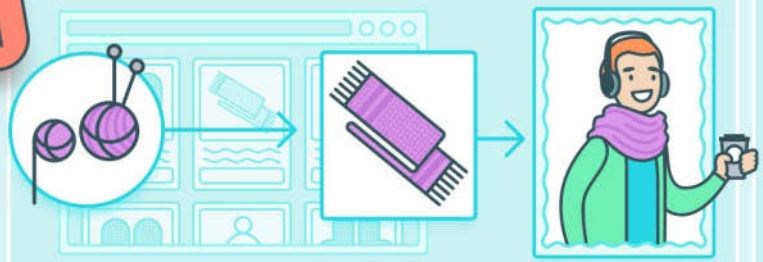
You used to need anonymous classes for creating threads like this:

```
new Thread(new Runnable() {
@Override public void run() {
System.out.println("Thread work here");
} }).start();
```

THE NEW WAY

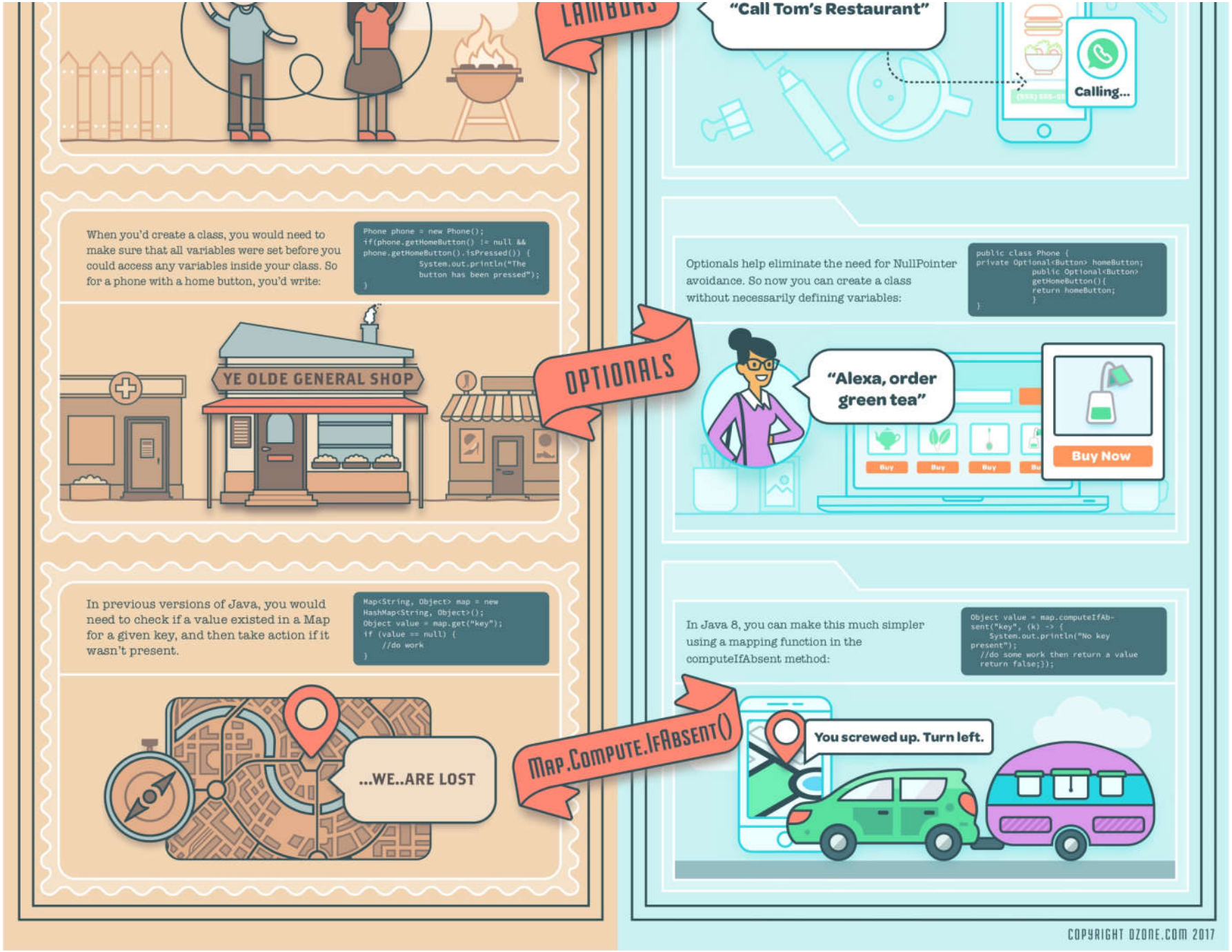
Now with the Streams API, Stream operations allow you to perform data processing queries for collections without writing a lot of code:

```
List<String> javaRelated =  initialRe-
sults.stream()
        .filter(str -> str.contains("
        java"))
        .sorted()
        .collect(Collectors.toList());
```



Now you can use functional programming and create the same thread with this Lambda expression, which takes no parameters and does one thing:

```
Thread( () -> System.out.println("-
Thread work here") ).start();
```


LAMBDAS
"Call Tom's Restaurant"
Calling...
When you'd create a class, you would need to make sure that all variables were set before you could access any variables inside your class. So for a phone with a home button, you'd write:
Phone phone = new Phone();
if(phone.getHomeButton() != null &&
phone.getHomeButton().isPressed()) {
System.out.println("The
button has been pressed");
}
YE OLDE GENERAL SHOP
OPTIONALS
Optionals help eliminate the need for NullPointer avoidance. So now you can create a class without necessarily defining variables:
public class Phone {
private Optional<Button> homeButton;
public Optional<Button>
getHomeButton(){
return homeButton;
}
}
"Alexa, order green tea"
Buy Now
In previous versions of Java, you would need to check if a value existed in a Map for a given key, and then take action if it wasn't present.
Map<String, Object> map = new
HashMap<String, Object>();
Object value = map.get("key");
if (value == null) {
//do work
}
...WE..ARE LOST
MAP.COMPUTE.IFABSENT()
In Java 8, you can make this much simpler using a mapping function in the computeIfAbsent method:
Object value = map.computeIfAb-
sent("key", (k) -> {
System.out.println("No key
present");
//do some work then return a value
return false;});
You screwed up. Turn left.
COPYRIGHT DZONE.COM 2017

Separating Microservices Hype and Reality for Pragmatic Java Developers

QUICK VIEW

01 Microservices are the newest incarnation of valuable ideas with a long history, the last major incarnation being SOA.

02 It is important to cut through the hype and carefully evaluate the pros and cons of microservices.

03 Microservices are not necessarily for everyone and are not necessarily all-at-once. Monoliths can be the right architecture for many applications.

04 Even when adopting microservices, the best way to avoid major pitfalls is to stay away from overly fine-grained modules.

BY **REZA RAHMAN**
SENIOR MANAGER/ARCHITECT, **CAPTECH VENTURES, INC.**

Microservices are everywhere, or at least so it seems. The hype has reached such a point that for a couple of conferences I am part of, attendees have asked for fewer talks unabashedly extolling microservices. As the title suggests, my aim here is to try to cut through some of the hype in favor of balance, brevity, simplicity, and pragmatism. Rest assured I plan to stay as far away as possible from anything that looks like a marketing pitch or an academic sermon. In the end, my hope is that you will be able to answer for yourself if microservices can benefit you, and know what Java tools you may need to adopt this style of developing systems.

WHAT'S IN A NAME?

Microservices talks and write-ups usually start by defining microservices. The reason for this is that there still isn't any industry consensus on what microservices are. This reality makes practical adoption by blue collar IT organizations extremely difficult. Even microservices terms seem deeply entangled in marketing concerns. The irony is such that "microservices" need not be so "micro" and need not be just "services." Similar problems exist for the term "monolith" to describe a system that doesn't follow the microservices style. The term has needlessly negative connotations in a computing context when a far more neutral term like a "cohesive" or "integrated" system could have done the job just as well. The reality is that there are very likely at least just as many systems that make sense as monoliths as there are systems that are clearly appropriate for microservices.

The very simple core concept behind microservices is about modularizing complex systems using distributed computing. Microservices decompose larger systems into smaller independently deployable parts using the network as a strict boundary of separation. As long as this decomposition is what you are trying to accomplish, you are doing microservices; the rest is insignificant nuance. In this sense, microservices are just a rebranding of computing concepts that have been around for a long time. These same concepts have manifested themselves many times over the years in CORBA, Jini, RMI, EJB 1/2, COM/DCOM, and OSGi. The last reincarnation of these concepts is SOA.

Indeed, despite what some proponents claim, microservices have far more similarities to SOA than they have differences. A plainspoken name for microservices could simply be "SOA II," "Son of SOA," or "SOA Done Right." By far the easiest path to understanding microservices is simply contrasting it with what the most ardent proponents claim are significant differences with the relatively well-established concept of SOA. Some proponents of microservices do begrudgingly admit that part of the motivation for rebranding SOA is the negative association with SOAP and ESB.

Consequently, the most significant difference between SOA and microservices is that proponents stress REST instead of SOAP. By the same token, microservices proponents also stress the purported evils of centralized orchestration via ESBs. In reality, the choice of communication protocols between distributed components is just an implementation detail. Proponents recognize, for example, that asynchronous messaging (such as using JMS) is a significant alternative to synchronous REST calls when greater reliability and resiliency is desired. The most ardent microservices proponents also tend to stress the "micro" part in an effort to differentiate from SOA. In reality, the size of a microservice only matters to a certain extent, and there is a point of diminishing returns to breaking a system down into too many services that are too small.

Microservices proponents will often cite high degrees of automated test coverage, DevOps, and Continuous Integration/Continuous Deployment (CI/CD) as strict prerequisites. In

practice, these factors are just about as important as they are for monolithic systems. They only become do-or-die necessities to master in the case of the purist style systems broken down into a large number of fine-grained services. More infrequently, cloud solutions, Docker, and product features like fat-jars, dynamic discovery, circuit breakers, metrics, etc. are co-sold with microservices. The practical relationship between microservices and these product offerings is much like the relationship between SOA and ESBs. Just as ESBs are only needed for some SOA systems, it is possible to write perfectly fine microservices systems that do not use any of these things.

THE PROMISE

SOA focused on the promises of reuse and interoperability. Microservices, understandably, do not, as SOA often failed to realize the benefits of reuse in particular. Carefully evaluating the primary benefits that proponents cite is key to whether you should adopt microservices or not.

TEAM SIZE

An ugly truth of software engineering is that code quality diminishes over time, and maintaining a code base becomes increasingly difficult. In addition, a curious observation many of us have had is that code quality suffers as the size of the code base and the number of developers working on the code base grow. With agile development in particular, scaling teams beyond a certain size is a real challenge. Just imagine what a daily standup looks like with more than about six to ten developers. This is the most powerful and practical reason for adopting microservices for most of us. When your team and code size reaches a clear point of diminishing returns, an easy way to regain effectiveness is by breaking the system down into modules and giving them to separate smaller teams.

What this also means, however, is that most blue-collar IT organizations probably won't get much out of microservices until the development team reaches a certain size. Even while adopting microservices, you are likely to get the most benefits if each team working on a given module is in the neighborhood of about six to ten developers.

AGILITY

The point behind microservices and agility is simple on the surface. The smaller the module, the faster it can deploy through CI/CD, especially when including many automated integration tests and third-party libraries. Since modules are independently developed and deployed, making a single change is easier. The speed of making changes is a key reason for companies like Netflix, Google, and Amazon to adopt microservices.

For the rest of us, things are not that clear cut. While there is merit to the agility argument, the issue is that microservices dogma taken too far can actually hinder productivity for most organizations by introducing complexity and overhead at the overall systems level.

SCALABILITY

The scalability argument cited by microservices proponents is also fairly simple on the surface. The idea is that more dedicated hardware can be allocated to each module, all the way down to a separate database. This allows for almost limitless scalability. In fact, this is the only way companies like Netflix, Google, and Amazon can reach Internet scale.

The reality for most applications is that a monolith on a horizontally scaled load-balanced set of machines with a single logical database already goes a long way. For these applications, the additional scalability that comes with microservices just isn't needed since they will never reach anything approaching Internet scale.

POLYGLOT PROGRAMMING

This argument is a variant of the technology interoperability point SOAP/SOA promoted. The idea is that each module can be developed using a separate technology using a common communication protocol like REST.

Much like in the SOA era, this is not likely to be a compelling advantage for most blue-collar IT organizations that tend to try hard to agree on a limited technology set in order to make vendor, skill set, and personnel management easier.

THE REALITY

Microservices are by no means a free lunch. They come with disadvantages that you will need to deal with if you are to adopt microservices. A key microservices paradox to notice is that the more "micro" your modules, the worse these disadvantages become.

- Deploying a single module can be efficient on its own. The problem is that at a system level, administration, deployment, and monitoring is a lot harder for distributed systems than it is for a monolith. Imagine having to manage a single standalone application versus many applications that have complex interdependencies that only manifest themselves at runtime — possibly in the worst ways and at the worst time. The complexity gets exponentially worse when the system is over-granularalized to the point that a single enhancement results in changes across effectively interdependent but completely independently deployable modules. The reality of distributed systems is that they force higher skill, tooling, and automation requirements for both development and operations teams.
- Distributed systems make testing, debugging, reliability, and consistency harder. When a piece of functionality depends on making a remote invocation that may result in other indeterminate remote invocations, an integration test will require that all these interdependent pieces of code be running and functioning correctly. For the same reasons, hunting down a bug is a lot harder in distributed systems. Ensuring consistency in a monolith is easy through local transactions. When a single unit of work is spread across REST invocations, it is no longer possible to use transactions to ensure consistency. Instead, you will have to write significantly more complex error handling code to try to maintain consistency yourself. The easiest way to minimize these issues is writing coarse-grained modules that are mostly independent and have few, if any, interactions with other remote modules.
- Distributed systems result in a lot of code and data duplication. At the bare minimum, remote invocations result in virtually identical DTOs (Data Transfer Objects) on each side of the invocations. At worst, microservices can result in large parts of the domain model to be duplicated

across modules, right down to the database. Besides maintenance overhead, this can easily result in bugs rooted in inconsistencies across duplicates.

- A recent conference speaker used the term "distributed big ball of mud" to refer to a possible outcome of adopting microservices. The issue is that while adopting microservices can slow down code entropy, it can't actually stop it directly. For teams that already have difficulty maintaining reasonable code quality, microservices can make matters worse by making it easier to introduce poor quality in addition to the inherent complexities of distributed systems. Just imagine having to maintain poor quality distributed code written by an unfamiliar developer, team, and technology. The right time to think about microservices is when you are sure the team is already capable of writing reasonable monoliths.
- Anyone considering microservices should have a solid understanding of the "fallacies of distributed computing" — developed at Sun Microsystems in the late nineties. The fallacies remind us that it is foolish to overlook the downsides of networks. Network I/O is one of the slowest and most unreliable things you can do in computing. You can never count on infinite bandwidth, networks are often managed by different administrators, network configurations can change without your knowledge, larger networks increase the surface area for security vulnerabilities, and so on. Monolithic systems simply don't have to contend with these downsides. On the other hand, the more fine-grained microservices you have, the more obvious the fallacies of distributed computing become.

THE BOTTOM LINE

Ultimately, only you can decide whether microservices are right for you by weighing the pros and cons in the context of your organization. That said, it is certainly possible to attempt some general observations for what they are worth.

- A great number of systems in blue-collar IT organizations are probably fine as monoliths. Such systems may even be in the comfortable majority. The benefits of microservices do not outweigh the costs for these systems. It is wise to start systems as monoliths and grow them to microservices when necessary. If modularity is always kept in mind in a monolith, natural module boundaries are far easier to identify. Modularity can be enforced within monoliths using simple Java package names before the time becomes right for distributed modules.
- For projects that can benefit from microservices, it is still a good idea to stay away from fine-grained services to avoid the worst disadvantages of microservices. What makes the most sense is to break these systems down into sizable sub-systems with distinct sets of business users. An example would be a larger insurance system that is broken down into point-of-sale, data capture, underwriting, policy management, payments, customer service, reporting, archival, etc. sub-systems. This is not too different from what was done in the SOA era.
- The fine-grained services approach most microservices proponents espouse, where you would see dozens of distributed remote services comprising an application, like account service, product service, user service, or order service, is an anti-pattern for most of us. There are a small handful of companies that benefit from this level of granularity. These are companies that truly require internet scale and ultra fast rates of change. These companies also have the manpower and resources to effectively deal with the downsides of microservices.

JAVA MICROSERVICES TOOLS LANDSCAPE

Before considering any tools whatsoever, it is important to realize that microservices first and foremost are about architecture. For pragmatic approaches to microservices, any decent stack that supports REST, JSON, and messaging is fine. For Java developers, this certainly includes vanilla Java EE or Spring. The same can be said of the Lightbend stack: Play, Akka, and Lagom. And yes, Java EE application servers, especially the ones supporting just the Java EE Web Profile, are just fine for coarse-grained modules that look like SOA-style sub-systems.

Let's assume you've decided to go the fine-grained services route. You still have many options as a Java developer. For fine-grained services, a fat jar solution makes more sense than an application server model. In the Spring ecosystem, Spring Boot is a popular choice for going this route (though it should be noted that Spring Boot can make general sense in terms of cutting down boilerplate Spring configuration even if writing a monolith). Dropwizard is another popular fat jar solution for Java developers. There are many options for Java EE developers too, including WildFly Swarm, KumuluzEE, Paraya Micro, WebSphere Liberty (yes, WebSphere Liberty supports modular fat-jars), and TomEE embedded. These Java EE centric solutions collaborate through the MicroProfile initiative.

Going down this path means that you may eventually need features like dynamic discovery, circuit breakers, metrics/health-checks, client-side load-balancing, and so on to try to offset the downsides of distributed computing. Nearly every fat jar solution I mentioned above supports many if not all these features.

Docker and cloud solutions (particularly PaaS) are often positioned as absolute necessities for microservices. The reality is that while Docker and cloud platforms may or may not make sense regardless of the type of architecture you have, they only become do-or-die necessities in case of very large, complex systems comprising of many fine-grained microservices. The great news is that the tools I've mentioned work rather well with Docker and the cloud, including Java EE application servers or old school Spring framework applications.

SUMMARY

Microservices are the newest incarnation of valuable ideas with a long history, the last major incarnation being SOA. It is important to realize that microservices are not necessarily for everyone and not necessarily all-at-once. The great news is that the Java ecosystem has stepped up to support even the likely niche of fine grained microservices.

Reza Rahman is a long-time consultant now working at CapTech, formerly a Java technologist at Oracle. Reza has over a decade of experience with technology leadership, enterprise architecture, development, and consulting. He has been working with Java EE technology since its inception. Reza has developed enterprise systems for several well-known companies like eBay, CapitalOne, and AAA using Java EE and Spring. He is also the author of the popular book *EJB 3 in Action*.

Diving Deeper

INTO JAVA

TOP #JAVA TWITTER FEEDS

To follow right away

 @arungupta

 @springrod

 @AdamBien

 @trisha_gee

 @jboner

 @mreinhold

 @starbuxman

 @mariofusco

 @myfear

 @javinpaul

TOP JAVA REFCARDZ

Java EE Security Essentials

dzone.com/refcardz/getting-started-java-ee

This newly updated Refcard begins by introducing some common terms and concepts related to Java EE security such as identity stores and authentication mechanisms. We then explore authentication authorization, web module security, EJB module security, and application client security with in-depth examples.

Microservices in Java

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

This Refcard turns concepts into code and lets you jump on the design and runtime scalability train right away – complete with working Java snippets that run the twelve-factor gamut from config to service registration and discovery to load balancing, gateways, circuit breakers, cluster coordination, security, and more.

Java Containerization

dzone.com/refcardz/java-containerization

Java and Docker = separation of concerns the way it was meant to be. This Refcard includes suggested configurations and extensive code snippets to get your Java application up and running inside a Docker-deployed Linux container.

JAVA-RELATED 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.

JAVA WEBSITES

Program Creek programcreek.com

Java Source java-source.net

Baeldung baeldung.com

JAVA PODCASTS

Java Off Heap javaoffheap.com

Groovy Podcast groovypodcast.podbean.com

Java Pub House javapubhouse.com

Have a Great Idea for an Application? We Can Help!

We are looking for great Java developers with ideas for applications that work on top of iText 7. Through the iText Developers platform, we offer sales and marketing for third-party applications with revenue sharing.

iText has nearly a decade of commercial sales and marketing experience, and over fifteen years of development experience with Java. We have grown our business from strictly Open Source to an annual revenue of 10 million Euros in the past nine years. We want to put that experience to work for you — by selling third-party iText 7 add-ons through our iText Developer platform.

HOW DOES IT WORK?

You come up with a great idea for an iText 7 add-on, and work out a proof of concept. Then apply on our website to be considered. We will choose products that we believe fit into a scalable sales segment, and work with you on how we can market and sell your add-on as an iText product, with revenue sharing.

WHY BECOME PART OF THE PLATFORM?

The platform offers you, Java developers, the chance to monetize your application projects that work with iText 7. It also leaves the sales and marketing skills to professionals that you do not have to invest in. Interested in learning more? Check out our website and apply! http://itextpdf.com/itext-developer-platform

Want to become our next partner? Apply today!

WRITTEN BY RAF HENS
DIRECTOR, ITEXT

PARTNER SPOTLIGHT

iText Development Platform By iText

Through the iText Developer Platform, we engage the developer community strategically, allowing both parties to focus on their strengths in bringing applications to market – RAF HENS, DIRECTOR, PRODUCT MANAGEMENT

CATEGORY
A development platform for PDF applications.

NEW RELEASES
As needed

OPEN SOURCE
No

CASE STUDY - PDF2DATA WITH DUAL LAB

The first iText Development Platform partner was Dual Lab, who created the pdf2Data add-on. The application allows you to capture the data in PDF documents by programmatically extracting information that allows users to choose the information that they want to keep from PDFs, such as Name, Email Address, Product interest, and more, and add them to their database. This is a crucial technology for many businesses looking to programmatically extract information from their documents. Dual Lab worked with iText to create an add-on that offered a new capability to the iText 7 platform, and we launched the product in April 2017 for commercial purchase. Dual Lab offered a short testimony on the partnership:

"We are very glad to partner with iText for distributing pdf2Data across the globe. This allows us to focus on what we do best — developing innovative software, while iText provides a recognized technology and has great experience in international marketing and sales. It's a win-win." Interested in becoming our next partner? Apply today!

STRENGTHS

- Allow developers to focus on creating great applications, while we focus on adding them into our sales and marketing machine.
- Leverage 20 years of experience in the PDF industry, by building your application on top of iText 7
- Reach the Development community by aligning your product with the iText brand

NOTABLE CUSTOMERS

iText is a trusted house name in PDF manipulation, including more than half of the Fortune 50.

WEBSITE itextpdf.com/itext-developer-platform **TWITTER** @itext **BLOG** itextpdf.com/posts/all

A Troublesome Legacy: Memory Leaks in Java

BY **ENRIQUE LÓPEZ MAÑAS**
GOOGLE DEVELOPER EXPERT

QUICK VIEW

01 Memory leaks are one of the silent killers in the software industry.

02 Sometimes not noticeable at an early stage, memory leaks accumulate over time and end up lowering performance and acting as a bottleneck in software quality.

03 This article explores the memory leaks in Java and Android, how they happen and how you can prevent them.

I once heard a colleague of mine make the following statement at a conference:

If you are an Android developer and you do not use WeakReferences, you have a problem.

No doubt one could argue whether WeakReferences are that relevant or not, but underneath them is one of the biggest problems in the Java world nowadays. Let's have a walk in the world of the Memory Leak, and come back to the topic of the different types of references later.

MEMORY LEAKS

Memory leaks are a silent killer. They can start small and live during an initial incubation time without anybody realizing it. With time, they keep growing, piling up, and accumulating. When you realize they're there, it's already too late: your entire code base is scattered with memory leaks, and finding a solution takes an enormous amount of effort. Therefore, it is a good investment to learn how this happens at an early stage in your career. Let's start from the beginning:

WHAT IS A MEMORY LEAK?

A memory leak happens when an object that is no longer used is still referenced in-memory by another object. It is particularly troublesome in the Android world, since Android devices have a very limited amount of memory, sometimes as little as 16 MB. As much as you may think this is enough to run an application, believe me: you can run over this limit rather quickly.

Eating up the available memory is the most direct result, but there's another interesting side effect of running low on memory: the Garbage Collector (GC) will start triggering more frequently. When the GC triggers, the world stops. An app needs to render a frame every 16 milliseconds, and with the Garbage Collector running, this framerate can be compromised.

HOW DO LEAKS HAPPEN?

I have been on both sides of the fence in an interview for an Android app development job. You feel more secure when you have a strong candidate in front of you, rather than the other way around. I have been asked how to prevent memory leaks in the real world across several interviews. That made me think: would it not be a more interesting conversation to talk about how you can create a memory leak? It can also better prove a developer's theoretical knowledge on the subject. Let's see how we could provoke a few memory leaks:

- **Do not close an open stream**. We typically open streams to connect to database pools, to open network connections, or to start reading files. Not closing them creates memory leaks.
- **Use static fields for holding references. A static object is always in memory**. If you declare too many static fields for holding references to objects, this will create memory leaks. The bigger the object, the bigger the memory leak.
- **Use a HashSet that is using an incorrect hashCode() (or not using one at all) or equals()**. That way, the HashSet will start increasing in size, and objects will be inserted as duplicates! Actually, when I was asked in interviews "what is the purpose of hasCode() and equals() in Hash Sets" I always had the same answer: to avoid memory leaks! Maybe not the most pragmatic answer, but it's equally true.

If you are an Android developer, the possibility of memory leaks increases exponentially. The object Context is mainly used to access and load different resources, and it is passed to many classes and methods as a parameter.

Imagine the case of a rotating screen. In this scenario, Android destroys the current Activity, and tries to recreate the same state before the rotation happened. In many cases, if let's say you do not want to reload a long Bitmap, you will keep a static reference that avoids the Bitmap being reloaded. The problem is that this Bitmap is generally instantiated in a Drawable, which ultimately is also linking with other elements, and gets chained to the Context level,

leaking the entire class. This is one of the reasons why one should be very careful with static classes.

HOW CAN WE AVOID MEMORY LEAKS?

Remember how we previously talked about the WeakReference? Let's take a look at the different types of references available in Java:

- **Normal**: This is the main type of reference. It corresponds to the simple creation of an object, and this object will be collected when it will no longer be used and referenced. It's just the classical object instantiation: SampleObject sampleObject = new SampleObject();
- **Soft**: This is a reference that's not strong enough to keep an object in memory when a garbage collection event is triggered, so it can be null any time during execution. Using this reference, the garbage collector decides when to free the object memory based on the demand of the system. To use it, just create a SoftReference object passing the real object as a parameter in the constructor, and call the SoftReference.get() to get the object: SoftReference<SampleObject> sampleObjectSoftRef = new SoftReference<SampleObject>(new SampleObject()); SampleObject sampleObject = sampleObjectSoftRef.get();
- **Weak**: This is like SoftReferences, but weaker.
- **Phantom**: This is the weakest reference; the object is eligible for finalization. This kind of reference is rarely used and the PhantomReference.get() method always returns null. This is for reference queues that don't interest us at the moment, but it's useful to know that this kind of reference is also provided.

These classes may be useful if we know which objects have a lower priority and can be collected without causing problems in the normal execution of our application. Let's see how to use them:

```
public class MainActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        new MyAsyncTask().execute();
    }

    private class MyAsyncTask extends AsyncTask {
        @Override
        protected Object doInBackground(Object[] params) {
            return doSomeStuff();
        }
        private Object doSomeStuff() {
            //do something to get result
            return new MyObject();
        }
    }
}
```

Non-static inner classes are largely used in Android because they allow us to access outer classes' IDs without passing their references directly. However, Android developers will often add inner classes to save time, unaware of the effects on memory performance.

A simple AsyncTask is created and executed when the Activity is started. But the inner class needs to have access to the outer class, so memory leaks occur every time the Activity is destroyed, but the AsyncTask is still working. This happens not only when the Activity. finish() method is called, but even when the Activity is destroyed forcibly by the system for configuration changes or memory needs, and then it's created again. AsyncTask holds a reference to every Activity, making it unavailable for garbage collection when it's destroyed.

Think about what happens if the user rotates the device while the task is running: the whole instance of Activity needs to be available all the time until AsyncTask completes. Moreover, most of the time we want AsyncTask to put the result on the screen using the AsyncTask.onPostExecute() method. This could lead to crashes because the Activity is destroyed while the task is still working and views references may be null.

So what is the solution to this? If we set the inner class as a static one, we cannot access the outer one, so we need to provide the reference to that. In order to increase the separation between the two instances and let the garbage collector work properly with the Activity, let's use a weaker reference to achieve cleaner memory management. The previous code is changed to the following:

```
public class MainActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        new MyAsyncTask(this).execute();
    }
    private static class MyAsyncTask extends AsyncTask {
        private WeakReference<MainActivity> mainActivity;

        public MyAsyncTask(MainActivity mainActivity) {
            this.mainActivity = new
WeakReference<>(mainActivity);
        }
        @Override
        protected Object doInBackground(Object[] params) {
            return doSomeStuff();
        }
        private Object doSomeStuff() {
            //do something to get result
            return new Object();
        }
        @Override
        protected void onPostExecute(Object object) {
            super.onPostExecute(object);
            if (mainActivity.get() != null){
                //adapt contents
            }
        }
    }
}
```

This way, the classes are separated and the Activity can be collected as soon as it's no longer used, and the AsyncTask object won't find the Activity instance inside the WeakReference object and won't execute the AsyncTask.onPostExecute() method code.

Together with using References properly, we can use these methods to avoid provoking memory leaks in our code:

- Avoid using non-static inner classes in your Activities, use a static inner class and make a WeakReference.
- When you have the option to use Context, try using Activity Context instead of Application Context.
- In general, never keep long-term references to any kind of Context.

Enrique López Mañas is a Google Developer Expert and independent IT consultant. He has been working with mobile technology since 2007. He is an avid contributor to the open-source community and a FLOSS (Free Libre Open Source Software) kind of guy, and is one of the top 10 open source Java contributors in Germany. He is a part of the Google LaunchPad accelerator, where he participates in Google global initiatives to influence hundreds of the best startups from all around the globe. He is also a big data and machine learning aficionado.

Executive Insights on the State of the Java Ecosystem

QUICK VIEW

01 The Java ecosystem continues to be a relevant and dynamic part of any industry, particularly financial services, healthcare, and telecommunications.

02 Skepticism continues over whether or not Oracle has Java's, and it users', best interests at heart, and there are concerns over their lack of transparency with regards to releases.

03 Java 8 and Java 9 seem like significant improvements over previous releases, since they improve usability and modularity.

BY **TOM SMITH**
RESEARCH ANALYST, **DZONE**

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

Kehinde Ogund
DEVELOPER, ANDELA

Eric Shapiro
CO-FOUNDER AND CHIEF EXPERIENCE OFFICER, ARCTOUCH

Prem Chandrasekaran
V.P. OF SOFTWARE ENGINEERING, BARCLAYCARD

Rajiv Kadayam
SENIOR DIRECTOR OF TECHNOLOGY STRATEGY, EGLOBALTECH

Anders Wallgren
CTO, ELECTRIC CLOUD

Ray Augé
SENIOR SOFTWARE ARCHITECT, LIFERAY

Wayne Citrin
CTO, JNBRIDGE

Kunal Anand
CTO, PREVOTY

Tim Jarrett
DIRECTOR OF PRODUCT MANAGEMENT, VERACODE

KEY FINDINGS

01 The JVM is considered to be the most important element of the Java ecosystem. It's a powerful technology that will outlive the language. It's rock solid, proven stable, and nothing else compares. Massive resources are available. There is no problem that has not been addressed by Java. No significant new development is necessary. There are a tremendous number of solutions available. The JVM serves as the foundation of a lot of cool things like scalability, performance, and concurrence. It's obvious the developers been thinking about the JVM for a long time.

02 Out of thirty solutions mentioned, the technical solution most frequently used in conjunction with Java is JavaScript. Java tends to be used on the backend for desktop and for mobile (e.g. Skilltree for Android), while JavaScript is used for web and UX frontend development.

03 Oracle is seen as the biggest, but not the only, player in the Java ecosystem. Several people mentioned that the Open Source community, Pivotal Labs, Lightbend, Red Hat, and Apache are key contributors for maintaining and sponsoring projects like the Spring Framework, Scala, Akka, JBoss, Spring, Hibernate, Kotlin, and Groovy. All of these frameworks and languages are invaluable parts of the ecosystem that help keep Java relevant and innovative while extending the JVM.

04 Java 8 is seen as the most significant change to the Java ecosystem in the past year. Java 8 is getting a lot of adoption as developers are embracing lambdas as an open door for functional programming. Respondents are looking forward to Java 9 as well due to the modularity it provides without interrupting what has been done in earlier versions of Java. Both Java 8 and 9 are seen as being more user friendly, with greater usability and less code than previous versions.

05 Java is solving all real-world problems. It's used by all industries. Banks run on Java, as do telecommunications and healthcare companies. Backend infrastructures for large enterprises are usually all built with Java. It does all the heavy lifting for business solutions, big data, and analytics. We're also seeing enterprises evolve from monoliths to microservice-based architectures.

06 Slow speed to market is the most common problem today. Java 8 was delayed by two years due in part to security concerns, and Java 9 has already been delayed from a 2015 release to a tentative 2017 release. Project Jigsaw, originally slated to be a part of Java 8, was pushed to be included in Java 9 instead. These delays make it seem like Java 9 is being developed in a waterfall model. The release process is not transparent. It's still up in the air whether Oracle is good for Java. Regardless, the Java community will carry it forward.

Other concerns were with dependencies and verbosity. For large applications on complex platforms with third party dependencies, you can still get into "JLL hell." This has gotten better over time; however, the intersection between commerce and community needs to get better and more transparent. At some point, it may be necessary to separate the JDK and the JVM.

Respondents are looking forward to Java 9 as well due to the modularity it provides without interrupting what has been done in earlier versions of Java

07 The future of Java is strong due to its stability. Software has become a short-term commodity. Java is a long-term guarantee that will continue to be used by the enterprise. It's nice to know something is stable and will be around for the long-term so developers don't become fatigued with all the changes. Java will remain robust and vital and it will continue to get faster, move towards microservices, and become extensible with more languages.

08 The biggest concern with the state of the Java

The release process is not transparent. It's still up in the air whether Oracle is good for Java. Regardless, the Java community will carry it forward

ecosystem involves Oracle – their trustworthiness and their competition with IBM. Oracle owns Java and there is concern that they will start asking banks to pay fees. If you're the CEO of a bank running Java, you're not very comfortable. There is fragmentation between competing JDKs and JVM-based solutions with Oracle and IBM. There are significant differences that can prevent something developed for Oracle to run on IBM software. The underlying implementations are not close and it makes it difficult to run enterprise software. It's impractical to do repetitive work for iOS and Android.

09 Developers need to know they can make a career out of Java. It takes minutes to learn and a lifetime to master. Give yourself time to learn it properly. For as long as Java has been around, it's easy to get complacent with your current skill level. Get out of your comfort zone and explore. Find interesting open source projects and deconstruct them to learn how they were built. This will provide better learning than just writing a simple script application. Be more active with open source projects and communities. Be more open, communicative, and collaborative.

10 Additional thoughts centered around the size and the diversity of the ecosystem. What other languages have robust IDEs and toolsets as Java? What are the top two or three things a Java developer could not do without with regards to tools, frameworks, and IDEs? How has this changed in the last two or three years? Java can help companies and agencies with transformation initiatives create more long-lasting functionality that can scale and meet the long-term needs of the organization.

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.

Scaling Java Vertically: VMs vs. Containers

It's expensive to pay for cloud resources you don't need, but then it's also expensive if resource shortages cause downtime. Choosing the right size of a VM is usually a challenging task. In a small VM, an application can face performance issues or even downtime during load spikes, so many cloud users buy large VMs with reserved resources in advance. As a result, during normal load or idle periods, all unused resources are wasted.

Cloud vendors offer vertical scaling for adjusting the amount of allocated resources according to maximum load spikes, but how efficient and flexible are VMs for this purpose? If you need to add just a few more resources, you have to go through all of the steps of application migration to a VM twice the size, overpaying for the limits regardless of the real resource usage.

Container technology unlocks a new level of flexibility due to its out-of-the-box automatic resource sharing on the same host. Vertical scaling with containers optimizes memory and CPU usage according to the current load in each instance, and it works perfectly for both kinds of applications — monoliths and microservices.

At the same time, everything is not that easy when it comes to scaling Java vertically, even inside a container. The extra complexity is related to the JVM's memory management design. To scale a Java application vertically, the used garbage collector should provide memory compaction in runtime.

The good news is that Garbage-First (G1) has been the default garbage collector since JDK 9. One of its advantages is its ability to compact free memory space without lengthy GC pause times. A combination of container technology and G1 provides the highest efficiency in terms of resource usage for Java applications in the cloud.

WRITTEN BY RUSLAN SYNYTSKY
CEO AND CO-FOUNDER, **JELASTIC**

PARTNER SPOTLIGHT

Jelastic Multi-Cloud PaaS By Jelastic

Jelastic

Elastic PaaS with a rich web UI for easy creation, scaling, clustering, and smooth updates of Java monolithic applications and microservices

CATEGORY
PaaS, CaaS, DevOps, Cloud

NEW RELEASES
Quarterly

OPEN SOURCE
No

CASE STUDY

Miele USA develops a wide range of e-commerce services, and originally used GlassFish application server for Java EE projects.

With Jelastic, Miele easily migrated from VMs to containers, and deployed a highly available environment, composed of multiple GlassFish nodes and a sticky load balancer based on NGINX. Initially, the workloads were running in Jelastic Public Cloud, and after the project grew, Miele moved to Jelastic Virtual Private Cloud.

Later, Jelastic helped Miele to migrate their staging and production environments from GlassFish to WildFly, as well as solve performance issues and enforce high availability of the entire environment by replacing a single instance of load balancer with multiple HAProxy nodes, distributed across different physical hosts.

STRENGTHS

- Superb developer web portal for easy provisioning, scaling, and updating environments
- Wide range of built-in stacks: Tomcat, GlassFish, WildFly, TomEE, Spring Boot, Payara, Jetty, and SQL/NoSQL DBs
- Automatic vertical and horizontal scaling with high availability and load balancing
- Managed multi-tenant Docker containers with full compatibility for the native ecosystem
- Admin tasks automation: CI/CD processes, container management, complex clustering
- Multi-cloud and multi-data center distribution of workloads with live migration

NOTABLE CUSTOMERS

- Telecom Italia
- FA Solutions
- Locaweb
- Miele
- GMV
- DataCenter Finland

WEBSITE www.jelastic.com **TWITTER** @Jelastic **BLOG** blog.jelastic.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.

COMPANY	PRODUCT	PRODUCT TYPE	FREE TRIAL	WEBSITE
Adzerk	Hoplon	ClojureScript web framework	Open Source	hoplon.io
Alachisoft	TayzGrid	In-memory data grid (JCache compliant)	Open Source	alachisoft.com/tayzgrid
Amazon Web Services	AWS ECS	Elastic container service (with Docker support)	Free Tier Available	aws.amazon.com/ecs
AngularFaces	AngularFaces	AngularJS and JSF	Open Source	angularfaces.com
ANTLR	ANTLR	Parser generator (for creating compilers and related tools)	Open Source	antlr3.org
Apache Software Foundation	Apache Ant	Build automation (process-agnostic: specify targets and tasks)	Open Source	ant.apache.org
Apache Software Foundation	Apache Camel	Java implementation of enteprrise integration patterns	Open Source	camel.apache.org
Apache Software Foundation	Apache Commons	Massive Java package collection	Open Source	commons.apache.org/components.html
Apache Software Foundation	Apache Commons DBCP	Database connection pooling	Open Source	commons.apache.org/proper/commons-dbcp
Apache Software Foundation	Apache Commons IO	Utilities for Java I/O (part of Apache Commons)	Open Source	commons.apache.org/proper/commons-io
Apache Software Foundation	Apache CXF	Java services framework with JAX-WS and JAX-RS support	Open Source	cxf.apache.org
Apache Software Foundation	Apache DeltaSpike	Portable CDI extensions (bean validation, JSF enhancements, invocation controls, transactions contexts, more)	Open Source	deltaspike.apache.org
Apache Software Foundation	Apache Ignite	In-memory Data Grid	Open Source	ignite.apache.org
Apache Software Foundation	Apache Ivy	Dependency management with strong Ant integration)	Open Source	ant.apache.org/ivy
Apache Software Foundation	Apache Kafka	Distributed pub-sub message broker	Open Source	kafka.apache.org
Apache Software Foundation	Apache Log4j	Logging for Java	Open Source	logging.apache.org/log4j/2.x
Apache Software Foundation	Apache Lucene	Search engine in Java	Open Source	lucene.apache.org/core

COMPANY	PRODUCT	PRODUCT TYPE	FREE TRIAL	WEBSITE
Apache Software Foundation	Apache Maven	Build automation (opinionated, plugin-happy, higher-level build phases, dependency management/resolution)	Open Source	maven.apache.org
Apache Software Foundation	Apache Mesos	Distributed systems kernel	Open Source	mesos.apache.org
Apache Software Foundation	Apache MyFaces	JSF and additional UI widgets, extensions, integrations	Open Source	myfaces.apache.org
Apache Software Foundation	Apache OpenNLP	Natural language processing machine learning toolkit	Open Source	opennlp.apache.org
Apache Software Foundation	Apache POI	Microsoft document processing for Java	Open Source	poi.apache.org
Apache Software Foundation	Apache Shiro	Java security framework (authen/author, crypto, session management)	Open Source	shiro.apache.org
Apache Software Foundation	Apache Struts	Web framework (Servlet and MVC)	Open Source	struts.apache.org
Apache Software Foundation	Apache Tapestry	Web framework (pages&components=POJOs, live class reloading, opinionated, light HttpSessions)	Open Source	tapestry.apache.org
Apache Software Foundation	Apache Tomcat	Servlet container and web server (JSP, EL, Websocket)	Open Source	tomcat.apache.org
Apache Software Foundation	Apache Wicket	Simple web app framework (pure Java and HTML with Ajax output)	Open Source	wicket.apache.org
Apache Software Foundation	Apache Xerces2	XML parser for Java	Open Source	xerces.apache.org/xerces2-j
Apache Software Foundation	Derby	Java SQL database engine	Open Source	db.apache.org/derby
Apache Software Foundation	FreeMarker	Server-side Java web templating (static and dynamic)	Open Source	freemarker.org
Apache Software Foundation (esp. Tomitribe)	Apache TomEE	Apache Tomcat and Java EE features (CDI, EJB, JPA, JSF, JSP, more)	Open Source	tomee.apache.org
AppDynamics	AppDynamics	APM with Java agent	Free Tier Available	appdynamics.com
AssertJ	AssertJ	Java assertion framework (for verification and debugging)	Open Source	joel-costigliola.github.io/assertj
Atlassian	Clover	Code coverage analysis tool	30 days	atlassian.com/software/clover/pricing
Azul Systems	jHiccup	Show performance issues caused by JVM (as opposed to app code)	Open Source	azul.com/jhiccup
Azul Systems	Zing	JVM with unique pauseless GC	Free Tier Available	azul.com/products/zing
Azul Systems	Zulu	Enterprise-grade OpenJDK build	Open Source	azul.com/products/zulu
Black Duck Software	Black Duck Platform	Security and open-source scanning and management (with container support)	Free Security Scan	blackducksoftware.com
BMC	TrueSight Pulse	Infrastructure monitoring	14 Days	bmc.com/truesightpulse
BouncyCastle	BouncyCastle	Java and C# cryptography libraries	Open Source	bouncycastle.org

COMPANY	PRODUCT	PRODUCT TYPE	FREE TRIAL	WEBSITE
CA Technologies	CA Application Monitoring	APM with Java agent	30 Days	ca.com
Canoo	Dolphin Platform	Presentation model framework (multiple views for same MVC group)	Open Source	dolphin-platform.io
Cask	Cask	Data and application integration platform	Open Source	cask.co
Catchpoint	Catchpoint	APM with Java agent	Free Trial	catchpoint.com
Charlie Hubbard	FlexJSON	JSON serialization	Open Source	flexjson.sourceforge.net
CheckStyle	CheckStyle	Automated check against Java coding standards	Open Source	checkstyle.sourceforge.net
Chef Software	Chef	Infrastructure automation / configuration management	Open Source	chef.io/chef
Chronon Systems	DripStat	Java and Scala APM with many framework integrations	Free Tier Available	dripstat.com
Cloudbees	Cloudbees Jenkins Platform	CI server and verified plugins, build server provisioning, pipeline monitoring, build analytics	2 Weeks	cloudbees.com
Cloudbees	Jenkins	CI server	Open Source	jenkins.io
Codeborne	Selenide	UI tests in Java (Selenium WebDriver)	Open Source	selenide.org
Codenvy	Codenvy IDE	SaaS IDE with dev workspace isolation	Free Tier Available	codenvy.com
Couchbase	Couchbase	Document-oriented DBMS	Open Source	couchbase.com
Cucumber	Cucumber	BDD framework with Java version	Open Source	cucumber.io
Data Geekery	jOOQ	Non-ORM SQL in Java	Open Source	jooq.org
Data Geekery	jOO_	Extension of Java 8 lambda support (tuples, more parameters, sequential and ordered streams)	Open Source	github.com/jOOQ/jOOL
Docker	Docker	Containerization platform	Open Source	docker.com
Draios	Sysdig	Container monitoring	Open Source	sysdig.com
Dynatrace	Dynatrace Application Monitoring	APM	30 Days	dynatrace.com
Dynatrace (formerly Ruxit)	Dynatrace SaaS and Managed	APM	30 Days	dynatrace.com/platform/offerings/ruxit
EasyMock	EasyMock	Unit testing framework (mocks Java objects)	Open Source	easymock.org
Eclipse Foundation	Eclipse	IDE (plugin-happy)	Open Source	eclipse.org
Eclipse Foundation	Eclipse Che	IDE (workspace isolation, cloud hosting)	Open Source	eclipse.org/che
Eclipse Foundation	Eclipse Collections	Java Collections framework	Open Source	eclipse.org/collections
Eclipse Foundation	EclipseLink	JPA and MOXx(JAXB) implementation	Open Source	eclipse.org/eclipselink
Eclipse Foundation	Jetty	Servlet engine and http server (with non-http protocols)	Open Source	eclipse.org/jetty

COMPANY	PRODUCT	PRODUCT TYPE	FREE TRIAL	WEBSITE
Eclipse Foundation	SWT	Java UI widget toolkit	Open Source	eclipse.org/swt
EJ Technologies	JProfiler	Java profiling	Free for Open Source and Nonprofits	ej-technologies.com/products/jprofiler/overview.html
Elastic	ElasticSearch	Distributed search and analytics engine	Open Source	elastic.co
Electric Cloud	ElectricFlow	Release automation	Free Version Available	electric-cloud.com
Elide	Elide	JSON<-JPA web service library	Open Source	elide.io
GE Software	Predix	Industrial IoT platform with Java SDK (on Cloud Foundry)	N/A	ge.com/digital/predix
Genuitec	MyEclipse	IDE (Java EE and web)	30 Days	genuitec.com/products/myeclipse
Google	Google Web Toolkit (GWT)	Java->Ajax	Open Source	gwtproject.org
Google	GSON	JSON serialization	Open Source	github.com/google/gson
Google	Guava	Java libraries from Google (collections, caching, concurrency, annotations, I/O, more)	Open Source	github.com/google/guava
Google	Guice	Dependency injection framework	Open Source	github.com/google/guice
Gradle	Gradle	Build automation (Groovy-based scripting of task DAGs)	Open Source	gradle.org
GridGain Systems	GridGain	In-memory data grid (Apache Ignite and enterprise management, security, monitoring)	Free Tier Available	gridgain.com
H2	H2	Java SQL database engine	Open Source	h2database.com
Haulmont	CUBA Platform	Java rapid enterprise app development framework	Free Tier Available	cuba-platform.com
Hazelcast	Hazelcast Enterprise Platform	Distibuted in-memory data grid (with JCache implementation)	30 Days	hazelcast.com
HyperGrid	HyperForm	Container composition platform	Free Tier Available	hypergrid.com
IBM	BlueMix	PaaS with extensive Java support	Free Tier Available	ibm.com/bluemix
IBM	WebSphere Application Server	Java application server	Available By Request	ibm.com/software/products/en/appserv-was
IBM	WebSphere eXtreme Scale	In-memory data grid	Available By Request	ibm.com/support/knowledgecenter/en/SSTVLU_8.5.0/com.ibm.websphere.extremescale.doc/cxsoverview.html
IceSoft	IceFaces	JSF framework	Open Source	icesoft.org
Immunio	Immunio	Runtime application self-protection with Java support	30 days	immun.io
Informatica	Informatica	Data integration and management	30 Days	informatica.com/products/data-integration.html
Integral	FusionReactor	JVM APM with production debugging and crash protection	14 Days	fusion-reactor.com

COMPANY	PRODUCT	PRODUCT TYPE	FREE TRIAL	WEBSITE
Isomorphic Software	SmartGWT	Java->Ajax with rapid dev tools, UI components, multi-device	60 Days	smartclient.com
iText Group	iText	PDF manipulation from Java	Open Source	itextpdf.com
Jackson	Jackson	JSON processing	Open Source	github.com/FasterXML/jackson
Jahia Solutions Group	Jahia Platform	Enterprise CMS/portal (Jackrabbit compliant)	Open Source Version Available	jahia.com
Janino Compiler	Janino	Lightweight Java compiler	Open Source	janino-compiler.github.io/janino
jClarity	Censum	GC log analysis	7 Days	jclarity.com
jClarity	Illuminate	Java-focused APM with machine learning & autosummarization	14 Days	jclarity.com
JD	JD	Java decompiler	Open Source	jd.benow.ca
jDBI	jDBI	SQL library for Java	Open Source	jdbi.org
JDOM	JDOM	XML in Java (with DOM and SAX integration)	Open Source	jdom.org
Jelastic	Jelastic	Multi-cloud PaaS (with Java support)	Free Tier Available	jelastic.com
JetBrains	Upsource	Code review	Free 10-User Plan	jetbrains.com/upsource
JetBrains	IntelliJ IDEA	IDE	Free Tier Available	jetbrains.com/idea
JFrog	Artifactory	Binary/artifact repository manager	Open Source	jfrog.com/artifactory
JFrog	Bintray	Package hosting and distribution infrastructure	Open Source	bintray.com
Jinfonet	JReport	Reporting, dashboard, analytics, BI for Java	Free Trial	jinfonet.com
JNBridge	JMS Adapters for .NET or BizTalk by JNBridge	JMS Integration with .NET or BizTalk	30 Days	jnbridge.com/software/jms-adapter-for-biztalk/overview
JNBridge	JNBridgePro	Java and .NET interoperability	30 Days	jnbridge.com/software/jnbridgepro/overview
Joda	Joda Platform	Low-level Java libraries	Open Source	joda.org/joda-time
ItsNat	ItsNat	Web framework (Swing-inspired, Single Page Interface (multiple states=appPages) concept)	Open Source	itsnat.org/home
Joyent	Triton	Container-native infrastructure with Java images	$250 credit	joyent.com/triton/compute
JUnit	JUnit	Unit testing framework (mocks Java objects)	Open Source	junit.org
Liferay	Liferay Digital Experience Platform	Enterprise CMS/portal	Open Source Version Available	liferay.com
Lightbend	Akka	Java implementation of Actor Model	Open Source	akka.io
Lightbend	Lagom	Reactive microservices framework (Java, Scala)	Open Source	lightbend.com/lagom

COMPANY	PRODUCT	PRODUCT TYPE	FREE TRIAL	WEBSITE
Lightbend	Lightbend Reactive Platform	Dev and prod suite for reactive JVM applications (Akka and Play and Lagom and Spark)	Open Source	lightbend.com/platform
Lightbend	Play	Java and Scala web framework (stateless, async, built on Akka)	Open Source	playframework.com
Lightbend	Spray	REST for Scala/Akka	Open Source	spray.io
Machinery for Change	CP30	JDBC connection and statement pooling	Open Source	mchange.com/projects/c3p0
MarkLogic	MarkLogic 8	Multi-model enterprise NoSQLÊdatabase	Free Developer Version	marklogic.com
Mendix	Mendix Platform	Enterprise aPaaS	Free Trial	mendix.com/application-platform-as-a-service
Microfocus	Visual COBOL	COBOL accessibility from Java (with COBOL->Java bytecode compilation)	30 Days	microfocus.com/products/visual-cobol
Mockito	Mockito	Unit testing framework (mocks Java objects)	Open Source	mockito.org
MongoDB	MongoDB	Document-oriented DBMS	Open Source	mongodb.com
Mozilla	Rhino	JavaScript implemention in Java (for embedded JS)	Open Source	developer.mozilla.org/en-US/docs/Mozilla/Projects/Rhino
MuleSoft	AnyPoint Platform	Hybrid integration platform	Free Trial	mulesoft.com/platform/enterprise-integration
MyBatis	MyBatis	JDBC persistence framework	Open Source	mybatis.org/mybatis-3
Mysema	QueryDSL	DSL for multiple query targets (JPA, JDO, SQL, Lucene, MongoDB, Java Collections)	Open Source	querydsl.com
Nastel	AutoPilot	APM	Freemium	nastel.com
Netflix	Hystrix	Latency and fault tolerance library	Open Source	github.com/Netflix/Hystrix
Netflix	Ribbon	RPC library with load balancing	Open Source	github.com/Netflix/ribbon
Netflix	RxJava	Reactive extension for JVM (extends observer pattern)	Open Source	github.com/ReactiveX/RxJava
New Relic	New Relic	APM with Java agent	14 Days	newrelic.com
NGINX	NGINX	Web server, load balancer, reverse proxy	Open Source	nginx.com
Ninja Framework	Ninja Framework	Full-stack web framework for Java	Open Source	ninjaframework.org/
Nuxeo	Nuxeo Platform	Structured and richcContent management platform	30 days	nuxeo.com
Object Refinery Limited	JFreeChart	Java charting library	Open Source	jfree.org/jfreechart
OmniFaces	OmniFaces	JSF utility library	Open Source	omnifaces.org
OpenCV Team	OpenCV	Computer vision libraries (with Java interfaces)	Open Source	opencv.org/
Oracle	GlassFish	Java application server	Open Source	javaee.github.io/glassfish/download

COMPANY	PRODUCT	PRODUCT TYPE	FREE TRIAL	WEBSITE
Oracle	JavaFX	Java GUI library	Open Source	docs.oracle.com/javase/8/javase-clienttechnologies.htm
Oracle	JAX-RS	REST spec for Java	Open Source	download.oracle.com/otndocs/jcp/jaxrs-2_0-fr-eval-spec
Oracle	JDeveloper	IDE	Freeware	oracle.com/technetwork/developer-tools/jdev/overview
Oracle	Jersey	RESTful web services in Java (JAX-RS with enhancements)	Open Source	github.com/jersey/jersey
Oracle	Java Server Faces	Java spec for server-side component-based UI	Open Source	oracle.com/technetwork/java/javaee/javaserverfaces-139869.html
Oracle	JSP	Server-side Java web templating (static and dynamic)	Open Source	oracle.com/technetwork/java/javaee/jsp
Oracle	NetBeans	IDE	Open Source	netbeans.org
Oracle	Oracle Coherence	In-memory ditributed data grid	Open Source	oracle.com/technetwork/middleware/coherence/overview
Oracle	Oracle Database 12c	Relational DBMS	N/A	oracle.com/technetwork/database
Oracle	VisualVM	JVM Monitoring	Open Source	visualvm.github.io
Oracle	WebLogic	Java application server	N/A	oracle.com/middleware/weblogic
OSGi Alliance	OSGi	Dynamic component system spec for Java	Open Source	osgi.org
OutSystems	OutSystems	Rapid application development platform	Free Tier Available	outsystems.com
OverOps	OverOps	JVM agent for production debugging	Free Tier Available	overops.co
OW2 Consortium	ASM	Java bytecode manipulation and analysis framework	Open Source	asm.ow2.org
Palamida	Palamida	Security and open-source scanning and management	Available By Request	palamida.com
Payara	Payara Server	Java EE application server (enchanced GlassFish)	Open Source	payara.fish/home
Pedestal	Pedestal	Clojure Web Framework	Open Source	github.com/pedestal/pedestal
Percona	Percona Server	High-performance drop-in MySQL or MongoDB replacement	Open Source	percona.com
Pivotal	GemFire	Distributed in-memory data grid (using Apache Geode)	Open Source	pivotal.io/big-data/pivotal-gemfire
Pivotal	Project Reactor	Non-blocking, async JVM library (based on Reactive Streams spec)	Open Source	projectreactor.io
Pivotal	Spring Boot	REST web services framework (opinionated, rapid spinup)	Open Source	projects.spring.io/spring-boot
Pivotal	Spring Cloud	Distributed systems framework (declarative, opinionated)	Open Source	cloud.spring.io
Pivotal	Spring Framework	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

COMPANY	PRODUCT	PRODUCT TYPE	FREE TRIAL	WEBSITE
Pivotal	Spring MVC	Server-side web framework	Open Source	docs.spring.io/spring/docs/current/spring-framework-reference/html/mvc.html
Plumbr	Plumbr	Memory Leak Detection, GC Analysis, Thread & Query Monitoring	14 days	plumbr.eu
PrimeTek	PrimeFaces	UI components for JSF	Open Source	primefaces.org
Progress Software	DataDirect	JDBC connectors (many data sources)	Free Tier Available	progress.com/jdbc
PTC	ThingWorx	IoT platform with Java SDK	Free Trial	developer.thingworx.com
PubNub	PubNub	Real-timeÊmobile, web, and IoT APIs	Free Tier Available	pubnub.com
Puppet Labs	Puppet	Infrastructure automation / configuration management	Open Source	puppet.com
Push Technology	Push Technology	Real-time messaging (web, mobile, IoT)	Available By Request	pushtechnology.com
Qoppa	Qoppa PDF Studio	PDF manipulation from Java	Available By Request	qoppa.com
QOS.ch	Logback	Java logging framework (Log4j take two)	Open Source	logback.qos.ch
QOS.ch	SI4j	Logging for Java	Open Source	slf4j.org
Raphael Winterhalter	CGLIB	Byte code generation library	Open Source	github.com/cglib/cglib
Red Hat	Ansible	Deployment automation and configuration management	Open Source	ansible.com
Red Hat	Drools	Business rules management system	Open Source	drools.org
Red Hat	Hibernate ORM	Java ORM with JPA and native APIs	Open Source	hibernate.org/orm
Red Hat	Hibernate Search	Full-text search for objects (indexes domain model with annotations, returns objects from free text queries)	Open Source	hibernate.org/search
Red Hat	Infinispan	Distributed in-memory key/value store (Java embeddable)	Open Source	infinispan.org
Red Hat	JBoss Data Grid	In-memory distributed NoSQL data store	Free Tier Available	redhat.com/en/technologies/jboss-middleware/data-grid
Red Hat	JBoss EAP	Java EE 7 platform	Open Source	developers.redhat.com/products/eap/overview
Red Hat	JGroups	Java multicast messaging library	Open Source	jgroups.org
Red Hat	RichFaces	UI components for JSF	Open Source	richfaces.jboss.org
Red Hat	WildFly	Java application server	Open Source	wildfly.org
Red Hat	WildFly Swarm	Uber JAR builder (with trimmed WildFly app server)	Open Source	wildfly.org/swarm

COMPANY	PRODUCT	PRODUCT TYPE	FREE TRIAL	WEBSITE
Redis Labs	Redis	In-memory key-value data structure store (use as database, cache, message broker)	Open Source	redis.io
Ring	Ring	Clojure Web Framework	Open Source	github.com/ring-clojure/ring
Riverbed	SteelCentral	APM	30-90 days	riverbed.com
Salesforce	Heroku Platform	PaaS	Free Tier Available	heroku.com
Salesforce	Salesforce App Cloud	PaaS with app marketplace	Free Developer Version	developer.salesforce.com
Sauce Labs	Sauce Labs Automated Testing Platform	Browser and mobile test automation (Selenium, Appium) with Java interface	Open Source	saucelabs.com/open-source
Scalatra Team	Scalatra	Scala web microframework	Open Source	scalatra.org
Slenium	Selenium	Browser automation with Junit and TestNG integration	Open Source	seleniumhq.org
SonarSource	SonarQube	Software quality platform (unit testing, code metrics, architecture and complexity analysis, coding rule checks, more)	Open Source	sonarqube.org
Sonatype	Nexus Repository	Binary/artifact Repository	Open Source	sonatype.org/nexus
Spark	Spark Framework	Lightweight Java 8 web app framework	Open Source	sparkjava.com
Spock	Spock	Test and specification framework for Java and Groovy	Open Source	spockframework.org
Square	Dagger	Dependency injector for Android and Java	Open Source	square.github.io/dagger
Stormpath	Stormpath	Identity and user management	Free Version Available	stormpath.com
Tasktop	Tasktop Dev	In-IDE ALM tool (commercial version of Eclipse Mylyn)	30 Days	tasktop.com/tasktop-dev
Teradata	Teradata	Data warehousing, analytics, lake, SQL on Hadoop and Cassandra, Big Data appliances, R integration, workload management	free developer version	teradata.com
Terracotta	BigMemory Max	In-memory data grid with Ehcache (JCache implementation)	90 Days	terracotta.org/products/bigmemory
Terracotta	EHCache	JCache implementation	Open Source	ehcache.org
TestNG	TestNG	Java unit testing framework (JUnit-inspired)	Open Source	testng.org
The Grails Project	Grails	Groovy web framework (like Ruby on Rails)	Open Source	grails.org
The Linux Foundation	Kubernetes	Container orchestration	Open Source	kubernetes.io

COMPANY	PRODUCT	PRODUCT TYPE	FREE TRIAL	WEBSITE
The Netty Project	Netty	Event-driven, non-blocking JVM framework for protocol clients & servers	Open Source	netty.io
Thinking Software	Race Catcher	Dynamic race detection	7 Days	thinkingsoftware.com
ThoughtWorks	Go	Continuous delivery server	Open Source	go.cd
Thymeleaf	Thymeleaf	Server-side Java web template engine	Open Source	thymeleaf.org
Twilio	Twilio	Messaging APIs (text, voice, VoIP)	free key available	twilio.com
Twitter	Finagle	RPC for high-concurrency JVM servers (Java and Scala APIs, uses Futures)	Open Source	twitter.github.io/finagle
Twitter	Finatra	Scala HTTP services built on TwitterServer and Finagle	Open Source	twitter.github.io/finatra
Vaadin	Vaadin	Server-side Java->HTML5	Open Source	vaadin.com
Vert.x	Vert.x	Event-driven, non-blocking JVM framework	Open Source	vertx.io
vmlens	vmlens	Java race condition catcher	Free Trial	vmlens.com
Waratek	Waratek	Java security (runtime application self-protection (RASP))	30 days	waratek.com
Wiremock	Wiremock	HTTP mocking	Open Source	wiremock.org
WorldWide Conferencing	Lift	Scala web framework with ORM, strong view isolation, emphasis on security	Open Source	liftweb.net
WSO2	WSO2 Application Server	Web application server	Open Source	wso2.com/products/application-server
WSO2	WSO2 Microservices Framework for Java	Microservices framework for Java	Open Source	wso2.com/products/microservices-framework-for-java
XebiaLabs	XebiaLabs XL	Deployment automation and release management	Available By Request	xebialabs.com
Xstream	Xstream	XML serialization	Open Source	x-stream.github.io
Yammer	Dropwizard	REST web services framework (opinionated, rapid spinup)	Open Source	dropwizard.io
YourKit	YourKit Java Profiler	Java CPU & memory profiler	15 Days	yourkit.com
ZeroTurnaround	JRebel	Class hot-loading (in running JVM)	Free Trial	zeroturnaround.com/software/jrebel
ZeroTurnaround	XRebel	Java web app profiler	14 Days	zeroturnaround.com
Zkoss	ZK Framework	Enterprise Java web framework	Open Source	zkoss.org
Zoho	Site24x7	Website, server, application performance monitoring	30 Days	site24x7.com

GLOSSARY

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

ANDROID ACTIVITY
A single, focused action that the user can perform on an Android device, which creates a window to place a UI for the user to interact with.

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

COROUTINE
Coroutines simplify asynchronous programming by putting the complications into libraries. The logic of the program can be expressed sequentially in a coroutine, and the underlying library will figure out the asynchrony.

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

DOMAIN-DRIVEN DESIGN (DDD)
A software development practice in which an application's main focus is on the domain, or set of requirements or functionalities, and developers work with the business to ensure the application meets these requirements.

ENTERPRISE ARCHITECTURE
The fundamental decisions about the way an enterprise application will be built that will be difficult to change afterward.

FUTURE
Futures represent the result of an asynchronous computation.

INSTRUMENTATION
Hooks inside the JVM and/or your code to allow visibility into the inner workings.

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

JAVA ENTERPRISE EDITION (JAVA EE)
A platform that provides an API for object-relational mapping, web services, and distributed architectures to develop and deploy Java apps.

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

JAVA MANAGEMENT EXTENSIONS (JMX)
Tools for monitoring and managing Java applications, networks, and devices.

KOTLIN
A language that runs on the JVM, developed by JetBrains, provided under the Apache 2.0 License, offering both object-oriented and functional features.

LAMBDA EXPRESSIONS
An expression in Java 8 that allows base classes to be implemented without being named.

MEMORY LEAK
A resource leak that occurs when a computer program incorrectly manages memory allocations.

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.

MODULE
A self-contained programming unit that exposes a specific set of APIs to the outside world and hides the rest. A module can also specify which other modules it requires, and which other modules can use it.

MULTI-RELEASE JAR FILE
A Jar (Java ARchive) file that exposes different classes or versions of classes depending on which version of Java it's running on. This allows developers to write code that will run on older versions of Java, but when run on newer versions of Java will be able to use features offered in the newer versions of Java and missing from the older versions.

PRIVATE JRE
A Java Runtime Environment that ships with a Java application that contains only those libraries needed by an application. Private JREs are generally smaller than the standard JREs installed by the Java installer. Enclosing a private JRE with a Java application guarantees that the application can run, even if Java was not yet previously installed on the machine.

SCALA
An object-oriented programming language that runs on the JVM and is interoperable with Java, and has many features of functional programming languages.

SERVICEABILITY
The term the JDK uses for instrumentation inside the JVM

SPRING FRAMEWORK
An opensource collection of tools for building web applications in Java.

STATIC FIELD
A field that will be held in common by all instances of a class.

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

WEAKREFERENCE
References which do not prevent their referents from being made finalizable, finalized, and then reclaimed by memory.