Java™ magazine

By and for the Java community

COMMENTS AS DESIGN ELEMENTS 05 | VISITOR DESIGN PATTERN 66 | QUIZ 76

SEPTEMBER/OCTOBER 2018

Working on the Leading Edge

17 GRAALVM: COMPILING TO NATIVE BINARIES

26 JAKARTA EE: THE NEW ENTERPRISE JAVA

37 CREATING RELIABLE MICROSERVICES EASILY WITH NETFLIX HYSTRIX

54 JAVA ON ARM: EASY PORTING TO ARM-BASED SERVERS

ORACLE®

Rogue Wave SOFTWARE | JRebel

Visit us at:

ORACLE CODE

Booth 5218

Would you rather be waiting... or coding?

41.1% of developers view **waiting for others** as a major bottleneck

35.1% say **subpar tools** hold them back

Productivity bottlenecks lead to tradeoffs
in quality, performance & security

Download the ebook!

roguewave.com/java-bestpractices

Top 5 best practices for streamlining Java development

//table of contents/

GraalVM: THE POLYGLOT VM AND JVM

By Oleg Šelajev

Easily combine languages in one project and benefit from ahead-of-time compilation.

COVER FEATURES

26
JAKARTA EE: BUILDING MICROSERVICES WITH JAVA EE's SUCCESSOR
By Josh Juneau
A first look at using the emerging enterprise Java release for building microservices

37
BUILDING RESILIENCE INTO MICROSERVICES WITH HYSTRIX
By Henry Naftulin
This easy-to-use library from Netflix handles delays and failures in distributed applications.

54
JAVA ON ARM PROCESSORS
By Aleksei Voitylov
Arm already is a leading architecture for IoT and embedded processors. Recent 64-bit releases are pushing Arm CPUs onto servers, where they have full JDK support.

OTHER FEATURES

66
The Visitor Design Pattern in Depth
By Ian Darwin
Perform one or more operations on different data types without disrupting existing code.

76
Fix This
By Simon Roberts and Mikalai Zaikin
More intermediate and advanced test questions

DEPARTMENTS

05
From the Editor
Using comments to design classes

08
Java Books
Review of *A Philosophy of Software Design*

10
Events
Upcoming Java conferences and events

14
User Groups
Instanbul JUG

91
Contact Us
Have a comment? Suggestion? Want to submit an article proposal? Here's how.

COVER ART BY WES ROWELL

EDITORIAL

Editor in Chief
Andrew Binstock
Senior Managing Editor
Leslie Steere
Copy Editors
Lea Anne Bantsari, Karen Perkins
Contributing Editors
Simon Roberts, Mikalai Zaikin
Technical Reviewer
Stephen Chin

DESIGN

Senior Creative Director
Francisco G Delgadillo
Design Director
Richard Merchán
Senior Designer
Arianna Pucherelli
Designer
Jaime Ferrand
Senior Publication Designer
Sheila Brennan
Production Designer
Kathy Cygnarowicz

PUBLISHING

Group Publisher
Karin Kinnear
Audience Development Manager
Jennifer Kurtz

ADVERTISING SALES

Tom Cometa
Mailing-List Rentals
Contact your sales representative.

RESOURCES

Oracle Products
+1.800.367.8674 (US/Canada)
Oracle Services
+1.888.283.0591 (US)

ARTICLE SUBMISSION

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

SUBSCRIPTION INFORMATION

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

MAGAZINE CUSTOMER SERVICE

java@omeda.com

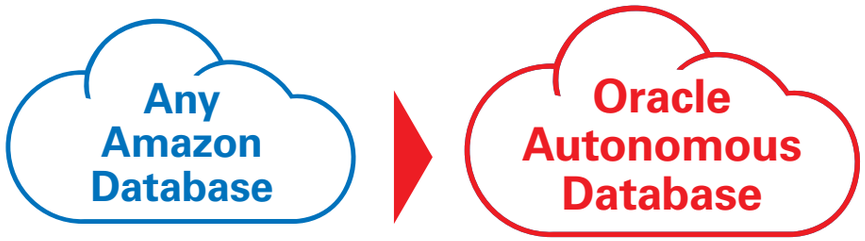
PRIVACY

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

Copyright © 2018, Oracle and/or its affiliates. All Rights Reserved. No part of this publication may be reprinted or otherwise reproduced without permission from the editors. *JAVA MAGAZINE* IS PROVIDED ON AN "AS IS" BASIS. ORACLE EXPRESSLY DISCLAIMS ALL WARRANTIES, WHETHER EXPRESS OR IMPLIED. IN NO EVENT SHALL ORACLE BE LIABLE FOR ANY DAMAGES OF ANY KIND ARISING FROM YOUR USE OF OR RELIANCE ON ANY INFORMATION PROVIDED HEREIN. Opinions expressed by authors, editors, and interviewees—even if they are Oracle employees—do not necessarily reflect the views of Oracle. The information is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle. Oracle and Java are registered trademarks of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.

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

Cut Your Amazon Bill in Half



Easy to Move—Guaranteed Savings

There's the cloud… and there's the Oracle Autonomous Cloud.

#thinkautonomous

ORACLE®

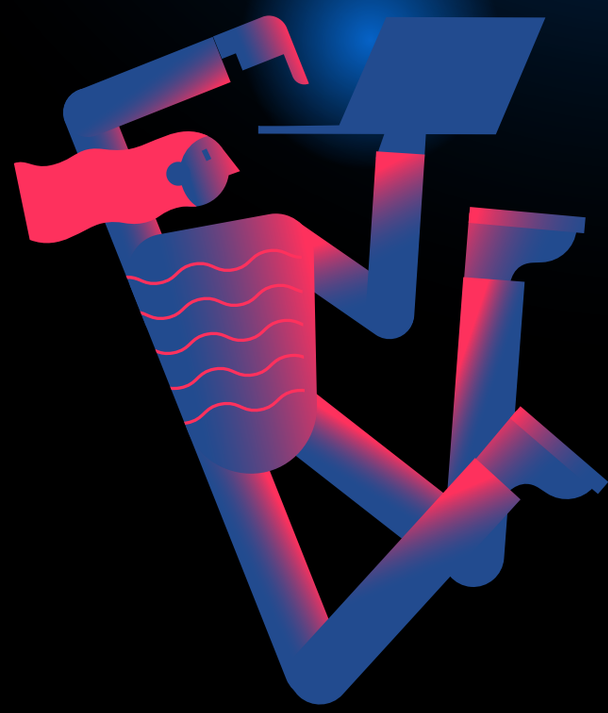
oracle.com/thinkautonomous

Guarantee applies to same data warehouse or transaction processing workload run on Oracle Autonomous Database Cloud and AWS. Pricing based on standard published pricing for Oracle bring your own license and AWS as of 3/1/18. Workload comparison to be based on actual required number of OCPU/VCPUs, storage amount, and time required to complete workload with minimum workload of one hour.
Any credits due will be applied to Universal Credit cloud account. Offer valid through 5/31/19. Copyright © 2018, Oracle and/or its affiliates. All rights reserved.
Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

IntelliJ IDEA

Capable and Ergonomic IDE for JVM

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



Download at jetbrains.com/idea

THE DRIVE
TO DEVELOP

JET
BRAINS

Using Comments to Design Classes

What is the first thing you write when creating a new class?

Like most developers, I believe, I create classes of two distinct types: the ones where my head is full of contextual information and I need to bang out a class that meets my immediate needs, and the ones that I patiently and carefully design. The former constitute the majority of my classes, alas, and they are initially sloppy beasts: I hit a keystroke combo and the IDE creates a new class with the copyright block in place, I give it a name, and I immediately start hacking. (If I'm refactoring code, then the IDE will either pull in the code from another class or generate new code for me. I love today's tools!) I can go for a while putting into the code all the things teeming in my brain without fear that I am creating an unsightly tangle. My lack of fear is because I know I will refactor the code, write the necessary tests, compile it, finish cleaning it up, run the tests, and move on.

But I'm increasingly coming to the opinion that this series of steps, which is familiar to every developer, creates a lot of unnecessary activity. A better approach is to write down all the same data in words rather than code. Suppose instead of code, I wrote the following comments:

> *This class validates a ticket number by computing a pair of check digits. Since we acquired [company name], the check-digit algorithm varies by vendor, so first look up the vendor #. The constructor accepts the ticket number, and the principal method returns an enum indicating a valid number or the type of error. All exceptions are caught and converted to error enums. This method is called only by [class name] and so should be private.*

PHOTOGRAPH BY BOB ADLER/STUDIO AT GETTY IMAGES FOR ORACLE

ORACLE

Java in the Cloud

Oracle Cloud delivers high-performance and battle-tested platform and infrastructure services for the most demanding Java apps.

Oracle Cloud.
Built for modern app dev.
Built for you.

Start here:
developer.oracle.com

#developersrule

Now, I've captured what I wanted to put into code and my most immediate problem is coming up with a good name for the class. With the key details captured, I can develop it at leisure and write good code that does not need a lot of refactoring. If I'm a TDD zealot, I can start writing a test. Either way, I'm good to go.

An excellent new book called *A Philosophy of Software Design* (which I review on page 8) advocates using comments as a design step for classes. The book is written by John Ousterhout, who is the creator of Tcl and Tk and who teaches at Stanford University (and earlier at Berkeley) when not working at his company, which specializes in large-project continuous delivery tools. Ousterhout suggests when creating a class that you use the following steps, which are a significant enlargement of what I've described above:

1. Write the class interface comment.
2. Write the interface comments and signatures of the most important public methods, but leave the method bodies empty.
3. Write comments and declarations for the most important instance variables.
4. Fill in the bodies of the methods, adding implementation comments as you go along.
5. As you discover the need for more methods, write the comments before the body.

According to Ousterhout's experience, the benefits are threefold: When the code is done, it's properly commented and the comments are entirely up to date. The comment-first approach enables you to focus on the abstractions rather than being distracted by the implementation. The code will reveal complexity—if a method or variable requires a long, complex comment, it probably needs to be rethought and simplified. That's a lot of benefits!

Of the things on which to comment, the most important in Ousterhout's view are abstractions (which are difficult to tease out from reading the implementation code) and the reason why the code exists. In sum, a developer working on your code for the first time should be able to scan the class's comments and have a good idea of what it does and an overview of the most important implementation aspects.

If this approach appeals to you—as it does to me—he suggests that you should use it until you're accustomed to writing code this way. He believes, and I agree, that doing so will convert you—by delivering cleaner, clearer code that is fully commented.

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

PS: Many open source projects wish they had more contributors, but their code is often so poorly commented and devoid of documentation that it's impossible for potential contributors to climb on board. Ousterhout's approach would go a long way toward addressing that problem.

ORACLE

The Best Resource for Modern Cloud Dev

The Oracle Developer Gateway is the best place to jump-start your modern cloud development skills with free trials, downloads, tutorials, documentation, and more.

Trials. Downloads. Tutorials. Start here:
developer.oracle.com

developer.oracle.com

#developersrule

ORACLE CODE ONE

Oct. 22–25, 2018 | San Francisco | #CodeOne

ORACLE CODE ONE
Code the Future, Together

- Discover the Latest on Java—from the Source
- Be a Part of Technologies that are Changing Everything
- Connect with Your Global Community

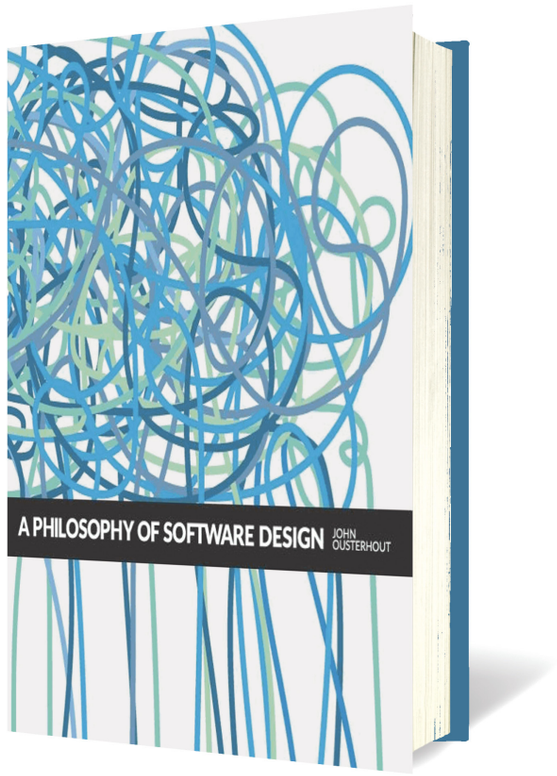
REGISTER NOW

ORACLE®

// Silver Sponsor

redhat.

*Discount based on the onsite registration price. Copyright © 2018, Oracle and/or its affiliates. All rights reserved. Oracle and Java are registered trademarks of Oracle and/or its affiliates.



A PHILOSOPHY OF SOFTWARE DESIGN

By John Ousterhout

There are at once too many and too few books on software design—particularly, software design in the small. By "in the small," I mean design at the level of implementation and coding, rather than at the architectural level. There are numerous books written by consultants that offer advice primarily designed to inveigle the reader in the author's preferred approach. But truly good books in this area are rare. Perhaps the most recent is *Growing Object-Oriented Software, Guided by Tests*, by Steve Freeman and Nat Pryce. It dates from 2010. Earlier, in 2007, there was Kent Beck's woefully underappreciated *Implementation Patterns*.

Comes now this thin, inexpensive volume from John Ousterhout, which addresses many of the class-level coding and design issues that confront us daily. Ousterhout is the inventor of the first widely adopted scripting language, Tcl, and the founder of Electric Cloud, a company that specializes in continuous delivery tools for sites that work on huge projects (in millions of lines of code). In between, he's been a professor of computer science at Stanford University and the University of California, Berkeley. It's safe to say he understands design, especially of large projects.

This book is a series of key principles aimed at reducing complexity and grounded in unwavering pragmatism. There is no overarching philosophy that Ousterhout is trying to convince you of (despite the book's title). Instead, he treats topics such as how to reduce innate complexity, how to code modules to contain and hide the complexity, how to think about abstractions when coding, and so on.

A pair of chapters, comprising some 25 pages, is devoted to comments—which might seem a lengthy diversion until you realize that they are a cornerstone to the other practices. Ousterhout recommends that the first thing you do when creating a class is to write comments, which he segregates into two types: high-level and low-level. High-level comments address what the class is about and its relationship to other classes. Reading these comments, you should not need to read anything else unless you need to know the implementation. The low-level comments are the ones that you sprinkle throughout the implementation to illuminate the why and how—only if it is not clear from the code itself. Comments, he points out, should lower the cognitive load and make code clearer. He then presents handy, pragmatic conventions for commenting.

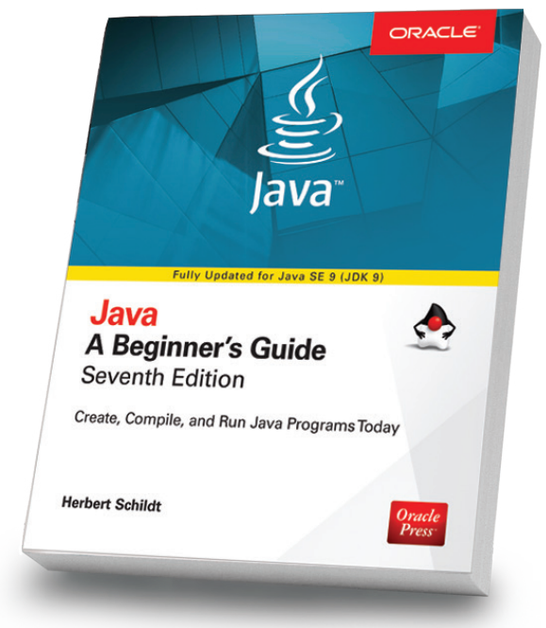
He also addresses module cohesion, code repetition, error handling, testing, and even performance tuning—always with an eye toward his central goal: reducing complexity, both intrinsic and incidental. At 170 pages, this book is an easy read and brimming with original ideas. Most code examples are in Java. Highly recommended.

—Andrew Binstock

Oracle Press™

Your Destination for Oracle and Java Expertise

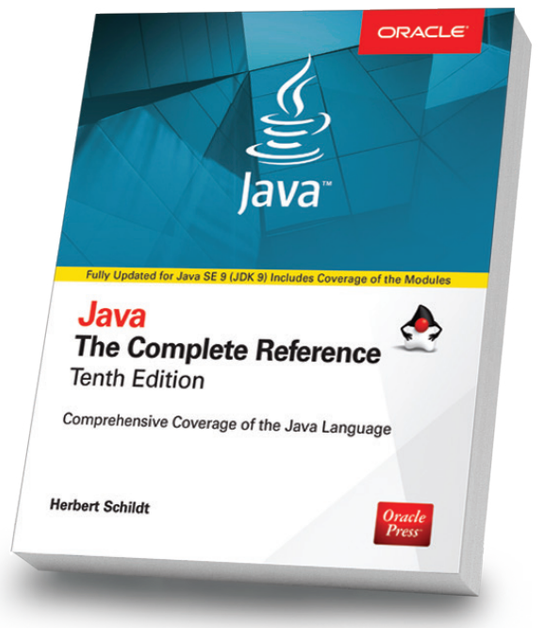
Written by leading experts in Java, Oracle Press books offer the most definitive, complete, and up-to-date coverage of Java available.



Java: A Beginner's Guide, 7th Edition

Herb Schildt

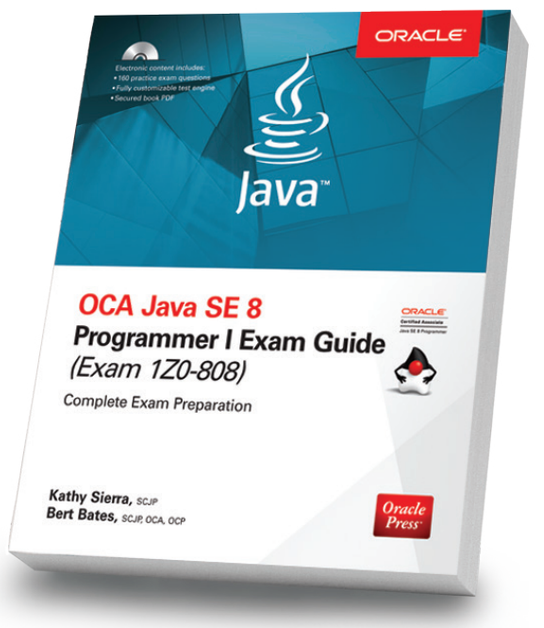
Revised to cover Java SE 9, this book gets you started programming in Java right away. Free online supplement covering key new features in JDK 10 available for download on the book's page on OraclePressBooks.com



Java: The Complete Reference, 10th Edition

Herb Schildt

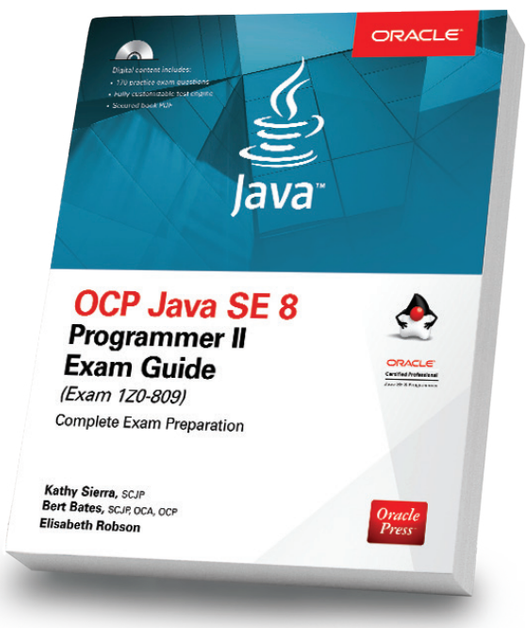
Updated for Java SE 9, this book shows how to develop, compile, debug, and run Java programs. Visit the book's page on OraclePressBooks.com to download free supplements on JDK's key new features.



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

Kathy Sierra, Bert Bates

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



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

Kathy Sierra, Bert Bates, Elisabeth Robson

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

Available in print and eBook formats.

Oracle Press™

www.OraclePressBooks.com • @OraclePress

//events/

Oracle Code One

OCTOBER 22–25

SAN FRANCISCO, CALIFORNIA

The annual JavaOne event has been remade as Oracle Code One, a new developer conference that includes more languages, technologies, and developer communities. Look for talks on Go, Rust, Python, JavaScript, and R, along with the great Java technical content that developers expect. Topics will include microservices, containers, AI, chatbots, blockchain, and databases. A Java keynote and community keynote will remain, and all of the Java-focused community activities are being carried forward including IGNITE sessions, Community Day (now as a track), Java Champion briefings, and Duke's Choice Awards. (Note that registration is to Oracle OpenWorld, which is collocated with Oracle Code One.)

PHOTOGRAPH BY ELIZABETH K. JOSEPH/FLICKR

JSCamp

SEPTEMBER 22

CHICAGO, ILLINOIS

JSCamp is a one-day single-track conference for developers interested in JavaScript. Speakers will cover topics such as Node, React, Vue, and Ember for both beginners and experts.

jDays

SEPTEMBER 25

GOTHENBURG, SWEDEN

jDays brings together software engineers from around the world to share their experiences in different areas such as Java, software engineering, IoT, digital trends, testing, agile methodologies, and security.

Strange Loop

SEPTEMBER 26–28

ST. LOUIS, MISSOURI

Strange Loop is a multidisciplinary conference that brings together the developers and thinkers building tomorrow's technology in fields such as emerging languages, alternative databases, concurrency, distributed systems, and security. Talks are generally code-heavy and not process-oriented.

Functions

SEPTEMBER 28

TORONTO, CANADA

Functions is a community-focused, single-track conference exploring serverless development and architecture. Past conference speakers have included JS Foundation's Darcy Clarke and Google's Sandeep Dinesh.

NFJS New England Software Symposium

SEPTEMBER 28–30

FRAMINGHAM, MASSACHUSETTS

This developer event covers the latest trends within the Java and JVM ecosystem. Scheduled are talks on Java 9, reactive APIs, and microservices. Team attendance is encouraged.

Emerging Technology Conference

OCTOBER 2–3

SAINT PAUL, MINNESOTA

This conference explores the emerging technologies that are shaping the world, including machine learning, blockchain, and virtual/augmented reality.

//events/

DevOpsDays Detroit
OCTOBER 3–4
DETROIT, MICHIGAN
DevOpsDays is a worldwide community conference series for anyone interested in information-technology improvement. This year marks its fourth annual iteration in Motor City.

KotlinConf
OCTOBER 3, WORKSHOPS
OCTOBER 4–5, CONFERENCE
AMSTERDAM, THE NETHERLANDS
This is the principal conference for the popular JVM language, Kotlin. Keynotes by Kotlin Project Lead Andrey Breslav and Purple Evolution CEO Alicia Carr are slated.

Desert Code Camp
OCTOBER 6
CHANDLER, ARIZONA
Desert Code Camp is a free, developer-based conference built on community content. This year's sessions include talks on serverless microservices and building a website with Angular.

JAX London
OCTOBER 8 AND 11, WORKSHOPS
OCTOBER 9–10, CONFERENCE
LONDON, ENGLAND
JAX London is a four-day conference for software engineers and enterprise-level professionals, bringing together the world's leading innovators in the fields of Java, microservices, continuous delivery, and DevOps. Topics slated for this year include delivering new features in the JDK, developing Java applications on blockchain with web3j, and cloud-native Java with OpenJ9.

JCON
OCTOBER 9–11, CONFERENCE
OCTOBER 12, TRAINING
DÜSSELDORF, GERMANY
JCON is for professional Java developers interested in core Java, enterprise Java, frameworks, and microservices. Daily focus starts with cloud and DevOps, moves to big data, and finishes with architecture and agile. A parallel XDEVCON conference focuses on rapid cross-platform development.

re:develop
OCTOBER 12
BOURNEMOUTH, ENGLAND
This platform-agnostic developer conference returns after a one-year hiatus and will address key new methodologies and practical advice for approaching development projects more effectively.

JFuture
OCTOBER 13
MINSK, BELARUS
This event gathers Java developers, software engineers, and technology enthusiasts and will focus on major updates of Java and popular Java frameworks. Modularity, Spring, Kotlin, and Rust are slated topics.

Java Enterprise Summit
OCTOBER 17–19
DÜSSELDORF, GERMANY
The Java Enterprise Summit is a Java EE training event exploring new paradigms such as microservices, API design, and state-of-the-art enterprise Java applications. (Website in German.)

All Things Open
OCTOBER 21–23
RALEIGH, NORTH CAROLINA
This annual conference explores open source, open tech, and the open web in the enterprise. Speakers this year include Netflix senior performance architect Brendan Gregg and AWS principal technologist Alolita Sharma. The theme this year is open source software and future disruption.

PHOTOGRAPH BY PAUL.WASNESKI/FLICKR

//events/

Open Source Summit Europe
OCTOBER 22–24
EDINBURGH, SCOTLAND
More than 2,000 attendees are expected for this annual gathering of developers, architects, and open source community and industry leaders.

EclipseCon Europe 2018
OCTOBER 22, COMMUNITY DAY
OCTOBER 23–25, CONFERENCE
LUDWIGSBURG, GERMANY
The Eclipse event for the European community will host presentations on Jakarta EE, Microprofile, and many other Java technologies. An OSGi community event is collocated with this conference.

Lambda World
OCTOBER 25–26
CÁDIZ, SPAIN
Billed as one of the largest functional programming events in Europe, this conference will cover languages such as Clojure, Scala, Kotlin, F#, and Java.

O'Reilly Software Architecture Conference
OCTOBER 29–31, CONFERENCE AND TUTORIALS
OCTOBER 31–NOVEMBER 1, TRAINING
LONDON, ENGLAND
For four days, expert practitioners share new techniques and approaches, proven best practices, and technical skills. Topics include application, microservices, event-driven, and evolutionary architectures.

Voxxed Days Microservices
OCTOBER 29–30, CONFERENCE
OCTOBER 31, WORKSHOPS
PARIS, FRANCE
Learn—and share—everything you need to know about microservices at Voxxed Days Microservices, a new event with two days of sessions and a day of workshops. Session tracks include organization and culture, architecture, testing, scaling, and integration.

QCon San Francisco
NOVEMBER 5–7, CONFERENCE
NOVEMBER 8–9, WORKSHOPS
SAN FRANCISCO, CALIFORNIA
QCon San Francisco is an international conference for professional software developers. Topics this year include microservices, next-gen architecture, 21st-century languages, JavaScript and web tech, and enterprise languages including Java.

W-JAX
NOVEMBER 5–9, CONFERENCE
NOVEMBER 6–8, EXPO
MUNICH, GERMANY
W-JAX is a conference dedicated to cutting-edge Java and web development, software architecture, and innovative infrastructures. Experts share their professional experiences in sessions and workshops. This year's event promises more than 160 speakers and more than 180 workshops, sessions, and keynotes.

DeveloperWeek Austin
NOVEMBER 6–8
AUSTIN, TEXAS
DeveloperWeek Austin will feature tracks devoted to JavaScript, virtual reality development, microservices, and AI development; a Hiring Mixer Expo; and two days of hackathons.

J-Fall
NOVEMBER 8
EDE, THE NETHERLANDS
J-Fall is organized by and for the Dutch Java community. With 1,500 Java professionals attending, J-Fall is the biggest Java conference of the Netherlands, boasting more than 45 sessions and four hands-on labs, more than 70 top speakers, and a preconference day with in-depth workshops and the Masters of Java contest. The lineup will be revealed soon. Last year's speakers included Apache Maven Project Chair Robert Scholte, Google Developer Advocate Ray Tsang, and AWS Senior Solutions Architect Brian Hammons.

Devoxx Belgium
NOVEMBER 12–16
ANTWERP, BELGIUM
The largest Java developer conference in Europe takes place again this year in Antwerp, Belgium, with multiple tracks covering Java, the mechanics of the JVM, and JVM languages. The event is held in a multiplex theater with code and slides shown on giant movie screens.

PHOTOGRAPH BY UWE BRODRECHT/FLICKR

KubeCon + CloudNativeCon

NOVEMBER 13–15
SHANGHAI, CHINA

This conference gathers leading technologists from open source cloud-native communities to further the advancement of cloud-native computing. Simultaneous Mandarin-English translation will be provided for all keynotes and sessions.

Codemotion Berlin

NOVEMBER 20–21
BERLIN, GERMANY

Codemotion conferences are devoted to developers sharing the latest tech information and best practices among the tech community worldwide. Confirmed speakers at this event include Picnic CTO Daniel Gebler and Apache Software Foundation member Kanchana Welagedara. The event is open to all languages and technologies and features coding lectures and workshops.

Topconf Tallinn

NOVEMBER 20–22
TALLINN, ESTONIA

Topconf Tallinn is an international software conference covering Java, open source, agile development, architecture, and new languages.

JVM-Con

NOVEMBER 27–28
COLOGNE, GERMANY

Among the topics slated for this conference devoted to JVM languages are the JRE, Java 9, Java EE 8, and cloud-native development. (Website in German.)

Codemotion Milan

NOVEMBER 29–30
MILAN, ITALY

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

DevTernity

NOVEMBER 30–DECEMBER 1
RIGA, LATVIA

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

ArchConf

DECEMBER 10–13
CLEARWATER, FLORIDA

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

CodeMash 2019

JANUARY 8–11, 2019
SANDUSKY, OHIO

CodeMash is an event that educates developers on current practices, methodologies, and technology trends in a variety of platforms and development languages such as Java, .NET, Ruby, Python, and PHP. The Java track features participation from many Java Champions.

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

ISTANBUL JUG

The Istanbul Java User Group, JUG Istanbul, was founded in March 2006. Since then, the community has been organizing monthly meetups as well as an annual conference called Java Day Istanbul, in which Java Champions, JUG leaders, and developers from around the world participate. This year's conference took place in Istanbul in May 2018 at the Hilton Bosphorus Istanbul and included trending topics such as effective Java, microservices, serverless architectures, DevOps, and Java EE microprofile.

The community created a green-field open source project named Second Opinion Doctor, a crowd-sourcing platform for medical professionals. The project began in November 2017 with a co-op hackathon. The project is still ongoing, and other open source communities have been participating in it. JUG Istanbul joined the Eclipse Foundation in May 2018.

Today, more than 3,000 JUG members participate in Turkey's Java community. For communication outside meetings, JUG Istanbul uses Facebook, Twitter (@jug_istanbul), Slack (jugistanbul.slack.com), and meetups. Email JUG Istanbul in advance if you are a member of another JUG, a Java Champion, or a technology evangelist coming to Turkey so that the group can host you and arrange some conversations.

Join the World's Largest Developer Community

Download the latest software, tools, and developer templates

Get exclusive access to hands-on trainings and workshops

Grow your network with the Developer Champion and Oracle ACE Programs

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

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

@OracleDevs **facebook.com/OracleDevs**

ORACLE®

Copyright © 2018, Oracle and/or its affiliates. All rights reserved. Oracle and Java are registered trademarks of Oracle and/or its affiliates.

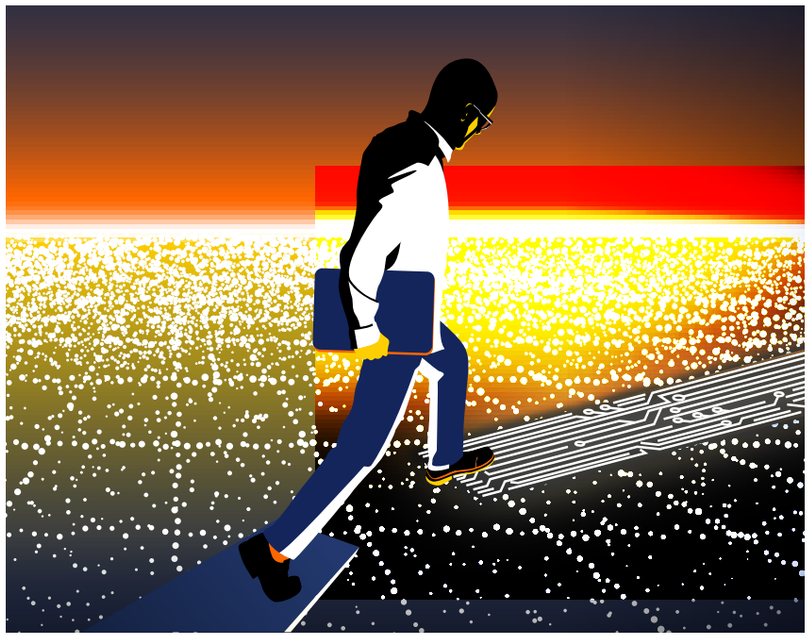
VOXXED DAYS

GRAALVM 17
JAKARTA EE 26
NETFLIX HYSTRIX 37
JAVA ON ARM 54

On the Leading Edge of Java Development

Few programming languages make it past 20 years and still retain high popularity. But Java has managed to remain widely used in many contexts because of its evolution—not just of the language, but of the larger direction of the ecosystem. None of this is clearer than in the advent of GraalVM, an ahead-of-time native compiler for Java code that coincidentally is written in Java and supports many other languages—both JVM-based and native. To understand how to use Graal for your own projects, see our article on page 17.

In the enterprise, Java EE has moved out from Oracle's aegis and is now hosted at the Eclipse Foundation under the name of Jakarta EE, which we examine in detail (page 26) in anticipation of its upcoming 1.0 release.



ART BY WES ROWELL

Much of Java's success comes from new tools contributed as open source. Hystrix from Netflix is an excellent library for assuring uptime in distributed apps—especially microservices. Our coverage of Hystrix (page 37) shows its benefits and elegance of implementation.

Finally, we look at one of the most exciting platform developments: running Java apps on power-sipping ARM processors. As our article (page 54) demonstrates, migration of existing code is not difficult, and on recent chip releases, it does not entail a compromise on performance.

We also include the next installment (page 66) of our series on design patterns, this time covering the Visitor pattern. And of course, this issue includes our usual quiz (page 76), editorial (page 5), and a book review (page 8) of an unusually interesting volume. Enjoy!

OLEG ŠELAJEV

GraalVM: The Polyglot VM and JVM

Easily combine languages in one project and benefit from ahead-of-time compilation.

GraalVM is a high-performance embeddable polyglot virtual machine capable of running different programming languages, for example:

- JVM-based languages, such as Java, Scala, Kotlin, and Groovy
- Interpreted languages, such as JavaScript, Ruby, R, and Python
- Native languages that work with LLVM, such as C, C++, Rust, and Swift

GraalVM efficiently supports polyglot apps, allowing you to mix languages in a single process without incurring significant performance overhead—thereby making it possible to include the most suitable solution to the problem at hand.

GraalVM is designed to execute programs in different environments: in the JVM, compiled into a standalone native image, or embedded into larger applications that include both Java and native code modules. In this article, I present a quick overview of what GraalVM is capable of, how to start using it for everyday tasks, and which parts of the project should you focus on.

Components of GraalVM

GraalVM is a large project with several moving parts that enable it to be as versatile as it is. Just to give you a taste, GraalVM can run Java code really fast, run Node.js applications and be a replacement for your Nashorn scripts, run Ruby and Python, and run R. It can compile some Java applications into executable native images that take up just a few megabytes for use in a Docker container and start in milliseconds. It also can execute JavaScript code as stored procedures inside databases without greedily taking over resources your database expects to have for itself.

GraalVM is open source and largely written in Java, so to get acquainted with the project you don't need to be an expert in the native code. This aspect also means that you can use the tooling you use for everyday Java development to explore and develop GraalVM. Now, let's look briefly at some GraalVM components.

Graal, the JIT compiler. Without an outstanding just-in-time (JIT) compiler, it's really hard to produce a high-performance virtual machine. So at the heart of GraalVM, you find a compiler called Graal. Graal can be used both as a JIT compiler and as a static, ahead-of-time compiler.

Just like the other components of GraalVM, Graal is written in Java. It implements several typical compiler optimizations: common subexpression elimination, dead-code elimination, constant folding, and so on. However, its inlining and escape analysis algorithms are where it really shines. Graal is an aggressively optimizing compiler that uses an internal representation (IR) that bears resemblance to the program dependence graph the C2 compiler in the Java HotSpot VM uses, although it differs in important ways. During the compilation, the IR is transformed from representing high-level operations (such as load Java field) to low-level operations (for example, read at address+offset). This low-level representation is eventually translated into the machine code. In addition to standard JVM options that can assist you with analyzing the JIT compilation (such as `-XX:+PrintCompilation`, `XX:+PrintAssembly`, and so on), the GraalVM distribution includes a utility called the Ideal Graph Visualizer, which you can use to debug and analyze these graph transformations.

Truffle. The next major component of the GraalVM project is Truffle, which is a framework for implementing programming languages. Truffle offers an API that you can use to implement an interpreter of a language based on the abstract syntax trees (ASTs) of the source programs. Evaluating ASTs is a relatively straightforward way to execute a program, so working on an interpreter is much easier than creating an optimizing compiler. But Truffle, with the help of the Graal compiler, can optimize these interpreters so their peak performance is on par and sometimes better than the code produced by conventional compilers.

To compile the interpreters for the implemented language, Truffle uses a technique called *partial evaluation*. In a nutshell, this means Truffle takes a language interpreter and a program,

and produces a specialized version of the interpreter specifically for the given program. It does this by attaching to the tree nodes the profiling and type information gathered at execution time. Then Truffle can speculatively optimize the program by using this profile. Truffle needs a runtime with a compiler aware of the partial evaluation, and Graal fits this requirement nicely.

There are plenty of examples of Truffle-based languages from which to take inspiration. You can look at the JavaScript engine, an LLVM bitcode interpreter, a Ruby implementation, a Python implementation, and an R implementation—and these are just the official projects by the GraalVM team. You can find several other implementations of programming languages on the GitHub page. There is even a demo programming language that was created to demonstrate and teach the features of Truffle, which can get you started if you want to experiment.

The best part of Truffle languages is that at runtime, all the interpreters are using the same interoperability protocol for objects in different programming languages, which means that from the runtime's point of view, there is no difference whether the program be written in JavaScript, Python, Ruby, any other implemented language, or a mix of those. The runtime is able to optimize polyglot programs written in different languages the same way it optimizes code normally—without any performance overhead for crossing the language barrier. This aspect opens the door to using libraries and modules from all ecosystems and truly picking the best programming language to focus on solving the problems you need to solve, rather than trying to reimplement the missing piece of functionality in the language of your project.

All the programs that run on the JDK are supported by GraalVM.

Another benefit of Truffle is that it virtualizes the language implementation, so from the runtime point of view, all languages look similar. This is an excellent opportunity for the tooling to become polyglot as well. For example, you can use the JavaScript debugger to step through a Ruby program or use VisualVM to analyze the memory use of a JavaScript program just as you'd normally use them for JavaScript and Java programs.

Native images. GraalVM has additional features, such as the SubstrateVM, which is a small virtual machine written in Java that can be used to compile some Java applications to native executable images. GraalVM native images do not require the JVM to run, don't need to load and initialize Java classes, and so on—so they have very fast startup. During generation of the native images, the Graal compiler analyzes the classes of your applications and compiles them to machine code ahead of time. SubstrateVM provides the services any virtual machine would: garbage collection, thread scheduling, code caches, etcetera. And its code can also be compiled ahead of time by Graal. The result is an executable that doesn't have the peak performance of fully warmed up JIT-compiled code, but has decent performance, has low runtime overhead, and starts up in milliseconds. In some production environments, such as the cloud or serverless deployments, startup is more important than the peak performance of the long-running code.

Additionally, it is possible to embed GraalVM in other runtime platforms, extending them with the polyglot capabilities. Currently, there are experimental builds of Oracle Database that embed GraalVM and enable you to write stored procedures in JavaScript instead of in PL/SQL. Similar functionality is available as a MySQL plugin, so you can use GraalVM in the MySQL database as well. At first sight, this might seem like a superficial capability, but this design gives you the opportunity to use the programming languages you already know and the existing ecosystem of modules and libraries.

Getting Started with GraalVM

There are several ways to try GraalVM or its parts, depending on how much effort you want to invest.

Of course, you can build GraalVM from the codebase because, as I mentioned earlier, it is an open source project under the GPL2 with the classpath exception license—the same license as that of OpenJDK. However, the easiest way to evaluate GraalVM is to download the prebuilt binaries.

The distribution you get is similar to the JDK, but it also bundles the JavaScript engine and an implementation of Node.js, the LLVM bitcode interpreter, and the native image utilities.

Download a GraalVM distribution, and unzip the archive into a `$GRAALVM_HOME` directory. Then you can run `java` from GraalVM:

```
> $GRAALVM_HOME/bin/java -version
java version "1.8.0_172"
Java(TM) SE Runtime Environment (build 1.8.0_172-b11)
GraalVM 1.0.0-rc3 (build 25.71-b01-internal-jvmci-0.45, mixed mode)
```

Or you can run JavaScript programs—for example, evaluating the following one-liner:

```
> $GRAALVM_HOME/bin/js -e 'console.log(1+2)'
3
```

You can install the experimental support for Ruby, R, and Python by using the `gu` utility on the command line:

```
$GRAALVM_HOME/bin/gu install {ruby|python|r}
```

Then the command-line launchers for the languages you install will also be available under the GraalVM directory—for example:

```
> $GRAALVM_HOME/bin/ruby -e 'puts 1 + 2'
3
```

Note that the prebuilt GraalVM distributions are based on OpenJDK. So all the programs that run on the JDK are supported by GraalVM. If you want to compare performance on an apples-to-apples basis, you can disable the Graal compiler in the GraalVM distribution by using the `-XX:-UseJVMCICompiler` switch and then just use the Java HotSpot VM compiler that OpenJDK uses normally.

You can follow the getting started guide to get acquainted with what GraalVM can do for you, try all the experiments in the article "Top 10 Things to Do with GraalVM," or try some examples gathered by the GraalVM team.

If you have a project ready to go with performance tests and all the infrastructure to measure the performance impact of GraalVM, a good starting point could also be trying out various microbenchmarks. For example, consider the following benchmark. It is implemented with the Java Microbenchmark Harness, which is the standard tool for Java microbenchmarks. The benchmark method executes a simple series of Stream API method calls, manipulating the numbers in a stream before adding everything up.

```
package org.graalvm.demos;

import org.openjdk.jmh.annotations.*;
import java.util.Arrays;
import java.util.concurrent.TimeUnit;

@Warmup(iterations = 1)
@Measurement(iterations = 3)
@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.NANOSECONDS)
@Fork(1)
public class JavaSimpleStreamBenchmark {

  static int[] values = new int[]{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

  @Benchmark
  public int testMethod() {
   return Arrays.stream(values)
     .map(x -> x + 1)
     .map(x -> x * 2)
```

```
    .map(x -> x + 2)
    .reduce(0, Integer::sum);
  }
}
```

On my machine, this benchmark executes several times faster on GraalVM than on OpenJDK 1.8. If you want to experiment with it, clone the repository for the benchmark:

```
git clone https://github.com/graalvm/graalvm-demos
cd graalvm-demos/java-simple-stream-benchmark
```

To see the difference for yourself, you can build and run it using `$GRAALVM_HOME/bin/java` and the normal OpenJDK:

```
mvn clean install
$GRAALVM_HOME/bin/java -jar target/benchmarks.jar
```

Naturally, this is not the most scientific assessment of the runtime performance, and you should always properly measure the performance impact yourself. But it shows that for some code, GraalVM execution is significantly faster than on the Java HotSpot VM.

Using GraalVM to Integrate Java with Other Languages

One of the most interesting features of GraalVM is its polyglot capability, so let's look at how GraalVM makes polyglot applications possible.

At the center of the polyglot API in GraalVM is the `Context` class. A Context is a representation of the global runtime state of all non-Java (compiled to the JVM bytecode) languages. You can initialize all available languages on demand and evaluate the code in the desired languages. The following snippet is the simplest example of the polyglot GraalVM application. It's normal Java code that evaluates a string of JavaScript, which declares a value of 42 and returns a `Value` object to Java.

```
Context context = Context.create();
Value result = context.eval("js", "42");
assert result.asInt() == 42;
```

`Value` is how languages communicate with each other. Any Java object can be converted to a `Value` with the `Value.asValue(Object value)` method call, and the `Value` can be converted to its Java counterpart with `Value.as(Class<T> targetType)`. The exact description of the conversion process is outside the scope of this article, but the API always tries to do the sensible thing: it converts numbers to numbers, strings to Strings, values that can be executed to functional interfaces, collections to collections, and so on. For example, all of the following expressions are true:

```
context.eval("js", "'foobar'").as(String.class).equals("foobar");
context.eval("js", "{foo:'bar'}").as(Map.class).get("foo").equals("bar");
@FunctionalInterface interface IntFunction { int f(int value); }
context.eval("js", "(function(a){a})").as(IntFunction.class).f(42) == 42;
```

Armed with the `Context` and the `Value`, you can pass data between the components written in different languages.

However, a modern application will probably hide the details of the polyglot implementation of its components behind some abstraction. For example, this demo application is a Spring Boot web app that uses R to plot CPU utilization data as an SVG image.

In the app, the GraalVM polyglot context is defined as a Spring `@Bean`:

```
@Bean
public Context getGraalVMContext() {
    return Context.newBuilder().allowAllAccess(true).build();
}
```

The function written in R that takes data and plots the result (the source of which is located in a resource file) is exposed as a Spring bean. The definition takes the GraalVM context, evaluates the R source, and returns the result as a Java `Function<Double, String>`.

```
@Bean
Function<Double, String> getPlotFunction(@Autowired Context ctx) {
   Source source =
     Source.newBuilder("R", rSource.getURL()).build();
   return ctx.eval(source).as(Function.class);
}
```

After that, there's no difference in using the R function or any other Java functional interface implementation. Similarly, you can bring other languages supported by GraalVM into your Java app.

Conclusion

In this article, I looked at the GraalVM project and its components—the Graal compiler, Truffle, and the native images utility—as well as the most important API for the polyglot programs, and I tried to provide a short guide on how you can get started with GraalVM.

Give GraalVM a try. Many Java applications run faster on GraalVM; many can benefit from instant startup; and some perhaps can be enhanced with modules written in other supported languages such as Ruby, JavaScript, R, and Python. And if you find any issues or want to participate in this exciting project, visit its Github repository. `</article>`

Oleg Šelajev (@shelajev) is a developer advocate at Oracle Labs, working on GraalVM—the high-performance embeddable polyglot virtual machine. He organizes the VirtualJUG, the online Java User Group, and a GDG chapter in Tartu, Estonia. In his spare time, he is pursuing a PhD in dynamic system updates and code evolution. He became a Java Champion in 2017.

JOSH JUNEAU

Jakarta EE: Building Microservices with Java EE's Successor

A first look at using the emerging enterprise Java release for building microservices

Developers have taken many approaches over the years to developing web and enterprise applications on the Java platform. At the inception of the JVM, the groundbreaking servlet technology was introduced in an effort to bring dynamic content to web applications, and technology such as applets provided a way to deliver rich internet applications to a user's desktop. Over time, developers created easier and more intuitive ways to work with servlets via frameworks such as JavaServer Pages (JSPs) and Apache Struts. Built on top of servlet technology, these solutions enabled developers to focus more on the front end than on boilerplate code.

The J2EE platform was introduced in 1999, providing a handful of APIs for the creation of enterprise-based applications. The APIs included JDBC, Enterprise JavaBeans (EJB), JSPs, and Java Message Service (JMS), to name a few. In the early era, J2EE was complex, because there were many configurations that needed to be made and the logic was difficult to follow. Configuration made the platform difficult to use.

As the years went on, J2EE evolved to include more APIs and the complexity level decreased. The platform name was changed in 2006 when Java EE 5 was introduced. This release was a significant, because it included some great productivity boosters such as the changes made in EJB 3.0, and the JavaServer Faces (JSF) framework was introduced to the platform. The next two releases, Java EE 6 and Java EE 7, focused primarily on developer productivity and platform modernization. Each of these releases significantly reduced the complexity of the platform by incorporating technologies such as annotations, rather than requiring XML for configuration. Contexts and Dependency Injection (CDI) was introduced to the core of the platform in Java EE 6, providing an easy way to utilize contextual objects throughout an application. Developing and deploying

modern applications within a single WAR file was then streamlined with the Java EE platform.

Although Java EE evolved significantly, rapid changes in technology placed demand on the platform to advance more quickly. The Java EE 8 release answered the call by providing APIs to facilitate using web services, a new security API, and improvements for deployment to container environments. However, the need to advance the platform at a more rapid pace remained an issue. To accommodate this requirement, Oracle open-sourced the platform through the Eclipse Foundation in 2017.

In this article, I take you through an introduction to the next evolution in the enterprise Java space: Jakarta EE. I explain the transfer of the specifications from Oracle to the Eclipse Foundation, and I demonstrate how to grab the latest code to get started with Jakarta EE.

How We Got to Jakarta EE

When Oracle decided to open source the Java EE platform via the Eclipse Foundation, the Eclipse Enterprise for Java (EE4J) project was formed. This project is a base repository that resides on GitHub, and it is in place for the purpose of transitioning and housing the codebase, documentation, and Technology Compatibility Kits (TCKs) for each of the Java EE specifications. EE4J is not going to become the open platform; rather, it is a project that contains each of the specifications for the new platform. Oracle began to transfer the documentation, codebase, and TCK for each of the specifications to their respective EE4J projects during the second half of 2016. At the time of this writing (mid-2018), the transition is still underway, and significant work is being done to make the transition as seamless and timely as possible.

The name *Jakarta EE* was chosen for the new open-sourced platform that was once known as Java EE. That is, all EE4J projects that are transferred from Oracle will be combined to create the Jakarta EE platform. (The Jakarta name has significance in the Java community, because a project known as the Jakarta Project was operated as an umbrella project under the Apache Software Foundation for several years. The project was retired in 2011, because most of the subprojects had formed independent projects within the Apache Software Foundation, so the project team felt that no confusion would arise.)

New Governance Process

A key feature of Jakarta EE is a more frequent release cadence than Java EE had. To this end, the Jakarta EE platform will be governed by the Jakarta EE Working Group. This group will help to evolve and promote broad adoption of the technologies related to EE4J. As part of its mandate, the working group will provide vendor-neutral marketing, define and manage a new governance process for formalizing specifications, define compatibility and branding rules, encourage community participation, and establish a funding model that will enable the working group and community to operate on a sustainable basis.

There are five different classes of Jakarta EE Working Group membership: Strategic, Enterprise, Participant, Committer, and Guest. The Strategic, Enterprise, and Participant classes are geared toward organizations and provide different levels of participation in the process. Committer members are individuals who are able to contribute and commit code to the Eclipse Foundation projects. Guest members are organizations that are Associate members of the Eclipse Foundation; they are invited by the steering committee for a renewable year of membership in which they participate in specific aspects. Under the Jakarta EE Working Group, members will be able to help steer the direction of the platform or contribute directly by committing patches or adding enhancements to the EE4J projects that make up the platform.

Getting Started with Jakarta EE

Let's look at how to start building applications on the Jakarta EE platform. The Jakarta EE 8 release is due prior to the end of 2018. With that in mind, some of the information discussed in the following section will be relevant only to those of you who are trying Jakarta EE 8 prior to its official release.

In the next few sections, I cover how to obtain the libraries required to develop various services that will be used together to compose an example application. The individual services in the example application are actually separate projects that use only the EE4J projects that are required. In the end, I demonstrate how to deploy each of the services to a single instance of the "full" version of Payara Server to deliver the application. This setup can be used to deploy appli-

cations that are composed of multiple WAR files to a single server. However, it is more common for each WAR file to be deployed to separate containers as microservices. Either choice is valid and depends upon the requirements of your project.

Building the Sports Roster Application

A Jakarta EE application is built on standards, and it's easy to develop. The example application will allow a sports team to register and query team members. The application I'm going to build utilizes a JSF 2.3 (Mojarra) front end, along with JAX-RS web services for communication with the Java Persistence API (JPA) that works with the database. The application is composed of three separate services, each of them Maven projects. The first project I demonstrate will be used to query the database, the second will register (insert or update to the database), and the third is a front-end user interface.

Consider that each service may be deployed to separate containers, such as Apache Tomcat, that do not bundle the full Jakarta EE stack. For that reason, I list only the dependencies in the POM that each service requires, rather than listing the entire stack. For this article, each of the services can be deployed to a single instance of GlassFish 5 or another application server or servlet container.

Setting Up the Environment

There are plenty of IDEs you can use for developing a Jakarta EE application. The main issue at the time of this writing is that none of them offers direct support for Jakarta EE, per se. However, most of them have complete support for Java EE 8. Because the initial release of Jakarta EE is aligned with Java EE 8, it is possible to make use of the IDE support for Java EE when developing Jakarta EE applications. Most of the major IDEs eventually will contain direct support for Jakarta EE.

The IDE that I use for this article is a release candidate of Apache NetBeans 9.0, built from the codebase. The Java EE plugins can be installed after the IDE has been built.

Obtaining Projects for Use Within Services

EE4J hosts the various projects that make up the Jakarta EE platform. As the Jakarta EE platform begins to have full releases, there will be Maven coordinates or a location where different variations of the Jakarta EE platform can be downloaded and added to a project. The platform will likely be available in its entirety or as separate APIs. Prior to the official release of Jakarta EE, the EE4J projects need to be downloaded or added as Maven dependencies separately. You can use a couple of paths to obtain the specifications or APIs that are required: download the codebases and build them yourself, or utilize the Maven Central repository to pull them into the project.

To download and build an EE4J API, visit its corresponding EE4J GitHub project page and find the respective GitHub repository, clone the codebase of the project, and (typically) use Maven to build the API. Follow the procedures on the respective project's GitHub repository homepage. See the Mojarra project for a good example, because it contains lots of documentation on how to build and use it in resulting projects. Most of the projects simply require the project to be built by issuing `mvn clean install`, and the dependency JAR will be produced. The resulting JAR can then be added to a local Maven repository, or it can be added directly to a project.

For the examples in this article, I use Maven Central by adding the dependencies to project POM files. Note that Maven Central might not yet have true Jakarta EE platform APIs registered within it, so some of the APIs still point to the Java EE 8 dependency.

Developing the Services

I created each of the services for the example as Maven projects in Apache NetBeans by selecting New Project > Maven > Web Application. The projects are named as follows: `SportsTeamQueryService`, `SportsTeamRegistrationService`, and `SportsTeamUIService`.

The following lines of SQL can be used to create the database table (Apache Derby) for the example. You can add the table to the sample Derby schema for convenience. If you are using GlassFish 5 or Payara Server 5 to deploy the example, the Derby sample schema should already be registered as a JDBC connection.

```
create table team_roster (
id                    numeric,
first_name            varchar(150),
last_name             varchar(150),
position              varchar(150),
registration_date     date,
PRIMARY KEY (ID)
);
```

Query service. `SportsTeamQueryService` is the most straightforward, because it involves only a few lines of code to produce—especially if you're using an IDE. **Listing 1** shows the POM for this service, because the service uses the EJB, JPA, and JAX-RS Jakarta EE specifications. (All the code—including the listings discussed but not inserted in this article—is available from my GitHub repository.)

A persistence unit is required so the service can connect to the database (**Listing 2**). The service uses an entity class for persistence. To create the `TeamRoster` entity class using Apache NetBeans, select New > Entity Classes from Database, and then select the database table that was created in the previous section. **Listing 3** shows the codebase for the `TeamRoster` entity class.

A JAX-RS service can be used to query the database. To enable JAX-RS within the service, create an `ApplicationConfig` class with the contents of **Listing 4**.

Listing 4.

```
package org.javamagazine.sportsteamqueryservice.service;

import java.util.Set;
import javax.ws.rs.core.Application;

@javax.ws.rs.ApplicationPath("rest")
public class ApplicationConfig extends Application {
```

```
@Override
public Set<Class<?>> getClasses() {
  Set<Class<?>> resources =
    new java.util.HashSet<>();
  addRestResourceClasses(resources);
  return resources;
}

private void addRestResourceClasses(
Set<Class<?>> resources) {
  resources.add(
    org.javamagazine.sportsteamqueryservice
    .service.TeamRosterFacadeREST.class
  );
}
```

The `@ApplicationPath` annotation is used to define the URL path to use as an entry point for each of the service's RESTful web services. All resource classes that are used as a RESTful service must be registered within the `ApplicationConfig`.

The REST service class is named `TeamRosterFacadeREST`, as shown in **Listing 5**.

Listing 5.

```
@javax.ejb.Stateless
@Path("teamrosterqueryservice")
public class TeamRosterFacadeREST  {

  @PersistenceContext(unitName =
    "SportsTeamQueryServicePU")
  private EntityManager em;
```

```
public TeamRosterFacadeREST() {
}

@GET
@Path("{id}")
@Produces({MediaType.APPLICATION_XML,
      MediaType.APPLICATION_JSON})
public TeamRoster find(
  @PathParam("id") BigDecimal id) {
    return (TeamRoster)
    em.createQuery(
       "select object(o) from TeamRoster o " + "where o.id = :id")
      .setParameter("id", id)
      .setHint("javax.persistence.cache.retrieveMode",
           CacheRetrieveMode.BYPASS)
      .getSingleResult();
  }

  @GET
  @Produces({MediaType.APPLICATION_XML,
        MediaType.APPLICATION_JSON})
  public List<TeamRoster> findAll() {
    return     em.createQuery("select object(o) from TeamRoster o")
          .setHint("javax.persistence.cache.retrieveMode",
               CacheRetrieveMode.BYPASS)
          .getResultList();
  }

  protected EntityManager getEntityManager() {
    return em;
  }
}
```

This class is annotated with `@javax.ejb.Stateless`, marking it as a stateless session bean. The `javax.ws.rs.Path` annotation defines the URL path that can be used to access the class service endpoints. The `TeamRosterFacadeREST` class contains two service methods, `find()` and `findAll()`, to find a `TeamRoster` or a `List` of `TeamRoster`, respectively.

The service can be used by calling the URL `http://localhost:8080/SportsTeamQueryService/rest/teamrosterqueryservice`, or a `TeamRoster` ID can be passed as a path parameter by appending it to the end of the URL.

Registration service. The `SportsTeamRegistrationService` exposes a web service capable of registering new players to a sports team. The service is nearly identical to the `SportsTeamQueryService`, with the exception of the `TeamRosterFacadeREST` class (**Listing 6**), which contains different RESTful web service methods. It contains a `@GET` method named `countRoster()` that returns a count of `TeamRoster` objects, and a `@POST` method named `addPlayer()` that accepts parameters of type `@FormParam` for a player's first name, last name, and position. The method creates a new `TeamRoster` object; sets the values passed in as parameters; and persists the object to the database, using an `EntityManager`. Finally, the method returns a `javax.ws.rs.core.Response` to indicate success or failure.

To register a player, a `javax.ws.rs.core.Form` can be sent via a JAX-RS client to the `addPlayer` web service. The following section demonstrates how to do so via a JSF user interface.

User interface service. The `SportsTeamUIService` drives the user interface, using the JSF framework. The service also requires the JAX-RS client dependency to query the `SportsTeamQueryService` web services and to register new players by using the `SportsTeamRegistrationService`. The POM file is shown in **Listing 7**. The UI service could use any web framework, but here I chose to use JSF. Therefore, the web views are created as XHTML files, and CDI controller classes are used to communicate between the front end and the back end.

The main view, the `index.xhtml` file (shown in **Listing 8**), simply uses a `DataTable` to list each of the registered players, and it also contains a form for registering new players. The backing beans behind `index.xhtml` are CDI controllers named `TeamRegistrationController` (shown in **Listing 9**) and `TeamQueryController` (shown in **Listing 10**).

The controller is annotated with `@RequestScoped`, which means that a new instance of the controller is constructed with each request. Therefore, each time the `index.xhtml` view is visited, the controller is constructed and the `getTeamRosterList()` method is invoked, returning the full list of players. A JAX-RS client is employed to query the `SportsTeamQueryService` to populate the list of players.

The `teamRoster` field of the controller populates and constructs new players (`TeamRoster` objects). Clicking the Register button in the view (**Figure 1**) calls on the `registerPlayer()` controller method. This method creates a new JAX-RS client to initiate a call to the `teamroster-registrationservice.addPlayer` web service. A new `javax.ws.rs.core.Form` is created, and the appropriate fields for a `TeamRoster` object are populated from the data entered into the view. The JAX-RS response is returned from the service call to indicate success or failure. Note that in a real-world scenario, a security API should be used to secure the web service, employing technology such as JSON Web Tokens.

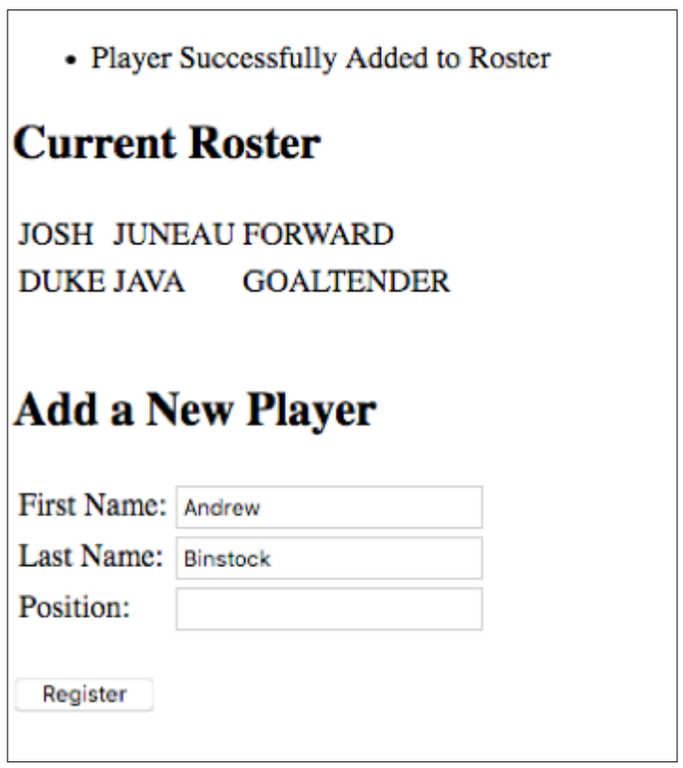


Figure 1: The user interface of the SportsTeamRegistrationService

Deployment Options and Configurations

Jakarta EE is a cloud platform, which means that applications can be deployed to almost any container. As previously mentioned, the three services demonstrated in this article can be deployed to a single application server container, or each of them can be deployed to separate containers such as Docker containers.

In situations where you use separate containers, it might be helpful to use a configuration API such as the one offered by MicroProfile for handling dynamic configurations such as web service URIs. The initial release of Jakarta EE does not contain a configuration API, but one may be added later. It also makes sense to use a standardized security solution such as the Java EE Security API or JSON Web Tokens for securing web service endpoints.

Roadmap

Jakarta EE is a new, open platform, which means it has the potential to evolve at a much faster pace than its predecessor, Java EE. The initial release of Jakarta EE will be at parity with Java EE 8. However, later releases of Jakarta EE will introduce new features, and possibly new APIs, into the platform. Throughout its evolution, the Java EE platform has become easier to use while adding more features along the way. The release of Jakarta EE 9, the second release under the new branding, will begin to paint the picture for the future of the platform.

However, now is the time to get involved with Jakarta EE. One of the most important ways to engage is by joining the mailing lists and participating in the conversations. You can also join the Jakarta EE Working Group or the Eclipse Foundation to become a Committer to the EE4J projects. Either way, if you're writing web apps or enterprise apps in Java, take the time to come up to speed on Jakarta EE. `</article>`

Josh Juneau (@javajuneau) works as an application developer, system analyst, and database administrator. He primarily develops using Java and other JVM languages. He is a frequent contributor to Oracle Technology Network and *Java Magazine* and has written several books for Apress about Java and Java EE. Juneau was a JCP Expert Group member for JSR 372 and JSR 378. He is a member of the NetBeans Dream Team, a Java Champion, leader for the CJUG OSS Initiative, and a regular voice on the *JavaPubHouse Off Heap* podcast.

HENRY **NAFTULIN**

Building Resilience into Microservices with Hystrix

This easy-to-use library from Netflix handles delays and failures in distributed applications.

When you build applications based on a microservices architecture or you break down a monolith into multiple services, you face several common challenges:

- How do you make sure that failure or slowness in one of the subsystems does not make your entire system unavailable to your clients?
- If you have several subsystems that you can use for servicing client requests, what pattern do you use to switch between them? For example, what if you are starting to use a microservice instead of a stored procedure, but you need to retain the ability to call the stored procedure during the migration period? Or perhaps your preference is to get the latest data from a database, but if the database is slow, how you can use an answer from your cache service?
- How do you monitor in real time which of the subsystems are getting too much load or becoming too slow?
- How do you automatically allow a subsystem to recover from a load spike?
- How do you open up your API to be used by other systems, while isolating the potential performance impact to your system?

In this article, I show how these common challenges can be addressed with Hystrix, an open source library developed by Netflix. Its description in GitHub states that Hystrix "is a latency- and fault-tolerance library designed to isolate points of access to remote systems, services and third-party libraries, stop cascading failure and enable resilience in complex distributed systems where failure is inevitable."

You will need knowledge of Spring Boot and Maven to follow along, because I will use these technologies in this article.

Setting the Stage

Imagine that you need to write a microservice that displays your stock brokerage account holdings: specifically, when you give it a user name, it will return the number of shares the user owns, the price of these shares, and the value of the holdings (number of shares multiplied by price per share).

To accomplish this task, you need to rely on a microservice created by your teammate, which accepts the ticker symbol of the stock and returns its price. Let's look first at the price microservice that gives you the price of the stock (**Listing 1**).

Listing 1: A simple service that returns a stock price

```
@RestController
public class TickerServiceController {
   ...

   @RequestMapping(value = "/getTickerPrice/{ticker}",
            method = RequestMethod.GET)
   public Double getTickerPrice(@PathVariable String ticker){
     Random r = new Random();

     long processingTime = (long)
      (r.nextGaussian() * stdDevProcessingTimeInMillis +
       meanProcessingTimeInMillis);

     try {
        Thread.sleep(processingTime);
     } catch (InterruptedException e) {
        LOGGER.error(e.getMessage(), e);
     }
```

```
      if (r.nextDouble() < (outOfServiceRate/100.0)) {
        throw new RuntimeException("Service not available");
      }
      return getTickerPriceFromExchanges(ticker);
    }
}
```

The service is a very simple one: for any name, it returns a price that is generated by the `getTickerPriceFromExchanges` method. For this service, I will simulate two important behaviors: the failure rate of the service and the time it takes to get the data from the datastore. In real life, failure could occur in the larger application when this subsystem is not working because of some unanticipated internal failure—for example, if the datastore is down.

Now let's look at the holdings service. This service uses Spring's `restTemplate`, which sends the request to the price microservice and gets the result back, as shown in **Listing 2**.

Listing 2: Service for requesting a price using Spring's restTemplate

```
@Service
public class TickerPriceRetrieverService  {
   public Double getLatestPrice(String ticker,
          RestTemplate restTemplate) throws Exception {
     Double price = restTemplate.exchange(
          "http://localhost:8098/getTickerPrice/{ticker}"
          , HttpMethod.GET
          , null
          , new ParameterizedTypeReference<Double>() {}
          , ticker).getBody();

     return price;
   }
}
```

I expose this service in my web application via the `getHoldings` endpoint, shown in **Listing 3**, which contains multiple classes.

Listing 3: The holdings service endpoint

```
public interface PriceDelegate {
   Double getLatestPrice(String ticker) throws Exception;
}

@Service
public class NonHystrixDelegate implements PriceDelegate {

   private TickerPriceRetrieverService priceService;

   private final RestTemplate restTemplate;

   public NonHystrixDelegate(TickerPriceRetrieverService priceService){
      this.priceService = priceService;
      restTemplate = new RestTemplate();
   }

   @Override
   public Double getLatestPrice(String ticker) throws Exception {
      return priceService.getLatestPrice(ticker, restTemplate);
   }
}

@RestController
public class HoldingsServiceController {
...

   private final NonHystrixDelegate nonHystrix;
```

```
private final HystrixDelegate hystrix;

...

@RequestMapping(value = "/getHoldings/{customer}",
         method = RequestMethod.GET)
public List<Holding> getHoldings(
             @PathVariable String customer)
             throws Exception {

  List<Holding> holdings = getHoldings();
  holdings.stream().forEach(h -> {
    try {
      h.setPrice(nonHystrix.getLatestPrice(h.getTicker()));
    } catch (Exception e) {
        LOGGER.error(e.getMessage(),e);
      }
    });
    return holdings;

  }

  private List<Holding> getHoldings() {
    return Arrays.asList(new Holding("IBM", 100.0));
  }
}
```

Let's test this implementation. I use Apache's HTTP server testing tool, ab, to load-test this solution. The ab tool will call the holdings service, which in turn calls the price service. I am interested in the resilience of the holdings service, regardless of how the price service behaves.

Let's test the application by simulating a total of 500 requests by 10 clients—first, when the price service takes 100 milliseconds (ms) to respond (test #1); then when the price service takes 1,000 ms to respond (test #2); and finally, when the price service takes 1,000 ms to retrieve the request with a 15% out-of-service rate (test #3).

Here is an example of how you would run the test:

```
ab -n 500 -c 10 http://localhost:8088/getHoldings/Jake
```

Table 1 shows the results from these tests.

As expected, the time for the holdings service to process requests depends directly on the amount of time it takes the underlying price microservice to retrieve results plus some small added overhead.

Once I set the out-of-service rate, some requests failed—just slightly over the expected failure rate of 15%. Because the holdings service processing time and error rate depends directly on the price microservice, this implementation is not resilient to the behavior of the underlying subsystem.

	test #1	test #2	test #3
NUMBER OF REQUESTS	500	500	500
PRICE SERVICE MEAN PROCESSING TIME (MS)	100	1,000	1,000
PRICE SERVICE OUT-OF-SERVICE RATE	0%	0%	15%
PRICE SERVICE STANDARD DEVIATION PROCESSING TIME (MS)	20	200	200
TIME TAKEN FOR TESTS (SECONDS)	5.31	50.666	51.052
NUMBER OF FAILED REQUESTS	0	0	85
REQUESTS PER SECOND	94.00	9.87	9.80
HOLDINGS SERVICE RESPONSE TIME 50TH PERCENTILE (MS)	105	1,011	1,010
HOLDINGS SERVICE RESPONSE TIME 95TH PERCENTILE (MS)	138	1,321	1,334

Table 1: Results of testing the app with three runs of ab

Adding Hystrix to the Mix

Now imagine that the requirements have changed and you need to cap the time you wait for the price microservice to 1,200 milliseconds. If you can't get the latest price from the price service, you'll return the price from the local cache. This scenario is realistic, because you don't want to wait forever for the underlying service to return the result. (And to know the value at risk, or perhaps to know the net worth, it might be better to have a good estimate of the asset price than to have no value at all.)

Although it's possible to write code that would introduce time-outs and implement failure, you are much better served by using the Hystrix library. I will use Hystrix to monitor the response time and execute the failover if the delay crosses the preset threshold. I do this by creating a Hystrix command and instantiating it as shown in the upcoming listings. You will see how Hystrix makes the implementation simple, and how it can be tuned to your needs and provide a way to monitor your interaction with the price microservice.

To start, you need to add a couple of dependencies to the project: one for Hystrix itself and one for the rxjava library, which is used by Hystrix under the covers:

```
<dependency>
   <groupId>com.netflix.hystrix</groupId>
   <artifactId>hystrix-core</artifactId>
   <version>1.2.0</version>
</dependency>

<!-- used by Hystrix -->
<dependency>
  <groupId>io.reactivex</groupId>
  <artifactId>rxjava</artifactId>
  <version>1.2.0</version>
</dependency>
```

Next, you need to define the Hystrix command, which will get the price for a given stock ticker symbol as shown in **Listing 4**.

Listing 4: Defining the Hystrix command

```
static class PriceCommand extends HystrixCommand<Double> {
   private final TickerPriceRetrieverService priceDelegate;
   private final String ticker;
   private final RestTemplate restTemplate;

   public PriceCommand(TickerPriceRetrieverService priceDelegate,
          String ticker, RestTemplate restTemplate,
         HystrixCommand.Setter config) {
     super(config);
     this.priceDelegate = priceDelegate;
     this.ticker = ticker;
     this.restTemplate = restTemplate;
   }

   @Override
   public Double run() throws Exception {
     return priceDelegate.getLatestPrice(ticker, restTemplate);
   }

   @Override
   public Double getFallback() {
     return Cache.getPrice(ticker);
   }
}
```

There are two methods that are usually overwritten in a Hystrix command: `run` and `getFallback`. The `run` method is the workhorse of the Hystrix command: it implements the logic of retrieving

the result from the underlying subsystem or API. Here the code is very simple: I delegate implementation to the existing API that I wrote earlier to get the ticker price (see **Listing 2**). I don't need to worry about setting a time-out here because it (along with other configuration parameters) is set in the configuration object that I use during the initialization of the command. The `getFallback` method is overwritten, because I use an alternative result when the `run` method fails due to time-out or due to the underlying service throwing an error. In my case, the fallback is to use the price from cache. If I didn't overwrite the `getFallback` method, then a time-out or an error received by the price delegate would be propagated to the calling code.

The command needs to be instantiated for every request, because it is stateful and cannot be reused. I instantiate it with a specific configuration, which I describe shortly. The code in **Listing 5** instantiates the Hystrix command.

Listing 5: Instantiating the Hystrix command

```
@Service
public class HystrixDelegate implements PriceDelegate {
   ...
   HystrixCommand.Setter config;

   public HystrixDelegate(
        TickerPriceRetrieverService priceDelegateImpl) {
     this.priceDelegateImpl = priceDelegateImpl;
     restTemplate = new RestTemplate();

     config = HystrixCommand
          .Setter
          .withGroupKey(HystrixCommandGroupKey
             .Factory.asKey("PriceCommand"));

     HystrixCommandProperties.Setter commandProperties =
           HystrixCommandProperties.Setter();
```

```
    commandProperties
     .withExecutionIsolationThreadTimeoutInMilliseconds(1200);

    config.andCommandPropertiesDefaults(commandProperties);
    config.andThreadPoolPropertiesDefaults(
      HystrixThreadPoolProperties.Setter()
        .withMaxQueueSize(-1)
        .withCoreSize(15));

  }

  @Override
  public Double getLatestPrice(String ticker)
      throws Exception {
    PriceCommand pc = new PriceCommand(priceDelegateImpl,
                      ticker, restTemplate, config);

      Future<Double> pcFuture = pc.queue();
      return pcFuture.get();
    }
  }
```

The goal here is to overwrite the default Hystrix configuration, specifically configuring Hystrix to abandon the original request sent to the price service if it takes more than 1,200 milliseconds for processing. In such a case, Hystrix will return the value given by the `getFallback` method.

Under the covers, Hystrix instantiates and uses a thread pool of 10 threads that is assigned to each command type the user had defined. In my case, I have only one Hystrix command, `PriceCommand`, so by default, my command will execute on one thread pool of 10 threads. A little later in the article, I discuss what happens when I use the default settings, but for now I will refer to Hystrix documentation to size my thread pool correctly.

To set the thread pool size that will be used in calling the price service, I use the formula from the documentation:

```
maximum requests per second at peak when healthy × 99th percentile latency in
seconds + some breathing room
```

Let's refer to the test results in **Table 1** rows "Requests per second" and "Holdings service response time 95th percentile" to get the optimal number of threads. When the price service is fast, I get 94 requests per second. Plugging that into the formula and multiplying by 0.138 seconds (95th percentile) equals approximately 12 threads. When the service is slow, the values for the formula are 9.87 requests per second multiplied by 1.321 seconds, which is approximately 13 threads. So, to give myself some breathing room, I set the thread pool configuration to have 15 threads.

An added benefit of using Hystrix is that it is easy to add a real-time monitor to report how the system behaves.

One of the benefits of using Hystrix is that the Hystrix API keeps statistics on Hystrix command execution: how long the command took to process results, whether the command timed out or not, whether there were enough threads in the thread pool to service the requests, and so forth.

These statistics can be used to allow real-time monitoring and to drive the Hystrix command's behavior. By default, when you create a command, Hystrix configures a circuit breaker that monitors command failures. In my example, command failures are the following:

- The ticker price taking more than 1,200 milliseconds (time-out) to return
- An exception being thrown by the command
- The price service being down

After a certain number of failures in a particular time interval, which is again a configuration value that can be changed, the Hystrix circuit breaker will switch from a closed state to an open state. In the closed state, which is the default state, the Hystrix circuit breaker will allow

requests to go to the target system. In the open state, which is usually reached after failure, thresholds are breached, Hystrix will not call the underlying system, and Hystrix will return the value provided by the `getFallback` method.

By default, the circuit is tripped after 20 failed requests or if more than 50% of the requests failed within a rolling time period, which defaults to 10 seconds. **Table 2** shows the results of the same set of ab tests run on the Hystrix implementation.

When the price microservice is fast—that is, configured to process requests within 100 milliseconds on average—the results are very similar to the non-Hystrix implementation. It does take a little longer for Hystrix to process the requests, but that extra time is not significant for my purposes here.

When the price service is configured to take an average of a full second to process requests, Hystrix will time-out the requests that take longer than 1,200 milliseconds, and in such cases, it will use the fallback solution. If a circuit breaker is tripped, requests will not be sent to the price

	test #1	test #2	test #3	test #4
NUMBER OF REQUESTS	500	500	500	500
PRICE SERVICE MEAN PROCESSING TIME (MS)	100	1,000	1,000	1,000
OUT-OF-SERVICE RATE	0%	0%	15%	SERVICE IS DOWN
PRICE SERVICE STANDARD DEVIATION PROCESSING TIME (MS)	20	200	200	200
MAXIMUM ALLOWED TIME FOR REQUEST (MS)	1,200	1,200	1,200	1,200
TIME TAKEN FOR TESTS (SECONDS)	5.401	49.559	49.869	4.723
NUMBER OF FAILED REQUESTS	0	0	85	0
REQUESTS PER SECOND	92.58	9.87	10.3	105.87
HOLDINGS SERVICE RESPONSE TIME 50TH PERCENTILE (MS)	108	997	1,001	36
HOLDINGS SERVICE RESPONSE TIME 95TH PERCENTILE (MS)	140	1,207	1,207	109
NUMBER OF REQUESTS USING FALLBACK METHOD	0	83	170	500

Table 2: Test results using Hystrix

service for a while and a fallback solution is used until Hystrix decides to close the circuit again.

You can see this behavior clearly when I bring the price service down. The application tries to access the service initially; then the circuit breaker trips and the application will not even try to access the underlying service. Specifically, in this example, you can see that on average, my pricing requests are taking much less time (50th percentile is 36 ms) than in the column where 15% of requests are failing.

This behavior highlights the main features of a Hystrix command that uses a circuit breaker:

- It returns a fallback value if the underlying library or service is not responsive or throws an error.
- It does not propagate the errors from underlying libraries to the client unless your requirements specify to do so.
- It gives the library or the underlying service some time to recover. Once the circuit breaker is open, most requests will not hit the underlying service. There will be one probing request sent once in a while to see if the underlying system recovered. If the probing request succeeds, the underlying system is deemed to be healthy and the requests are routed to use the system.

Monitoring

An added benefit of using Hystrix is that it is easy to add a real-time monitor to report how the system behaves. To do that, I need to add the following library, which is responsible for emitting Hystrix statistics, into my holdings project:

```
<dependency>
   <groupId>com.netflix.hystrix</groupId>
   <artifactId>hystrix-metrics-event-stream</artifactId>
   <version>1.2.0</version>
</dependency>
```

And then I configure the stream servlet as shown in **Listing 6**.

Listing 6: Configuring the stream servlet to get real-time statistics

```
@Configuration
public class HystrixStreamServletDefinition {

   @Bean(name = "hystrixRegistrationBean")
   public ServletRegistrationBean servletRegistrationBean() {
     ServletRegistrationBean registration =
       new ServletRegistrationBean(
          new HystrixMetricsStreamServlet(),"/hystrix.stream");
     registration.setName("hystrixServlet");
     registration.setLoadOnStartup(1);
     return registration;
   }
}
```

That's it. Now I need to create a monitoring application. I decided to keep my monitoring application as a separate service, because I didn't want it to affect the performance of the holdings service. In real life, you would probably configure a similar service adding the right authentication and authorization and then let it monitor all your Hystrix streams in one place.

To start a standalone monitoring service, you can download the entire setup, but it is just as easy to create the service with Spring Boot. To create it, you need to have the following libraries in Spring Boot:

```
<dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-starter-hystrix-dashboard</artifactId>
   <version>1.4.5.RELEASE</version>
</dependency>
```

Listing 7 shows how I configured the monitoring application.

Listing 7: Configuring the monitoring application

```
@SpringBootApplication
@EnableHystrixDashboard
public class HystrixMonitorApplication {

   public static void main(String[] args) {
     SpringApplication.run(HystrixMonitorApplication.class, args);
   }
}
```

Here, the annotation that enables the Hystrix dashboard does the trick. Run the services and paste the stream URL (http://localhost:8088/hystrix.stream) into the Hystrix dashboard application (at http://localhost:8078/hystrix/), as shown in **Figure 1**.

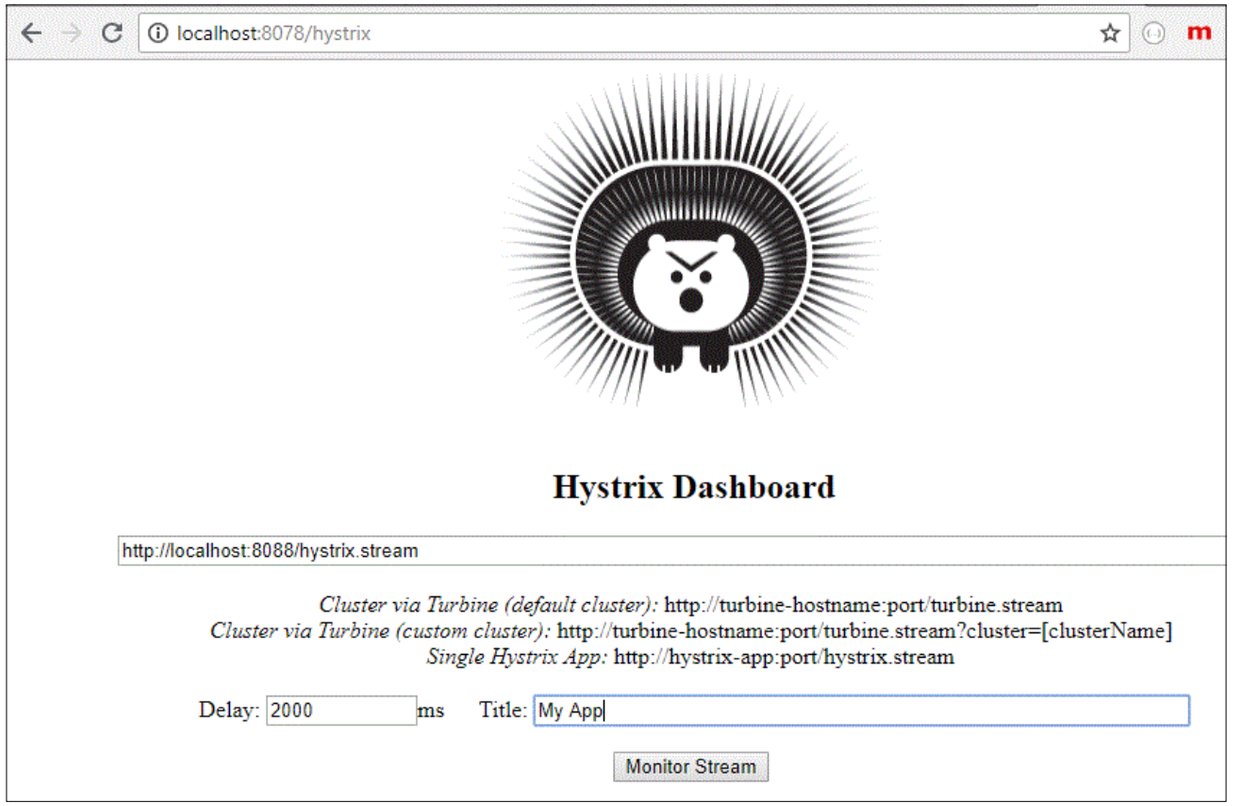


Figure 1: Hystrix dashboard application

Figure 2 shows a screenshot of the monitoring app while I reran one of the tests. In Figure 2, the yellow circle indicates the volume and the health of the requests, and the blue line indicates the request rate. I had 66 successful requests within this time period, 16 time-outs, 12 exceptions, and a 24% command error rate for the reported time interval. The Hystrix command processed requests at a speed of 9.8 requests per second.

Hystrix monitoring provides useful information as you look at systems you depend on. And based on my experience, running the dashboard helps you understand how Hystrix works. The dashboard is also extremely helpful in figuring out which parameters to adjust.

For example, for a while I ran my example application configured with the Hystrix default thread pool size of 10 threads. I observed that responses were coming to the client much faster than I expected. It turns out there were not enough threads processing the requests,

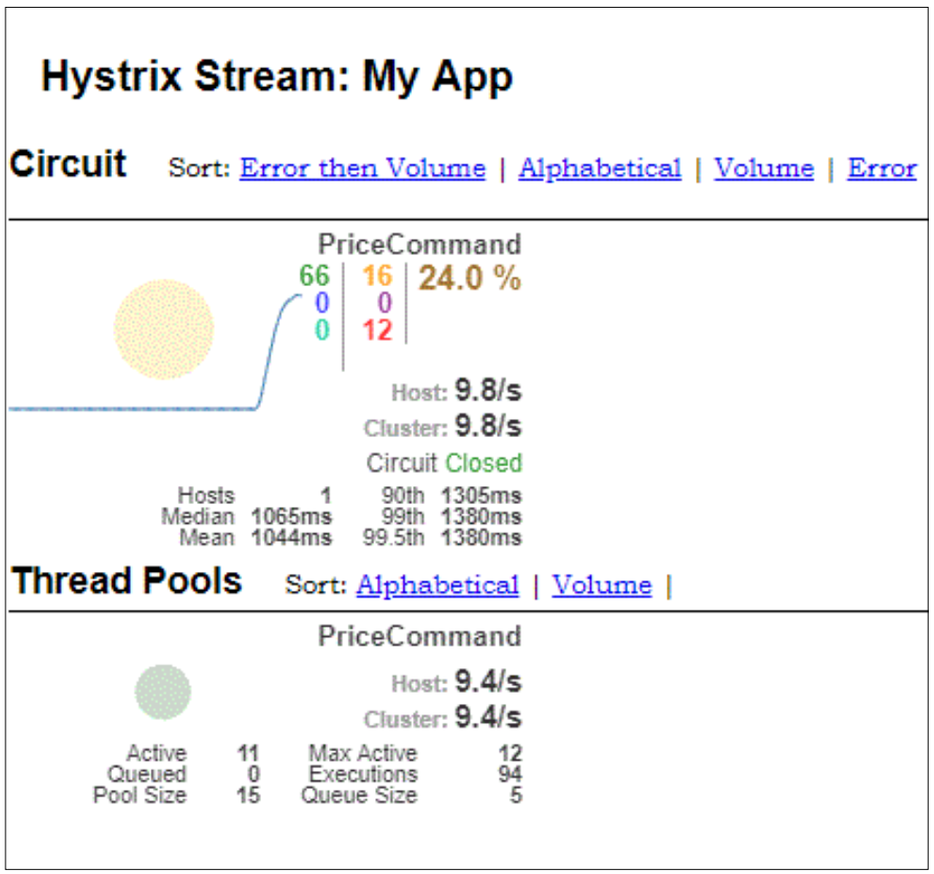


Figure 2: Screenshot of monitoring app

which resulted in Hystrix counting each such failed request as an error; after a while that forced Hystrix to open the circuit breaker. Once the circuit breaker was open, the fallback response was served to the client, and that was much faster than a response served from the price service.

Also, while running the Hystrix dashboard, I saw that the circuit breaker opened shortly after requests were sent and that the thread pool rejection count was getting high. Once I adjusted the thread pool size using the formula Hystrix provides, as discussed earlier, the thread pool rejections stopped and the tests behaved as I expected.

Conclusion

In this article, I have demonstrated how to use Hystrix in a simple, real-life example that showed how to ensure failure not make the application unavailable. This is a good example of a system where getting real-time prices is a preferred solution, while falling back to cached prices is acceptable.

Hystrix is used by many enterprises, because it is a very useful library that can help make your systems more resilient. This example showed how easy it is to use Hystrix and the considerable benefits it delivers. `</article>`

Henry Naftulin has been designing Java EE distributed systems for more than 15 years. He is currently leading development of a proprietary award-winning fixed-income trading platform for one of the largest financial companies in the United States.

ALEKSEI VOITYLOV

Java on Arm Processors

Arm already is a leading architecture for IoT and embedded processors. Recent 64-bit releases are pushing Arm CPUs onto servers, where they have full JDK support.

Today, Arm-based processors primarily are viewed as targeting the embedded market, because they offer sufficient performance while keeping power consumption low. But many hardware vendors are now using this architecture to build server CPUs and to compete with x86 architecture in the cloud and in the high-performance computing (HPC) market. This range of deployment platforms adds to the complexity of the Java Arm port, because the port must support a variety of CPU vendors and workloads.

In this article, I explore the evolution of Java and the Java ecosystem and their status on Arm architectures. I also discuss some recent developments in Java features and performance for Arm processors, emphasizing both server and IoT/embedded deployments. [The company behind Arm processors, Arm Limited, is transitioning its brand from the acronym ARM to *Arm* and *arm*. We use the traditional capitalized form *ARM* when referring to specific processor models and *Arm* in all other instances. –*Ed.*]

The State of the Arm Architecture

Leaving aside the embedded and mobile markets, where Arm dominates with its 32-bit ARMv5, ARMv6, ARMv7, and ARMv8 instruction set architectures (ISAs), it's no longer stretching the point to say that Arm provides a viable alternative for markets that are currently dominated by the x86 architecture. Unlike microprocessor vendors such as Intel that focus on shipping processors, Arm is primarily an architecture design company selling architectural and core licenses to its customers, which turn that intellectual property into actual silicon. This model allows a great variety of actual implementations of the same architecture to coexist and compete in different market segments.

It is clear from recent developments in the Arm architecture itself that the focus has shifted to competitive Arm-based server CPU designs. In 2016, Arm finalized a 64-bit- and 32-bit-capable ARMv8-A ISA that targets both the embedded and server markets. This architecture mandated the presence of a single instruction, multiple data (SIMD) instruction set (called *NEON*) and introduced optional instructions for AES encryption and for SHA-1, SHA-256, and CRC32 calculations, which some vendors use to boost cryptographic and checksum performance.

In 2017, Arm extended this architecture by adding new atomic instructions. Later, the ARMv8.2-A ISA added half-precision floating-point data processing and specialized SIMD instructions that improve the performance of machine-learning computations. In addition, in the ARMv8.2-A ISA, optional Scalable Vector Extension (SVE) instructions were introduced for better support of vectorization (compared to the NEON instruction set), thereby making the ARMv8 architecture much more suitable for technical computing. Most recently, the ARMv8.3-A ISA added SIMD complex-number support.

It's no longer stretching the point to say Arm provides a viable alternative for markets currently dominated by the x86 architecture.

The ARMv8 architecture leaves room for vendor design selection to achieve performance, complexity, and power goals. It adopts a relaxed hardware memory model that is weaker than that of x86 processors (which use x86-TSO). Thus, you can observe more out-of-order effects. This architecture also adds new concurrency primitives, including the load-acquire and store-release instructions, as well as weaker barrier instructions. But most Java developers will not notice these changes because the JVM hides them inside the implementation.

Several hardware vendors compete with Intel in the server market with their ARMv8-based processor design. Some hardware vendors are already established in the Arm-based server market and have delivered 64-bit production systems used in data centers for several years now.

In technical computing, Sandia National Labs is deploying an Arm-based supercomputer that has a theoretical peak of more than 2.3 petaflops. All major Linux distributions support

Arm, including Debian, Oracle Linux, Red Hat Linux, SUSE, and Ubuntu. All the tooling at the OS and kernel level is stable and ready for production use.

Availability of Java on Arm Architectures

End users will find a good choice of providers for Java and OpenJDK binaries for Arm-based architectures. The Java ports for both the ARMv7 and ARMv8 ISAs are fully functional, and the codebases are available from OpenJDK under the GPLv2.1 license with the classpath extension, which enabled most Linux distributions to bundle them.

If your favorite Linux distribution does not contain the required packages or you are looking for commercial support, an excellent set of Java/OpenJDK binaries is provided by AdoptOpenJDK, Azul, BellSoft, and Oracle. At the time this article was written, Azul and Oracle provide only JDK 8 binaries for the ARMv8 and ARMv5/6/7 ISAs, while BellSoft offers binaries for JDK 9 and 10 that, for the Raspberry Pi, include the OpenJFX and Device I/O API modules. Azul, BellSoft, and Oracle provide supported binaries that comply with the Java SE specification, and they verify their binaries with the Java Compatibility Kit (JCK) test suite.

Features of the Java Ports for Arm Architectures

Although it is very important to ensure the compatibility of Java implementations, passing the JCK test suite is not the only requirement for a successful Java port. To meet startup and throughput performance expectations, Java ports for both the ARMv7 and ARMv8 ISAs implement C1 and C2 JIT compilers, thus allowing them to produce optimized code that takes advantage of the underlying architecture specifics. On top of that, the `-XX:+TieredCompilation` command-line option is supported and turned on in the server virtual machine (VM), which allows faster startup and higher C2 throughput. A full set of garbage collectors (GCs)—the parallel G1, the serial GCs, and the deprecated CMS—is supported in both the ARMv7 and ARMv8 Java ports.

For embedded use cases, some ARMv7 ports include a lightweight minimal VM. On JDK 9 or higher, the new Java modules enable building Java runtime images that have a low static footprint. Running the following commands on the BellSoft ARM JDK 10, for example, pro-

duces a Java runtime with the `java.base` module that has a static footprint as small as 16 MB. Surprisingly, `java.base` (perhaps with the addition of several other modules) is sufficient for many Java applications that are tailored for constrained IoT gateways. For example, a runtime capable of running Apache Felix or Jetty fits into 32 MB.

```
OUTPUT=~/out
bin/jlink --module-path jmods --compress=2 --add-modules java.base --output $OUTPUT
rm -r $OUTPUT/lib/client $OUTPUT/lib/server
echo "-minimal KNOWN" > $OUTPUT/lib/jvm.cfg
```

Over the years, the ARMv8 port received built-in optimized assembly intrinsics for CPU-intensive operations. At present, only several intrinsics present in the x86 port are absent in the ARMv8 port, and the gap will be closed soon by JEP 315.

All the common features that appear in other Java ports also work on Arm, including Docker support and Application Class-Data Sharing (AppCDS) v2, as specified in JEP 310.

Table 1, on the following page, provides a detailed comparison of major JVM features on x86/64, ARMv8 64-bit, and Arm 32-bit ports.

Performance of the Arm 64-Bit JVM Port

Let's dive into the performance of the ARMv8 port, because the server market is where performance matters most. To make a valid comparison, it is important to find x86- and Arm-based server equivalents. Luckily, the recently released Cavium ThunderX2 ARMv8 CPU line provides a processor that's comparable to the Intel Xeon processors based on SPECint2017 rates.

For this comparison, I selected the Cavium ThunderX2 CN9975 and the Intel Xeon Gold 6140 single-socket systems, both equipped with DDR4-2666 memory and running Ubuntu 16.04. (Dual-socket systems with these CPUs are also available.) The ThunderX2 CN9975 CPU has 112 threads (28-core system with 4-way SMP), and the comparable Intel Xeon Gold 6140 CPU has 36 threads (18-core system with Hyper-Threading).

To assess the performance of the JVM ARMv8 and x86 ports, I ran the widely used SPECjbb2015 1.01 and SPECjvm2008 1.01 benchmarks with OpenJDK 11 EA build 18. All benchmarks were executed 20 times, and the mean values were collected. The SPECjbb2015 benchmark was used to obtain an overall score, while the SPECjvm2008 benchmark provided additional insights into the performance of the ARMv8 HotSpot JVM port.

		x86/64	ARMv8 (64-BIT)	ARM (32-BIT)
VMS	CLIENT	YES	NO	YES
	SERVER	YES	YES	YES
	MINIMAL	YES (32-BIT)	NO	YES
JIT	C1 COMPILER	YES	YES	YES
	C2 COMPILER	YES	YES	YES
	`TieredCompilation`	YES	YES	YES
	GRAAL JIT COMPILER (EXPERIMENTAL)	YES, SINCE JDK 10	YES, SINCE JDK 11	NO
GC	SERIAL GC	YES	YES	YES
	PARALLEL GC	YES	YES	YES
	CMS GC	YES; DEPRECATED	YES; DEPRECATED	YES; DEPRECATED
	G1 GC	YES	YES	YES
	Z GARBAGE COLLECTOR (ZGC)	EXPERIMENTAL	IN DEVELOPMENT	NO
RUNTIME	CONTAINER SUPPORT	YES	YES	YES
	APPCDS	YES	YES, SINCE JDK 10	YES, SINCE JDK 10
	LINUX HUGEPAGES	YES	YES	YES
	NUMA SUPPORT	YES	YES	NO
SERVICEABILITY	FLIGHT RECORDER (JEP 328)	YES	YES, SINCE JDK 11	YES, SINCE JDK 11

Table 1: Comparison of features for major x86 and Arm JVM ports

Because the intent of this article is not to report the best score obtainable on a specific hardware system but instead to study the performance a typical end user would see, I intentionally did not fine-tune low-level JVM parameters or kernel settings on either system. Check the SPEC scores for the processors, as reported by the hardware vendors, to compare the highest achievable numbers available with JVM options tuning.

SPECjbb2015 results. Figure 1 presents the SPECjbb2015 1.01-Composite results (Critical-jOPS and Max-jOPS) for a single-socket Intel Xeon Gold 6140 system and a ThunderX2 CN9975 single-socket system, both with DDR4-2666 memory and running Ubuntu 16.04. (Higher scores are better.)

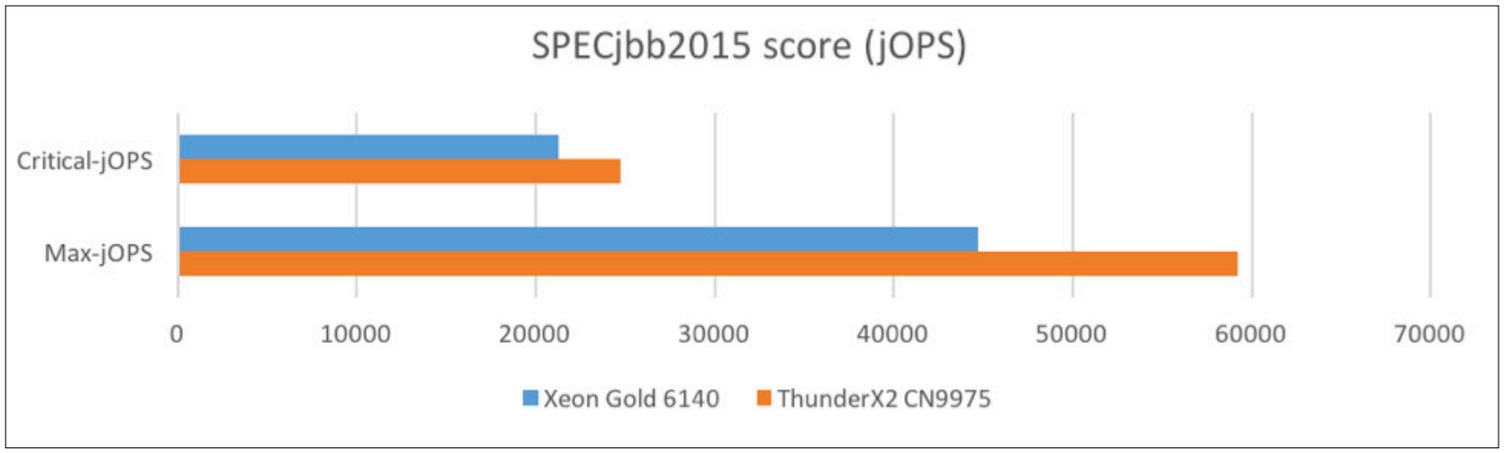


Figure 1: SPECjbb2015-Composite performance results

The JVM command-line options used for these runs were very common for SPECjbb2015 runs. On the Arm-based system, I used the following:

```
-Xmx24G -Xms24G -Xmn16G -XX:+AlwaysPreTouch -XX:+UseParallelGC
-XX:+UseTransparentHugePages -XX:-UseBiasedLocking
```

On the x86-based system, I used this:

```
-Xmx24G -Xms24G -Xmn16G -XX:+AlwaysPreTouch -XX:+UseParallelGC
-XX:+UseTransparentHugePages -XX:+UseBiasedLocking
```

(Switching biased locking off for the ARMv8 architecture and leaving it on for the x86 architecture gave both platforms slightly better results.)

A full set of garbage collectors—the parallel G1, the serial GCs, and the deprecated CMS—is supported in both the ARMv7 and ARMv8 Java ports.

As you can see in **Figure 1**, the OpenJDK 11 ARMv8 port running on the ThunderX2 CN9975 system outperformed the x86 port running on the Intel Xeon Gold 6140 system by 33% for the Max-jOPS score and by 16% for the Critical-jOPS score. This suggests the ThunderX2 system with the ARMv8 JVM port is very suitable for enterprise workloads represented by the SPECjbb2015 benchmark.

To assess per-thread performance, I also limited the number of CPU threads on the ThunderX2 system to be the same as on the Intel Xeon Gold 6140 system, which used only 32% of its CPU threads. Unsurprisingly, in this case the SPECjbb2015 results clearly favored the Xeon Gold 6140 system, giving it a 30% advantage.

SPECjvm2008 results. **Figure 2** presents the SPECjvm2008 base results for individual benchmarks together with the composite base results for a single-socket Xeon Gold 6140 system and a single-socket ThunderX2 CN9975 system, both of which had DDR4-2666 memory and were running Ubuntu 16.04. (Higher scores are better.) Because the SPECjvm2008 "compiler" subbenchmark has not worked in this suite since JDK 8, the composite geometric mean base score was manually calculated without a "compiler" benchmark result.

As you can see in **Figure 2**, the OpenJDK 11 ARMv8 port running on the ThunderX2 CN9975 system outperformed the x86 port running on the Xeon Gold 6140 system by 28% in the SPECjvm2008 benchmark composite base score. There are two main reasons for the overall better score on the ARMv8-based system. The first is that the system has a higher memory bandwidth (eight channels compared with six channels on the Xeon Gold 6140 system). The second is related to the work done in the ARMv8 Java port that allowed the full utilization of the CPU potential and extensions.

To gain additional insights, let's explore the scores for individual SPECjvm2008 workloads. In eight out of nine SPECjvm2008 benchmarks, the ARMv8 results outperformed the Intel pro-

cessor, and in the remaining result, the Intel processor was faster. The crypto benchmark results clearly favor an ARMv8-based system, giving it a 62% advantage, which could not be attained if the ARMv8 port didn't fully utilize the AES and SHA extensions available on the Arm chip.

The compress benchmark (in which the ARMv8 system leads by 12%) uses the CRC32C intrinsic. The XML benchmark (in which the ARMv8 processor leads by 29%) and the mpegAudio benchmark (in which the ARMv8 leads by 44%) use the `java.lang.String` and `java.lang.Arrays` intrinsics. Some of these intrinsics were recently improved in JDK 10 and 11 for the ARMv8.

It is also important to understand the results for the benchmark where the x86 OpenJDK port did better (by 29%): scimark.small. The reason for that is the benchmark code: the FFT, LU, SOR, and SPARSE scimark sub-benchmarks all contain heavy loops and matrix computa-

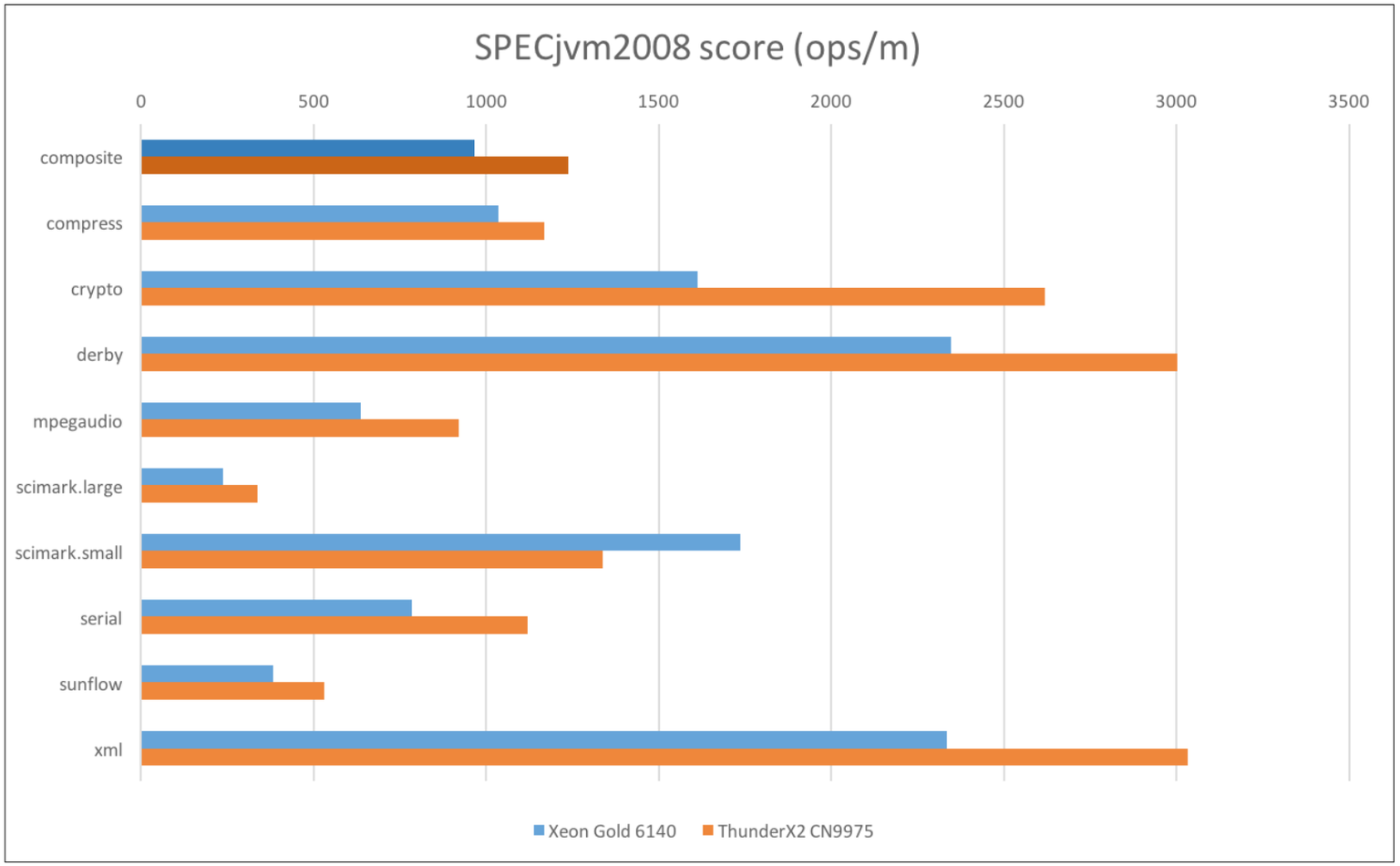


Figure 2: SPECjvm2008 performance results

tion code. Over the years, Intel has invested a lot of effort into loop unrolling and vectorization, which allowed the mapping of such code sequences to AVX instructions on x86 processors. This work has not yet been completed for the ARMv8 C2 port, and the absence of a good equivalent to the Intel AVX 512-bit instruction set does not help.

Performance diagnostics via JDK tools such as Java Management Extensions (JMX) and the JVMTI API work on Arm-based systems just as they do on x86 systems.

There is definitely some work ahead to bring the ARMv8 port's scientific workload performance up to par with that of the x86 implementation. However, for regular server-side Java business application workloads (data processing, XML, crypto operations, and so forth), the OpenJDK 11 ARMv8 port running on Cavium ThunderX2 units currently provides better performance compared with the x86 equivalent.

Performance Diagnostics

Performance diagnostics tools are essential for understanding the bottlenecks of a Java application being developed or run in production.

Regular performance diagnostics via JDK tools such as Java Management Extensions (JMX) and the JVMTI API work on Arm-based systems just as they do on x86 systems. For more-thorough Java performance analysis, a group I work with ported the Async Profiler and Honest Profiler to the ARMv8 and contributed the changes back to the project. These ports enabled enhancement of the performance of an application as complex as Hadoop on ARMv8 systems. If you intend to work on a complex Java application and would like to profile the JVM bottlenecks on the Arm architecture (or any other architecture), these are the open source tools I would recommend.

Flight Recorder, which was open-sourced by Oracle and contributed to OpenJDK 11, is also available in Arm-based ports.

Java Ecosystem on Arm Systems

In theory, all software written in Java should work on all Arm-based systems. However, some big projects make specific tweaks that tie them to a specific architecture, such as using natively built libraries. The following popular projects, although not claiming official support for the ARMv8 ISA, were tested and work on Arm systems without modification: Hadoop 3.1.0, Tomcat 9.0.8, Spark 2.3.0, Kafka 1.1.0, Cassandra 3.11.2, Lucene 7.3.0, and Flink 1.4.2.

Future Developments

Several companies—including Arm itself, Azul, BellSoft, Cavium, Linaro, Oracle, Red Hat, and others—collaborate in the OpenJDK codebase to ensure the long-term future of Arm-based ports. This work includes gradual improvements in performance and stability, as well as work on a fully supported GraalVM and Graal as a JIT compiler on ARMv8 processors. Future projects such as Valhalla and Panama will be part of this effort as well.

Conclusion

The upstream Arm 32-bit and ARMv8 Java ports are ready for production use, and all of the relevant features are on par with those of x86 platforms.

The 32-bit Arm port provides all the necessary functionality for embedded and IoT deployments, including the C1 compiler for fast startup, a low dynamic memory footprint, and a minimal VM, which allows for the production of Java runtime images that have a low static footprint (under 16 MB). This port works well on such popular devices as the Raspberry Pi and, after proper device and application-specific tuning, the 32-bit Arm port can be used in production under the GPL license.

The ARMv8 port that is aimed primarily at the server market shows better performance results when compared with x86 counterparts on equivalent hardware (a 16% advantage in the SPECjbb2015 Critical-jOPS

For embedded and IoT use cases, the Arm platform is already the primary platform of choice.

benchmark, a 33% advantage in the SPECjbb2015 Max-jOPS benchmark, and a 28% advantage in the SPECjvm2008 base composite benchmark). As demonstrated by the SPECjvm2008 benchmarks for typical server-side Java business applications that process and encrypt data and XML files, the OpenJDK 11 ARMv8 port running on a Cavium ThunderX2 system is faster than the Intel counterpart.

The Java software ecosystem is ready for production deployments on Arm-based systems. For embedded and IoT use cases, the Arm platform is already the primary platform of choice, but why would server manufacturers and major cloud providers consider moving to a different architecture if the performance advantage is only tens of percents? Price/performance is the answer. Given the performance of the JVM on Arm-based systems and the price of the CPUs, this starts to make sense. And it becomes very easy to try—considering how little effort is required to take existing Java applications to a new architecture. </article>

Aleksei Voitylov is the CTO of BellSoft, where he drives innovation for the company's customers, including optimizing OpenJDK for Cavium systems. Prior to cofounding BellSoft, he ran teams at Oracle and Sun Microsystems and, in particular, he helped deliver multiple components of the JDK 8 and 9 releases, including the HotSpot JVM, and the Java language. He holds a PhD from Saint Petersburg State University.

Java™

developer.oracle.com/java

DEVELOP WITH THE
GLOBAL STANDARD

#1 Developer Choice for the Cloud

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

Copyright © 2018, Oracle and/or its affiliates. All rights reserved. Oracle and Java are registered trademarks of Oracle and/or its affiliates.

ORACLE®

IAN DARWIN

The Visitor Design Pattern in Depth

Perform one or more operations on a collection of different data types without disrupting existing code.

Suppose it's your first day at a new job at a midsize company. You'll probably be escorted around the building and introduced to every department of the organization. At each one you'll say "Glad to meet you" a few times and talk with the team there to discuss your common projects, and then you'll say "Nice to have met you." And you'll repeat this for each department. Congratulations! You have just implemented the Visitor design pattern in humanware.

The Pattern

Visitor is a useful pattern when you have many objects of different types in your data structure and you want to apply some operation to several or all of them. The pattern is helpful when you don't know ahead of time all the operations you will need; it gives you flexibility to add new operations without having to add them to each object type. The basic idea is that a `Visitor` object is taken around the nodes of a data structure by some kind of iterator, and each node "accepts" the visitor, allowing it access to that node object's internal data. When a new function is needed, only a new visitor needs to be written. The iteration is conceptually simple:

```
for (Node node : collection) {
   node.accept(visitor);
}
```

(There are two main code examples in this article; both can be found in my GitHub repository.

Code from other articles in this series on design patterns can be found further up the trunk of that repository.)

The `Node` objects must know how to accept the `Visitor`, and they will usually call a method on the `Visitor` that is appropriate to the type of the node—for example:

```
class TextNode implements Node {
   void accept(Visitor v) {
     v.visitTextNode(this);
   }
   // other state and methods
}
```

Therefore, one consequence of this pattern is that the `Visitor` needs to know about all the node types it might encounter.

Double Dispatch

Many explanations of the Visitor pattern refer to it as *double dispatch*. This term sometimes makes readers think of a two-step dispatching process, as with a pointer to another pointer used in some languages. That's not what is meant. The term refers to the fact that both the type of the visitor and the type of the node (or "receiver") are used in sorting out which method winds up doing the work. You can see this in the `accept()` method above: there's the call to `accept()` and the call back to `visitTextNode()`.

Visiting the Text

Suppose I need to maintain a word processor that was written in Java. There are a few data types (text node, image node, and so on). Common operations, such as editing text, setting fonts, and setting colors, are taken care of. But there are many supplemental operations that need to be performed on the text, and new ones come along often as customers provide feedback. Here's what the text node's class started as:

```
public class TextNodeOld extends Node {
   private StringBuilder text = new StringBuilder();

   public TextNodeOld() {
     // empty
   }

   public TextNodeOld(String s) {
     // Here, you know the StringBuilder exists and is empty
     text.append(s);
   }

   public String getText() {
     return text.toString();
   }

   public void setText(String text) {
     this.text.setLength(0);
     this.text.append(text);
   }

   // Lots of supplemental functionality methods here
   // that will be added below
}
```

It's becoming annoying that all the data types need to be modified every time somebody has an idea for a new function. I know from experience that I'm unlikely to be able to predict, at the start of the maintenance, all the remaining functionality that will be needed. So I'll introduce a Visitor pattern.

The basic data structure is still the `Node`, with subclasses `TextNode` and `ImageNode`. A real word processor would have more types of nodes, but I want to focus on the Visitor pattern, not com-

pete with the well-known word processor that's out there. Therefore, `Node` is now an interface with just one method:

```
public interface Node {
   abstract void accept(Visitor v);
}
```

I was tempted to call this interface `Visitable` instead of `Node`. On one hand, `Visitable` is a more descriptive name for this version. On the other hand, most formal definitions of Visitor use the term `Node`. I know some of you will go to Wikipedia to get a second opinion after reading this, and I don't want to confuse anyone.

`Node` could alternatively be an abstract class, but that would force all the implementation classes to be related by inheritance, which may be an unnecessary restriction.

`Node` uses `Visitor` as a type, so the next step is to define `Visitor`:

```
public abstract class Visitor {
   public abstract void visitTextNode(TextNode textNode);

   public abstract void visitImageNode(ImageNode imageNode);

   // And so on for TableNode, SectionNode, VideoNode, and so forth
}
```

Note that you could make all the methods be overloads of a single method called `visit()`, because the argument types are unique, but I think this way is clearer. It's a stylistic choice, so pick one way and try to be consistent.

At any rate, here you meet the one complication of the Visitor pattern: *Visitors need to know how to visit every main kind of node.*

The revised node classes themselves are not that interesting, so I didn't show their code—the `Text` node has a `Text` property; the `Image` node has a `FileName`, a width, a height, and an

optional `Caption` (which is subclassed from `TextNode`); and so on.

With all that structure in place, it's time to start to write visitors. First, suppose there's a requirement to print a quick draft of the document, without trying to display the images (this capability was in the requirements from the days when graphics printers were expensive). The text stored in a `TextNode` might contain more characters than fit on a line, so I use an existing program called `Fmt` to crudely format lines to fit. `Fmt` wants its input as a stream (even though in this case it's only one string), so the `visitTextNode()` method wraps the current `TextNode`'s string in an array and streams that to the `format()` method of `Fmt`.

The iteration doesn't need to be a for loop or even an Iterator—any means of traversing all the nodes is fine.

```
static Visitor draftPrinterVisitor = new Visitor() {
  @Override
  public void visitTextNode(TextNode textNode) {
    String[] lines = { textNode.getText() };
    Fmt.format(Stream.of(lines), out);
  }

  @Override
  public void visitImageNode(ImageNode imageNode) {
    String caption = imageNode.caption != null ?
      imageNode.caption.getText() : "no caption";
    System.out.printf("Image: name='%s',
      caption='%s'%n", imageNode.fileName, caption);
  }
};
```

The `Fmt` program requires a `PrintWriter` for output, so the code on the following page wraps `System.out` in a `PrintWriter` before passing the `draftPrinterVisitor` around to all the nodes.

The `visitImageNode()` method doesn't need to use `Fmt`, because image captions are assumed to be one line long. The method simply gets the text from the `ImageNode`'s caption (which is a subtype of `TextNode`, so it has a `getText()` method), defaulting to "no caption" if there is no caption, and prints the result to `System.out`.

The main demo program, `WordProcessorDemo`, creates a demo document and iterates over its `Node` instances like this:

```
out = new PrintWriter(System.out);
for (Node n : nodes) {
  n.accept(draftPrinterVisitor);
}
out.flush();
```

Note that the iteration doesn't need to be a `for` loop or even an iterator—any means of traversing all the nodes is fine.

Suddenly, someone from marketing rushes in and says, "Gee, this draft format is neat. But the boss wants it to show the word count as well. Can you add a function to compute that too?"

"No problem," you can say, turning back to the code. Soon the new code takes shape. The following `Visitor` counts the number of words in text nodes, and it even descends into `ImageNode`s to get the word count of the caption, if there is one.

```
public class WordCountVisitor extends Visitor {

  int wordCount = 0;

  public int getWordCount() {
    return wordCount;
  }

  @Override
```

```
public void visitTextNode(TextNode textNode) {
  wordCount += wordCount(textNode.getText());
}

@Override
public void visitImageNode(ImageNode imageNode) {
  // You might say there's nothing to do, but let's try this:
  if (imageNode.caption != null) {
    visitTextNode(imageNode.caption);
  }
}

/** Simplistic implementation of word counting */
private int wordCount(String text) {
  // Replace all nonspace chars with nothing;
    // add one because "hello word" has one space,
    // but it is two words.
    return text.trim().
      replaceAll("[^\\s]", "").length() + 1;
  }
}
```

This code is plugged into `main()` in a similar fashion:

```
Visitor wordCountVisitor = new WordCountVisitor();
for (Node n : nodes) {
  n.accept(wordCountVisitor);
}
System.out.printf("The document has about %d words%n",
  ((WordCountVisitor) wordCountVisitor).getWordCount());
```

And when the code is run with the sample document, it prints this, which turns out to be the correct answer:

```
The document has about 78 words
```

Revisiting the JDK

Java 8 and later versions include two sets of visitor types: `ElementVisitor` or `TypeVisitor`, and `FileVisitor`. The `ElementVisitor` types are part of the package `javax.lang.model`, which bills itself as "Classes and hierarchies of packages used to model the Java programming language." I don't have room in this article to write my own Java compiler, so I'll skip the language-modeling types. But I'll note that the Visitor pattern is often explained in terms of a program language compiler visiting the nodes of an abstract syntax tree (AST), which is the output of the parsing phase.

The `FileVisitor` and its solitary implementation class, `SimpleFileVisitor` in `java.nio`, are specialized for processing file hierarchies. They do follow the `visit...` naming pattern. A typical use is to subclass `SimpleFileVisitor`—overriding one or two of its four methods—and pass an instance of it to the `Files.walkFileTree()` method. The nodes you're visiting this time are actual file system nodes (represented by inodes in the UNIX/Linux sense). The `walkFileTree()` method performs the iteration, and it calls your `FileVisitor`'s visitation methods to "do something" at the beginning and end of each directory and for each file in each directory. A simple directory lister, for example, can be made with just the following visitor class (this example is in the `visitor.file` package in the GitHub repository):

The Visitor pattern allows you to retain flexibility to add new methods at a slight cost: the reduction of encapsulation and the need for every visitor to know about all the different node types.

```
public class TrivialListerVisitor extends SimpleFileVisitor<Path> {
```

```
    @Override
    public FileVisitResult preVisitDirectory(Path dir,
      BasicFileAttributes attrs) throws IOException {
      System.out.println("Start directory " + dir);
      return FileVisitResult.CONTINUE;
    }

    @Override
    public FileVisitResult visitFile(Path file,
      BasicFileAttributes attrs) throws IOException {
      System.out.println(file.getFileName());
      return FileVisitResult.CONTINUE;
    }
}
```

I also need to invoke the `walkFileTree()` method to do the iteration, using a `Path` object to describe the directory. This code is in the `main()` method of `FileVisitorDemo.java`:

```
// Set the starting path
Path startingPath = Paths.get(".");

// Instantiate the Visitor object
FileVisitor<Path> visitor = new TrivialFileVisitor();

// Use the built-in walkFileTree client to
// visit all directory and file nodes
Files.walkFileTree(startingPath, visitor);
```

This code works, although it's obviously not a replacement for something like the UNIX/Linux/MacOS `ls` command, which sorts the entries and has a zillion options.

A slightly fancier version might indent one tab stop for each directory level. There's a start at making such a thing in the class `IndentingFileVisitor` in my GitHub repository, although it doesn't work superbly yet. To try it, just change the instantiation of the `FileVisitor` in the `main` method.

Conclusion

Besides the examples of the word processor add-on and directory navigation, are there other uses of the Visitor pattern? Certainly! Examples include its use in compilers (as mentioned earlier), report writing where different people need different reports, and graphics programs—in short, any application in which you need to add functionality across a hierarchy without disrupting (or even changing) the nodes in the hierarchy.

Visitor is like walking around a new company visiting all the teams and having them accept you (the introductions) and give you their impressions of your job (the visit). The Visitor pattern allows you to retain flexibility to add new methods at a slight cost: the reduction of encapsulation and the need for every visitor to know about all the different node types. It's not a one-size-fits-all pattern. It's optimal when the number of functionalities that you (might) have to add is significantly greater than the number of node types in your data structure. If the number of data types (node types) to be added is greater than the functions you'll need to add or you truly know that you won't need to add new functions very often, don't use this pattern, but for the other cases, you'll find Visitor to be an elegant solution. `</article>`

Previous design patterns presented in this series:

State Pattern

Command Pattern

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

SIMON ROBERTS

MIKALAI ZAIKIN

Quiz Yourself

More intermediate and advanced test questions

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

Answer 1
page 80

Question 1 (intermediate). The objective is to create and overload constructors and to differentiate between default and user-defined constructors. Given three classes tied by inheritance:

```
public class Buffer {
   int capacity = 10;
   String name = "Unknown";
   public Buffer() { super(); }
}

public class NamedBuffer extends Buffer {
   protected NamedBuffer(String name) {
      this.name = name;
   }
}

public class HugeNamedBuffer extends NamedBuffer {
   public HugeNamedBuffer(String name) {
```

```
        this.name = name;
        this.capacity = 1000;
    }
}
```

Which statement is true?

A. Compilation fails at the `Buffer` class.
B. Compilation fails at the `NamedBuffer` class.
C. Compilation fails at the `HugeNamedBuffer` class.
D. Compilation succeeds.

Answer 2
page 83

Question 2 (intermediate). The objective is to determine the effect upon object references and primitive values when they are passed into methods that change the values. Given these two classes:

```
public class Message {
    public String text;
    public int code;
    public Message(String text, int code) {
        this.text = text;
        this.code = code;
    }
}

public class Logger {
    static public void log(final Message msg, boolean status) {
        // ... writing message to the log file
        msg.code = 0; // line n1
        status = true;
    }
}
```

And given the following code fragment:

```
Message m = new Message("Critical error", 255);
boolean handled = false;
Logger.log(m, handled);
System.out.printf("Error code: %d, was handled: %b\n", m.code, handled);
```

What is the result?

A. `Error code: 0, was handled: true`

B. `Error code: 255, was handled: false`

C. `Error code: 0, was handled: false`

D. Compilation fails because of line n1.

Answer 3 page 86

Question 3 (advanced). The objective is to create and use `ArrayList`, `TreeSet`, `TreeMap`, and `ArrayDeque` objects. Given this class:

```
public class Item {
   private String name;
   Item(String name) {
      this.name = name;
   }
   public String toString() {
      return name;
   }

   @Override
   public int hashCode() {
      return name.hashCode();
   }
  @Override
   public boolean equals(Object o) {
```

```
        return name.equals(((Item)o).name);
    }
}
```

And given this code fragment:

```
Item i1 = new Item("2");
Item i2 = new Item("3");
Item i3 = new Item("1");
TreeSet<Item> ts = new TreeSet<>();
ts.add(i2);
ts.add(i1);
ts.add(i3);
System.out.print(ts);
```

What is the result?

A. `[1, 2, 3]`
B. `[2, 3, 1]`
C. `[3, 2, 1]`
D. The result is a representation of the items 1, 2, and 3, but the items are in an undetermined order based on the hashcode values of String objects.
E. No output and runtime exception is thrown.

Answer 4
page 88

Question 4 (advanced). The objective is to use the `Files` class to check, read, delete, copy, move, and manage metadata of a file or directory.

Given the text file `text.txt`, which contains:

```
AAA
BBB
CCC
```

Assuming that the file is in the default working directory and is accessible and that, in general, `p` correctly refers to the file after this statement executes:

```
Path p = Paths.get("text.txt");
```

Which of the following three lines, if run independently, will print the content of the text file to the console?

A. `Files.lines(p).forEach(System.out::println);`
B. `Files.lines(p).stream().forEach(System.out::println);`
C. `Files.readAllLines(p).forEach(System.out::println);`
D. `Files.readAllLines(p).stream().forEach(System.out::println);`
E. `Files.list(p).forEach(System.out::println);`

Question 1
page 76

Answer 1. The correct answer is option C. This question investigates the process of object creation, object initialization, and, in particular, the rules for passing control from one constructor to a constructor of a parent class. Of course, the Java compiler checks that the code conforms to the requirements.

The `Buffer` class is syntactically sound. The constructor provided and its call to `super();` are correct. The explicitly provided constructor is exactly what the compiler would have inserted if no constructors had been provided in the source code.

A *default constructor* is a zero-argument constructor inserted by the compiler. Such a constructor passes control directly to the parent class constructor (`Object`, in this case) with a call to `super()`, and it does nothing else. It has the accessibility of the class as a whole (or it is private in the case of enum types). A default constructor is created by the compiler in the specific situation

of a class that has no constructors in its source code. You might think that coding such a constructor, as in the `Buffer` class in this question, is pointless. However, default constructors are a kind of emergency fallback; as soon as any constructor is provided explicitly, the default constructor goes away. That can be surprising. From a stylistic perspective, it probably makes more sense to entirely avoid the default constructor. If you want to be able to create an object with no explicit initialization, write that constructor deliberately. If you don't (and incompletely initialized objects are probably bad for code reliability anyway), don't let the compiler create it in the first place.

The call to the superclass constructor must happen in all classes except `Object`. You don't always have to code it, because if no explicit call to either `super` or `this` is in the code, the compiler inserts the call to `super()` with zero arguments (exactly as is coded explicitly in the `Buffer` class).

Given this information, you know that the `Buffer` class compiles correctly, and option A is incorrect.

During object initialization, the object's memory is first allocated (for the entire object, with data space allocated as needed for all the fields of all the classes in the hierarchy). Then the initialization process starts by passing control up through the inheritance hierarchy to the `Object` that is always found at the top of the hierarchy. This process is performed by the `super` calls at the start of every constructor. (Note that a call to `this` might replace a call to `super`, but these amount to sidesteps in the same class, and before any real initialization happens in this class, a call to `super` will definitely take control up the hierarchy.) When control reaches the top (`java.lang.Object`), the body of the `Object` constructor is executed, and then control returns down to the previous constructor. That constructor is executed and returns to the next class down the hierarchy. This process continues all the way back to the constructor on which `new` was invoked. Finally, when that constructor completes, you have an initialized object.

As a side note, no matter how many constructors are invoked in the hierarchy, this process initializes the single object of the class on which `new` was called. Of course, other objects might be created during this process by explicit calls to `new` on other classes. The essential observa-

tion here is that a single call to `new` might result in many constructors being executed, but those executions are all working to initialize a single object.

Therefore, one of the following three situations will occur in the very first line of any constructor, and that situation determines execution flow:

- If the first line is an explicit call to another *overloaded* constructor in the same class (written as `this(...);`), execution sidesteps to that constructor.
- If the first line is an explicit call to a superclass constructor (written as `super(...);`), execution moves up to that superclass constructor.
- If the first line is neither `this(...)` nor `super(...)`, a compiler-generated call to the zero-argument constructor in the parent class is invoked.

In the `NamedBuffer` class, there's no explicit call to `this(...)` or `super(...)`. Therefore, the third situation described above applies. The resulting call to the zero-argument constructor is successful, because such a constructor is defined (explicitly, in this case) in the parent class `Buffer`. Because of this, you can determine that the `NamedBuffer` class compiles successfully and option B is incorrect.

The constructor of the `HugeNamedBuffer` class also follows the third situation described above—that is, neither `this(...)` nor `super(...)` is mentioned. Therefore, just as with the `NamedBuffer`, the compiler will insert `super();` implicitly. This results in generated bytecode equivalent to this source code:

```
public HugeNamedBuffer(String name) {
  super(); // added by compiler
  this.name = name;
  this.capacity = 1000;
}
```

Of course, the parent class (`NamedBuffer`) has one explicit constructor in the source code. The existence of that constructor prevents the compiler from generating a default, so this is the only constructor available for initialization of any `NamedBuffer` or the `NamedBuffer` elements of

any subclass of `NamedBuffer`. That constructor requires a single String argument. Because of this, when the code in the `HugeNamedBuffer` attempts to call a zero-argument constructor, none exists, and compilation fails. Therefore, option C is correct and option D is incorrect.

Question 2
page 77

Answer 2. The correct answer is option C. This question investigates how values are passed into methods. This topic is the essence of the Associate exam objective to determine the effect upon object references and primitive values when they are passed into methods that change the values.

Unlike some other languages, Java is not a "pure" object-oriented language in that eight data types, known as the primitive data types, are handled not as objects but as simple values. Those types, by the way, are `boolean`, `byte`, `short`, `char`, `int`, `long`, `float`, and `double`. These types all have names that are entirely lowercase; don't confuse them with their object wrapper counterparts (`Boolean`, `Byte`, `Short`, `Character`, `Integer`, `Long`, `Float`, and `Double`). This distinction is not just esoteric; it has a significant practical impact on how the language works and the effects that you might see when passing data into method calls.

When you call a method, it's often necessary to pass information from the caller to the method. How that happens can have important consequences. Consider this analogy. Imagine a chef—let's call her Rachel—who is preparing a breakfast banquet. Rachel has numbered file cabinets in her office, and one of those cabinets contains her overall plan for the banquet, including a uniquely numbered sheet on which she has written the quantity of pancakes required. Rachel wants Jamie, one of her assistants, to make the pancakes. Rachel might tell Jamie how many pancakes to make in either of two distinct ways. She could copy the quantity of pancakes from her files onto a new piece of paper and hand that paper to Jamie. Alternatively, she could write the file cabinet and sheet numbers on a piece of paper, hand that information to Jamie, and invite Jamie to go into the file when she needs to know what to do.

Notice that these two techniques have different consequences if either Rachel or Jamie decides to change the quantity of pancakes.

If Jamie has the quantity written on her own paper, there are two copies of the num-

ber, and if either of the cooks writes a new number on her own copy, the other cook's copy remains unchanged.

However, if Jamie goes directly to Rachel's files, then either of them changing the value will cause both of them to see the new number—if they look at the file after the change is made.

These two forms are exact analogies to how data can be passed into methods in Java. Furthermore, the consequences of the visibility of changes are the same.

In Java, the value of a variable is always passed by copying that value into a new variable that is local (or exclusive) to a particular method invocation. (Indeed, if you call the same method three times, three separate copies will have been created.) This is called *pass by value* or *pass by copy*. It is equivalent to Jamie getting a piece of paper with something written on it. However, what a "variable" is, and therefore the significance of what is written on the paper, depends on the data type.

A variable of primitive type contains the actual value represented. But for any data that is of object type—that is, anything except one of the eight primitive types—the value of the variable describes *how to find* the data. That is, the variable contains information equivalent to the filing cabinet and sheet number where the information can be found.

In the `log` method, two arguments are passed. One, `Message`, is an object. The other is a primitive `boolean`. From the analogy, this means that for `Message`, the `log` method has instructions on where to find the data and that—potentially—any changes it makes to that data would alter the original value seen after the `log` call. However, for the `status` value, the `log` method gets a copy of the `true` or `false` value, and it has no ability to interact with the original data in the caller. Therefore, it has no chance of changing the value that is printed by the `printf` call after it returns.

So, at this point you can see that the assignment in line n1 would change the message `code` to zero (if it works at all); therefore, any final output message cannot include the error code `255`. This means that option B must be incorrect.

This also means that the assignment to `status` immediately after line n1 affects the local copy of the `status` value and cannot change the value of `handled` in the caller. Importantly, this would still be the case even if the variable were called `status` in the caller. These are entirely

distinct variables (this would also be true of `m` and `msg`). Because of this, the final output message cannot contain the output `true`, and option A must be incorrect.

Java is not a "pure" object-oriented language in that eight data types, known as the primitive data types, are handled not as objects but as simple values.

An important issue that's not yet been investigated is the effect of the modifier `final` on the `Message` argument. In Java, the modifier `final` means that the value of the affected *variable* must be assigned exactly once and cannot be changed afterward. Why is *variable* emphasized in the preceding sentence? Because a variable that has an object type does not *contain* the object; it contains *instructions on how to find* the object (this is what's meant by a *reference*). So, when it is said that a `final` variable of object type can never be changed (and it cannot!), what this means is that the variable cannot refer to a *different* object. However, it does *not* mean that the contents of the object to which it refers can't be changed. The mutability of the object is a completely separate issue that depends on how the class is defined (for example, all the primitive wrapper classes, such as `Integer` and `String`, are immutable, but a `StringBuilder` is mutable). In this case, the class has nonfinal fields, and it is entirely mutable. In view of this, you can determine that there's no problem with the assignment at line n1, and there's no reason to expect compilation to fail. Therefore, option D is incorrect and, in fact, option C is correct.

A final side note is that although you can modify the contents of the original object via the passed-in reference, you cannot replace (substitute) the whole object in the caller. In the analogy, when Jamie is given instructions on how to find the information in the cabinet, she gains the ability to modify the contents of the file. However, Jamie doesn't have the ability to change the file in which Rachel will look when she is hunting for her information. To do that, Jamie would have to write on the notes that are on Rachel's desk. Those notes tell Rachel which cabinet and sheets contain the plans for the banquet. Jamie never gets to modify Rachel's variables; she always gets a copy of them. But, if the variable refers to a filing cabinet, Jamie can change (and see changes in) the contents of those filing cabinets.

In Java, all arguments to method calls are pass by value. That is, the value of the argument variable is copied into a new variable for use by the method. However, the value of a variable of object type is itself a *reference*, so the method has access to the original object through that reference.

Question 3
page 78

Answer 3. The correct answer is option E. The exam objectives include determining whether exam takers have knowledge of several elements of the Collections API, including the `TreeSet`. Two things are significant in the word `TreeSet`. A `Set` rejects duplicate entries, but in this question no duplicates are offered. The second thing is that a tree structure facilitates fast data searches by putting the data items in locations based on their order. For this to happen, there must be an order associated with the items being stored.

In a Java `TreeSet`, this ordering is determined when the collection is initially built, and it can be supplied in either of two ways:

- Using the natural order of the items. A "natural order" is defined in the Java API documentation as being the order implicit in the object's implementation of the `java.lang.Comparable` interface. The `Item` class in this question does not implement that interface.
- Using an externally provided ordering. Such an order is provided via an implementation of the `java.lang.Comparator` interface. As already mentioned, the order is determined at the time the collection is constructed; therefore, the comparator must be supplied as a constructor argument. In this example, no such argument is provided.

Because neither of the options for specifying order is used in this question, the tree cannot work properly. So, how might this failure be visible?

There are three possibilities: the code doesn't compile, the code fails to store items in a rapidly retrievable way (perhaps storing them randomly and then failing to find them), or an exception is thrown. In fact, after a `TreeSet` has been created without a `Comparator` to use for ordering, any attempt to add an object that does not implement the `Comparable` interface will cause the `TreeSet.add` method to throw a runtime exception. Because of this, option E is correct.

Let's examine the background for some of the other options.

Clearly, if `TreeSet` is expected to put items into order, then option A is tempting, because it

presents the items in alphabetical order. Indeed, if `Item` implemented `Comparable<Item>` based on the `String` contents of `Item`, that is the output you would see. The provision of the `equals` and `hashCode` methods based on that `String` value is distracting and further nudges unwary exam takers toward this wrong interpretation. But option A is incorrect.

Given that `Item` implements `equals` and `hashCode` based on the `String`, it could be tempting to favor the very verbose option D. However, while this might be true for a `HashSet` that contained these `Item` objects, it's not applicable here. Further, the hashcode value of `String` objects formed from single digits is actually the character code of those digits, and those run in monotonically increasing order (with character `1` having the value 49 and characters `2` and `3` having the values 50 and 51, respectively). Therefore, if the assumptions that might tempt you to select option D were not incorrect anyway, the correct answer would in fact have been option A, not option D. But both option A and option D are incorrect.

The alternate ordering suggested by option B is tempting because it's the order in which the `Item` objects are initialized, but this is irrelevant. They are not added in the order in which they're created, and unlike with a `List`, the order of adding isn't significant to a traditional `Set` anyway. Therefore, option B is incorrect.

The order suggested by option C might be tempting if either you expected the descending order to somehow be natural or you fell for the distraction of the hashcode while simultaneously believing the hashcode order for `String` objects formed from single digits to be in descending order. But option C is incorrect.

A distractor answer that was not offered could have suggested a compiler error. Indeed, it's interesting to consider why Java doesn't reject the use of the zero-argument `TreeSet` constructor to build a `TreeSet` of some object that doesn't implement the `Comparable` interface. The problem is that this situation falls through a "crack" in Java's generics system. The generics mechanism allows you to constrain a generic type to require that its type argument implement an interface such as `Comparable`. For `TreeSet`, that would be a declaration along the following lines:

```
public class TreeSet<E extends Comparable<E>>
```

However, this constraint is not appropriate for `TreeSet`, because it's OK to have non-`Comparable` contents, as long as a `Comparator` is supplied at construction time. This level of conditional complexity cannot be expressed by the generics mechanism.

It's also not possible for the generics mechanism to perform a compile-time check on the argument to the `add` method, because the `add` method is declared in the `Set` interface, where no such constraints apply. As a result, the Java compiler cannot analyze the "correctness" of this use of `TreeSet`, and the following error occurs at runtime, specifically at `ts.add(i2)`:

```
Exception in thread "main" java.lang.ClassCastException: Item cannot be cast to
java.lang.Comparable
  at java.util.TreeMap.compare(TreeMap.java:1290)
  at java.util.TreeMap.put(TreeMap.java:538)
  at java.util.TreeSet.add(TreeSet.java:255)
  at TreeSetDemo.main(TreeSetDemo.java:44)
```

Question 4
page 79

Answer 4. The correct answers are options A, C, and D. This is one of those rather frustrating questions for which you must simply know some API methods. Such "rote learning" questions do exist on real exams, and although they're kept to a minimum, you will obviously score better if you are able to answer them. Generally, such questions are focused on frequently used methods (which a regular practitioner would likely have learned in day-to-day usage) or particularly interesting methods. The methods used here are both very useful and interesting, and with a little luck you'll be glad you learned them.

The Java 8 release introduced many functional features to the language and the APIs. The Stream API and the addition of lambda expressions to the language are probably the most noticeable, but other additions in both the language and APIs exist.

One change that is frequently used is *internal iteration*. In traditional iteration, you write a loop, and in that loop you write code that extracts one item at a time from some kind of collection and then performs an operation with the extracted item as an argument. Internal iteration changes this sequence, and instead behavior is passed into the collection. The collection then

performs the iteration and applies the provided method to each item in turn. This is the job of the `forEach` methods.

A `forEach` method exists in both the `Iterable` interface of the collections API and in the various Streams interfaces (both the regular object stream and the three primitive variants). This question investigates how to obtain the appropriate interface implementations and then use these methods.

The signatures of both the `forEach` methods are the same. They take a `Consumer<? super E>`, where `E` is the element type in the `Iterable` or `Stream`. The return type is `void`, and this means that when it is called on a stream, `forEach` is a "terminal operation." `Consumer` is one of Java's standard functional interfaces. It accepts one argument and returns `void`.

If the `forEach` method is to be valid, the target object must be either an `Iterable` (such as a `List` or other a collection) or a `Stream`. Also, the argument type provided to the `Consumer` must be applicable to the item type. In this case, all the `Consumers` have the same `println` behavior, and because `println` can accept any type, you can safely assume that no problems will arise because of type constraints in any of the options.

Let's look at the options in turn.

In option A, the subexpression `Files.lines(...)` has the type `Stream<String>`. That stream will produce all the lines in the text file. From the perspective of the question, however, because it's a `Stream`, you can call `forEach` on it. Therefore, option A is correct.

The expression in option B attempts to invoke a `stream()` method on the stream expression just investigated for option A. Needless to say, there is no `stream` method on a `Stream`, so option B will not compile and is incorrect.

Option C illustrates another useful method on the `Files` class. `Files.readAllLines(p)` reads all the data from the file into memory and creates a `List<String>`. That list and, therefore, the entire contents of the file, are returned to the caller. In contrast, the `lines` method used in option A is generally lazy, as most stream methods are. This means that while it probably reads chunks of data into a buffer, the `lines` method should not read anything until some data is requested downstream. Further, it should read the entire file only if that file is smaller than the

buffer size. Of course, the `java.util.List` returned from the `readAllLines` method implements `Iterable`; therefore, you can call `forEach` on it and print all lines. Notice this `forEach` is a totally different method from the one on `Stream`, but its external behavior is identical. Given this, you can determine that option C compiles successfully and is correct.

In option D, the initial subexpression `Files.readAllLines(p)` is the same as in option C. Therefore, this is an expression of `List<String>` type. Because you can extract a `Stream` from a `List`, option D is also correct.

Notice that the code in option D is definitely not optimal: First, it draws the entire file into memory (which might be a problem with a large file). Then it wraps a `Stream` around that `List`. This approach takes extra effort, uses the maximum memory possible for this file, and then makes a `Stream`. Going directly to the `Stream` would reduce memory use, create simpler code, and still support all the operations you might want to perform using the Stream API.

The `Files.list()` method shown in option E does not supply the contents of a file. Instead, it returns a listing of the contents of a directory. The contents are provided in the form of a stream, specifically `Stream<Path>`, which makes this a very useful method too, and the code will compile successfully—although it throws an exception at runtime—because the `Path` that is provided refers to a file, not to directory. Because of this, option E is incorrect. `</article>`

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

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

//contact us/

Comments

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

Article Proposals

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

Customer Service

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

Where?

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

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

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