// MARCH/APRIL 2015 /

Java™ magazine

By and for the Java community

ACTION!

Netflix powers through 2 billion content requests per day with Java-driven architecture

+

ORACLE.COM/JAVAMAGAZINE

ORACLE®

//table of contents /

EDITORIAL

Interim Editor in Chief
Kay Keppler

Community Editor
Yolande Poirier

Java in Action Editor
Michelle Kovac

Technology Editor
Tori Wieldt

Contributing Writer
Kevin Farnham

Contributing Editors
Claire Breen, Blair Campbell, Stephen Chin, Kay Keppler, Karen Perkins, Reza Rahman, Simon Ritter, James Weaver

DESIGN

Senior Creative Director
Francisco G Delgadillo

Design Director
Richard Merchán

Contributing Designers
Jaime Ferrand, Diane Murray, Arianna Pucherelli

Production Designers
Sheila Brennan, Kathy Cygnarowicz

PUBLISHING

Publisher
Jennifer Hamilton +1.650.506.3794

Associate Publisher and Audience Development Director
Karin Kinnear +1.650.506.1985

Audience Development Manager
Jennifer Kurtz

ADVERTISING SALES

President, Sprocket Media
Kyle Walkenhorst +1.323.340.8585

Western and Central US, LAD, and Canada, Sprocket Media
Tom Cometa +1.510.339.2403

Eastern US and EMEA/APAC, Sprocket Media
Mark Makinney +1.805.709.4745

Advertising Sales Assistant
Cindy Elhaj +1.626.396.9400 x 201

Mailing-List Rentals
Contact your sales representative.

RESOURCES

Oracle Products
+1.800.367.8674 (US/Canada)

Oracle Services
+1.888.283.0591 (US)

Oracle Press Books
oraclepressbooks.com

ARTICLE SUBMISSION

If you are interested in submitting an article, please e-mail the editors.

SUBSCRIPTION INFORMATION

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

MAGAZINE CUSTOMER SERVICE

java@halldata.com **Phone** +1.847.763.9635

PRIVACY

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

Java Magazine is published bimonthly with a free subscription price by Oracle, 500 Oracle Parkway, MS OPL-3C, Redwood City, CA 94065-1600.

Digital Publishing by GTxcel

Continually improving Java performance is more important than ever. Not only does the platform continue to evolve, but cloud computing magnifies the effect of a single CPU. From configuring the Java Virtual Machine (JVM) to using best practices in your application code, you have lots of ways to improve performance—and in this issue, we offer several ideas.

Java and Performance

We begin with an analysis from Kirk Pepperdine, who examines when adding parallelism using the fork/join framework might help performance. Michael Heinrichs, in "The Quantum Physics of Java," looks at modern chip design and its effect on Java programs. Chris Newland discusses the effects of small source code changes and Java HotSpot VM switches in Java just-in-time (JIT) compilation, and Josh Juneau explores incorporating performance tuning into the development lifecycle. Our JCP Executive Series features Anil Kumar, a performance architect at Intel. And check out the big data Q&A: a panel from JavaOne 2014 answers the audience's most pressing questions.

Finally, our cover story showcases Netflix, which fulfills 2 billion content requests every day. Now that's what I call performance! The company can do so because most of the services running in its architecture are built on Java and the JVM.

On a personal note, I've been filling in during the search for a new and permanent editor in chief. I want to thank the regular staff, who helped me over the rough spots and worked especially hard to get this issue out. And now I'm pleased to announce that Andrew Binstock, a veteran technology journalist most recently from *Dr. Dobb's Journal*, will take over the helm of *Java Magazine*. Welcome, Andrew!

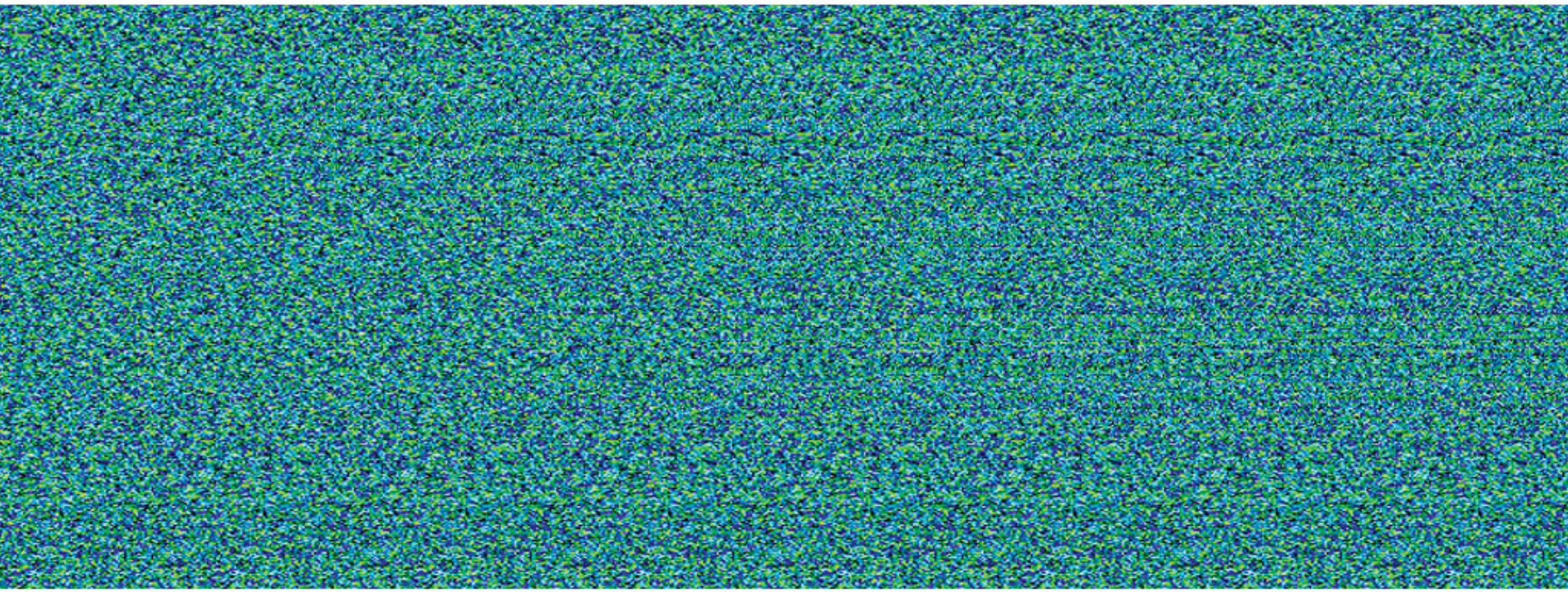
Kay Keppler, Interim Editor in Chief BIO

//send us your feedback /

We'll review all suggestions for future improvements. Depending on volume, some messages might not get a direct reply.

PHOTOGRAPH BY BOB ADLER

The answer is right in front of you



Java Image Enabling SDKs that Help You See the Big Picture

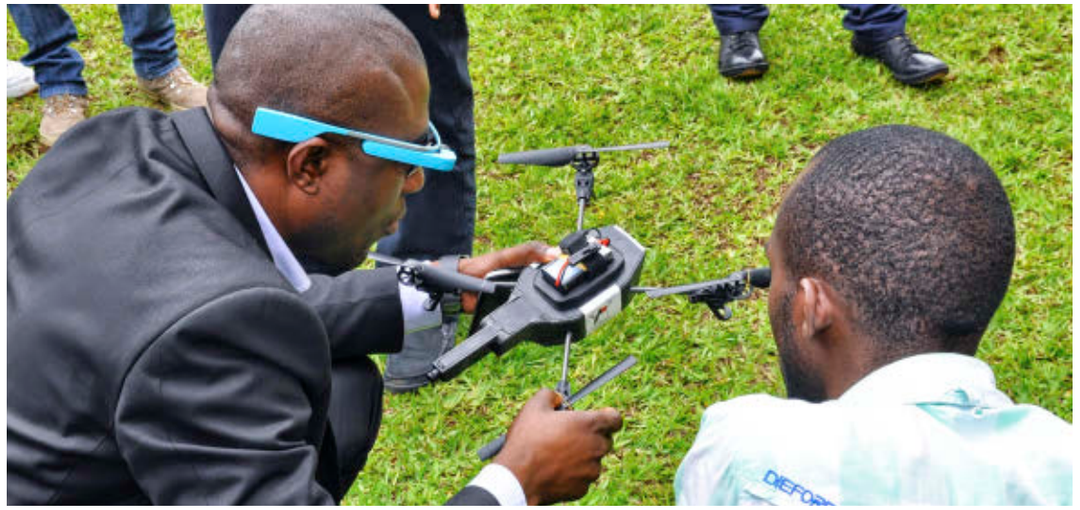
At first glance it may seem difficult, but it's really quite simple. Atalasoft's JoltImage product is a proven SDK for image enabling your Java-based web applications, easily. Image enabling helps to add dimension to your data, so you can uncover insights such as correlations and causations hidden inside your 2-dimensional documents. Our SDK does the heavy lifting for you, saving time, money, and the headaches of figuring it out yourself. Backed by our highly knowledgeable & caffeinated support engineers, JoltImage will enable your success and make the big picture so much easier to see.

Atalasoft JoltImage

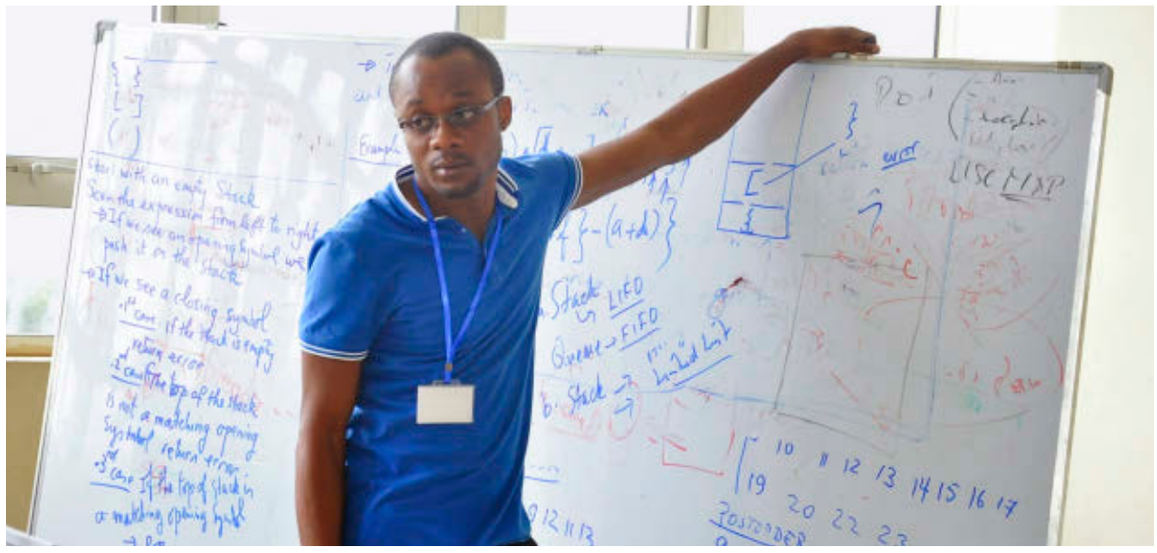
Click for tips on viewing the stereogram

Image enabling experts & bacon connoisseurs. Visit us online to see our full line of SDK products for .NET and Java

Atalasoft
A Kofax Company
www.atalasoft.com



The JCertif Kigali Java developer conference was a huge success, sharpening the skills of hundreds of Java programmers. JCertif International has plans to expand its events to more African nations.



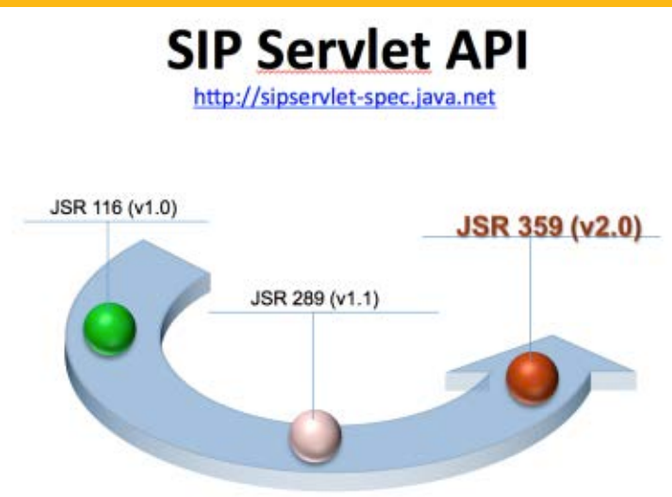
JCERTIF
TRAINING DEVELOPERS IN AFRICA

A winner of the 2014 Duke's Choice Awards, JCertif International has trained more than 5,000 new Java developers across Africa since its start in 2010. JCertif Kigali and JCertif Tunisia were just its latest great successes. JCertif Kigali was the leading developer community conference in Rwanda. Founder and Managing Director **Max Bonbhel** says, "We're still digesting the great number of Likes collected during the event." The events are two- to five-day Java workshops for groups of 20 to 50 developers. JCertif is planning a series of events in 2015 including in Cameroon, Benin, Senegal, the two Congos, Ivory Coast, and Rwanda.

PHOTOGRAPHS COURTESY OF JCERTIF INTERNATIONAL

FEATURED JAVA SPECIFICATION REQUEST

JSR 359: SIP SERVLET 2.0



JSR 359: SIP Servlet 2.0 is all about advancing the state of Session Initiation Protocol (SIP) in the Java ecosystem. SIP is a communications protocol for signaling and controlling communications sessions over Internet Protocol (IP) networks.

Oracle's **Binod PG**, the Specification Lead for JSR 359, describes the objectives of this JSR as "improving the RFC coverage of SIP specification (SIP outbound), modernizing programming (using POJOs), better alignment with Java EE (CDI, latest servlet specification), SIP over websockets, extensibility, better concurrency control, updates to B2B, and so on."

The JSR 359 Expert Group includes representatives from small and large companies, including AT&T, Ericsson AB, IBM, Cisco Systems, Thrupoint, OpenCloud, and TeleStax, all of which are seeking greater alignment between SIP servlets and Java EE.

If you'd like to follow the progress, comment on the work, or contribute new suggestions, visit the JSR 359 project at Java.net. There, you'll find a variety of ways you can share your ideas and get involved.

YOUNG DEVELOPER

Hania Guiagoussou

Guiagoussou holds her prize from the SITIC Innovation Competition.

California high school student **Hania Guiagoussou** became a passionate Java developer after attending a Java programming summer workshop at Oracle two years ago. "If it weren't for Alice, I wouldn't be interested in programming," she explains in an interview. Her most recent project, a "Water Saver System," tracks the need for watering using humidity and temperature sensors in the soil. She and her teammate won awards at a local engineering and science fair as well as third place in the Computer Science, Math, and Engineering category of awards sponsored by the local utility company. She also won third place and US$10,000 in the Digital Innovative Challenges, an African competition.

FEATURED JAVA USER GROUP

ISRAEL JAVA USER GROUP

The Israel Java User Group (Java.IL) was formed in mid-2014 as a consolidation of three smaller user groups: HaBotzia (founded by **Asaf Mesika** and **Uri Shamay**), JJTV (founded by **Ron Gross** and **Tomer Gabel**), and ILJUG (founded by **Haim Yadid**).

Gabel notes that "the idea for combining the groups originated with Haim, whose efforts to bootstrap ILJUG led to the realization that community thrives on diversity, and that each of the three groups seems to operate in its own silo. In consolidating the groups we've increased our reach, and having a much larger organizational body enables more frequent, diverse events."

The combined Java.IL group had 540 members at the start of 2015. "Our membership spans the entire Israeli software landscape, which includes young startups, established enterprises, and consulting firms," Gabel says. Between 50 and 100 members participate in monthly meetups.

Gabel describes Java.IL meetups as focusing largely on "production war stories...whether it's an alternative language (for example, Clojure), a brand-new in-memory database, or an amazing REST documentation framework, we seek and promote speakers with hands-on production experience. Sharing the real caveats can save an immense amount of time for participants who are often afraid to take the plunge without knowing the risks."

The Java.IL leadership takes a genuine interest in career development among the community, including developing presentation skills. "While we strive to get as many people as possible from the Israeli community on stage to showcase their projects, not all of us are accomplished public speakers," says Gabel. "We therefore offer guidance and assistance in everything from initial preparation and abstract, through developing a slide deck, to providing an opportunity for dry runs of the lecture."

Although Java.IL is relatively new, the group is already involved in regional events. For example, Java.IL organized the back-end track at CodeMotion 2014 in Tel Aviv, receiving 30 submissions from the community in just two weeks. "This makes us immensely proud of our membership," says Gabel.

The group is also involved in the Adopt OpenJDK effort. Java.IL members Haim Yadid and **Anat Levinger** developed and contributed mjprof (the "monadic JVM profiler"), a command-line tool for analyzing series of Java stack traces.

Updates, recommended links, and other tidbits are regularly published on the Java.IL Twitter channel (@Java_IL). The first issue of the Java.IL monthly newsletter has been published.

And there's much more to come. According to Gabel, "We have high plans for 2015!"

PHOTOGRAPHS BY ILAN SHACHAM/GETTY IMAGES AND TOMER GABEL

Virtual Technology Summit Global Series

Oracle Technology Network organizes the Virtual Technology Summit (VTS), an interactive online event that takes place four times a year. The presenters are Java Champions and Oracle product experts with hands-on experience. In the last summit, the Java track showed how to get the most out of Java wherever it is: from the largest servers to the tiniest devices. Technical sessions focused on how to improve the compatibility and security of large applications; how to work with Maven, HTML5, and JavaScript; how to get started with the Internet of Things (IoT) and wearables; and more. Learn how to take advantage of Java's depth and breadth. Check out the next event. VTS is also available on demand.

JAVA CHAMPION PROFILE

MARKUS EISELE

Markus Eisele *is a software architect, developer, and consultant. He works daily with customers on enterprise Java projects. Eisele became a Java Champion in March 2014.*

Java Magazine: Where did you grow up?
Eisele: In Germany, between Hannover and Paderborn.
Java Magazine: When did you first become interested in computers?
Eisele: Early on, when my parents bought me a Schneider CPC464, running CP/M 2.0 and Basic 1.0, with a cassette recorder for storage. One of my first programs was a patient and appointment registry for a physical therapy practice. Thousands of lines of code and GOTOs!
Java Magazine: What was your first professional programming job?
Eisele: I developed a vacuum cleaner bag finder on the internet. You entered your vacuum brand and model, and it spit out the fitting bag type from the product line.
Java Magazine: What do you enjoy for fun and relaxation?
Eisele: I'm most relaxed and happy in front of a computer. But I enjoy social interaction as well.
Java Magazine: What happens on your typical day off work?
Eisele: Since I love my work, there's no real "day off." I catch up with news from the Java community and my JBoss peers daily. If the weather is fine, we try to make something exciting happen for the kids—theme parks or stuff like that.
Java Magazine: What side effects of your career do you enjoy the most?
Eisele: Traveling. I love experiencing countries, people, habits, and culture; talking about technology; and engaging with people. Twitter and blogs are fine, but traveling helps me understand the culture of countries, so I can work effectively with developers and customers there.
Java Magazine: Has being a Java Champion changed anything for you with respect to your daily life?
Eisele: Yes, mostly from Oracle's handling of the Oracle ACE and Java Champion community programs. They support us by maintaining a strong brand and helping us make the right connections.
Java Magazine: What are you looking forward to in the coming years?
Eisele: The second edition of JavaLand, which I want to be even bigger and better than the inaugural edition. I look forward to talking about the amazing open source communities that make up Red Hat's success, meeting new peers, and seeing their unique solutions. Also, I want my girls to have a great start in their new adventures: elementary school for Svenja and kindergarten for Anna.

You can find Markus Eisele on Twitter as @myfear, and you can visit his blog.

PHOTOGRAPHS COURTESY OF TDC 2014 BRAZIL/GLOBALCODE AND MARKUS EISELE

EVENTS

Devoxx France *APRIL 8–10*
PARIS, FRANCE
Devoxx France brings together more than a thousand Java enthusiasts. This year's event has a bigger venue to accommodate the growing number of attendees. The conference focuses on Java, Java Virtual Machine (JVM) languages, web technologies, mobile, cloud, and agile development. This year's theme, "Infinite Possibilities," is about Java innovation and the Java ecosystem.

JAX 2015
APRIL 20–24
MAINZ, GERMANY
This conference covers a range of current and future-oriented technologies, from Java and Scala to Android and web technologies and agile development models as well as DevOps. More than 200 international speakers present 230 talks and 20 day-long sessions.

Great Indian Developer Summit (GIDS)
APRIL 21–25
BANGALORE AND HYDERABAD, INDIA
Thirty thousand developers attend this five-day conference, held in Bangalore on April 21 through 24 and in Hyderabad on April 25. The main GIDS activities in Bangalore focus on cloud, mobile, dynamic languages, and analytics. The one-day GIDS. MINI conference in Hyderabad offers sessions that reflect the main conference. The dedicated Java track focuses on dynamic languages such as Scala, Clojure, Groovy, and JRuby, which run on the JVM.

GeeCON 2015
MAY 13–15
KRAKOW, POLAND
The three-day GeeCON 2015 focuses on Java and JVM-based technologies, dynamic languages, enterprise architectures, patterns, distributed computing, software craftsmanship, mobile, and much more.

JEEConf 2015
MAY 22–23
KIEV, UKRAINE
JEEConf is the largest developer conference in Ukraine. The annual conference focuses on Java technologies for application development.

Devoxx UK
JUNE 17–19
LONDON, ENGLAND
In its third year, Devoxx UK is organized by the local Java community. Its focus is on Java, web, mobile, and JVM languages. The conference is run by top developer talent, community groups, and expert event specialists.

JavaOne Brazil
JUNE 23–25
SÃO PAULO, BRAZIL
JavaOne Brazil is the learning opportunity for everything Java in Latin America, from Java 8 and the new features of Java 9 to the Internet of Things, JVM languages, and more. Meet Oracle and community experts in a fun atmosphere.

PHOTOGRAPH BY DENNIS JARVIS AT FLICKR

JAVA BOOKS

JAVA 8 IN ACTION: LAMBDAS, STREAMS, AND FUNCTIONAL-STYLE PROGRAMMING
By Raoul-Gabriel Urma, Mario Fusco, and Alan Mycroft
Manning Publications, August 2014
Java 8 in Action, a guide to the new features of Java 8, shows you how to write more-concise code in less time and automatically benefit from multicore architectures. It introduces lambdas using real-world Java code. It covers the new Stream API and shows how you can use it to make collection-based code easier to understand and maintain. It also explains other major Java 8 features, including default methods, Optional, CompletableFuture, and the new Date and Time API. Written for programmers familiar with Java and basic object-oriented programming, it demonstrates how Java 8 is a game changer.

JAVA EE 7 DEVELOPMENT WITH NETBEANS 8
By David R. Heffelfinger
Packt Publishing, January 2015
Learn how you can use the features of NetBeans to accelerate your development of Java EE applications. Covering the latest versions of Java EE APIs such as JavaServer Faces (JSF) 2.2, Enterprise JavaBeans (EJB) 3.2, Java Persistence API (JPA) 2.1, Contexts and Dependency Injection 1.1, and JAX-RS 2.0, *Java EE 7 Development with NetBeans 8* walks you through application development using JSF component libraries such as PrimeFaces, RichFaces, and ICEfaces. Find out how you can quickly and easily develop applications taking advantage of JPA, develop aesthetically pleasing JSF web applications, and develop SOAP-based and RESTful web services using NetBeans, including exposing EJB functionality as web services.

THE JAVA TUTORIAL: A SHORT COURSE ON THE BASICS, 6TH EDITION
By Raymond Gallardo, Scott Hommel, Sowmya Kannan, Joni Gordon, and Sharon Biocca Zakhour
Addison-Wesley Professional, December 2014
Based on Java SE 8, this updated edition introduces the new features added to the platform, including lambda expressions, default methods, aggregate operations, and more. An accessible and practical guide for programmers of any level, *The Java Tutorial* focuses on how to use the rich environment Java provides to build applications, applets, and components. Expanded coverage includes chapters on the Date and Time API, annotations, and the latest deployment best practices. Key security sections include "Security in Rich Internet Applications" and "Guidelines for Securing Rich Internet Applications."

CONTINUOUS ENTERPRISE DEVELOPMENT IN JAVA
By Andrew Lee Rubinger and Aslak Knutsen
O'Reilly Media, March 2014
Learn a use-case approach for developing Java enterprise applications in a continuously test-driven fashion. With this hands-on guide, authors and JBoss project leaders Rubinger and Knutsen show you how to build high-level components, from persistent storage to the user interface, using the Arquillian testing platform and other JBoss projects and tools.

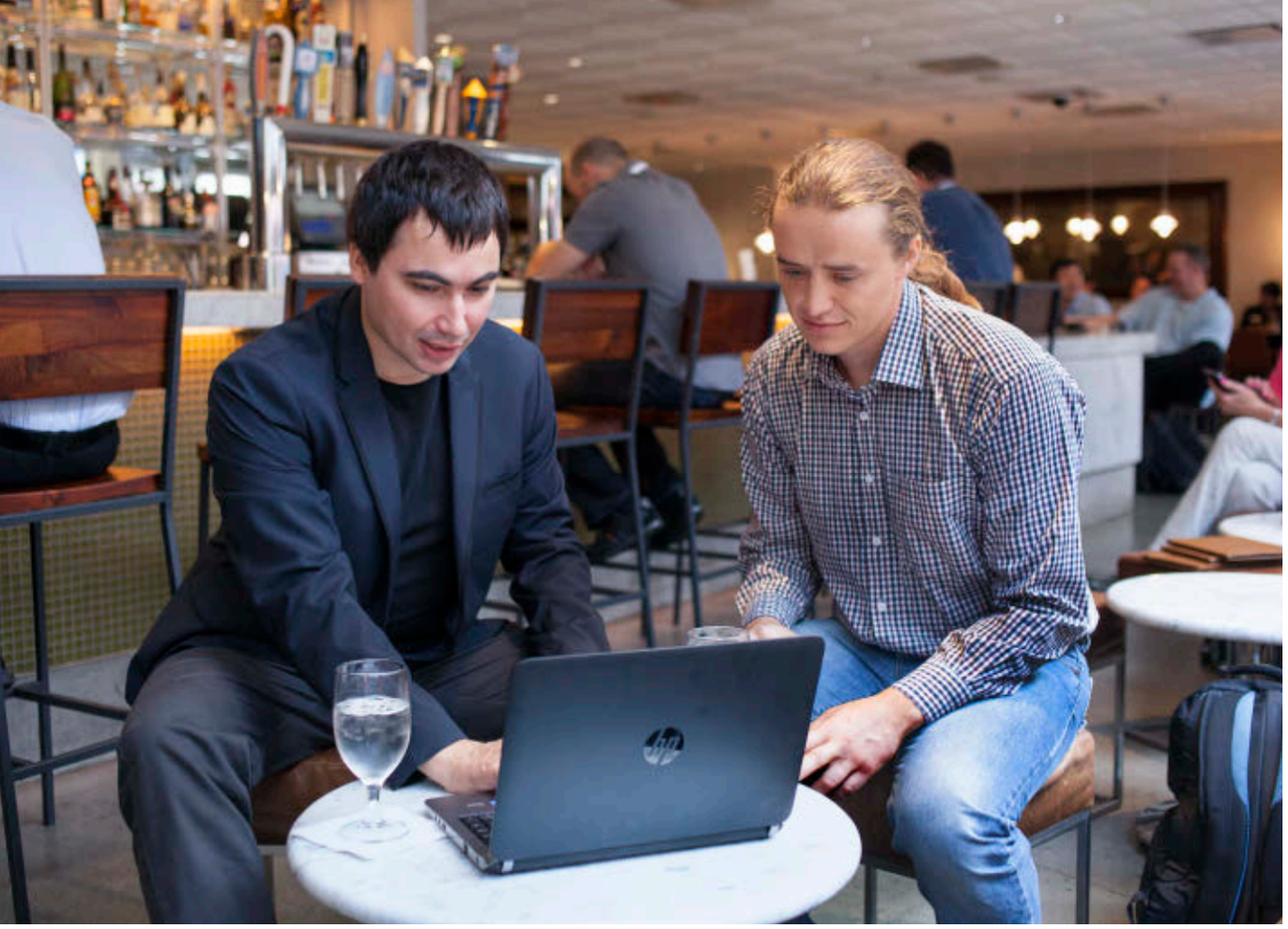
Reach More than 800,000 Oracle Customers with Oracle Publishing Group

Connect with the Audience that Matters Most to Your Business

Oracle Magazine
The Largest IT Publication in the World
Circulation: 500,000
Audience: IT Managers, DBAs, Programmers and Developers

Profit
Business Insight for Enterprise-Class Business Leaders to Help Them Build a Better Business Using Oracle Technology
Circulation: 100,000
Audience: Top Executives and Line of Business Managers

Java Magazine
The Essential Source on Java Technology, the Java Programming Language and Java-Based Applications
Circulation: 200,000 and Growing Steady
Audience: Corporate and Independent Java Developers, Programmers, and Architects



Smart Greenhouse IoT Developer Challenge winners Vladimir Redzhepov (left) and Dzmitry Yasevich

Click here to watch time-lapse photography of the beans growing.

IoT Developer Challenge Winner

Smart Greenhouse

Control the light, watering, temperature, and humidity of your greenhouse—automatically. **BY KEVIN FARNHAM**

Smart Greenhouse, one of three professional category winners in the 2014 IoT Developer Challenge, is an Internet of Things (IoT) device and application that monitors and controls a greenhouse environment. The concept for Smart Greenhouse came into being after the core team—Dzmitry Yasevich, Pavel Vervenko, and Vladimir Redzhepov—attended JavaOne Russia in April 2013. There, the team saw presentations of a smart house, various robots, and other devices, all controlled by Java.

Yasevich notes, "We were impressed by these solutions and had an idea to do something like that. Pavel Vervenko suggested making an automated greenhouse. Everyone liked the idea!"

First, the team selected the hardware. "We started to use Raspberry Pi as a basis," Yasevich says. "It is a compact but full-fledged computer with 700 MHz and memory at 512 MB. This system costs around $35."

However, early on, a safety concern arose. "Current under high voltage passes in the greenhouse, and there is an automatic watering system, so it was necessary to properly consider all the aspects related to insulation," Yasevich says.

As the software was being implemented, a hardware driver difficulty arose. "We had a problem with Pi4J, because it does not have the necessary API to work with the DHT11 humidity sensor," says Yasevich. To solve this, the team used JNI/C, writing a custom driver for the sensor.

More people became involved, including intelligent-hardware expert Vasili Slapik, and Vladimir Redzhepov, who was instrumental in making an open source Linux-based distribution available for Smart Greenhouse. Yasevich says, "Any farmer can download this distribution, buy a Raspberry Pi and the necessary sensors, and get their own smart greenhouse into operation." Visit the Smart Greenhouse Dropbox, and download the source code at BitBucket. `</article>`

FAST FACTS

- Smart Greenhouse automates greenhouses, controlling light and watering, monitoring temperature and humidity, and providing live photography.
- The hardware consists of a Raspberry Pi, sensors, an Arduino expansion board, power supply, SD card, cooler, and web camera.
- The backing software is composed of Oracle Java SE Embedded, JNI/C, Pi4J, servlets, and a Jetty container with an HTML/JavaScript front end. Watch the video.

PHOTOGRAPH BY BOB ADLER/GETTY IMAGES

JRebel

ZEROTURNAROUND

RELOAD CODE CHANGES INSTANTLY

TRY IT NOW!

Get a free t-shirt!

Anil Kumar (left) and Mike Jones, of the Intel Software and Services Group, analyze a problem together. The goal is to enable a better user experience.

PHOTOGRAPHY BY BOB ADLER/GETTY IMAGES

JCP Executive Series

INTEL AND ORACLE: Performance, Innovation, and Success

Intel's Anil Kumar describes how collaboration drives a healthier Java ecosystem. **BY STEVE MELOAN**

Java and Performance

Anil Kumar, a performance architect in the Software and Services Group at Intel, plays an active role in the Java ecosystem. He contributes to several benchmarks and standards organizations—as Java chair for the Open Service Gateway (OSGi) specification, and as Intel's representative on the Java Community Process (JCP) Executive Committee. He enables better user experience and resource use through customer

applications, and identifies optimization opportunities and default performance for hardware and software configurations. Kumar has presented at JavaOne, SouJava Brazil, and the International Conference on Performance Engineering (ICPE) on topics related to Java optimizations, application latency, and system power consumption.

Java Magazine: What are the coengineering processes that Intel and Oracle have engaged in around performance—within the realms of big data, cloud, Internet of Things [IoT], and computationally intensive applications?

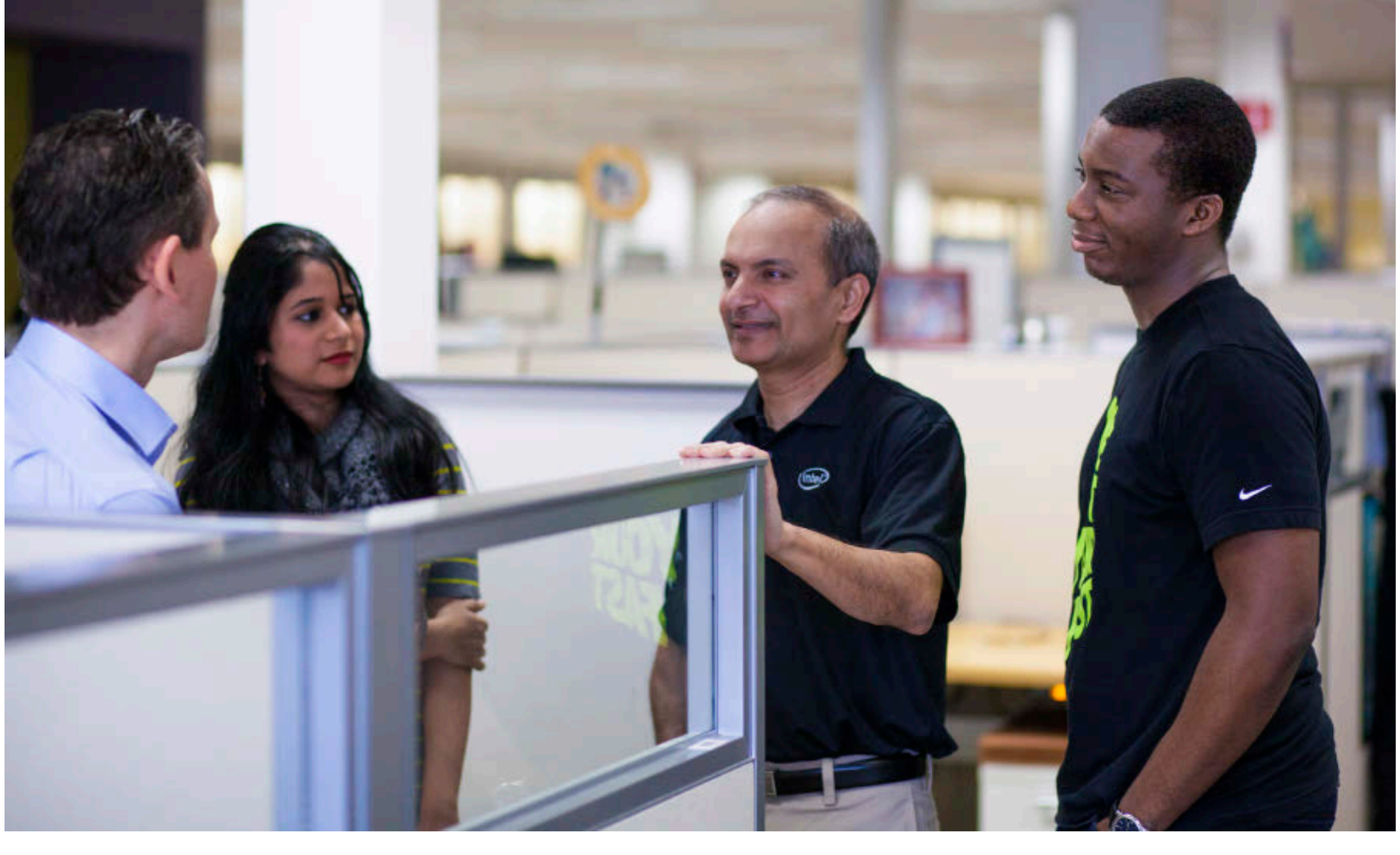
Kumar: The coengineering work between Intel and Oracle is multifaceted. These areas can be diverse, and they each have large ecosystems. As a result, we have multiple coengineering efforts, with teams working at varying levels. Some are focused at the component level, while others operate at the strategy level. But the core goal is to maximize performance and to seamlessly enable these ecosystems.

From Intel's perspective, *performance* means the complete end-user experience—response time, certainly, but also ease of use, and interoperability between the components within these environments. The computationally intensive application space is mature but still growing. This includes everything from weather prediction to graph simulations, chemical and atomic reactions modeling, life sciences, and statistical process control.

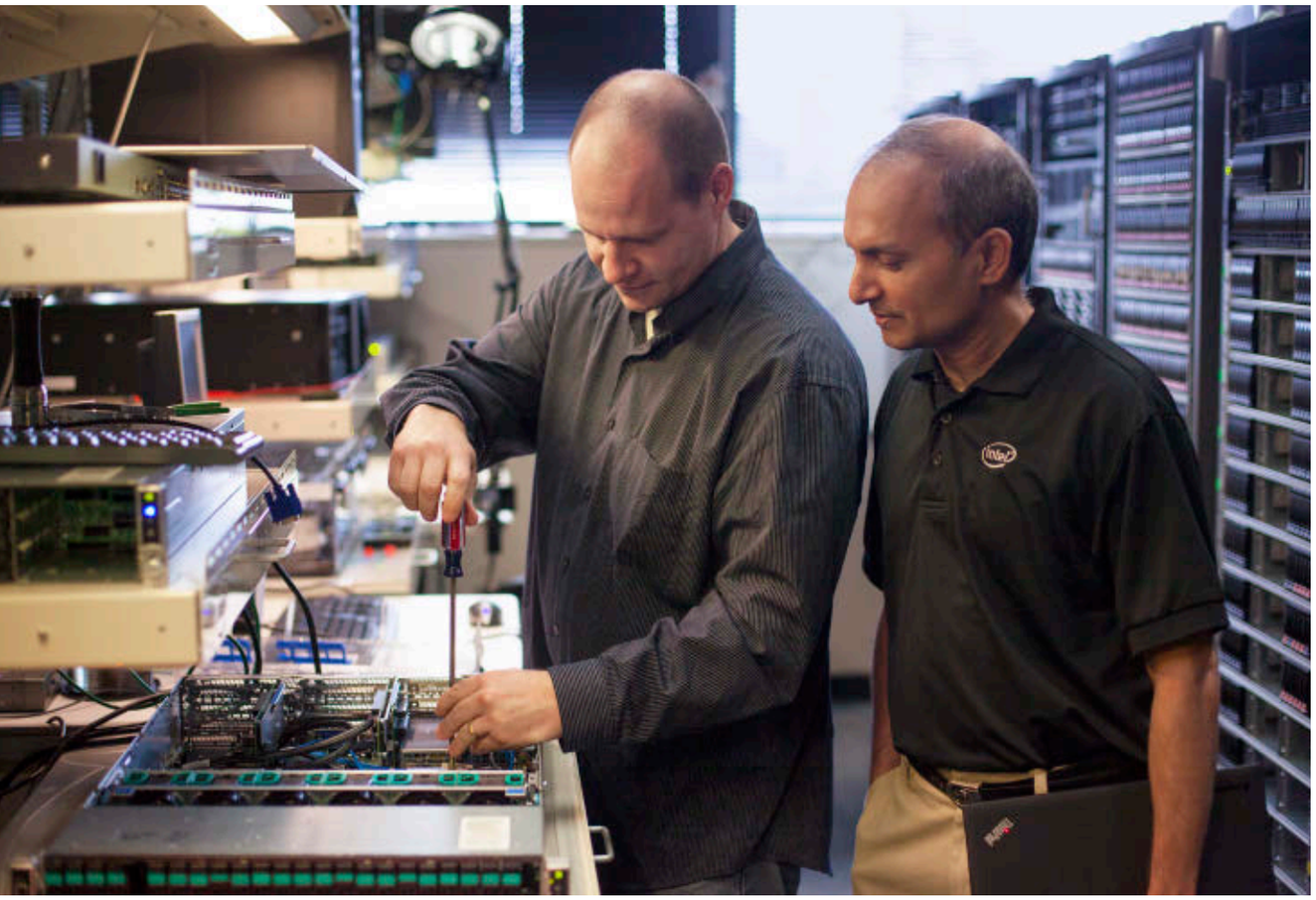
The cloud plays an important technology role—enormous numbers of mobile devices require processing and storage in the cloud. So Intel and Oracle also collaborate there to better ensure security, reliability, and performance.

And the IoT has seen explosive growth in the past few years. Many players—both hardware and software—occupy this space. But there is also a great deal of fragmentation, with different approaches, different types of software, and different hardware environments. It will be important for Intel and Oracle to collaborate around the evolution of standards in this space, and to develop effective strategies around such collaborations.

But the good news is that Intel and Oracle have a long history of fruitful collaborations. If we look at standard benchmark data—for example, specification JBB [Java Business Benchmark]—we see that hardware and software total performance increased 30 times from 2007 to 2013. And such collaborations have benefits far beyond simple performance in terms of the evolution of Java standards. When we work together around standards, it creates a healthier ecosystem—more competition, which drives innovation and which



Left to right: Eric Kaczmarek, Pranita Maldikar, Anil Kumar, and Soji Denloye collaborate informally.



Mike Jones (left) and Anil Kumar do some hands-on work for the hardware side. The group identifies optimization opportunities and default performance for hardware and software configurations.

THE BENEFIT OF STANDARDS

"When we work together around standards, it creates a healthier ecosystem—more competition, which drives innovation and which results in better performance."

results in better performance and ease of development and deployment.

Java Magazine: You've been Intel's representative on the JCP Executive Committee since 2011. Can you discuss what motivated Intel to join the JCP and the benefits you see for current and future technology initiatives?

Kumar: In 2002, when Intel first joined the JCP, a hardware platform was typically tied to just one vertical stack. But Java enabled developers to deploy the same applications on different architectural platforms. Intel joined the JCP to help contribute to the Java platform's growth, evolution, and innovation. It was an opportunity for Intel to help empower our portion of the Java ecosystem, which was undergoing major changes at the time. And that process has been fruitful. If you look at the past 10 to 15 years, Java has played a powerful and influential role within a variety of technology ecosystems. Intel has participated in 14 JSRs.

Since 2011, it has been my opportunity to participate in defining Java standards within the JCP. With the IoT and other technology evolutions on the horizon, plenty of challenges and collaborative opportunities lie ahead. So I'm enjoying the process and looking forward to the future. [More details of Intel's earlier participation on the JCP Executive Committee can be found here.]

Java Magazine: What other benefits do you see in working with the JCP?

Kumar: Intel's focus goes beyond simply increasing compute power. We have become a platform company, and that means it's no longer just about the CPU. Many other evolving features and elements benefit the end user. Being a part of the JCP provides us a path for contributing to the evolution of the Java platform and leveraging our latest hardware features most effectively. But the resulting standards also benefit developers and end users. By enabling a single powerful implementation across multiple environments, the entire ecosystem benefits.

Java Magazine: At JavaOne 2014, Intel announced it was joining the OpenJDK community as a contributing member. What are the motivations and benefits for this role within OpenJDK?

Kumar: OpenJDK provides an environment for innovation at a rapid pace. And many of the most important technology players are already part of the OpenJDK community. That has created a larger and healthier ecosystem for competition and collaboration. Intel's new features—including security features, networking, and storage—are rapidly evolving. The JCP standards process understandably

takes time, because it requires a more thorough scrutiny. But at the same time, OpenJDK provides a fast path to enable these hardware features in a robust manner.

Java Magazine: Math libraries and analytic solutions are important within the big data space. Is this also part of Intel's work within the OpenJDK community?

Kumar: From Intel's perspective, there are two basic types of performance-related innovations. At the hardware layer, we focus on accelerators in functional areas. But it's also possible to fine-tune the software layer of the stack, which can exploit either existent or new hardware features more efficiently. Because our goal is always better performance for the end user, Intel strives to contribute to both the software and hardware stack evolutions within OpenJDK.

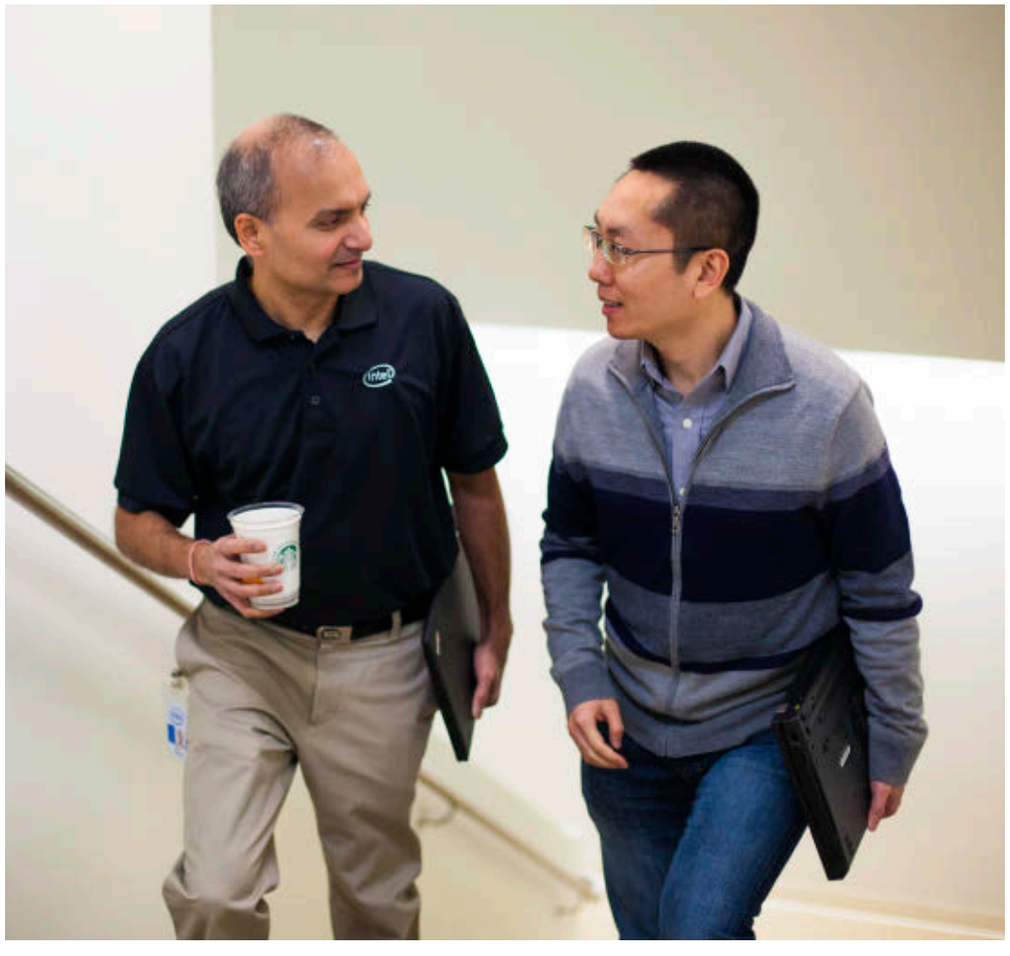
Java Magazine: At JavaOne 2014, Michael Greene [Intel vice president, Software and Services Group] discussed Intel's work with the Hadoop community to optimize hardware related to big data solutions. What is Intel's vision of empowering the flow from data-nibbling sensors, to number-crunching servers, to end-user services and insights? What are the implications for performance demands?

Kumar: As part of our focus on big data–related libraries, Intel is working with the Hadoop community and the OpenJDK community to ensure that stacks achieve maximum performance. Garbage collection is one major area of interest. And for future deployments, we are working on accelerators for encryption, compression, and computational analytics. We've recently published several white papers and videos that explore how end users can benefit on these fronts within the big data space. (See the "Working with Hadoop" sidebar.)

At the CES 2015 keynote, our CEO, Brian Krzanich, announced several breakthrough technologies for consumer electronics, including innovations within the IoT space. There are many evolutions occurring in the IoT, including our own internal performance streamlining. I am involved in this process with data processing and analytics at the back end. The pain points for end users are enormous data transfers, large memory footprints, and security issues. Therefore we are working

BUILDING OPENNESS AND COMPETITION

"Any technology environment is made more robust by greater openness and competition. Openness serves to speed evolution and innovation."



Anil Kumar (left) and Liqi Yi catch up. Intel works with the Hadoop and OpenJDK communities to ensure that stacks achieve maximum performance.

to improve our deployments on these fronts, and to maximize our solutions in future hardware designs.

Java Magazine: What are the challenges of preserving standards while also promoting future innovation? How does that affect Intel and its initiatives?

Kumar: The existent standards have worked very well up to now. But discussion and debate within the community have been increasing around IoT. This area is rapidly evolving, and now is the time for a major focus on the balance between innovation and standards. Even if we begin considering these issues now, it will be a one- or two-year time frame for the processes to play out. So we hope more companies will take a role in developing new standards around the IoT space.

Java Magazine: What recent JSRs has Intel been involved with, and what were the issues for your industry initiatives?

Kumar: We've been contributing to several JSRs. One was focused on greater openness within the JCP. From Intel's perspective, any technology environment is made more robust by greater openness and competition. Openness serves to speed evolution and innovation. And of course, performance is also a key objective. So there are relevant JSRs in the performance realm—for example, lambda expressions in Java (JSR 335), aiding computationally intensive tasks, enabling multiple cores (JSR 166 fork/join framework), and making the overall programming paradigm easier. As more data is collected through sensors, telemetry, health devices, and so on, companies need to analyze it in multiple dimensions, which requires tremendous compute power. The way that data is vectorized can also produce manyfold benefits. Intel is exploring a JSR focused on autovectorization of data—where a developer codes in the traditional fashion, but at the JIT [just-in-time] level we would autovectorize. A talk at JavaOne 2014, "Optimizing the Future of Java Through Collaboration," highlighted advanced features such as Intel AVX/AVX2 SIMD, upcoming Intel TSX, and java.util.zip CRC32 acceleration, which is another example of end users benefiting from Intel and Oracle collaboration.

Java Magazine: Has Oracle delivered on the promise of increased transparency and openness within the JCP?

Kumar: From my experience as an Executive Committee member, I would say very much so. Oracle and the program management office delivered the transparency JSR, which has had a significant impact. And I've seen the JCP program office working with the Specification Leads to ensure that JSRs adhere to the new transparency requirements. That demonstrates a concerted effort toward openness, not just in following the letter of the law.

Java Magazine: What structural/procedural changes could you imagine further improving or strengthening the JCP?

Kumar: Two JSRs (JSRs 348 and 358) are related to this question. One—to simplify the membership and contribution process—is focused on individual members. The other JSR addresses recent JSPA [Java Specification Participation Agreement] discussions focused on how to better manage IP [intellectual property] flow policy. We are seeing innovations in the IoT space, as well as in the traditional back-end data space. But the current IP flow is complex. If we can streamline that process, it will better facilitate collaboration and participation among both current and new JCP members.

Java Magazine: How can the JCP better serve the Java community?

Kumar: Several JSRs pertaining to the cloud and big data have been func-

Working with Hadoop

Intel has a dedicated team working to improve the performance and the end-user experience for big data users. Readers can find many basic to expert-level optimizations for Hadoop in these Hadoop-related white papers, talks, and videos:

"Optimizing Java and Apache Hadoop for Intel Architecture"

"Accelerate Big Data Analysis with Intel Technologies"

"Tuning Java Garbage Collection for HBase"

Video interviews with Eric Kaczmarek, big data and Java performance architect, and Soji Denloye, Java performance engineer: Part 1, Part 2, Part 3, and Part 4.

Anil Kumar (left) and Soji Denloye discuss issues on the fly. Increasingly, debate in the Java community is about the Internet of Things (IoT) and how to balance innovation and standards.

tioning well—there's one on concurrency (JSR 166), and others are related to annotation (JSRs 269 and 308) and management (JSRs 73, 77, 88, 352, and so on), which enable easier deployment. But similar approaches and JSRs in the IoT space are still somewhat lacking. The JCP serves primarily as a governance and approval body, but an active push toward facilitating new JSRs around IoT would both enhance the perception of the JCP and better serve the Java community.

Java Magazine: What would you like Java developers to understand about the processes of the JCP?

Kumar: Before I joined the Executive Committee, my impression as a Java developer was that corporate representatives played the primary roles within the JCP and other such governing bodies. But having now been a part of the JCP for the past three years, I've discovered that individual contributors and Java developers can also play important roles. There were mechanisms in place for such contributions, and the JCP has worked hard to establish even greater openness and transparency. It would be wonderful to provide examples of past and current roles that Java developers have played within the JCP—to explore the contributions that have been made, and how that benefited the entire Java community. It would serve to inspire others to become involved, and to demonstrate the real-world possibilities of contributing within the JCP.

Java Magazine: Any parting thoughts?

Kumar: When I look back over the past 10 years, I think few could have imagined the innovations in the hardware space—mobile computing, the cloud, big data, and IoT. Looking at platform-level innovations that have been delivered by Intel and Oracle, I couldn't have imagined we would each be playing such critical roles in these areas. It has been a fascinating decade to witness. When I consider the next five years, while it will be different on many levels, I am certain that Oracle and Intel will continue to play major roles. I want to see those roles continue along a collaborative path, through a standards body such as the JCP—one that is open, agile, and proactive. Such collaboration breeds competition and innovation, which ultimately benefits the entire technology ecosystem and the end user. `</article>`

MORE ON TOPIC:

Java and Performance

Steve Meloan is a former C/UNIX software developer who has covered the web and the internet for such publications as *Wired*, *Rolling Stone*, *Playboy*, *SF Weekly*, and the *San Francisco Examiner*. He recently published a science-adventure novel, *The Shroud*, and regularly contributes to *The Huffington Post*.

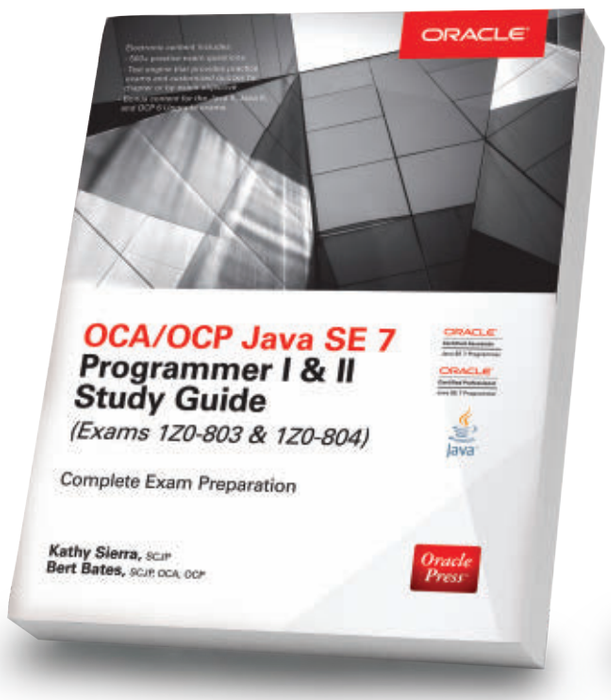
LEARN MORE

- The Java Community Process

Oracle Press™

Your Destination for Java Expertise

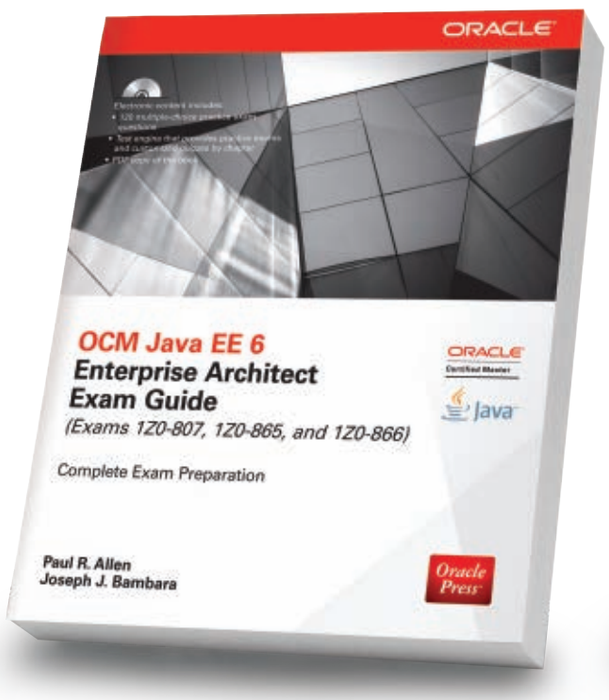
Written by leading Java experts, Oracle Press books offer the most definitive, complete, and up-to-date coverage of Java available.



OCA/OCP Java SE 7 Programmer I & II Study Guide

Kathy Sierra, Bert Bates

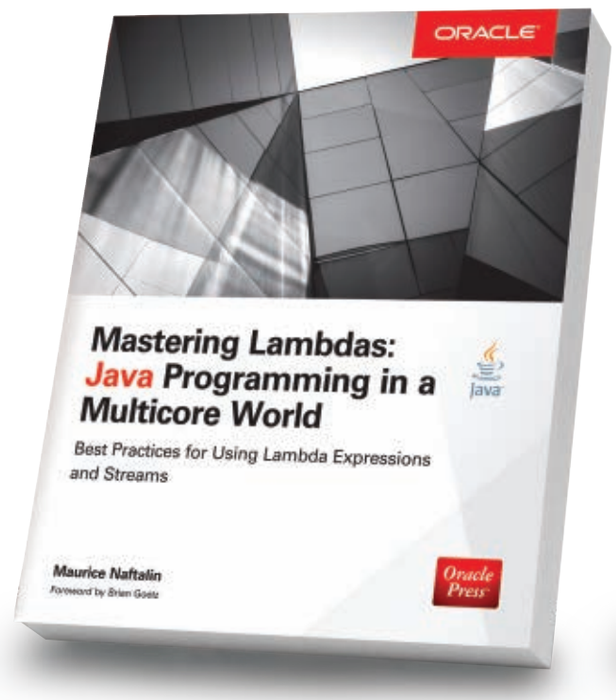
This self-study tool offers full coverage of OCA/OCP Exams 1Z0-803 and 1Z0-804. Electronic content includes 500 practice exam questions.



OCM Java EE 6 Enterprise Architect Exam Guide

Paul R. Allen, Joseph J. Bambara

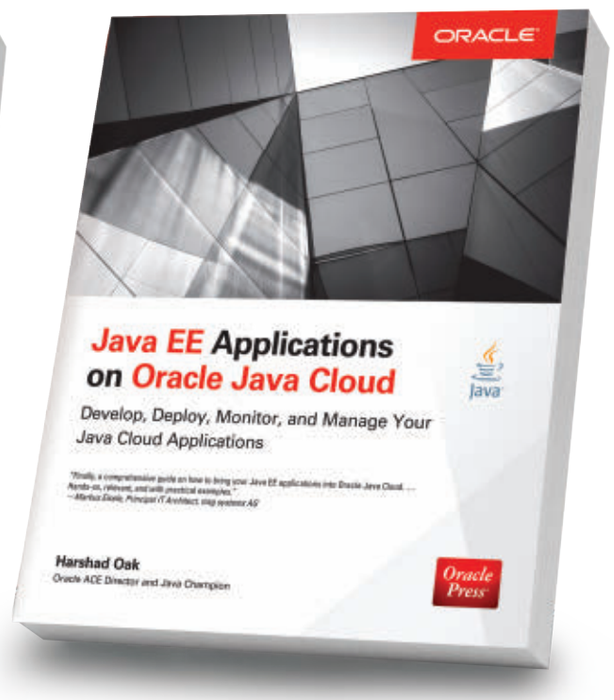
Get complete details on Oracle Certified Master Java EE 6 Enterprise Architect Exams 1Z0-807, 1Z0-865, and 1Z0-866. 120 multiple-choice practice exam questions are included.



Mastering Lambdas Java Programming in a Multicore World

Maurice Naftalin

Effectively use lambda expressions to maximize performance improvements provided by multicore hardware.



Java EE Applications on Oracle Java Cloud

Harshad Oak

Build highly available, scalable, secure, distributed applications on Oracle Java Cloud.

Available in print and eBook format

www.OraclePressBooks.com • @OraclePress OraclePress

ALWAYS ON

BY DAVID BAUM

Netflix powers through 2 billion content requests per day with Java-driven architecture.

Putting members in control and serving up targeted content based on their preferences has made Netflix the world's leading internet television network. Just about everyone is familiar with its flexible, on-demand model. Some consumers are even ditching their traditional satellite and cable services in favor of an "all-you-can-eat" TV service. For about US$10 per month, Netflix members can watch as many movies and shows as they want—anytime, anywhere, on nearly any internet-connected

Java and Performance

From left to right: Rob Fletcher, Clay McCoy, Andy Glover, Adam Jordens, Brian Johnson, and Shelley Bower are part of the always-on Netflix team.

PHOTOGRAPHY BY BOB ADLER/GETTY IMAGES

NETFLIX
netflix.com

Headquarters:
Los Gatos, California

Revenue:
US$4.4 billion

Employees:
2,300

Java technologies used:
Java EE 6, Java EE 7, and Java EE 8

Andy Glover and Jon Schneider (left) check processes, while Glover and Adam Jordens (top) confer. "The vast majority of the services running within our architecture are built on Java and the Java Virtual Machine," Glover says.

screen. They can play, pause, and resume watching, all without commercials or commitments.

Java's innate scalability has played a key role in enabling Netflix to expand its service to more than 57 million members in 50 countries. Collectively, these viewers enjoy more than 1 billion hours of shows and movies per month, including original series, documentaries, and feature films.

"The vast majority of the services running within our architecture are built on Java and the Java Virtual Machine [JVM]," says Andrew Glover, delivery engineering manager at Netflix. "Netflix uses a stateless architecture, so as we bring in more customers we are able to bring up more instances relatively easily. We have thousands of Java processes running all the time, yet as we grow we don't have huge infrastructure challenges. We also have a lot of open source tools that are Java-based, which makes it easy to monitor, upgrade, and scale our services."

Glover has a distinguished history of executive leadership, yet his passion for writing code hasn't wavered. Before joining Netflix, he held senior positions at Vanward Technologies, Stelligent, and App47. Today, he manages a team of seven engineers who are responsible

FACILITATING INNOVATION

"We need to make decisions fast, based on the data we collect."
—*Andy Glover*

"We have thousands of Java processes running all the time, yet as we grow we don't have huge infrastructure challenges," Glover says.

for the continuous delivery and deployment of new software within the vast, worldwide Netflix infrastructure. "Rapid turnaround is an important differentiator for IT, and it is certainly a competitive advantage for Netflix," he emphasizes. "Thanks to our continuous delivery platform, if a business unit wants a new feature pushed into production, we can do it within a couple hours."

Glover is echoing one of the core mantras at Netflix: move fast, make decisions fast, and do it all on a solid architecture. "Because we are a data company we need to be able to facilitate rapid innovation across the globe," he adds. "We need to make decisions fast, based on the data we collect, so we can provide the viewing experiences that our customers want—and stay ahead of competitors."

HOW NETFLIX WORKS

Netflix members love being able to browse, select, and play movies with a few clicks of a remote control or mouse, but not many of them stop to think about the sophisticated hardware and software infrastructure that delivers the content. Whenever you launch Netflix to queue up *Caddyshack*, *Some Like it Hot*, *Breaking Bad*, or any other movie or television show, Netflix first executes some core Java services to determine who you are, where you live, if you paid your bill, and what display device you are using. Then its content delivery network determines the closest cache from which to stream your content, buffered to the optimum audio and video quality for your connection at that moment.

"When a customer launches Netflix, behind the scenes the system kicks off about a dozen different processes to authorize that person, figure out what device he or she is using, make sure the account is current, and look at their recent activity," explains Glover. "That kind of orchestration is done with Eureka, an open source tool based on Java."

Netflix stores master copies of digital films in Amazon Web Services and Amazon Simple Storage Service clouds. Each film is encoded into more than 50 versions based on video resolution and audio quality—amounting to more than a petabyte of data. From there it is distributed to content delivery networks such as Akamai, Limelight, and Level 3, which in turn feed it to local internet service providers, and on to viewers.

The infrastructure is based on a service-oriented architecture (SOA) that handles approximately 2 billion content requests per day. "We have thousands of microservices running all the time," explains Glover. "This configuration offers lots of independence to our developers as they deploy new software."

Netflix Fast Facts

Year founded: 1997

Total number of subscribers: 57.4 million

Number of hours per month that people spend watching Netflix: More than 1 billion

Percentage of viewers who binge-watch shows: 61 percent

Bandwidth used: During peak times, Netflix accounts for 33 percent of North America's downstream internet traffic.

Shelley Bower, Brian Johnson, and Mike McGarr strategize. Netflix uses its continuous delivery system to schedule deployments based on up-to-the-second traffic statistics.

CONTINUOUS DELIVERY

Netflix engineers constantly deliver new features to the company's worldwide user base via continuous delivery. It's the cornerstone of all the company's business initiatives, with engineers using continuous delivery processes to rapidly push enhancements and bug fixes to customers, at low risk and with minimal overhead. According to Glover, this automated pipeline enables engineering teams to move concise feature sets from development and testing into a stable production environment. "The Netflix software environment changes a little bit every day," he adds. "The continuous delivery environment is the conduit for speed and innovation. We don't have the luxury of downtime. We can't tell customers to come back in three hours after we have completed a deployment."

Performance problems can cause buffering errors for viewers, so Netflix uses its continuous delivery system to schedule each deployment based on up-to-the-second traffic statistics. For example, if Europe is in its peak viewing time, the delivery team might elect to delay a deployment until off-peak hours. "One of the core metrics that guides the business is the number of streaming starts per second," continues Glover. "We watch this closely in every region. If there is a dip, we are alerted immediately."

Chris Berry (foreground) and Andy Glover consider options. Netflix uses its data to determine what content it should buy or produce.

1 PETABYTE DAILY

"The bread and butter of Netflix involves using data to recommend shows that customers might like."

—Andy Glover

FOLLOW THE DATA

Streaming video content over the internet isn't difficult. Technically, there is a low barrier to entry. What differentiates Netflix among video-on-demand services is all the data it collects about what customers are watching, how often they use the service, what choices they make before and after each show is completed, which genres they prefer, and many other interrelated data points. Java programs capture about a petabyte of data each day for the recommendation engine, which suggests other shows and movies that it thinks viewers will like.

Netflix also uses this data to determine what content it should buy versus produce—as well as just how to produce original content to maximize viewership. For example, when Netflix purchased the licensing rights to *House of Cards*, it was comfortable with the big investment because David Fincher, the showrunner, and Kevin Spacey, its star, were very popular on Netflix.

"The bread and butter of Netflix involves using data to recommend shows that customers might like," Glover says. "Everyone at Netflix plays a role in either capturing that data or making use of that data. Our business units can make changes quickly, partly because the entire infrastructure is built on Java."

GROUNDED IN JAVA

Netflix Open Connect includes Java processes that are installed in third-party data centers to govern how content is exchanged, cached, and streamed. According to Glover, this SOA includes thousands of instances that communicate with one another to execute business processes. "As Sun [Microsystems] used to say, 'The network is the computer.' It's all commodity hardware running Linux Ubuntu and Java EE 7. If a node fails, we can bring up a new one easily."

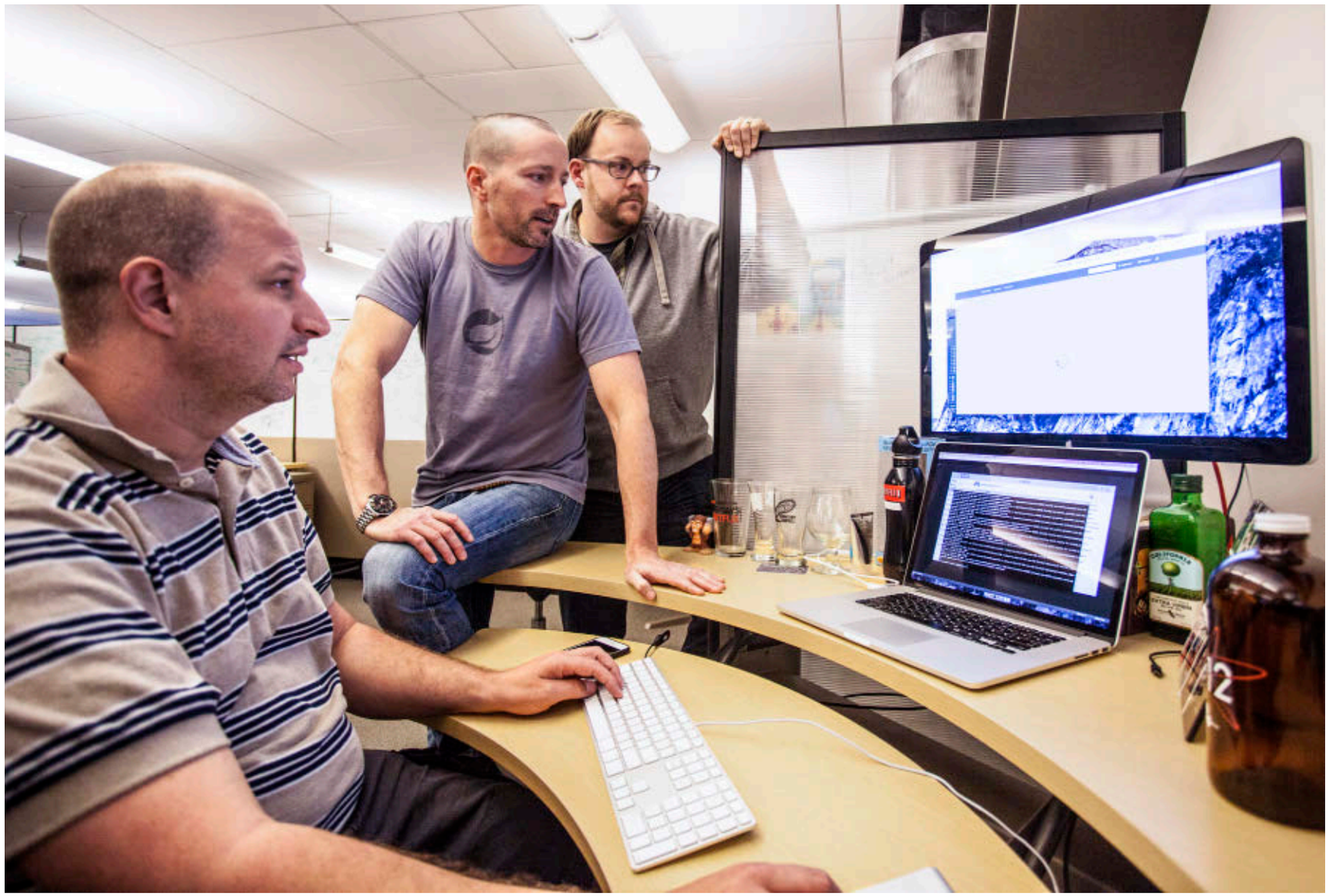
While the infrastructure runs on Linux computers, video encoding is often done on Windows machines, facilitated by a Java-based framework. Java's easy portability among platforms permits flexibility, now and in

the future. It also makes it easy to hire skilled engineers. "When it comes to recruiting, it is easy to find Java developers," Glover notes. "We can always find top talent because we are not chasing some brand-new technology or language that only hipsters know."

Glover believes Java 8 has revolutionized Java, calling it "the most significant release of Java in the last 10 years." He especially likes default methods, lambdas, and the java.util.stream API, which enables functional-style operations on streams of elements, such as Hadoop MapReduce transformations on collections. "Functional-style programming as a core language feature is a powerful construct that will have a big impact on the developmental landscape," he predicts. "Java 8 dramatically changes the Java language for the better."

Netflix uses Spring Boot as the basis of its SOA because it offers the scalability and maturity of the JVM. "Netflix is a giant SOA," Glover says. "Java platform services make it possible for a developer to quickly come up to speed and write a service that works in our architecture. Our operational footprint on the JVM is tried and true. Spring Boot is a good example of a web framework that you can get up and running in short order. There is still a tremendous amount of innovation within Java and in the Java ecosystem. It's solid. It's not going anywhere. That's why there is so much infrastructure being built on Java, such as Hadoop."

Java provides both the flexibility and scalability Netflix needs. "Our technology infrastructure is not a house of cards," he concludes. "In order to continue to be the number one streaming video content provider on the internet, we have to continually innovate. Java is the bedrock of our software architecture. The core operational infrastructure of Netflix, including the monitoring tools and platform services, has all been built in Java." `</article>`



From left to right: Cameron Fieber, Andy Glover, and Zan Thrash analyze results. Netflix uses commodity hardware running Linux Ubuntu.

MORE ON TOPIC:

Java and Performance

Based in Santa Barbara, California, **David Baum** writes about innovative businesses and emerging technologies.

KIRK PEPPERDINE

BIO

Improving the Observability of Fork/Join Operations

How to determine when adding parallelism might help performance

The introduction of lambdas and streams in Java SE 8 allows Java developers to easily add parallelism to their applications. But will making an application more parallel actually improve its performance? This is the sort of question that we at jClarity face on a regular basis.

To answer that question, let's take a look at how ForkJoinPool, the common thread pool introduced in Java SE 7, is configured and how it is being utilized. Let's also take a look at how to instrument this framework so that we can get metrics that will tell us when and how to configure it.

In the March 2005 edition of *Dr. Dobb's Journal*, Herb Sutter uttered the now infamous phrase, "the free lunch is over." That was followed by "Concurrency is the next major revolution in how we write software," which was followed by lots of talk about Amdahl, Moore's law, heat dissipation, how programmers and programming languages needed to change, and so on. There was also a lot of talk about read-write barriers and lock-free, wait-free algorithms as well as the use of techniques that rely on "divide and conquer" techniques. One of the divide and conquer techniques that has gained a lot of favor is the fork/join framework.

The Fork/Join Framework

The fork/join framework allows you to divide a large task into a number of smaller subtasks. Each of these smaller tasks is broken down even further until, finally, you have atomic units of work that will all be executed. The results of processing the atomic units are then joined together to produce a final result.

Java SE 7 introduced top-level support for the fork/join framework with ForkJoinPool, ForkJoinTask, and ForkJoinWorkerThread. Unfortunately, the programming idiom for using the fork/join framework is fairly complex and, consequently, this technique has been mostly ignored by everyday Java developers banging out business applications.

That all changed with the release of Java SE 8 and the introduction of streams. Under the hood, parallelized streams are converted into fork/join tasks, and that makes it easy for developers to add parallelism to their applications without under-

Click here to listen to Stephen Chin (right) interview Kirk Pepperdine at Jfokus.

Java and Performance

PHOTOGRAPH BY ARPAD KURUCZ/GETTY IMAGES

standing the underlying mechanism. As much as adding parallelism might help performance, indiscriminate use of the fork/join framework can certainly harm performance. Let's explore this further by looking at how streams work.

Streams

A stream takes a collection as its source and (similarly to an iterator) makes each element available to some process. Streams have no storage and might or might not be bounded. In addition, streams support intermediate (aka lazy) and terminal (aka eager) operations. Intermediate operations always return a new stream, whereas terminal operations produce a result. All of this happens serially unless a parallel stream is asked for, and that option is supported by the underlying datasource. **Listing 1** shows examples of sequential and parallel streams.

Note: For hyper-threaded CPUs, there are two logical cores for every physical core.

In **Listing 1**, stream() executes sequentially over the datasource, which is an array of garbage collection (GC) log records called gcLogRecords. Calling parallelStream() on the datasource results in the work being executed as a fork/join operation. By default, the fork/join framework uses a common thread pool, and the default size of that thread pool is equal to the number of logical cores.

Response Time, Threads, and Queues

The response time (or latency) of a request is the composite of the time when the thread is making forward progress and the time when processing is stalled (known as dead time). Typically, dead time accounts for a significant portion of the response time and, typically, most of that time is spent in a queue. Thus, if we can reduce the time spent in queues, we should be able to see significant reduction in latencies.

In the case of a fork/join operation, what we'd like to know is how many threads are in the common thread pool and how many outstanding requests are in the work queue. It is this information—combined with other resource utilization metrics—that will help us determine whether our system could benefit from a larger or smaller common thread pool.

It's easier to talk about queues if we first talk a bit about queueing theory and, in particular, Little's law. Little's law states that the number of tasks in a system (L) is equal to the average amount of time a task spends in the system (lambda) multiplied by the average rate of entry into the system (N). That is expressed as *L = lambda * N*.

The powerful thing about Little's law is that it allows you to treat the entire system as a black box. For example, I worked on a system that had a requirement for 1,500,000 transactions per hour. The average latency was 300 milliseconds.

1,500,000 transactions per hour equals 417 transactions per second, so if we peek into the system, the expected number of active requests at any point in time would be 0.300 * 417 = 125. If our thread pool size was limited to 8, that would result in 8 requests making forward progress and 117 requests sitting in a work queue. Assuming there is enough hardware to support more active requests, it's very likely that this system would benefit from a larger thread pool.

Instrumenting Fork/Join Operations

To get the statistics needed for the type of analysis performed above, we need to add some instrumenta-

LISTING 1

```
public DoubleSummaryStatistics calculateSequentially(
    List<String> gcLogRecords) throws IOException {
  return gcLogRecords.stream().
        map(applicationStoppedTimePattern::matcher).
        filter(Matcher::find).
        mapToDouble(matcher ->
           Double.parseDouble(matcher.group(2))).
        summaryStatistics();
}

public DoubleSummaryStatistics calculateParallel(
    List<String> gcLogRecords) throws IOException {
  return gcLogRecords.parallelStream().
        map(applicationStoppedTimePattern::matcher).
        filter(Matcher::find).
        mapToDouble(matcher ->
           Double.parseDouble(matcher.group(2))).
        summaryStatistics();
}
```

Download all listings in this issue as text

tion. After careful examination, I determined that I needed to instrument only ForkJoinTask. However, the monitoring would be attached to ForkJoinPool.

Let's define some test cases that will provide a context for the rest of this discussion. At jClarity, we've spent a lot of time looking at GC logs with Censum. I took this as an opportunity to look at how lambdas might be used to parse a GC log. In the following example, we'll focus on records for "application stopped time" and "application concurrent." **Listing 2** shows a rule for parsing application stopped times from a GC log.

The code in **Listing 2** takes a list of GC log entries, filters them by those that match the pattern, and then calculates a number of statistics that describe the stopped-time behavior. We can execute this code by submitting it directly to the ForkJoinPool, as shown in **Listing 3**, which shows the explicit execution of a fork/join operation.

The code in **Listing 3** creates a new ForkJoinTask and submits it to the common ForkJoinPool. When the task has been completed, a call to ForkJoinTask::setCompletion() is made, after which the results can be retrieved. The parameter to that call is a status word that lets the caller know whether the task completed normally or whether an error was trapped. Any call to get() prior to setCompletion() being called will result in the calling thread being blocked.

Ideally we would like to measure the time from the first call to submit() to the time when the job is complete. This is easily done when we explicitly call the fork/join operation. However, this technique fails with the code in **Listing 4** even though the expression in that listing is transformed into something similar to the code shown in **Listing 3**.

To work around this, the monitoring can be placed deeper into the call stack. Doing this introduces a small amount of error in the timing, but it is well below the level of noise that you'll find in any application. The code in **Listing 5** was lifted from OpenJDK and instrumented with a call to emit a signal when the task starts and when the task is complete.

Defining the ForkJoinPoolMonitorMXBean

Now that we have a signal to work with, we need something to analyze it. My favored technique in Java is to introduce an MXBean. A complete discussion about MXBeans is beyond the scope of this article. However, it's a technique that I teach in my performance tuning workshop, and it is covered in an online tutorial on Oracle Technology Network. Let's define some requirements and then introduce an interface for the MXBean.

The ForkJoinPoolMonitorMXBean should expose the following: arrival and retirement rates, the total number of tasks submitted, and the average time a task spends in the system. If you recall, Little's law tells us that the average time of a task in the system multiplied by the arrival rate equals the average number of tasks in the system. The average queue length combined with the number of worker threads and a set of timers can be used to calculate the average amount of dead time accumulated by tasks waiting in the queue. This could lead us to adjust the size of the thread pool, assuming there is enough hardware to support the reconfiguration. The definition of the ForkJoinPoolMonitorMXBean is shown in **Listing 6**.

In addition to these methods, ForkJoinPoolMonitor implements submitTask(ForkJoinTask) and retireTask(ForkJoinTask). The implementation (shown in **Listing 7**) makes use of a ConcurrentHashMap to track all tasks currently in the system.

LISTING 2 / LISTING 3 / LISTING 4 / LISTING 5 / LISTING 6 / LISTING 7

```
public class ApplicationStoppedTimeStatistics {

  public DoubleSummaryStatistics calculateParallelStream(
      List<String> logEntries) throws IOException {
    return  logEntries.parallelStream().
        map(applicationStoppedTimePattern::matcher).
        filter(Matcher::find).
        mapToDouble(matcher ->
          Double.parseDouble(matcher.group(2))).
        summaryStatistics();
  }
}
```

Download all listings in this issue as text

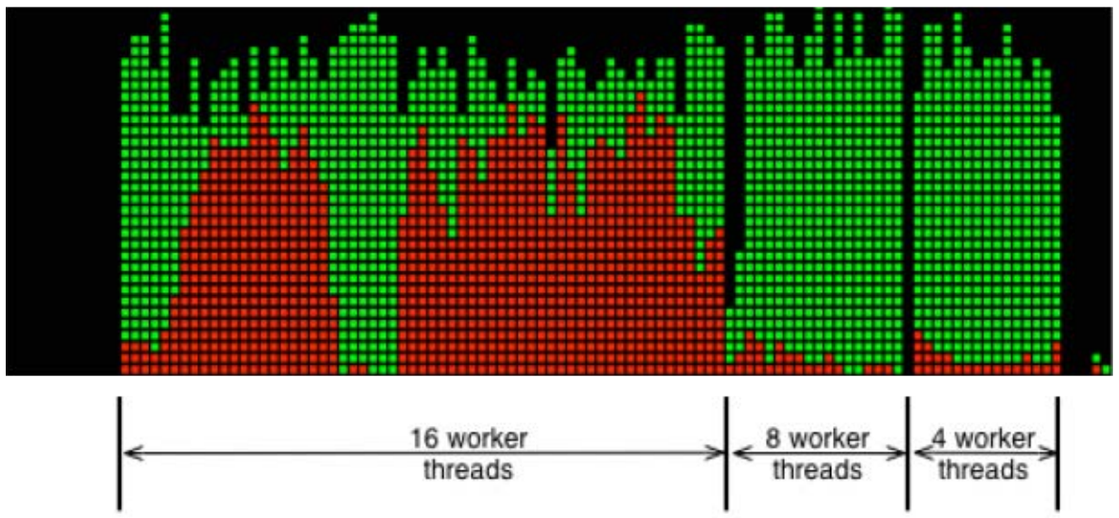


Figure 1

There is also code to wire the MXBean into the PlatformMBean Server. This exposes the MXBean to Java Management Extensions (JMX) clients such as jConsole or VisualVM's JMX plugin.

The Unit of Work

The call to concurrentParallelStream() in **Listing 8** results in the repeated execution of two Runnable lambdas. The submit() method takes the Runnable and wraps it in a ForkJoinTask. Inside the calculateParallel() is the call sequence Files.lines(path) .parallel(). This will result in a new ForkJoinTask that will, in parallel, process all the lines in the GC log file. The two tasks are submitted one after another and before the call to get(). That means they should run in parallel. However, the underlying lambda expressions in the calls to calculateParallel() will disrupt this because they intrinsically make a call to get(). So while there will be some overlap in the execution of the calls to the fork/join operation, as the numbers will show, the overlap is limited.

Getting It to Run

To get all this to work, we need to execute our modified versions of ForkJoinPool and ForkJoinTask that we borrowed from OpenJDK. In OpenJDK, the original implementations are found in rt.jar and that file is loaded by the bootstrap class loader. Fortunately we don't have to tear rt.jar apart and rebuild it with our versions of the classes. The Java Virtual Machine (JVM) gives us the option to prepend classes to the boot classpath using the parameter -Xbootclasspath/p:<path>, as shown in **Listing 9**.

LISTING 8 LISTING 9

```
public long concurrentParallelStream(int repeat,
                                     String fileName) {
 ForkJoinTask<DoubleSummaryStatistics> applicationTime;
 ForkJoinTask<DoubleSummaryStatistics>
  applicationStoppedTime;
 DoubleSummaryStatistics
  applicationStoppedTimeStatistics = null;
 DoubleSummaryStatistics applicationTimeStatistics = null;
 long timer = System.nanoTime();
 try {
  Path path = new File(fileName).toPath();
   for ( int i = 0; i < repeat; i++) {
    applicationTime =
     ForkJoinPool.commonPool().submit(()
     -> new ApplicationTimeStatistics()
       .calculateParallel(path));
    applicationStoppedTime =
     ForkJoinPool.commonPool().submit(()
     -> new ApplicationStoppedTimeStatistics()
       .calculateParallel(path));
    applicationTimeStatistics = applicationTime.get();
    applicationStoppedTimeStatistics =
     applicationStoppedTime.get();
   }
   timer = System.nanoTime() - timer;
   System.out.println("\n-Concurrent Parallel---");
   System.out.println("Concurrent Stats             : "
    + applicationTimeStatistics);
   System.out.println("Stopped Stats                : "
    + applicationStoppedTimeStatistics);
   System.out.println("Combined Time (client)    : "
    + ((double) timer) / 1000000.0d + " ms");
  } catch (Throwable t) {
   t.printStackTrace();
 }
 return timer;
}
```

Download all listings in this issue as text

Note: Class loading in Java SE 9—and, thus, these details—will most likely change.

Discussion of Results

The two units of work used in this example are both big and complex enough that they should benefit from parallelism. The first unit of work reads all of its data from memory. The second reads its data from disk. Each of the units of work is run serially using parallel streams, and then each is run again using fork/join operations. The results of running the two different workloads under these different conditions can be found in **Table 1**, which shows GC log parsing excluding I/O operations, and **Table 2**, which shows GC log parsing including I/O operations.

Before we look at the results further, I should give a word of caution. There is enough noise in these experiments that I wouldn't put any stock in the fine differences. The variation between runs was on the order of several seconds, which is why the results are coarsened to two significant digits. That said, the benchmark appears to be good enough to answer the following question: Can we do better by adapting the size of the ForkJoinPool? The answer in this case is mixed.

- Parallelizing brings benefits most of the time. The only (expected) loss is when parallelization puts extra pressure on the I/O subsystem.
- CPUs are about 20 percent idle when the lambda expression implicitly uses the fork/join framework. Explicitly using the fork/join framework soaked up that 20 percent, which resulted in a corresponding reduction in response time.
- The reductions in response time don't hold when I/O operations are part of the workload. This is a reasonable result, because I/O operations are many orders of magnitude slower than memory operations.

These observations call for more experiments. Let's mix the CPU- and I/O-bound units of work and increase the level of concurrency. We can control the number

	TASKS SUBMITTED	TIME IN FORKJOINPOOL (SECONDS)	INTER-REQUEST INTERVAL (SECONDS)	EXPECTED NUMBER OF TASKS IN FORKJOINPOOL	TOTAL RUNTIME (SECONDS)
LAMBDA PARALLEL	20	2.5	2.5	1	50
LAMBDA SERIAL	0	6.1	0	0	123
SEQUENTIAL PARALLEL	20	1.9	1.9	1	38
CONCURRENT PARALLEL	20	3.2	1.9	1.7	39
CONCURRENT FLOOD (FORK/JOIN)	20	6.0	1.9	3.2	38
CONCURRENT FLOOD (STREAM)	0	2.1	0	0	41

Table 1

	TASKS SUBMITTED	TIME IN FORKJOINPOOL (SECONDS)	INTER-REQUEST INTERVAL (SECONDS)	EXPECTED NUMBER OF TASKS IN FORKJOINPOOL	TOTAL RUNTIME (SECONDS)
LAMBDA PARALLEL	20	2.8	2.8	1	56
LAMBDA SERIAL	0	7.5	0	0	150
SEQUENTIAL PARALLEL	20	2.6	2.6	1	52
CONCURRENT PARALLEL	20	5.8	3.0	1.9	60
CONCURRENT FLOOD (FORK/JOIN)	20	43	6.5	6.6	130
CONCURRENT FLOOD (STREAM)	0	3.0	0	0	61

Table 2

	TASKS SUBMITTED	TIME IN FORKJOINPOOL (SECONDS)	INTER-REQUEST INTERVAL (SECONDS)	EXPECTED NUMBER OF TASKS IN FORKJOINPOOL	TOTAL RUNTIME (SECONDS)
16 FORK/JOIN THREADS	20	210.6	15.2	13.83	304.5
8 FORK/JOIN THREADS	20	27.2	4.1	6.65	81.9
4 FORK/JOIN THREADS	20	14.0	3.60	3.88	72.0

Table 3

of threads by setting the property -Djava.util.concurrentForkJoinPool.common.parallelism=N. **Table 3** shows the results of the mixed workload with 4, 8, and 16 instances of ForkJoinWorkerThreads.

The results in **Table 3** confirmed my suspicion that adding more threads would not have any significant benefit. The CPU utilization chart in **Figure 1** offers some explanation of why this might be the case.

The two tasks were submitted such that subtasks from both workloads should have intermixed. However, there is a telling sign in **Figure 1**: the big red blobs due to overloading the kernel with disk I/O operations. The first run with 16 threads is dominated by kernel activity. In contrast, the subsequent runs with 8 and 4 worker threads show much less kernel activity. I should remind you that the exact same workload was offered up in all three cases. The only difference is the number of threads in the common thread pool.

ADDING THREADS
Adding threads on **an already stressed I/O system** can be detrimental to performance.

Conclusion

What these experiments indicate is that there doesn't seem to be much benefit in adding more threads. In fact, these results show that adding more threads and, hence, adding more pressure on an already stressed I/O system, can be exceptionally detrimental to performance.

"Measure, don't guess" tells us that when we tune, we always want to make informed decisions driven by solid measurements. To get the needed measurements, we often need to improve the observability of the application being investigated. In this case, we were better able to understand how the fork/join framework performed by adding instrumentation. To simplify the task of adding instrumentation, the source code for ForkJoinPool was directly modified. A better way to instrument would be to use bytecode manipulation (through the agent framework).

A project is being incubated at Adopt OpenJDK on GitHub and it is hoped that it will be included in Java as a formal patch. I invite you to add other measures—such as steal counts (already estimated by the fork/join framework) or the number of active threads—or any other measure you think would help improve the visibility of the fork/join framework. `</article>`

MORE ON TOPIC:

Java and Performance

LEARN MORE

- Censum
- ForkJoinPoolMonitorMXBean project on GitHub

CHRIS NEWLAND AND BEN EVANS

BIO

Part 3

Understanding Java JIT Compilation with JITWatch

See the effects of small source code changes and Java HotSpot VM switches.

Oracle's Java HotSpot VM contains dynamic compilers. These compilers gather statistics from your running program that are used to optimize performance by selectively replacing interpreted bytecode with faster native code using just-in-time (JIT) compilation techniques.

JITWatch is a free, open source tool that analyzes the complex compilation log file output generated by Java HotSpot VM and helps you visualize and understand the optimization decisions it made. Chris Newland developed JITWatch as part of the Adopt OpenJDK project, and it is available for download from GitHub. Follow Chris on Twitter @chriswhocodes for updates.

In Part 2 of this three-part series, we explored how to make Java HotSpot VM produce the information JITWatch needs. In this article, we walk through some of the more advanced Java HotSpot VM features and explore how you can use the JITWatch Sandbox to test the effects of code changes and configuration settings on Java HotSpot VM behavior.

JITWatch Refresher

You can download the JITWatch binary from GitHub, or you can build it from source code using the following command:

```
mvn clean install
```

Then, you can launch JITWatch using ./launchUI.sh (for Linux or Mac OS) or ./launchUI.bat (for Microsoft Windows). Full instructions for getting started are detailed in the wiki.

As we look at more-advanced Java HotSpot VM features, you will find it useful to have the disassembly binary (called hsdis) installed in your Java runtime environment (JRE) so that you can view the disassembled native code the JIT compilers produce. Instructions for building hsdis can be found here.

Enter the Sandbox

The Sandbox is a JITWatch feature that lets you edit code and then compile, execute, and analyze the Java HotSpot VM JIT logs, all from within the JITWatch application. It's intended to help developers understand what goes on "under the hood" of Java HotSpot VM when a program is run and to see the effects of small source code changes and Java HotSpot VM switches.

Note that isolated testing of algorithms inside the Sandbox might not result in the same Java HotSpot VM compiler decisions that would be made when running the full application. That's because Java HotSpot VM will have less profiling information on which to base its optimization decisions.

ANALYZE THIS

JITWatch is **a free, open source tool** that analyzes the complex compilation log file output generated by Java HotSpot VM and helps you visualize and understand the optimization decisions it made.

Java and Performance

CHRIS NEWLAND PHOTOGRAPH BY DAVID NEWLAND; BEN EVANS PHOTOGRAPH BY JOHN BLYTHE

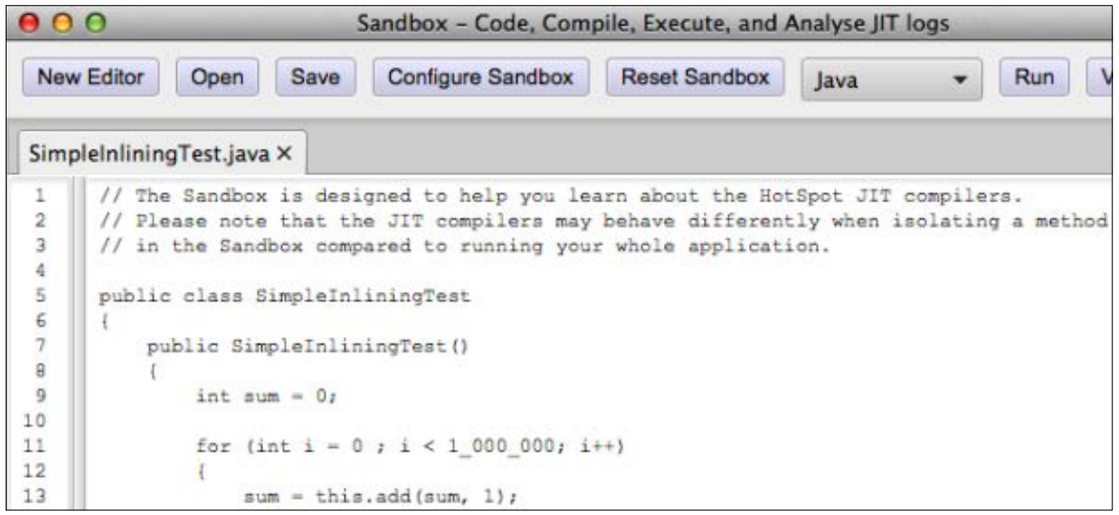


Figure 1

Once you've launched JITWatch, click the **Sandbox** button in the top left of the main window to open a Sandbox window like the one shown in **Figure 1**.

As well as supporting Java source code, the Sandbox has support for Scala and Groovy, because these languages compile to bytecode that executes on the Java Virtual Machine (JVM). Other JVM languages—such as Kotlin, Clojure, JRuby, and JavaScript (using Nashorn)—will be supported in the future.

Behind the scenes, JITWatch uses a java.lang.ProcessBuilder to compile and execute your code.

Experimenting with Java HotSpot VM Switches

Let's begin by looking at which Java HotSpot VM switches can be used to control JIT compilation.

Click the **Configure Sandbox** button to open the Sandbox Configuration window shown in **Figure 2**. You can set up each VM language you wish to use in the Sandbox by specifying its home directory.

Next, select **Show Disassembly** to instruct the Java HotSpot VM to produce human-readable assembly language from the JIT-compiled native code. This capability requires the hsdis binary.

You can override the default setting for the **Tiered Compilation** option, which controls when code is optimized. Optimization is done first quickly with the C1 client compiler and then sometimes again with the more advanced C2 server compiler after more runtime statistics have been gathered. The **Tiered Compilation** option is disabled by default in Java SE 7 and

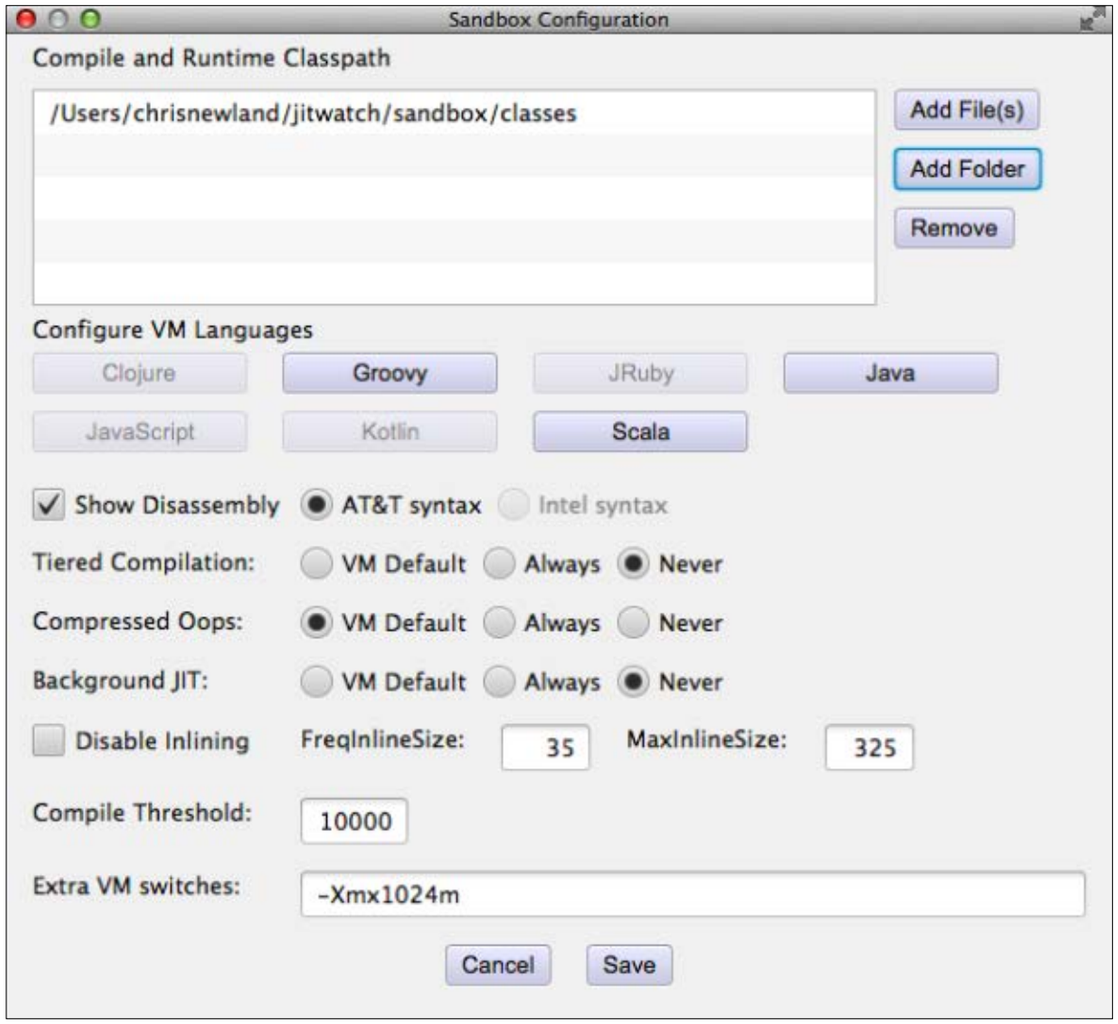


Figure 2

enabled in Java SE 8.

Java HotSpot VM manages a pointer to an object, called an *oop* (ordinary object pointer), that is usually the same size as the native machine pointer. So on a 64-bit system, the required heap space will be larger than on a 32-bit system. Java HotSpot VM is able to save heap space by representing 64-bit pointers as 32-bit offsets from a 64-bit base; this is known as *compressed oops*. Disabling the **Compressed Oops** option simplifies the inspection of the disassembled native code, because it eliminates the pointer arithmetic code needed to support compressed oops. Make sure you understand the implications before altering this setting in a production environment. For more information, see "Compressed Oops".

The **FreqInlineSize** and **MaxInlineSize** fields control the bytecode size thresholds up to which Java HotSpot VM will inline small and hot methods, respectively. You can also choose to disable inlining completely. (You might find it interesting to measure how much performance is lost when you do this.)

Finally, the **Compile Threshold** field sets the number of method invocations required before the C2 server compiler is invoked, and the **Extra VM switches** field allows you to pass any additional tuning parameters.

Hovering over an option will display a tool tip that shows the exact switch passed to Java HotSpot VM.

JIT Method Inlining

We'll now use the Sandbox to learn about how Java HotSpot VM employs a JIT technique called *method inlining*, which is when the call to a target method is replaced with the method body itself to eliminate the invocation overhead. This early step is important for the JIT compiler, because it brings related code closer together before further optimizations (such as loop unrolling) are performed.

Java HotSpot VM can inline code when the method is called on an interface type and the concrete types are not known until runtime. It does this by counting the number of implementations that are used during program execution:

- When the call is observed as being *monomorphic* (a single implementation is seen), the code can be simply inlined.
- When the call is *bimorphic* (two implementations are observed), Java HotSpot VM can inline both implementations and use a simple condition to choose between them.
- When the observed behavior is *megamorphic* (more than two implementations), lookup tables are required and Java HotSpot VM cannot inline the method implementations.

Java HotSpot VM adds an *uncommon trap* to the optimized code in case the number of observed implementations increases and it has to reconsider the decisions it made.

Let's use the example of an interface called Coin that declares a single method deposit() and three implementations of the interface named Nickel, Dime, and Quarter. You can observe the different behaviors when two implementations (bimorphic dispatch) and three implementations (megamorphic dispatch) are seen at runtime.

In the Sandbox, open the source file PolymorphismTest.java, which is included in the JITWatch examples.

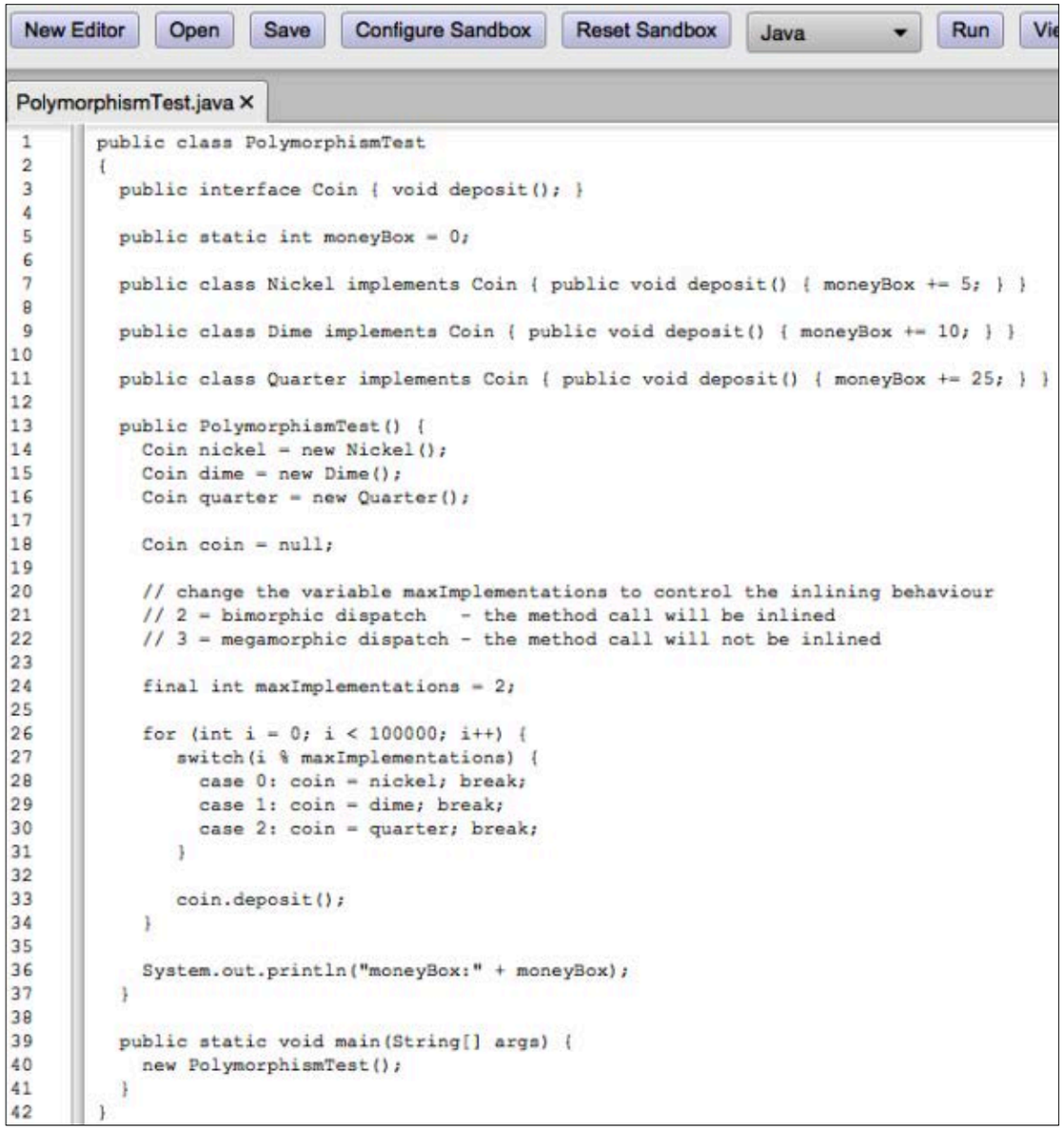


Figure 3

See the listing shown in **Figure 3**. On line 24, you can control the number of implementations that are used by setting the variable maxImplementations.

Click the **Run** button in the Sandbox window to execute the program. The JITWatch TriView window, shown in **Figure 4**, opens as soon as the Java HotSpot VM log file has been analyzed.

When two of the Coin implementations are used, hovering over the invokeinterface instruction at offset 93 in the Bytecode pane shows a tool tip confirming that the method

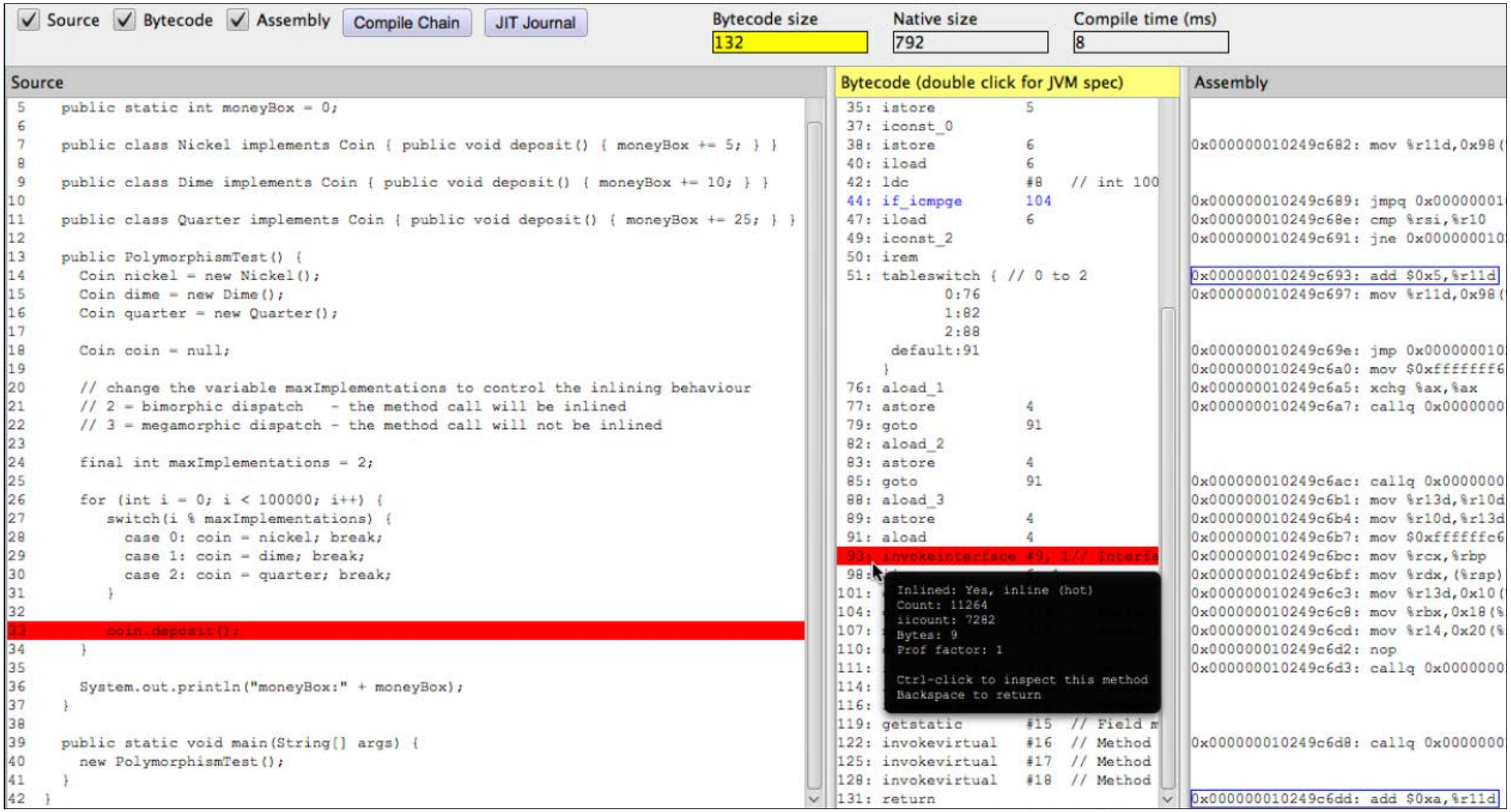


Figure 4

was successfully inlined, as shown in **Figure 4**.

In the Assembly pane, you can see the native code inlined from the deposit() method on inner class Polymorphism$Nickel. This adds 5 to the static variable moneyBox on source line 7 using the assembly instruction at 0x000000010249c693: add $0x5,%r11d.

Lower down in the Assembly pane, you can see the corresponding inlined code for adding 10 to the static variable moneyBox from Polymorphism$Dime on source line 9 using the assembly instruction at 0x000000010249c6dd: add $0xa,%r11d.

To see the exact set of optimizations performed, click the **JIT Journal** button in the top left of the TriView screen. The uncommon trap for the bimorphic inlining is indicated in **Figure 5** with a red dot.

When all three Coin implementations are involved, the method call to deposit() can't be inlined

```
<method bytes="9" name="deposit" flags="1" holder="782" id="793" iicount="7282
<call method="793" inline="1" count="11264" prof_factor="1"/>
<inline_success reason="inline (hot)"/>
<predicted_call klass="781" bci="93"/>
<uncommon_trap reason="null_check" bci="93" action="maybe_recompile"/>
<predicted_call klass="782" bci="93"/>
<uncommon_trap reason="bimorphic" bci="93" action="maybe_recompile"/>
<parse method="793" stamp="0.704" uses="11264"> <!-- void PolymorphismTest$Nic
  <parse_done nodes="264" memory="66184" stamp="0.704" live="258"/>
</parse>
<parse method="792" stamp="0.704" uses="11264"> <!-- void PolymorphismTest$Dim
  <parse_done nodes="281" memory="69680" stamp="0.704" live="274"/>
</parse>
<bc code="178" bci="104"/>
```

Figure 5

and there is no inlining tool tip at bytecode offset 93, as shown in **Figure 6**.

In addition, you won't find the inlined assembly code for deposit(), as was seen in **Figure 4**, and the virtual_call for the invokeinterface bytecode hasn't been eliminated. Also note that the native code size of 504 bytes is smaller than with bimorphic dispatch, because no inlining has occurred to increase the size of the native code.

That's as deep as we want to dive this time, but if you'd like to know more, I recommend running the examples that come with the Sandbox and examining the optimizations that Java HotSpot VM can make.

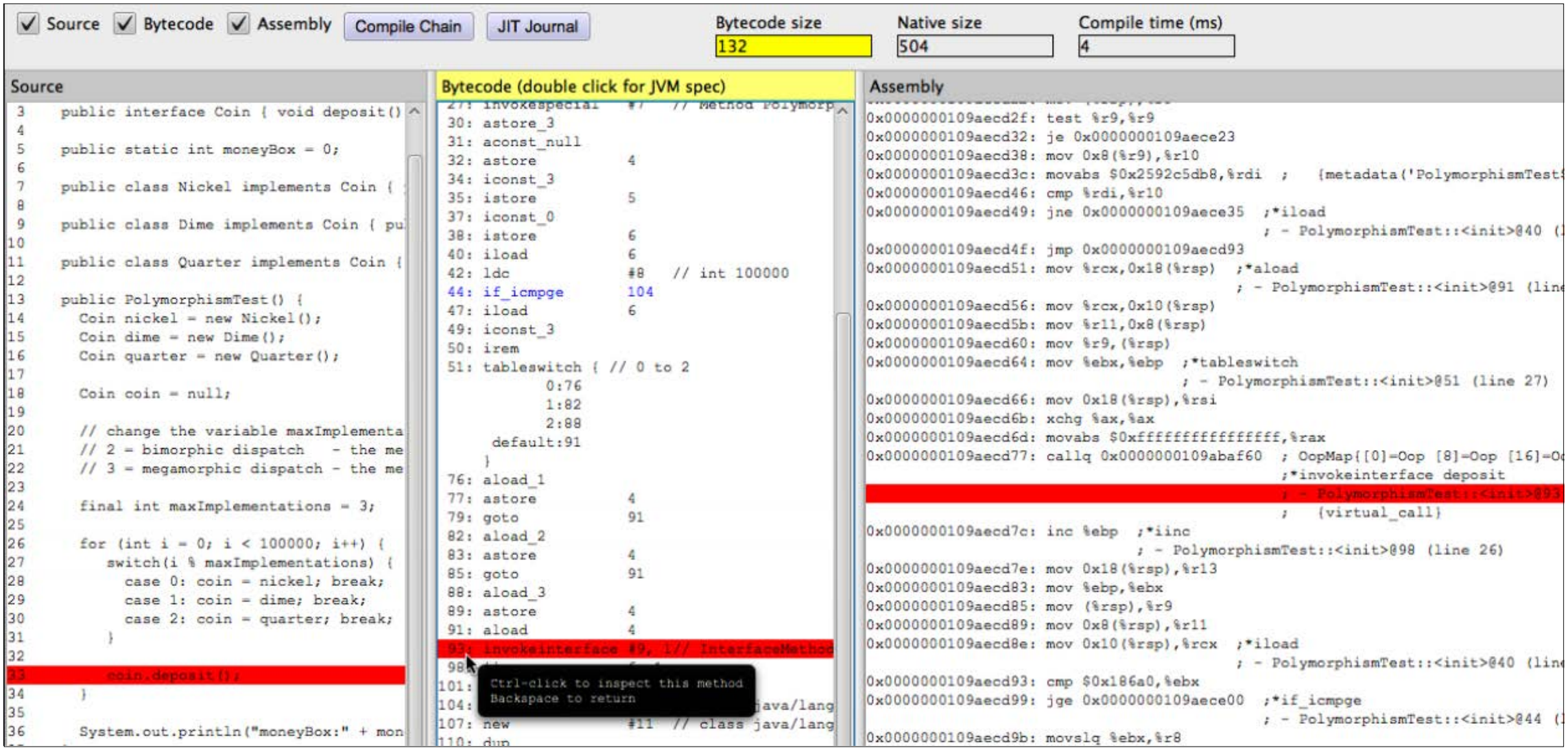


Figure 6

JarScan Tool

The JITWatch download includes a tool called JarScan that will scan a list of JAR files and count the bytecode size of every method and constructor it finds. The purpose of this tool is to highlight methods that have more bytes of bytecode than the Java HotSpot VM threshold for inlining "hot" methods, so you can get hints on where to benchmark to see whether decomposing code into smaller units can increase the program's performance.

The JIT compiler determines the "hotness" of a method using a set of heuristics that include the method's call frequency and the number of times back branches (such as on a loop) are taken.

There are limits on what can be inlined, including a limit on the bytecode size of the called method. This behavior can be tuned using the Java HotSpot VM switch -XX:FreqInlineSize=n (the default "hot" bytecode size threshold is 325 bytes on 64-bit Linux).

More details on the JarScan tool can be found in the wiki.

Just because a method's bytecode size exceeds FreqInlineSize doesn't guarantee that it's a performance bottleneck. The code might never be invoked enough times to reach the hot code limit. (Remember that JarScan is a static analysis tool and has no knowledge of how methods are used.)

We will now explore using the JarScan tool to examine the rt.jar file, which contains the core Java libraries (such as java.lang.* and java.util.*) from the latest version of Java.

Run the command shown in **Listing 1**. This command might take a couple of minutes to complete, because the tool disassembles each method in the JAR file's class files using the API form of the javap program called com.sun.tools.javap.JavapTask.

In Java SE Development Kit 8, Update 31 (JDK 8u31) on 64-bit Linux, 3,605 methods exceed the FreqInlineSize threshold, and many of these aren't a surprise. For example, java.util.GregorianCalendar.computeFields() has a lot of work to do that justifies its 1,571 bytes of bytecode. Some of the results, however, might raise an eyebrow because they include core library methods—such as java.lang.String.toUpperCase() and java.lang.String.toLowerCase()—that you might expect to find inside tight loops. These both weigh in at 439 bytes of bytecode.

The Javadoc explains that this is due to the requirement to support character sets in all languages: "Since case mappings are not always 1:1 char mappings, the resulting String may be a different

LISTING 1 LISTING 2a LISTING 2b LISTING 3

```
./jarScan.sh /home/chris/jdk1.8.0_31/jre/lib/rt.jar > methods.csv
```

Download all listings in this issue as text

length than the original." So they contain code to check and resize the underlying character arrays, if needed.

This spawned the idea that if your application domain doesn't require the flexibility the core libraries offer and performance is critical, creating your own less-flexible version of a core library method can result in significant performance improvements.

As a test, a method was written that converts String objects to uppercase. It works only on the ASCII characters *a* through *z*, and its performance is measured against java.lang.String.toUpperCase using the OpenJDK JMH benchmarking framework. Run this benchmark on the command line, rather than from the Sandbox, and load the resulting Java HotSpot VM log file into JITWatch for analysis.

The code for this benchmark is shown in **Listings 2a** and **2b**.

It turns out there is a heavy price to pay for supporting all character sets. The custom version achieved more than 4.5 times as many operations per second as the core library version, as shown in **Listing 3**.

Analyzing the log file from JMH shows that the method toUpperCaseASCII generates 69 bytes of bytecode and every method call it makes was successfully inlined (highlighted in green, as shown in **Figure 7**).

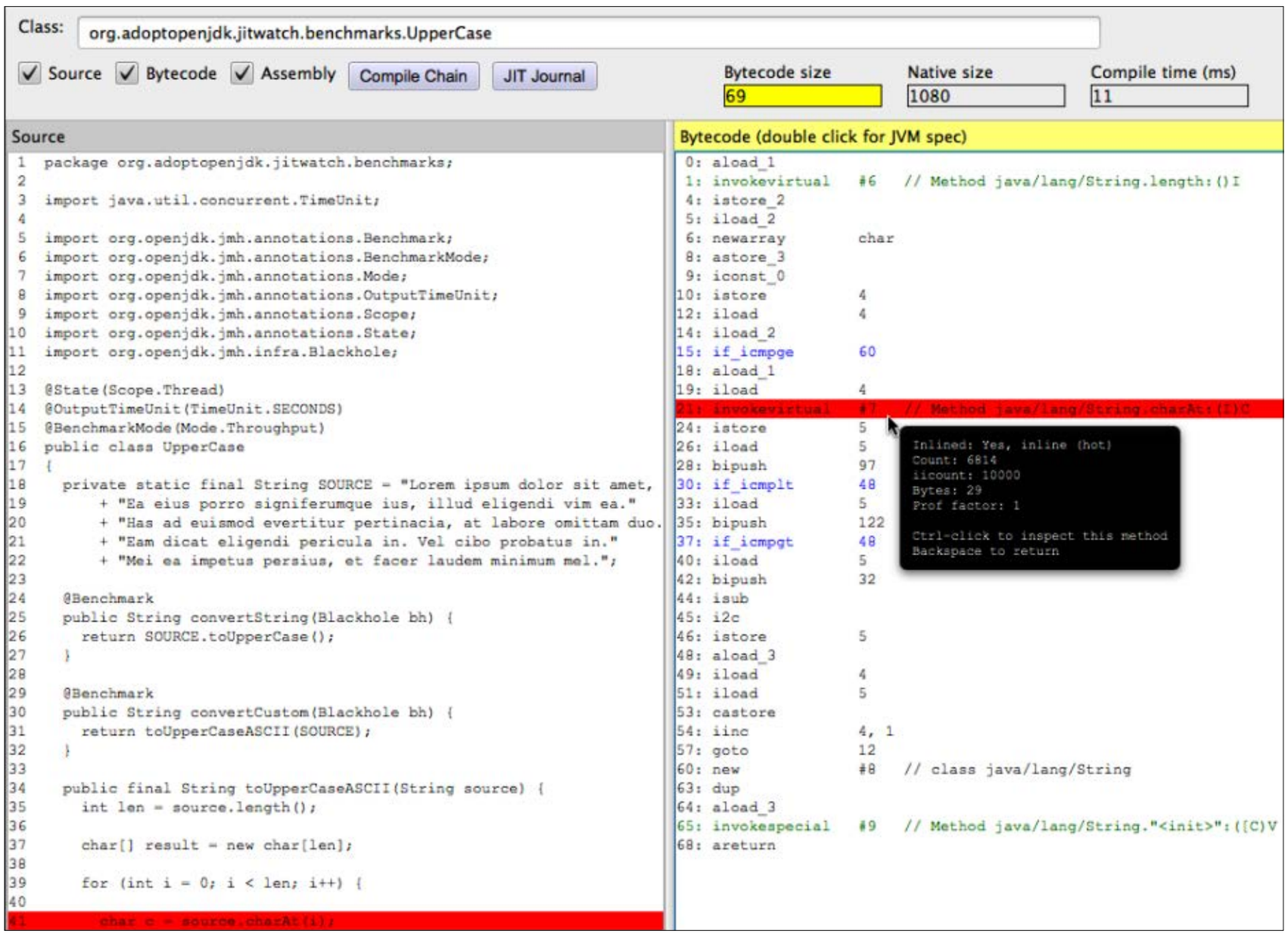


Figure 7

Conclusion

That wraps up this series on some of the optimizations that Java HotSpot VM can perform. Check out the "Performance Techniques" section of the wiki, and play with the examples that come with the JITWatch Sandbox.

If you have any questions or a bug to report, contact Chris through the GitHub project. `</article>`

MORE ON TOPIC:

Java and Performance

LEARN MORE

- GitHub repository for JITWatch

MICHAEL HEINRICHS

BIO

Part 1

The Quantum Physics of Java

An introduction to modern chip design and its effect on Java programs

Imagine you have an array with 67,000 integer elements and you run two loops over the array, as shown in **Listing 1**. Both loops multiply the elements of the array by three. However, while the first loop changes every element, the second loop modifies only every sixteenth element.

How much faster will the second loop be compared to the first? Take a guess!

The surprising answer is that if the code is executed on a typical laptop, both loops take the same amount of time. **Table 1** shows measurements from three computers. The difference is negligible. The second loop does only a fraction of the work, so how is it possible that the first loop runs as fast as the second?

To understand this behavior you have to consider how the CPU and the memory system work. On the lowest level, modern computers can show surprising behavior, very much like quantum mechanics, that seems to contradict daily experience. But sometimes quantum mechanics has noticeable effects on our "real world." And sometimes the effects of processes at the hardware level have a noticeable effect on our programs. This article takes a look at how modern computers work at the lowest level and explores the things that can affect performance.

Instructions

If you try to imagine how the loops in **Listing 1** are executed, your initial interpretation might be that the array is stored in main memory and the CPU reads element after element, multiplies each element by three, and writes the result back, as you can see in **Figure 1**. This interpretation is useful for understanding the functionality of the loops, but it's not what really happens inside a computer.

Figure 2 shows a graph of relative performance improvements that CPUs and memory went through in recent decades. Memory performance improved steadily during the whole period, but that was nothing compared to the improvements in CPU speed, especially during the 1990s. In recent years, plain CPU speed hit a limit, but do not be fooled! The scale in **Figure 2** is logarithmic. Even

	SMALL STEPS EACH ELEMENT	LARGE STEPS EVERY 16TH ELEMENT
I7-4980HQ @ 2.8 GHZ, MAC OS X YOSEMITE	30.4 MS	29.7 MS
I7-3770 @ 3.4 GHZ, LINUX MINT 14	25.8 MS	26.1 MS
T7200 @ 2 GHZ, LINUX MINT 14	193.0 MS	184.2 MS

Table 1

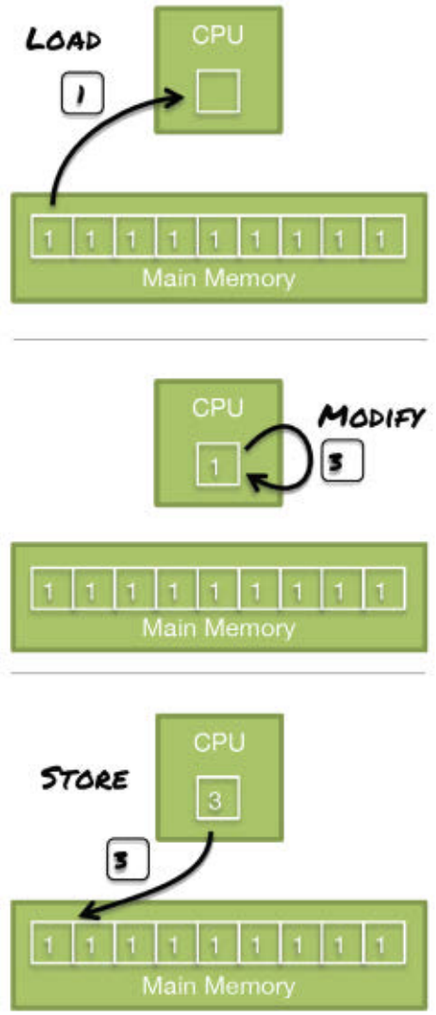


Figure 1

Java and Performance

PHOTOGRAPH COURTESY OF DEVOXX

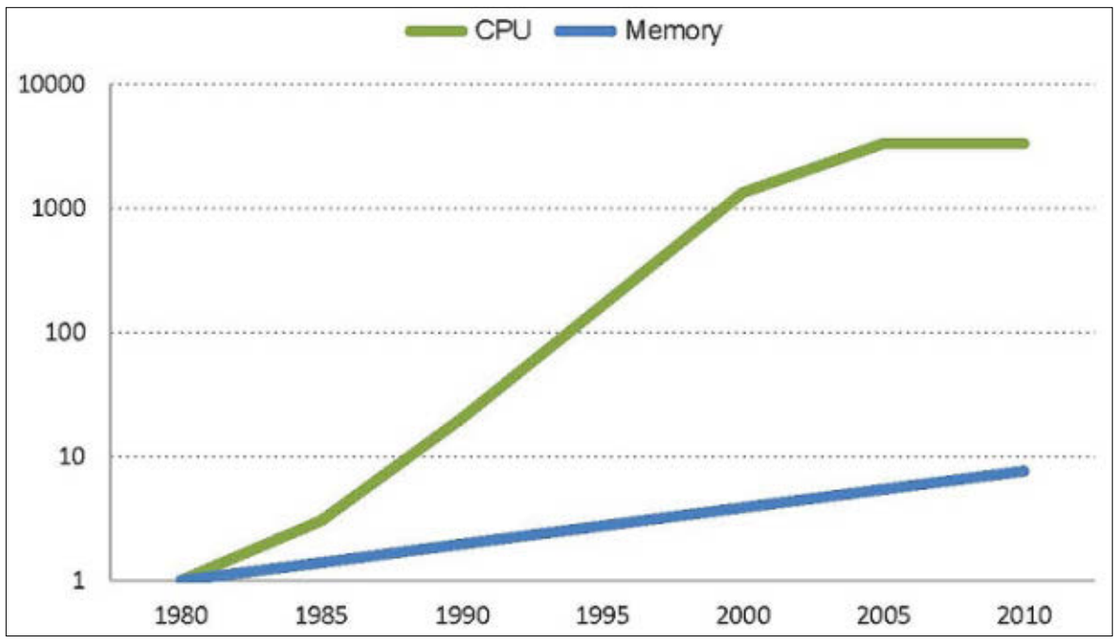


Figure 2

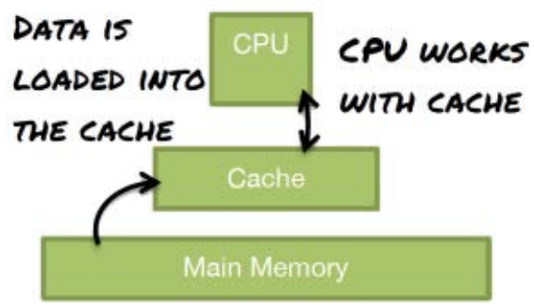


Figure 3

though it might look as if memory performance is catching up, the gap is still huge.

For our example, this means that if a computer worked as we imagined in **Figure 1**, it would be terribly slow. The CPU would wait most of the time for the memory to deliver the next element. To overcome this bottleneck, processor designers added a cache between the CPU and main memory. The cache is a smaller and much faster memory module, whose whole purpose is to mitigate the performance gap. **Figure 3** shows an improved model of the CPU and memory system.

Programs tend to access the same data and code several times within a short period of time (*temporal locality*), and memory access is often limited to small regions (*spatial locality*). This means that if you load all the data that you use into the cache, there is a high chance that you'll need it again later. And because the next time you need it the data is available in the cache, the performance of your programs increases tremendously.

Now you might wonder—if we can put faster memory between CPU and main memory, why can't we just make the whole memory faster? There are mostly two reasons. For one, main memory is a lot larger than cache, and it simply takes more time to find the right address within 16 GB (the typical size of main memory, as of this writing) than to find the right address within 8 KB (the typical size of Level 1 [L1] cache). However, probably the more important reason is that the electronic components of cache (SRAM) are much more expensive than the ones used in main memory (DRAM) in terms of heat and space. Heat and space are the limiting factors in modern chip design.

To exploit spatial locality, the cache doesn't work with individual bytes but uses cache lines instead. A *cache line* is an adjacent part of the memory, typically 64 bytes.

What really happens when you iterate over the large loops from **Listing 1** can be seen in **Figure 4**. The CPU loads a complete cache line from main memory into the cache and modifies the elements in the cache directly. The first loop modifies all elements in the cache line, while the second loop modifies only one element (16 integers, each 4 bytes long). The limiting factor in this setup is loading the cache line into the cache; it almost doesn't matter how many operations you execute on each cache line. This explains why the performance of both loops is roughly the same.

Counting instructions to estimate the performance of an algorithm is

LISTING 1

```
private static final int ARRAY_SIZE = 64 * 1024 * 1024;
public int[] array = new int[ARRAY_SIZE];


for (int i = 0, n = array.length; i < n; i++) {
  array[i] *= 3;
}


for (int i = 0, n = array.length; i < n; i+=16) {
  array[i] *= 3;
}
```

Download all listings in this issue as text

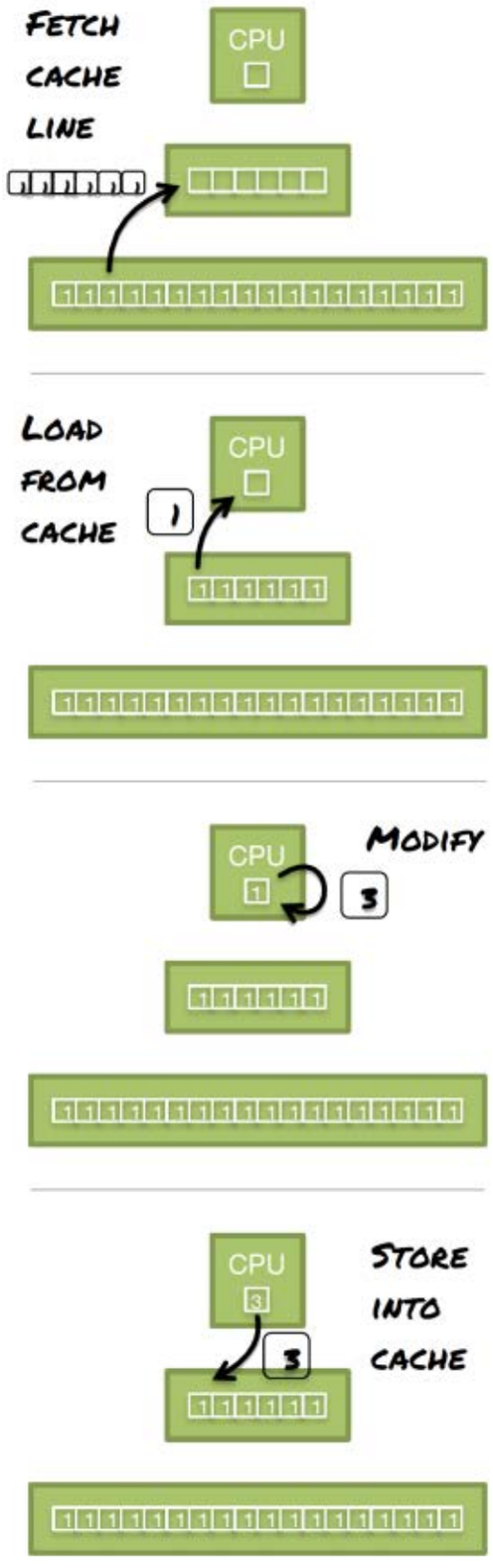


Figure 4

a useful approximation, because it's easy to assess and usually gives a good indication. But as this example shows, you have to keep in mind that it's just an approximation. In reality, the execution times of single instructions vary widely, and you can't rely on this number only.

Data Size

Does the size of a data structure affect performance? To answer this question, run a small experiment using the code in **Listing 2**. Take the second loop from the first code example and run it repetitively. This time, change the size of the array and measure the average time to run a single loop iteration. The purpose of this experiment is to run a trivial algorithm over a data structure whose size you can control. Is there a relationship between the size of the array and the time needed to modify a single element?

Before you look at the results, consider briefly what you expect. Accessing a single array element requires constant time, O(1). Thus, the inner part of the loop should be executed in constant time, too. That means for large enough arrays, you will hit an upper bound that is constant. But what happens before that? Will the execution time be constant all the way through?

Figure 5 shows the dependency between array size and access time.

LISTING 2

```
private static final int ARRAY_CONTENT = 777;
@Param({"1024", "2048", "4096", "8192", "16384", ..., "536870912"})
 public int size;

public int[] array;
 public int counter;
 public int mask;

@Setup(Level.Iteration)
public void setUp() {
   final int elements = size / 4;
   final int indexes = elements / 16;
    mask = indexes - 1;
    array = new int[elements];
   Arrays.fill(array, ARRAY_CONTENT);
   counter = 0;
    for (int i = 0; i < indexes; i++) {
      seqIndex[i] = 16 * i;
   }
}

@Benchmark
public void benchLoop() {
    array[16 * counter] *= 3;
   counter = (counter + 1) & mask;
}
```

Download all listings in this issue as text

As you can see, there is a relationship between these values. A single modification is faster if the array is small. But it's not that simple. The resulting curve looks like a staircase. The access time remains constant until the array size exceeds a specific threshold, and then it jumps to a new level where it remains until the next threshold is reached. Why is there a dependency at all, and where do these levels come from?

The cache is usually not a single unit, but consists of several levels with different sizes and access

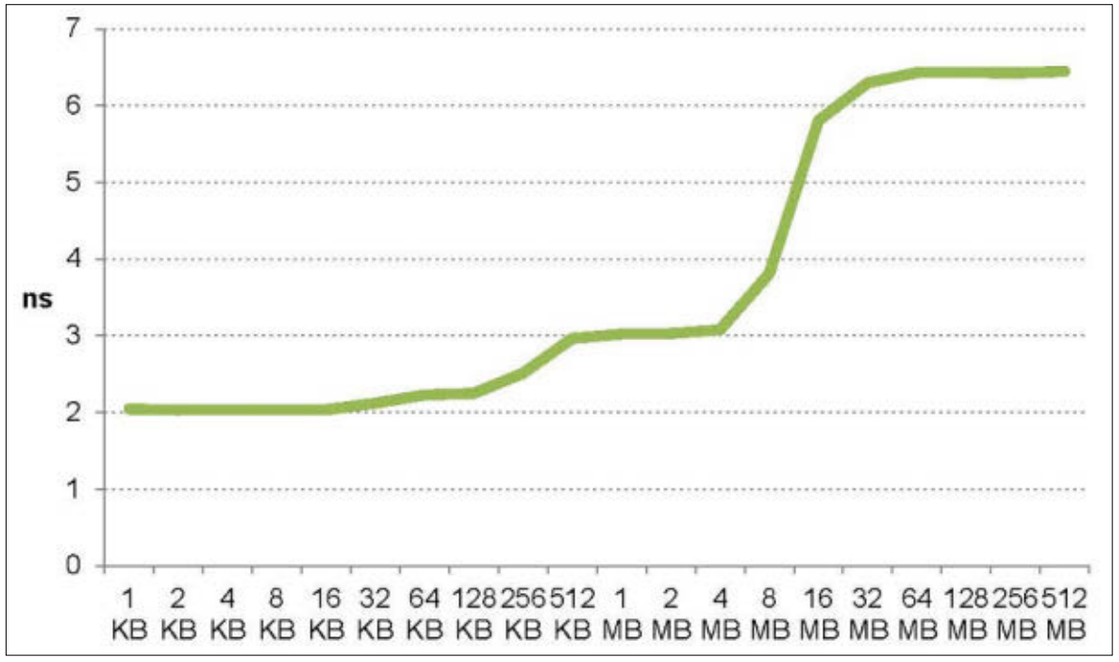


Figure 5

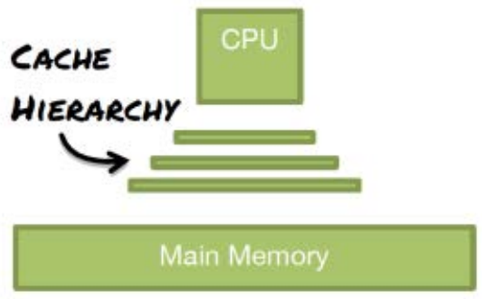


Figure 6

times. L1 cache is the smallest and fastest. Current CPUs typically have three cache levels, with each level being slower and significantly larger than the level before. **Figure 6** shows an improved version of our model that contains the cache hierarchy.

How large are the performance gaps between the different cache levels? To explain this in a form that is more accessible to human beings, my former colleague Richard Thompson came up with the beer cache hierarchy. Imagine that you're sitting in front of your TV watching your favorite team and you're thirsty.

- L1 cache is the bottle of beer in your hand. Access time is almost immediate (< 1 ns), but the quantity is extremely limited (for example, 32 KB on my system).
- L2 cache is the cooler next to your sofa. Access time is still pretty low (7 ns), and the quantity is significantly larger (256 KB, which is equivalent to 8 bottles of beer).
- L3 cache is the fridge in the kitchen. Access time is noticeably larger (25 ns), but the size is so large that the analogy falls apart (8 MB, which is equivalent to 256 bottles of beer).
- Main memory is the corner store. Access time is huge (100 ns), but the quantity of beer is probably more than enough for a lifetime (16 GB, which is equivalent to more than half a million bottles of beer).

Looking at these numbers, it becomes quite obvious why both loops in the initial example took the same amount of time. It doesn't really matter how many sips of beer you drink if you have to run to the corner store for each bottle.

With the cache level hierarchy in mind, take a look at the graph in **Figure 5**. Each plateau in the graph corresponds to a level of the cache hierarchy. As long as the array fits into L1 and L2 cache, access time is very low. But as soon as the array becomes too large and has to be read from L3 cache, access time increases noticeably. And the same happens again as soon as the array does not fit into L3 cache and has to be read from main memory. If you look closely, you can even see the small jump between the L1 and L2 cache.

Size does matter. Even though memory is cheaper than ever

LISTING 3

```
public int[] rndIndex;

@Setup(Level.Iteration)
public void setUp() {
  ...

  rndIndex = new int[indexes];
  final List<Integer> list = new ArrayList<>(indexes);
  for (int i=0; i<indexes; i++) {
    list.add(16 * i);
  }
  Collections.shuffle(list);
  for (int i=0; i<indexes; i++) {
    rndIndex[i] = list.get(i);
  }
}
```

Download all listings in this issue as text

before, try to avoid wasting it. The smaller the size of the data you use, the higher the chance that it will fit into the cache, which can lead to significantly better performance.

Access Patterns

So the size of data influences performance. Does the order in which you access your data—the data access pattern—have an influence, too? You can change the previous experiment slightly to find an answer. Instead of simply running an index through the array, create a second array that stores the access order. Access the array sequentially as before for one time, and then access it randomly and measure the difference. You can see the code for both experiments in **Listing 3**.

If you run the experiment with different array sizes and plot the result in a graph, you get two curves, as shown in **Figure 7**. Not surprisingly, you can see the already familiar staircase pattern. Both curves show similar access times on the lower two levels, which correlate to the L1 and L2 caches. But on the third level, the performance of the sequential access pattern is noticeably better. The fourth level shows a significant difference. Why is the access order insignificant for small arrays, but plays a major part for large arrays?

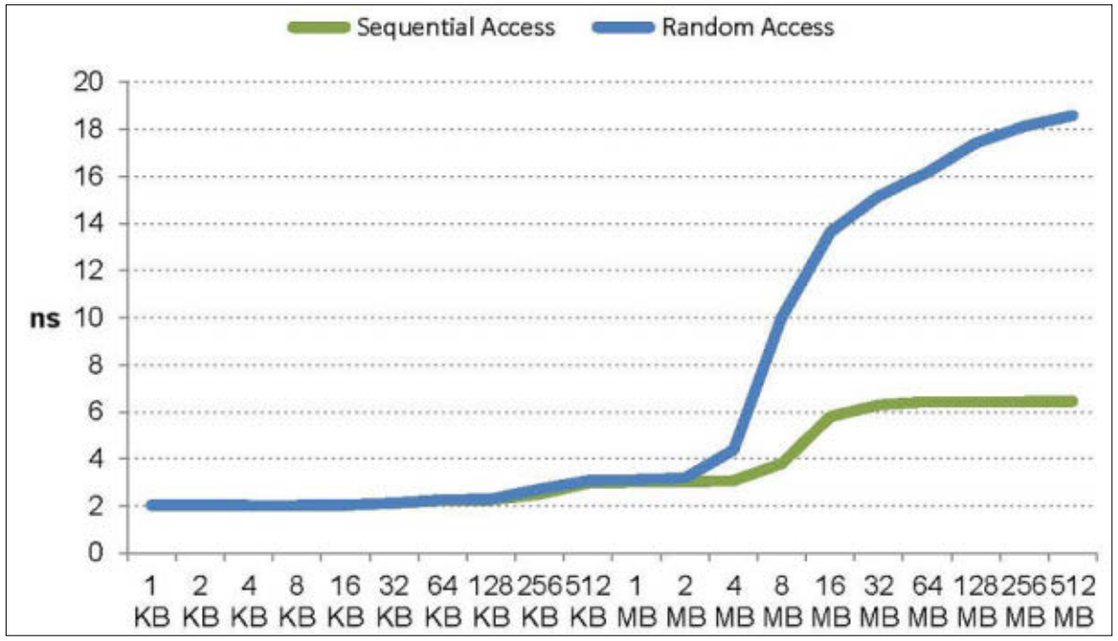


Figure 7

```
michael@desktop ~/repos/quantum/target/classes $ perf stat -p 3195 -B -r 40 sleep 5

 Performance counter stats for process id '3195' (40 runs):

      4997,277608 task-clock                #    0,999 CPUs utilized           ( +-  0,00% ) [100,00%]
              529 context-switches          #    0,106 K/sec                   ( +-  0,04% ) [100,00%]
                0 CPU-migrations            #    0,000 K/sec                   [100,00%]
                0 page-faults               #    0,000 K/sec                   ( +- 35,51% )
   19.429.967.708 cycles                    #    3,888 GHz                     ( +-  0,00% ) [83,28%]
   19.006.043.778 stalled-cycles-frontend   #   97,82% frontend cycles idle    ( +-  0,00% ) [83,31%]
   18.296.349.545 stalled-cycles-backend    #   94,17% backend  cycles idle    ( +-  0,01% ) [66,74%]
    1.386.651.837 instructions              #    0,07  insns per cycle
                                            #   13,71  stalled cycles per insn ( +-  0,05% ) [83,37%]
       23.934.690 branches                  #    4,790 M/sec                   ( +-  0,07% ) [83,37%]
           23.139 branch-misses             #    0,10% of all branches         ( +- 11,00% ) [83,30%]

       5,000586757 seconds time elapsed                                        ( +-  0,00% )
```

Figure 8

A Valuable Tool

To get a better understanding of what is going on inside the computer, you can use the Linux profiler tool perf, which collects and prints out events generated by the CPU and the memory system while a program is executed. The command-line interface for perf is similar to that of git. You call perf with the command you want to execute:

```
perf COMMAND [ARGS]
```

To get a list of all commands, use perf –help. To get help for a specific command, you can run:

```
perf help COMMAND
```

The most useful command is stat. It allows perf to run another program and tracks hardware events during execution:

```
perf stat [ARGS] PROGRAM
```

Without any arguments, this command starts PROGRAM, tracks some general events, and prints out the statistics as soon as the program ends. You can see a typical output in **Figure 8**. You can

specify which events should be tracked by using the -e option. To get a list of supported events, run the command:

```
perf list
```

In Java, you usually don't want to track the whole program, because this would include bootstrapping the VM, JIT compilation, and so on, which typically we're not interested in. However, with perf you can hook into a running process with the -p option. Perf is a little weird when used with the -p option, because you still have to specify a program that will be executed, and the measurement ends once this program ends. Typically you'd use the sleep command, which enables you to specify the duration of the test.

First, measure both your loops using the default settings of perf. This provides a good overview. The most-interesting results can be seen in **Table 2**. The number of stalled front-end and back-end cycles differs significantly between both runs. A stalled cycle means the CPU is idle and waiting for something. One of the most likely causes of a stalled front-end cycle is a cache miss, which results in the CPU waiting for data to arrive from main memory or a slower cache. To validate this assumption, you can run perf again, but this time to specifically measure cache loads and cache misses. You can see the result in **Table 3**.

The ratio between successful cache loads and cache misses differs tremendously. When accessing the array in sequential order, only about 6 percent of all memory loads result in a cache miss. But when accessing the array in random order, two out of three loads result in a cache miss. The high number of cache misses is expected, because the array doesn't fit into the cache, and you have to load everything from main memory. But why are there almost no cache misses when you access the array sequentially?

	SEQUENTIAL ACCESS		RANDOM ACCESS	
CYCLES	19,430,435,800		19,429,967,708	
STALLED FRONT-END CYCLES	7,217,361,632	(37.14%)	19,006,043,778	(97.82%)
STALLED BACK-END CYCLES	843,462,646	(4.34%)	18,296,349,545	(94.17%)

Table 2

	SEQUENTIAL ACCESS		RANDOM ACCESS	
L1 CACHE LOADS	5,758,001,370		170,655,221	
L1 CACHE MISSES	360,757,378	(6.27%)	365,959,699	(214.44%)

Table 3

Loading data from main memory into the cache is often a major bottleneck. For this reason, the CPU tries to help by guessing which data you'll use next and loading it into the cache in the background, as you can see in **Figure 9**. While you modify the elements of a cache line, functionality called the *prefetcher* loads the next cache line into the cache. Thus, when you need the data, it's already available in the cache.

The prefetcher is not particularly smart. It can guess the next memory location correctly only if the memory loads follow a regular pattern. In the first case, when you went through the array sequentially, guessing the next memory location was easy, and the prefetcher could mitigate a substantial part of the performance loss by prefetching the next memory location. But when you accessed the array randomly, guessing the next memory location correctly was impossible, and the algorithm had to wait until data was loaded from main memory.

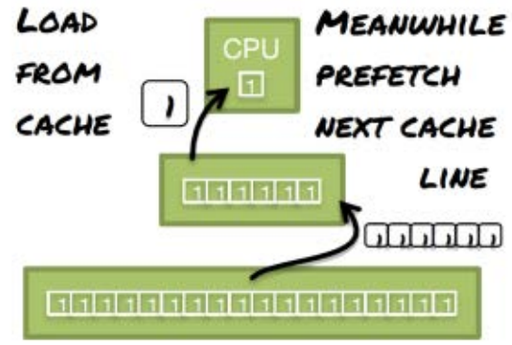


Figure 9

The access pattern has a significant influence on the performance of an algorithm, but the chances to use this knowledge within Java are limited. You have close to no control over how your data is arranged in memory. But there is hope. The proposed *value types* might provide this ability one day, because they allow you to arrange object-like structures sequentially in memory.

Another Valuable Tool

Microbenchmarking is another valuable tool for getting more insight into your programs. Probably the most important rule of microbenchmarking is to always use a tool that helps you to avoid some of the many pitfalls, such as the insufficient warmup of the VM, dead code elimination, and loop unrolling. The Java Microbenchmark Harness (JMH) from Oracle is probably the best harness available right now.

ANNOTATION	DESCRIPTION
@BENCHMARKMODE	SPECIFIES WHAT SHOULD BE MEASURED—FOR EXAMPLE, THROUGHPUT OR AVERAGE TIME.
@OUTPUTTIMEUNIT	SPECIFIES THE TIME UNIT USED IN THE OUTPUT—FOR EXAMPLE, TimeUnit.MILLISECONDS.
@WARMUP	SPECIFIES THE WARMUP PHASE. YOU CAN SET THE NUMBER OF ITERATIONS AND THE DURATION OF A SINGLE ITERATION.
@MEASUREMENT	SPECIFIES THE MEASUREMENT PHASE AND IS SIMILAR TO @WARMUP.
@FORK	SPECIFIES HOW OFTEN YOU WANT TO FORK THE JAVA VIRTUAL MACHINE (JVM) AND RUN THE TESTS. YOU SHOULD ALWAYS DO RUNS IN SEVERAL FORKS.

Table 4

Tests written for JMH are similar to JUnit tests. The code you want to benchmark needs to be in a single method, which must be annotated with @Benchmark. The test can be configured with annotations at the class level. **Table 4** shows the most-important annotations and their meaning.

Conclusion

Most of the time, processes at the hardware level have no significant effect on programs, but sometimes they do. Therefore, it's useful to have a rough understanding of what goes on at the hardware level and to keep up with the latest developments. Besides being useful, this knowledge is also fascinating and a great way to impress your fellow developers.

This was the first part of a two-part series about the quantum physics of Java. This part focused on memory and, specifically, the cache hierarchy. The second part looks a little more into memory and then takes a deep dive into the inner workings of a modern CPU. `</article>`

MORE ON TOPIC:

Java and Performance

LEARN MORE

- "What Every Programmer Should Know About Memory"
- Igor Ostrovsky Blogging
- Martin Thompson's blog, Mechanical Sympathy
- "Linux kernel profiling with perf" tutorial

JOSÉ PAUMARD

BIO

Shakespeare Plays Scrabble

New ways of solving problems with the Stream API in Java SE 8

The Collections Framework, first released with Java 2, was designed on the Iterator patterns (among others) and is at the heart of all our applications. With the inclusion of generics in Java 5, the Collections Framework was rewritten, but the patterns for using it weren't modified.

Java SE 8 introduced lambda expressions and the Stream API. The Stream API brings new patterns and new ways of solving the problems we encounter in business applications that were previously solved with the Collections Framework. This article shows how you can use the Stream API to solve a simply stated problem built on the Scrabble game in just a few lines of code.

Shakespeare and Scrabble

Let's say we have two files. One contains (supposedly) all the words Shakespeare used in his plays and poems. The other contains all the words in *The Official Scrabble Players Dictionary*.

The question is this: How well would Shakespeare have played Scrabble? Or, what is the best word Shakespeare could have played, and how much would it have scored?

It turns out that this question is a map/filter/reduce question that can be solved with the Stream API.

Compute the Score of a Single Word

Computing a score is about taking a given word and returning a score that is an integer. This can be done with `Function<String, Integer>`, which can be implemented with a lambda expression.

Let's build an array, `letterScores`, which gives the score of each letter in the English Scrabble game (**Listing 1**). From this array, we can see that the score of *m* is 3, and the score of *y* is 4.

Inside this scoring function, we also need a mapping function to get the score of a given letter. Let us write this mapping function first. It is very simple and uses the old trick of reading the correct cell of the array that holds the scores of the letters (`letterScores`).

```
IntUnaryOperator letterScore =
  letter ->
  letterScores[letter – 'a'];
```

This mapping function is an `IntUnaryOperator`, a specialized type of function that takes and returns an `int`.

We can then write the scoring function using the mapping function:

```
Function<String,
Integer> score = word ->
  word.chars() // IntStream
    .map(letterScore)
    .sum();
```

Having the `chars()` method added to the `String` class proves handy. Many classes have new methods that return streams on their internal structure.

The choice has been made to represent letters by integers, and the `chars()` method returns an `IntStream` instead of a `Stream<Integer>`.

Three streams of primitive types (`IntStream`, `LongStream`, and `DoubleStream`) are created for performance reasons. Using them is much more efficient than using the equivalent streams of objects, because we don't have to pay the cost of boxing and unboxing their elements at each step. But sometimes we need to convert this `IntStream` into a `Stream<Integer>`, which is the case when we have to build histograms later.

How Shakespeare Performs

Now that we can compute the score of a given word, building the histogram of Shakespeare's words by their

PHOTOGRAPH COURTESY OF THE BELGIUM JAVA USER GROUP (BEJUG)

scores is easy. A histogram is a simple map, computed by grouping the words by their scores. Thus, the keys of that map are the scores of the words, and the values are the words themselves, regrouped into lists.

Are we interested in all the content of the histogram? No. What we want to see is only the best scores. So we need to select the best key-value pairs from that map—with the best being the ones that have the greatest values for their keys. A simple way of computing that is like this:

1. Build the map.
2. Sort the map according to its keys, in descending order.
3. Extract the best (for example, the first three) key-value pairs.

Build the map. Building a map by regrouping is done with a collector. Collector is a new interface in the Java SE 8 Stream API that provides a mutable reduction. *Mutable* here means that we reduce our stream into a mutable container—a map, in the example shown in **Listing 2**.

Sort the map according to its keys. In **Listing 2**, shakespeareWords is a collection of all the words of Shakespeare, from which a stream can be obtained. We can regroup these words by the scores of each word. The Collectors.groupingBy() code does just that. It exists in several versions, taking different sets of parameters. Here, we pass a supplier to build the resulting map, which is implemented by a TreeMap. We pass a comparator as a parameter to its constructor to have the map sorted in the descending order of its keys. The last parameter is a downstream collector to specify that we want our words in a list.

We added an extra filtering step to make sure that we process only words that are allowed in Scrabble.

Extract the best key-value pairs. Selecting the best key-value pairs can easily be done with the Stream API, but we need to use a trick here, because the Map interface doesn't define a stream() method. We can't build streams directly on maps in Java SE 8. The trick is to build that stream on the set of the key-value pairs, which can be built using the entrySet() method call. This method returns a Set<Map.Entry<K,V>>, on which we can build a stream. Because we built a TreeMap, this set is in fact a NavigableSet, which keeps the order of the key-value pairs. Displaying the three best elements becomes an easy task:

```
wordsByScore.entrySet()
 .stream()
 .limit(3) // or any value
 .forEach(System.out::println);
```

LISTING 1 LISTING 2 LISTING 3 LISTING 4

```
private static final int [] letterScores = {
// a, b, c, d, e, f,  g, h, i,  j, k, l, m, n, o, p,  q,  r, s, t, u, v, w, x,  y, z
   1, 3, 3, 2, 1, 4,  2, 4, 1, 8, 5, 1,  3,  1, 1,  3, 10, 1, 1, 1,  1, 4,  4, 8, 4, 10}
```

Download all listings in this issue as text

At this point, the best word is *whizzing*, which is worth 33 points.

Not All Words Can Be Written

Take a closer look at that word *whizzing*. Can we really write that? Notice that it contains two occurrences of the letter *z*, but only one is available in the Scrabble game. How can we discard such a word?

What we need to do to tell whether a given word can be written with the letters that are available in the game is to count them, letter by letter, and compare the number of needed letters to the number of available letters. If this test fails, then the word cannot be written.

This operation can be done with an allMatch() call. Counting all the letters of a given word consists of building a histogram in which the keys are the letters, and the values are the number of times each letter is used to write that word. Let us first build this histogram (see **Listing 3**).

The call to the boxed() method converts the IntStream returned by the chars() call into a Stream<Integer>. This is necessary because the collect(Collector) method we want to use is defined on streams of objects, not on streams of primary types (IntStream, LongStream, and DoubleStream).

Then we need to build a stream on its key-value pairs and compare those entries with the available letters in the game, as shown in **Listing 4**. In each pair, the key is the letter and the value is the number of times that letter is used in the word.

The number of available letters in the game is given by reading the array scrabbleAvailableLetters, declared at the beginning of **Listing 4**. From this array we can see that nine *i*'s are available, and only one *z*.

Using this predicate, we can now add a filtering step in the computation of the best word.

At this point, the best word is *squeezes*, which is worth 26 points.

What About the Blanks?

Thinking again, though, we realize that we could write the word *whizzing*. The game uses blank letters, and we could use one for the second *z*. Using blanks has a double impact. First, it changes the way we filter out the words. Second, it changes the way we compute the scores of the words, because a blank does not score any points.

THINK SOLUTIONS
Java SE 8 requires us to **think about the solutions** for our problems differently. As a bonus, we get cleaner and more-efficient code.

Write a word using blanks. Computing the number of blanks needed to write a given word is also a matter of comparing the histogram we built for the given word to the histogram of the available letters in the game. Instead of comparing the number of letters one by one, we count the number of letters that are greater than the ones available. Subtracting the needed letters from the available letters does what we need: If the result is negative, then we don't need a blank. If not, then the result is the number of blanks needed for that letter. All we need to do is compute which is greater, this number or zero, for each letter of the given word. Summing the results gives us the number of blanks needed for that word.

First, write a function that takes a key-value pair from the word's histogram and returns the number of blanks for a given letter (see **Listing 5**). We can then use this function to compute the number of blanks for a word, as shown in **Listing 6**.

We can now add a filtering step in the computation of the best word, built on the following predicate, knowing that we have two blanks in the game:

```
Predicate<String> checkBlanks =
  word ->
  nBlanks.apply(word) <= 2;
```

Compute the score, including blanks. Computing the score of a word can't be conducted with the scoring method we used earlier, because all the letters don't necessarily count. What we can do is, once again, use this histogram and multiply the score of each letter with the number of the letter used, maxed by the number of available letters.

Let us write a function that computes the score of a given key-value pair from this word's histogram. See **Listing 7**. We can then use the function created in **Listing 7** to compute the score of a word, as shown in **Listing 8**.

Using the new scoring function created in **Listing 8** instead of the first one we created, and adding the filtering step to check whether the word can be written with two blanks, confirms that *squeezes* is the best word with 26 points. The first word we found, *whizzing*, isn't far away, with 23 points.

Place the Word on the Board

But our problems aren't quite over yet. What about placing the word on the board? We have two special squares, not far from the central one, that provide a bonus because they are marked "DOUBLE LETTER SCORE." Would it be possible to take those squares into account to compute the final score of our first move?

First, notice that one of these squares is four squares away to the right of the central square. So it will give a bonus to a word that contains five or more letters. For a word that has between five letters and seven letters (the maximum any player ever has), there are three placement possibilities to the right of the central square and three other possibilities to the left of the central square. So, for a seven-letter word, optimizing the placement of the word determines the highest-scoring letter from the word's first three and last three letters.

To compute the final score, we need to find this special letter, add the normal score that we previ-

LISTING 5 LISTING 6 / LISTING 7 / LISTING 8

```
ToLongFunction<Map.Entry<Integer, Long>> blank =
        entry ->
        Long.max(
          0L,
          entry.getValue() -
          scrabbleAvailableLetters[entry.getKey() - 'a']
        );
```

Download all listings in this issue as text

ously computed, multiply by two (for the first-move bonus), and add 50 points if the word is seven letters long (the bonus for using all our letters in one word). Of course, the hard part is to find the special letter.

The first and last three letters of a given word can be extracted with the code shown in **Listing 9**. These two streams hold the letter that will be double-scored. Note that if the word we consider is shorter than four letters, the streams will be empty. But that's not a problem, because the stream.max() method returns an instance of Optional, which can be empty.

How can we merge those two streams into one? We have two solutions for that: Use either the concat() method or the of() method. Because the concat() call is stateful, we might think it won't be as efficient in parallel as of(), so let's use of() (see **Listing 10**).

This merged stream of letters still needs to be mapped to a stream of scores for the letters, and then the maximum score needs to be determined. Because there are words of fewer than four letters, the two streams can be empty and merging them will also return an empty stream. So we should ensure that we cope with an empty Optional correctly. If we have no letter that can be placed on the "DOUBLE LETTER SCORE" square, this function should return zero, meaning that no bonus will be awarded. See **Listing 11**.

We can then write our last method, which computes the full score of the best first move, as shown in **Listing 12**.

We should also add another filter step to remove words longer than seven letters, because seven is the maximum number of letters allowed in a Scrabble word. With this new scoring function, we can see that there are two best words, *jezebel* and *quickly*, for a score of 120 points. Not bad for an opening move!

LISTING 9 LISTING 10 LISTING 11 LISTING 12

```
Function<String, IntStream> first3 =
  word -> word.chars().limit(3);
Function<String, IntStream> last3 =
  word -> word.chars().skip(Integer.max(0, word.length() - 4);
```

Download all listings in this issue as text

Conclusion

The problems we solved in this article might look like trivial game problems, but they solve many of the problems we face in general business applications: building histograms, comparing them, extracting maximum values, and handling corner cases raised by empty streams. Using the Stream API, we can solve these problems without much code—and the solution is still efficient from a pure performance point of view.

We used some of the new patterns introduced in Java SE 8, but the Iterator and the if-then-else patterns are absent from the solution. As we can see here, Java SE 8 requires us to think about the solutions for our problems differently. As a bonus, we get cleaner and more-efficient code.

The Official Scrabble Players Dictionary *and the words in the works of Shakespeare, as well as many other interesting data sets, can be found on Robert Sedgewick's page "Real-World Data Sets."* `</article>`

LEARN MORE

- Stream API
- Collector interface
- Java 8

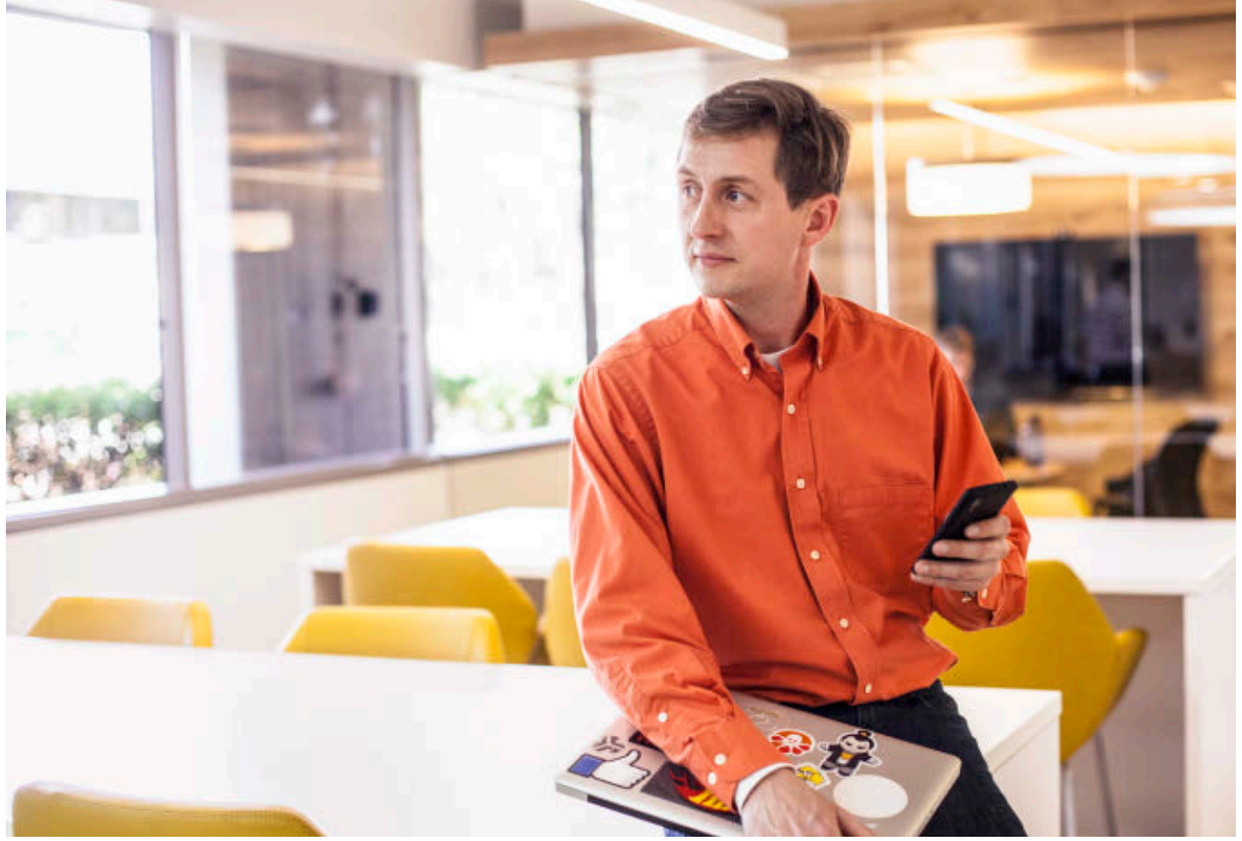
BIG DATA AND JAVA

Experts answer your questions.

BY FABIANE NARDON, FERNANDO BABADOPULOS, DANIEL TEMPLETON, SIMON ELLISTON BALL, AND CHRIS RICHARDSON

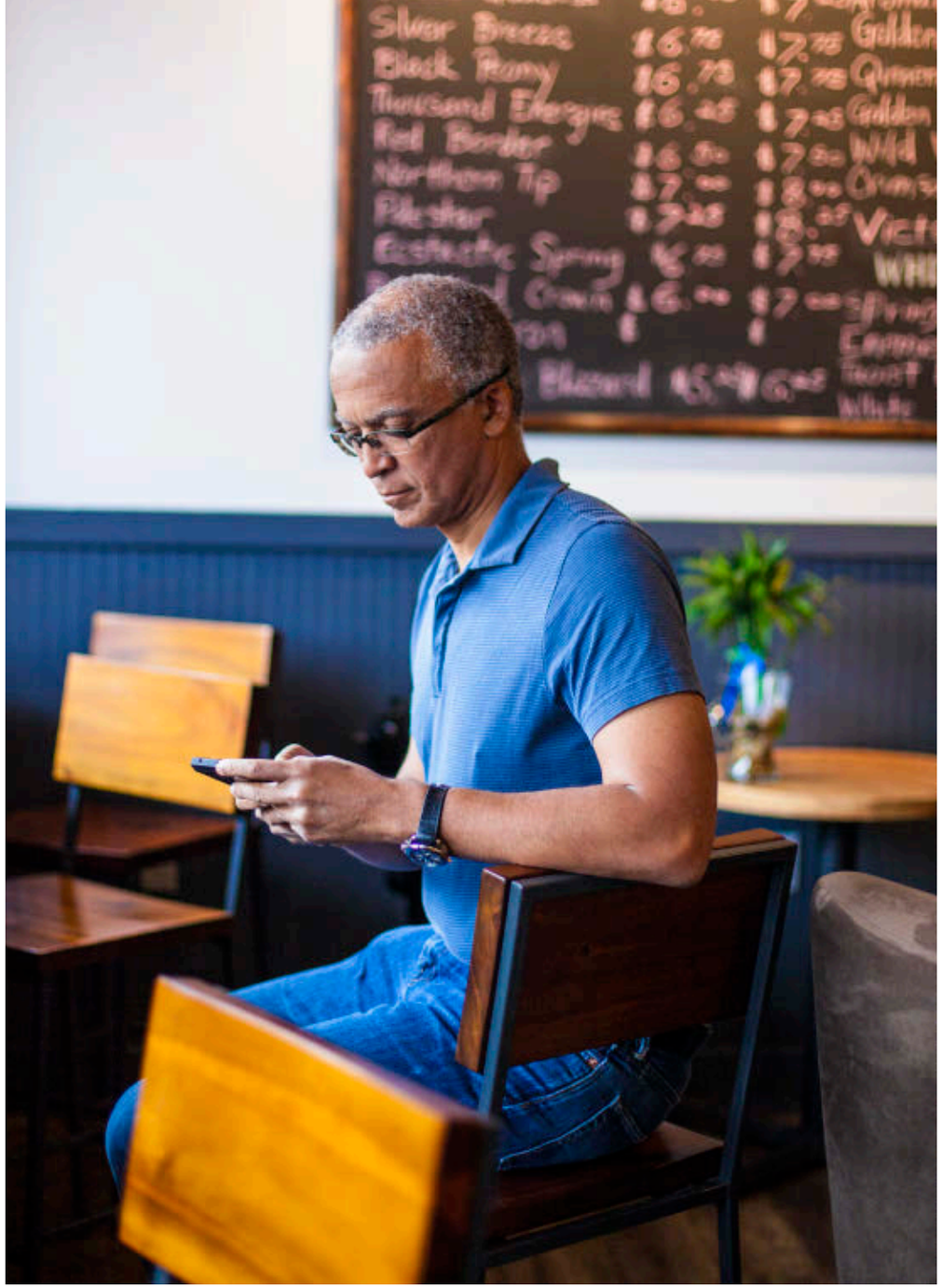
With tools such as Apache Hadoop, Apache Hive, Apache Pig, Apache HBase, and others, Java has become the driving force behind the big data revolution. As a Java developer, are you prepared to be part of this new generation of Java applications? Do you have a million questions and nobody to ask? See how these big data experts answered the best questions posed by the audience during the "Big Data and Java: Ask the Experts" panel at JavaOne 2014.

Fabiane Nardon, a Duke's Choice Award winner, is the chief scientist at Tail Target, where she architects new disruptive data science services. Fernando Babadopulos, an expert in designing high-performance software, is the CTO at Tail Target. Daniel Templeton has focused on big data since 2009 and works at Cloudera building its developer and data science Cloudera Certified Professional certifications. Simon Elliston Ball, a solutions



Comparing Hadoop to a traditional relational system, Daniel Templeton (top) says, "There might be more differences than similarities." Fabiane Nardon (bottom): "The choice of enterprise-ready NoSQL database you need depends on the problem you want to solve."

PHOTOGRAPHS BY BOB ADLER/GETTY IMAGES AND PAULO FRIDMAN



RICHARDSON:
"Big data isn't necessarily a large volume of data. It can be data that is generated at a high velocity. Big data can also be data that has a lot of variety, such as unstructured data."

Question: What is big data?
Richardson: *Big data* is a somewhat nebulous term that describes data that can't be processed by traditional data processing techniques, such as an RDBMS-based application running on a single machine. Big data isn't necessarily a large *volume* of data. It can be data that is generated at a high *velocity*. Big data can also be data that has a lot of *variety*, such as unstructured data.

Big data technology, such as Apache Hadoop, tackles the problems of volume and velocity by scaling horizontally using fault-tolerant software, which tends to be cheaper and more scalable than the more traditional approach of vertically scaling very reliable hardware. Apache Hadoop deals with variety by using storage formats that support both unstructured and structured data. Machine learning (ML) algorithms are commonly used to process big data.

Question: What are the main NoSQL databases?
Nardon: There are plenty of enterprise-ready NoSQL databases out there. The choice depends greatly on the problem you want to solve. Considering the popular data model classification, some good examples are

- Column store: Apache Cassandra, Apache HBase, and Apache Accumulo
- Document store: MongoDB, CouchDB, and RavenDB
- Key-value store: Redis, Riak, Amazon DynamoDB, Aerospike, and FoundationDB
- Graph store: Neo4j, Titan, OrientDB, Apache Giraph, and InfiniteGraph

Question: What is the best way to get started with Apache Hadoop?
Templeton: Hadoop is complex, and getting started with it can be daunting. However, many tutorials and training options are available to help you.

First, go buy (and read) a copy of *Hadoop: the Definitive Guide*. It's the single, most referenced and useful text on Hadoop.

Online, you'll find many tutorials. The venerable Yahoo! Hadoop Tutorial is still a good place to start. You should also look at the "Hands-on Hadoop" lab,

"Big data … can't be processed by traditional data processing techniques," says Java Champion Chris Richardson.

engineer at Hortonworks, helps customers solve problems with Hadoop. Chris Richardson, a Java Champion, is the author of POJOs in Action *and the founder of the original Cloud Foundry, a Java platform as a service (PaaS) for Amazon EC2. He now consults on microservices and works on his third startup.*

PHOTOGRAPH BY BOB ADLER/GETTY IMAGES

Fernando Babadopulos, CTO, and Fabiane Nardon, chief scientist, both at Tail Target, discuss how much data you need to start using Hadoop and NoSQL.

BABADOPULOS: "Because Hadoop needs to set up the job and synchronize all nodes to process the data, the more data it has to process at once, the more efficient it will be."

which was session HOL4041 at JavaOne 2014. At the time of this article, Oracle had not yet posted the lab materials online. The full lab is available on GitHub. The lab manual includes instructions for performing the lab outside of the JavaOne environment. The slides used during the lab are also online.

For more hands-on training, Cloudera has the most popular set of training courses, with options such as data analysis with SQL on Hadoop, introduction to data science, and Apache Spark.

To get your hands dirty, Cloudera offers Cloudera QuickStart VM and Hortonworks offers Hortonworks Sandbox. Both are VM images that include a complete single-node cluster. Cloudera also offers Cloudera Live, which is a live, dedicated cluster hosted for free (for up to two weeks) in the cloud.

An easier entry point for Hadoop than MapReduce is a higher-level project, such as Apache Hive, Apache Pig, or Cloudera Impala. Because Hive and Impala use a language that is close to SQL, the learning curve is often shallow. Tutorials for all three are online. The Hortonworks Sandbox includes Hive and Pig. The Cloudera QuickStart VM and Cloudera Live include all three.

If you're just getting started with Hadoop, you should also take a look at Apache Spark, which is en route to replace MapReduce as the core computation engine in the Hadoop ecosystem. The available resources for learning Spark are still somewhat limited. A good source of learning materials is the latest UC Berkeley AMP Camp archive. Spark is also much easier to simply download and play with, because it offers a standalone mode. Spark is included in the Cloudera QuickStart VM, Cloudera Live, and the Hortonworks Sandbox.

Question: How much data do you need to start using Hadoop and NoSQL?

Babadopulos: It depends on the kind of job Hadoop is running. Because Hadoop needs to set up the job and synchronize all nodes to process the data, the more data it has to process at once, the more efficient it will be. To optimize space inside the Hadoop Distributed File System [HDFS], files must be bigger than the size specified by the dfs.namenode.fs-limits.min-block-size property.

Nardon: With NoSQL databases, many times the decision of using one depends more on the need to use a more flexible data model than on the amount of data itself. For example, even if you don't have a huge documents database, it might be more convenient and natural to store your documents in a MongoDB database than in a traditional relational database, because the MongoDB data model is optimized for document-like structures.

Question: How can you move data into Hadoop if you aren't using NoSQL?

Nardon: If you use a relational database

PHOTOGRAPH BY PAULO FRIDMAN

and want to move the data into HDFS, process it, and then maybe move the result back to the relational store, you can use Apache Sqoop. Alternatively, you can use the Hadoop DBInputFormat component to get relational data into Hadoop.

If you have your data in log files, you can use Apache Flume to get the data into HDFS.

Question: Is it possible to use Apache Hadoop without needing to code in Java?

Templeton: Absolutely! There's a long list of options that let you use Hadoop without Java. What follows is a quick summary of the most-popular options.

- SQL on Hadoop: You now have a myriad of options to run SQL or SQL-like queries against your data stored in Hadoop. The two most popular are Hive and Impala, both of which are Apache-licensed open source projects.
- Apache Spark: Spark is the newest addition to the Hadoop family and might eventually replace MapReduce as the job execution engine. Spark supports code written in Scala, Python, and Java.
- Hadoop streaming: Hadoop includes a package called streaming that uses a simple input/output contract to let you write MapReduce jobs in *any* language you like. In fact, a common use of Hadoop streaming is to execute standard Linux commands such as grep, cut, and awk against a large data set.
- Pig Latin: Another common non-Java option is Apache Pig, which lets you manipulate data in a high-level language called Pig Latin.

Question: What is the largest amount of data that Hadoop can handle?

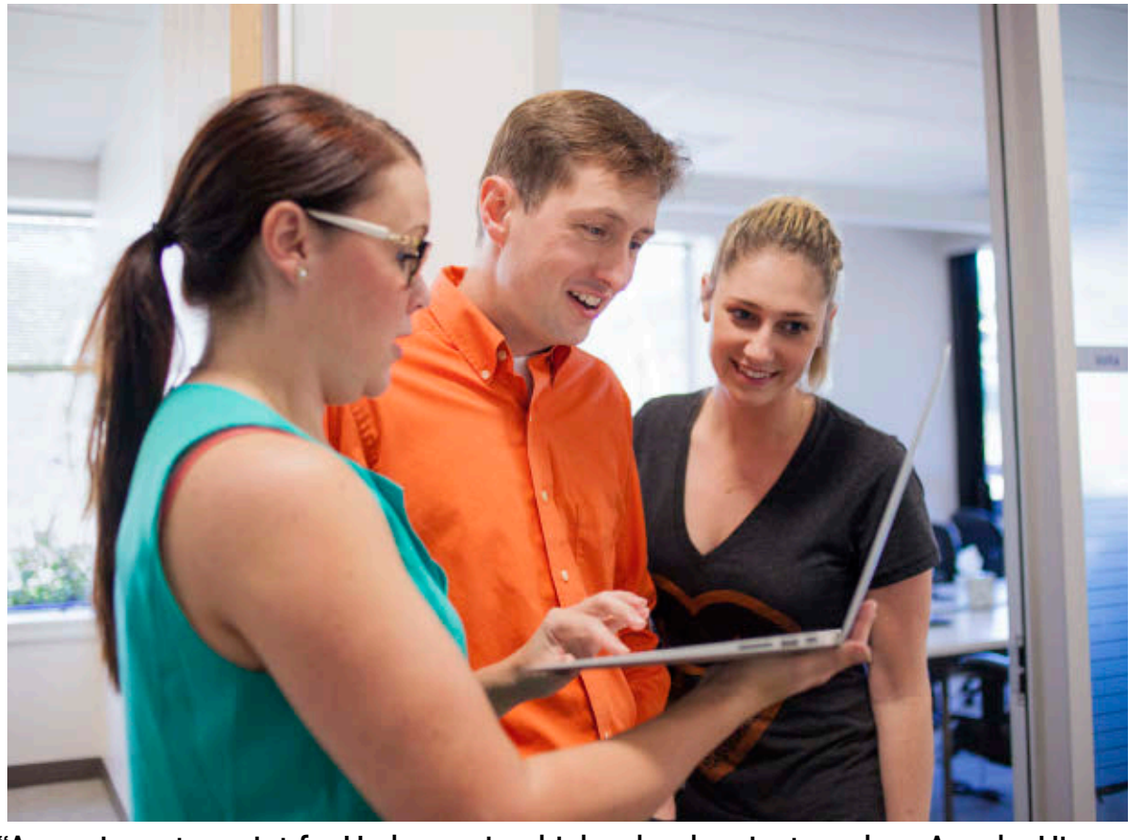
Babadopulos: Hadoop can store data inside a proprietary distributed file system called HDFS as well as in cloud-based storage systems—for example, Amazon's Simple Storage Service [S3]. The largest amount of data Hadoop can handle is limited by the amount of data those file systems can store. Because HDFS is a distributed file system, the cluster can be scaled up to support more data as needed. A DataNode can also use more than one disk to optimize and increase storage. The storage capacity of HDFS clusters can be increased either by adding a new DataNode to the cluster or by adding disk volumes to existing DataNodes.

Question: How does Apache Hadoop differ from a traditional relational system in terms of infrastructure?

Templeton: There might be more differences than similarities. The following are the two most significant infrastructure differences:

- Hadoop is built on an inherently distributed model. Hadoop assumes that clusters are composed of a large number of inexpensive, unreliable nodes, and so the entire system is built around redundancy and fault tolerance. The typical relational system is built as a centralized system that runs on a small number of powerful and reliable nodes.
- In Hadoop, the data is completely decoupled from the metadata or table structure. This approach is called *schema on read*. For example, you can upload data into HDFS and then create a table over that data in Apache Hive. The table metadata exists only in the Hive metastore; the data itself is unaffected. If you

TEMPLETON:
"Hadoop is complex, and getting started with it can be daunting. However, many tutorials and training options are available to help you."



"An easier entry point for Hadoop ... is a higher-level project, such as Apache Hive, Apache Pig, or Cloudera Impala," says Daniel Templeton at Cloudera.

PHOTOGRAPH BY BOB ADLER/GETTY IMAGES

"The Lambda Architecture is a much-debated topic in the big data community at the moment," says Simon Elliston Ball at Hortonworks.

PHOTOGRAPH BY JOHN BLYTHE

change your mind about the table schema, you can drop the table (without affecting the data) and create a new one. You can even create more than one table over the same data files. Relational systems are *schema on write*, meaning that the data and the table metadata are inextricably bound together when the data is brought into the system.

Question: What is the overhead of using Hive over Hadoop?

Elliston Ball: Hive works by building a series of MapReduce steps from the SQL queries. This is the main source of overhead. Hive was designed to run very fast over large data sets—as soon as it gets started—but it isn't the fastest out of the gate. Something else to note here is the recent progress on Hive performance with Apache Tez. The big difference Tez makes is to optimize the graph of MapReduce jobs created by Hive. In addition, it skips an important source of overhead: Hive used to have to write a lot of intermediate data out to disk between each MapReduce step. Tez can remove a lot of this overhead, if you turn it on.

Long-running Tez sessions and reuse of containers can take several seconds off the startup time, and a wide range of tuning options also help reduce overhead from the Java Virtual Machine [JVM] startup. Most distributions don't turn on all the optimizations, so the most common cause of overhead at the moment is just not turning on the new features.

Question: Can you talk about the relationship between the Lambda Architecture and Hadoop?

Elliston Ball: The Lambda Architecture is a much-debated topic in the big data community at the moment. Even before the ink dried on Nathan Marz' and James Warren's book—*Big Data: Principles and best practices of scalable realtime data systems*—which outlined the architecture, developers were lining up on either side to critique it.

The approach works by separating the batch and speed layer of your application. Hadoop has traditionally been all about batch processing, so for absolutely up-to-the-second analytics, you needed a streaming system on top to do intermediate updates on the earlier batch results. A serving layer would then combine them.

The best use for the Lambda Architecture is when you have a slow-running algorithm running in a batch across a large data set, which can be approximated by a short-running version over a stream of updates. Periodic runs of the batch approach are then used to get a more precise answer. An example is a recommender system that performs complex math over your entire buying history, which can be incrementally improved using a simpler formula on today's transactions and then be brought in with the batch algorithm by an overnight job.

Hadoop provides a perfect platform for implementing the Lambda Architecture. The batch layer can be done in MapReduce, or something like Hive and Pig, while Apache Storm provides a good platform to implement the real-time speed layer, using HBase as the serving layer. With YARN, all this lives on one cluster. Spark—and Spark Streaming—also provides a great framework for the Lambda Architecture, and because of its micro-batch approach, it might allow some reuse of code between the layers. However, if you are reusing a lot of code, you're probably not using the Lambda Architecture as it was intended.

Question: Is there "Hadoop as a Service"?

Babadopulos: Yes, some companies provide Hadoop as a service—for example:

- Amazon EMR deploys a Hadoop cluster inside an Amazon EC2 infrastructure.

- Qubole can deploy Hadoop as a service (and many other tools around the Hadoop ecosystem, such as Hive, Pig, Apache Oozie, and Sqoop) either using Amazon EC2 or Google's Compute Engine.
- Xplenty has an intuitive interface for designing ETL dataflows and also provides Hadoop with a few clicks.
- Microsoft's HDInsight also provides Hadoop as a service using the Microsoft Azure cloud platform.

Question: What can you say about machine learning algorithms?

Richardson: The big idea in ML is the development and use of algorithms that learn from data by constructing models that make predictions or decisions. For example, a common use of ML is to make product or article recommendations based on a user's past behavior. One widely used ML library is Apache Mahout. The scalable algorithms implemented by Apache Mahout were originally built using Hadoop MapReduce. However, more-recent implementations are based on the much faster Apache Spark. Apache Spark also has its own ML library called MLlib. A key part of many applications is a big data pipeline, which extracts data from an online application into the big data system and runs ML algorithms to construct models, which are then used by the online application.

Question: What about tools for governance in big data?

Elliston Ball: Forgoing governance is no longer an option. In the early days of Hadoop, there weren't a lot of options for good data governance and security. However, now governance is very much part of the mainstream data architecture as a key tool.

A lot of things come under the banner of governance—for example, ensuring security, controlling and auditing access to data, managing metadata, and managing data lineage—but essentially it boils down to people and policies. The Hadoop platform has tools that assist with these tasks, but often good governance comes down to a culture of metadata awareness.

On the tools front, each major distribution handles issues of governance in a different way. Cloudera has a proprietary solution (Cloudera Manager and Cloudera Navigator). Hortonworks' tools are 100 percent open source, with Apache Ranger, Apache Knox Gateway, and Apache Falcon. Tools from traditional ETL vendors, such as Informatica and Talend, also work well to provide governance-related services.

Question: Is Hadoop enterprise-ready?

Nardon: Yes! Many companies use it in production right now to run important services. There is a thriving ecosystem around it. Many commercial companies offer services for Hadoop, such as distributions, support, hosting, consulting, and so on. A few of them are Cloudera, Hortonworks, MapR, and Amazon.

Richardson: There are also newer companies, such as Databricks. `</article>`

"The big idea in ML is the development and use of algorithms that learn from data by constructing models that make predictions or decisions," says Chris Richardson.

NARDON:
"Many companies use Hadoop in production right now to run important services. There is a thriving ecosystem around it."

LEARN MORE

- "Big Data and Java: Ask the Experts" panel at JavaOne 2014

PHOTOGRAPH BY BOB ADLER/GETTY IMAGES

JOSH JUNEAU

BIO

Improving the Performance of Java EE Applications

Incorporate performance tuning into your development lifecycle.

Java and Performance

Imagine that your application, after months of development time and lots of testing, is finally ready for production deployment. The application goes live, and for the first few minutes everything seems like a success.

Suddenly you receive a phone call from one of the users telling you that they can't access the application, because it appears to load indefinitely. Moments later you receive another call, and yet another. It seems as though your application doesn't allow concurrent access to enough users. This is a nightmare scenario in which the application development team must make critical performance tuning decisions under the gun, and they must make each of the changes to the production environment directly.

Where do you begin in this situation? What could be wrong? Is a server configuration preventing enough connections for your production environment, or does the application code contain a bottleneck that causes the users to wait?

Many applications are not tuned for performance until after they've been put into production. Unfortunately, many organizations do not deem performance tuning to be an essential part of the development lifecycle, but rather, treat it as a triage step when things go wrong. To make matters worse, Java EE performance tuning can sometimes be much like finding a needle in a haystack. The cause of the performance issue can lie just about anywhere, from the application source code to a server configuration.

In this article, you'll see some of the processes that you can use for tuning your Java EE applications proactively, so you can try to avoid scenarios like this one.

Approaching the Beast: Proactive Tuning

Performance tuning should be made part of the standard development lifecycle. As such, applications should be designed from the ground up with performance in mind.

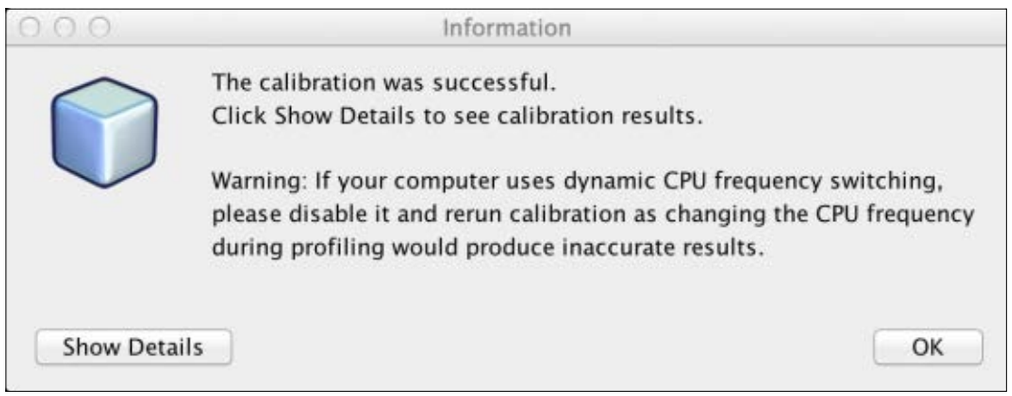


Figure 1

To be forward-thinking about performance means to consider carefully all approaches for the implementation of application solutions, not just the fastest approach or the one that is easiest to implement. Java EE applications can be particularly difficult to develop around performance, because several points of contention in a Java EE environment can add performance burdens to an application.

The top performance issues experienced in Java EE appli-

cation environments tend to be related to configuration issues and environment issues. Oftentimes, the applications themselves are coded fine, but the application server to which they are deployed is not tuned correctly or it is configured inappropriately for the applications or the intended user capacity.

To tune properly for a production employment, perform the following steps in the order listed:

- Application tuning
- Server tuning
- Java runtime tuning
- Server operating system and platform tuning

Next, focus on decreasing the chance of incurring performance issues after a production deployment.

Coding for performance. There are bevies of coding situations that might lead to performance overhead in a Java EE application. Java EE applications are executed concurrently, which can lead to bottlenecks, because users might be competing for resources such as web services or databases. Each remote call can add latency, and processes such as serialization can be CPU-intensive, causing further performance degradation. With these kinds of issues in mind, you need to craft Java EE applications carefully, ensuring that proper resource handling is used.

An application's performance tuning should begin with the source code. Even though the top causes of performance issues in Java EE applications point to the environment, proper coding can still play a key role in an application that performs well.

The following poor-coding practices, among others, can lead to performance issues:

- Over-serialization and deserialization
- Overuse of finalizers
- Too much synchronization
- Not discarding unused variables
- Too many dynamic variables
- Rampant use of System.out .printIn()
- Sessions that are not released when they are no longer needed
- Failing to close resources (for example, database and network connections)

Performing code reviews is imperative for reducing code that inhibits an application's performance. While poorly crafted code can slip past multiple developers, the more eyes that examine it, the better. In addition to code reviews, more than one individual should run performance and load tests against an application, and compare results of current tests to those run previously.

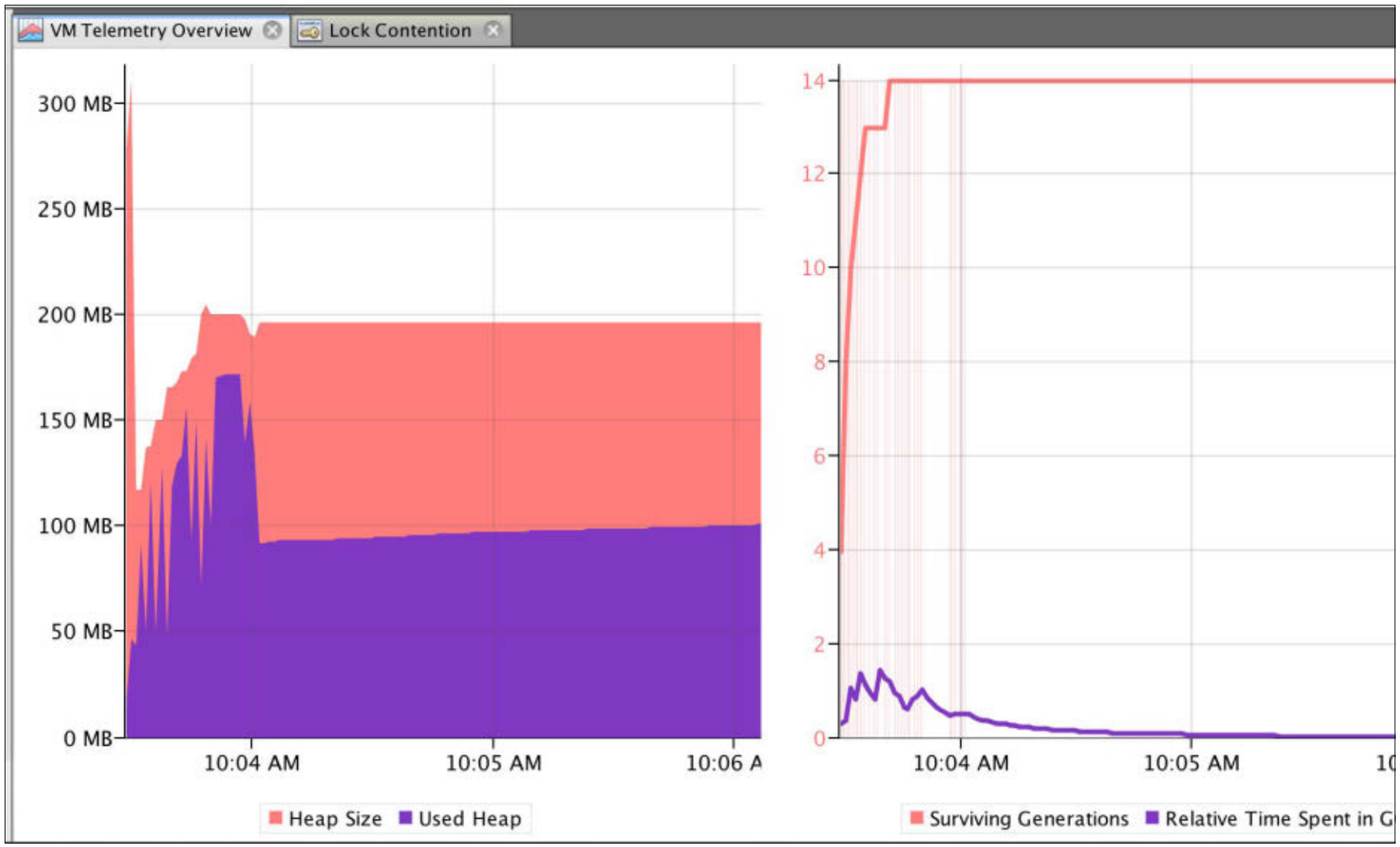


Figure 2

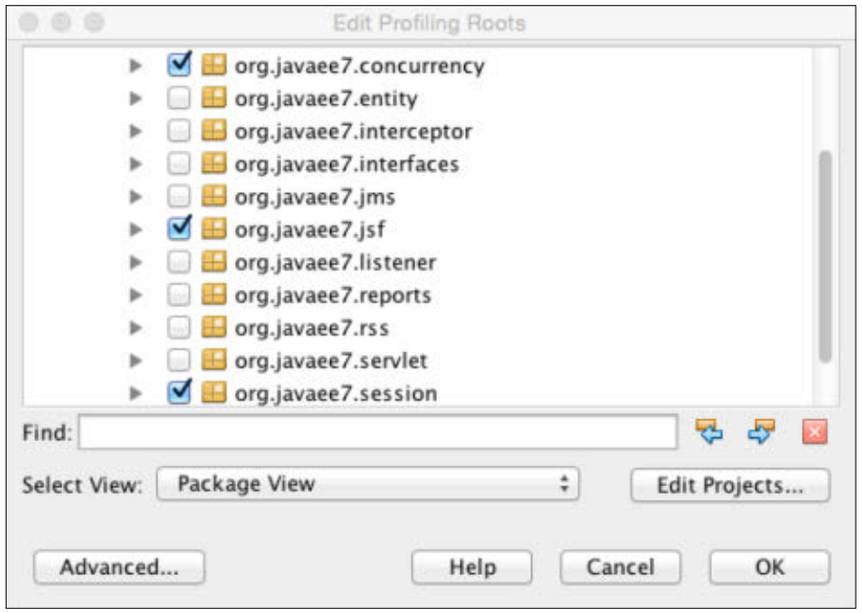


Figure 3

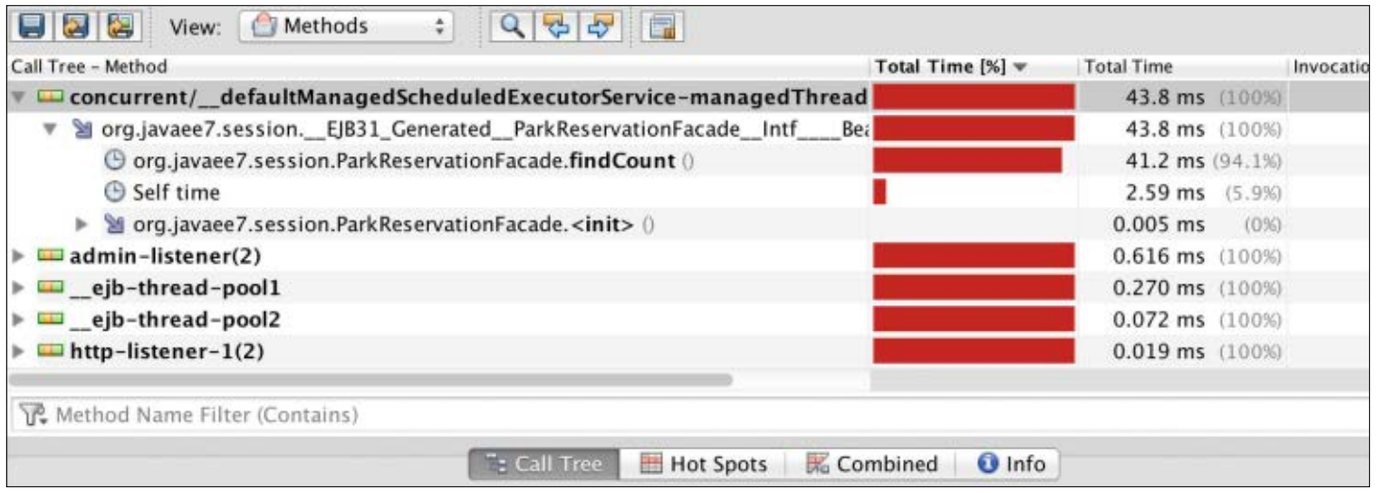


Figure 4

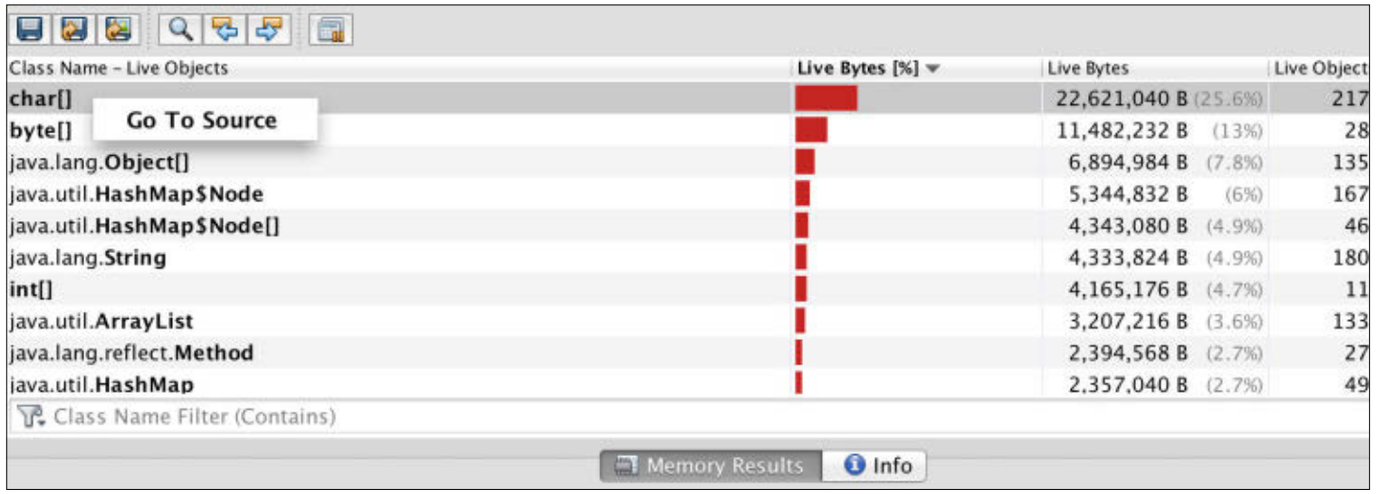


Figure 5

Tuning the environment. Many environmental factors can make a difference in the performance of an application. Learn to love the administrative console or command-line utility for your application server, because you spend lots of time with it. For GlassFish 4.x, the default domain is configured appropriately for testing purposes, but it is likely not appropriate for a production environment without further configuration.

Ensure that deployment settings are configured properly for your applications. If your application server allows autodeployment or dynamic application reloading, be sure to disable those settings in a production environment, because they can have a significant impact on performance. Also, take note of how often logs are written, because frequent logging can cause untimely performance issues.

Consider the application server configuration settings for enterprise components, such as the Enterprise JavaBeans (EJB) container, Java Message Service (JMS), and Java Transaction Service. Always review the default configuration settings, and modify them accordingly to support a production environment. Configure server clustering to provide high availability through load balancing, where appropriate.

Do not treat your database as a black box for your data. Database access can become a point of contention for applications, either when querying a large data set or when performing too many small queries. Whatever the case, have a contact number for your DBA, who might be able to create an index on a database table, or perhaps even incorporate application logic into the database where that makes sense.

Planning for Capacity and Preparing an Application

Many tools can help you prepare an application for production release. Profiling tools can play a key role in an application's development lifecycle. Such tools can be used to forecast how an application will

perform after it's released into production under a normal or heavy load of users.

Two such tools are the NetBeans profiler (a module of the NetBeans IDE) and Apache JMeter. Both of these tools can provide useful information about the performance of an application before it deploys to production, and both are open source.

The NetBeans profiler makes it easy to learn important runtime information about your application by providing the ability to monitor thread states, CPU performance, and memory usage. The Apache JMeter desktop application simulates a heavy load on an application before it goes into production.

To try out these useful tools, experiment by profiling a simple Java EE application named AcmeWorld. You can download the AcmeWorld application from GitHub, and then use it to follow along as you read the rest of this article.

Using the NetBeans profiler. The NetBeans profiler is built into NetBeans IDE 8, so you can use it for profiling your applications out of the box. To perform an accurate analysis of your applications, the profiler must first be calibrated to obtain data about the targeted Java platform and environment. The calibration needs to be performed

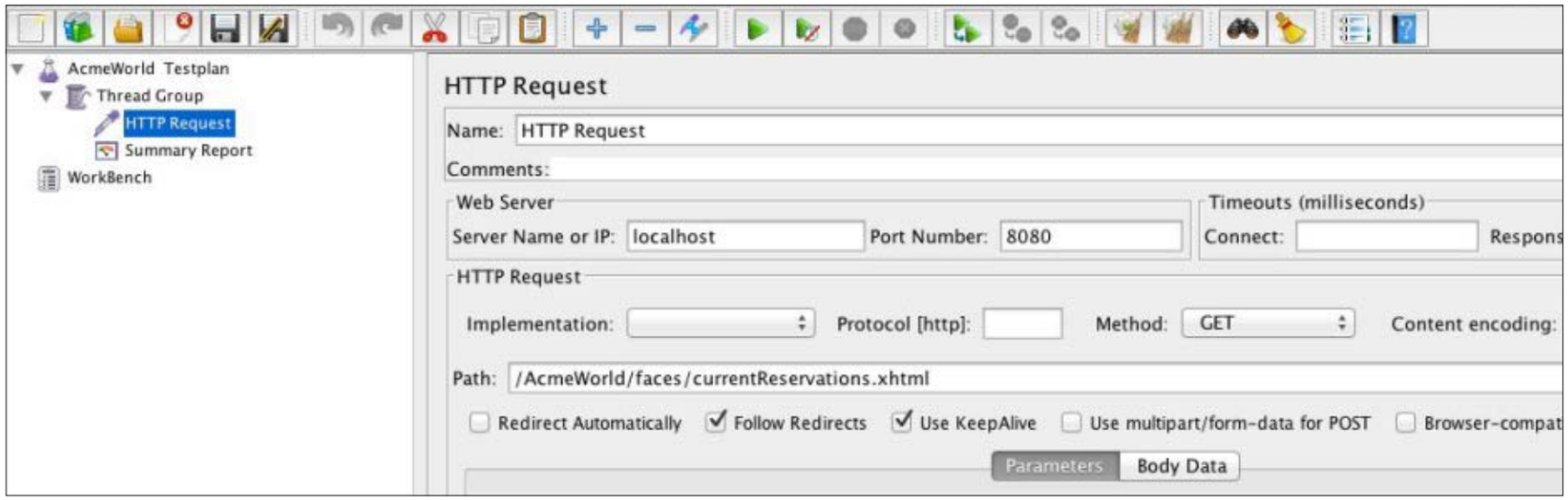


Figure 6

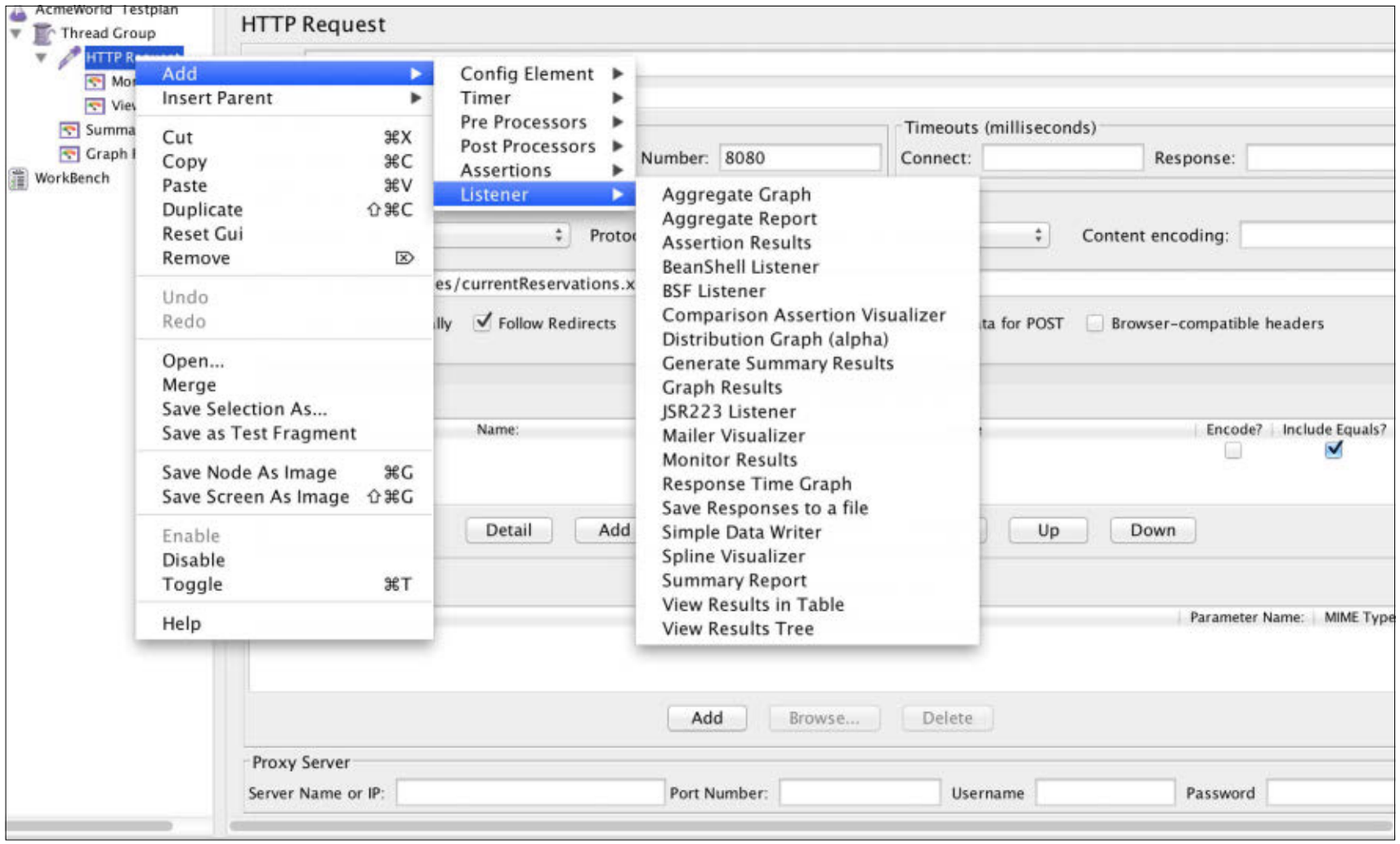


Figure 7

only once, but if you make any significant changes to the environment, recalibrating is a good idea.

To calibrate, choose **Advanced Commands -> Manage Calibration Data** from the Profile menu. Next, choose the targeted Java platform, and click the **Calibrate** button. After the calibration is complete, a success message is displayed (see **Figure 1**).

KNOW YOUR APP

Before production deployment, it is imperative that developers have an accurate portrayal of how an application performs for its users.

By using the profiler, you can perform a myriad of profiling tasks against a NetBeans project. This article focuses on the default profiling configuration, but other options are available. The three main options are Monitor, CPU, and Memory. Each of those contains its own suboptions for performing quick or customized profiling. Read the "Introduction to Profiling Java Applications in NetBeans IDE" for more information.

To monitor the AcmeWorld application, right-click the project in the **Projects** menu, and then choose **Profile**, which opens the profiler's main window.

Then, choose **Monitor**. A few options (for thread monitoring and lock contention monitoring) are disabled by default. Accept the defaults and then click **Run**. Doing so deploys and starts the application, and opens both a Profiler window and a VM Telemetry Overview window (see **Figure 2**) within the NetBeans IDE.

The Profiler window provides the ability to take snapshots, view dump heaps, and more. Take the opportunity to hover over the charts within the VM Telemetry Overview window to see detailed information about the heap or garbage collector at the specified time selection.

After you are comfortable with these options, run the profiler again by right-clicking the NetBeans project and choosing **Profile**. If the profiler is still active, you are prompted to end the current session and begin a new one; otherwise, you can stop the profiler manually by clicking **Stop** within the Profiler window.

This time, choose **CPU**, and then select the **Advanced (instrumented)** option. The CPU profiler allows you to profile the performance of all application classes or only selected classes. That said, choosing the default **Quick (sampled)** option for the CPU profiler can sometimes incur too much overhead and produce output that is not meaningful without performing extended analysis.

Click the **customize** link that appears near the end of the Advanced (instrumented) option. This link opens up the Edit Profiling Roots dialog box, which enables you to choose which servlets, listeners, or application packages to profile. By default, the Select View option is set to **Web Application View**, allowing you to select servlets and listeners. Change the Select View option to **Package View**, and then expand the tree menu to select only the org.javaee7.concurrency, org.javaee7.jsf, and org.javaee7.session packages, as shown in **Figure 3**.

Within the Edit Profiling Roots dialog box, you can also select separate classes, down to the constructors and methods, to pinpoint your performance measurements.

Click **OK** to close the dialog box, and then ensure that the Filter option is set to **Profile only project classes**. Because enterprise applications typically invoke Java classes

Figure 8

that reside in the target server, it is a good idea to limit the profiling to those classes that are internal to the application. Lastly, click **Run**.

When the CPU profiler is running, the Profiler window and VM Telemetry Overview window become active. The CPU profiler enables you to take snapshots, which display the runtime of active classes and methods (see **Figure 4**). These snapshots can be compared against others to determine which methods are overworked or cause contention.

The snapshot editor provides the option to save the snapshots; view by package, class, or method; and filter by method name. You can also compare two snapshots, and when you're attempting to pinpoint a performance issue, this profiler option can be handy. There are also options to view by the "call tree," by "hot spots," or by a combination of those.

The NetBeans profiler also provides the option to profile application memory. Again, this feature allows you to select between quick and advanced profiling.

For our demo, begin by clicking the red stop icon in the Profiler menu to stop the current profiler task, if you haven't already. Next, right-click the project and choose **Profile**, and then select **Memory**. Next, select the **Quick (sampled)**

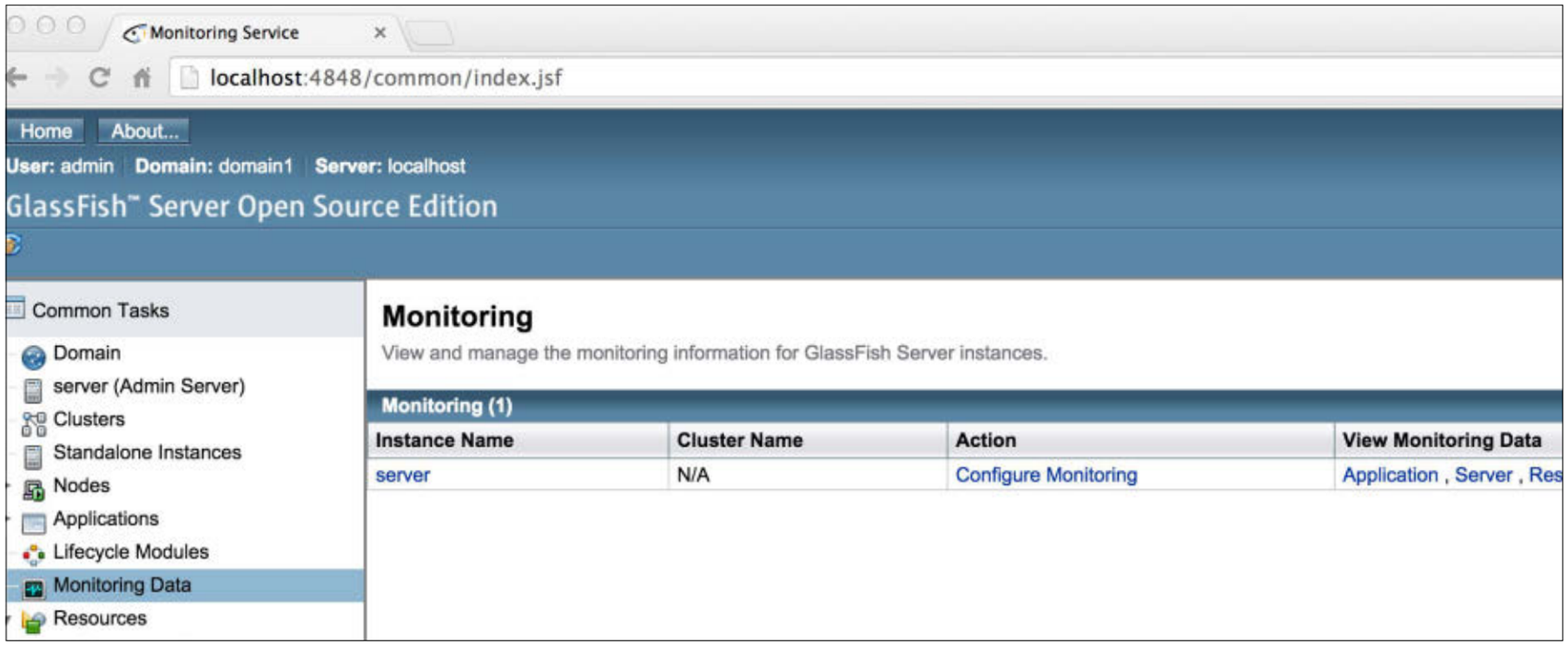
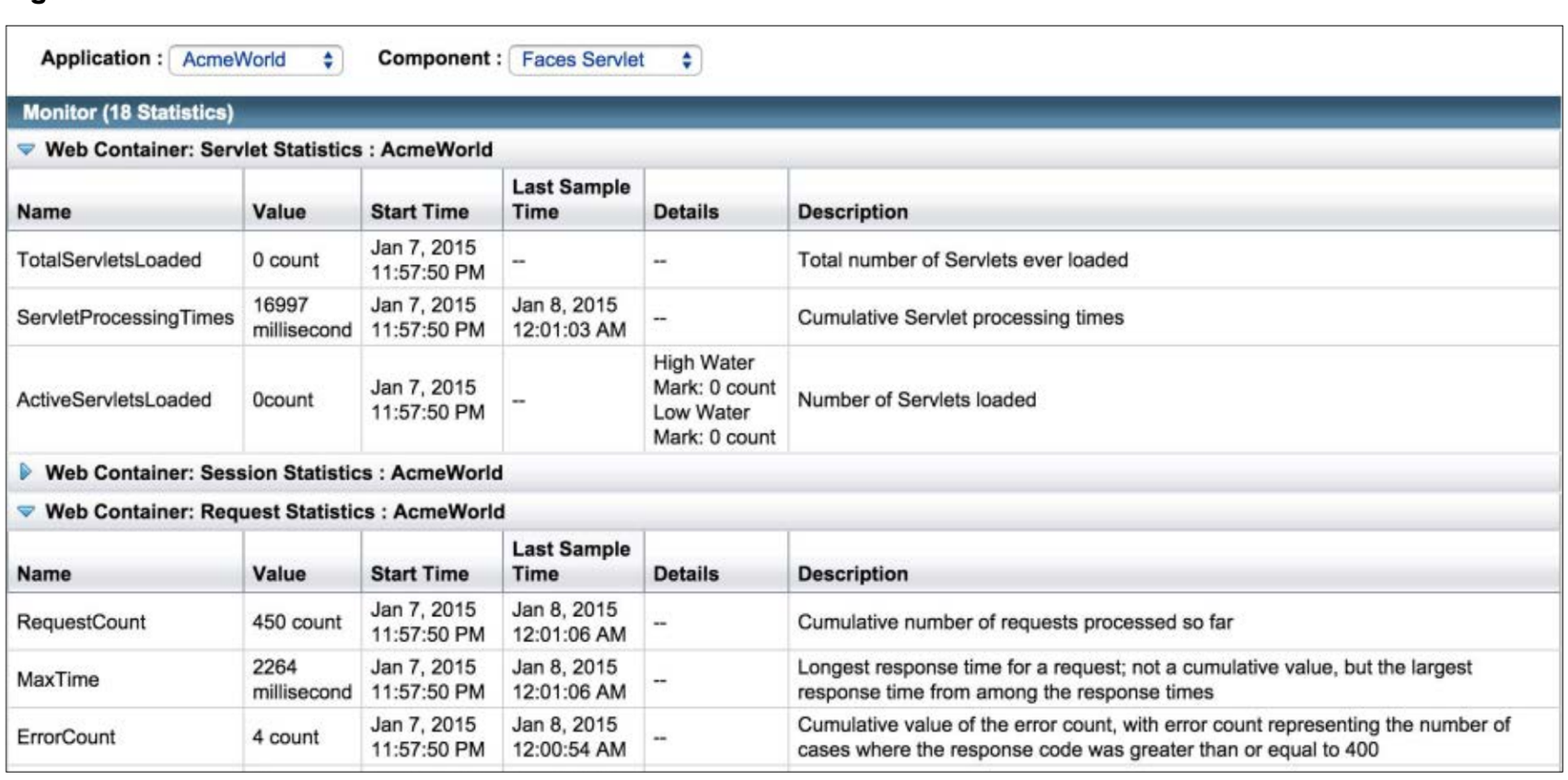


Figure 9



Application : AcmeWorld Component : Faces Servlet

Monitor (18 Statistics)

Web Container: Servlet Statistics : AcmeWorld

Name	Value	Start Time	Last Sample Time	Details	Description
TotalServletsLoaded	0 count	Jan 7, 2015 11:57:50 PM	--	--	Total number of Servlets ever loaded
ServletProcessingTimes	16997 millisecond	Jan 7, 2015 11:57:50 PM	Jan 8, 2015 12:01:03 AM	--	Cumulative Servlet processing times
ActiveServletsLoaded	0count	Jan 7, 2015 11:57:50 PM	--	High Water Mark: 0 count Low Water Mark: 0 count	Number of Servlets loaded

Web Container: Session Statistics : AcmeWorld

Web Container: Request Statistics : AcmeWorld

Name	Value	Start Time	Last Sample Time	Details	Description
RequestCount	450 count	Jan 7, 2015 11:57:50 PM	Jan 8, 2015 12:01:06 AM	--	Cumulative number of requests processed so far
MaxTime	2264 millisecond	Jan 7, 2015 11:57:50 PM	Jan 8, 2015 12:01:06 AM	--	Longest response time for a request; not a cumulative value, but the largest response time from among the response times
ErrorCount	4 count	Jan 7, 2015 11:57:50 PM	Jan 8, 2015 12:00:54 AM	--	Cumulative value of the error count, with error count representing the number of cases where the response code was greater than or equal to 400

Figure 10

option and then choose **Run**. The Profiler window opens again, providing the ability to view telemetry information and take snapshots, this time of application memory.

In the snapshot view, the live class objects are sorted in the Live Bytes [%] column in descending order, making it easy to see which classes use the most memory. Right-clicking a class object provides the option to go to the source code (see **Figure 5**).

Now that you understand the profiling options, open a browser and navigate to the AcmeWorld application. Click the first example in the application, which is Current Reservation List. This invokes the findAll() method of the ParkReservationFacade class (see **Listing 1**).

After the page loads, go back into NetBeans and click the **Live Results** option within the Profiler window. This opens a live results editor pane, which displays the active "hot spot" methods. You should see that the AbstractFacade .findAll() method is on top, because the ParkReservationFacade class extends AbstractFacade.

The NetBeans profiler can be a big help for locating performance problems within an enterprise application before production deployment. This article just scratches the surface of what you can do with it.

Using Apache JMeter. One of the most problematic performance concerns for a Java EE application environment is user capacity. How many users can access the application at the same time? This question is impossible to answer in the development environment, because so many variables are at play. Even if the development environment is configured exactly the same as the production environment, some inconsistencies between development and production almost always occur. User capacity testing is often a game of chance. However, the Apache JMeter tool can help you hone your capacity testing.

APP DEVELOPMENT

Java EE applications can be particularly difficult to develop around performance, because several points of contention in a Java EE environment can add performance burdens to an application.

The JMeter tool is easy to use, and it contains useful utilities for measuring performance and load testing. For more information about installing and using JMeter, see the *Apache JMeter User's Manual*.

LISTING 1

```
public List<ParkReservation> currentReservations() {
  return ejbFacade.findAll();
}
```

Download all listings in this issue as text

To begin, open JMeter and create a new test plan. Name the test plan AcmeWorld Testplan.

Next, right-click the test plan in the left menu, and select **Add -> Threads (Users)**. When the Thread Group panel is displayed, enter a value of 500 into the **Number of Threads** field to represent 500 concurrent users.

Next, add an HTTP Request to the test by right-clicking the newly created thread group and selecting **New -> Sampler -> HTTP Request** (see **Figure 6**). In the HTTP Request panel, fill in the server name and the port number, and provide a path to test. In this case, use /AcmeWorld/faces/currentReservations.xhtml for the path.

The JMeter test is now ready to run. Start the application within your IDE. Then click the **Start** button within JMeter to run the 500-user test on the currentReservations view.

You can use JMeter to read the results of the test by taking a look at different listeners. Right-click **HTTP Request** and select **Add -> Listener** to see a list of listeners that can be used to view various performance measures (see **Figure 7**).

The most useful graphs are the Response Time Graph (shown in **Figure 8**), the View Results in Table, and the Summary Report.

Putting it all together. The tools and techniques discussed so far can be used together to provide a variety of simulated performance and loading tests and help you

- Perform code reviews to ensure that proper coding techniques are used, and to help reduce costly mistakes.
- Take an abundance of snapshots using the NetBeans profiler, and save JMeter reports for comparison under different environmental situations.
- Perform capacity load testing with more threads than the number of users you expect on a normal day.
- Use JMeter capacity testing and take NetBeans profiler snapshots in unison to measure the performance of your application processes under a heavy load.

GlassFish Server Open Source Edition 4.1 REST Interface

- activatedsessionstotal
  - unit : count
  - lastsampletime : -1
  - name : ActivatedSessionsTotal
  - count : 0
  - description : Total number of sessions ever activated
  - starttime : 1420726822075
- activeservletsloadedcount
  - unit : count
  - current : 4
  - lastsampletime : 1420726823914
  - lowwatermark : 0
  - name : ActiveServletsLoaded
  - description : Number of Servlets loaded
  - highwatermark : 4
  - starttime : 1420726822074
- activesessionscurrent
  - unit : count
  - current : 0
  - lastsampletime : -1
  - lowwatermark : 0
  - name : ActiveSessions
  - description : Number of active sessions
  - highwatermark : 0
  - starttime : 1420726822075
- errorcount
  - unit : count
  - lastsampletime : -1
  - name : ErrorCount

Figure 11

While performing a true production test in a development environment is nearly impossible, the combination of these techniques can help you refine your application as well as your production environment so that less time is required for reactive tuning.

Taming the Beast: Reactive Tuning

No matter how well a system has been designed for performance, something can happen to slow it down. More often than not, performance tuning is done in a reactive manner, so having a plan for performing reactive tuning is imperative. When tuning a Java EE application, you must peel back the onion of the stack, analyzing and investigating each layer as the possible culprit for application performance degradation.

In the case of GlassFish, the server contains built-in monitoring APIs, so you can monitor the modules of your Java EE application server. Monitoring can be configured with the asadmin command-line utility or through the administrative console (shown in **Figure 9**).

Each of the modules can be configured to OFF (the default), LOW, or HIGH. After the modules are configured, an application can be monitored at the console (see **Figure 10**) or with the asadmin utility. The asadmin utility provides the added benefit of letting you pipe output into a file or a database to be analyzed at a later date against other saved monitoring data.

Applications and server modules can also be monitored on the web at the default URL of http://<localhost>:4848/monitoring/domain/server, as seen in **Figure 11**. Because the GlassFish monitoring API is REST-based, applications can be built to interrogate the data and display in useful formats. One such utility is LightFish, which Java EE expert Adam Bien wrote. If you use another Java EE application server, search for the monitoring tool that best suits the environment.

Careful monitoring of the application server using monitoring resources, such as those offered by GlassFish, is key to pinpointing performance issues in Java EE applications after deployment. You must also monitor server resources, such as memory and CPU usage, to ensure that the server is not overloaded. Database monitoring might also be useful in a reactive tuning toolset.

Conclusion

To deem an application successful, it should include all of the intended functionality and perform well. Users that experience poor performance often consider the application unworkable.

Incorporating proactive performance tuning measures into your development lifecycle can alleviate many hours of reactive tuning on the tail end. Before production deployment, it is imperative that developers have an accurate portrayal of how an application performs for its users. The NetBeans profiler and Apache JMeter are two profiling tools that can help you forecast how an application will perform under a normal or heavy load of users after it is released to production. `</article>`

MORE ON TOPIC:

Java and Performance

LEARN MORE

- NetBeans profiler
- Apache JMeter
- LightFish

Writing JavaFX Applications for Mobile Devices

With recent JavaFX ports, you can now create Java client applications for mobile devices.

JOHAN VOS

BIO

Java developers can use JavaFX to create great client applications that have a compelling user interface. By leveraging the power of the Java platform, developers can create client applications that address today's requirements for performance, as well as look and feel. This article explores the state of JavaFX on mobile devices, with a specific focus on the Android platform.

In the past, Java client applications focused on desktop and laptop environments. That made sense, because most of the transactions started from those environments. Also, the mobile landscape was so fragmented that creating applications capable of running on most mobile devices was difficult and expensive.

Today, the situation has changed. End users often use mobile devices (smartphones and tablets) instead of laptop and desktop systems, and an increasing number of business transactions originate from those mobile devices. On mobile devices, consumers clearly prefer native applications over browser-based applications.

The mobile landscape has changed significantly over the last few years. Today, more than 90 percent of mobile devices use either the Google Android operating system or the Apple iOS operating system. Both platforms provide a convenient way for developers to distribute their applications through an "app store." The Apple App Store accepts applications created for iOS; the Google Play Store is where Android applications should be uploaded.

The combination of a new mobile landscape and JavaFX provides a huge opportunity to Java developers. With JavaFX on mobile devices, Java developers can now use their favorite language to create powerful, modern-looking applications that work on desktops, laptops, mobile devices, and embedded systems.

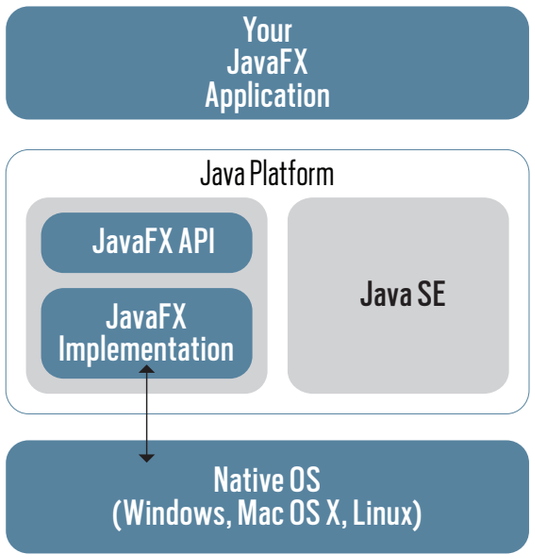


Figure 1

Porting JavaFX to Mobile Devices

One of the key benefits of Java is the "write once, run anywhere" paradigm. JavaFX is no exception to this rule. If you write a JavaFX application, the runtime environments handle the platform-specific issues so you can focus on application-specific needs.

On desktop systems with either Microsoft Windows, Linux, or Mac OS X, the situation is as shown in **Figure 1.**

Oracle provides JavaFX implementations for Windows, Mac OS X, and Linux.

The JavaFX porting efforts for mobile devices and embedded ARM systems are coordinated by JavaFXPorts, an initiative from the Java community. The codebase for

PHOTOGRAPH BY TON HENDRIKS

JavaFXPorts is based on the code for OpenJFX, which is the official repository for the JavaFX code available at https://wiki.openjdk.java.net/display/OpenJFX/Main. One goal of JavaFXPorts is to send all the changes that are made to build JavaFX on mobile devices back to OpenJFX. This ensures convergence rather than divergence among all implementations of JavaFX.

The Android platform comes with its own virtual machine, named the Dalvik Virtual Machine (or Dalvik for short). While Dalvik is by no means a fully functional Java SE 8 implementation, it has many similarities to a Java SE 7 runtime.

As part of the JavaFX Android port, the platform-specific JavaFX implementation leverages Dalvik. Furthermore, the native parts of the JavaFX runtime are compiled for Android, as shown in **Figure 2**.

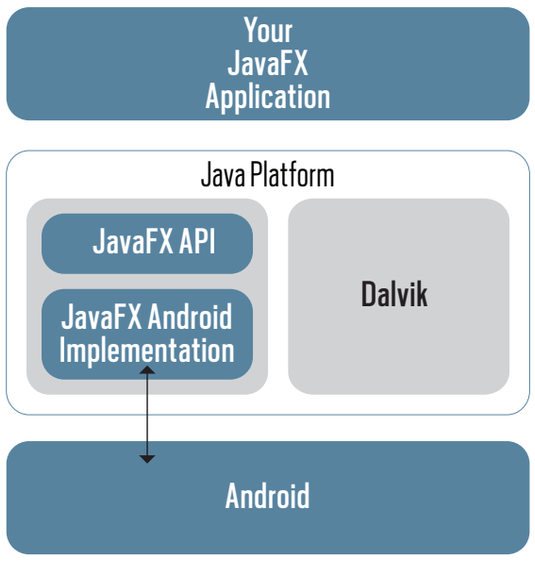


Figure 2

It is important to note that although Java SE 8 is not available on Android, the JavaFX 8 APIs are available. Also, the internal implementation of JavaFX 8 on Android and iOS does not use Java SE 8–specific APIs.

Moreover, it is possible to use lambda expressions, because a great tool called RetroLambda can convert them (at the bytecode level). RetroLambda inspects class files, and it replaces all invokedynamic calls with invoke calls that are supported in Java SE 7.

On iOS, the situation is a bit more difficult, because no Java runtime is available on the iOS platform. Worse, Apple has restricted the just-in-time (JIT) options that are used by typical Java Virtual Machines (JVMs) and that compile bytecode to native code at runtime.

However, RoboVM provides a smart, ahead-of-time compiler that translates Java code to native iOS code, eliminating the need for a JIT processor running on the devices. RoboVM offers more or less the same base classes as the Java classes in the Dalvik Virtual Machine, which means that at this moment, it offers a significant subset of the Java SE 7 platform. As a consequence, JavaFX 8 applications that use Java SE 7 APIs can be created on desktop environments and subsequently converted into Android or iOS applications.

In **Figure 1** and **Figure 2**, it is clear that your JavaFX application can be executed on desktops and laptops, as well as on Android or iOS devices. One restriction: The Java runtime environments on Android and iOS do not support Java 8–specific APIs at the moment. Therefore, avoid using them in code that you want to run on mobile platforms.

Creating a JavaFX Application on Android

Writing Java code is one thing; deploying that code on mobile devices is another thing. Android applications are created by performing operations that convert Java classes and resources into an Android package (a .apk file). This Android package can be uploaded to the Google Play Store, where it can be downloaded by all Android users. While doing development, you can also attach an Android device to your development system with a USB cable and transfer the Android package to your device.

Next, see how you can create an Android package based on your JavaFX code. Note that this build process is under continuous improvement. See the JavaFXPorts website for the latest (and simplest) build instructions.

Prerequisites. Before you can build Android packages, you need to have the Android SDK. Install this SDK in a directory—for example, in /opt/android-sdk.

You also need to install the additional packages that provide the latest SDK tools, platform tools, and build tools, as explained here.

In addition, you need to install the latest javafx-for-dalvik runtime (also called the dalvik-sdk), which can be downloaded from this Bitbucket page. Unpack the zip file in a directory—for example, in /opt/dalvik-sdk.

The Dalvik SDK contains a samples directory that includes a sample JavaFX HelloWorld application, along with the required build scripts to create an Android package. You might also want to download the samples that are part of the JavaFXPorts project, which show specific scenarios. These samples are available at this Bitbucket page.

The build system. Now that you have all the required tools, you can create a JavaFX application. This article uses the Android-Gradle plugin, so I recommend using the Gradle build system. However, the Android-Gradle plugin cannot be combined with the Java-Gradle plugin in a single project. Because developers often want to run their applications on a desktop or a

laptop first, it's a good idea to use a multiproject Gradle setup.

The HelloWorld sample in the samples directory of the Dalvik SDK shows how this is structured. We have a root project with three subprojects: core, desktop, and android. The core subproject contains all code and shared resources (for example, .fxml files). It uses the Java plugin to build the code.

The desktop subproject contains no code or resources, but only a simple build.gradle file. The contents are shown in **Listing 1**. This build file specifies that we want to create an application that depends on the code in the core subproject, and that we want the HelloAndroid class to be the class that starts the application.

While this desktop subproject is extremely simple, it is often good practice to have this subproject and to use it during development. The development lifecycle on Android is slightly longer than on a desktop, and during development, testing the application on a desktop or a laptop is often easier.

The Android Gradle project. The third subproject, which is named android, is a typical Android Gradle project. Gradle is the preferred build system for Android applications, and the Android-Gradle plugin already contains most of the tasks and build logic required for creating native Android applications.

The android directory in the samples/HelloWorld demo has a typical setup for an Android app:

- A build.gradle file for building the packages.
- An AndroidManifest.xml file, which contains application-specific information.
- An assets directory with some property files that will be compiled into the package. You don't have to change those, but they should be present.

Note that a typical Android application has more files and directories, which allows for more customization of the packages. However, in the HelloWorld demo, everything is kept to a minimum. One of the benefits of the Gradle system is that it relies on the "convention over configuration" principle. The default behavior is sufficient in most cases, but there are tons of configuration options so you can configure the application the way you want it.

The AndroidManifest.xml file in the src/main directory can be used to supply information about the application. At a minimum, the information shown in **Listing 2** must be specified in this manifest file.

The text in italics should be altered for each application. Note that the android:name part of the activity tag can be chosen freely,

LISTING 1 LISTING 2 LISTING 3 LISTING 4

```
apply plugin: 'application'

mainClassName = 'org.javafxports.helloworld.HelloAndroid'
dependencies {
  compile project(':core')
}
```

Download all listings in this issue as text

and it is not necessarily the same as the main.class attribute specified in the meta-data section of the activity tag. The latter has to be the name of the JavaFX application class.

In the build.gradle file, we have to instruct the Android tools to take into account the JavaFX runtime libraries (both the Java classes and the native libraries). The build.gradle file contains a dependency section, in which we have to declare that the application depends on the code compiled in the core subproject. We also have to add the JAR files provided by the Dalvik SDK (see **Listing 3**).

In the android section of the build.gradle file, we have to specify the location of native libraries and properties (see **Listing 4**). The dalvikSdkLib that is used at several locations in the build.gradle file should point to the rt/lib directory in the Dalvik SDK—for example:

```
def dalvikSdkLib =
/opt/dalvik-sdk/rt/lib
```

With the combination of these three projects, you can now run the JavaFX application on a desktop or laptop system or on a mobile device. Executing the command gradle tasks shows an overview of available tasks. For example, gradle desktop:run executes the application on your development environment.

To create an Android debug application and send it to an Android device that is connected to your development environment, execute gradle android:installDebug. To create a release version for the Android application that you want to upload

to the Play Store, you have to enter gradle android:assembleRelease.

JavaFX-on-Android Features

The JavaFX runtime on Android is based on the code in the OpenJFX source repository, and it supports most of the features of the corresponding JavaFX release for desktop platforms. For example, dalvik-sdk-8u40 releases are based on the same codebase as JavaFX 8u40 for desktop platforms. At this moment, the media framework is not supported. More features of JavaFX have been added to the Android port. For example:

- JavaFX on Android now supports JavaFX 3D.
- New components that are introduced in JavaFX 8 have been added, including the convenient DatePicker control.
- You can use FXML to build your Scene.

Although using only standardized Java code is best, sometimes you can benefit from using Android-specific code. Apart from the standard Java classes, Android APIs provide a rich framework that allows developers to integrate device-specific functionality such as the location providers (including GPS), a near field communication (NFC) reader, and so on. Including Android APIs breaks portability, so use a modular approach in your application, where the Android-specific parts are implemented in the android subproject.

To access the Android APIs, a hook into the Android system (which is offered in the JavaFXPorts libraries) is required. Your application is started automatically by an instance of javafxports.android.FXActivity, a regular Android Activity that extends the Android Context class. You can access this Activity, and consequently all functionality that can be reached through an Activity, by calling the following:

```
FXActivity activity =
FXActivity.getInstance();
```

As an example, you can get the Android NFC adapter by using the code shown in **Listing 5**.

We've seen how you can use Android code in a JavaFX application. In some cases, you want to do the opposite: You have an Android application, and you want to use JavaFX components inside your Android application. While this adds more requirements for the integration of user interfaces, it is technically possible.

The samples repository, available here, contains the Kokos project, which is an example of doing this. This example was created with the Android Studio IDE, and it starts with a regular Android application. By using the Android Studio interface, you created a new <code>fragment<code> as a subclass of javafxports.android.FXFragment.

The FXFragment class is a subclass of the Android Fragment class, and it allows you to run a JavaFX application inside its view. In the Kokos example, the javafxports.org.kokos.MyFxApp class is a JavaFX application that is assigned to the created fragment, as shown in **Listing 6**.

Note that the possibility of using JavaFX application code inside Android fragments is still under early development. Start with a JavaFX application whenever possible, rather than starting with an Android application and adding a JavaFX application to it.

Conclusion

JavaFX became mature right on time for being part of the mobile revolution. Native applications on mobile devices are becoming an interesting area of development.

The porting efforts for providing JavaFX on mobile devices are a Java community initiative, and they still require lots of work. To get involved in the porting efforts, send an e-mail to info@javafxports.org.

The port of JavaFX to Android is a continuous effort. The build process has been simplified since this article was written, and you can now build iOS applications leveraging the same code and tools. Read more about this effort here. We wrote a Gradle plugin that runs your existing JavaFX code on desktop, Android, and iOS devices. Using the new javafx-mobile plugin is very simple; it's described here. **</article>**

LISTING 5 **LISTING 6**

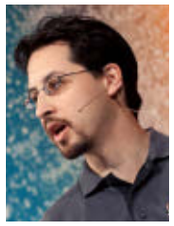
```
android.content.Context ctx = FXActivity.getInstance();
android.nfc.NfcAdapter nfcAdapter =
NfcAdapter.getDefaultAdapter(ctx);
```

Download all listings in this issue as text

LEARN MORE

- JavaFXPorts website
- JavaFXPorts sample applications

//fix this /



In the November/December 2014 issue, Abhishek Gupta showed us a problem using the JAX-RS API.

The correct answer is #3. Because the ChildResource class overrides the @Consumes annotation on the greet method, one would have to include all the other metadata annotations that are there on the greet method in the ParentResource class. In this case, it happens to be the @POST annotation.

This issue's challenge comes from Stephen Chin, Oracle Java Technology Ambassador and JavaOne content chair.

1 THE PROBLEM

The new Date and Time API introduced in Java 8 is a dramatic improvement over the original functionality in the JDK. JSR 310 not only introduces a new API that is simpler and less error-prone, but it also introduces new functionality to better handle time zones and unit testing of complicated date logic.

One of the new capabilities of the Java 8 Date and Time API is the concept of TemporalAdjusters, which can be used to externalize date adjustments. Built-in TemporalAdjusters handle many common operations such as finding the last day of the month or year. But what if you want to do something specific to your application, such as finding the next occurrence of Friday the 13th, an unlucky day in Western superstition?

2 THE CODE

Here is a simple code snippet that defines a TemporalAdjuster lambda expression to do exactly this. Given an arbitrary input date (for example, LocalDate or LocalDateTime), this code should return the next occurrence of a Friday on the 13th of the month.

```
TemporalAdjuster friday13Adjuster = temporal -> {
    temporal = temporal.with(ChronoField.DAY_OF_MONTH, 13);
    while (temporal.get(ChronoField.DAY_OF_WEEK) !=
           DayOfWeek.FRIDAY.getValue()) {
        temporal = temporal.plus(1, ChronoUnit.MONTHS);
    }
    return temporal;
};

// Examples usage:
LocalDate today = LocalDate.now();
LocalDate nextFriday13 = today.adjustInto(friday13Adjuster);
```

3 WHAT'S THE FIX?

Something is wrong with the above TemporalAdjuster. What could it be?

1) Variations in month. Does the code need to account for differences in month length when incrementing the temporal?
2) A time zone issue. Does the code need to account for input dates with time zone components?
3) An error in the formula. Maybe the calculation of Friday the 13th is fundamentally flawed?
4) Something else entirely?

GOT THE ANSWER?

Look for the answer in the next issue. Or submit your own code challenge!

ART BY I-HUA CHEN