Java™ magazine

By and for the Java community

LOCAL-VARIABLE TYPE INFERENCE 60 | SCALA 47 | BLOCKCHAIN 36

JANUARY/FEBRUARY 2017

TOOLS

17
POLYGLOT: MOVING MAVEN PAST XML

22
INSIDE THE ARCHITECTURE OF BUILD TOOLS

29
BUILD YOUR OWN JVM DEBUGGING TOOLS

ORACLE.COM/JAVAMAGAZINE

ORACLE®

//table of contents /

17

POLYGLOT FOR MAVEN: MOVING MAVEN INTO SCRIPTING

By Jason van Zyl and Manfred Moser

A new tool from the developer of Maven enables POM files to be written in Ruby, YAML, Groovy, and other languages.

COVER ART BY I-HUA CHEN

22

THE DESIGN AND CONSTRUCTION OF MODERN BUILD TOOLS

By Cédric Beust

A look inside a modern JVM build tool—its architecture and implementation

29

CREATING YOUR OWN DEBUGGING TOOLS

By Andrei Pangin

JDK serviceability technologies allow you into the JVM to solve difficult debugging problems.

03

From the Editor

The Polyglot Future: As the nature of programming changes, should we add regular coverage of JavaScript to the magazine?

05

Letters to the Editor

Comments, questions, suggestions, and kudos

08

Events

Upcoming Java conferences and events. And introducing Oracle Code, a new series of 20 free one-day events on polyglot programming, DevOps, and cloud computing.

13

Java Books

Review of *Core Java, Volume II (10th Edition)*, by Cay S. Horstmann

36

Cryptocurrency

Blockchain: Using Cryptocurrency with Java

By Conor Svensson

Integrating the Ethereum blockchain into Java apps using web3j

47

JVM Languages

Scala: Deeply Functional, Purely Object-Oriented

By Adriaan Moors

A mature, practical, and type-safe language for the JVM

52

Cloud

Java in Containers in the Cloud

By Harshad Oak

Deploy Java apps in Docker containers using Oracle Application Container Cloud Service.

60

Java Proposals of Interest

JEP 286: Local-Variable Type Inference

63

Fix This

By Simon Roberts

Our latest code quiz

21

User Groups

Chicago JUG

68

Contact Us

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

EDITORIAL
Editor in Chief
Andrew Binstock
Managing Editor
Claire Breen
Copy Editors
Karen Perkins, Jim Donahue
Technical Reviewer
Stephen Chin

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

PUBLISHING
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
Sales Director
Tom Cometa
Account Manager
Mark Makinney
Mailing-List Rentals
Contact your sales representative.

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

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

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

MAGAZINE CUSTOMER SERVICE
java@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 email address not be included in this program, contact Customer Service.

Copyright © 2017, Oracle and/or its affiliates. All Rights Reserved. No part of this publication may be reprinted or otherwise reproduced without permission from the editors. *JAVA MAGAZINE* IS PROVIDED ON AN "AS IS" BASIS. ORACLE EXPRESSLY DISCLAIMS ALL WARRANTIES, WHETHER EXPRESS OR IMPLIED. IN NO EVENT SHALL ORACLE BE LIABLE FOR ANY DAMAGES OF ANY KIND ARISING FROM YOUR USE OF OR RELIANCE ON ANY INFORMATION PROVIDED HEREIN. Opinions expressed by authors, editors, and interviewees—even if they are Oracle employees—do not necessarily reflect the views of Oracle. The information is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle. Oracle and Java are registered trademarks of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.

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

O'REILLY®

OSCON
OPEN SOURCE CONVENTION

The brightest minds in tech are coming to Austin.

Learn which open source models are right for all—or even part—of your business at OSCON.

May 8-11, 2017
oscon.com

JAVA MAGAZINE READERS GET
20% OFF WITH CODE **JAVA**

The Polyglot Future

As the nature of programming changes, should we cover Java *and* JavaScript?

It's no secret that almost all commercial programs today depend on more than one language. While Java still stretches from the UI all the way to the back-end work (a true end-to-end proposition), its role is frequently complemented by functionality written in other languages. Sometimes, those parts are written using scripts or JVM languages. At other times, a more common scenario unfolds: a Java core surrounded by a UI written in JavaScript. The multilanguage phenomenon—referred to as *polyglot programming*—often implies the use of more than two languages. But if there is just one other language, that language is overwhelmingly JavaScript.

In many ways, JavaScript has been an ever-emerging partner to Java: starting with its name, which Netscape chose to key off of Java's popularity, thereby causing endless confusion for nontechnical audiences; then the use of both languages in web apps, followed by their joint use in mobile; and of course, then the use of JavaScript as a scripting language for Java, via Nashorn. In addition, tools such as the Google Web Toolkit, which transpiles Java into JavaScript, have furthered the connection between the two languages.

JavaScript is also seeping into back-end computing. This trend is most evident with Node (the former node.js), a JavaScript framework that creates a server-like environment for business logic by providing "an event-driven, non-blocking I/O model that makes it lightweight and efficient." Its emergence is like Ruby on Rails a decade ago: a convenient solution to small-scale projects. Node is likely to expand its presence in that sector due to its reliance on a widely used language.

PHOTOGRAPH BY BOB ADLER/VERBATIM

ORACLE

Level Up at Oracle Code

Step up to modern cloud development. At the Oracle Code roadshow, expert developers lead labs and sessions on PaaS, Java, mobile, and more.

Get on the list for event updates:
go.oracle.com/oraclecoderoadshow

developer.oracle.com

#developersrule

I should note that the Java community has responded to Node with an excellent framework, Vert.x, that provides all the former framework's capabilities and more and is rapidly gaining adherents. One of its appealing aspects is represented by the .x in its name, which is a direct counterpoint to the .js in Node's original name. While Node is a JavaScript framework, the *x* in Vert.x signifies support for many languages. Here again we see the advent of polyglot programming.

Oracle employs tens of thousands of developers and interacts with millions more. The company's considered opinion is that the polyglot trend is real, enduring, and important. Underlying this conviction is the belief that the cloud will hasten the adoption of multiple languages due to the ease of setting up and testing out different stacks and different toolchains in the cloud. With this in mind, the company has asked me whether *Java Magazine* should broaden its mission to cover polyglot programming, primarily via the addition of coverage of JavaScript.

It might seem odd to consider this step for a publication with *Java* in the name. But let's put aside the issue of the name for a moment. Looking at the last two years, we have regularly covered other JVM languages. In this sense, we've already had a regular taste of polyglot coverage in our pages. But the question I've been asked involves a deeper commitment to the polyglot reality, more than just JVM languages: Should we add regular coverage of JavaScript?

As my job is to serve you, I'd like your opinion. Would you like to see the addition of tutorials, perhaps morphing into regular coverage, of JavaScript in the magazine? The guidance I'm specifically looking for is how much JavaScript coverage would help you in your programming work. Feel free to say "absolutely none," or "lots of it," or anything in between. And please include any further thoughts as well as any recommendations you have on the current mix of articles you find in our pages.

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

ORACLE CODE

Register Now

Oracle Code

New One-Day, Free Event | 20 Cities Globally

Explore the Latest Developer Trends:

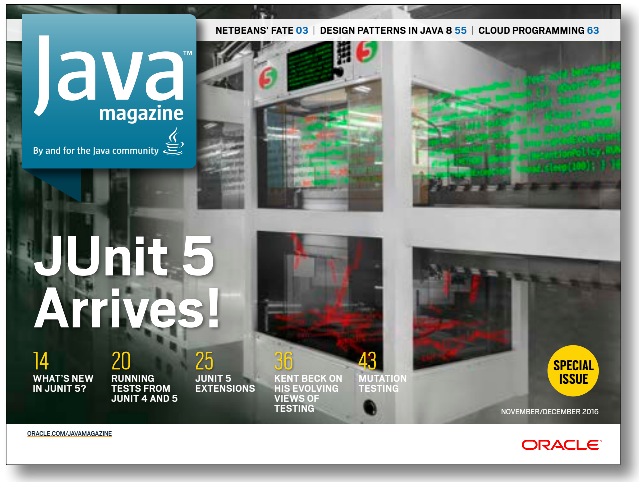
- DevOps, Containers, Microservices & APIs
- MySQL, NoSQL, Oracle & Open Source Databases
- Development Tools & Low Code Platforms
- Open Source Technologies
- Machine Learning, Chatbots & AI

Find an event near you:
developer.oracle.com/code

Live for the Code

ORACLE

//letters to the editor /


NETBEANS' FATE 03 | DESIGN PATTERNS IN JAVA 8 55 | CLOUD PROGRAMMING 63
Java™ magazine
By and for the Java community
JUnit 5 Arrives!
14 WHAT'S NEW IN JUNIT 5?
20 RUNNING TESTS FROM JUNIT 4 AND 5
25 JUNIT 5 EXTENSIONS
36 KENT BECK ON HIS EVOLVING VIEWS OF TESTING
43 MUTATION TESTING
SPECIAL ISSUE
NOVEMBER/DECEMBER 2016
ORACLE.COM/JAVAMAGAZINE
ORACLE

NOVEMBER/DECEMBER 2016

NetBeans' Evolution

Thanks for your editorial re: NetBeans in the November/December 2016 issue of *Java Magazine*, "NetBeans Gets a New Life—or Does It?" It continues to be odd to see the process of moving a project to the Apache Software Foundation (ASF) being described as anything other than a good thing.

—Geertjan Wielenga
Principal Project Manager, Oracle

Andrew Binstock responds: "As a prudent person, I prefer to take a wait-and-see attitude given that, as I wrote in the editorial you refer to, there is no history of a project like NetBeans moving to the ASF. So there is no reason to view it with anything but the dispassion I expressed—unless you're personally attached to the product. However, I believe the real reason for concern, if there is one, does not come from moving to the ASF, but whether anyone will step up and fund NetBeans' continued development after Oracle's commitment to it ceases. If no sponsors emerge or if a sponsor underwrites only the costs of maintenance but not development, then I believe the move to the ASF will mark what will likely be the final phase of NetBeans' existence.

"In the event that a sponsor or sponsors do make a public commitment to continue developing NetBeans after Oracle's support ends, we will share this good news with our readers who still rely on the product."

JUnit 5's Status

Thank you for your multiple articles on JUnit in the November/December 2016 issue. The final delivery of version 5 is now expected to be in the third quarter of 2017. We intend to introduce as few breaking changes as possible, especially in the programming model (Jupiter API) that you covered in your articles. We publish detailed release notes in our User Guide for people upgrading from existing milestones.

We would love to get the remaining work done as quickly as possible. However, we are very short on time at the moment. As you probably know, we had some initial funding through a crowdfunding campaign, but that was merely a start. If there is a company out there with a strong interest in Java and JUnit that would like to help out, we would certainly be open to discussing possible options if it reached out to us.

—Marc Philipp
JUnit 5 team

JUnit 5—What Is It?

I want to know a little more about JUnit 5. What does JUnit 5 actually do?

—Nikhil Deshmukh

Indeed, we did not provide any basic introduction to JUnit, in part because it is an almost universal tool in Java development. To quote the entry in Wikipedia, "JUnit is a unit testing framework for the Java programming language. JUnit has been important in the development of test-driven development, and is one of a family of unit testing frameworks." In a nutshell, it's the primary tool by which Java developers test their own code before checking it in.

Missing Class

In the article "Implementing Design Patterns with Lambdas" in the November/December 2016 issue, I think you should explain where the class `OnlineBankingLambda` came from, as it is not included in the source code.

—Carlos Carvajal

This article was excerpted from the book Java 8 in Action, *whose lead author, Raoul-Gabriel Urma, replies: "For the code example, the learning outcome is really about the method* processCustomer *itself (the template method pattern) and how it can be parameterized with a behavior. The class* OnlineBankingLambda *is not important (actually, it doesn't matter what its implementation is as long as it declares the* processCustomer *method).*

You'll find in the book that its (pre–Java 8) parent class is:

```
abstract class OnlineBanking {
    public void processCustomer(int id){
        Customer c = Database.getCustomerWithId(id);
        makeCustomerHappy(c);
}
    abstract void makeCustomerHappy(Customer c);
}
```

After introducing the refactoring with behavior parameterization, it becomes:

```
abstract class OnlineBanking {
    public void processCustomer(
       int id, Consumer<Customer> makeCustomerHappy){
           Customer c = Database.getCustomerWithId(id);
           makeCustomerHappy.accept(c);
    }
}
```

and the `LambdaOnlineBanking` *class:*

```
class LambdaOnlineBanking extends OnlineBanking {
    // it's inheriting processCustomer
   // ... more stuff can come here
}
```

I think it would be simpler if I didn't declare an abstract class that you have to inherit from. I will change this for the second edition."

Back-Issue Downloads

Although I'm glad to have access to back issues, I'm puzzled as to how to download them. Could you explain?

—Several readers

The content delivery network requires downloads to be done from a desktop rather than a mobile device. So, if you go to the website for this issue from a desktop, you'll see a panel of icons on the right. Click the top icon and a new panel will open, showing the contents of the current issue plus the covers of the back issues from 2015 and 2016. Double-click the back issue you want to download. A button will pop up on the top of the panel marked "View." Click this button and the back issue will load in the browser. Once it has loaded, go the panel on the right, which has an icon with a down arrow. Click this icon and a PDF download of the issue will begin immediately. (Note that you can download a PDF of the current issue using the same technique.)

Contact Us

We welcome your comments and suggestions. Write to us at javamag_us@oracle.com. For other ways to reach us, see the last page of this issue.

JRebel

ZEROTURNAROUND

RELOAD CODE CHANGES INSTANTLY

TRY IT NOW!

Get a free t-shirt!

JRebel
Java just got faster...

Devoxx US

MARCH 21–23
SAN JOSE, CALIFORNIA

Devoxx US focuses on Java, web, mobile, and JVM languages. The conference includes more than 100 sessions in total, with tracks devoted to server-side Java, architecture and security, cloud and containers, big data, Internet of Things (IoT), and more.

PHOTOGRAPH BY THINKSTOCK/GETTY IMAGES

Jfokus

FEBRUARY 6, TUTORIAL
FEBRUARY 7–8, CONFERENCE
STOCKHOLM, SWEDEN

Jfokus is the largest annual Java developer conference in Sweden. Conference topics include Java SE and Java EE, continuous delivery and DevOps, IoT, cloud and big data, trends, and JVM languages. This year, the first day of the event will include a VM Tech Summit, which is an open technical collaboration among language designers, compiler writers, tool builders, runtime engineers, and VM architects. The schedule will be divided equally between traditional presentations of 45 minutes and informal, facilitated deep-dive discussion groups among smaller, self-selected participants. Space is limited, as this summit is organized around a single classroom-style room to support direct communication between participants.

SnowCamp 2017

FEBRUARY 8, WORKSHOP
FEBRUARY 9–10, CONFERENCE
FEBRUARY 11, "UNCONFERENCE"
GRENOBLE, FRANCE

This technical conference held in the French Alps focuses on Java, JavaScript, and software architecture, with sessions presented in French (primarily) and English. A workshop day precedes the two-day conference. An "Unconference" at the end is for hitting the slopes with your fellow attendees.

DevNexus

FEBRUARY 22–24
ATLANTA, GEORGIA

DevNexus is devoted to connecting developers from all over the world, providing affordable education, and promoting open source values. The 2017 conference will take place at the Georgia World Congress Center in downtown Atlanta. Presenters will include Josh Long, author of O'Reilly's upcoming *Cloud Native Java: Designing*

Resilient Systems with Spring Boot, Spring Cloud, and Cloud Foundry, and Venkat Subramaniam, author of Pragmatic's *Functional Programming in Java: Harnessing the Power of Java 8 Lambda Expressions*.

Voxxed Days Zürich

FEBRUARY 23
ZÜRICH, SWITZERLAND
Sharing the Devoxx philosophy that content comes first, Voxxed Days events see both internationally renowned and local speakers converge. Past presentations have included "Bringing the Performance of Structs to Java (Sort Of)," by Simon Ritter, and "Java Security Architecture Demystified," by Martin Toshev.

Topconf Linz 2017

FEBRUARY 28, WORKSHOPS
MARCH 1–2, CONFERENCE
LINZ, AUSTRIA
Topconf covers Java and JVM, DevOps, reactive architecture, innovative languages, UX/UI, and agile development. Presentations this year include "Java Libraries You Can't Afford to Miss," "8 Akka Antipatterns You'd Better Be Aware Of," and "Spring Framework 5: Reactive Microservices on JDK 9."

QCon London 2017

MARCH 6–8, CONFERENCE
MARCH 9–10, WORKSHOPS
LONDON, ENGLAND
For more than a decade, QCon London has empowered software development by facilitating the spread of knowledge and innovation in the developer community. Scheduled tracks this year include "Performance Mythbusting" and "Every Last Nanosecond: Low Latency Java."

jDays

MARCH 7–8
GOTHENBURG, SWEDEN
jDays brings together software engineers from around the world to share their experiences in different areas such as Java, software engineering, IoT, digital trends, testing, agile methodologies, and security.

ConFoo Montreal 2017

MARCH 8–10
MONTREAL, QUEBEC, CANADA
ConFoo Montreal is a multitechnology conference for web developers that promises 155 presentations by popular international speakers. Past ConFoo topics have included how to write better streams with Java 8 and an introduction to Java 9.

Embedded World

MARCH 14–16
NUREMBERG, GERMANY
The theme for the 15th annual gathering of embedded system developers is Securely Connecting the Embedded World. Topics include IoT, connectivity, software engineering, and safety and security.

JavaLand

MARCH 28–30
BRÜHL, GERMANY
This annual conference features more than 100 lectures on subjects such as core Java and JVM languages, enterprise Java and cloud technologies, IoT, front-end and mobile computing, and much more. Scheduled presentations include "Multiplexing and Server Push: HTTP/2 in Java 9," "The Dark and Light Side of JavaFX," "JDK 8 Lambdas: Cool Code that Doesn't Use Streams," "Migrating to Java 9 Modules," and "Java EE 8: Java EE Security API."

PHOTOGRAPH BY JAN REMUND/FLICKR

O'Reilly Software Architecture Conference

APRIL 2–3, TRAINING
APRIL 3–5, TUTORIALS AND CONFERENCE
NEW YORK, NEW YORK

This event promises four days of in-depth professional training that covers software architecture fundamentals; real-world case studies; and the latest trends in technologies, frameworks, and techniques. Past presentations have included "Introduction to Reactive Applications, Reactive Streams, and Options for the JVM," as well as "Microservice Standardization."

JAX DevOps

APRIL 3 AND 6, WORKSHOPS
APRIL 4 AND 5, CONFERENCE
LONDON, ENGLAND

This event for software experts features in-depth knowledge of the latest technologies and methodologies for lean businesses. The focus is on accelerated delivery cycles, faster changes in functionality, and increased quality in delivery. Conference tracks include agile and company culture, cloud platforms, container technologies, continuous delivery and automation, microservices, and real-world case studies. The conference is preceded and followed by a day of workshops. There's also a two-in-one conference package that provides free access to a parallel conference, JAX Finance.

Devoxx France

APRIL 5, WORKSHOPS
APRIL 6–7, CONFERENCE
PARIS, FRANCE

Devoxx France presents workshops, tutorials, and keynotes from prestigious speakers, followed by a cycle of eight mini conferences every 50 minutes. You can build your own calendar and follow the sessions as you wish. Founded by developers for developers, Devoxx France covers topics ranging from web security to cloud computing. (No English page available.)

GREAT INDIAN DEVELOPER SUMMIT

APRIL 25–28
BANGALORE, INDIA

The Great Indian Developer Summit (GIDS), now in its 10th year, offers four days of content grouped by theme. April 26 focuses on Java and JVM languages. Other days focus on web, mobile, DevOps, and big data. Register for each day separately.

Riga Dev Days 2017

MAY 15–17
RIGA, LATVIA

The biggest tech conference in the Baltic states, this three-day event is a joint project of Google Developer Group Riga, Java User Group Latvia, and Oracle User Group Latvia. By and for software developers, Riga Dev Days focuses on the most relevant topics and technologies for that audience with more than 50 sessions on Java, web, and cloud programming.

J On The Beach

MAY 17, WORKSHOPS
MAY 18–19, TALKS
MALAGA, SPAIN

JOTB is an international rendezvous for developers interested in big-data technologies. JVM and .NET technologies, embedded and IoT development functional programming, and data visualization will all be discussed. Scheduled speakers include longtime Java Champion Martin Thompson and Director of Developer Experience at Red Hat Edson Yanaga.

JEEConf

MAY 26–27
KIEV, UKRAINE

JEEConf is the largest Java conference in Eastern Europe. The annual conference focuses on Java technologies for application development. This year offers five tracks and more than 50 speakers with an emphasis on practical experience and development of real projects. Topics include modern approaches in the development of distributed, highly loaded, scalable enterprise systems with Java, among others.

jPrime

MAY 30–31
SOFIA, BULGARIA

jPrime is a relatively new conference, with two days of talks on Java, JVM languages, mobile and web programming, and best practices. The event is run by the Bulgarian Java User Group and provides opportunities for hacking and networking.

O'Reilly Fluent Conference

JUNE 19–20, TRAINING
JUNE 20–22, TUTORIALS & CONFERENCE
SAN JOSE, CALIFORNIA

Fluent offers practical training for building sites and apps for the modern web. This event is designed to appeal to application, web, mobile, and interactive developers, as well as engineers, architects, and UI/UX designers. Training days and tutorials round out the conference experience.

EclipseCon 2017

JUNE 20, "UNCONFERENCE"
JUNE 21–22, CONFERENCE
TOULOUSE, FRANCE

EclipseCon is all about the Eclipse ecosystem. Contributors, adopters, extenders, service providers, consumers, and business and research organizations gather to share their expertise. The two-day conference is preceded by an "Unconference" gathering.

QCon New York

JUNE 26–28, CONFERENCE
JUNE 29–30, WORKSHOPS
NEW YORK, NEW YORK

QCon is a practitioner-driven conference for technical team leads, architects, engineering directors, and project managers who influence innovation in their teams. The conference covers many different developer topics, frequently including entire Java tracks.

JavaOne

OCTOBER 1–5
SAN FRANCISCO, CALIFORNIA

Whether you are a seasoned coder or a new Java programmer, JavaOne is the ultimate source of technical information and learning about Java. For five days, Java developers gather from around the world to talk about upcoming releases of Java SE, Java EE, and Java FX; JVM languages; new development tools; insights into recent trends in programming; and tutorials on numerous related Java and JVM topics.

Have an upcoming conference you'd like to add to our listing? Send us a link and a description of your event four months in advance at javamag_us@oracle.com.

Oracle Code Events

Oracle Code is a free event for developers to learn about the latest development technologies, practices, and trends, including containers, microservices and API applications, DevOps, databases, open source, development tools and low-code platforms, machine learning, AI, and chatbots. In addition, Oracle Code includes educational sessions for developing software in Java, Node.js, and other programming languages and frameworks using Oracle Database, MySQL, and NoSQL databases.

US AND CANADA

MARCH 1, San Francisco, California
MARCH 8, Austin, Texas
MARCH 21, New York, New York
MARCH 27, Washington DC
APRIL 18, Toronto, Ontario, Canada
JUNE 22, Atlanta, Georgia

EUROPE AND MIDDLE EAST

APRIL 20, London, England
APRIL 24, Frankfurt, Germany
APRIL 28, Prague, Czech Republic
MAY 22, Moscow, Russia
JUNE 6, Brussels, Belgium

ASIA PACIFIC

MAY 10, New Delhi, India
MAY 17, Tokyo, Japan
JULY 14, Beijing, China
JULY 18, Sydney, Australia
AUGUST 30, Seoul, South Korea

LATIN AMERICA

JUNE 27, São Paulo, Brazil
JUNE 29, Mexico City, Mexico

DEVOXX™
UNITED STATES

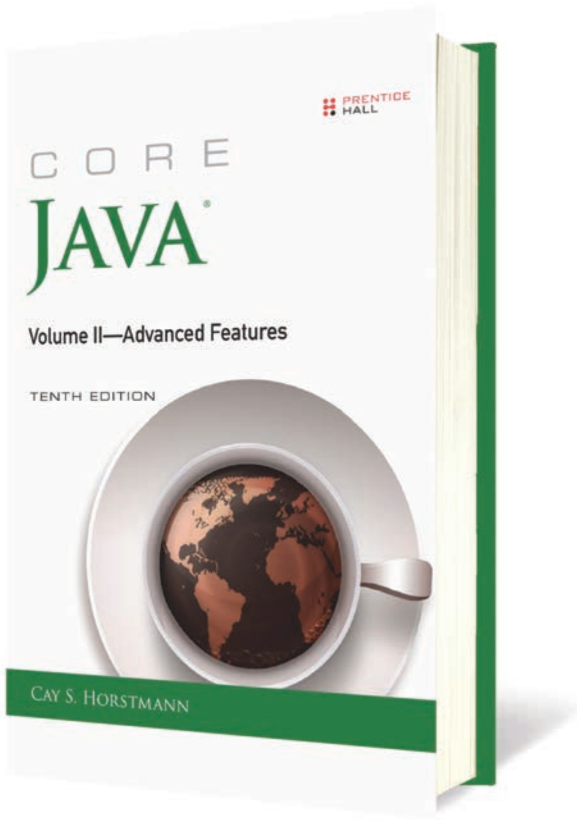
MARCH 21-23, 2017 | SAN JOSE, CA

WORLD'S LARGEST VENDOR INDEPENDENT DEVELOPER CONFERENCE IS COMING TO THE USA

WILL YOU BE THERE?

FROM DEVELOPERS, FOR DEVELOPERS

REGISTER NOW | WWW.DEVOXX.US



CORE JAVA, VOLUME II—ADVANCED FEATURES, 10TH EDITION

By Cay S. Horstmann
Prentice Hall

Few programming books have a guaranteed, preallocated place on my bookshelf, but updates to Volumes I and II of Cay S. Horstmann's *Core Java* have occupied the equivalent of front-row center seats on my shelf for years. That is because these two books together represent the clearest, deepest, and most extensive explanation of the Java language and its principal libraries. As with previous editions, each volume tips in at somewhat more than 1,000 pages—which no one could argue is insufficient. However, the sufficiency does not arise from the bulk but the thoughtfulness and depth of the explanations.

A shining example from Volume I, which I previously reviewed, is Horstmann's introduction to lambdas. I've read many explanations of lambdas, but not one explained them so approachably, with easy-to-grasp examples that successively pull readers along into the complexity of functional interfaces and eventually dump them out at inner classes. In this way, inner classes now make sense in the context of lambdas—rather than explaining inner classes *in order* to demonstrate lambdas.

Along the way, Horstmann throws in tidbits about the historical solutions that preceded lambdas, explains how lambdas differ from similar constructs in other languages, and offers warnings on usage while pointing out where unexpected traps lie. This is how you want to learn a language: deeply and with an author concerned about the reader. This approach also makes *Core Java* an excellent resource for later reference.

While Volume I presents the basics and focuses primarily on the language, the just-released Volume II lays out advanced features of likely interest to professionals. This volume covers the Java 8 Stream library, advanced I/O, XML, networking, database programming, the completely rewritten Date and Time API, annotation processing, security, native methods, and—curiously—advanced Swing and AWT.

In addition to the admirable text, Horstmann supports his explanations with snippets and, where necessary, complete programs exceeding 100 lines of lucid code.

My only gripe about Volume II is how long it took to come to market. It covers Java SE 8 (not Java 9), which means that more than two years elapsed for it to see the light of day—a long time for a reference and too long for a tutorial. Given that Java 9 is expected to ship this year, you see the problem. But for developers who expect to work with Java 8 for the next few years, there is no better coverage of the language and libraries than in this fine volume. *—Andrew Binstock*

IJ

IntelliJ IDEA

Making development*
an enjoyable experience

Get it now

JET
BRAINS

*Java, Groovy, Kotlin, Scala, Android and much more

Java and JVM Tools

POLYGLOT MAVEN 17 | INSIDE BUILD TOOLS 22 | JVM DEBUGGING 29 | LOCAL-VARIABLE TYPE INFERENCE 60

Few things are as exciting to us developers as tools. Nearly everything we do, we do indirectly through the use of tools on which we depend. And in fact, good developers are always characterized by deep knowledge of the tools they use and how best to apply them. With this in mind, in our first article (page 17) we examine the latest evolution of Maven, called Polyglot Maven, which frees us from creating build files in XML. Polyglot opens the door to using real scripting language for writing build files. This idea is taken one step further in our article (page 22) on a new build tool, Kobalt, being written in Kotlin, a Java-like JVM language. This article is actually more about the architecture of build tools—what goes on underneath the covers. However, the author's design uses Kotlin as the language for defining the build. In this way, as he points out, not only do you gain true expressivity in the build file, but your IDE can catch errors as you describe your build. (The increasingly popular build tool Gradle is moving in the same direction: migrating its build files from Groovy syntax to Kotlin.)

25482125482

We also cover (page 29) the uncommon, but not rare, situation in which your debugger does not give you all the information you need to fix your code. We examine a debugging API for the JVM that enables you to extract information about running processes, view variables and classes, query counters, and get at details you might need. This information is surprisingly easy to access and enables you to write one-off debugging scripts to solve unusual or complex problems.

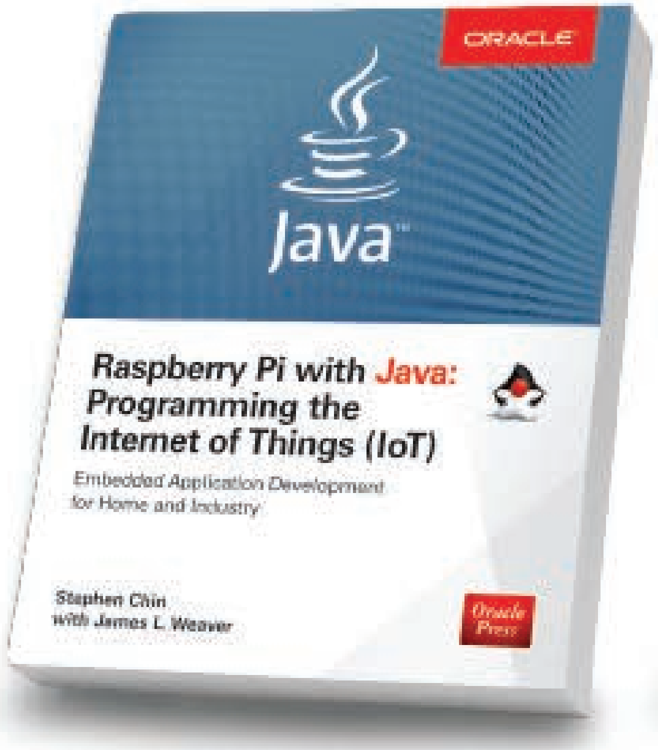
To complement this information, Oracle's Brian Goetz discusses lexical language changes supporting local-variable type inference (page 60). Also, we do a deep dive into Scala (page 47). Finally, one of the coolest articles (page 36) we've run in a long time: how to get started programming Blockchain, the technology behind cryptocurrencies. Add to this our in-depth language quiz (page 63), and we hope you'll find this issue to be packed with useful information.

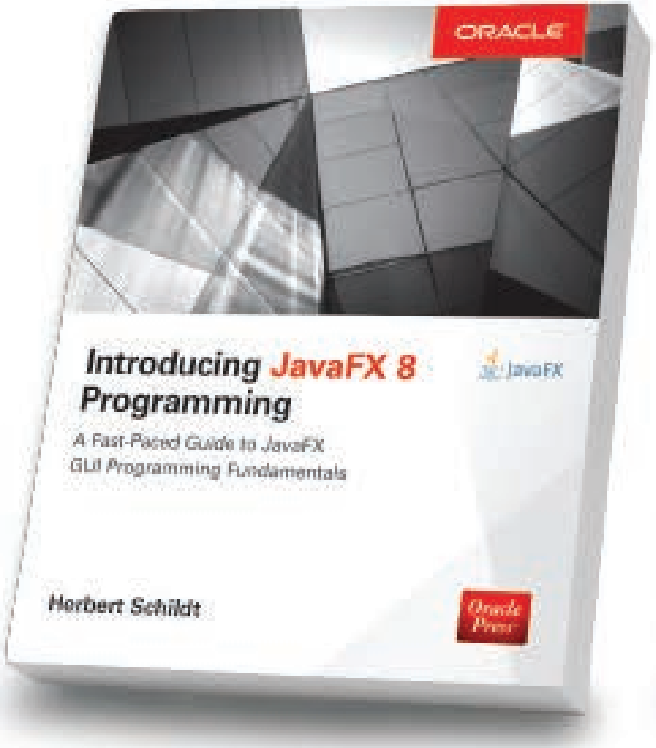
ART BY I-HUA CHEN

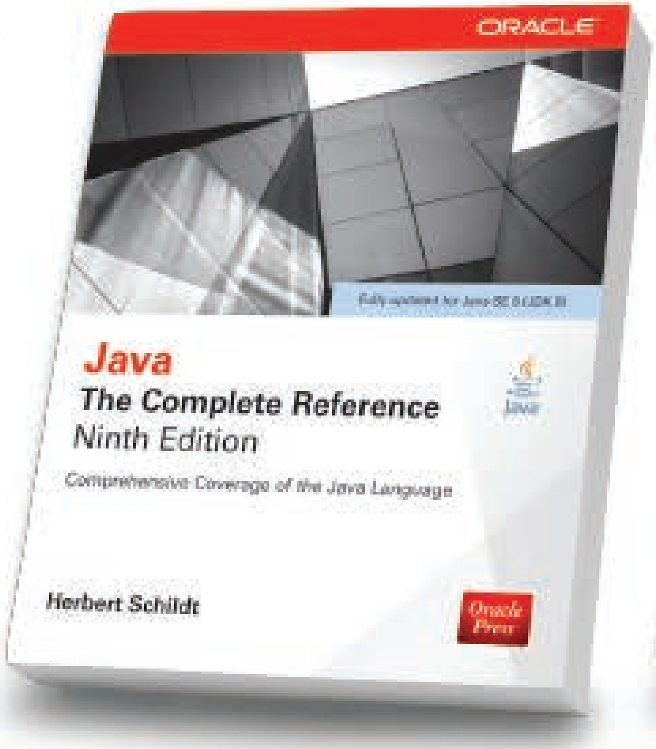
Oracle Press™

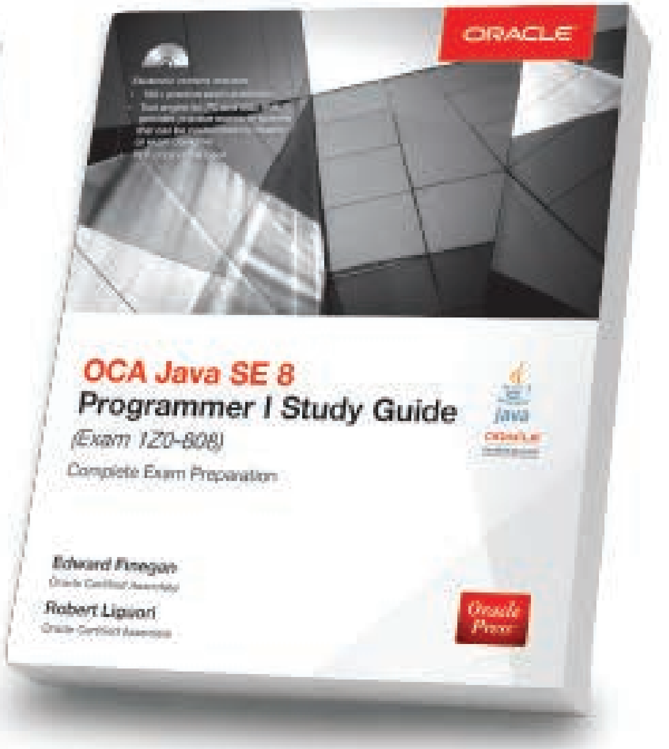
Your Destination for Java Expertise

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


ORACLE
Java™
Raspberry Pi with Java: Programming the Internet of Things (IoT)
Embedded Application Development for Home and Industry
Stephen Chin with James L. Weaver
Oracle Press


ORACLE
Introducing JavaFX 8 Programming
JavaFX
A Fast-Paced Guide to JavaFX GUI Programming Fundamentals
Herbert Schildt
Oracle Press


ORACLE
Java
The Complete Reference
Ninth Edition
Comprehensive Coverage of the Java Language
Herbert Schildt
Oracle Press


ORACLE
OCA Java SE 8
Programmer I Study Guide
(Exam 1Z0-808)
Complete Exam Preparation
Edward Finegan
Robert Liguori
Oracle Press

Raspberry Pi with Java: Programming the Internet of Things (IoT)
Stephen Chin, James Weaver
Use Raspberry Pi with Java to create innovative devices that power the internet of things.

Introducing JavaFX 8 Programming
Herbert Schildt
Learn how to develop dynamic JavaFX GUI applications quickly and easily.

Java: The Complete Reference, Ninth Edition
Herbert Schildt
Fully updated for Java SE 8, this definitive guide explains how to develop, compile, debug, and run Java programs.

OCA Java SE 8 Programmer I Study Guide (Exam 1Z0-808)
Edward Finegan, Robert Liguori
Get complete coverage of all objectives for Exam 1Z0-808. Electronic practice exam questions are included.

Oracle Press™

Available in print and as eBooks

www.OraclePressBooks.com • @OraclePress

JASON **VAN ZYL** AND
MANFRED **MOSER**

Polyglot for Maven: Moving Maven into Scripting

A new tool from the developer of Maven enables POM files to be written in Ruby, YAML, Groovy, and other languages.

Apache Maven is the most widely used build tool for the JVM. During the last decade, it introduced new ideas such as dependency management and the usage of repositories to the development ecosystem. The central repository, also known as Maven Central, is the default repository in Maven from which plugins and dependencies are downloaded. It is an exchange hub for open source artifacts in the JVM ecosystem and is used by Maven and other build tools such as Gradle, sbt, Ant/Ivy, and others. As such, Maven continues to be an important tool for the open source community as well as for enterprise users.

XML's Pros and Cons

One common complaint from Maven users over the years is the use of XML as the markup language for the main build file in the project object model (POM or pom.xml for short).

XML was a natural choice when Maven was created more than 10 years ago. It has proven to be very useful thanks to its stable structure, well-known syntax, and the powerful tooling available for it. However, times change, and XML is not the latest and greatest option anymore. Today developers demand a terse syntax and features such as inlined scripts. This article shows how Polyglot for Maven, an open source project, provides the ability to use Ruby, YAML, Groovy, and other languages to define the POM.

On the surface, Maven has always used the pom.xml file. A minimal pom.xml file simply defines an identifier for the project. It comprises the values for groupId, artifactId, and version—the so-called *GAV coordinates* (see Listing 1).

Listing 1.

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.example.project</groupId>
  <artifactId>hello-world-demo</artifactId>
  <version>1.0.0</version>
</project>
```

With this simple setup, you can build your JAR archive and as part of that build, you can assemble resources, compile source code and test it, and package the archive. This simple example shows the power of Maven, which derives from several built-in conventions. But it also shows how the verbose nature of the XML file can bury the important information.

In reality, a pom.xml file ends up typically being longer due to the inclusion of dependencies, configuration information for plugins, and other aspects about the build and the project at hand.

Picking up the pom.xml file from the project directory, parsing it, and then proceeding to initiate the build by invoking plugins in sequence is all wired into Maven. Under the hood, the pom.xml file is transformed into an actual POM of plain old Java objects (POJOs). This is where Polyglot for Maven comes into the picture. It provides extensions to Maven that allow Maven to read other files—such as pom.rb (in Ruby) or pom.groovy (in Groovy)—parse them, and create the POM from them. While this appears simple, there is—as usual—complexity in the details.

Polyglot for Maven supports writing POM files using Groovy, Ruby, Clojure, or Scala.

Maven Extensions

If you have been using Maven at all, you know that it is powered by many plugins that perform tasks such as copying and filtering resources, compiling code, running tests, creating packages, and many others. Maven calls these plugins to perform the individual tasks.

Extensions, by contrast, offer modifications or additions to the core of Maven itself. In the past, you were obliged to create a JAR file and add it to the classpath to accomplish this, and then you would have essentially created a custom Maven distribution. And managing custom distributions of tools in a large organization or open source project turns out to be a difficult chore. The extension-loading mechanism in Maven, on the other hand, allows you to simply define an extension as part of your project and use the standard Maven distribution.

Extensions are available via the central repository just like plugins and dependencies. To load them as part of your project build, you declare them in an extensions.xml file in the directory .mvn located in the root folder of your project (see **Listing 2**).

Listing 2.

```
<?xml version="1.0" encoding="UTF-8"?>
<extensions>
  <extension>
    <groupId>io.takari.polyglot</groupId>
    <artifactId>polyglot-ruby</artifactId>
    <version>0.1.19</version>
  </extension>
</extensions>
```

Plugins extend Maven by supporting different project lifecycles and goals, so new packaging types and tasks can be performed. In contrast, extensions change the behavior of Maven itself. Support for loading extensions and other configuration data via the .mvn folder has been part of the standard Maven releases since version 3.3.1.

Besides containing the file for loading extensions from a project local directory, the .mvn folder can also contain project-specific configuration for JVM parameters and Maven options in the jvm.config and maven.config files, respectively.

Markup Formats

Polyglot for Maven provides extensions for Maven that enable it to load POM files written in different dialects. The latest version is 0.1.19. The community has developed many examples in the project. These examples all contain the configuration for the relevant extensions.

Polyglot for Maven extensions use the `groupId` `io.takari.polyglot` and the `artifactId` value `polyglot-*`. So in addition to the Ruby version from **Listing 2**, there are also `polyglot-yaml`, `polyglot-groovy`, `polyglot-scala`, and other extensions, all of which are provided by the Polyglot for Maven project.

Listing 3 shows a sample Polyglot for Maven file: in this case, a minimal pom.yaml file. It contains the definition of the GAV coordinates and the setting of a property value.

Performing a build of this project in the yaml folder of the examples project is identical to performing a build for every other Maven project; that is, just run `mvn clean install`.

Listing 3.

```
modelVersion: 4.0.0
groupId: io.takari.polyglot
artifactId: yaml-project
version: 0.0.1-SNAPSHOT
name: 'YAML Maven Love'
properties: {sisuInjectVersion: 0.0.0.M2a}
```

The YAML syntax allows for the declaration of any list with a simple parent node. The example in Listing 4 shows a list of dependencies using this syntax together with the short syntax for GAV coordinates, including the dependency scope.

Listing 4.

```
dependencies:
  - {artifactId: junit, groupId: junit, scope: test,
      version: 4.12}
  - org.codehaus.groovy:groovy-all:2.4.3
```

Further examples of the YAML format are available as test resources in the main project. The YAML format is stable and mature. It is actively supported and used for daily development and release activities by the SnakeYAMLproject.

The Atom format is another non-XML-based format with a similar terseness to the YAML format. Take a look at the example project to have a glance at its syntax.

Another implementation of Polyglot for Maven with a markup language uses XML again: polyglot-xml. Yet instead of pure usage of elements, as done by the original core Maven pom.xml file, the new version uses attributes. This allows for the XML files to be substantially shorter, while maintaining the possible usage of XML-based tools.

The YAML, Atom, and XML formats all provide a different, terser syntax for the POM, which can feel more natural and readable to developers.

Language Formats

Polyglot for Maven goes beyond the markup formats and supports full programming language–based dialects for the markup using Groovy, Ruby, Clojure, or Scala. All these dialects bring the familiar syntax of their respective languages to the pom.groovy, pom.rb, pom.clj, and pom.scala files. In addition, they allow you to embed executable code snippets within the POM.

These formats are terse and look familiar to developers who use those programming languages. Each format allows code execution via the polyglot-maven-plugin and its execute goal. A minimal pom.rb file is valid with only a simple execute block, as shown in Listing 5.

Listing 5.

```
pom.ruby snippet with Ruby execution
execute :id => 'hello', :phase => 'validate' do
  puts 'hello world'
end
```

Similar functionality is wrapped in the build section of the pom.groovy snippet shown in Listing 6, which is written in Groovy.

Listing 6.

```
  build {
    $execute(id: 'hello', phase: 'validate') {
      println "Hello! I am Groovy inside Maven."
    }
  }
```

Real-World Usage

The language-based formats benefit from the fact that IDE support for these languages already exists. And both IntelliJ IDEA and the Eclipse IDE have started to support Polyglot for Maven.

The most prolific user of Polyglot for Maven so far has been the JRuby team. If you have a look at their project source code, you can find a good example of a larger, complex pom.rb file. You can see the beginning of the pom.rb file in Listing 7.

> **Once you are ready to ship a release,** ensure that a correct pom.xml is published to the central repository or your internal repository manager.

Listing 7.

```
version = ENV['JRUBY_VERSION'] ||
  File.read( File.join( basedir, 'VERSION' ) ).strip

project 'JRuby', 'https://github.com/jruby/jruby' do

  model_version '4.0.0'
  inception_year '2001'
  id 'org.jruby:jruby-parent', version
  inherit 'org.sonatype.oss:oss-parent:7'
  packaging 'pom'

  organization 'JRuby', 'http://jruby.org'

  [ 'headius', 'enebo', 'wmeissner', 'BanzaiMan',
    'mkristian' ].each
  do |name|
    developer name do
      name name
      roles 'developer'
    end
  end
```

Listing 7 shows some interesting usage patterns, such as reading an external property file with the version number and then using that value in the POM. The parent POM is defined with the `inherit` declaration. The developers are declared as a collection over which an iteration with `.each` defines the `developer` collection in the POM. You will also notice that a pom.xml file exists in the repository. This brings us to an important role the pom.xml file plays beyond defining the build.

The POM is a metadata container for many aspects of the project including the name, URL, source code manager information, and much more. Importantly, it also includes the declaration of parent POMs and dependencies.

These latter definitions are crucial for the consumption of artifacts. They enable Maven and other build tools using the central repository as well as other Maven repositories to determine which dependencies to download in addition to the directly declared dependencies or plugins. All these tools understand the pom.xml format.

So once you are ready to ship a release, you need to ensure that a correct pom.xml is published to the central repository or your internal repository manager with your artifacts. This allows the tools accessing your artifacts to read the metadata and download everything needed.

This download can be done by many different tools—including other build tools such as Gradle, sbt, Leiningen, and Ant/Ivy—that all work with the Maven repository format to retrieve dependencies. This functionality is based on the POM in its original XML format and is an important interoperability factor for the open source community.

Polyglot for Maven in Ruby and other languages, therefore, produces an XML-formatted POM in addition to the POM in Ruby. Support for this creation of an XML POM for consumption still needs to be worked out in some dialects. The lessons learned are brought back to the core Maven project, where a separation of this consumption POM from the build POM might provide a path to evolve the native XML POM format in a future version of Maven that won't break backward compatibility for already published artifacts.

A next step the JRuby community has taken is the creation of a native wrapper for Maven. It allows you to use the `rmvn` command, which automatically includes the Ruby extension of Maven, or to run Maven from within a Ruby script (see **Listing 8**).

Listing 8.

```
require 'ruby-maven'
RubyMaven.exec( '--version' )
```

Conclusion

Interest in and adoption of Polyglot for Maven have been rising since its inception a couple of years ago. It is slowly spreading into the open source community and into enterprises. By providing support for POM files that don't rely only on XML, new ways of expressing builds that are clearer and easier to write are now possible. Take a look at Polyglot for Maven yourself, and please do provide some feedback. `</article>`

Jason van Zyl is the founder of the Apache Maven project and many other successful open source projects. He started The Central Repository—the largest artifact exchange hub for the open source community—and he leads the development tooling and developer support team at a Fortune 500 company, bringing the results to the open source community at large.

Manfred Moser is an Apache Maven committer and plugin developer. He has trained and helped thousands of Maven users. Moser supports van Zyl as a trainer, author, and developer advocate.

THE CHICAGO JUG

The Chicago Java Users Group (CJUG) was started by Philip McGlauchlin in 1995 and has grown to nearly 2,000 members in 2016. It is one of the largest JUGs in the United States. The current group of officers includes Java authors, speakers, open source project leads, members of the JCP, the Apache Software Foundation, and a Java Champion.

CJUG's current focus is on expanding the contributor base to the Java ecosystem in Chicago. To accomplish this, CJUG has been running an Adopt-a-JSR program for the past two years. The program focuses on JSR 366 (Java EE 8) and Java 9. The effort is spearheaded by Josh Juneau and Bob Paulin. This year, the program featured a quarterly call-in discussion.

CJUG also supports membership via an online Slack community called the Chicago Tech on Slack. CJUG has taken a unique approach to sponsorship by constantly moving the meetup to different Chicago companies. This approach means that any developer can experience a broad cross-section of Chicago companies just by attending CJUG's bimonthly meetings.

While CJUG attracts name speakers, its focus is on developing local speaking talent via quarterly lightning talks. The group's current community leader is Java Champion Freddy Guime, who started the Java OffHeap podcast to provide current news and interviews with Java thought leaders.

CJUG is always looking for new contributors to join and help continue the mission of making Chicago a great place to be a Java developer.

CÉDRIC BEUST

The Design and Construction of Modern Build Tools

A look inside a modern JVM build tool—its architecture and implementation

The JVM has seen many build tools in the past 20 years. For Java alone, there's been Ant, Maven, Gradle, Gant, and others, while in JVM languages, sbt is used for Scala, and Leiningen for Clojure. These tools are run by developers and by back-end systems multiple times a day, yet very little is formally documented about how they are implemented, what functionalities they do and should offer, and why.

In this article, I give an overview of what it takes to create a modern build tool, the kinds of functionality you should expect, and how that functionality is implemented in modern tools. These observations are derived from my experience building software for decades and also from an experimental build tool I am working on called Kobalt, which performs builds for the JVM language Kotlin, previously described in this magazine.

Motivation

Kobalt was born from my observation that while the Gradle tool was a clear step forward in versatility and expressive build file syntax, it also suffered from several shortcomings, some of them related to its reliance on Groovy. I therefore decided to create a build tool inspired by Gradle but based on Kotlin, JetBrains' language.

Kobalt is not only written in Kotlin, its build files are also valid Kotlin programs with a thin DSL that will look familiar to seasoned Gradle users. (Note that the latest version of Gradle also adopted Kotlin for its build file syntax, and the Groovy build file is going to be phased out, thus validating the approach taken with Kobalt.)

In this article, I discuss general concepts of build files and demonstrate Kobalt's take on that feature. For readers unfamiliar with Kotlin, you should be able to follow along because the syntax is similar enough to Java's.

Morphology of a Build Tool

The vast variety of build tools available on the JVM and elsewhere share a common architecture and similar basic functionalities:

- The user needs to create one (or more) build files describing the steps required to create an application. The syntax to describe these build files is extremely varied: from simple property files to valid programming language source files and everything in between.
- The build tool parses this build file and turns it into a graph of tasks that need to be executed in a specific order.
- The tool then executes these tasks in the required order. The tasks can either be coded inside the build tool or require the invocation of external processes. The build tool should allow for tasks to do pretty much anything.

One of the oldest build tools on the JVM is Ant, which broke from the previous standard, make, that was in use up until that point. Ant introduced XML as the language for build files, which was quite innovative at the time. Ant also came up with the concept of tasks that can be defined either in the XML file or as external tasks implemented in Java that the tool looks up. Even though Ant is still used in legacy software, it is largely considered deprecated today and looked at as the "assembly language" of build tools: very flexible, offering a decent plugin architecture, but a tool that requires a lot of boilerplate for even the simplest build projects.

Now, let's dive more specifically into what you should expect from your build tool.

Syntax

Let's start with the most visible aspect of a build tool: the syntax of its build file.

Obviously, I want my build tool to have a clean and intuitive syntax, but you'll be hard-pressed to find a general consensus on what a "clean syntax" is, so I won't even try to define it. I would also argue that sometimes syntax is not as important as the ease with which you can write and edit your build file. For example, Maven's pom.xml file has a verbose syntax, but I've found that editing it is pretty easy when I use an editor that's aware of this file's schema.

Gradle's popularity came in good part from the fact that the syntax of its build file was Groovy, which is much more concise than XML. My personal experience has been that Gradle's build files are easy to read but hard to write, and I'll come back to this point below, but overall, there is a general agreement that the Gradle syntax is a step in the right direction.

What is more controversial is the question of whether a build file should be using a purely declarative syntax (for example, XML or JSON) or be a valid program in some programming language.

My experience has led me to conclude that I want the syntax to look declarative (because it's sufficient for most of my build files) but that I also want the power of a full programming language should I need it. And I don't want to be obliged to escape to some other language when I have such a requirement: the build file needs to use the same syntax. This observation came from my realization that I have often wanted to have access to the full power of object-oriented programming in my build files, so that I can create base-class tasks that I can specialize with a few variations here and there. For example, if my project creates several libraries, I want to compile all these libraries with the same flags, but the libraries need to have a different name and version. This kind of problem is trivially solved in object-oriented languages with inheritance and method overriding, so having this kind of flexibility in a build file is desirable.

The project directive (which is an actual Kotlin function) can take dependent projects as parameters, and Kobalt will build its dependency tree based on that information.

On a more general level, complex projects are often made of modules that share a varying degree of common settings and behaviors. The more your build system avoids requiring you to repeat these settings in multiple locations or different files, the easier it will be to maintain and evolve your build.

All build tools offer, in varying degrees, to help you set up your modules, their dependencies, and their shared parameters. But I can't say I have found one that makes this "inherit and specialize" aspect very intuitive—not even my own build tool. Maybe the solution is for modules to be represented by actual classes in the programming language of your build files so you can use the familiar "inherit and override" pattern to capture this reuse.

In the meantime, I think the approach of using a DSL that is a valid program and that offers you all the facilities of a programming language should you ever need them (`if`, `else`, classes, inheritance, composition, and so on) is a clear step in the right direction.

Dependencies

The #1 job of a build tool is to execute a sequence of tasks in a certain order and to take various actions if any of these tasks fail. It should come as no surprise that all the build tools I have come across (on the JVM or outside) allow you to define tasks and dependencies between these tasks. However, a lot of tools don't adequately address project dependencies: how do you specify that project C can be built only once projects A and B have been built?

No matter how extensive the build tool is, it will never be able to address all the potential scenarios that developers encounter every day, so it also needs to be expandable.

With Gradle, you need to manipulate multiple build .gradle files that, in turn, refer to multiple settings.gradle files. In this design, dependent modules need to be defined across multiple files with different roles (build.gradle and settings.gradle), which can make keeping track of these dependencies challenging.

When I started working on Kobalt, I decided to take a more intuitive approach by making it possible to define multiple projects in one build file, thereby making the dependency tree much more obvious. Here is what this looks like:

```
val lib1 = project {
  name = "lib1"
  version = "0.1"
}

val lib2 = project {
  name = "lib2"
  version = "0.1"
}

val mainProject = project(lib1, lib2) {
  name = "mainProject"
  version = "0.1"
}
```

The project directive (which is an actual Kotlin function) can take dependent projects as parameters, and Kobalt will build its dependency tree based on that information. All the projects are then sorted topologically, which means the build order can be either "lib1, lib2, mainProject" or "lib2, lib1, mainProject"—both of which are valid.

Also, note that the previous example is a valid and complete build file (and the repetition can be abbreviated further, but I'm keeping things simple for now).

Being able to keep the project dependencies in one centralized place makes it much easier to understand and modify a build, even for a complex project. Using multiple build files in the subprojects' own directories should still be an option, though.

Make Simple Builds Easy and Complex Builds Possible

A direct consequence of the functionality described in the previous section is that the build tool should let you create build files that are as bare bones as possible.

Convention over configuration. Most projects usually contain just one module and are pretty simple in nature, so the build tool should make such build files short, and it should implement as many sensible defaults as possible. Some of these defaults include those shown in Table 1.

With such defaults, the simplest build file for a project should literally be less than five lines long. And of course, the build tool could perform further analysis to do some additional guessing, such as inferring certain dependencies based on the imports.

Complex builds. Once you get past simple projects, the ability to easily modify a build is critical. Such modifications can be made statically (with simple changes to the build files) or dynamically (passing certain switches or values to the build run in order to alter certain settings). The latter operation is usually performed with profiles—values that trigger different actions in your build without having to modify your build file.

Maven has native support for profiles, but Gradle relies on the extraction of environment values in Groovy to achieve a similar result, which reduces its flexibility. Profiles in Kobalt combine these two approaches with conditionals. You define profiles as regular Kotlin values, as shown here:

```
val experimental = false
val premium = false
```

You can use them in regular conditional statements anywhere in your build file, as shown in the following examples:

```
val p = project {
  name = if (experimental) "project-exp"
         else "project"
  version = "1.3"
  ...
```

NAME	NAME OF THE CURRENT DIRECTORY
VERSION	"0.1"
LANGUAGE(S)	AUTOMATICALLY DETECTED
SOURCE DIRECTORIES	`src/main/java, src, src/main/{language}`
MAIN RESOURCES	`src/main/resources`
TEST DIRECTORIES	`src/test/java, test, src/test/{language}`
TEST RESOURCES	`src/test/resources`
MAVEN REPOSITORIES	MAVEN CENTRAL, JCENTER
BINARY OUTPUT DIRECTORY	{SOMEROOT}`/classes`
ARTIFACT OUTPUT DIRECTORY	{SOMEROOT}`/libs`

Table 1. Sensible defaults for a Java-aware build tool

Profiles can then be activated on the command line:

```
./kobaltw -profiles \
    experimental,premium assemble
```

This is an area where having your build file written in a programming language really brings benefits, because you can insert profile-triggered operations anywhere that is legal in that programming language:

```
dependencies {
  if (experimental)
    "com.squareup.okhttp:okhttp:2.5.0"
  else
    "com.squareup.okhttp:okhttp:2.4.0"
}
```

Here, `if (experimental)` refers to the profile specified on the command line.

Performance

You want your build tool to be as fast as possible, which means that the overhead it imposes should be minimal and most of the time building should be expended by the external tools invoked by the build. On top of this obvious requirement, the build tool should also support two important features needed for speed: incremental tasks and parallel builds.

Incremental tasks. For the purposes of build tools, a task is incremental if it can detect all by itself whether it needs to run. This is usually determined by calculating whether the output of the current run would be the same as that of the previous run and returning right away if such is the case. Most build tools support incremental tasks to varying degrees, and you can test how well your build tool performs on that scale by running the same command twice in a row and see how much work the build tool performs during the second run.

Parallel builds. In contrast to incremental tasks, few build tools support parallel builds. I suspect the reason has a lot more to do with hardware than software; the algorithms to run tasks in parallel in the correct order are well known (if you are curious, look up "topological sorting"), but until recently, running build tasks in parallel wasn't really worth it and didn't always result in faster builds because of a simple technological hurdle: mechanical hard drives.

Build tasks are typically very I/O intensive. They read and write a lot of data to the disk, and with mechanical hard drives, this results in a lot of "thrashing" (the head of the hard drive being moved to different locations of the hard drive in rapid succession). Because build tasks keep being swapped in and out by the scheduler, the hard drive must move its head a lot, which results in slow I/O.

The situation has changed these past few years with the emergence of solid-state drives, which are much faster at handling this kind of I/O. As a consequence, modern build tools should not only support parallel builds but actually make them the default.

For example, **Figure 1** is a diagram showing the multiple modules and their dependencies in the ktor project, a Kotlin web server.

From this diagram, you can see that the `core` module needs to be built first. Once this is done, several modules can then be built in parallel, such as `locations` and `features` (as they both depend on `core`, but not on each other). As more modules are built, additional modules are scheduled to be built based on the dependency graph in **Figure 1**.

The gain in build time with parallel build can be significant. For example, a recent build that was timed at 68 seconds in parallel required 260 seconds in a sequential build—effectively, a 4x differential.

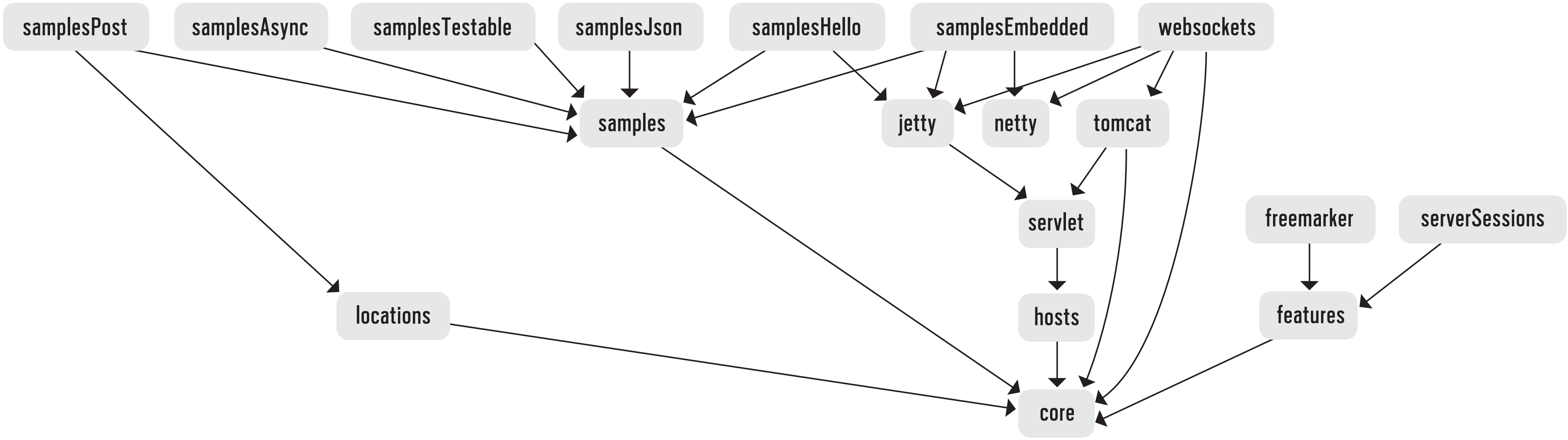


Figure 1. The dependencies in the ktor project

Plugin Architecture

No matter how extensive the build tool is, it will never be able to address all the potential scenarios that developers encounter every day, so it also needs to be expandable. This is traditionally achieved by exposing a plugin architecture that developers can use to extend the tool and have it perform tasks it wasn't originally designed for. A build tool's usefulness is directly correlated to the health and size of its plugin ecosystem.

These days, nobody has time to go to a website, download a package, and manually install it. **Build tools should be no exception, and they should self-update.**

Interestingly, while OSGi is a respectable and well-specified architecture for plugin APIs, I don't know of any build tool that uses it. Instead, build tools tend to invent their own plugin architecture, which is unfortunate.

This topic would require an entire book chapter of its own, so I'll just mention that there are basically two approaches to plugin architecture. The first one is to give plugins full access to the internal structure of your tool, which is the approach adopted by Gradle (driven and facilitated by Gradle's Groovy foundation).

In contrast, the Eclipse and IntelliJ IDEA development environments and Kobalt expose documented endpoints that plugins can connect to and use in order to observe and modify values in an environment that the build tool completely controls. I prefer this approach because it's statically verifiable and much easier to document and maintain.

Package Management

On top of being a very versatile and innovative build system, Maven introduced what will most likely be a legacy that will far outlast it: the repository. I'm pretty sure that even if Maven becomes outdated and no longer used, we'll still be referencing and downloading packages from the various Maven repositories that are available today.

Given the popularity of these repositories, a modern build tool should do the following:

- Transparently support the automatic downloading of dependencies from these repositories (Maven, Gradle, and Kobalt all do this; Ant requires an additional tool).
- Make it as easy as possible to make my projects available on these repositories. Maven and Gradle require plugins and quite a bit of boilerplate in your build file; Kobalt natively supports such uploads.

Auto-Completion in Your IDE

Developers spend several hours every day in their IDE, so it stands to reason that they would expect all the facilities offered by their IDE to be available when they edit a build file. Build tools have been moderately successful at this, frequently offering partial syntactical support.

Interestingly, Maven with its POM file has always been very well supported in IDEs because of its reliance on an XML format that's fully described in an XML schema. The file is verbose, but auto-completion is readily available and reliable, and Maven's schema is a very good example of how to define a proper XML file with very strict rules (for example, no attributes are ever used, so there's never any hesitation about how to enter the data).

The more modern Gradle has been less successful in that area because of its reliance on Groovy and the fact that this language is dynamically typed. Kobalt's reliance on Kotlin for its build file enables auto-completion to work in IntelliJ IDEA, without requiring any special efforts. Obviously, the upcoming Kotlin-based Gradle will enable similar levels of auto-completion as well.

Self-Updating

These days, nobody has time to go to a website, download a package, and manually install it. Build tools should be no exception, and they should self-update (or, at least, make it easy for you to update the tool). If the tool is available on a standard package manager on your system (brew, dpkg, and so on), great. But it should also be able to update itself so that such updates can be uniform across multiple operating systems. However, the tool itself is not the only part of your build that you want to keep updated. Dependencies are very important as well, and your build tool should help you stay current with these by informing you when new releases of dependencies are available.

Conclusion

Software built in 2016 is much more complex than it was 20 years ago, and it's important to hold our tools to a high standard. Libraries, IDEs, and design patterns are only a few components that need to adapt and improve as our software needs increase. Build tools are no exception, and we should be as demanding of them as we are of every other type of tool we use. `</article>`

Cédric Beust (@cbeust) has been writing Java code since 1996, and he has taken an active role in the development of the language and its libraries through the years. He holds a PhD in computer science from the University of Nice, France. Beust was a member of the expert group that designed annotations for the JVM.

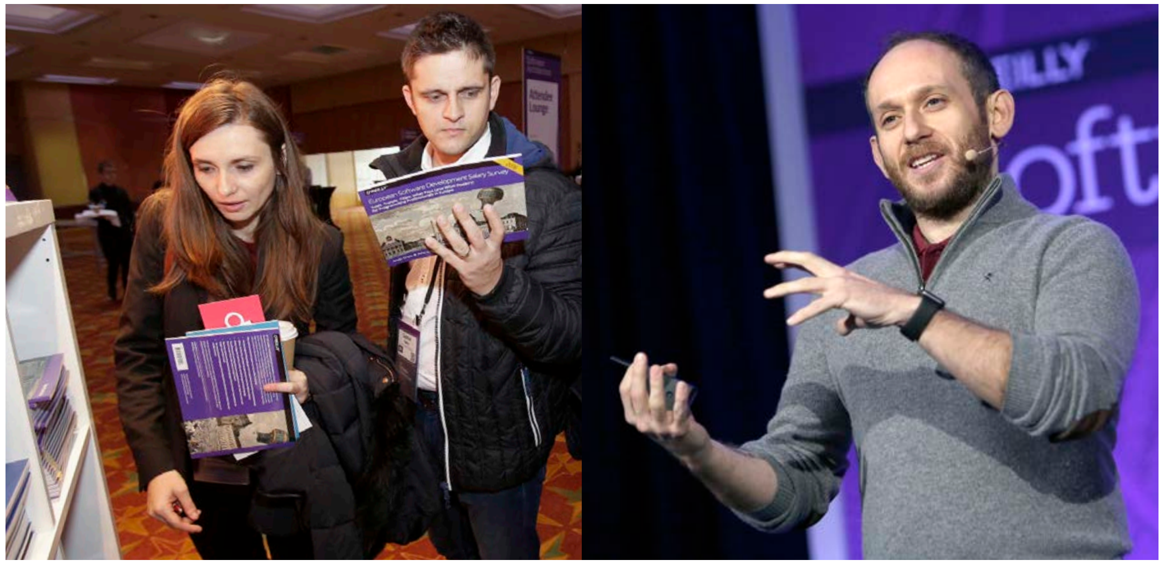
learn more

Gradle
Kotlin

O'REILLY®

Software Architecture
ENGINEERING THE FUTURE OF SOFTWARE

The brightest minds in tech are coming to New York.



Practical training in the tools, techniques, and leadership skills you need for the evolving world of software architecture.

April 2–5, 2017
softwarearchitecturecon.com/ny

JAVA MAGAZINE READERS GET **20% OFF** WITH CODE **JAVA**

ANDREI **PANGIN**

Creating Your Own Debugging Tools

JDK serviceability technologies allow you into the JVM to solve difficult debugging problems.

Java is more than just a programming language; it is a comprehensive platform for developing and running application software. One of the most recognized advantages of the Java platform is the large set of serviceability and maintainability features that are especially important for enterprise applications. Besides the numerous built-in tools, there is a lot of third-party software to assist with troubleshooting.

Every decent IDE for Java has a powerful debugger that supports step-by-step execution, breakpoints, watches, and so on. To address performance issues, there are well-known profilers such as Oracle Java Mission Control, JProfiler, and YourKit. Memory leaks are another prevalent problem. There are appropriate tools to deal with memory issues, too—for example, VisualVM and Eclipse Memory Analyzer. General-purpose tools are good for solving typical problems developers often face. However, they do not always fit uncommon situations. I believe most of you know how to use standard tools, so here I discuss how to create new tools customized for your specific cases.

Why Build a Custom Tool?

Imagine a situation in which something has gone wrong with an application on a production server while you are out of the office. It would be helpful to take a heap dump for analysis, but the internet connection might not be good enough to download a multigigabyte dump file. The only thing you can do is run something small remotely. But what tool would you run? Sometimes it is faster to make a specialized tool by yourself rather than search for an existing one that might or might not help in a particular case.

Another instance where a custom tool can be effective is with the problem of ignored exceptions. It is a common mistake to catch declared exceptions and discard them without handling them. The problem is worse when this happens in a third-party library that you cannot modify. I encountered this bug in a proprietary JDBC driver that did not handle an unexpected error from a database server and, thus, could not properly invalidate a stale connection. Even when you have no control over exceptions in a third-party library, you can still intercept them with a specialized tool. I show how to do that in this article.

If you ever wanted to patch a running application without creating a service interruption, you might be interested in a tool capable of modifying loaded code. Of course, there are commercial solutions that can do the job, but why not fix the problem yourself with just a few lines of code?

The list of tasks that might benefit from a custom tool is endless. Thanks to serviceability components included in Java SE Development Kit (JDK), the creation of such tools is much easier. While each of these components deserves a separate article, the following overview sheds light on what these technologies can do.

jvmstat Performance Counters

Monitoring the JVM is one of the key approaches to ensure that a system works well. Java HotSpot VM provides a huge amount of telemetry data through jvmstat performance counters. This data includes several hundred indicators covering nearly all JVM areas: class loading, garbage collection, multithreading, just-in-time compilation, and more. Despite the name, the tools are not all actually counters, and not all of them are about performance; nevertheless, they are very useful for monitoring JVM health. You might think of performance counters as gauges and dials in the cockpit of an aircraft.

You can ask the JVM to dump a Java heap, print stack traces, change certain VM flags, load an agent library, and so on.

jvmstat counters are available at no cost; that is, Java HotSpot VM exports them anyway, whether you read them or not. The JVM publishes the telemetry data onto the file system as a memory-mapped file in a temporary directory, often called /tmp/hsperfdata_{user}/{pid}, where *{pid}* is a Java process ID. This naming convention makes it possible for tools to find running Java processes in the system.

Fortunately, there is no need to replicate the directory scanning logic, because there is already a convenient Java API for that. Although the jvmstat API is not standard, it is supplied with the standard JDK bundle. You only need to include {JAVA_HOME}/`lib/tools.jar` in the classpath.

Here is how to get the process IDs of all running Java HotSpot VMs in the system:

```
import sun.jvmstat.monitor.MonitoredHost;
...
MonitoredHost host =
    MonitoredHost.getMonitoredHost((String) null);

Set<Integer> processIds = host.activeVms();
```

In this code, `null` stands for the local host. It is also possible to monitor remote virtual machines if a remote host runs the `jstatd` utility. Once you have a process ID, you can obtain an instance of `MonitoredVm` and read its jvmstat counters (or monitors). The `Monitor` type can be Integer, Long, or String. For example, a monitor named `sun.rt.javaCommand` contains the main class and the arguments used to start the given Java application. To get all the monitors matching the specified regular expression, use the `findByPattern` method, as shown next:

```
MonitoredVm vm = host.getMonitoredVm(
    new VmIdentifier(processId.toString()));

vm.findByPattern(".*").forEach(monitor -> {
    System.out.println(monitor.getName() + " = " +
                       monitor.getValue());
});
```

That simple code lists all available monitors with their values. In its output, you can find interesting metrics that are hardly shown by standard tools. For example:

```
// Total time spent in class initializers
sun.cls.classInitTime = 2545394
// Number of contended synchronizations
sun.rt._sync_ContendedLockAttempts = 55
// Duration of stop-the-world VM pauses
sun.rt.safepointTime = 811588
```

For instance, safepoint time is critical for low-latency applications, because it shows how long the application threads were forcibly stopped by the VM. If you choose to monitor safepoint pauses or any other of the 250+ counters, this is already a good monitoring tool, isn't it? It could be further improved to show a dynamic profile collected over time.

Unfortunately, you cannot do much just with read-only counters. Two-way communication with the JVM requires a different technology.

Dynamic Attach and Instrumentation API

The Dynamic Attach mechanism provides the means to connect to a running VM and execute one of several predefined commands. You can ask the JVM to dump a Java heap, print stack traces, change certain VM flags, load an agent library, and so on. The VM executes commands on its own, so it must be alive and healthy in order to respond.

The Java API for Dynamic Attach is available in the same tools.jar file. Note that this is a vendor-specific API applicable only to OpenJDK and Oracle's JDK.

Attaching to a running Java process is straightforward; you need to know only the target process ID (pid) as shown in the following code. (Dynamic Attach requires no special VM options. It can connect to any local HotSpot JVM unless it is started with the `-XX:+DisableAttachMechanism` flag.)

```
import com.sun.tools.attach.VirtualMachine;
...
VirtualMachine vm = VirtualMachine.attach(pid);
try {
    vm.loadAgent(agentJarPath, options);
} finally {
    vm.detach();
}
```

This shows how to inject a Java agent into a running VM. A *Java agent* is a utility program for instrumenting an application. It should be packed into a JAR file and contain a class with an `agentmain` method.

The instrumentation API enables Java agents to transform the bytecode of existing classes. When used together with Dynamic Attach, it enables you to change the code of a running application, even if the application is started without any debugging facilities. Here is a simple agent that installs a new version of `MyClass`:

```
public static void agentmain(String args,
            Instrumentation instr) throws Exception {

    Class oldClass = Class.forName("org.pkg.MyClass");
    Path newFile = Paths.get("/path/to/MyClass.class");
    byte[] newData = Files.readAllBytes(newFile);

    instr.redefineClasses(
            new ClassDefinition(oldClass, newData));
}
```

Remember the limitations of the `redefineClasses` API: a new version of a class file cannot add new methods or fields, nor can it remove existing members. Basically, only method bodies can be changed, but this is often enough for a hot fix.

The ability to instrument running Java processes makes the Attach API an important tool for maintenance of enterprise applications. The Dynamic Attach mechanism requires full cooperation from the JVM. It becomes useless if the JVM has hung or become too busy. When this happens, it is time to call for brute force, such as the serviceability agent.

Serviceability Agent

The HotSpot Serviceability Agent (SA) provides a low-level view of a Java process from a VM perspective. It knows everything about Java HotSpot VM internal structures, including the heap layout, the system dictionaries, the compiled code, the threads, and the stacks. Moreover, this information is available through a clear and simple Java API, so developers can benefit from it without having experience in disassemblers and other hacker facilities.

The SA was originally invented by Java HotSpot VM engineers for debugging crashes inside the JDK. However, they later realized that it could be helpful for a wider group of developers, and now it is bundled with the regular JDK. Start using the SA by including {JAVA_HOME}`/lib/sa-jdi.jar` in the classpath, but remember that the API is not standard and is subject to change in any future JDK release.

Custom tools typically extend an existing `Tool` class, which is already capable of parsing arguments and attaching to a running VM. You just need to implement custom logic inside the overridden `run` method.

```
import sun.jvm.hotspot.runtime.VM;
import sun.jvm.hotspot.tools.Tool;

public class MyTool extends Tool {

    @Override
    public void run() {
        // Actual implementation
        VM.getVM()...
    }

    public static void main(String[] args) {
        new MyTool().execute(args);
    }
}
```

`VM.getVM()` is the starting point to access Java HotSpot VM internal structures. The next example employs `SystemDictionary` to traverse all loaded classes with their class loaders. A similar technique might be useful in detecting memory leaks related to class loading.

```
VM.getVM().getSystemDictionary()
        .classesDo((klass, loader) -> {
    String className = klass.getName().asString();
    System.out.print(className);

    String loaderName = (loader == null)
            ? "Bootstrap ClassLoader"
            : loader.getKlass().getName().asString();
    System.out.println(" loaded by " + loaderName);
});
```

That was rather simple. The real power of the SA is to restore VM structures, either from the memory of a live Java process or from the core dump of an abnormally terminated process when the operating system is configured to create such dumps. The SA provides the reflection-like API to inspect Java objects and to extract the required fields. Unlike the reflection, which works from within the same process, the SA reads memory of a different process or parses a core dump file. Tools based on this feature can do impressive tricks such as stealing private keys from a running web server. The following code scans the heap of a target process looking for the instances of `java.security.PrivateKey` and printing their contents.

```
Klass keyClass = VM.getVM().getSystemDictionary()
        .find("java/security/PrivateKey", null, null);

VM.getVM().getObjectHeap()
      .iterateObjectsOfKlass(new DefaultHeapVisitor() {
   @Override
   public boolean doObj(Oop obj) {
      InstanceKlass c = (InstanceKlass) obj.getKlass();
      OopField f = (OopField) c.findField("key", "[B");
      TypeArray key = (TypeArray) f.getValue(obj);
      key.printOn(System.out);
      return false;
   }
}, keyClass);
```

The SA needs no cooperation from the target JVM, and there is no way to protect against SA interactions. This is not a reason to worry, though. The SA typically requires root privileges to attach to a running process. Also, keep in mind that the target JVM remains suspended while the SA is attached.

The JVM tool interface (JVM TI) is a standard programming interface designed especially for debugging, monitoring, and profiling software intended to run on top of the JVM.

So far, I have focused on JDK internal APIs. If you are looking for a more standard way to build your own tool, consider using the JVM tool interface.

JVM Tool Interface

The JVM tool interface (JVM TI) is a standard programming interface designed especially for debugging, monitoring, and profiling software intended to run on top of the JVM. The best thing about JVM TI is its public specification, which is not tied to any particular implementation. It is not required that every JVM provide all JVM TI functionality; however, most popular JVMs do.

The interface is exposed through the header file `jvmti.h`. JVM TI–based tools, called agents, are typically written in C or C++. They run within the same process and communicate with the JVM directly by calling JVM TI functions. The interface looks somewhat similar to Java Native Interface (JNI), so if you have ever written JNI code, you will easily grasp the principles of using JVM TI.

An agent may start at JVM bootstrap (when specified in `-agentlib` or `-agentpath` JVM arguments), or it can be loaded later at runtime using the Dynamic Attach mechanism. To support these options, an agent should define one or several entry points:

- `Agent_OnLoad`, which is called automatically by the JVM early at startup time
- `Agent_OnAttach`, which is called whenever the library is loaded at runtime

The first thing an agent typically does is get the reference to the JVM TI environment (`jvmtiEnv`), which is necessary for calling JVM TI functions.

```
#include <jvmti.h>

JNIEXPORT jint JNICALL
Agent_OnLoad(JavaVM *vm, char *args, void *unused) {
    jvmtiEnv *jvmti;
    vm->GetEnv((void**)&jvmti, JVMTI_VERSION_1_0);

    // Initialization code

    return 0;
}
```

JVM TI has functions for everything debuggers usually do. You can manage threads, walk through their stacks, iterate through the Java heap, query local variables, set breakpoints, manipulate Java classes, intercept native methods, and do many other things. Besides that, an agent may subscribe to event notifications: the JVM will invoke a provided callback function whenever an event occurs.

The access to JVM TI functionality is capability-based; that is, an agent must explicitly request the capabilities it is going to use. Most of the capabilities are available at runtime, but some can be requested only during the `OnLoad` phase (the time when no classes have been loaded and no bytecodes have been executed). For example, the `can_access_local_variables` capability is available only at startup, because the JVM needs to disable certain optimizations beforehand in order to retain information about all local variables.

The following example requests a capability to generate exception events and sets up the callback to receive notifications about all thrown Java exceptions, both caught and uncaught.

```
jvmtiCapabilities capabilities = {0};
capabilities.can_generate_exception_events = 1;
jvmti->AddCapabilities(&capabilities);

jvmtiEventCallbacks cb = {0};
cb.Exception = ExceptionCallback;

jvmti->SetEventCallbacks(&cb, sizeof(cb));
jvmti->SetEventNotificationMode(
    JVMTI_ENABLE, JVMTI_EVENT_EXCEPTION, NULL);
```

The callback function receives all details about an exception: a thread, a method, and the bytecode index for the thrown exception. The callback also has a reference to the JNI environment. This means you can invoke any JNI function from within. For instance, you can use JNI to call `Throwable.printStackTrace()`. Thus, an agent will print all the exceptions, including ignored exceptions, just before they are caught.

```
void JNICALL ExceptionCallback(
        jvmtiEnv *jvmti, JNIEnv *env, jthread thread,
        jmethodID method, jlocation location,
        jobject exception, jmethodID catch_method,
        jlocation catch_location)
{
    jclass cls = env->FindClass("java/lang/Throwable");
    jmethodID print_method = env->
        GetMethodID(cls, "printStackTrace", "()V");
    env->CallVoidMethod(exception, print_method);
}
```

You can do a lot more useful things with the JVM TI. Besides exceptions, it is possible to trace class loading, garbage collection, lock contention, thread activity, and more.

JVM TI is often confused with the Java debugger agent. There is a popular misconception that JVM TI compromises security and degrades the performance of Java applications. However, the Java Debug Wire Protocol (JDWP) agent is just one example of a JVM TI–based tool; the technology itself does not imply security or performance consequences. Whether an application will suffer from agent overhead solely depends on what the agent does and which capabilities it requests. Consider JVM TI as a sort of extension to JNI. This technology is definitely worth trying.

Changes in JDK 9

All the technologies discussed previously, including private APIs, will remain functional in the upcoming JDK 9. However, the new module system imposes certain restrictions on how you can access these APIs. No longer will tools.jar and sa-jdi.jar be separate libraries. JDK 9 serviceability features are supplied in the dedicated modules. **Table 1** shows the location of key JAR files in Java 9.

By default, applications cannot access an API from the modules that do not export packages externally. In order to use the private APIs, you need to explicitly break the encap-

FEATURE	MODULE	PUBLIC
JVMSTAT PERFORMANCE COUNTERS	`jdk.jvmstat`	NO
DYNAMIC ATTACH API	`jdk.attach`	YES*
INSTRUMENTATION AP	`java.instrument`	YES
SERVICEABILITY AGENT	`jdk.hotspot.agent`	NO

* `com.sun.tools.attach.VirtualMachine` is accessible from outside, but `sun.tools.attach.HotSpotVirtualMachine` is not.

Table 1. The location of serviceability APIs in Java 9

sulation with the `--add-exports` JVM command-line argument. For example, to run a tool that depends on the sun.jvmstat.monitor package, use:

```
java --add-exports jdk.jvmstat/sun.jvmstat.monitor=
     ALL-UNNAMED MyTool
```

[The previous line should be written as a single line with no space after the = sign. —*Ed.*]

Feel free to use the private APIs for your own purposes, but do it with care: there are no guarantees that the APIs will continue to work in future JDK updates.

Conclusion

The Java platform comes with a set of versatile technologies for building custom debugging, monitoring, and troubleshooting tools. Some of them are covered by Java SE standards, while others are specific to OpenJDK and Oracle's JDK. Despite the lack of thorough documentation on private APIs, the source code of the OpenJDK project (particularly, the source code of JDK built-in tools) might serve as a good starting point for learning serviceability technologies.

Software development and maintenance can hardly succeed without proper tools. Although many tools exist in the market, there is no silver bullet to address all problems. As a Java developer, you can create your own tools to solve tasks that no other software solves. JDK serviceability technologies are your friends. </article>

Andrei Pangin (@AndreiPangin) leads the development of the Odnoklassniki social network. He previously worked on the HotSpot JVM, which became his favorite topic and area of expertise. Pangin is a frequent speaker at Java conferences and one of the top JVM answerers on Stack Overflow.

ORACLE
UNIVERSITY

Get Java Certified

Earn Your #javatshirt

Java Certified

Danny Muthama
@DannyMuthama
#oca #javatshirt @OracleCert achievement unlocked
8 Dec 2016

swati sharma
@swati_sharma_
Got my new #javatshirt. Preparing for OCP
16 Oct 2016

Abdullah Shabbir
@FarigLog
Thanks Oracle for the Java tshirt. Now preparing for Java OCP. #javatshirt
14 Sept 2016

백상빈
@bsbin150
#javatshirt 이벤트 참여 완료 ~
27 Aug 2016

The Java certification recognition t-shirt is available to all candidates who earn a Java certification. Offer expires March 31, 2017.

ORACLE

CONOR **SVENSSON**

Blockchain: Using Cryptocurrency with Java

Integrating the Ethereum blockchain into Java apps using web3j

Hardly a day goes by without a mention of blockchain in the technology or financial press. But what's all the fuss about this technology, and how can you work with it from your Java applications? Before I talk about a library, web3j, that makes interaction possible, let me explain what blockchain is and how it works.

A Very Brief History

Blockchain technology started with the cryptocurrency Bitcoin. Bitcoin emerged in 2008, and although it was not the first proposed cryptocurrency, it was the first that was completely decentralized, requiring neither a central authority for issuance nor transaction verification. Bitcoin maintains a distributed ledger containing details of all Bitcoin transactions. All Bitcoin ownership is derived from these ledger entries, known as the Bitcoin blockchain. Transactions on the blockchain are generated using a simple scripting language.

Fast-forward five years later to 2013, when a 19-year-old named Vitalik Buterin started developing a new decentralized platform based on ideas from Bitcoin that provided a more-robust scripting language for the development of applications. That platform, named Ethereum, provided a full Turing-complete language. It was first proposed in early 2014, and it launched in July 2015.

Since then, several other blockchain technologies from different groups have emerged; however, Ethereum is presently the most mature (if you can call it that) by far, and it is the basis of this article.

The Ethereum blockchain is driven by the established cryptocurrency Ether. Ether is the second-largest cryptocurrency after Bitcoin with a market capitalization of approximately US$1 billion dollars, versus Bitcoin's US$10 billion capitalization.

Ethereum is intended to serve the back end of a secure, decentralized internet, where communication is with peers, and you no longer have to interact directly with single entities or organizations. web3j is a lightweight Java library for working with Ethereum, which I use in this article.

What Is a Blockchain?

A blockchain can be thought of as a decentralized, immutable data structure that undergoes state transitions that modify its state. State iterates with transactions or operations on the blockchain, the details of which are written to the blockchain. These transactions are grouped into blocks, which are linked together, making up the blockchain (see **Figure 1**). Much like the event log used in event sourcing, current state is derived by replaying the state transitions that have taken place previously.

A blockchain is a *distributed ledger technology* (DLT), a term that has emerged for describing technologies such as blockchain that provide decentralized storage of data. Not all DLTs use a common blockchain behind the scenes as in Ethereum; some keep data private between those parties entering into a transaction.

In Ethereum, data is written to, and subsequent state transitions take place within, the Ethereum Virtual Machine (EVM), which—like the Java Virtual Machine—interprets a bytecode format to execute the instructions in a transaction. But unlike the JVM, it uses a network of fully decentralized nodes. Each node executes these instructions locally, verifying that its local version of the blockchain matches those that have been written to the Ethereum blockchain previously. For the addition of a new block (containing new transactions) to be successful, certain nodes must reach a consensus with a majority of nodes on the network regarding what the new state should be. The nodes that create these new blocks are called *miners*.

Mining

The mining nodes use a consensus mechanism called *proof of work* (which incidentally is the same style of consensus mechanism Bitcoin uses). It relies on raw computing power to create new blocks on the blockchain. Blocks are created by continually hashing groups of new transactions that haven't been assigned to a block yet. When a combination of transactions is found that solves a predefined computational problem (referred to as *difficulty*), the miner that found the solution is rewarded by the network with newly minted currency, and that block is then added to the blockchain.

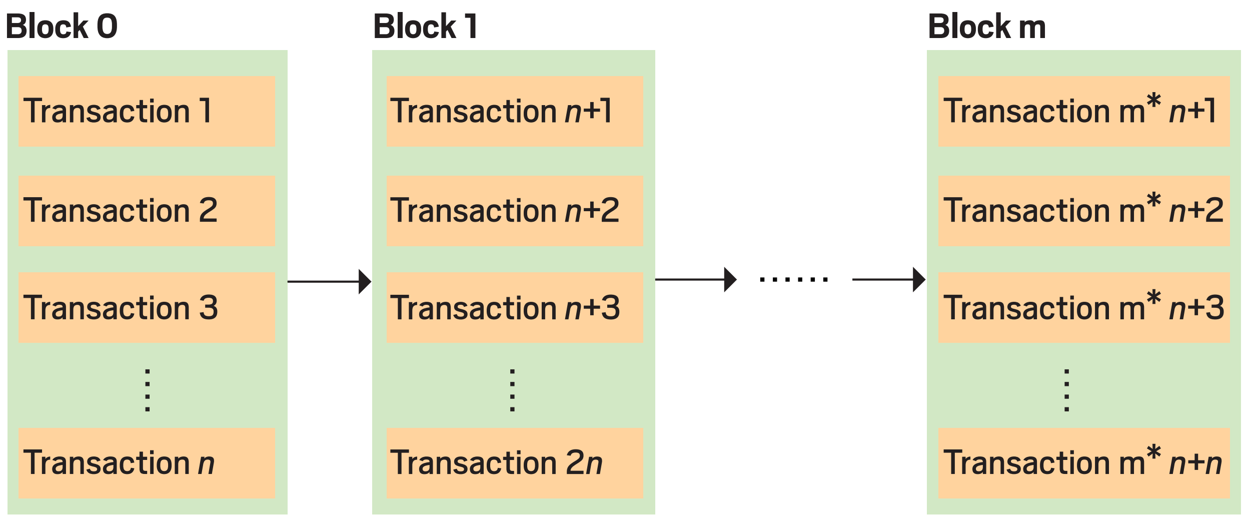


Figure 1. The structure of a blockchain

Smart Contracts

The programs that execute in the EVM are referred to as *smart contracts*. A smart contract is a computerized contract. In Ethereum, smart contracts are a collection of code (functions) and data (state) associated with an address in the Ethereum blockchain.

Getting Started with Ethereum

To start working with the Ethereum network, you need to have access to an Ethereum client (also called a *node* or *peer*). The Ethereum clients synchronize the Ethereum blockchain with one another, and they provide a gateway to interacting with the blockchain.

The two most widely used and complete Ethereum clients available are Geth and Parity.

Simply download one of the clients and then start it up:

```
$ geth --rpcapi personal,db,eth,net,web3 \
    --rpc --testnet

$ parity --chain testnet
```

This will start the client, and it will commence syncing the testnet Ethereum blockchain (which is multiple gigabytes in size). There are two public versions of the Ethereum network: mainnet and testnet (also known as Ropsten), reflecting the live and test environments, respectively. You are going to work with testnet in this article; otherwise, you'll need to start spending real money!

You will also need the latest available version of the web3j command-line tools. These tools provide several useful utility commands for working with Ethereum.

To transact on either the mainnet or testnet, you need to have some Ether cryptocurrency available.

The transaction process relies on public key cryptography, in which users have a public/private key pair. They use the private key (which only they know) to sign a transaction, which is published with the associated public key. The public key can then be used by other network participants to verify that the transaction is authentic, based on the transaction signature.

Fortunately for the users of Ethereum and web3j, the public key cryptography is abstracted away, so you need to work with only an *Ethereum wallet*, a digital file containing account credentials for transacting with Ethereum. A user can have any number of wallet files. It contains a password-encrypted private key and an address that is derived from the public key (note that the public key and address can both be derived from the private key). All transactions on the network are associated with such an address.

You can create a new secure wallet file for Ethereum using the web3j command-line tools, as follows:

```
$ ./web3j-1.0.7/bin/web3j wallet create


              _      _____ _     _
             | |    |____ (_)   (_)
__      _____| |__      / /_     _   ___
\ \ /\ / / _ \ '_ \     \ \ |   | | / _ \
 \ V  V /  __/ |_) |.___/ / | _ | || (_) |
  \_/\_/ \___|_.__/ \____/| |(_)|_| \___/
                         _/ |
                        |__/

Please enter a wallet file password:
Please re-enter the password:
Please enter a destination directory location
    [~/.ethereum/testnet-keystore]:
Wallet file <timestamp>--<UUID>.json created in:
~/.ethereum/testnet-keystore
```

Ensure that you select a secure password and that you don't forget it. There is no way of recovering your funds if you forget your password, or if someone steals your wallet file and figures out your password.

Once you have an address for your wallet, you can head over to Etherscan to view the details of the current balance and all transactions associated with this address. **Figure 2** shows a wallet address balance on Etherscan.

Testnet has been configured to allow easy mining of Ether, so you can start up a client in mining mode and in a matter of minutes have enough Ether to start creating your own transactions on the network.

Details are available for running a miner with Geth and with Parity.

Getting Started with web3j

With the Ethereum client running, you're now ready to write some Java code to connect to it using web3j. Note that complete listings of all the code are available on GitHub.

The Ethereum clients expose several methods over JSON-RPC, which is a stateless remote procedure call (RPC) protocol using JSON. web3j supports all of the Ethereum JSON-RPC API. However, there is a lot more plumbing behind the scenes that web3j takes care of than just providing JSON-RPC protocol wrappers. The JSON-RPC API is available over HTTP and interprocess communication (IPC), and there is a WebSocket implementation in Geth. However, the most common implementation employs HTTP, which I use in the following examples.

web3j releases are published to both Maven Central and JFrog's JCenter. At the time of this writing, the current release is 1.0.9; however, I'd recommend you check the web3j main project page to use the most current release in your project.

Add the relevant dependency to your build file:

Maven:

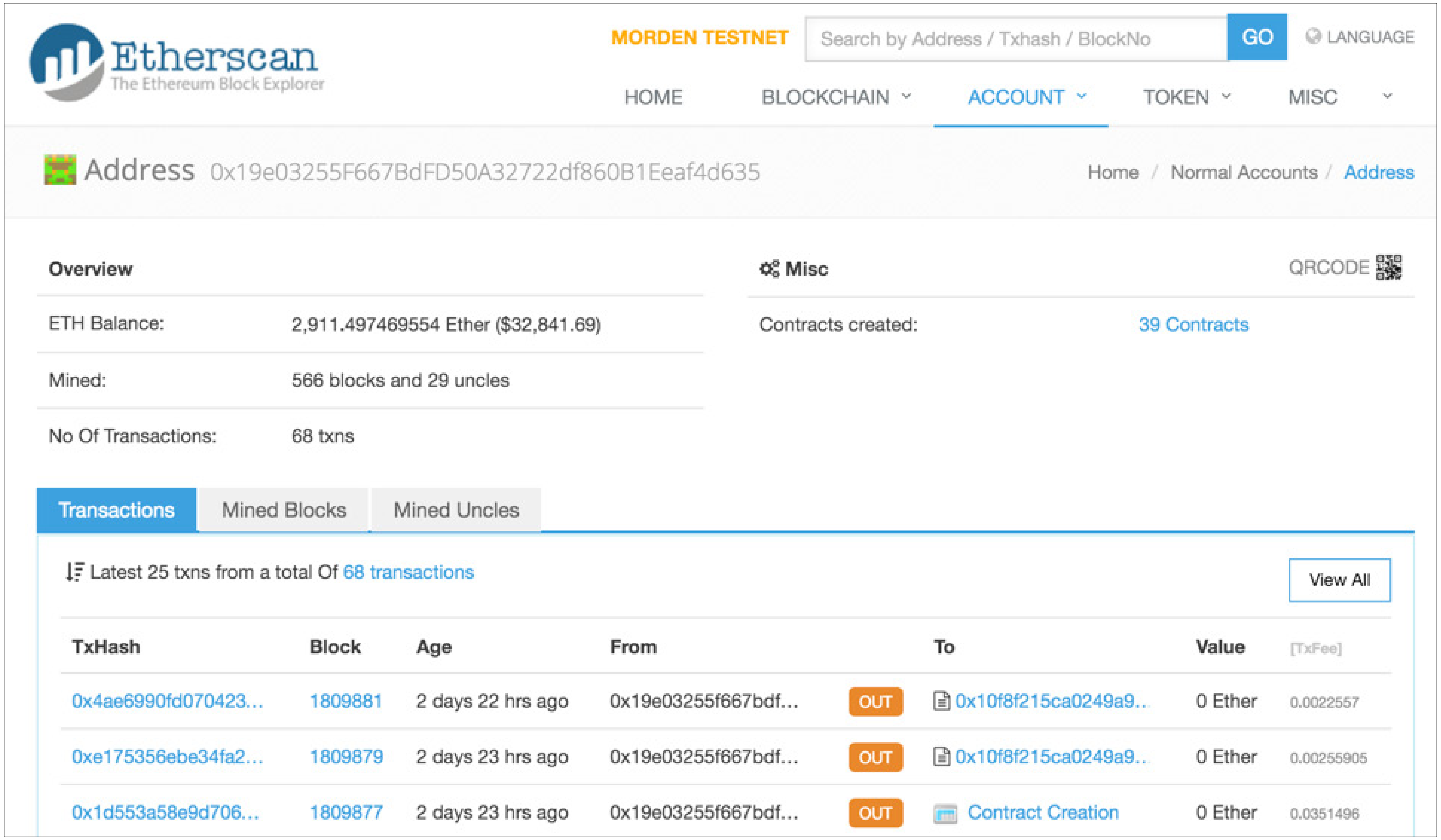
```
<dependency>
  <groupId>org.web3j</groupId>
  <artifactId>core</artifactId>
  <version>1.0.9</version>
</dependency>
```

Gradle:

```
compile ('org.web3j:core:1.0.9')
```

Place the following code in a runnable class, which displays the Ethereum client version:

```
// defaults to http://localhost:8545/
Web3j web3 = Web3j.build(new HttpService());
```


Etherscan
The Ethereum Block Explorer
MORDEN TESTNET
Search by Address / Txhash / BlockNo
GO
LANGUAGE
HOME
BLOCKCHAIN
ACCOUNT
TOKEN
MISC
Home / Normal Accounts / Address
Misc
QRCODE
Transactions
Mined Blocks
Mined Uncles
View All

Address 0x19e03255F667BdFD50A32722df860B1Eeaf4d635

Overview

ETH Balance:	2,911.497469554 Ether ($32,841.69)
Mined:	566 blocks and 29 uncles
No Of Transactions:	68 txns

Contracts created: 39 Contracts

Latest 25 txns from a total Of 68 transactions

TxHash	Block	Age	From		To	Value	[TxFee]
0x4ae6990fd070423...	1809881	2 days 22 hrs ago	0x19e03255f667bdf...	OUT	0x10f8f215ca0249a9..	0 Ether	0.0022557
0xe175356ebe34fa2...	1809879	2 days 23 hrs ago	0x19e03255f667bdf...	OUT	0x10f8f215ca0249a9..	0 Ether	0.00255905
0x1d553a58e9d706...	1809877	2 days 23 hrs ago	0x19e03255f667bdf...	OUT	Contract Creation	0 Ether	0.0351496

Figure 2. Ethereum wallet address balance on Etherscan

```
Web3ClientVersion clientVersion =
  web3.web3ClientVersion().sendAsync().get();

if (!clientVersion.hasError()) {
  System.out.println("Client is running version: " +
      clientVersion.getWeb3ClientVersion());
}
```

When you run this code, if all is well, you should see details of your Ethereum client version:

```
Client is running version:
    Geth/v1.5.4-stable-b70acf3c/darwin/go1.7.3
```

All JSON-RPC method implementations in web3j use the following structure, where `web3` is the instantiated web3j implementation (which manages the requests):

```
web3.<method name>([param1, …, paramN])
    .[send()|sendAsync()]
```

The method names and parameters are according to the JSON-RPC specification. The transmission of requests can be done either synchronously or asynchronously via the `send()` and `sendAsync()` methods, respectively.

Now that the connectivity to the Ethereum client has been verified, you're ready to start working with the Ethereum blockchain.

Transactions on the Ethereum Blockchain

To create a new transaction on the Ethereum blockchain, you typically perform one of three actions:

- Transferring Ether from one account to another
- Deploying a new smart contract
- Issuing a method call to an existing smart contract that modifies state

There is also a separate read-only method call to examine an existing smart contract, which does not create a transaction on the blockchain.

These transaction interactions require multiple underlying calls via JSON-RPC to Ethereum clients. web3j takes care of this lower-level functionality; however, all JSON-RPC calls are available to users in case they wish to have their own implementation.

Transferring Ether

I'll start by demonstrating the most basic transaction type: transferring 0.2 Ether from one account to another. Make sure that you know the location of the Ethereum wallet file you created earlier, because you won't be able to send any funds without it.

```
Web3j web3 = Web3j.build(new HttpService());

Credentials credentials =
  WalletUtils.loadCredentials(
    "my password", "/path/to/walletfile");

TransactionReceipt transactionReceipt =
  Transfer.sendFundsAsync(
      web3, credentials,
      "0x...", BigDecimal.valueOf(0.2),
      Convert.Unit.ETHER).get();

System.out.println(
  "Funds transfer completed, transaction hash: " +
  transactionReceipt.getTransactionHash() +
  ", block number: " +
  transactionReceipt.getBlockNumber());
```

This code produces the following output (the transaction hash has been trimmed and lines have been wrapped for legibility):

```
Funds transfer completed, transaction hash:
0x16e41aa9d97d1c3374a...34,
block number: 1840479
```

The transaction and block hashes are your identifiers for the transaction on the Ethereum blockchain. Behind the scenes, the actual transaction process, illustrated in Figure 3, is as follows:

1. The transfer Ether request is submitted to web3j.
2. A transaction message is submitted to an Ethereum client.
3. The client verifies the transaction, and then:
   a. Propagates the transaction on to other Ethereum nodes
   b. Takes a hash of the submitted transaction and sends this to the client in a synchronous HTTP response

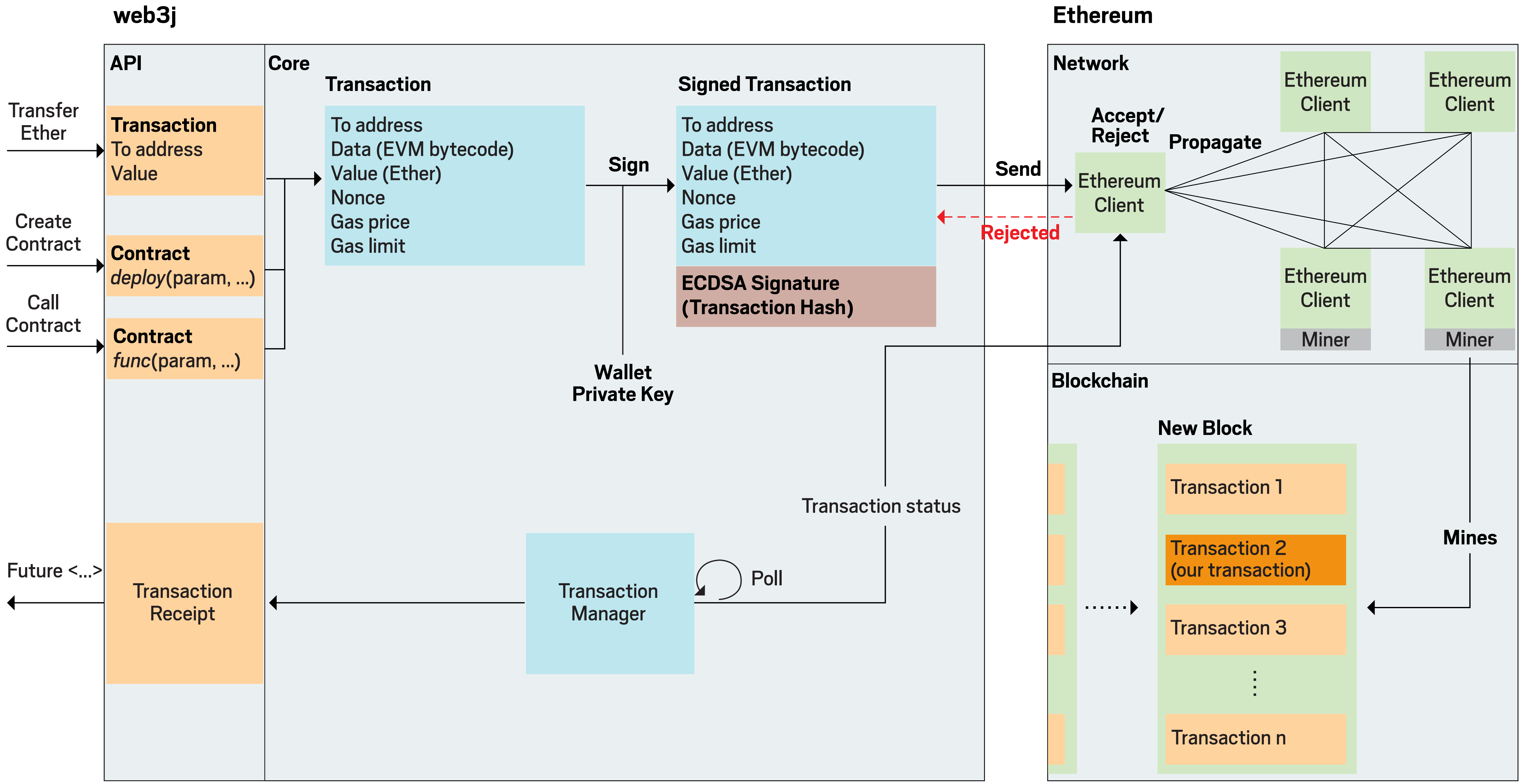
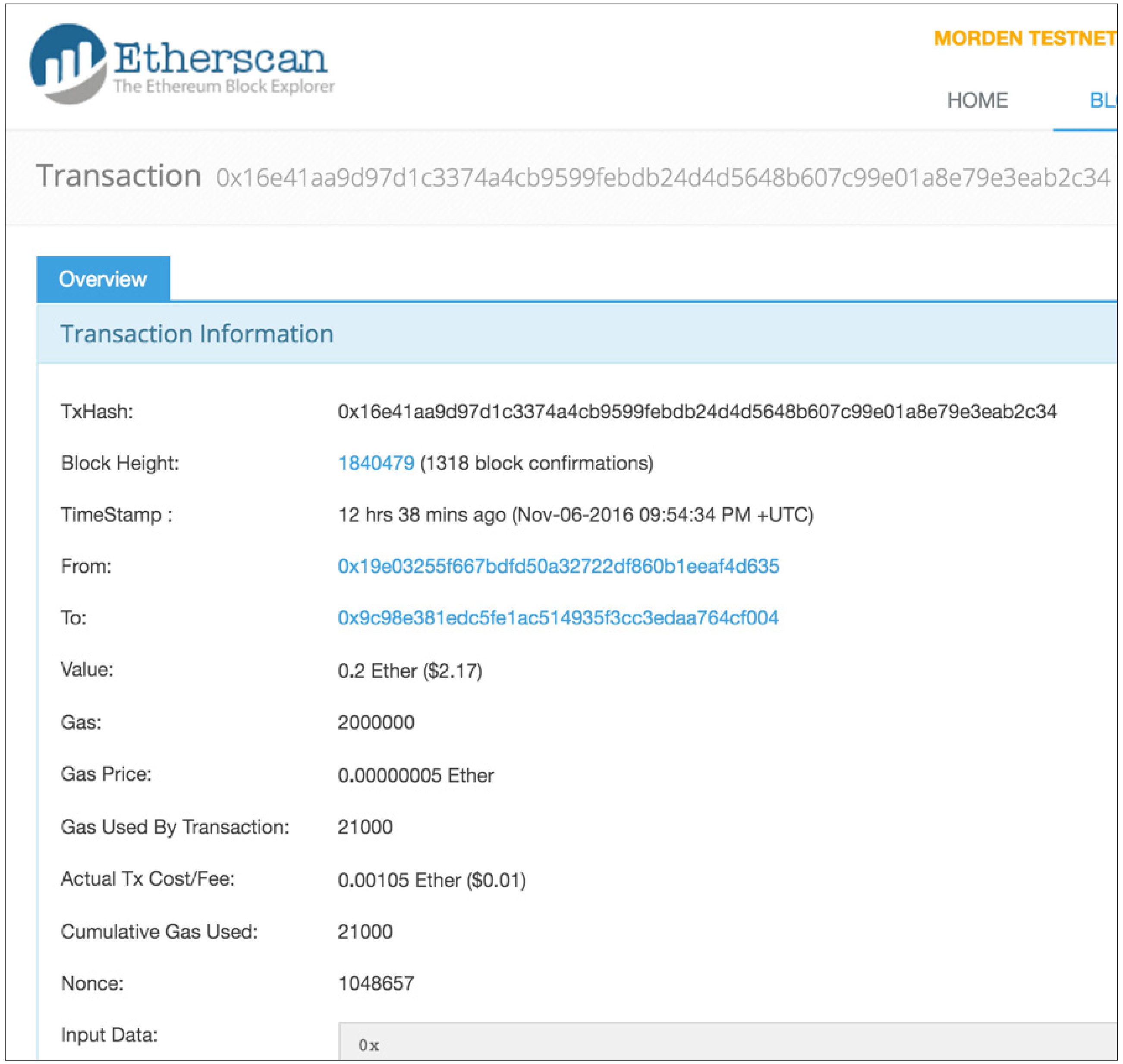

web3j
Ethereum
API
Core
Transfer Ether
Create Contract
Call Contract
Future <...>
Transaction
To address
Value
Contract
deploy(param, ...)
Contract
func(param, ...)
Transaction Receipt
Transaction
To address
Data (EVM bytecode)
Value (Ether)
Nonce
Gas price
Gas limit
Sign
Wallet
Private Key
Signed Transaction
To address
Data (EVM bytecode)
Value (Ether)
Nonce
Gas price
Gas limit
ECDSA Signature
(Transaction Hash)
Send
Rejected
Transaction Manager
Poll
Transaction status
Network
Accept/
Reject
Propagate
Ethereum Client
Ethereum Client
Ethereum Client
Ethereum Client
Ethereum Client
Miner
Miner
Blockchain
New Block
Transaction 1
Transaction 2
(our transaction)
Transaction 3
Transaction n
Mines

Figure 3. Transaction on Ethereum via web3j

4. This transaction is combined with other new transactions to form a block by miners on the Ethereum network. Once a valid block is formed, the block and details of its associated transactions are immortalized on the blockchain.

You can head back over to Etherscan to view the details of your transaction (**Figure 4**).

Figure 5 show the contents of the block on the blockchain in which the transaction resides.



Transaction 0x16e41aa9d97d1c3374a4cb9599febdb24d4d5648b607c99e01a8e79e3eab2c34

Transaction Information

TxHash:	0x16e41aa9d97d1c3374a4cb9599febdb24d4d5648b607c99e01a8e79e3eab2c34
Block Height:	1840479 (1318 block confirmations)
TimeStamp :	12 hrs 38 mins ago (Nov-06-2016 09:54:34 PM +UTC)
From:	0x19e03255f667bdfd50a32722df860b1eeaf4d635
To:	0x9c98e381edc5fe1ac514935f3cc3edaa764cf004
Value:	0.2 Ether ($2.17)
Gas:	2000000
Gas Price:	0.00000005 Ether
Gas Used By Transaction:	21000
Actual Tx Cost/Fee:	0.00105 Ether ($0.01)
Cumulative Gas Used:	21000
Nonce:	1048657
Input Data:	0x

Figure 4. A receipt of the transaction

Gas

I touched earlier in the article on Ether, which is used to pay for the execution of code in the EVM. There are two parameters that need to be specified with respect to the cost you are prepared to pay for transactions: the gas price and the gas limit. The gas price is the price in Ether you are prepared to pay per gas unit. Each EVM opcode contains a gas cost associated with it. The gas limit is the total amount of gas usage you are willing to pay for the transaction execution. This ensures that all transactions have a finite cost of execution. Details about the gas associated with a transaction are visible in the transaction receipt and Etherscan.

Hello (Ethereum) World

Now that I've demonstrated a simple transaction, things will start to get interesting as I create the first smart contract. Ethereum smart contracts are usually written in a language named Solidity, which is a statically typed high-level language. Describing how to use Solidity could fill many articles, so I'm going to keep the example simple. You can read more about Solidity online.

Now let's use the Greeter contract example. The Greeter contract is the "hello world" example of a smart contract of Ethereum. When you deploy the contract, you pass a UTF 8-encoded string to its constructor. Then, whenever you call the deployed contract, the value of this string is returned by the node on the Ethereum network that processes your request.

```
contract mortal {
    /* Define variable owner of the type address*/
    address owner;

    /* this function is executed at initialization
     and sets the owner of the contract */
    function mortal() { owner = msg.sender; }
    /* Function to recover the funds on the
    contract */
    function kill() {
        if (msg.sender == owner) suicide(owner);
    }
}

contract greeter is mortal {
    /* define variable greeting of the type string */
    string greeting;

    /* this runs when the contract is executed */
    function greeter(string _greeting) public {
        greeting = _greeting;
    }

    /* main function */
    function greet() constant returns (string) {
        return greeting;
    }
}
```

Although the language is unfamiliar, the example above is fairly intuitive to follow. The `mortal` contract is the base class, and the `greeter` contract is a child class.

The creator of the contract is the instance variable `owner`, of type `address`; `msg.sender` is the sender of any transactions with the contract; and the instance variable `greeting` is another instance variable.

There are two methods to be concerned with:

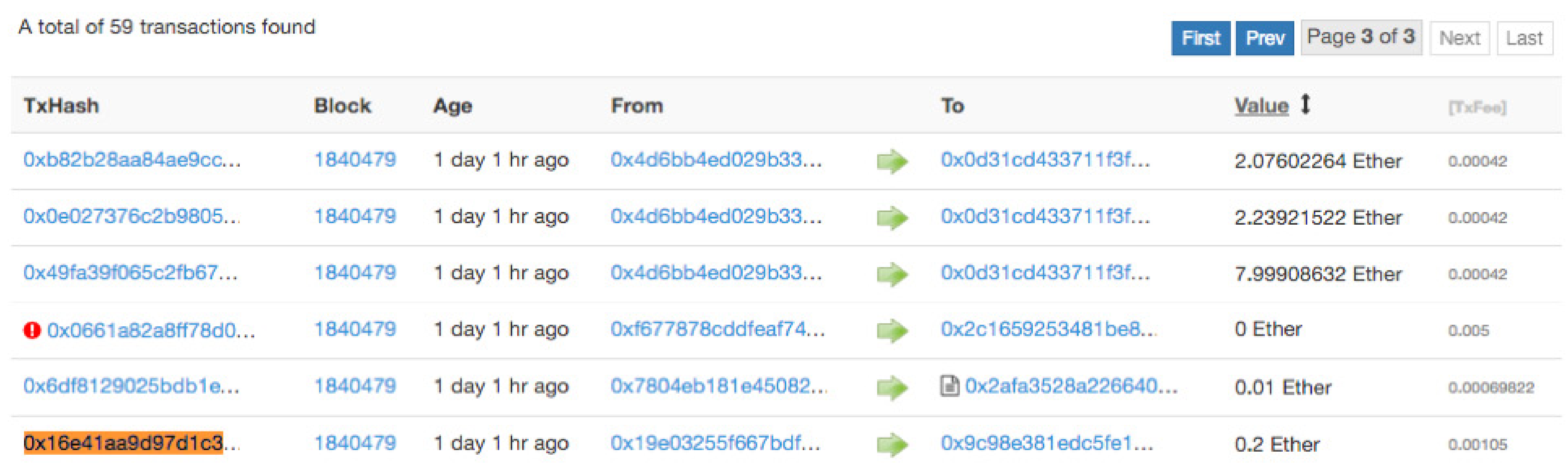
- `greeter(string _greeting)` is the constructor.
- `greet()` is a getter method returning the value of `greeter`.

There are two options for compiling the contract: either install the Solidity compiler or use the browser-based Solidity editor to do it online.

Once the compiler is installed, you can now compile the Greeter contract:

```
$ solc Greeter.sol --bin --abi --optimize -o build/
```

This action creates two files: a binary (.bin) file, which is the smart contract code in a format the EVM can interpret, and an application binary interface (.abi) file, which defines the



A total of 59 transactions found

TxHash	Block	Age	From		To	Value	[TxFee]
0xb82b28aa84ae9cc...	1840479	1 day 1 hr ago	0x4d6bb4ed029b33...	→	0x0d31cd433711f3f...	2.07602264 Ether	0.00042
0x0e027376c2b9805...	1840479	1 day 1 hr ago	0x4d6bb4ed029b33...	→	0x0d31cd433711f3f...	2.23921522 Ether	0.00042
0x49fa39f065c2fb67...	1840479	1 day 1 hr ago	0x4d6bb4ed029b33...	→	0x0d31cd433711f3f...	7.99908632 Ether	0.00042
! 0x0661a82a8ff78d0...	1840479	1 day 1 hr ago	0xf677878cddfeaf74...	→	0x2c1659253481be8...	0 Ether	0.005
0x6df8129025bdb1e...	1840479	1 day 1 hr ago	0x7804eb181e45082...	→	0x2afa3528a226640...	0.01 Ether	0.00069822
0x16e41aa9d97d1c3...	1840479	1 day 1 hr ago	0x19e03255f667bdf...	→	0x9c98e381edc5fe1...	0.2 Ether	0.00105

Figure 5. The block containing the transaction

public methods available on the smart contract.

web3j requires these two files in order to generate the smart contract wrappers for working with the smart contract in Java.

I generate the greeter wrappers using the web3j command-line tools, this time with the solidity command (full paths have been shortened for brevity):

```
$ ./web3j-1.0.9/bin/web3j solidity generate \
build/greeter.bin build/greeter.abi \
-p org.web3j.javamag.generated -o src/main/java/
```

This command will create the class file `org.web3j.example.generated.Greeter`, which wraps all the smart contracts' methods so they can be called from Java:

```
public final class Greeter extends Contract {
  private static final String BINARY =
      "6060604052604051610269380380610269833981...";

  private Greeter(
    String contractAddress, Web3j web3j,
    Credentials credentials,
    BigInteger gasPrice, BigInteger gasLimit) {
    super(
        contractAddress, web3j, credentials,
        gasPrice, gasLimit);
  }

...

  public Future<Utf8String> greet() {
      Function function = new Function("greet",
          Arrays.<Type>asList(),
          Arrays.<TypeReference<?>>asList(
              new TypeReference<Utf8String>() {}));
    return executeCallSingleValueReturnAsync(
        function);
  }

  public static Future<Greeter> deploy(
      Web3j web3j, Credentials credentials,
      BigInteger gasPrice, BigInteger gasLimit,
      BigInteger initialValue,
      Utf8String _greeting) {
    String encodedConstructor =
        FunctionEncoder.encodeConstructor(
            Arrays.<Type>asList(_greeting));
    return deployAsync(
        Greeter.class, web3j, credentials,
        gasPrice, gasLimit,
        BINARY, encodedConstructor, initialValue);
  }

  public static Greeter load(
      String contractAddress, Web3j web3j,
      Credentials credentials,
      BigInteger gasPrice, BigInteger gasLimit) {
    return new Greeter(
        contractAddress, web3j, credentials,
        gasPrice, gasLimit);
  }
}
```

I can now both deploy and call the smart contract:

```
Credentials credentials =
  WalletUtils.loadCredentials(
    "my password", "/path/to/walletfile");

Greeter contract = Greeter.deploy(
    web3, credentials, BigInteger.ZERO,
    new Utf8String("Hello blockchain world!"))
    .get();

Utf8String greeting = contract.greet().get();
System.out.println(greeting.getTypeAsString());
```

Running this code should produce the following output:

```
Hello blockchain world!
```

Now let's run through a more complex smart contract.

A Quick Note on Solidity Types

Solidity has several different native types. Although they are similar to some of those available in Java, web3j requires you to convert native Java types into Solidity types. This requirement is to guarantee the consistency of data being written to, or read from, the Ethereum blockchain.

It's also worth keeping in mind that the EVM uses unsigned 256-bit integers by default, which is why you'll find yourself working with `BigInteger` types with web3j.

Issuing Your Own Virtual Tokens

Smart contracts can be used to issue and manage token holdings, where a token is representative of proportional ownership tied to some real asset or even its own virtual currency. For instance, you could have a smart contract representing shared ownership of a property. Investors in this property could be provided with the number of tokens that represent their proportion of ownership of that property.

A standard contract API for tokens has been defined for Ethereum. It defines the following methods for interacting with the smart contract:

```
// total supply of tokens
function totalSupply() constant
    returns (uint256 totalSupply)

// tokens associated with address
function balanceOf(address _owner) constant
    returns (uint256 balance)

// transfer tokens from sender account to address
function transfer(address _to, uint256 _value)
    returns (bool success)

// approve spending for _spender of up to _value of
// your tokens
function approve(address _spender, uint256 _value)
    returns (bool success)

// delegated transfer of tokens by approved spender
function transferFrom(address _from, address _to,
    uint256 _value) returns (bool success)

// get the number of tokens the spender is still
// allowed to make
function allowance(address _owner, address _spender)
    constant returns (uint256 remaining)
```

This smart contract also introduces events, which are used in Ethereum to record specific details during smart contract execution on the blockchain. Because a transaction in Ethereum in a smart contract cannot return a value, these events enable you to query information from transactions that took place.

```
// notification of a transfer of tokens
event Transfer(address indexed _from,
    address indexed _to, uint256 _value)

// notification of an approval of delegated transfer
// of tokens
event Approval(address indexed _owner,
    address indexed _spender, uint256 _value)
```

Consensys has made available a full implementation of this smart contract, which you can download and then compile with the Solidity compiler:

```
$ solc HumanStandardToken.sol --bin --abi \
    --optimize -o build/
```

You can then generate smart contract wrappers for this smart contract, as shown next. (Again, full paths are trimmed for brevity.)

```
$ ./web3j-1.0.7/bin/web3j solidity generate \
build/HumanStandardToken.bin \
build/HumanStandardToken.abi \
-p org.web3j.example.generated -o src/main/java/
```

You're now ready to work with some of your very own tokens, and start issuing them:

```
// deploy your smart contract
HumanStandardToken contract = HumanStandardToken
    .deploy(
        web3, credentials, BigInteger.ZERO,
        new Uint256(BigInteger.valueOf(1000000)),
        new Utf8String("web3j tokens"),
        new Uint8(BigInteger.TEN),
        new Utf8String("w3j$"))
    .get();

// print the total supply issued
Uint256 totalSupply = contract.totalSupply().get();
System.out.println("Token supply issued: " +
    totalSupply.getValue());

// check your token balance
Uint256 balance = contract.balanceOf(
    new Address(credentials.getAddress()))
    .get();
System.out.println("Your current balance is: w3j$" +
    balance.getValue());

// transfer tokens to another address
TransactionReceipt transferReceipt =
    contract.transfer(
        new Address("0x<destination address"),
        new Uint256(BigInteger.valueOf(100))).get();
```

You can refer to this article's accompanying code for the full example.

Conclusion

In this article, I have scratched the surface of working with the Ethereum blockchain. There are many further details I have had to gloss over or omit entirely, but I hope this has given you some appreciation of what this fascinating technology is capable of.

For further information, you can refer to the web3j project source code and the documentation, which provides a lot more background information on Ethereum and web3j than I can fit into a single article. `</article>`

Conor Svensson (@conors10) is the author of web3j, the Java library for integrating applications with the Ethereum blockchain. He previously cofounded the startups coHome and Huffle. He is currently helping Othera build its blockchain lending platform and exchange. He blogs about technology and finance. When not in front of a screen, Svensson likes to make the most of surfing at his local beach, Maroubra, in Sydney, Australia.

ADRIAAN **MOORS**

Scala: Deeply Functional, Purely Object-Oriented

A mature, practical, and type-safe language for the JVM

Scala is a pragmatic, object-oriented JVM language that integrates a comprehensive set of functional programming (FP) features, such as pattern matching and immutable collections, using a compact syntax with powerful type inference. It has gained widespread adoption in many different industries, with some codebases in the millions of lines. Thanks to Apache Spark, Scala is also widely used in the data science community.

Interoperability with Java is a core feature of Scala, and with version 2.12, the language now includes the features of the Java 8 platform. For example, Scala 2.12 compiles lambdas in the same way as Java 8. Similarly, a Scala trait compiles to a Java interface with default methods.

Scala's Origins

Scala was originally developed by Martin Odersky and his research group at the École Polytechnique Fédérale de Lausanne (EPFL), a leading Swiss technical university. Before designing Scala, Odersky codesigned Generic Java (GJ), which brought generics from FP to Java, and he developed a compiler for it. Sun Microsystems adopted the GJ compiler as the standard javac compiler from version 1.3 on (with its support for generics enabled in Java 5).

Meanwhile, Odersky and his research group started work on Scala, releasing version 2.0 in 2006. Around 2007, prominent internet startups such as Twitter and Foursquare started to use Scala in production, and adoption has grown ever since. In 2011, the company Typesafe was founded to drive commercial adoption of Scala as the language for multicore and cloud computing, using the Akka middleware framework.

Today, Typesafe is known as Lightbend. My team at Lightbend continues to develop the Scala compiler and library in collaboration with an active open source community. A third of the commits in the latest release came from the community. Other vendors are investing in Scala as well. For example, JetBrains' Scala plugin for IntelliJ IDEA has been downloaded almost 5 million times.

The Scala Philosophy

In this article, I explain some of Scala's core concepts and illustrate them with small code snippets. I hope to convince you that, while some developers consider Scala a big language, it actually has a small purely object-oriented core so flexible that it can take many different shapes.

The key idea behind Scala's design is that functional and object-oriented programming are complementary and, thus, combine well into a single language. While FP is often associated with complicated types and deep math, the true appeal of FP in Scala is that it streamlines implementing common tasks in a way that is easier to understand and to scale up. The popularity of lambdas in Java 8 is a good example of this.

Scala aims to be regular and concise, with few restrictions or exceptions. In a definition, all redundant parts are optional.

Functional thinking and design are ingrained in Scala's DNA, from the choice of keywords (`val` for immutable and `var` for mutable variables), having type ascriptions follow rather than precede the variable name (so they can be omitted easily, thanks to type inference), and pattern matching (deeply integrated into the object-oriented core) to preferring definition-site variance over wildcards (the latter tends to clutter users' code).

In Scala, a *function* is a small unit of abstraction, well suited to capture a single computation where the result is solely determined by a few inputs. A function is easy to understand in isolation, which also means it can more readily be distributed across many cores or machines. In FP, functions play the role that objects have in object-oriented programming: they are first-class values and the key building blocks that determine a program's behavior. Objects and classes represent larger abstractions that encapsulate the interplay of many different members. Where a composition of functions expresses an algorithm, a composition of objects models a system.

Scala unifies the safety and performance of static type checking with the ease of use of a scripting language through type inference. Because of type inference, most type annotations can be omitted when expedient. Types can often be omitted when they impede readability, where they are a barrier to potential refactorings, to the rapid development of a prototype, or to the exploration of a new problem in Scala's interactive shell or a data science worksheet.

Many programmers think of FP as disavowing mutable state (or, more generally, side effects), requiring lazy evaluation or deep mathematical reasoning. This is not how functional programming is interpreted in Scala. Eventually, code must have an effect on the outside world, whether that code is functional or object-oriented. However, unneeded or unexpected side effects usually adversely affect maintainability and scalability—in *both* functional and object-oriented programming. Scala gives you a choice between "pure" programming, which favors immutable variables and recursion, and more-imperative programming, which uses mutable variables and `while` loops.

However, Scala does try to nudge you away from side effects. There is no separate notion of a statement, because it is considered an expression that computes an uninteresting result. An expression of type `Unit`, the equivalent of `void`, is just another type, which is integrated with generics. It just happens to have only one value—the uninteresting result—which is usually discarded. This unification of expressions and statements simplifies a few things: functions need not distinguish whether their body is a statement (that is, returning `void`) or an expression.

Ultimately, Scala is a purely object-oriented language. This means that, conceptually, every value in Scala is an object, and every operation is a method call. Other high-level "conveniences" of the language, such as functions, are translated into this core by expanding them to method calls behind the scenes.

This capability makes it easy to write domain-specific languages (DSLs) in Scala. The use of such DSLs has been implemented successfully by Apache Spark (for data science), Slick (for database queries), and the sbt build tool (for declarative build definitions).

One aspect of Scala's deep integration of FP is simply to translate a function application f(x) to the method call `f.apply(x)`. A function is modeled as an object with an `apply` method, and any object that has this method is, thus, eligible for this convenience.

The very same idea is used to translate Scala's more-advanced features to its object-oriented core. Code using `for`, such as `for (x <- coll) println(x)`, is shorthand for `coll.foreach(x => println(x))`. This expansion generalizes to more-complicated comprehensions, and it applies equally to collection types defined outside the standard library, so that it can be used, for example, to express SQL queries or big data analytics.

One of my favorite translations is for interpolated strings such as `s"Hello $name"`. It expands to the ordinary method call `StringContext("Hello ", "").s(name)`. This provides two hooks to adapt its behavior to your domain: you could bring your own `StringContext` class into scope to override the standard interpolators, or you could use Scala's mechanisms for retroactive extension to enrich the existing `StringContext` with a `sql` method to support `sql"SELECT $col FROM $table"`.

Scala aims to be regular and concise, with few restrictions or exceptions. In a definition, all redundant parts are optional—meaning the signature of a method may be omitted when it can be inferred—and a method is declared `abstract` simply by omitting its body. Public visibility is the default, and definitions can be nested arbitrarily (a method can be local to another method).

To allow user-defined classes to blend in naturally with primitive types, an identifier can be symbolic, and punctuation is optional in a method invocation. This means that there is no conceptual difference between `x + y` and `x.+(y)`. They are just invocations of the method called `+`. Although values of primitive type are considered objects, they compile to efficient bytecode behind the scenes. To further reduce punctuation, the semicolon is optional in separating expressions when it can be inferred from a line break.

In keeping with the mantra that every method invocation targets an object, Scala avoids the static keyword, and instead offers direct support for the singleton design pattern through object definitions.

Some Examples

I think it's time to make these ideas more concrete. In the examples below, what follows the `scala>` prompt is input to the Scala shell (launched using the `scala` command), and other lines are feedback from the shell or output from the program.

I start by defining a class `C` with a single method `message`:

```
scala> class C { def message = "hello" }
defined class C
```

The compiler infers `message's` result type. For a public method, it's usually a good idea to specify the result type:

```
scala> class C { def message: String = "hello" }
defined class C
```

To harmonize definitions with and without result types, a method's result type comes last. More generally, type ascriptions always follow the entity they describe.

Of course, you shouldn't hardcode the class's message. For the common case where a constructor immediately stores each of its arguments in a field for later use, a class' signature may define a list of arguments, prefixing them with the desired keyword for the corresponding member (`val` or `var`):

```
scala> class C(val message: String)
defined class C

scala> new C("Hi!").message
res1: String = Hi!
```

Following a unification proposed in the Eiffel programming language as the uniform access principle, users of the improved class will not notice the change from `def` message to `val` message. You could even transparently opt for a lazy `val`, which is computed on its first access and then cached. This also generalizes to mutable fields, which are modeled as a getter and setter.

When processing data, it's often convenient to bundle a few values together in a tuple. Because tuples are so common, Scala ships with a set of TupleN case classes with N fields.

The following snippet declares an abstract, mutable variable in a trait, deferring its implementation in terms of its getter and setter to a subclass (perhaps adding validation). I've omitted the implementation using the `???` method, which is defined in the standard library to throw a `NotImplementedError`.

```
trait T {
  var v: Int
}

class Sub extends T {
  def v: Int = ???
  def v_=(x: Int): Unit = ???
}
```

A trait may also define a concrete `val` or `var` member, which the compiler implements automatically in the trait's subclasses.

In keeping with the mantra that every method invocation targets an object, Scala avoids the `static` keyword, and instead offers direct support for the singleton design pattern through object definitions. It's common practice to define a class and an object of the same name at the same time, and what would be static methods in Java become methods on the class' so-called "companion object."

Here, I define a singleton instance of the C class, instead of creating a new instance explicitly:

```
scala> object o extends C("Hi!")
defined object o

scala> println(o.message)
Hi!
```

Because it doesn't introduce new members, the object definition corresponds closely to lazy `val o = new C("Hi!")`.

Perhaps you've spotted another aspect of Scala's uniformity: definitions are always introduced by their keyword (one of `val`, `var`, `def`, `object`, `trait`, `class`, or `type`), possibly prefixed by some modifiers (such as `lazy` or `private`). Then come the name and the signature, and finally is the right-hand side.

Case Classes and Pattern Matching

As another example of how Scala nudges you toward immutable design, it's very convenient to immediately store constructor arguments in an immutable `val` (or, if you prefer, in a `var`). To make this even more convenient, both `val` and `new` keywords can be omitted when defining a case class:

```
scala> case class C(x: Int)
defined class C
```

Case classes model immutable structured data. They are called *case classes* because they are so easy to use with the `case` keyword when pattern matching, as in this example:

```
scala> val y = C(1) match { case C(x) => x }
y: Int = 1
```

The expression in `y`’s right-hand side makes a new object, `C(1)`, and immediately picks it apart by comparing it against the pattern `C(x)` using pattern matching. A pattern can be thought of as a value with “holes” that are bound to variables so as to extract their values. In this case, the pattern variable x is assigned to 1, the value passed as `C`’s first constructor argument, and it is then used as the result of the match expression. Patterns can be nested to reach directly into the desired part of a data structure. In general, the compiler ensures that you have covered all possible cases and that all of them are reachable.

When processing data, it’s often convenient to bundle a few values together in a tuple. Because tuples are so common, Scala ships with a set of TupleN case classes with N fields, for N ranging from 1 to 22. So, a tuple behaves just like a regular case class except that its name need not be used when calling its constructor. For example, `Tuple2("a", 1)` can be abbreviated to `("a", 1)`. It’s useful to think of `Unit` as Tuple0, with its sole value written as `()`.

A pattern match with one case, such as the one above, can be shortened to `val C(y) = C(1)` to introduce the variable `y` and bind it to the value of `C`’s first argument, as explained previously. This is especially useful when dealing in tuples, because it provides a convenient way to return multiple values from a method and immediately name them. Pattern matching, and tuples in particular, help shift coding patterns away from storing intermediate results in (mutable) local variables and toward computations that can be run efficiently on massive data sets using Apache Spark or Apache Flink, for example.

The easiest way to get started with Scala is to install an IDE with Scala support (IntelliJ IDEA or Scala IDE for Eclipse).

Conclusion

The easiest way to get started with Scala is to install an IDE with Scala support (IntelliJ IDEA or Scala IDE for Eclipse). These Scala IDEs provide an interactive worksheet, the equivalent of a read-eval-print loop (REPL). On the command line, you may either install a Scala distribution and run the scala command or install the sbt Scala build tool sbt.

To quickly create an empty Scala project with sbt, execute the following at the command prompt: `sbt new scala/scala-seed.g8`. After launching sbt from the directory created for your new project, use the console task to launch the interactive shell and try out the examples in this article.

Learn More

To dig deeper, there are some great books on Scala (such as *Scala for the Impatient*, *Programming Scala*, and *Programming in Scala*), several free courses on Coursera (the first one is “Functional Programming Principles in Scala”), and many more online resources (such as Twitter’s Scala School). Have fun! `</article>`

Adriaan Moors (@adriaanm) leads the Scala team at Lightbend. His first contribution to the Scala compiler, in 2007, was support for type constructor polymorphism, which was also the topic of his PhD dissertation. After a post-doc in Martin Odersky’s lab at the EPFL, he joined Typesafe (now Lightbend) in 2012. Over the years, he has worked on all aspects of the language, from the underlying theory (the Dependent Object Types calculus) and the compiler implementation, down to the continuous integration and release infrastructure.

HARSHAD **OAK**

Java in Containers in the Cloud

Deploy Java apps in Docker containers using Oracle Application Container Cloud Service.

There has been tremendous interest in application containers during the past couple of years. Application containerization comes with the promise of "build once, run anywhere" in an isolated environment that consumes significantly fewer resources than a virtual machine.

Given the widespread interest and growing adoption, many vendors are looking at offering container solutions. Oracle Application Container Cloud Service leverages Docker and provides a service that can run your Java, PHP, and Node.js applications. In this article, I look at the Java and Java EE capabilities of Oracle Application Container Cloud Service. To follow along, you'll need basic knowledge about Maven, Java EE, and containerization.

Containers for Java EE

Before diving into Oracle Application Container Cloud Service, let's quickly go over some things to consider if you're looking to switch to containers from established Java EE deployment servers.

First, it's important to consider the ease of use of Java EE today and the smart, self-contained, and out-of-the-box nature of modern Java EE tools and servers.

Containerized applications are lightweight because they ship with only what they need and nothing else. This means that you need to know exactly which resources, dependencies, and versions to include, which can take some getting used to, especially for Java EE developers accustomed to using application servers that provide everything an application might require. With most current Java EE applications, you provide your application code and the server pretty much provides and figures out most other things, using a mix of conventions and configuration in your application.

Containerized applications by their very nature seem suited to public cloud deployments. However, even for traditional Java EE deployments, there are many platform-as-a-service (PaaS) cloud solutions that do a great job of running your Java EE in the cloud. In most cases, you don't need to worry about the underlying hardware, resources, or servers. I discussed these deployments in previous articles in *Java Magazine*, most recently "Getting Onboard Oracle Java Cloud Service," where I showed how to use Oracle Java Cloud Service for Java EE applications.

There's also a lot of expertise and tooling in place for the established Java EE server-based approach. The container market is still in a state of flux.

Once you're sure that containers are the way to go for your application, you can decide on the kind of container solution to adopt. Let's take a closer look at Oracle's solution.

About Oracle Application Container Cloud Service

Oracle Application Container Cloud Service is a simple solution that runs your application in its own Docker container in the Oracle Cloud. It currently supports Java and JVM languages, PHP, and Node.js. It promises fast, easy-to-use, self-service provisioning of your applications in isolated container environments in the Oracle Cloud.

The service provides integration with other cloud services offered by the company, including Oracle Database

Cloud Service and Oracle Java Cloud Service. It also offers continuous integration and development if integrated with Oracle Developer Cloud Service.

Although applications in Oracle Application Container Cloud Service run in a Docker container based on Oracle Linux, the service is intended for you to run your applications—not for you to run your Docker containers.

A key benefit of Oracle Application Container Cloud Service over running Docker containers yourself is that for your Java applications, the service comes set up with Oracle Java SE Advanced, which includes Flight Recorder diagnostics and profiling capabilities. Oracle will take care of Java patching and upgrades. The service also handles load balancing and scaling your application. By using it, you get to operate at a higher level of abstraction and don't have to worry about the underlying Java setup, the Oracle Linux base, or the Docker container.

How to Deploy to Oracle Application Container Cloud Service

To use Oracle Application Container Cloud Service, you don't need to code your application any differently. Understanding the service is mostly about packaging, deploying, and managing your application.

Instead of spending time on elaborate code snippets in this article, I'll pick up one of the sample applications in the service documentation and see how you can go about tweaking it a bit and then packaging and running it with Oracle Application Container Cloud Service. I use NetBeans to make the application easier to understand.

Let's begin by downloading the code for the "Java SE 8: Creating a Web App with Bootstrap and Tomcat Embedded for Oracle Application Container Cloud Service" tutorial. It's a simple standalone web application that uses servlets, JavaServer Pages (JSPs), the Bootstrap front-end framework, and an embedded version of Tomcat.

If you're a Java EE developer used to deploying WAR and EAR files on application servers, you might wonder why we need an embedded Tomcat server. It's worth pointing out here that the Oracle Application Container Cloud Service environment has Java running on it, but there is no server there to run your Java EE application. It's up to you to package the server as part of your application.

Unzip the downloaded code and you will find a Maven project and the requisite project files. I use NetBeans to better understand the project and its constituent parts, especially how the project is configured for deployment on Oracle Application Container Cloud Service. Most IDEs support Maven projects, so you could use some other IDE. If you prefer, you could even use Maven commands from the command prompt.

Using NetBeans and Maven

Open the NetBeans IDE and select File -> Open Project from the menu. Next, select the directory where you extracted the code. NetBeans will figure out that it's a Maven project and handle it accordingly.

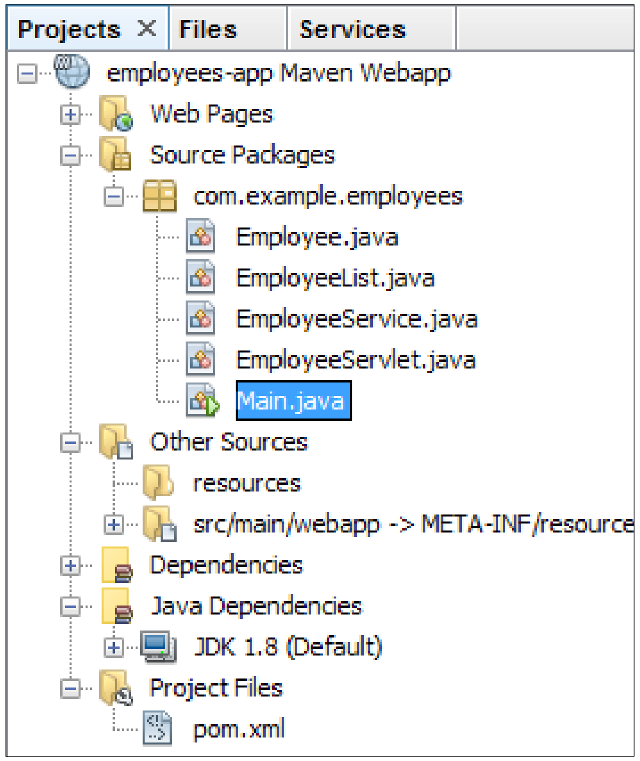


Figure 1. NetBeans Project view

As shown in **Figure 1**, you have web pages, a few Java classes, dependencies, and the Maven pom.xml file. The project object model contains project information and the configuration that is used by Maven to build the project. Open the pom.xml file in NetBeans, which offers a Graph view of the file that can help better show the project dependencies.

Now open the Source view and you will see the XML. Let's start from the top and look at some of the key tags for our project.

As shown in **Listing 1**, the Tomcat version is specified as 7.0.57. Tomcat provides different versions based on which specifications are supported. While the creators of the sample application have used version 7.0.57, you can use the latest release of version 7. However, running the application with Tomcat 8.x or 9.x will require additional tweaks.

Listing 1.

```
<properties>
    <tomcat.version>7.0.57</tomcat.version>
</properties>
```

The dependencies are shown in **Listing 2**. Except for JSP Standard Tag Library, they're derived by Maven from the dependency on the Tomcat version specified in **Listing 1**.

Listing 2.

```
<dependencies>
    <dependency>
        <groupId>org.apache.tomcat.embed</groupId>
        <artifactId>tomcat-embed-core</artifactId>
        <version>${tomcat.version}</version>
    </dependency>
      . . .
    <dependency>
        <groupId>org.apache.tomcat</groupId>
        <artifactId>tomcat-jsp-api</artifactId>
        <version>${tomcat.version}</version>
    </dependency>
    <dependency>
        <groupId>jstl</groupId>
        <artifactId>jstl</artifactId>
        <version>1.2</version>
    </dependency>
</dependencies>
```

Further down the file, as shown in **Listing 3**, are the tags for the Maven Assembly Plugin.

Listing 3.

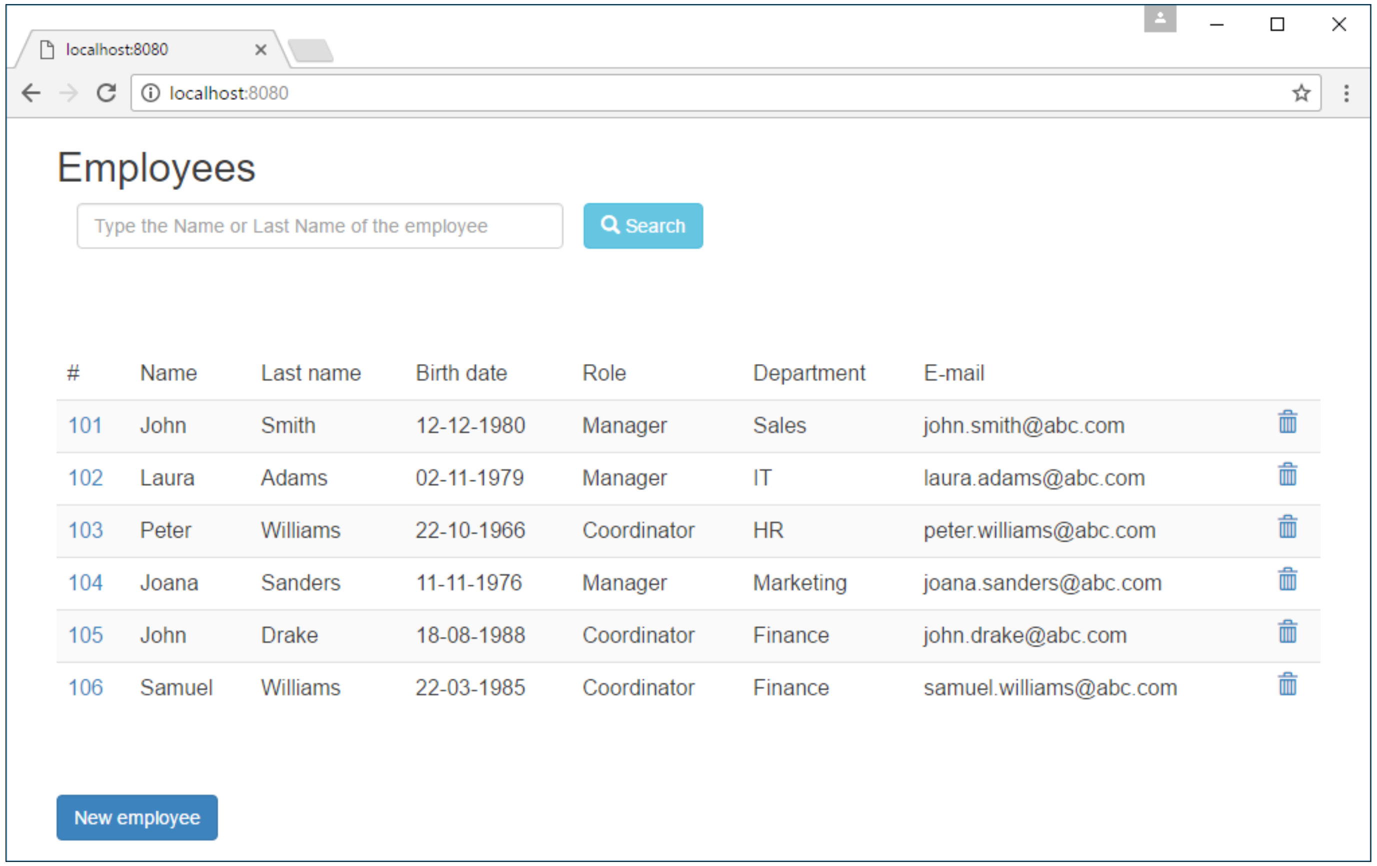
```
<configuration>
    <descriptorRefs>
      <descriptorRef>
        jar-with-dependencies
      </descriptorRef>
    </descriptorRefs>
    <finalName>
      ${project.build.finalName}-${project.version}
    </finalName>
    <archive>
      <manifest>
        <mainClass>
          com.example.employees.Main
        </mainClass>
      </manifest>
    </archive>
</configuration>
```

The `mainClass` here is specified as `com.example.employees.Main`. This declaration is important because this is the class that will be executed when the packaged JAR file is run using the `java -jar` command. So let's look at this `mainClass` next, in **Listing 4**.

Listing 4.

```
public class Main {
    public static final Optional<String> PORT =
  Optional.ofNullable(System.getenv("PORT"));
public static final Optional<String> HOSTNAME =
  Optional.ofNullable(System.getenv("HOSTNAME"));

public static void main(String[] args)
  throws Exception {
    String contextPath = "/" ;
    String appBase = ".";
    Tomcat tomcat = new Tomcat();
    tomcat.setPort(Integer.valueOf(
        PORT.orElse("8080") ));
    tomcat.setHostname(
        HOSTNAME.orElse("localhost"));
    tomcat.getHost().setAppBase(appBase);
    tomcat.addWebapp(contextPath, appBase);
    tomcat.start();
        tomcat.getServer().await();
    }
}
```



Employees

Type the Name or Last Name of the employee

Search

#	Name	Last name	Birth date	Role	Department	E-mail
101	John	Smith	12-12-1980	Manager	Sales	john.smith@abc.com
102	Laura	Adams	02-11-1979	Manager	IT	laura.adams@abc.com
103	Peter	Williams	22-10-1966	Coordinator	HR	peter.williams@abc.com
104	Joana	Sanders	11-11-1976	Manager	Marketing	joana.sanders@abc.com
105	John	Drake	18-08-1988	Coordinator	Finance	john.drake@abc.com
106	Samuel	Williams	22-03-1985	Coordinator	Finance	samuel.williams@abc.com

New employee

Figure 2. Running the web application

This code picks up the port and host name from the environment and starts the Tomcat server.

The rest of the code in the project is all about building the actual web application and not directly important for executing in the Oracle Application Container Cloud.

Build the Application

Before you can deploy to Oracle Application Container Cloud Service, the application must be built. Right-click the project name and click Build. You will see in the log that Maven gets the requisite JAR files, copies, compiles, and packages the application such that the executable JAR file is created in the target directory. This file is what is often referred to as a *fat JAR* or an *uber JAR*, because this JAR file includes your code as well as the dependent libraries. The generated JAR includes a manifest.mf file, which specifies `com.example.employees.Main` as the main class to run the JAR file.

If you run this JAR file on your local machine using the `java -jar employees-app-1.0-SNAPSHOT-jar-with-dependencies.jar` command, you will start a Tomcat server on your local machine with your application running on it. Go to `http://localhost:8080` in your browser, and you will get a page as shown in **Figure 2**.

Because our intent is to run the application on Oracle Application Container Cloud Service, there is still some work to be done.

manifest.json Configuration

To upload and deploy the application to the cloud service, you must include a manifest.json file in the application archive. The archive could be a .zip, .tgz, .tar, or .gz file. In the manifest .json file, you need to state the Java version to use and the file to run to start the application. The manifest can contain some other configuration values, but those are optional. The manifest.json file is shown in **Listing 5**.

Listing 5.

```
{
    "runtime": {
        "majorVersion": "8"
    },
    "command":
        "java -jar employees-app-1.0-
         SNAPSHOT-jar-with-dependencies.jar"
}
```

[The last line is wrapped here due to width constraints.—*Ed.*] Next, package the manifest.json file and the employees-app-1.0-SNAPSHOT-jar-with-dependencies.jar file into a zip file. You're now set to deploy the application to Oracle Application Container Cloud Service.

Deployment

Once you log in to Oracle Application Container Cloud Service, you will see a dashboard screen with a button on the right side with the text "Create Application." Click on the button, and Oracle Application Container Cloud Service will ask you for your application platform. Click on Java SE. Next, you will get a screen as shown in **Figure 3**.

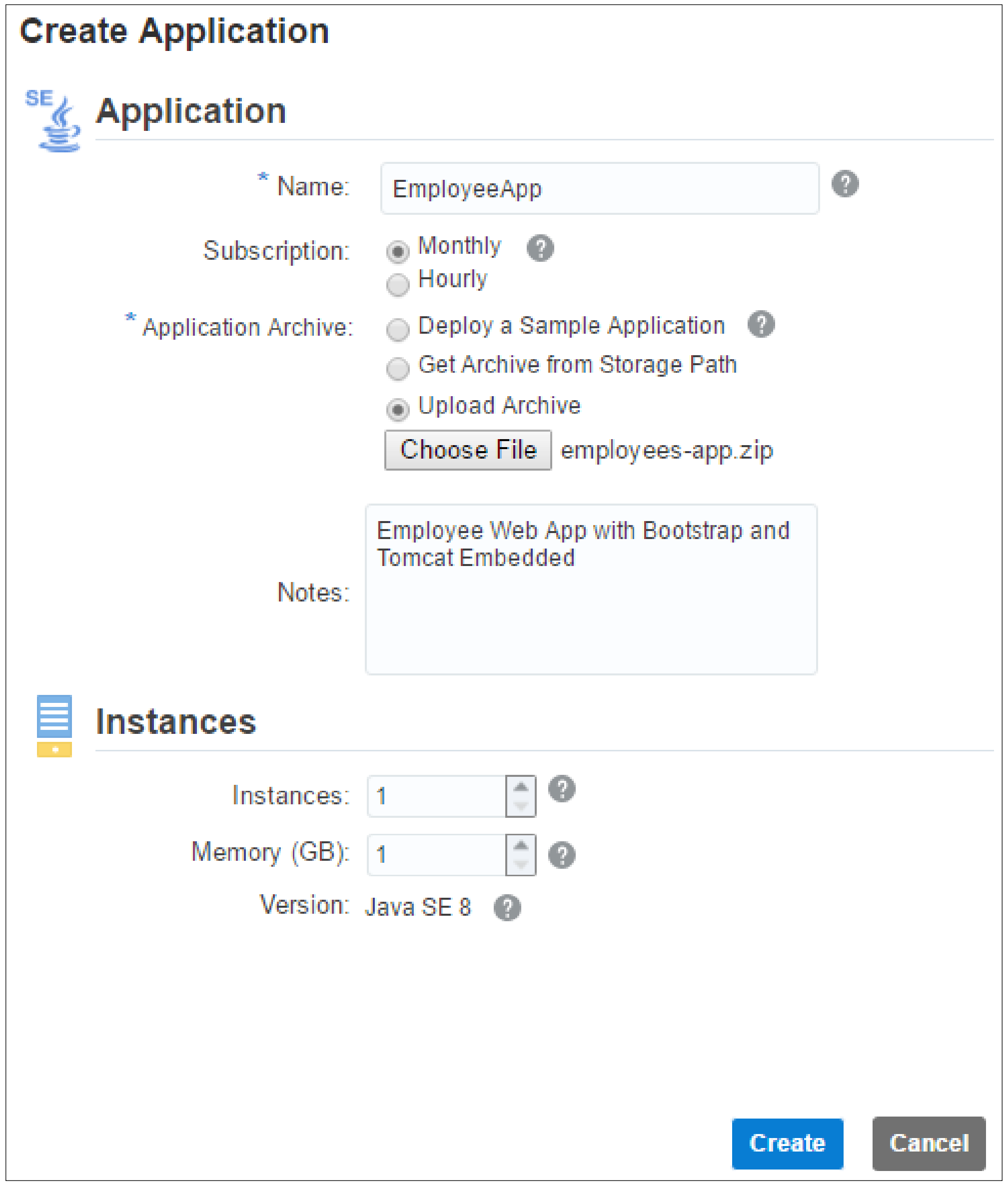
Although it's a fairly simple screen, it presents most of the key configuration options that are offered by Oracle Application Container Cloud Service. You can select the subscription model, and you can either upload the application archive or specify the link to Oracle Storage Cloud. You can also select the number of instances of your application as well as the memory to be allocated to each such instance.

Name the application and upload the zip file created earlier, which includes the JAR file and the manifest.json file.

Use the default settings for Instances, and specify that you want a single instance of the application with 1 GB of memory allocated to it.

You should now get a Processing Archive message on the screen, followed by the applications page, as shown in **Figure 4**.

The application creation can take a few minutes. You can follow the current status either in the Activity section in **Figure 4** or on the main cloud service dashboard.



Create Application

Application

* Name: EmployeeApp

Subscription: ● Monthly ○ Hourly

* Application Archive: ○ Deploy a Sample Application ○ Get Archive from Storage Path ● Upload Archive

Choose File employees-app.zip

Notes: Employee Web App with Bootstrap and Tomcat Embedded

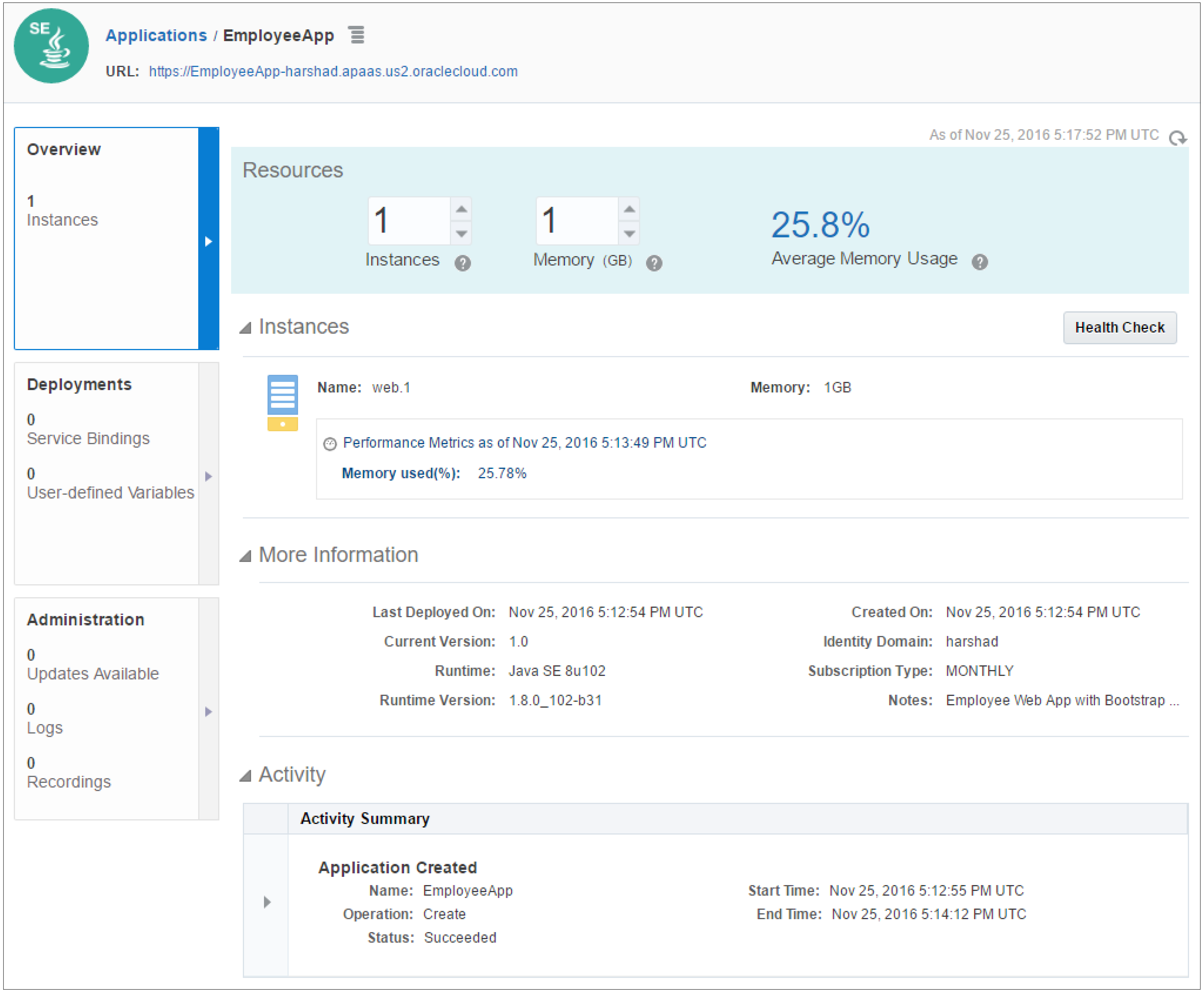
Instances

Instances: 1

Memory (GB): 1

Version: Java SE 8

Figure 3. Create application


SE
Applications / EmployeeApp
Overview
Deployments
Administration
Resources
Health Check
Instances
More Information
Activity
Activity Summary

URL: https://EmployeeApp-harshad.apaas.us2.oraclecloud.com

1 Instances

0 Service Bindings

0 User-defined Variables

0 Updates Available

0 Logs

0 Recordings

As of Nov 25, 2016 5:17:52 PM UTC

1 Instances

1 Memory (GB)

25.8% Average Memory Usage

Name: web.1

Memory: 1GB

Performance Metrics as of Nov 25, 2016 5:13:49 PM UTC

Memory used(%): 25.78%

Last Deployed On: Nov 25, 2016 5:12:54 PM UTC

Current Version: 1.0

Runtime: Java SE 8u102

Runtime Version: 1.8.0_102-b31

Created On: Nov 25, 2016 5:12:54 PM UTC

Identity Domain: harshad

Subscription Type: MONTHLY

Notes: Employee Web App with Bootstrap ...

Application Created

Name: EmployeeApp

Operation: Create

Status: Succeeded

Start Time: Nov 25, 2016 5:12:55 PM UTC

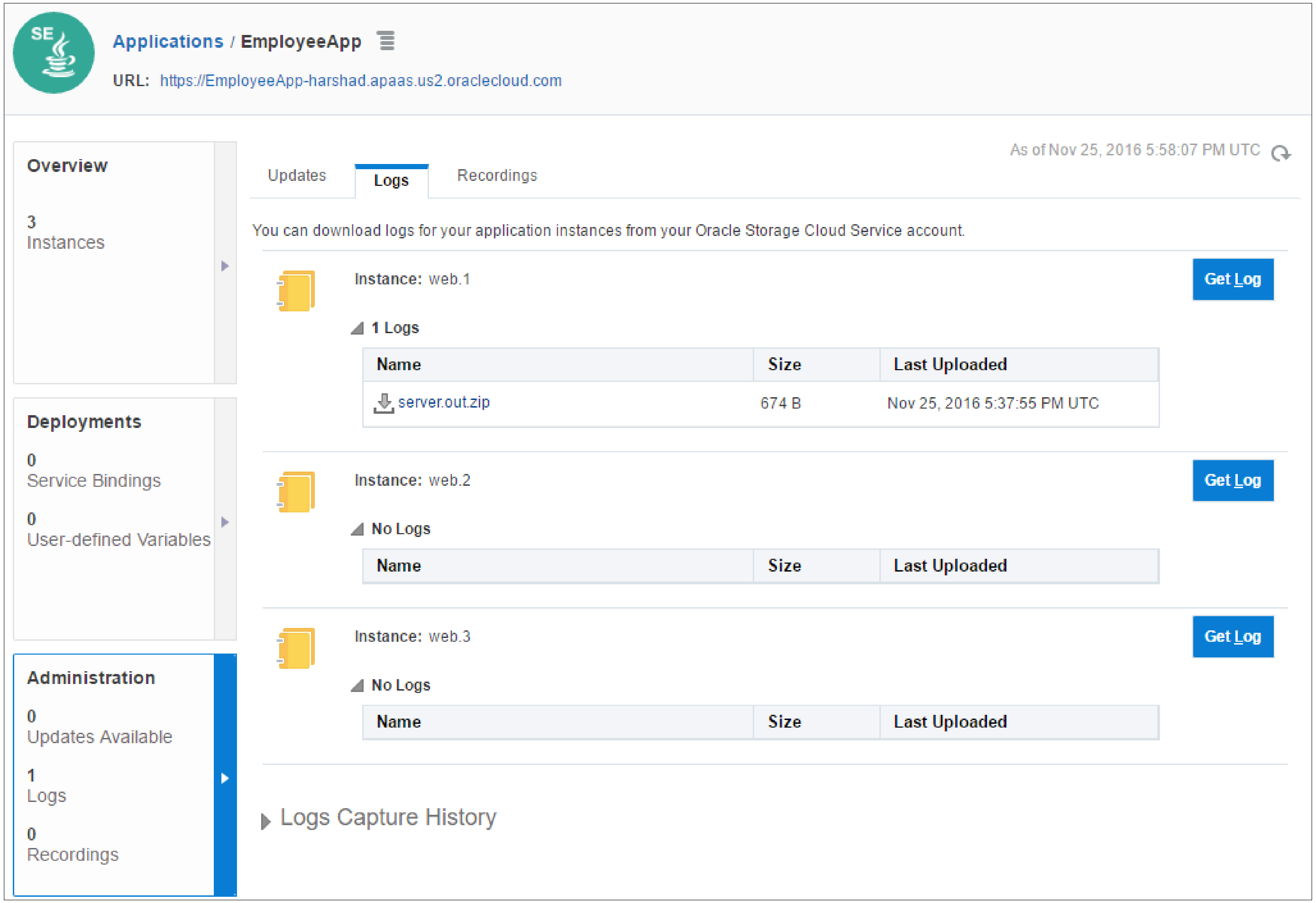
End Time: Nov 25, 2016 5:14:12 PM UTC

Figure 4. Application overview page

Modify Resources

Figure 4 displays the number of instances and the allocated memory. One of the key features of containerization is that you can easily add identical new containers as required. Change the number of instances to any number between 1 and 16, and within minutes you get identical containers up and running. The existing containers will continue running uninterrupted while this process is under way.

The memory can be set from 1 GB to 20 GB. Note that modifying the allocated memory will lead to all containers being restarted. Because scaling up the memory requires a restart of all containers and potential downtime, Oracle Application Container Cloud Service provides a Rolling Restart option that stops and restarts container instances



Applications / EmployeeApp

URL: https://EmployeeApp-harshad.apaas.us2.oraclecloud.com

Overview

3
Instances

Deployments

0
Service Bindings

0
User-defined Variables

Administration

0
Updates Available

1
Logs

0
Recordings

As of Nov 25, 2016 5:58:07 PM UTC

Updates | **Logs** | Recordings

You can download logs for your application instances from your Oracle Storage Cloud Service account.

Instance: web.1 — Get Log

1 Logs

Name	Size	Last Uploaded
server.out.zip	674 B	Nov 25, 2016 5:37:55 PM UTC

Instance: web.2 — Get Log

No Logs

Name	Size	Last Uploaded

Instance: web.3 — Get Log

No Logs

Name	Size	Last Uploaded

Logs Capture History

Figure 5. The Administration options, showing the Logs tab

one at a time. A Rolling Restart is slower but is a useful feature to have because it means no downtime for the application.

In a short period of time and without much effort, the web application is running in multiple, identical, isolated containers in the Oracle Cloud.

Updates, Logs, and Recordings

If you click on the Administration section as shown on the left side of **Figure 4** and **Figure 5**, you will find an Update tab that provides details of the Java runtime version, as well as a Logs tab, which holds the logs for each container. There is also a Recordings tab, where you can initiate recordings that can be used for Java Flight Recorder diagnostics and profiling.

Service Bindings

If you click on the Deployments section as shown on the left side of **Figure 5**, you will see a Service Bindings section. If you have any existing services in Oracle Java Cloud Service or Oracle Database Cloud Service, you can bind your application to those services there. You also have the option of specifying these bindings via a deployment.json file.

So in a short period of time and without much effort, I have a full-fledged web application running in multiple, identical, isolated containers in the Oracle Cloud. Not visible but present is a load balancer provided by Oracle Application Container Cloud Service that routes traffic to the application containers.

In this article, we have used an existing application to try out Oracle Application Container Cloud Service. However, you could create a new Maven project or update an existing project to run your application using Oracle Application Container Cloud Service. The key things you need to do to run your web application using Oracle Application Container Cloud Service are (1) package the dependencies, (2) package the embedded server, and (3) specify how to run the packaged application and the server.

Oracle Cloud is about multiple cloud services integrating and collaborating to deliver your solution. For Oracle Application Container Cloud Service, apart from the bindings with Oracle Java Cloud Service and Oracle Database Cloud Service, you should also take a look at Oracle Developer Cloud Service, which was discussed in the previous issue. Apart from its many "developer-environment-as-a-service" features, Oracle Developer Cloud Service can also simplify and automate building and deploying your applications to Oracle Application Container Cloud Service.

Conclusion

Containers' promise of an isolated, consistent environment and resource efficiency has many developers and businesses interested. Oracle Application Container Cloud Service delivers containerization in a fast, easy-to-use, and easy-to-manage format. A free trial is available via the Oracle Application Container Cloud Service web page. `</article>`

Harshad Oak is a Java Champion and an Oracle Ace Director. He is the founder of IndicThreads and Rightrix Solutions. He is the author of *Pro Jakarta Commons* (Apress, 2004) and has written several books on Java EE. Oak has spoken at conferences in many countries.

learn more

Docker containers

Oracle Developer Cloud Service

BRIAN GOETZ

PHOTOGRAPH BY
BOB ADLER/VERBATIM

JEP 286

Local-Variable Type Inference: var and maybe val

[The following writeup is taken from public notes by Brian Goetz, architect for the Java language at Oracle, who then added additional information and observations. It shows the care exerted by language designers when adding a feature that is small in appearance: in this case, the possible addition of a keyword—`var`, `val`, or `let`—to reduce the amount of boilerplate needed for declaring variables. The syntax of the leading proposals was discussed in the March/April 2016 issue of *Java Magazine*. *—Ed.*]

More than 2,500 people participated in two rounds of surveys on JEP 286. Overall, the responses were strongly in favor of adding local-variable type inference; 74 percent were strongly in favor, with another 12 percent mildly in favor. 10 percent thought it was a bad idea.

The written comments had more of a negative bias, but this shouldn't be surprising; people generally have more to say in disagreement than in agreement. The positive comments were very positive; the negative comments were very negative. While there were some passionate arguments against, the numbers speak loudly: this is a feature that most developers want. (It is the most frequently repeated request of developers coming to Java from other languages.) So no matter what happens here, some people are going to be very disappointed.

When given a choice, the most popular syntax choice was "val and var," with "var only" as the second choice. But when asked how they felt about these choices, there was a divergence: more people liked "val and var," but more people hated it, too. Although people expressed strong preferences for their favorite syntax, it's important to remember that syntax is just the surface. Language features like this have a lot of complexity under the hood, even when they look simple.

Readability

The biggest category of negative comments regarded worries about readability (although most of these came from folks who have never used the feature; those who have used it in other languages were overwhelmingly positive). It is a well-established core design principle of the Java language that reading code is more important than writing code; but plenty of folks assumed that this feature would inevitably lead to less-readable code. Of course, it's easy to construct strawman examples, such as

```
var x = y.getFoo()
```

to support the belief that this feature would harm readability. But if you dig deeper, you realize that the readability problem here stems from the fact that `x` is just a poorly chosen variable name. (Having a manifest type might make up for the programmer's laziness, but it would be better to just choose a good variable name in the first place.)

Like any convention, the use of `var` can do a lot of damage if not applied properly. But the Java team believes that, when it is used properly, readability can actually be *enhanced*. The reason for this is that it moves the variable name into a more predictable place in the code. Consider a block of locals:

```
UserModelHandle userDB = broker.findUserDB();
List<User> users = db.getUsers();
Map<User, Address> addressesByUser =
    db.getAddresses();
```

In most cases, the variable names are more important than anything else on these lines, because they capture the role of the variable in the current program. And in these examples, the variable names are not so easy to visually pick out from the code above—they're stuck in the middle of each line, and they occur at a different place on each line.

Using inferred types brings the variable name prominently into the reader's view. If the block above were rewritten with inferred types, as follows,

```
var userDB = broker.findUserDB();
var users = db.getUsers();
var addressesByUser = db.getAddresses();
```

then the true intent of the code pops out much more readily, because the variable names are almost right up front, in a predictable place. The lack of manifest types is not an impediment, because good variable names were chosen.

Mutability

Many comments were not so much about the use of type inference, but about mutability. A lot of people like the idea of reducing the ceremony associated with finality. We like this idea, too.

An initial exploration of this feature used inference only for effectively final locals. But after working with a prototype, it was immediately clear how this violated the "bring the variable names front and center" imperative described earlier. Mutable locals would stick out badly:

```
var immutableLocal = ...
var anotherImmutableLocal = ...
var alsoImmutable = ...
LocalDateTime latestDateSeenSoFar = ...
var thisOneDoesntChangeEither = ...
```

This irregularity was visually jarring to readers of the code. Many people would prefer that we use `val` to mean `final var`, as Scala does, and assumed that they were so similar that readers could mostly ignore the difference if they wanted to. But usability experiments suggested that some people found the subtle difference between `var` and `val` in large blocks of locals, as in the example above, to be distracting. (Others found the more different `var` and `let`, as Swift uses, also distracting.)

The conclusion drawn by the Java team is that while immutability is important, in reality, local variables are the least important place where developers need more help making things immutable. The biggest risk of mutability is data races, but local variables are immune to data races. And the majority of local variables are effectively final anyway. Where more help is needed in encouraging immutability is for *fields*, but applying type inference there would be foolish.

The desire to reduce the ceremony of immutability is certainly well taken. However, in this case, it is pushing on the wrong end of the lever.

Further, the `var`/`val` distinction offers more leverage in other languages than it would in Java. In Scala, for example, all variables—locals and fields alike—are declared with `val name : type`, where you can omit the `: type` if you want inference. So, not only is mutability orthogonal to inference, but the power of the `var`/`val` distinction is greater because it is used in multiple contexts. However, Java would use it only for local variables, and only when their types are inferred. In sum, `var`/`val` is a feature that *looks* like it carries over cleanly, but in reality it suffers greatly in translation.

Syntax Choice

So, after considering the pros and cons at length, the Java team concluded there was an obvious winner: `var` only. Some of the reasons for this choice include the following:

- While it was not the most popular choice in the survey, it was clearly the choice that the most people were OK with. Many hated "var and val"; others hated "var and let." Almost no one hated "var only."
- Experience with C#, which has "var only," has shown that this is a reasonable solution in Java-like languages. There is no groundswell of demand for `val` in C#.
- The desire to reduce the ceremony of immutability is certainly well taken, but in this case, it is pushing on the wrong end of the lever. Where Java developers need help for immutability is with fields, not with locals. But `var` and `val` don't apply to fields, and they almost certainly never will.
- Finally, if variable names are more important than types, they're more important than mutability modifiers, too.

As can be seen, these decisions would not have been possible without the comments and help of many developers who took the time to share their opinions with the Java team. `</article>`

Brian Goetz (@BrianGoetz) is architect for the Java language at Oracle. He is the lead author of *Java Concurrency in Practice* (Addison-Wesley Professional, 2006) and a frequent speaker at Java conferences.

ORACLE

Java Is Just the Beginning

Your Java applications need high-performance and battle-tested platform and infrastructure services to build, test, deploy, and monitor. Oracle Cloud delivers.

Start with Java in the cloud—or choose whatever language, database, compute service, and OS option you need. Rapid scalability. True portability. It's all here. Now.

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

Start here:
developer.oracle.com

#developersrule

Quiz Yourself

Inner classes, collections, and connections—are you ready for this?

SIMON **ROBERTS**

To start the new year off right, I've put together some more problems that simulate questions from the 1Z0-809 Programmer II exam, which is the certification test for developers who have been certified at a basic level of Java 8 programming knowledge and now are looking to demonstrate more-advanced expertise.

Shortly after I finished these questions, a regular reader of this section asked whether I might include questions for the Oracle Certified Associate (OCA) exam, which is a more preliminary level of certification than the questions that have appeared here during the last year. I'll certainly include some in the quiz for the next issue.

Question 1. Given this code:

```
public class Scratch {
    public class Itch {
        static String type = "Text"; // line n1
    }
}
```

Which of the following are true? Choose two.

a. The code compiles successfully.

b. The code would compile if line n1 included the modifier `private`.

c. The code would compile if line n1 included the modifier `final`.

d. The code would compile if line n1 included the modifier `public`.

e. The code would compile if the modifier `static` were removed from line n1.

Question 2. Given this code:

```
class Ordered implements Comparable<Ordered> {
    int order;  String name;
    public Ordered(int order, String name) {
        this.order = order; this.name = name;
    }
    public int compareTo(Ordered other) {
        return this.order - other.order;
    }
}
```

And this code:

```
Set<Ordered> so = new TreeSet<>();
so.add(new Ordered(9, "Mélina"));
so.add(new Ordered(3, "Sonia"));
so.add(new Ordered(9, "Jaquelina"));
so.add(new Ordered(11, "Alaïs"));
so.forEach(i->System.out.println(i.name));
```

What is the result? Choose one.

a. `Mélina`
`Sonia`
`Jaquelina`
`Alaïs`

b. `Sonia`
`Mélina`
`Jaquelina`
`Alaïs`

c. `Sonia`
`Mélina`
`Alaïs`

d. `Sonia`
`Jaquelina`
`Alaïs`

e. The order of output is platform-dependent.

Question 3. Given the following code fragment:

```
// line n1
Connection conn = DriverManager.getConnection(
  "jdbc:derby://localhost:1527/sample; " +
   "user=app;password=app");
// line n2
Statement statement = conn.createStatement();
// line n3
ResultSet rs = statement.executeQuery(
     "SELECT * FROM APP.CUSTOMER");
// line n4
while (rs.next()) {
    String name = rs.getString("NAME");
    System.out.printf("%s\n", name);
}
// line n5
```

Which releases all the database resources? Choose one.

a. Insert the following:
At line n1: `try (`
At line n3: `) {`
At line n5: `}`

b. Insert the following:
At line n2: `try (`
At line n4: `) {`
At line n5: `}`

c. Insert the following:
At line n3: `try (`
At line n4: `) {`
At line n5: `}`

d. Insert at line n5: `rs.close(); statement.close();`

Question 1. The correct answers are options C and E. This question probes an aspect of your understanding of inner classes. The term *inner class* is used fairly loosely most of the time, but it actually has a fairly specific meaning in the documentation. An inner class is a nonstatic class declared inside another class. If the class is static, then it's considered to be a nested class, not an inner class. And there are corner-case rules about inner classes. Section 8.1.3 of the *Java Language Specification*, "Inner Classes and Enclosing Instances," states the following: "It is a compile-time error if an inner class declares a member that is explicitly or implicitly static, unless the member is a constant variable." You might well ask, what is a constant variable? Section 4.12.4 of the specification states that a constant variable is "a final variable of primitive type or type `String` that is initialized with a constant expression." Interestingly, the compiler even rejects the value null as the constant expression for this assignment.

These rules mean that the code shown in the question can't work, because it has an inner class attempting to declare a static variable that doesn't qualify as a constant variable. As a result, option A must be incorrect. Also, nothing in the documentation suggests that this rule varies based on accessibility, so options B and D are incorrect, too.

That leaves only options C and E as the two correct answers. I should explain why these two are correct, not simply accept the elimination of all the others (even if doing that would have been sufficient for Sherlock Holmes).

Considering option C, if you make the declaration final, you actually have achieved a constant variable. Recall that the definition of a constant variable required a primitive or `String`, that the variable be final, and that it be initialized with a constant expression. I should justify the assertion that you have a constant expression. Section 15.28 of the specification, "Constant Expressions," describes the criteria for this, and one of the first items in a fairly long list is the `String` literal. So, option C is correct.

Option E actually describes the case where the field is reverted to an instance field rather than a static one. There are no restrictions on a regular instance field in an inner class, so option E is also correct.

There's just one thing left to consider, and that is why this rule exists. It probably seems a little arbitrary. I don't have insight into the minds of those who put this together, but I do have an explanation that makes reasonable sense and might serve to make the rule seem sensible—and, therefore, easier to remember—even if it's not actually the logic that led to the rule.

With an inner class—that is, a nonstatic inner class—it's as if the class itself is a member of the enclosing instance. That would require that every new outer instance create a new inner class, just like a new instance creates a new copy of every field. In this case, the inner class is based on the same source code as all the other inner classes. Each inner class would have its own version of any static members, but how would you refer to those static members? You'd need to distinguish which class you're referring to when you want to use one of these fields. That could be complex, ugly, error-prone, and—as history has shown—unnecessary. (Java has survived with this restriction since the inception of inner classes in Java 1.1.)

I'll reiterate that this perspective might not be what the designers had in mind, but they did call these things inner *classes* and not inner *objects*. Either way, I hope the mental model serves to help you remember that statics are not permitted in inner classes. Nested classes, however, have no such restriction.

Caution should be exercised when using a comparator capable of imposing an ordering inconsistent with equals.

Question 2. The correct answer is option C. This question investigates the behavior of sets in the Java APIs. The most fundamental behavior of a set is that it does not permit duplicate entries. In the API documentation for the `Set` interface, this requirement is expressed in terms of the `equals` method: specifically, that no two non-null entries may return true when one is tested against the other using the `equals` method, and also that at most one null item is permitted. To do this, before actually adding a new item, the set must test to see if the offered item is already in the set. A simple check of every item in turn would be very time-consuming, particularly as the number of items in the set grows. Therefore, the concrete implementations provided by Java seek to improve the speed of this check.

In `TreeSet`, the items are added to the set in order, and this means that the set can use a binary search to locate an item (or determine that it's not present). A binary search is much more efficient, particularly as the number of items in the set grows. However, in the example in the question, the ordering is determined solely by the `int` field called `order`, and this means that the two objects with `order` equal to 9 are at the same point in the ordering, even though they have clearly different values. The Java documentation refers to this as "ordering that is *inconsistent with equals*." The documentation for `TreeSet` warns: "Note that the ordering maintained by a set. . .must be *consistent with equals* if it is to correctly implement the `Set` interface."

In Java 7, the JDBC API was extended to make good use of the try-with-resources feature.

The effect is that `TreeSet` treats those two objects that have the value 9 in the `order` field as being equal (in fact, the implementation of `TreeSet` never uses the `equals` method when considering the items in the set). As a result, the second item with `order` equal to 9 is considered a duplicate. In this particular example, there is no override of the `equals` method, which strictly means that every object is considered unique, and even objects created with identical field values should properly be permitted to be simultaneous set members.

Because of this, the example considers that the two items with the names Mélina and Jaquelina are duplicates. As a result, they cannot both be in the set at the same time. Therefore, options A and B must be false.

The next consideration is exactly how a `Set` behaves when presented with a duplicate. Does it reject the new item or replace the existing item? This information allows you to decide if option C or option D correctly defines the contents of the set. This consideration is resolved by the documentation for the `add` method of the `Set` interface, which states: "Adds the specified element to this set if it is not already present . . . . If this set already contains the element, the call leaves the set unchanged."

Because of this, you know that the first of the two items that have the same value for `order` will remain in the set, and the second will be rejected. Therefore, Mélina remains, and Jaquelina does not make it into the set. This means that option C is possible, but option D must be wrong.

Finally, option E suggests that the order might not be predictable. This is an interesting issue. On one hand, a `Set` differs from a `List` in that it does not honor the order in which items are added. But, the reason that no such guarantee is offered is that the `Set` implementation is at liberty to use an internal order that speeds up the search for duplicates. In the case of `TreeSet`, the order will be the order specified for use in the tree-building mechanism. In this example, that's simply the order defined by the `compareTo` method of the `Ordered` class. At this point, you could reasonably object to a question that appears to depend on an implementation detail as the underpinning of the correct answer. However, `TreeSet` also implements another interface, `NavigableSet`, that makes the order part of the public contract. Further, the documentation for `TreeSet` states that in a `NavigableSet` implementation, the elements are ordered using their natural ordering or by a comparator provided at set creation time, depending on which constructor is used.

As a result, the order will not vary between implementations and option E is incorrect while option C is correct.

As a final observation, Java's API documentation recommends that whenever possible, natural ordering should be "consistent with equals." The API documentation for `Comparable` states the following: "It is strongly recommended (though not required) that natural orderings be consistent with equals. This is because sorted sets (and sorted maps) without explicit comparators behave 'strangely' when they are used with elements (or keys) whose natural ordering is inconsistent with equals. In particular, such a sorted set (or sorted map) violates the general contract for set (or map), which is defined in terms of the `equals` method." Notice that the documentation explicitly calls out the weirdness this question has been investigating.

While this is a reasonable goal for the `Comparable` interface and the "natural ordering," the whole point of the `Comparator` interface is to permit the expression of multiple different orderings for a class. This is important, because I might, for example, want a list of students in grade order or, if I'm the basketball coach, in height order. Similarly, if I'm the accountant, I might want students in order of the amount of money each owes the school. It's not possible to have multiple orders and expect them all to be consistent with equals. However, that's generally still considered to be a good thing if it's possible, as the documentation for `Comparator` notes: "Caution should be exercised when using a comparator capable of imposing an ordering inconsistent with equals to order a sorted set (or sorted map)."

Mostly, however, it's important to be aware of the wrinkles that arise when order and equality are inconsistent.

Question 3. The correct answer is option A. Proper release of nonmemory resources is often crucial to the correct long-term running of any server-type program. Database connections are certainly important resources requiring proper treatment; and in Java 7, the JDBC API was extended to make good use of the try-with-resources feature. In particular, `Connection`, `Statement`, and `ResultSet` all implement `AutoCloseable`.

It's probably tempting to guess that option D, which doesn't use try-with-resources, is incorrect, but you can't reject it based on not liking it. Closer inspection, however, shows that this option does not close the main database connection. There's no mandate that absolutely requires that a connection be used once and then released, but the question specifically asks which option releases *all* the resources. Because of this, option D can properly be rejected because it leaves the connection open.

The other three options all use the try-with-resources mechanism but auto-close different resources. Option A closes the `Connection` and the `Statement` but not the `ResultSet`. Option B closes the `Statement` and the `ResultSet` but not the `Connection`. Option C closes the `ResultSet` alone.

Is it possible that none of these works fully correctly, according to the requirements of the question? Well, it turns out that the specification of these behaviors helps out. If you look at the API documentation for the `Connection.close()` method, you can see that it says the method "releases this `Connection` object's database and JDBC resources immediately." This suggests that if you auto-close the `Connection` alone, you have fulfilled the requirements of the question. However, there's no such option.

Option A does, however, close both the `Connection` and the `Statement`. These will close in reverse order, so the `Statement` is closed first and then the `Connection`. Regardless, the effect is that *all* resources will be closed. In fact, there's a comment about the `Statement.close()` method that mirrors the one for the `Connection.close()` method: "Releases this `Statement` object's database and JDBC resources immediately. . . . It is generally good practice to release resources as soon as you are finished with them to avoid tying up database resources." This tells you that option A achieves the desired effect, even though it does not explicitly close the `ResultSet`.

The auto-closing of "subresources" can be significant in this type of coding. Imagine that you've prepared a result set containing the data you need, and then you decide to be a good citizen and release the unneeded `Statement` and `Connection`. This, unfortunately, would close your `ResultSet`, too, taking your data away—potentially before you have finished using it. </article>

Simon Roberts joined Sun Microsystems in time to teach Sun's first Java classes in the UK. He created the Sun Certified Java Programmer and Sun Certified Java Developer exams. He wrote several Java certification guides and is currently a freelance educator who teaches at many large companies in Silicon Valley and around the world. He remains involved with Oracle's Java certification projects.

Comments

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

Article Proposals

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

Customer Service

If you're having trouble with your subscription, please contact the folks at java@halldata.com (phone +1.847.763.9635), who will do whatever they can to help.

Where?

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

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

- Subscription application
- Download area for code and other items
- *Java Magazine* in Japanese