Java™ magazine

By and for the Java community

SEPTEMBER/OCTOBER 2017

More Java 9

ORACLE.COM/JAVAMAGAZINE

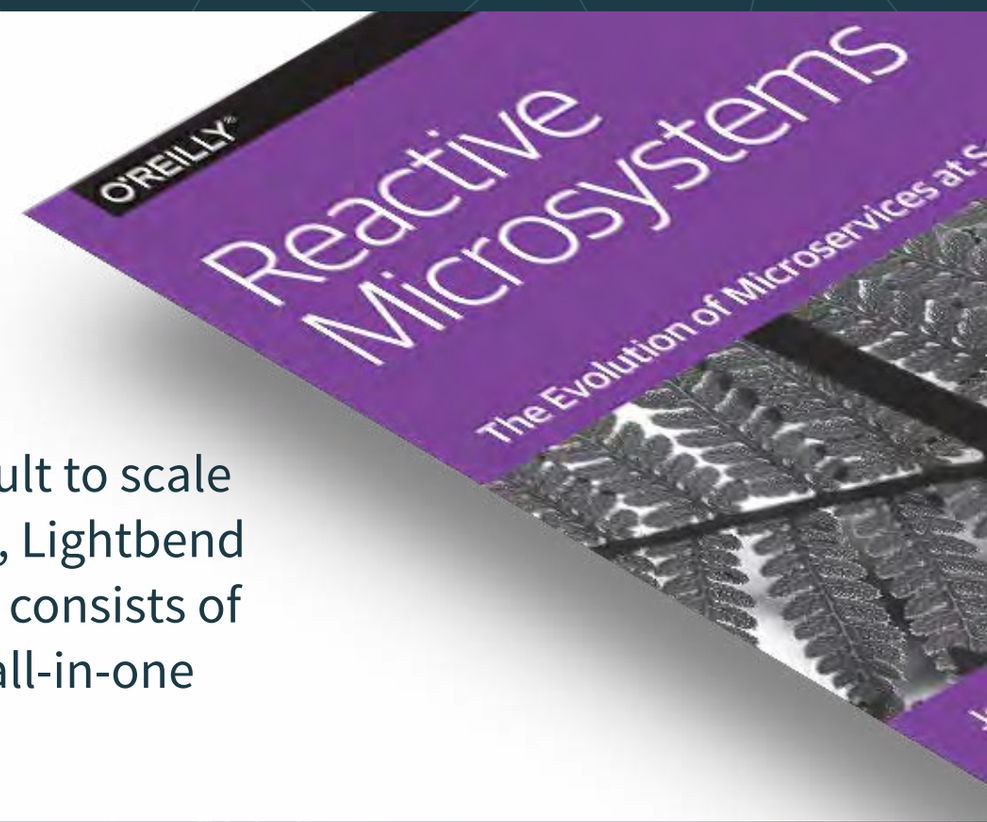
ORACLE®

Reactive Microsystems

Lightbend

If you want your application to provide millisecond response times and close to 100% uptime, traditional architectures with single SQL databases and thread-per-request models simply cannot compete with microservices. This report discusses strategies and techniques for building scalable and resilient microservices, and helps you work your way through the evolution of a scalable microservices-based system.

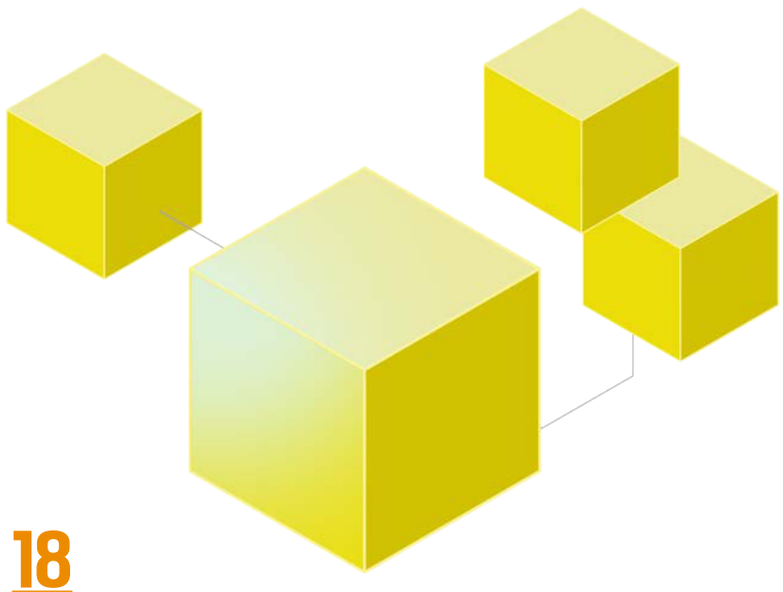
Learn how to refactor a monolithic application step-by-step!



DOWNLOAD EBOOK

Still chugging along with a monolithic enterprise system that's difficult to scale and maintain, and even harder to understand? In this concise report, Lightbend CTO Jonas Bonér explains why microservice-based architecture that consists of small, independent services is far more flexible than the traditional all-in-one systems that continue to dominate today's enterprise landscape.

//table of contents /



COVER ART BY WES ROWELL

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 and Audience Development Director
Karin Kinnear

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@omeda.com

PRIVACY

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

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.

Kotlin Conf

2-3 NOV. 2017 SAN FRANCISCO, PIER 27

TWO DAYS OF LEARNING AND NETWORKING WITH KOTLIN CREATORS AND COMMUNITY ENTHUSIASTS

-10% COUPON CODE FOR JAVA MAGAZINE READERS:

JAVA-MAG-READER-AT-KC

Valid through October 30, 2017

JET BRAINS

The first official conference by JetBrains, on Kotlin programming language

The Convergence of OpenJDK and the Oracle JDK

The differences between the releases are scheduled to disappear shortly.

In a significant burst of news, Oracle announced several important changes to Java that it will undertake shortly. The announcement in early September focused on two things primarily: the frequency of releases and the convergence of OpenJDK and the Oracle product.

In my previous editorial, I discussed the role of community in determining major release dates. I also hinted that Oracle was looking to move to a new cadence of releases. A detailed proposal for the new schedule was presented by Mark Reinhold, the chief architect of the Java Platform Group, in a blog post. The summary version is that Java will have feature releases every six months rather than the present multi-year cycles. Reinhold does an excellent job of articulating the rationale for this new schedule, and I recommend reading the entire post.

If the proposed schedule had been used since the launch of Java 8, many of the features discussed in this issue and in the previous issue of *Java Magazine* would have been available to developers without having to wait for the implementation of modules to be completed—a factor that delayed delivery of Java 9 for many months.

If Reinhold's proposal is adopted for the most part, as I expect will happen, these changes will have a positive impact for developers and for sites that run Java. Once an approved version of the cadence is announced, I'll discuss some of those ramifications in an editorial, if not in a full-length article, in these pages.

PHOTOGRAPH BY BOB ADLER/THE VERBATIM AGENCY

The second announcement, which I want to focus on, is the intention to make OpenJDK and Oracle JDK binaries interchangeable. The Oracle JDK is the generally free version that you can download from Oracle with a few closed source utilities that can be licensed commercially. Those include the well-regarded Java Flight Recorder and other closed source features, which will be open sourced sometime in 2018 after discussion with OpenJDK contributors.

This step complements the already open source status of OpenJDK. Having OpenJDK built by Oracle with the goal of being easily interchanged with the Oracle JDK should facilitate cloud deployments and eliminate any questions about what's permissible with regard to deploying Java in containers.

The goal of this effort, per Reinhold, is to "make the OpenJDK builds more attractive to developers and to reduce the differences between those builds and the Oracle JDK." This is a worthy goal because in the past there have been doubts and concerns that the two JDKs were different enough that OpenJDK could not be entirely depended on. Vendors that provide builds for OpenJDK, such as RedHat, Azul, and Linaro, have long had to fight against this perceived uncertainty. Now, at last, the matter will be resolved and the entire Java SE development kit will be open source—a single, unified source of technology.

As a Java developer, I'm excited by both developments and foresee benefits in both the new schedule and the extended commitment to open source.

You'll notice as you read the principal articles in this issue that we have moved from the previous multiple-column format to a single column. This change should make it easier to read *Java Magazine* on a mobile device while maintaining its legibility in a browser. It also enables us to print wider images and present code without having to reformat it for narrow columns. Would you let us know whether you like this new presentation, hate it, or have other suggestions that would make the reading experience better? Many thanks!

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

HELD CAPTIVE BY YOUR **DEVELOPMENT PLATFORM?**

Free yourself with a polycloud, polyglot application development and modernization platform that allows developers to create cloud-native and cloud-enabled applications faster with microservices and containers.

RED HAT® OPENSHIFT APPLICATION RUNTIMES

- Multiple runtimes
- Multiple frameworks
- Multiple clouds
- Multiple languages
- Multiple architectural styles

INTERESTED?
VISIT THE RED HAT BOOTH AT JAVAONE (#6101).

LEARN MORE

http://red.ht/rhoar

redhat.

JULY/AUGUST 2017

Collaboration and Scheduling

I wonder how many open source developers experienced the assembly lines of the past, marked along the way by PERT chart dates that could not be missed or hell had to be paid! And who made these schedules, and how were the elemental dates computed? Not with input from developers, nor with any sense of collaboration. Scared-out-of-their-skin nontechnical project managers got orders from on high: "Here is the completion date that senior management promised; make it happen."

What you described in the July/August 2017 editorial ("The Noisy, Successful Undertaking of Collaborative Work") regarding Java's release deadline will surely be envied by the legions of retired assembler, COBOL, and Fortran programmers. Keep up the excellent advocacy for collaboration!

—Richard Elkins

Department of Corrections

I went through the article by Simon Ritter ("Nine New Developer Features in JDK 9") and found it very instructive about the new features of JDK 9. But I am not sure that what he says on page 12 about `takeWhile(Predicate)` and `dropWhile(Predicate)` methods is quite correct. I think that in both cases, "until the `test()` method of the `Predicate` returns true" should be replaced by either "until the `test()` method of the `Predicate` returns *false*" or "*while* the `test()` method of the `Predicate` returns true."

—Alain-Michel Chomnoue Nghemning

Author Simon Ritter responds: "Looking at the section you reference, you are correct; the wording should be 'while the `test()` method of the `Predicate` returns true' rather than 'until.' Of course, this makes perfect sense, because the methods are `dropWhile` and `takeWhile`. Sorry for the confusion. In addition, I should point out one other correction, which was kindly brought to my attention by reader Richard Grin. I mentioned that `ifPresent(Consumer)` comes with Java 9, but in fact it first shipped as part of Java 8."

Editor Andrew Binstock adds: "An error—more of a typo—was brought to our attention by reader Sriram Muthaiah, who points out that `Stream.of(property)` in the second code block in the left column on page 23 ('Java 9 Core Library Updates: Collections and Streams' by Raoul-Gabriel Urma and Richard Warburton) should be `Stream.of(prop)`. We regret these errors and have corrected them in the currently posted version of this issue. If you previously downloaded this issue as a PDF, we suggest redownloading it so that you have the freshest, most correct version."

Contact Us

We would like your feedback on the one-column format we've implemented in the following pages.

In addition, we welcome comments, suggestions, grumbles, kudos, article proposals, and chocolate chip cookies. All but the last two might be edited for publication. If your note is private, please indicate this in your message. Write to us at javamag_us@oracle.com. For other ways to reach us, check out the last page of this issue.

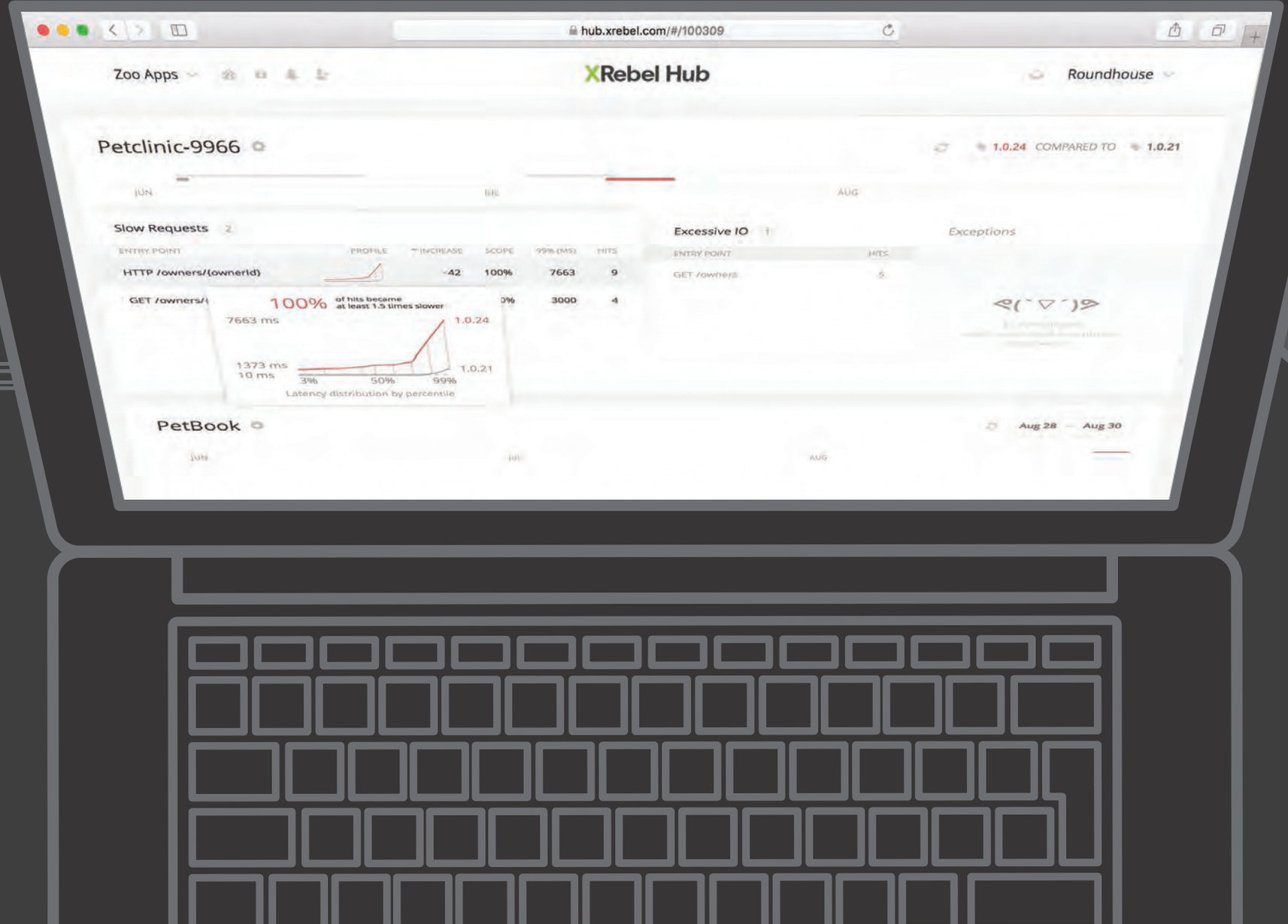
75% of application issues make it to production.

Find, diagnose and fix them, before they reach your customers.

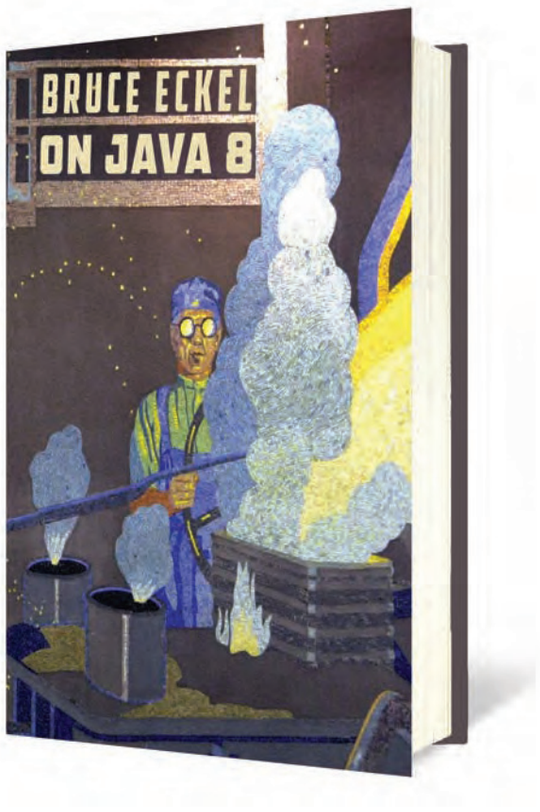
REQUEST A DEMO!

XRebel Hub

The only APM built for development and testing



ZEROTURNAROUND



ON JAVA 8

By Bruce Eckel

The name Bruce Eckel might not be familiar to many readers, but he's a Java Champion who had a seminal influence on teaching Java to programmers. His principal contribution was a book called *Thinking in Java*, which was a fat tutorial on all aspects of the language. It was a sequel to an earlier work on C++ that was distinguished by lengthy explanations of object orientation that were integrated with instruction on the language details. So, you learned how to think in an OO way as you acquired language skills. This approach to Java went through four editions—covering up through Java 5—when Eckel soured on the language and moved on to other work. Per the introduction to this volume, the changes he saw in Java 8 brought him back to the fold.

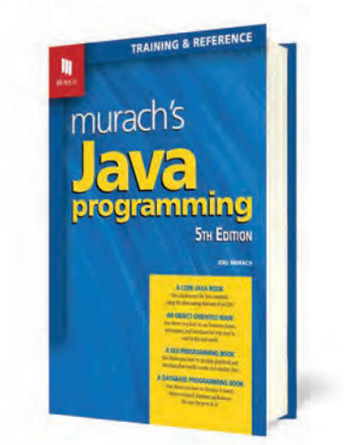
As with his previous books, Eckel follows his own path in both content and production. For example, this book exists solely in electronic form and is available only from Google Play. It is also self-edited and laid out by the author. I'll come back to this in a moment.

The content is unique compared with the formal tutorials that I've reviewed several times in this column. Eckel uses a friendly, informal approach with a compelling "come look over my shoulder as I do this" tone. It's undeniably engaging. Because of this "we're in this together" conviviality, however, you need to follow where Eckel takes you. Where your interests coincide with his, you'll be well fed. An excellent example of this is his discussion of CompletableFuture, which is one of the most detailed explanations I've seen in any tutorial. And it's remarkably approachable. However, when Eckel gets into topics of little interest, you'll find yourself flipping pages quickly. For example, his summary of Java operators has 13 pages of one-line examples. It's a recap of the preceding 29 pages on operators. In total, operators take up 57 pages. In Cay Horstmann's *Core Java*—itself a massive volume—operators consume a mere nine pages, which I expect is more to the liking of most readers.

Where the book really shines is predictably in the explanations of how core features in Java are the implementation of object orientation. In these discussions, Eckel deftly compares Java's way of doing things with how they're done in C++ (and occasionally Python). If you know those languages, especially C++, you'll find this book a very interesting read.

Despite all this unique goodness—and there is a surprising amount given how many excellent Java tutorials already exist—I encountered two frustrating limitations. The first is the layout, which works well on tablets but is very unsatisfactory in a browser. Because I'm far more likely to read

code examples in my browser than on my tablet, this is an important limitation. The other is the timing of the book. This volume is strictly about Java 8, but it went "to press" just weeks before the release of Java 9—a rather unfortunate choice that makes the contents immediately out of date. Talking about completable futures, if this book were to add coverage of Java 9, I would easily recommend it, if you're comfortable with the caveats above. *—Andrew Binstock*



MURACH'S JAVA PROGRAMMING, 5TH EDITION

By Joel Murach

The Murach books take a unique approach to teaching code: language topics are presented for the most part as two-page spreads. The left page explains the topic, while the right page provides a full working program that illustrates the details. This code-intensive approach makes the work both an excellent tutorial and a book of recipes. Want to connect to a database via JDBC? Turn to that section, and you'll find multiple short programs that show you how to set up a connection, formulate a query, get your results, and do updates. By providing complete examples, one of beginners' most frequent problems is eliminated: the question "how do I use the code that I think I now understand?"

For instructors, these books have a special appeal. Each two-page lesson (and given the book's 760+ pages, there are many lessons) can serve as a working example in the classroom. They can also form the basis for homework assignments. (For example, take the example on lists and implement the same idea using a database.) In this regard, I have long felt that the Murach books are the ones I'd use for teaching an introductory Java programming course. However, the books rarely reach beyond the introductory level (nor claim to). So advanced topics such as the complexities of concurrency are not explored.

This fifth edition adds coverage of Java 9—focusing mostly on the basics of modules and explaining the new requirements for how to compile and build programs. *—A.B.*

PROPOSED JDK ENHANCEMENT PROPOSAL

Making Java Container-Aware

There's no doubt that part of what makes microservices viable is the ability to deploy the services in a container—that is, in a stripped-down virtualized environment, such as Docker. This JDK Enhancement Proposal (JEP), suggested by Bob Vandette of Oracle's Embedded Java team, seeks to define technology that makes Java inside a container container-aware. The principal problem it seeks to address is how Java can be made aware of container policies that might affect the Java runtime—in particular, being accurately informed of resource constraints.

Right now, when the JVM starts up, it obtains information about the CPU and the environment. This information can be inaccurate if the physical configuration is set by command-line switches for Docker containers. The data the JVM collects can lead it to overestimate the memory available to it, which leads to incorrect decision-making internally and ultimately to poor performance.

This proposal (note that it has not yet reached the stage of a JEP) suggests that a new API be developed that would capture the correct data for numerous configuration details of the container, including memory limits and memory usage; CPU counts, sets, and usage; block I/O details; device I/O read and write rates; and host details such as shared memory size.

In whatever form it takes, it's clear that support for the use of Java in containers is likely to come to the language sooner rather than later. When it does, it will first appear in JEPs, as proposed here, and in Java Specification Requests (JSRs).

Oracle Press™

Your Destination for Oracle and Java Expertise

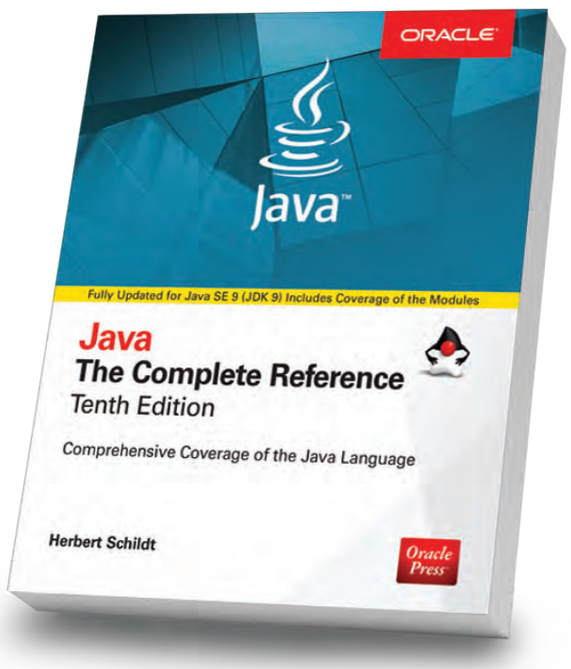
Visit the Oracle Store in the Moscone West lobby at Oracle OpenWorld and JavaOne 2017 to see our latest releases.

Purchase two or more Oracle Press books to receive a free Oracle Press teddy bear!

Java: A Beginner's Guide, 7th Edition

Herb Schildt

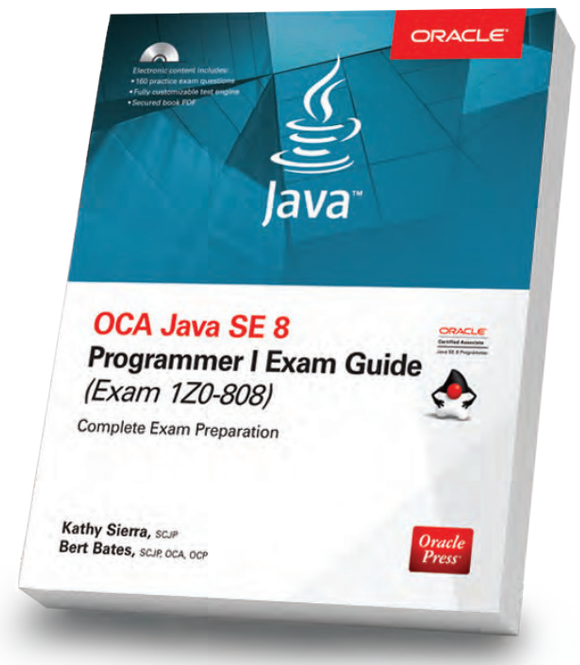
Revised to cover Java SE 9, this book gets you started programming in Java right away.



Java: The Complete Reference, 10th Edition

Herb Schildt

Updated for Java SE 9, this book shows how to develop, compile, debug, and run Java programs.



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

Kathy Sierra, Bert Bates

Get complete coverage of all objectives for Exam 1Z0-808. Electronic practice exam questions are included.

Rapid Modernization of Java Applications

G. Venkat

Adopt a high-performance enterprise Java application modernization strategy.

Available in print and eBook formats.

Oracle Press™

www.OraclePressBooks.com • @OraclePress

//events /

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 JavaFX; JVM languages; new development tools; insights into recent trends in programming; and tutorials on numerous related Java and JVM topics.

PHOTOGRAPH BY DALE CRUSE/FLICKR

Strange Loop

SEPTEMBER 28–30
ST. LOUIS, MISSOURI
Strange Loop is a multidisciplinary conference that brings together the developers and thinkers building tomorrow's technology in fields such as emerging languages, alternative databases, concurrency, distributed systems, security, and the web. In general, talks are code-heavy, not process-oriented. A preconference day on September 28 is optional and not included in the conference rate.

NFJS Boston

SEPTEMBER 29–OCTOBER 1
BOSTON, MASSACHUSETTS
Since 2001, the No Fluff Just Stuff (NFJS) Software Symposium Tour has delivered more than 450 events with more than 70,000 attendees. This event in Boston covers the latest trends within the Java and JVM ecosystem, DevOps, and agile development environments.

JAX London

OCTOBER 9–12
LONDON, ENGLAND
JAX London is a four-day conference for cutting-edge software engineers and enterprise-level professionals, bringing together the world's leading innovators in the fields of Java, microservices, continuous delivery, and DevOps. Conference sessions, keynotes, and expo happen on October 10–11. Hands-on workshops take place the day preceding and the day following the main conference.

Codemotion Berlin

OCTOBER 12–13
BERLIN, GERMANY
This year's Codemotion will be held at Kulturbrauerei Berlin, and more than 500 visitors are expected to attend the conference. The event is open to all languages and technologies, and features coding lectures and workshops.

Desert Code Camp

OCTOBER 14
CHANDLER, ARIZONA
Desert Code Camp is a free, developer-based conference built on community content. This year's sessions include talks on AI methodologies, blockchain, Clojure, Kotlin, and Groovy.

Java Enterprise Summit
OCTOBER 16–18
FRANKFURT, GERMANY
Java Enterprise Summit is a training event that explores microservices, API design, single-page applications, and cloud considerations with Java EE 7 and 8. Two tracks, "Architecture" and "Tech Deep Dive," are slated. (No English page available.)

O'Reilly Software Architecture Conference
OCTOBER 16–18, CONFERENCE AND TUTORIALS
OCTOBER 18–19, TRAINING
LONDON, ENGLAND
For four days, expert practitioners share new techniques and approaches, proven best practices, and exceptional technical skills. At this conference, you'll hear about the best tools to use and why, and the effect they can have on your work. You'll learn strategies for meeting your company's business goals, developing leadership skills, and making the conceptual jump from software developer to architect.

Java2Days
OCTOBER 17–19
SOFIA, BULGARIA
This conference features more than 80 in-depth sessions across 12 different tracks for software developers and IT professionals.

Voxxed Days Belgrade
OCTOBER 19–20
BELGRADE, SERBIA
Voxxed Days Belgrade is a Devoxx-branded conference hosted by the local HeapSpace developer community. This year's themes are languages and architecture, machine learning and artificial intelligence, augmented reality and virtual reality, and security.

JCON
OCTOBER 24–26
DUSSELDORF, GERMANY
JCON is a conference for professional Java development in practice, architecture, project management, and innovation. The main conference, on the first two days, is devoted to Java, frameworks, and microservices. Participation on these two days is free for all JUG members. The third day covers architecture and agile.

KotlinConf
NOVEMBER 2–3
SAN FRANCISCO, CALIFORNIA
KotlinConf is a JetBrains event that provides two days of content from Kotlin creators and community members.

Devoxx
NOVEMBER 6–10
ANTWERP, BELGIUM
The largest gathering of Java developers in Europe takes place again this year in Antwerp. Dozens of expert speakers deliver hundreds of presentations on Java and the JVM. Tracks include server-side Java, cloud, big data, and extensive coverage of Java 9.

W-JAX
NOVEMBER 6–10
MUNICH, GERMANY
W-JAX is a conference dedicated to cutting-edge Java and web development, software architecture, and innovative infrastructures. Experts share their professional experiences in sessions and workshops. This year's focus is on Java core and enterprise technologies, the Spring ecosystem, JavaScript, continuous delivery, and DevOps.

QCon San Francisco
NOVEMBER 13–15, CONFERENCE
NOVEMBER 16–17, WORKSHOPS
SAN FRANCISCO, CALIFORNIA
Although the content has not yet been announced, recent QCon conferences have offered several Java tracks along with tracks related to web development, DevOps, cloud

PHOTOGRAPH BY MARCO VERCH/FLICKR

computing, and more.

Javaneiros
NOVEMBER 18
CAMPO GRANDE, BRAZIL
The Javaneiros conference is a community-organized day of presentations related to Java programming. (No English page available.)

JVM-Con
NOVEMBER 28–29
COLOGNE, GERMANY
Among the topics slated for this conference devoted to JVM languages are Java 9, Java EE, Scala, Jython, Kotlin, Clojure, Groovy, JRuby, cloud native development, microservices, and security. (Website in German only.)

DevTernity
DECEMBER 1, KEYNOTES
DECEMBER 2, WORKSHOPS
RIGA, LATVIA
This conference focuses on modern approaches to coding, architecture, testing, engineering scalability, and operations. Twenty speakers present in three parallel tracks, followed by a full day of workshops.

ConFoo
DECEMBER 4–6
VANCOUVER, BRITISH COLUMBIA, CANADA
This multitechnology conference features targeted sessions for Java and JVM developers. Sessions include presentations on concurrency models with Java 9, Angular/TypeScript, and Practical Symfony 4.0.

CodeMash
JANUARY 9–12, 2018
SANDUSKY, OHIO
CodeMash is a unique event that educates developers on current practices, methodologies, and technology trends in a variety of platforms and development languages such as Java, .NET, Ruby, Python, and PHP.

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

THE EDINBURGH JUG

Although Scotland's capital is best known for its festivals and world heritage sites, Edinburgh is also a technology city. It has one of the highest concentrations of technology workers in the United Kingdom and is home to three universities, numerous financial services companies, the UK's largest tech incubator, and a thriving startup community. Many of these companies are of course built on Java.

The Edinburgh Java User Group (EdJug) was originally formed in 2011 to bring together Java developers from across Scotland. After activity trailed off, it was revamped in 2014 with a narrower focus on Edinburgh's local tech community.

Since 2014, EdJug has grown to about 80 members, consisting of professional developers, students, and others with an interest in Java and the JVM. It runs a meetup on the last Wednesday of every month, with talks from both local speakers and experts from around the world. Recent talks have covered static analysis, Android security, and property-based testing.

With the help of its sponsor, the JUG also organizes one-off events, including free weekend training courses. Past courses have covered Scala, Clojure, and Java 8.

Along with its neighbors in Glasgow and St. Andrews, Edinburgh is part of an international center of functional programming expertise. The activities of the JUG reflect this, collaborating with the local functional programming group to arrange training and workshops on languages such as Haskell and Idris.

UNLEASHING
THE NEXXT WAVE

November 14, 15 and 16, 2017 | @Casablanca Morocco

devoxx.ma | info@devoxx.ma | devoxx.ma/register

More Java 9

The advances in Java 9 are so extensive that we could dedicate several issues to them and still not cover them all. This was in all ways a *major* release of Java and the JDK. In the previous issue, we covered many of the major changes as well as how to transition from Java 8 to Java 9. However, we were unable to cover modules, the most important innovation in Java 9. The reason for the delay was explained in my editorial in that issue: a new release of the Java platform is a cooperative effort between Oracle and key partners as well as the community of users. When the final vote on modules was taken, not enough ayes were received for the release to go forward. After a few minor changes, however, the vote on the release of modules was successful. Unfortunately, the go/no-go uncertainty made it impossible for the magazine to cover modules without the risk of printing inaccurate or incomplete information. We've made up for it in this issue with a 15-page introduction to modules: what they are and how to use them, written by well-known trainer Paul Deitel.

We also continue the examination of language changes by authors Raoul-Gabriel Urma and Richard Warburton, who introduce new capabilities of Optionals and CompletableFutures—two features that were made popular in Java 8 and enhanced in this release. Finally, we have a look in JDK 9 at how method invocation works.

In addition, we have a very approachable introduction to Clojure (a Lisp-like JVM language), more coverage of JavaFX, and our usual quiz with the world's most detailed quiz answers.

We're also shifting to this new single-column format to make the magazine more readable, especially on mobile devices. Did we get it right? Could it be better still? How? Let us know at javamag_us@oracle.com. Thanks!

ART BY WES ROWELL

PAUL DEITEL

Understanding Java 9 Modules

What they are and how to use them

In this article, I introduce the Java 9 Platform Module System (JPMS), the most important new software engineering technology in Java since its inception. Modularity—the result of Project Jigsaw—helps developers at all levels be more productive as they build, maintain, and evolve software systems, especially large systems.

What Is a Module?

Modularity adds a higher level of aggregation above packages. The key new language element is the *module*—a uniquely named, reusable group of related packages, as well as resources (such as images and XML files) and a *module descriptor* specifying

- the module's *name*
- the module's *dependencies* (that is, other modules this module depends on)
- the packages it *explicitly* makes available to other modules (all other packages in the module are *implicitly unavailable* to other modules)
- the *services it offers*
- the *services it consumes*
- to what other modules it allows *reflection*

History

The Java SE platform has been around since 1995. There are now approximately 10 million developers using it to build everything from small apps for resource-constrained devices—like those in the Internet of Things (IoT) and other embedded devices—to large-scale business-critical and mission-critical systems. There are massive amounts of legacy code out there, but until now, the Java platform has primarily been a monolithic one-size-fits-all solution. Over the years,

there have been various efforts geared to modularizing Java, but none is widely used—and none could be used to modularize the Java platform.

Modularizing the Java SE platform has been challenging to implement, and the effort has taken many years. JSR 277: Java Module System was originally proposed in 2005 for Java 7. This JSR was later superseded by JSR 376: Java Platform Module System and targeted for Java 8. The Java SE platform is now modularized in Java 9, but only after Java 9 was delayed until September 2017.

Each module must explicitly state its dependencies.

Goals

According to JSR 376, the key goals of modularizing the Java SE platform are

- Reliable configuration—Modularity provides mechanisms for explicitly declaring dependencies between modules in a manner that's recognized both at compile time and execution time. The system can walk through these dependencies to determine the subset of all modules required to support your app.
- Strong encapsulation—The packages in a module are accessible to other modules only if the module explicitly exports them. Even then, another module cannot use those packages unless it explicitly states that it requires the other module's capabilities. This improves platform security because fewer classes are accessible to potential attackers. You may find that considering modularity helps you come up with cleaner, more logical designs.
- Scalable Java platform—Previously, the Java platform was a monolith consisting of a massive number of packages, making it challenging to develop, maintain and evolve. It couldn't be easily subsetted. The platform is now modularized into 95 modules (this number might change as Java evolves). You can create custom runtimes consisting of only modules you need for your apps or the devices you're targeting. For example, if a device does not support GUIs, you could create a runtime that does not include the GUI modules, significantly reducing the runtime's size.
- Greater platform integrity—Before Java 9, it was possible to use many classes in the platform that were not meant for use by an app's classes. With strong encapsulation, these internal

APIs are truly encapsulated and hidden from apps using the platform. This can make migrating legacy code to modularized Java 9 problematic if your code depends on internal APIs.
- Improved performance—The JVM uses various optimization techniques to improve application performance. JSR 376 indicates that these techniques are more effective when it's known in advance that required types are located only in specific modules.

Listing the JDK's Modules

A crucial aspect of Java 9 is dividing the JDK into modules to support various configurations. (Consult "JEP 200: The Modular JDK." All the Java modularity JEPs and JSRs are shown in **Table 1**.) Using the `java` command from the JDK's `bin` folder with the `--list-modules` option, as in:

```
java --list-modules
```

lists the JDK's set of modules, which includes the standard modules that implement the Java Language SE Specification (names starting with `java`), JavaFX modules (names starting with `javafx`), JDK-specific modules (names starting with `jdk`) and Oracle-specific modules (names starting with `oracle`). Each module name is followed by a version string—`@9` indicates that the module belongs to Java 9.

Module Declarations

As we mentioned, a module must provide a module descriptor—metadata that specifies the module's dependencies, the packages the module makes available to other modules, and more. A module descriptor is the compiled version of a module declaration that's defined in a file named `module-info.java`. Each module declaration begins with the keyword `module`,

JEP 200:	THE MODULAR JDK
JEP 201:	MODULAR SOURCE CODE
JEP 220:	MODULAR RUN-TIME IMAGES
JEP 260:	ENCAPSULATE MOST INTERNAL APIS
JEP 261:	MODULE SYSTEM
JEP 275:	MODULAR JAVA APPLICATION PACKAGING
JEP 282:	JLINK: THE JAVA LINKER
JSR 376:	JAVA PLATFORM MODULE SYSTEM
JSR 379:	JAVA SE 9

Table 1. Java Modularity JEPs and JSRs

followed by a unique module name and a module body enclosed in braces, as in:

```
module modulename {
}
```

A key motivation of the module system is strong encapsulation.

The module declaration's body can be empty or may contain various *module directives*, including `requires`, `exports`, `provides…with`, `uses` and `opens` (each of which we discuss). As you'll see later, compiling the module declaration creates the module descriptor, which is stored in a file named `module-info.class` in the module's root folder. Here we briefly introduce each module directive. After that, we'll present actual module declarations.

The keywords `exports`, `module`, `open`, `opens`, `provides`, `requires`, `uses`, `with`, as well as `to` and `transitive`, which we introduce later, are restricted keywords. They're keywords *only* in module declarations and may be used as identifiers anywhere else in your code.

requires. A `requires` module directive specifies that this module depends on another module—this relationship is called a *module dependency*. Each module must explicitly state its dependencies. When module A `requires` module B, module A is said to *read* module B and module B *is read by* module A. To specify a dependency on another module, use `requires`, as in:

```
requires modulename;
```

There is also a `requires static` directive to indicate that a module is required at compile time, but is optional at runtime. This is known as an *optional dependency* and won't be discussed in this introduction.

requires transitive—implied readability. To specify a dependency on another module and to ensure that other modules reading your module also read that dependency—known as *implied readability*—use `requires transitive`, as in:

```
requires transitive modulename;
```

Consider the following directive from the `java.desktop` module declaration:

```
requires transitive java.xml;
```

In this case, any module that reads `java.desktop` also implicitly reads `java.xml`. For example, if a method from the `java.desktop` module returns a type from the `java.xml` module, code in modules that read `java.desktop` becomes dependent on `java.xml`. Without the `requires transitive` directive in `java.desktop`'s module declaration, such dependent modules will not compile unless they *explicitly* read `java.xml`.

According to JSR 379 Java SE's standard modules must grant implied readability in all cases like the one described here. Also, though a Java SE standard module may depend on non-standard modules, it *must not* grant implied readability to them. This ensures that code depending only on Java SE standard modules is portable across Java SE implementations.

exports and exports...to. An `exports` module directive specifies one of the module's packages whose `public` types (and their nested `public` and `protected` types) should be accessible to code in all other modules. An `exports...to` directive enables you to specify in a comma-separated list precisely which module's or modules' code can access the exported package—this is known as a *qualified export*.

uses. A `uses` module directive specifies a service used by this module—making the module a service consumer. A *service* is an object of a class that implements the interface or extends the `abstract` class specified in the `uses` directive.

provides...with. A `provides...with` module directive specifies that a module provides a service implementation—making the module a *service provider*. The `provides` part of the directive specifies an interface or `abstract` class listed in a module's `uses` directive and the `with` part of the directive specifies the name of the service provider class that `implements` the interface or `extends` the `abstract` class.

open, opens, and opens...to. Before Java 9, reflection could be used to learn about all types in a package and all members of a type—even its `private` members—whether you wanted to allow this capability or not. Thus, nothing was truly encapsulated.

A key motivation of the module system is strong encapsulation. By default, a type in a module is not accessible to other modules unless it's a public type *and* you export its package. You expose only the packages you want to expose. With Java 9, this also applies to reflection.

Allowing runtime-only access to a package. An `opens` module directive of the form

```
opens package
```

indicates that a specific *package*'s `public` types (and their nested `public` and `protected` types) are accessible to code in other modules at runtime only. Also, all the types in the specified package (and all of the types' members) are accessible via reflection.

Allowing runtime-only access to a package by specific modules. An `opens…to` module directive of the form

```
opens package to comma-separated-list-of-modules
```

indicates that a specific *package*'s `public` types (and their nested `public` and `protected` types) are accessible to code in the listed modules at runtime only. All of the types in the specified package (and all of the types' members) are accessible via reflection to code in the specified modules.

Allowing runtime-only access to all packages in a module. If all the packages in a given module should be accessible at runtime and via reflection to all other modules, you may `open` the entire module, as in:

```
open module modulename {
   // module directives
}
```

Reflection Defaults

By default, a module with runtime reflective access to a package can see the package's `public` types (and their nested `public` and `protected` types). However, the code in other modules can

access *all* types in the exposed package and *all* members within those types, including `private` members via `setAccessible`, as in earlier Java versions. For more information on `setAccessible` and reflection, see Oracle's documentation.

At compile time, if multiple modules have the same name a compilation error occurs. At runtime, if multiple modules have the same name an exception occurs.

Dependency injection. Reflection is commonly used with dependency injection. One example of this is an FXML-based JavaFX app. [See "Define Custom Behavior in FXML with FXMLLoader" in this issue for more on using FXML. —*Ed.*] When an FXML app loads, the controller object and the GUI components on which it depends are dynamically created as follows:

- First, because the app depends on a controller object that handles the GUI interactions, the `FXMLLoader` injects a controller object into the running app—that is, the `FXMLLoader` uses reflection to locate and load the controller class into memory and to create an object of that class.
- Next, because the controller depends on the GUI components declared in FXML, the `FXMLLoader` creates the GUI controls declared in the FXML and injects them into the controller object by assigning each to the controller object's corresponding `@FXML` instance variable. In addition, the controller's event handlers that are declared with `@FXML` are linked to the corresponding controls as declared in the FXML.

Once this process is complete, the controller can interact with the GUI and respond to its events. We use the `opens…to` directive to allow the `FXMLLoader` to use reflection on a JavaFX app in a custom module.

Modularized Welcome App

In this section, we create a simple `Welcome` app to demonstrate module fundamentals. We

- create a class that resides in a module
- provide a module declaration
- compile the module declaration and `Welcome` class into a module
- run the class containing `main` in that module

After covering these basics, we also demonstrate

- packaging the `Welcome` app in a modular JAR file
- running the app from that JAR file

Welcome app's structure. The app we present in this section consists of two `.java` files—`Welcome.java` contains the `Welcome` app class, and `module-info.java` contains the module declaration. By convention, a modularized app has the following folder structure:

```
AppFolder
    src
        ModuleNameFolder
            PackageFolders
                JavaSourceCodeFiles
            module-info.java
```

For our app, which will be defined in the package `com.deitel.welcome`, the folder structure is shown in Figure 1.

The `src` folder stores all of the app's source code. It contains the module's *root folder*, which has the module's name—`com.deitel.welcome` (we'll discuss module naming in a moment). The module's root folder contains nested folders representing the package's directory structure—`com/deitel/welcome`—which corresponds to the package `com.deitel.welcome`. This folder contains `Welcome.java`. The module's root folder contains the required module declaration `module-info.java`.

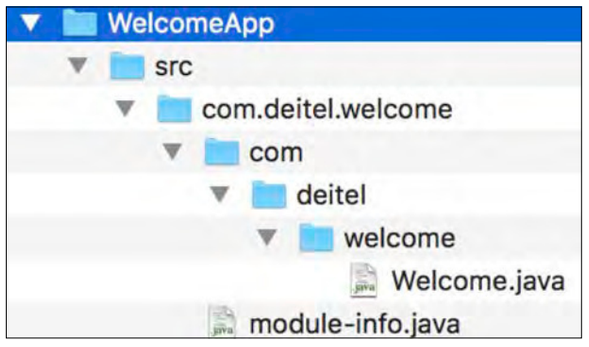


Figure 1. Folder structure for the Welcome app

Module naming conventions. Like package names, module names must be unique. To ensure unique package names, you typically begin the name with your organization's Internet domain name in reverse order. Our domain name is `deitel.com`, so we begin our package names with `com.deitel`. By convention, module names also use the reverse-domain-name convention.

At compile time, if multiple modules have the same name a compilation error occurs. At runtime, if multiple modules have the same name an exception occurs.

This example uses the same name for the module and its contained package, because there is only one package in the module. This is not required, but is a common convention. In a modular app, Java maintains the module names separately from package names and any type names in those packages, so duplicate module and package names *are* allowed.

Modules normally group related packages. As such, the packages will often have commonality among portions of their names. For example, if a module contains the packages

```
com.deitel.sample.firstpackage;
com.deitel.sample.secondpackage;
com.deitel.sample.thirdpackage;
```

you would typically name the module with the common portion of the package names—`com.deitel.sample`. If there's no common portion, then you'd choose a name representing the module's purpose. For example, the `java.base` module contains core packages that are considered fundamental to Java apps (such as `java.lang`, `java.io`, `java.time` and `java.util`), and the `java.sql` module contains the packages required for interacting with databases via JDBC (such as `java.sql` and `javax.sql`). These are just two of the many standard modules. The online documentation for each standard module, such as `java.base`, provides the module's complete list of exported packages.

Class Welcome. The following code presents a `Welcome` app that simply displays a string at the command line. When defining types that will be placed in modules, every type *must* be placed into a package (as in the third line below):

```
// Welcome.java
// Welcome class that will be placed in a module
package com.deitel.welcome; // all classes in modules must be packaged

public class Welcome {
   public static void main(String[] args) {
```

```
        // class System is in package java.lang from the java.base module
        System.out.println("Welcome to the Java Platform Module System!");
    }
}
```

Module declaration: module-info.java. The following listing contains the module declaration for the `com.deitel.welcome` module.

```
// module-info.java
// Module declaration for the com.deitel.welcome module
module com.deitel.welcome {
    requires java.base; // implicit in all modules, so can be omitted
}
```

Again, the module declaration begins with the keyword `module` followed by the module's name and braces that enclose the declaration's body. This module declaration contains a `requires` module directive, indicating that the app depends on types defined in module `java.base`. Actually, all modules depend on `java.base`, so the `requires` module directive is *implicit* in all module declarations and may be omitted, as in:

```
module com.deitel.welcome {
}
```

Compiling a module. To compile the `Welcome` app's module, open a command window, use the cd command to change to the `WelcomeApp` folder, then type:

```
javac -d mods/com.deitel.welcome ^
    src/com.deitel.welcome/module-info.java ^
    src/com.deitel.welcome/com/deitel/welcome/Welcome.java
```

[Note: The ^ symbol is the Microsoft Windows line-continuation character. The above command could be entered on a single command line without the line-continuation characters. Linux

and macOS users should replace the carets (^) in the commands with the backslash (\) line-continuation character when breaking the command across multiple lines.] The `-d` option indicates that `javac` should place the compiled code in the specified folder—in this case a `mods` folder that will contain a subfolder named `com.deitel.welcome` representing the compiled module. The name `mods` is used by convention for a folder that contains modules.

Welcome app's folder structure after compilation. If the code compiles correctly, the `WelcomeApp` folder's `mods` subfolder structure contains the compiled code (see **Figure 2**). This is known as the *exploded-module folder*, because the folders and `.class` files are not in a JAR (Java archive) file. The exploded module's structure parallels that of the app's `src` folder described previously. We'll package the app as a JAR shortly. Exploded module folders and modular JAR files together are module artifacts. These can be placed on the module path—a list of module artifact locations—when compiling and executing modularized code.

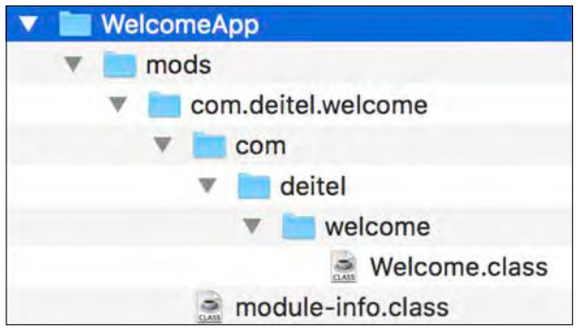


Figure 2. Welcome app's mods folder structure

Viewing the module's description. You can use the `java` command's `--describe-module` option to display information from the `com.deitel.welcome` module descriptor. Run the following command from the `WelcomeApp` directory:

```
java --module-path mods --describe-module com.deitel.welcome
```

The resulting output is:

```
com.deitel.welcome pathContainingModsFolder/mods/com.deitel.welcome/
requires java.base
contains com.deitel.welcome
```

The output begins with the module's name and location. The rest of the output shows that the module requires the standard module `java.base` and contains the package `com.deitel.welcome`. Although the module contains this package, it is *not* exported. Therefore, its contents cannot be used by other modules. The module declaration for this example *explicitly* required `java.base`, so the preceding output included

```
requires java.base
```

If the module declaration had *implicitly* required `java.base`, then the listing instead would have included

```
requires java.base mandated
```

There is no `mandated` module directive—it is included in the `--describe-module` output simply to indicate that all modules depend on `java.base`.

Running an app from a module's exploded folders. To run the `Welcome` app from the module's exploded folders, use the following command from the `WelcomeApp` folder (again, macOS/Linux users should replace ^ with \):

```
java --module-path mods ^
   --module com.deitel.welcome/com.deitel.welcome.Welcome
```

The `--module-path` option specifies where the module is located—in this case, the `mods` folder. The `--module` option specifies the module name and the fully qualified class name of the app's entry point—that is, a class containing `main`. The program executes and displays

```
Welcome to the Java Platform Module System!
```

In the preceding command, `--module-path` can be abbreviated as `-p` and `--module` can be abbreviated as `-m`.

Packaging a module into a modular JAR file. You can use the `jar` command to package an exploded module folder as a *modular JAR file* that contains all of the module's files, including its `module-info.class` file, which is placed in the JAR's root folder. When running the app, you specify the JAR file on the module path. The folder in which you wish to output the JAR file must exist before running the `jar` command.

If a module contains an app's entry point, you can specify that class with the `jar` command's `--main-class` option, as in:

```
jar --create -f jars/com.deitel.welcome.jar ^
   --main-class com.deitel.welcome.Welcome ^
   -C mods/com.deitel.welcome .
```

The options are as follows:

- `--create` specifies that the command should create a new JAR file.
- `-f` specifies the name of the JAR file and is followed by the name—in this case, the file `com.deitel.welcome.jar` will be created in the folder named `jars`.
- `--main-class` specifies the fully qualified name of the app's entry point—a class that contains a `main` method.
- `-C` specifies which folder contains the files that should be included in the JAR file and is followed by the files to include—the dot (.) indicates that all files in the folder should be included.

Running the Welcome app from a modular JAR file. Once you place an app in a modular JAR file for which you've specified the entry point, you can execute the app as follows:

```
java --module-path jars com.deitel.welcome
```

or, using the abbreviated form:

```
java -p jars -m com.deitel.welcome
```

If you did not specify the entry point when creating the JAR, you can still run the app by specifying the module name and fully qualified class name, as in:

```
java --module-path jars ^
   com.deitel.welcome/com.deitel.welcome.Welcome
```

or

```
java -p jars -m com.deitel.welcome/com.deitel.welcome.Welcome
```

Classpath vs. module path. Before Java 9, the compiler and runtime located types via the classpath—a list of folders and library archive files containing compiled Java classes. In earlier Java versions, the classpath was defined by a combination of a `CLASSPATH` environment variable, extensions placed in a special folder of the JRE, and options provided to the `javac` and `java` commands.

Because types could be loaded from several different locations, the order in which those locations were searched resulted in brittle apps. For example, many years ago, I installed a Java app from a third-party vendor on my system. The app's installer placed an old version of a third-party Java library into the JRE's extensions folder. Several other Java apps on my system depended on a newer version of that library with additional types and enhanced versions of the library's older types. Because classes in the JRE's extensions folder were loaded before other classes on the classpath, the apps that depended on the newer library version stopped working, failing at runtime with `NoClassDefFoundErrors` and `NoSuchMethodErrors`—sometimes long after the apps began executing. (For more information on class loading, see "Understanding Extension Class Loading.")

The reliable configuration provided by modules and module descriptors helps eliminate many such runtime classpath problems. Every module explicitly states its

The reliable configuration provided by modules and module descriptors helps eliminate many runtime classpath problems.

dependencies and these are resolved *as an app launches*. The module path may contain only one of each module and every package may be defined in only one module. If two or more modules have the same name or export the same packages, the runtime immediately terminates before running the program. </article>

Paul Deitel, CEO and chief technical officer of Deitel & Associates, is a graduate of MIT with 35 years of experience in computing. He is a Java Champion and has been programming in Java for more than 22 years. He and his coauthor, Dr. Harvey M. Deitel, are the world's best-selling programming-language authors. Paul has delivered Java, Android, iOS, C#, C++, C, and internet programming courses to industry, government, and academic clients internationally.

[This article was adapted from the recently published book *Java 9 for Programmers,* by Paul and Harvey Deitel. The book will be reviewed in a forthcoming issue of *Java Magazine*. —*Ed.*]

learn more

Project Jigsaw: Module System Quick-Start Guide

Paul Deitel's live online course *Introduction to Modularity with the Java 9 Platform Module System (JPMS)*, available at no additional charge to SafariBooksOnline.com subscribers

Java 9 Modularity: Patterns and Practices for Developing Maintainable Applications by Sander Mak and Paul Bakker (O'Reilly Media, 2017)

RAOUL-GABRIEL URMA

RICHARD WARBURTON

Java 9 Core Library Updates: Optionals and CompletableFutures

Changes to Optional and CompletableFuture help you better model error cases for business applications.

In the previous article on Java 9's changes, "Java 9 Core Library Updates: Collections and Streams," we showed that there were lots of goodies that make developers more productive on a day-to-day basis. Java 9 isn't just about big-picture improvements, such as modules and the Java 9 read-eval-print loop (REPL). In this article, we complete the examination of major changes to core libraries by looking at improvements to the Optional and CompletableFuture APIs.

Optional

Optional, a feature introduced in Java 8 to facilitate work with streams, was updated in Java 9. This release introduced the features discussed here: `stream()`, `ifPresentOrElse()`, and `or()`.

stream(). If you've been using Java 8's Stream API in conjunction with the Optional class, you might have encountered a situation in which you wanted to replace a stream of Optionals with values. For example, suppose you have a collection of settings that might have been set by a user. You've implemented the following `lookupSettingByName()` method, which returns an Optional<Setting> if the configuration setting has been set by the user:

```
List<Setting> settings =
   SETTING_NAMES.stream()
                .map(this::lookupSettingByName)
                .filter(Optional::isPresent)
                .map(Optional::get)
                .collect(toList());
```

In this example, you combined two Optional methods to achieve your goal. You used a `filter` on the `isPresent()` method in order to remove empty Optionals. Then you unboxed the Optional objects that you knew had a value with the `get()` method call.

You can make this code less clunky in Java 9. A method on Optional has been added that returns a stream called, funnily enough, `stream()`. It will give you a Stream with an element in it if the Optional has one, or it will be empty otherwise. Let's see how the code looks with this approach:

```
List<Setting> settings =
   SETTING_NAMES.stream()
                .map(this::lookupSettingByName)
                .flatMap(Optional::stream)
                .collect(toList());
```

This new addition also means that it is simpler to integrate Optional with APIs expecting to work with streams.

ifPresentOrElse(). The new `ifPresentOrElse` method encodes a common pattern for performing one action if an Optional value is present or performing a different action if it's absent. To understand this feature, let's take an example of someone trying to check in for an airline flight to see how you would write this code first using null checks and then with the Optional type.

The user provides a booking reference, with which you look up their booking. If you have a booking associated with that reference, you display the check-in page; otherwise, you display a page explaining that the booking record is missing.

If the `lookupBooking` method were to return null in order to indicate that the booking is missing, your code might look like the following:

```
Booking booking = lookupBooking(bookingRef);
if (booking != null) {
   displayCheckIn(booking);
} else {
```

```
    displayMissingBookingPage();
}
```

Now in Java 8, you could have refactored `lookupBooking` to return an Optional value, which would give you an explicit indication whether the booking can be found. The Optional would also help you think about the distinction between the booking being present and absent. This approach is better than checking for null because it forces you to explicitly think in terms of the domain model. The simplest refactoring for this code would have been to the following:

```
Optional<Booking> booking = lookupBooking(bookingRefer);
If (booking.isPresent()) {
    displayCheckIn(booking.get());
} else {
    displayMissingBookingPage();
}
```

This pattern of taking an Optional and calling `isPresent()` and `get()` isn't a particularly idiomatic use, because it makes the code verbose. In fact, it basically leaves you with code that is similar to the null-check version. Ideally, you want to be able to call a method on the Optional that is appropriate for your use case—effectively moving to a tell-don't-ask style of coding. Optional from Java 8 has an `ifPresent` method that will invoke its callback if the value inside the Optional is present, for example:

```
lookupBooking(bookingReference)
    .ifPresent(
        this::displayCheckIn);
```

Unfortunately, this doesn't meet your needs here, because it won't handle the case in which the value is absent and you want to display the "missing booking" page. This is the use case that Java 9's Optional improvements address. You could refactor your original code as follows:

```
lookupBooking(bookingReference)
    .ifPresentOrElse(
        this::displayCheckIn,
        this::displayMissingBookingPage);
```

or(). Another method that has been added to Optional in Java 9 is the succinctly named `or()` method. It takes as an argument a function that creates an Optional. If the object on which it is invoked has a value present, the value is returned; otherwise, the function is called and its result is returned.

This is particularly useful when you have several methods that all return Optionals and you want to return the first one that is present. Suppose you want to look up information about clients using a company identifier, such as the company number. First, you want to check your existing client datastore to see whether the company is in there. If it isn't, you want to create a new client by looking up information about the company in an alternative service.

Perhaps the provided ID is a typo from a user and the ID can't be looked up at all. If you suppose that your methods just return null in order to indicate that the value is missing, you might write the following code:

```
Client client = findClient(companyId);
if (client == null) {
    client = lookupCompanyDetails(companyId);
}
// client could still be null
```

If you refactor `findClient()` to return an Optional, you can use the `orElseGet()` method from Java 8, which will call your `lookupCompanyDetails()` method only if the Optional is absent, for example:

```
Client client =
   findClient(companyId)
```

```
    .orElseGet(() ->
         lookupCompanyDetails(companyId));
```

However, this still does not model your use case correctly. Because the `companyId` might not correspond to an actual company identifier, the `lookupCompanyDetails()` method can still fail. If you take the route of modeling failure using the Optional type, you should also make that method return an Optional.

This is where the new `or()` method comes into play. You get an `Optional<Client>` back. If you could look up the client in your database, then that client would be the value that is present. If it is a valid new company, that will be returned. If it contains a typo, the Optional will be empty. Here's the code:

```
Optional<Client> client =
    findClient(companyId)
    .or(() -> lookupCompanyDetails(companyId));
```

So far, this article has discussed how Optional in Java 8 has been improved in Java 9 with a series of targeted methods that satisfy missing use cases. The primitive, specialized Optional classes—such as `OptionalInt`—aren't getting all the same love: `stream()` and `ifPresentOrElse()` were added to them, but `or()` was not.

CompletableFuture

Java 8 introduced `CompletableFuture<T>` as an enhancement to `Future<T>`. It is a class that lets you express the flow of information from different tasks using a callback-driven style. In the rest of this article, you will see the improvements that Java 9 brings to these Futures. We'll first introduce a typical problem. We'll then show several ways to address the problem in Java 8, and then we'll demonstrate in the final section how Java 9 provides better alternatives.

Combining two services. Suppose that you'd like to combine the result of two services over the network: a best-price finder for a flight route and an exchange service that converts USD

to GBP. Both services introduce a certain delay before responding with a result. This delay is due to the costs of network communication with the service.

You could solve this problem by making use of `CompletableFuture` as follows. (Note for advanced readers: By default a `CompletableFuture` uses the common thread pool, but this can be parameterized with an Executor using an overload of `supplyAsync`.)

```
BigDecimal amount =
    CompletableFuture.supplyAsync(
        () -> findBestPrice("LDN - NYC"))
    .thenCombine(CompletableFuture.supplyAsync(
        () -> queryExchangeRateFor("GBP")),
    this::convert)
    .get();
```

In this code, the method `convert` takes the two `BigDecimal` results from `findBestPrice` and `queryExchangeRateFor` and calculates the final amount.

Timeout mechanism. There are a few problems associated with the code above. First, `get()` is a blocking call. This means that the main thread will need to wait until the result is ready before it can progress. Ideally, you'd like the main thread to do other useful work while the result is calculated in the background. Second, the main thread could be blocking indefinitely, because a timeout is not specified. What if one of the services is overloaded and doesn't respond? To add a timeout mechanism, you can use the other version of `get` inherited from `Future`, which throws a `TimeoutException` when the overall pipeline takes longer than a specified amount of time to return the result:

```
BigDecimal amount =
    CompletableFuture.supplyAsync(
        () -> findBestPrice("LDN - NYC"))
    .thenCombine(CompletableFuture.supplyAsync(
        () -> queryExchangeRateFor("GBP")),
```

```
                this::convert)
    .get(1, TimeUnit.SECONDS);
```

Unfortunately, this reworked code is still blocking, and it prevents the main thread from doing useful work in the meantime. To tackle this issue, you can refactor the code to use `thenAccept` and provide a callback that is executed when the result is finally available, as in the last line of the following code:

```
CompletableFuture.supplyAsync(
        () -> findBestPrice("LDN - NYC"))
    .thenCombine(CompletableFuture.supplyAsync(
        () -> queryExchangeRateFor("GBP")),
            this::convert)
    .thenAccept(amount ->
        System.out.println(
            "The price is: " + amount + "GBP"));
```

However, by using this approach, you lose the timeout functionality. Ideally, you would like to specify a timeout using a nonblocking method. There isn't built-in, elegant support to solve this problem in Java 8. Solutions available in Java 8 are a little clunky and include using `acceptEither()` or `applyToEither` with the `CompletableFuture` on which you are waiting for the result using another `CompletableFuture`. That `CompletableFuture` wraps a `ScheduledThreadPoolExecutor` that throws a `TimeoutException` after a specified time:

```
CompletableFuture.supplyAsync(
  () -> findBestPrice("LDN - NYC"))
.thenCombine(CompletableFuture.supplyAsync(
  () -> queryExchangeRateFor("GBP")),
  this::convert)
.acceptEither(
```

```
  timeoutAfter(1, TimeUnit.SECONDS),
  amount -> System.out.println(
     "The price is: " + amount + "GBP"));
```

A simple implementation of `timeoutAfter` is as follows, where the variable `delayer` stores an instance of a `ScheduledThreadPoolExecutor`:

```
public <T> CompletableFuture<T> timeoutAfter(
    long timeout,
    TimeUnit unit) {
        CompletableFuture<T> result =
            new CompletableFuture<T>();

    delayer.schedule(() ->
        result.completeExceptionally(
            new TimeoutException()), timeout, unit);
    return result;
}
```

The `timeoutAfter` method will return a `CompletableFuture`, which schedules a `TimeoutException` after a specified time. You can then use it in combination with the method `acceptEither` to select whichever `CompletableFuture` is first to complete: the actual service or the timeout. Unfortunately, as you can see, the implementation requires you to write a lot of code.

Java 9 improvements. Java 9's `CompletableFuture` introduces several new methods. Among them are `orTimeout` and `completeOnTimeOut`, which provide built-in support for dealing with timeouts so you don't need to implement them yourself.

The method `orTimeout` has the following signature:

```
public CompletableFuture<T> orTimeout(
    long timeout, TimeUnit unit)
```

It internally uses a `ScheduledThreadExecutor` and completes the `CompletableFuture` with a `TimeoutException` after the specified timeout has elapsed. It also returns another `CompletableFuture`, meaning that you can further chain your computation pipeline and deal with the `TimeoutException` by providing a friendly message back, for example:

```
CompletableFuture.supplyAsync(
        () -> findBestPrice("LDN - NYC"))
    .thenCombine(CompletableFuture.supplyAsync(
        () -> queryExchangeRateFor("GBP")),
           this::convert)
    .orTimeout(1, TimeUnit.SECONDS)
    .whenComplete((amount, error) -> {
         if (error == null) {
             System.out.println(
                 "The price is: " + amount + "GBP");
         } else {
             System.out.println(
                "Sorry, we could not return you a result");
  }
});
```

The method `completeOnTimeout` has the following signature:

```
public CompletableFuture<T> completeOnTimeout(
    T value, long timeout, TimeUnit unit)
```

It also uses a `ScheduledThreadExecutor` internally, but in contrast to `orTimeout`, it provides a default value in the event that the `CompletableFuture` pipeline times out. The `orTimeout` method is conceptually similar to the `orElse()` method using `java.util.Optional`.

In the following code, if the services are slow to respond, a default price is provided; otherwise, the result of combining the two services is returned:

```
CompletableFuture.supplyAsync(
        () -> findBestPrice("LDN - NYC"))
    .thenCombine(CompletableFuture.supplyAsync(
        () -> queryExchangeRateFor("GBP")),
             this::convert)
    .completeOnTimeout(DEFAULT_PRICE, 1, SECONDS)
    .thenAccept(amount -> {
        System.out.println(
             "The price is: " + amount + "GBP");
  });
```

Conclusion

Java 9 helps you get around some clunky patterns when you use the Optional and `CompletableFuture` data types. These changes help you write more-reliable code by enabling you to better model the error cases that real business applications have to deal with. The Optional improvements let you use Optionals in a way that was previously difficult, and the `CompletableFuture` class lets you implement timeouts on external operations very easily. `</article>`

Raoul-Gabriel Urma (@raoulUK) is the CEO and cofounder of Cambridge Spark, a leading learning community for data scientists and developers in the UK. He is also chairman and cofounder of Cambridge Coding Academy, a community of young coders and students. Urma is coauthor of the best-selling programming book *Java 8 in Action* (Manning Publications, 2015). He holds a PhD in computer science from the University of Cambridge.

Richard Warburton (@richardwarburto) is a software engineer, teacher, author, and Java Champion. He is the author of the best-selling *Java 8 Lambdas* (O'Reilly Media, 2014) and helps developers learn via Iteratr Learning and at Pluralsight. Warburton has delivered hundreds of talks and training courses. He holds a PhD from the University of Warwick.

DO YOU LIKE THIS NEW SINGLE-COLUMN FORMAT FOR ARTICLES?
LET US KNOW AT JAVAMAG_US@ORACLE.COM.

BEN EVANS

The Mechanics of Java Method Invocation

Special bytecodes make calling methods particularly efficient. Knowing how they operate reveals how the JVM executes your code.

In this article, I explain how the JVM executes methods in Java 8 and Java 9. This is a fundamental topic about the internals of the JVM that is essential background for anyone who wants to understand the JVM's just-in-time (JIT) compiler or tune the execution of applications.

Examining Some Bytecode

To get started, let's look at a simple bit of Java code:

```
long time = System.currentTimeMillis();
HashMap<String, String> hm = new HashMap<>();
hm.put("now", "bar");
Map<String, String> m = hm;
m.put("foo", "baz");
```

To see the bytecode that the Java compiler produces for this code, use the `javap` tool with the `-c` switch to display the decompiled code:

```
0: invokestatic #2 // Method
                   java/lang/System.currentTimeMillis:()J
3: lstore_1
4: new #3 // class java/util/HashMap
7: dup
```

PHOTOGRAPH BY JOHN BLYTHE

```
8: invokespecial #4 // Method java/util/HashMap."<init>":()V
11: astore_3
12: aload_3
13: ldc #5 // String now
15: ldc #6 // String bar
17: invokevirtual #7 // Method java/util/HashMap.put:
                     (Ljava/lang/Object; Ljava/lang/Object;)
                     Ljava/lang/Object;
20: pop
21: aload_3
22: astore 4
24: aload 4
26: ldc #8 // String foo
28: ldc #9 // String baz
30: invokeinterface #10, 3 // InterfaceMethod java/util/Map.put:
                           (Ljava/lang/Object;Ljava/lang/Object;)
                           Ljava/lang/Object;
35: pop
```

[Indented lines are continued from the previous line. —*Ed.*] Java programmers who are new to looking at code at the JVM level might be surprised to learn that Java method calls are actually turned into one of several possible bytecodes of the form `invoke*`.

Let's take a closer look at the first part of the decompiled code:

```
0: invokestatic #2 // Method java/lang/System.currentTimeMillis:()J
3: lstore_1
```

The static call to `System.currentTimeMillis()` is turned into an `invokestatic` opcode that appears at position `0` in the bytecode. This method takes no parameters, so nothing needs to be loaded onto the evaluation stack before the call is dispatched.

Next, the two bytes `00 02` appear in the byte stream. These are combined into a 16-bit number (`#2`, in this case) that is used as an offset into a table—called the *constant pool*—within the class file. All constant pool indices are 16 bits, so whenever an opcode needs to refer to an entry in the pool, there will always be two subsequent bytes that encode the offset of the entry.

The decompiler helpfully includes a comment that lets you know which method offset `#2` corresponds to. In this case, as expected, it's the method `System.currentTimeMillis()`. In the decompiled output, `javap` shows the name of the called method, the types of parameters the method takes (in parentheses), followed by the return type of the method.

Upon return, the result of the call is placed on the stack, and at offset `3` you see the single, argument-less opcode `lstore_1`, which saves the return value in a local variable of type `long`.

Human readers are, of course, able to see that this value is never used again. However, one of the design goals of the Java compiler is to represent the contents of the Java source code as faithfully as possible—whether it makes logical sense or not. Therefore, the return value of `System.currentTimeMillis()` is stored, even though it is not used after this point in the program.

Let's look at the next chunk of the decompiled code:

```
4: new #3 // class java/util/HashMap
7: dup
8: invokespecial #4 // Method java/util/HashMap."<init>":()V
11: astore_3
12: aload_3
13: ldc #5 // String now
15: ldc #6 // String bar
17: invokevirtual #7 // Method java/util/HashMap.put:
                        (Ljava/lang/Object;Ljava/lang/Object;)
                        Ljava/lang/Object;
20: pop
```

Bytecodes `4` to `10` create a new `HashMap` instance, before instruction `11` saves a copy of it in a local

variable. Next, instructions 12 to 16 set up the stack with the `HashMap` object and the arguments for the call to `put()`. The actual invocation of the `put()` method is performed by instructions 17 to 19.

The `invoke` opcode used this time is `invokevirtual`. This differs from a static call, because a static call does not have an instance on which the method is called; such an instance is sometimes called the *receiver* object. (In bytecode, an instance call must be set up by placing the receiver and any call arguments on the evaluation stack and then issuing the `invoke` instruction.) In this case, the return value from `put()` is not used, so instruction 20 discards it.

The sequence of bytes from 21 to 25 seems rather odd at first glance:

```
21: aload_3
22: astore 4
24: aload 4
26: ldc #8 // String foo
28: ldc #9 // String baz
30: invokeinterface #10, 3 // InterfaceMethod java/util/Map.put:
                              (Ljava/lang/Object;Ljava/lang/Object;)
                              Ljava/lang/Object;
35: pop
```

The `HashMap` instance that was created at bytecode 4 and saved to a local variable 3 at instruction 11 is now loaded back onto the stack, and then a copy of the reference is saved to local variable 4. This process removes it from the stack, so it must be reloaded (from variable 4) before use.

This shuffling occurs because in the original Java code, an additional local variable (of type `Map` rather than `HashMap`) is created, even though it always refers to the same object as the original variable. This is another example of the bytecode staying as close as possible to the original source code. One of the main reasons for this so-called "dumb bytecode" approach that Java takes is to provide a simple input format for the JVM's JIT compiler.

After the stack and variable shuffling, the values to be placed in the `Map` are loaded at instructions 26 to 29. Now that the stack has been prepared with the receiver and the arguments, the call to `put()` is dispatched at instruction 30. This time, the opcode is

invokeinterface—even though the same method is being called. Once again, the return value from `put()` is discarded, via the pop at instruction 35.

So far, you've seen that invokestatic, invokevirtual, or invokeinterface can be produced by the Java compiler, depending on the context of the call.

JVM Bytecodes for Invoking Methods

Let's take a look at all the five JVM bytecodes that can be used to invoke methods (see **Table 1**). In each case, the bytes b0 and b1 are combined into the constant pool offset represented by c1.

OPCODE NAME	ARGUMENTS	DESCRIPTION
invokevirtual	b0 b1	INVOKES THE METHOD FOUND AT CP#C1 VIA VIRTUAL DISPATCH
invokespecial	b0 b1	INVOKES THE METHOD FOUND AT CP#C1 VIA "SPECIAL," THAT IS, EXACT, DISPATCH
invokeinterface	b0 b1 x0 00	INVOKES THE INTERFACE METHOD FOUND AT CP#C1 USING INTERFACE OFFSET LOOKUP
invokestatic	b0 b1	INVOKES THE STATIC METHOD FOUND AT CP#C1
invokedynamic	b0 b1 00 00	DYNAMICALLY LOOKS UP WHICH METHOD TO INVOKE AND CALLS IT

Table 1. JVM bytecodes for invoke methods

It can be a useful exercise to write some Java code to see what circumstances produce each form of the bytecodes by disassembling the resulting Java class with javap.

The most common type of method invocation is invokevirtual, which refers to virtual dispatch. The term *virtual dispatch* means that the exact method to be invoked is determined at runtime. To facilitate this, you need to know that each class present in a running application has an area of memory inside the JVM that holds metadata corresponding to that type. This area is called a *klass* (in Java HotSpot VM, at least) and can be thought of as the JVM's representation of information about the type.

In Java 7 and earlier, the klass metadata lived in an area of the Java heap called *permgen*. Because objects within the Java heap must have an object header (called an *oop*), the klasses

were known as *klassOops*. In Java 8 and Java 9, the klass metadata was moved out of the Java heap into the native heap, so the object headers are no longer required. Some of the information from the klass is available to Java programmers via the `Class<?>` object corresponding to the type—but they are separate concepts.

One of the most important areas of the klass is the *vtable*. This area is essentially a table of function pointers that point to the implementations of methods defined by the type. When an instance method is called via `invokevirtual`, the JVM consults the vtable to see exactly which code needs to be executed. If a klass does not have a definition for the method, the JVM follows a pointer to the klass corresponding to the superclass and tries again.

This process is the basis of method overriding in the JVM. To make the process efficient, the vtables are laid out in a specific way. Each klass lays out its vtable so that the first methods to appear are the methods that the parent type defines. These methods are laid out in the exact order that the parent type used. The methods that are new to this type and are not declared by the parent class come at the end of the vtable.

This means that when a subclass overrides a method, it will be at the same offset in the vtable as the implementation being overridden. This makes the lookup of overridden methods completely trivial, because their offset in the vtable will be the same as the offset of their parent.

Figure 1 shows an example defined by the classes `Pet`, `Cat`, and `Bear` and the interface `Furry`.

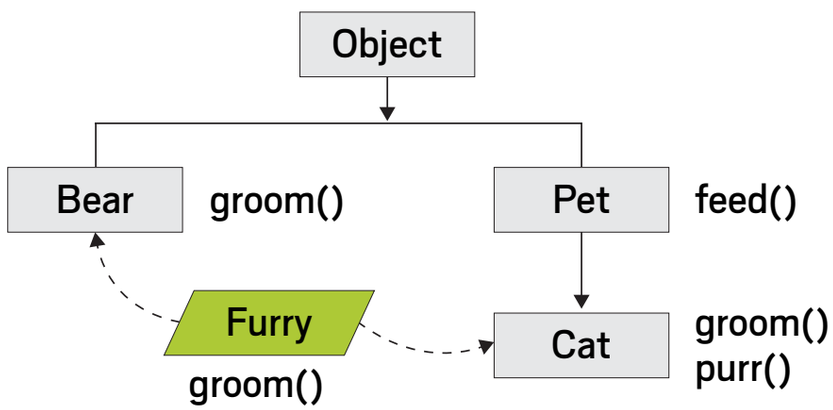


Figure 1. Simple inheritance hierarchy

The vtables for these classes are laid out in Java 7, as shown in **Figure 2**. As you can see, this figure shows the Java 7 layout within permgen, so it refers to klassOops and has the two words of the object header (shown as `m` and `kk` in the figure). As discussed previously, these entries would not be present in Java 8 and Java 9, but all else in the diagram remains the same.

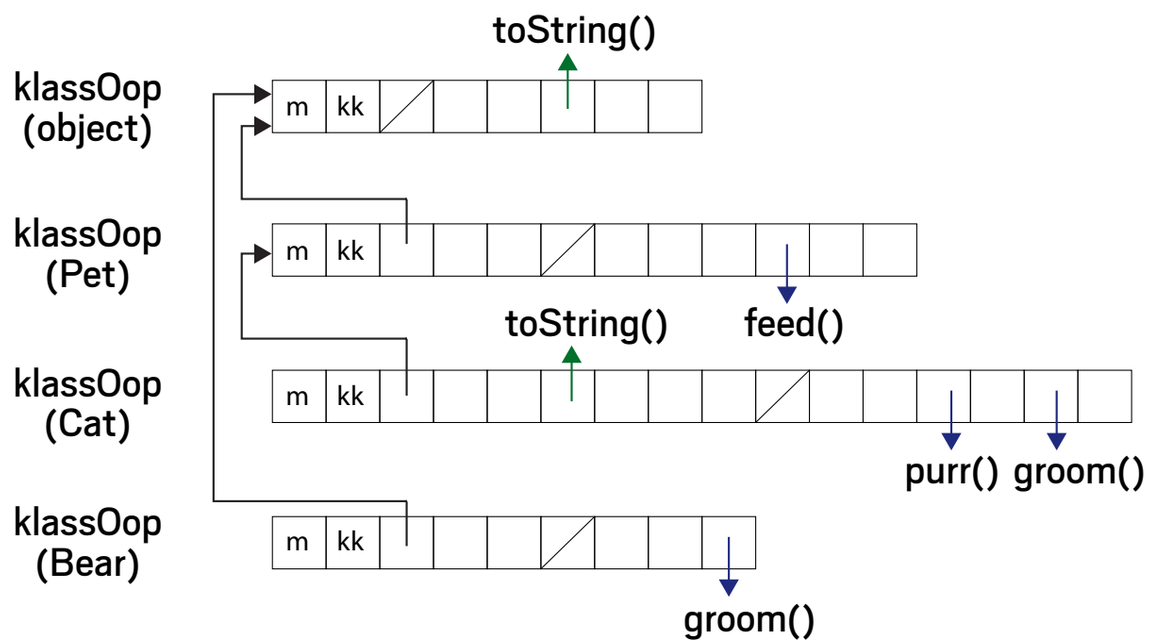


Figure 2. Structure of the vtables for the classes shown in Figure 1

If you call `Cat::feed`, the JVM will not find an override in the `Cat` class and instead will follow the pointer to the klass of `Pet`. This klass does have an implementation for `feed()`, so this is the code that will be called. This vtable structure works well because Java implements only single inheritance of classes. This means there is only one direct superclass of any type (except for `Object`, which has no superclass).

In the case of `invokeinterface`, the situation is a little more complicated. For example, the `groom()` method will not necessarily appear in the same place in the vtable for every implementation of `Furry`. The different offsets for `Cat::groom` and `Bear::groom` are caused by the fact that their class inheritance hierarchies differ. The result of this is that some additional lookup

is needed when a method is invoked on an object for which only the interface type is known at compile time.

Note that even though slightly more work is done for the lookup of an interface call, you should not try to micro-optimize by avoiding interfaces. Remember that the JVM has a JIT compiler, and it will essentially eliminate any performance difference between the two cases.

Example of Invoking Methods

Let's look at another example. Consider this bit of code:

```
Cat tom = new Cat();
Bear pooh = new Bear();
Furry f;

tom.groom();
pooh.groom();

f = tom;
f.groom();

f = pooh;
f.groom();
```

This code produces the following bytecodes:

```
Code:
0: new           #2            // class scratch/Cat
3: dup
4: invokespecial #3            // Method scratch/Cat."<init>":()V
7: astore_1
8: new           #4            // class scratch/Bear
```

```
11: dup
12: invokespecial #5          // Method scratch/Bear."<init>":()V
15: astore_2
16: aload_1
17: invokevirtual #6          // Method scratch/Cat.groom:()V
20: aload_2
21: invokevirtual #7          // Method scratch/Bear.groom:()V
24: aload_1
25: astore_3
26: aload_3
27: invokeinterface #8,  1    // InterfaceMethod scratch/Furry.groom:()V
32: aload_2
33: astore_3
34: aload_3
35: invokeinterface #8,  1    // InterfaceMethod scratch/Furry.groom:()V
```

The two calls at 27 and 35 look like they are the same, but they actually invoke different methods. The call at 27 will invoke `Cat::groom`, whereas the call at 35 will invoke `Bear::groom`.

With this background on `invokevirtual` and `invokeinterface`, the behavior of `invokespecial` is now easier to understand. If a method is invoked by `invokespecial`, it does not undergo virtual lookup. Instead, the JVM will look only in the exact place in the vtable for the requested method. This means that an `invokespecial` is used for three cases: private methods, calls to a superclass method, and calls to the constructor body (which is turned into a method called `<init>` in bytecode). In all three cases, virtual lookup and the possibility of overriding must be explicitly excluded.

Final Methods

There remains one corner case that should be mentioned: the case of final methods. At first glance, it might appear that calls to final methods would also be turned into `invokespecial` instructions. However, section 13.4.17 of the *Java Language Specification* has something to say

about this case: “Changing a method that is declared final to no longer be declared final does not break compatibility with pre-existing binaries.”

Suppose a compiler had compiled a call to a final method into an `invokespecial`. If the method then changed to no longer be final, it could be overridden in a subclass. Now, suppose that an instance of the subclass were passed into the compiled code. The `invokespecial` would be executed, and then the wrong implementation of the method would be called. This would be a violation of the rules of Java’s object orientation (strictly speaking, it would violate the Liskov Substitution Principle).

For this reason, calls to final methods must be compiled into `invokevirtual` instructions. In practice, the Java HotSpot VM contains optimizations that allow final methods to be detected and executed extremely efficiently.

Conclusion

I have now examined four of the five invocation instructions that the JVM supports. The remaining case is `invokedynamic`, and it is such a rich and interesting subject that it requires an article all to itself. The second article in this two-part series will be devoted to covering `invokedynamic` and related subjects. `</article>`

Ben Evans (@kittylyst) is a Java Champion, tech fellow and founder at jClarity, an organizer for the London Java Community (LJC), and a member of the Java SE/EE Executive Committee. He is the author of four books, including the forthcoming *Optimizing Java* (O’Reilly).

learn more

- javap
- JVM instruction set for Java

JavaOne™

Oct. 1–5, 2017 | San Francisco

REGISTER NOW

Save $200 by Sept. 29*

oracle.com/javaone

// Innovation Sponsor

// Diamond Sponsor

// Gold Sponsor

ORACLE®

*Discount based on the onsite registration price.

ANDRÉS ALMIRAY

Define Custom Behavior in FXML with FXMLLoader

Inject custom behavior into JavaFX applications using FXML.

In the first article in this series, I discussed the basics of the FXML format and a handy utility named FXMLLoader, which, as the name implies, reads an FXML resource, parses its contents, and constructs a SceneGraph based on the definitions found in the FXML file. However, FXMLLoader can do more than just that, especially when it is combined with a common technique of modern Java development: dependency injection (DI). If you're not familiar with FXMLLoader, it might be difficult to follow this article without first reading the earlier article.

The designers of the FXML format recognized that developers would like to tweak some settings using a programmatic approach, such as inserting a dynamic set of items to a `ListView` or a `TableView` right after the UI has been created but before the user has any chance to interact with the application. To perform such a task, JavaFX developers must be able define custom behavior that should be called at a specific time during the application's initialization. They also need a way to find the target widget (the `ListView`, for example) on which the customizations will take place. This behavior sounds very similar to what modern DI frameworks provide, and that's precisely what the designers of FXML allow you to do. Let's see how it's done.

The Controller

The JavaFX views you saw in the first article were devoid of any application-specific behavior, and that's a good thing, because it keeps responsibilities separated. View elements should be responsible for defining how the UI looks but not how it behaves; that's the responsibility of the controller part. In FXML terms, a controller is an object that participates in DI and can react

to events triggered by UI elements defined by its associated view. Unlike the main entry point of a JavaFX application, where you have to extend a specific class (`javafx.application.Application`), a controller class may be of any type; there are also no marker interfaces that need to be implemented. This gives you free range to develop your own types and controller hierarchies as needed.

Let's begin with a simple application that has a UI that comprises a button and a `ListView` whose contents will be initialized programmatically. A new item will be appended to the list every time the button is clicked. The FXML definition for such a UI looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<?import javafx.scene.control.Button?>
<?import javafx.scene.control.ListView?>
<?import javafx.scene.layout.VBox?>
<VBox xmlns="http://javafx.com/javafx"
    xmlns:fx="http://javafx.com/fxml"
    fx:controller="sample.AppController"
    prefHeight="300.0" prefWidth="100.0">
    <Button text="Add item" onAction="#addItem"/>
    <ListView fx:id="theList"/>
</VBox>
```

There are new features in this file. The first is the `fx:controller` attribute applied to the root element of the UI. This attribute instructs FXMLLoader to instantiate an object of the specified class and set it as the controller for this view. An object instantiated in this way requires its class to have a public, no-argument constructor.

The second feature is found in the button definition: there's an extra attribute named `onAction`. Take special note of the format of its value; it's the `#` character followed by an identifier, which turns out to be a method defined in the controller class.

Finally, note the usage of `fx:id` on `ListView`; this instructs FXMLLoader to keep an internal reference to the `ListView`, using the attribute's name as a variable name. This variable will be

used to inject the ListView reference into the controller. Let's take a moment and see how the controller looks:

```
// sample/AppController.java
package sample;

import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.scene.control.ListView;

public class AppController {
    @FXML private ListView<String> theList;

    public void initialize() {
        for (int i = 0; i < 5; i++) {
            theList.getItems().add("Item " + i);
        }
    }

    public void addItem(ActionEvent ignored) {
        theList.getItems().add("Item "
            + theList.getItems().size());
    }
}
```

As you can see, the controller looks like a regular Java class; however, there's a new element in use: the `@FXML` annotation. This is the necessary link to enable FXMLLoader to perform DI on the target controller instance. The use of this annotation takes advantage of the variable name defined for the `ListView` reference. Notice that the field name is the same as the variable name; it also defines the type of elements the `ListView` can contain: in this case, a plain

Java string. Following this naming convention, FXMLLoader will match field names to variable names and match field types to widget types. Thus, you'll get a runtime exception if the field were to be defined as `Choicebox<String>` instead. This potential error can be avoided if your IDE of choice is aware of FXML conventions.

JSR 330 defines the basic principles for performing dependency injection in a Java program.

The second feature is the definition of a public method whose name must be `initialize` and which takes no arguments. This method is invoked by FXMLLoader after all widget injections have been performed. This is the right time to perform data initialization, because the widget references have been fully resolved and injected up to this point. That's precisely what happens in this example by filling the `ListView` with default data. If the method were to be named something else, FXMLLoader would not call it. This is a hard constraint defined by the FXMLLoader conventions.

Finally, you catch a glimpse of an action handler, defined with a method whose name matches the value of the `onAction` handler associated with the button—as you saw in the FXML file. This handler reacts to the button being clicked; thus, it takes an `ActionEvent` as argument. Make note of this type, because it will change depending on the type of event you want the code to react to. For example, the argument should be of type `MouseEvent` if the code should react to mouse movements. This event handler is called immediately as the button is clicked, and it's invoked inside the UI thread as well. That's not a problem for this trivial application, because the handler simply adds a new item to the list. However, be aware of this setup, because typically an application would open a network connection, access a database or file, or perform all sorts of computations that must be executed outside the UI thread. When data is ready, you'll have to push it back to the UI elements inside the UI thread. So, be aware of the threading constructs provided by JavaFX; if threading is done incorrectly, you might get runtime exceptions, an unresponsive application, data corruption, or worse.

In summary, a controller is responsible for providing behavior to an FXML view, and you establish a relationship between them using the `fx:controller` attribute on the root node of an

FXML view. The FXMLLoader instantiates the controller automatically, and that's OK most of the time. However, what if you need the controller to be instantiated in a different way? Perhaps you have additional customizations that need to be done before all `@FXML`-annotated elements have been injected. Don't worry; you can use a DI framework for this task. The next section shows how to do this.

JSR 330 and Dependency Injection

JSR 330 defines the basic principles for performing DI in a Java program. There are a handful of implementations out there. For simplicity's sake, I won't show the setup of a specific DI framework, but rather I'll use the standard API provided by JSR 330 for the following examples.

Let's redo the previous example using a data-provider mechanism instead of inserting data into the `ListView` using a simple `for` loop. This data-provider mechanism is defined using an interface such as this:

```
// sample/DataProvider.java
package sample;

import java.util.Collection;

public interface DataProvider<T> {
    Collection<T> getData();
}
```

Now, you're free to implement this interface in any way required by the application, such as grabbing the data from a file or a database connection via JDBC or other means. The updated controller class now looks like this:

```
//sample/AppController.java
package sample;
```

```
import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.scene.control.ListView;

import javax.inject.Inject;

public class AppController {
    @FXML private ListView<String> theList;

    private final DataProvider<String> dataProvider;

    @Inject
    public AppController(DataProvider<String> dp) {
        this.dataProvider = dp;
    }

    public void initialize() {
        theList.getItems().addAll(dataProvider.getData());
    }

    public void addItem(ActionEvent ignored) {
         theList.getItems().add("Item "
           + theList.getItems().size());
    }
}
```

Notice that the controller defines a constructor with a single argument and is also annotated with `@Inject`. This simple change contradicts the FXMLLoader conventions for instantiating a controller. You must find another way to let FXMLLoader work with the controller class. Luckily there is such a way. FXMLLoader can be configured to use a different strategy to instantiate the

controller: by setting a `ControllerFactory`. Armed with this knowledge, you only need to go back to the application's entry point and configure the FXMLLoader instance with this new information, for example:

```
//sample/AppMain.java
package sample;

import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.scene.Scene;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;

import java.net.URL;

public class AppMain extends Application {
    @Override
    public void start(Stage primaryStage) throws Exception {
        // Initialized via DI framework
        AppController controller = ...
        URL location = getClass().getResource("app.fxml");
        FXMLLoader fxmlLoader = new FXMLLoader(location);

        // set the ControllerFactory
        fxmlLoader.setControllerFactory(param -> controller);

        VBox vbox = fxmlLoader.load();
        Scene scene = new Scene(vbox);
        primaryStage.setScene(scene);
        primaryStage.sizeToScene();
        primaryStage.show();
```

```
    }
}
```

There you have it. The controller instance is fetched from the DI container. This fetch operation varies from framework to framework and, thus, is omitted from the example. You can even apply a lambda expression to define a simple `ControllerFactory`. Given that the `AppController` class is now managed by the DI container, you can leverage other features such as methods annotated with `@PostConstruct`, thus allowing further customizations on the controller. Just remember that any methods annotated with `@PostConstruct` will be invoked before any `@FXML`-annotated fields are injected into the controller instance. As a side effect of this setup, the value of `fx:controller` becomes irrelevant. However, it is still a good idea to specify the actual type of the controller supplied by the DI container, because your IDE will give you hints on the types and names of widgets that can be injected using the `@FXML` annotations, as well as the names of event handlers supplied by the controller. As an alternative, you could set the controller property directly in FXMLLoader, but I decided to show the factory approach, because that might come in handy for lazily calculating the value.

FXML provides a declarative way to define JavaFX UIs, but it can't be used to define behavior.

Up to this point, you've seen views and controllers. But as you know, there are several design patterns that can be used to design a UI application, most of them being a variation of the model-view-controller (MVC) pattern, such as Model-View-Presenter and Model-View-View. These patterns make an explicit differentiation of each member by responsibility, which keeps the implementation neat and tidy, making it easy to find your way around the application.

You might be wondering where the model could be in the examples you've seen. The answer is that it's implicit inside the controller. If you were to keep a state, such as an `ObservableList<String>` of all items, it's likely that the first place to put it would be in the controller itself. For a small application such as this one, this decision might not be wrong, but as

the application grows in complexity and features, you will quickly realize that this decision does not scale well. Thus, I highly recommend you think in terms of the MVC pattern and its variants and, therefore, place behavior in the controller only, leaving data and state to a model class. Instances of this class may be created manually or via a DI container; it's up to you to decide, given the particular application requirements.

Conclusion

FXML provides a declarative way to define JavaFX UIs, but it can't be used to define behavior. A controller can be specified to provide the required behavior. Controllers can participate in DI in order to reach the UI components they need to configure. Controllers can also define event handlers that can be hooked directly into UI components using a special format in the FXML file. FXMLLoader is capable of instantiating controllers using a default strategy; however, this strategy can be changed if a different instantiation mechanism is needed. </article>

Andrés Almiray is a Java and Groovy developer and a Java Champion with more than 17 years of experience in software design and development. He has been involved in web and desktop application development since the early days of Java. He is a true believer in open source and has participated in popular projects such as Groovy, JMatter, Asciidoctor, and others. He's a founding member and current project lead of the Griffon framework, and he is the specification lead for JSR 377.

DO YOU LIKE THIS NEW SINGLE-COLUMN FORMAT FOR ARTICLES?
LET US KNOW AT JAVAMAG_US@ORACLE.COM.

learn more

Oracle's tutorial on FXML

BRAD CYPERT

Clojure

The elegance of Lisp with the performance of the JVM

Clojure is a dynamic, functional, general purpose JVM-based programming language that combines the approachability and interactive development of a scripting language with an efficient and robust infrastructure for multithreaded programming. Clojure was my first exposure to the Lisp dialects of languages, and despite some of the common complaints about its frequent use of parentheses, I felt that the language itself was both beautiful to write and elegant to read. Let's take a look at the classic Hello World code:

```
(ns demo.hello-world)

; this is a comment
(println "Hello World")
```

To run this, you need to have Clojure's JAR file. You can get it here or from Maven. Once you've downloaded it, save your code in hello.clj and run the following:

```
java -cp clojure.jar clojure.main hello.clj
> Hello World
```

I use comments in my source examples under the expressions to show you the output or return value of that expression. In the example above, I'm defining a new namespace called `demo.hello-world` and, in that namespace, I'm printing `Hello World`. That's fairly simple, isn't it?

While this is a noble first step, printing a statement simply doesn't show the power of most languages—Clojure included. Perhaps a more challenging problem is in order. Assume you have

a list of integers and you want to double all of the integers and then sum them. You might notice that this is simply a map-and-reduce problem. Syntactically, Clojure expresses that exact idea:

```
(ns demo.hello-lists)

(def my-list '(1 2 3 4 5))

(reduce
  +
  (map #(* % 2) my-list))
; 30
```

Clojure has support for an idea known as a keyword, which should not be confused with the concept of reserved keywords in other languages including Java (such as static or final).

The code above demonstrates a few simple ideas:

- I define a new list containing the values 1, 2, 3, 4, and 5. In Clojure, a list can be defined using a single quote, followed by a pair of parentheses containing the values you want in your list.
- I write an expression to handle the reduction. `reduce` takes a function and a list. I pass into it the function bound to the `+` symbol and the list returned from the `map` function.
- Using the list defined on the first line, I map over each value and multiply it by two. I handle this using an anonymous function provided as the first argument to `map`. The `%` is a placeholder the argument passed into it. With multiple arguments, you can index them such as `%1` and `%2`.

Like most functional languages, Clojure takes a high-level approach to problems. Instead of requiring code to process each individual step, functional languages are often seen as a way to write code that explains the problem you're trying to solve instead of worrying about the implementation details, such as iterating over a list.

Clojure for the Java Developer

Despite being a compiled language, Clojure brings a form of dynamism to the JVM that allows it to feel like a powerful scripting language. Every feature of Clojure is supported at runtime as well as at compile time. Additionally, with Clojure, you won't have to leave the comfort of your favorite Java libraries: Clojure has full support for Java interoperability, which allows you to

leverage existing Java code. This also means that there is no need to commit to a full rewrite if you'd like to try the language—you can write Clojure code alongside Java code and vice versa.

Immutability Is Fundamental

As a functional programming language, Clojure features a rich set of immutable and persistent data structures. If you find yourself in need of mutable state, you can leverage Clojure's software transactional memory system and reactive agent system that ensure clean, correct, multithreaded designs.

On the subject of multithreading, Clojure offers its async toolkit, known as `core.async`, as part of the standard library. Because it is loosely based on Go's concurrency model, Clojure leads you to work intimately with channels. Channels in Clojure operate with two principal activities: you can take values from a channel and put values into a channel.

Digging into Clojure's async merits an entire article, if not a book. I won't go into async in depth here, but I would like to reiterate that Clojure's async library is an elegant abstraction over threads.

Tooling and Community

Clojure's community is spearheaded by Rich Hickey, the inventor of Clojure and the CTO of Cognitect. Hickey is an experienced Java developer and has delivered numerous well-received talks, including especially his "Simple Made Easy" talk. In fact, the community at large has adopted this "Simple Made Easy" mantra when it comes to building libraries, helping newcomers, and fostering creativity. With this idea in mind, you'll notice "frameworks" tend to be avoided in the Clojure community. It's far more common to tell newcomers to piece together a few libraries to fit the need than to point them to the next big framework. As a newcomer to the language, this can seem intimidating, especially if you've used large-scale frameworks. However, with a bit of time and dedication, you'll not only be able to solve the problems you were looking to solve but you'll have a stronger grasp on Clojure itself.

A very popular tool built by the community is Clojure's de facto build tool, Leinengen. Leinengen aims to solve a similar problem that Gradle has solved for Java, but it does much

more. It houses template projects to help you get started on a project quickly, and it even offers support for project add-ons called *profiles*. For example, if I wanted to scaffold a new project via the luminus template with support for PostgreSQL, I could simply open my terminal and run this:

```
lein new luminus myapp +postgres
```

It's important to note that each template hosts its own profiles, so you might not be able to scaffold with the postgres profile if you're using a different template.

Leinengen takes care of creating a `project.clj` file for you, which is similar to `build.gradle` or `package.json` files. In it, you can define your dependencies, set up your project's metadata, leverage Leinengen plugins, and create your build pipeline.

Leinengen also features a rich read-eval-print loop (REPL). If you're familiar with Python or Node development, you've probably found yourself testing out ideas via the Python integrated development and learning environment (IDLE) or the Node REPL. Leinengen's REPL is a similar idea. Once you have Leinengen installed, you can test new Clojure ideas with ease: simply run `lein repl` from your command line. You'll be placed into a clean Clojure environment where you can try new ideas.

Testing in Clojure

Clojure ships with a lightweight testing library under the clojure.test namespace. It is built around the `is` macro, which allows you to define assertions of arbitrary expressions. Clojure gives you two patterns for documenting test cases: the first is inline with your source code using the `with-test` macro, and the second allows you to place your tests in separate namespaces via the `deftest` macro. Testing a simple arithmetic expression can be as simple as this:

```
(ns demo.hello-tests)

(deftest math-tests
```

```
  (is (= 7 (+ 4 3)))
```

Ideally, your tests aren't simply testing the boundaries of a computer's most fundamental actions, but they are instead testing business logic. Possibly a better example is a test case that's run in its own namespace, such as this:

```
(ns app.test.modules.users
  (:require [clojure.test :refer :all]
            [app.modules.users :refer [get-user]]))

(deftest user-tests
  (testing "Fetching a user"
    (let [user (get-user 1)]
      (is (some? (:email user))))))
```

The code above is the biggest example yet, so let's look at it in more detail. All it is doing is setting up a new namespace and defining a single test that ensures that when I call `get-user` with a key of 1, I get back a data structure with an email defined. It's important to note that `get-user` returns a key-value pair consisting of a key containing "email" and a value containing the email address.

To do this

- I define a new namespace, `app.test.modules.users`.
- This namespace has some imports. It needs access to `clojure.test` and `app.modules.users`. You'll notice that I refer all of the functions in `clojure.test`; however, I can choose to expose only what I plan to test as well, as seen in `app.modules.users`.
- I define a new test called `user-tests`.
- I use the `testing` macro to create a new scope and give this test context a label of `Fetching a user`.
- I use a `let` expression to bind the key-value pair from `(get-user 1)` to a local binding named `user`.

- Finally, I define my assertion via the `is` macro. This macro's body contains the `some?` function, which returns `true` if the value passed to it is not `nil` (a value representing null) or `false` otherwise. Lastly, I take the user data structure that I bound earlier and look up the value in the `:email` key.

What's with the Colon?

Clojure has support for an idea known as a *keyword*, which should not be confused with the concept of reserved keywords in other languages including Java (such as static or final). Keywords are a great way to write expressive maps or pattern matching conditionals. Plus, when they are output, they evaluate to a string version of themselves, minus the colon. When you write Clojure code, you might find yourself using keywords frequently. Keywords help me express and declare the intentions of the code that I write. You can use keywords to define the keys in a map of user data or the values of a list:

```
(ns demo.hello-keywords)

(def brads-details
  {:name "Brad"
   :languages '(:clojure :java :javascript)})
```

Keywords are often compared to enums in Java. Sometimes they're used in the way that enums would be used, but it's also worth mentioning that there is no requirement that a keyword be defined at compile time or runtime. You can actually even create a keyword out of a string during runtime using the "keyword" function. In the previous example, you'll notice that the keywords `name`, `languages`, `clojure`, `java`, and `javascript` are all introduced by a colon.

Functional Concepts

Clojure is predominantly a functional programming language. As such, it treats its code as a set of mathematical expressions and attempts to avoid changing state or mutating data. Clojure

approaches this paradigm through a very common functional toolkit consisting of immutable, persistent data structures and first-class functions. Let's compare this approach with a map in Java. Here is the Java code:

```
import java.util.HashMap;

HashMap<String, Integer> map = new HashMap<>();
map.put("java", 7);
System.out.println(map);
map.put("java", 8);
System.out.println(map);
```

I'm sure what this is doing is rather obvious. When it prints the map the first time, you have a key named `java` and a value of 7. When it prints after putting a new value in the map with the `java` key, it prints a map with a key named `java` and a value of 8. In Clojure, this operation would appear like this:

```
(ns demo.hello-immutability)
(def map {:java 7})
(println map)
(assoc map :java 8)
(println map)
```

In Clojure, this code will print `{:java 7}` both times. This is because maps, like most data structures in Clojure, are immutable. The line `(assoc map :java 8)` actually returns a new map that is cloned from the original, and it would look like this: `{:java 8}`. Immutability is a core concept of the language that allows Clojure to avoid common parallel programming pitfalls such as deadlocks or data races.

Functions in Clojure are first-class. This means that they can be stored in data structures or passed around as arguments. In fact, function application—the idea of applying a function to all

values in a list—is handled with a function that accepts another function and a list as an argument. For example, you can use `apply` to apply a function to each value in a given list. You can choose to use an anonymous function as the parameter or one already bound in a namespace or local binding. You could write the initial example, which doubled values, in one of these two ways.

```
(ns demo.hello-apply)

(def double [x] (* x 2))

(apply double (list 1 2 3 4))
; (2 4 6 8)

(apply #(* % 2) (list 1 2 3 4))
; (2 4 6 8)
```

If you've worked with Lisp or a metalanguage before, you've probably come to expect a certain finesse from functional programming languages known as *destructuring*. It is a convenient way of extracting multiple values from data stored in (possibly nested) key-value pairs and lists. Essentially, destructuring allows you to declaratively define the pieces of data from a data structure that you'd like to use or bind. Destructuring is offered by Clojure out of the box, and it makes working with complex data structures much easier. If you have a function that takes in a list of keywords and returns the first keyword as a string, you might write something like this:

```
(ns demo.hello-transform)

(defn first-keyword->string
  [my-list]
  (name
    (first my-list)))
```

Notice the expressiveness of the function name. Allowing special characters (such as the hyphen or greater-than symbol) in function names enables you to write concise yet expressive function names especially for transforming data from one structure to another. My function takes in a single parameter, `my-list`, and calls the `name` function, which converts its argument to a string. In this case, the argument is the first item from `my-list`.

I could rewrite this code using destructuring as well:

```
(ns demo.hello-transform)

(defn first-keyword->string
  [[head tail]]
  (name head))
```

This code is quite a bit more expressive, wouldn't you agree? In Clojure, you can destructure lists, maps, vectors, and much more. Destructuring quickly becomes a standard part of Clojure development, especially considering how often large data structures such as lists and maps are passed around.

The Macro System

A trait heavily inherited from other Lisp languages is Clojure's support for macros, which I referred to previously. If you've been working with Java for most of your career, you might not be familiar with the concept of macros. A macro is evaluated at compile time and expands into code in a way similar to that of a compiler extension—similar to the preprocessor in C.

Macros are extremely powerful and very commonly used in Clojure. In fact, many of the core constructs in Clojure are actually merely macros. For example, a very common library for web development, named Compojure, uses macros to allow you to define URL routes and dispatch incoming requests that match those routes to functions. After pulling the Compojure dependency into your codebase, you can simply define your routes using the following syntax:

```
(ns demo.hello-compojure)

(def get-user-by-id
  [id]
  (str "getting user for id: " id))

(defroutes my-routes
  (GET "/user/:id" [id] get-user-by-id)
```

In this example, I use Compojure to set up a route for a web server on /user/1. If I request this resource using `curl` or a web browser, I'll receive `getting user for id: 1`. Indeed, you should replace `get-user-by-id` with something more substantial, but this shows the power of macros—the ability to write a lot of code using an expressive shorthand.

The web server I use here is Ring, which is a Clojure HTTP server abstraction. Most of my experience with Clojure has been web-focused, and Ring has been my abstraction of choice. It has been around for a long time, and it has had support and commits from many members of the Clojure community.

Conclusion

Considering the syntax and the language's strengths, Clojure has proven to be a wonderfully powerful language that I love writing in. There's so much more to cover, and this article touches on only a few points that I think make an enticing argument for you to try the language. If you're interested in learning more about Clojure, you can find out more about the language at its home site, the free online tutorial that got me started, or in my writings. `</article>`

Brad Cypert is a software engineer living in Louisville, Kentucky. He has a passion for Clojure, EdTech, and the JVM. Cypert has worked for large-scale tech companies such as LinkedIn and Carfax. In his spare time, he works on his own project, Porios, a Clojure and TypeScript web application that aims to provide a better experience for listening to podcasts.

SIMON ROBERTS

Quiz Yourself

More intermediate and advanced test questions

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

Let me reemphasize that these questions rely on Java 8. I'll begin covering Java 9 in future columns, of course, and I will make that transition quite clear when it occurs.

Question 1 (intermediate). Given this code:

```
int i1 = 1;
Integer io = new Integer(i1);
// line n1
Integer ib1 = i1;
Integer ib2 = 1;

System.out.println(
    (i1 == io) + " " +
    (ib1 == ib2));
```

What is the result?

a. `true true`

b. `true false`

c. `false true`

d. `false false`

e. Compilation fails after `line n1`.

Question 2 (intermediate). Given these two classes:

```
class Base {
  int x = 1;
}

class Sub extends Base {
  int y = 1;

  // line n1
}
```

Which are true of a static method that is inserted at line n1 if no other additions are made to either of the classes?

a. It can *never* access fields `int x` and `int y`.

b. It can *never* access field `int x`, but it can access field `int y`, given a suitable reference.

c. It can access fields `int x` and `int y`, given suitable references.

d. It can access *only* fields `int x` and `int y` using the form `this.x` and `this.y`.

e. It can access fields `int x` and `int y` using the form `this.x` and `this.y`, although other forms might be possible.

Question 3 (advanced). Given the following:

```
public void kickOff() {
  ExecutorService es = Executors.newFixedThreadPool(4);
  Callable<String> c = new Callable<String>() {
    public String call() {return "Done";}
```

//fix this /

```
  };
  String result = es.submit(c).get(); // line n1
  // ...
}
```

Which two are true? Choose two.

a. The code is successful and assigns a value of "Done" to the variable `result` at `line n1`.

b. Modifying the declaration of the call method to the following would require you to add appropriate exception handling to allow `line n1` to compile:

```
public String call() throws MyException
```

c. Modifying the declaration of the call method to the following would prevent the method from compiling:

```
public String call() throws MyException
```

d. Code at `line n1` cannot compile because an `InterruptedException` is not handled.

e. Code at `line n1` cannot compile because an `ExecutionException` is not handled.

Question 4 (advanced). Given this code:

```
class Parent {}
class Child extends Parent {}

public void doBatch(/* formal parameter type */ data) {
  for (Parent p : data) {
  }
}
```

```
/* argument type */ lc = null;
doBatch(lc);
```

Which two pairs of formal parameters and actual argument types could be substituted into the corresponding commented regions to allow successful compilation? Choose two.

a. Formal parameter: `Parent[]`
   Argument: `Child[]`
b. Formal parameter: `List<Parent>`
   Argument: `List<Child>`
c. Formal parameter: `List<? extends Parent>`
   Argument: `List<Child>`
d. Formal parameter: `List<? super Child>`
   Argument: `List<Parent>`
e. Formal parameter: `List<Parent>`
   Argument: `List<? extends Parent>`

Question 1. The correct answer is option A. This question investigates a couple of distinct concepts within the scope of equality and numeric wrapper types. As a reasonable working approximation, the == operator tests if the bit patterns of the values of the expressions on the left and right sides of the operator are the same. For primitive expressions, this means the == operator returns true if the numbers in the two expressions represent the same number (possibly after promotions and unboxing). For object reference expressions, == returns true if the two expressions refer to the same object.

What do the two tests actually amount to? First, let's consider `i1 == io`. This test compares an `int` primitive value with an `Integer` object that's explicitly created by wrapping the same `int` value. In this case, the behavior of the operator is to extract the primitive `int` value wrapped by the object form and perform a comparison of primitive values. Because the values on both sides are 1, the comparison returns `true`. Notice particularly that *Java Language Specification* section 15.21.1 indicates that the primitive value of the wrapper is extracted, rather than a new `Integer` object being created by boxing the primitive. If the latter had happened, the result would have been `false`.

Java allows any data type to be concatenated with a String using the + operator.

In the second case, `ib1 == ib2`, two object references are being compared. This means that the comparison will return true only if the two references `ib1` and `ib2` refer to the same object. If they had been created using invocations of the `new` operator, they would have been different, but in the special case of boxing small integers such as these, the underlying system actually shares the objects from a pool. Because of this, the two references actually do refer to the same object in memory, not merely to two objects with the same semantic meaning or boxed value. So, again, the test produces `true`.

It's perhaps worth taking a moment to expand on that observation about pooling of objects used for boxing. The first observation is that Java guarantees for a specific range of numbers, that if you have the following assignments, the test `a == b` will be true:

```
Integer a = 3;
Integer b = 3;
```

This is possible and safe because the wrapper objects are immutable. Therefore, it's safe to share them, because they can never be changed. It's relevant—and worth doing—to reduce the use of memory and, thereby, to also reduce the load on the garbage collector. The specification says

(in subparagraphs under section 3.10) that this happens for values that fall in the range of –128 through +127 and for Boolean values.

In light of the preceding paragraphs, it's clear that the two comparison expressions both result in the value `true`. Because Java allows any data type to be concatenated with a String using the + operator, the effect is that the resulting output is the word `true`, followed by a space, which is followed by a second occurrence of the word `true`. Therefore, option A is correct, and options B, C, and D are incorrect.

The possibility of a compilation error might be suspected in the later lines, because a primitive is being assigned to an object reference. It's certainly true that Java will not permit an object reference, even one of `Integer` type, to refer to a primitive value, and in a general case, such code would be suspect. But of course, because the boxing feature was added (a long time ago in Java 5) the necessary conversion happens naturally, and the two assignments that follow `line n1` are entirely correct; therefore, no error occurs. Thus, option E is incorrect.

Question 2. The correct answer is option C. This question investigates the nature of a static method and what differentiates it from an instance method.

Suppose this method were added to the sample code in the question:

```
public static void doStuff() {
  y = 99;
}
```

This code would fail to compile due to complaints about accessing the variable `y`. Interestingly, the IDE that I prefer for Java development makes the following imprecise and misleading statement: "non-static variable y cannot be accessed from a static context." It's also pretty common to hear casual observations such as "you can't access instance fields from static methods."

However, the command-line compiler in Oracle's JDK 8 makes a subtly, but critically, different statement. It reports "non-static variable `this` cannot be referenced from a static context." That comment hits the nail on the head. The implicit reference `this` exists only in a non-static

context and, therefore, a static method in the place described in this question cannot use `this`.

Now, given that the unqualified variable `y` in the example static method is implicitly a reference to `this.y`, it's clear that such a form of reference cannot work. Given this observation, options D and E are incorrect. One thing you know for sure is that `this.y` and `this.x` must fail.

The next consideration is whether the static method can gain access to instance variables. Such access is simple, and all that's needed is a valid reference to an instance. Although the `this` reference is not available, a reference to an instance can easily be passed into the static method through its argument list. So, suppose the following method were inserted at `line n1`:

```
public static void doStuff(Sub self) {
  System.out.println("x is " + self.x + " y is " + self.y);
}
```

In this case, the behavior is to print the values of the `x` and `y` fields of the object passed as the actual parameter to the invocation of `doStuff`. Another possibility is that a static method can create an instance of the class to which it belongs; this is common behavior and forms the basis of the static factory concept. In other words, option C is precisely correct: the only requirement for access to the fields is a viable reference. As a result, options A and B, which suggest that there is no possibility of accessing these fields, are incorrect.

The infrastructure of the get method must handle any exceptions from any Callable, regardless of whether they actually exist in a particular implementation.

Another observation is that the suggestion in option B that the usability of the field might depend on whether it's in the class or parent class is an irrelevant distraction. Such a restriction could exist, if the field in the parent class were private, for example. But that's not the case here. There is also a restriction on a child class accessing a parent field if the target field has default access and the child class is in a different package. However, in this case, no information is given about packaging, and the exam's guidance says the following:

> "**Missing package and import statements:** If sample code does not include package or import statements, and the question does not explicitly refer to these missing statements, then assume that all sample code is in the same package, or import statements exist to support them."

Therefore, you should assume that the parent and child classes are in the same package.

Question 3. The correct answers are options D and E. The `ExecutorService` interface is used to encapsulate thread pools. Such a thread pool supports the execution of code presented to the pool using either the `Runnable` or `Callable` interface. While the `Runnable` interface defines a method run that does not declare any checked exceptions, the `Callable` interface defines a method `call` which throws `Exception`. Because of this, option C is wrong; the `call` method can throw any exception that suits it. Of course, such an exception must be handled, and it's not possible for the pool infrastructure to make a semantic decision about how this should be done.

The `submit` method of the `ExecutorService` returns a `Future` object. That `Future` object is really just a "job handle," and it provides several useful ways of interacting with the job after submission. One such behavior is the `get` method, which is used to obtain the result of the completed job. If the job throws an exception from the `call` method, the `get` method must report that exception back to its caller.

Also, the `get` method will block until the job is completed (if necessary), and it's a general convention that any blocking method can be interrupted. If this happens, the blocking call terminates abnormally and throws an `InterruptedException` to the caller. The `get` method adheres to this convention and, of course, `InterruptedException` is a checked exception; therefore, code calling `get` must address this according to the normal rules of Java's exception mechanism. From this, you can deduce that option D is correct.

Notice that the `call` method in the `Callable` interface is declared as `throws Exception`, so the implementation could properly throw an `InterruptedException`. Because the `get` method throws a checked exception for its own reasons, it would be confusing if the same exception could arise from the `get` method for two different reasons. Because of this, along with a gen-

eral simplification of the resulting code around the `get` operation, any exception that arises in the `call` method is wrapped in an `ExecutionException` before being thrown out of the `get` method. The `ExecutionException` is also a checked exception. Therefore, option E is also correct.

If the compiler can't prove safety, the generics system must refuse to compile the code. However, a workaround is provided by the generics syntax.

Given that options D and E are correct, the same logic shows that option A must be incorrect. Further, because the requirement already exists for handling all exceptions that could arise from the `call` method, option B is incorrect. The call to `get` requires exception handling, but this has nothing to do with whether the `call` method is modified. The infrastructure of the `get` method must handle any exceptions from any `Callable`, regardless of whether they actually exist in a particular implementation.

Question 4. The correct answers are options A and C. This question investigates the relationship between generalization and collections in general. It's tempting to think that a `List<Child>` might be a valid substitute for a `List<Parent>` and that option B might be valid. However, even though this code does nothing that would actually cause any problems at runtime, it would not compile. The reason is that, in general, such a substitution is not safe. Imagine if the `doBatch` method attempted to add an item to the `List` it receives. It's proper to add a `Parent` object to a `List<Parent>`; however, it's not safe to do that with a `List<Child>`. For this reason, the compiler must not permit the pairing described in option B, because it would create the possibility of runtime errors. Therefore, option B is incorrect.

It's probably fair, however, to think that much of the time when you pass a collection of something into a method, you want to perform some iterative processing on each member of the collection and you won't be adding items to the collection. If you could do that safely, the ability to pass many `Child` objects into a method that expects the `Parent` type would be quite useful. In fact, if the "many objects" comprise an array, the operation is reasonably safe. This is

because, unlike the collections, arrays know their base type. That is, if you try to store a `Parent` object in an array of `Child`, you'll get an exception (specifically, an `ArrayStoreException`). It's better, perhaps, if the compiler can help you avoid ever making the mistake, but a reliable exception when you perform the insertion is probably acceptable, even though allowing an inappropriate object into the collection is not. Because of this, Java has always permitted the call pairing described in option A, and that option is correct.

However, in the generics mechanism—and, therefore, in the Collections API—there's no such runtime safety. If the compiler can't prove safety, the generics system must refuse to compile the code. But a workaround is provided by the generics syntax. By using the `List<?...` syntax, you can say "I don't know exactly what this will be a list of, but..." and then fill in what you do know. So, what do you need to know for this to work? What's happening is that you're reading `?` objects; that is, whatever the question mark represents in this particular situation, you'll be reading that type of object and then assigning those objects to a variable of type `Parent`. This assignment happens, in this case, in the `for` loop, assigning to the variable `p`. What can the question mark represent if you want to be able to safely assign objects of that type to the `Parent` reference? Anything that is `Parent` or a subtype of `Parent`; therefore, you can express this using the `List<? extends Parent>` syntax. Because of this, option C is correct.

Option D might look plausible, but the pairing can be used only in a method that stores `Child` objects into the list that is passed as the argument. This form fails in a method that assigns the contents of the list to any type other than `Object` (this is because `? super Child` might be a list of literally anything that's a parent class of `Child`, including `Object`). So, given this method body, option D is incorrect.

Option E creates the same possibility of inserting a `Parent` object into a `List` that's not ready to accept it. Because of this, option E is also incorrect. `</article>`

Simon Roberts joined Sun Microsystems in time to teach Sun's first Java classes in the UK. He created the Sun Certified Java Programmer and Sun Certified Java Developer exams. He 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.

//contact us /

Comments

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

Article Proposals

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

Customer Service

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

Where?

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

While 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-3133, Redwood Shores, CA 94065, USA.

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