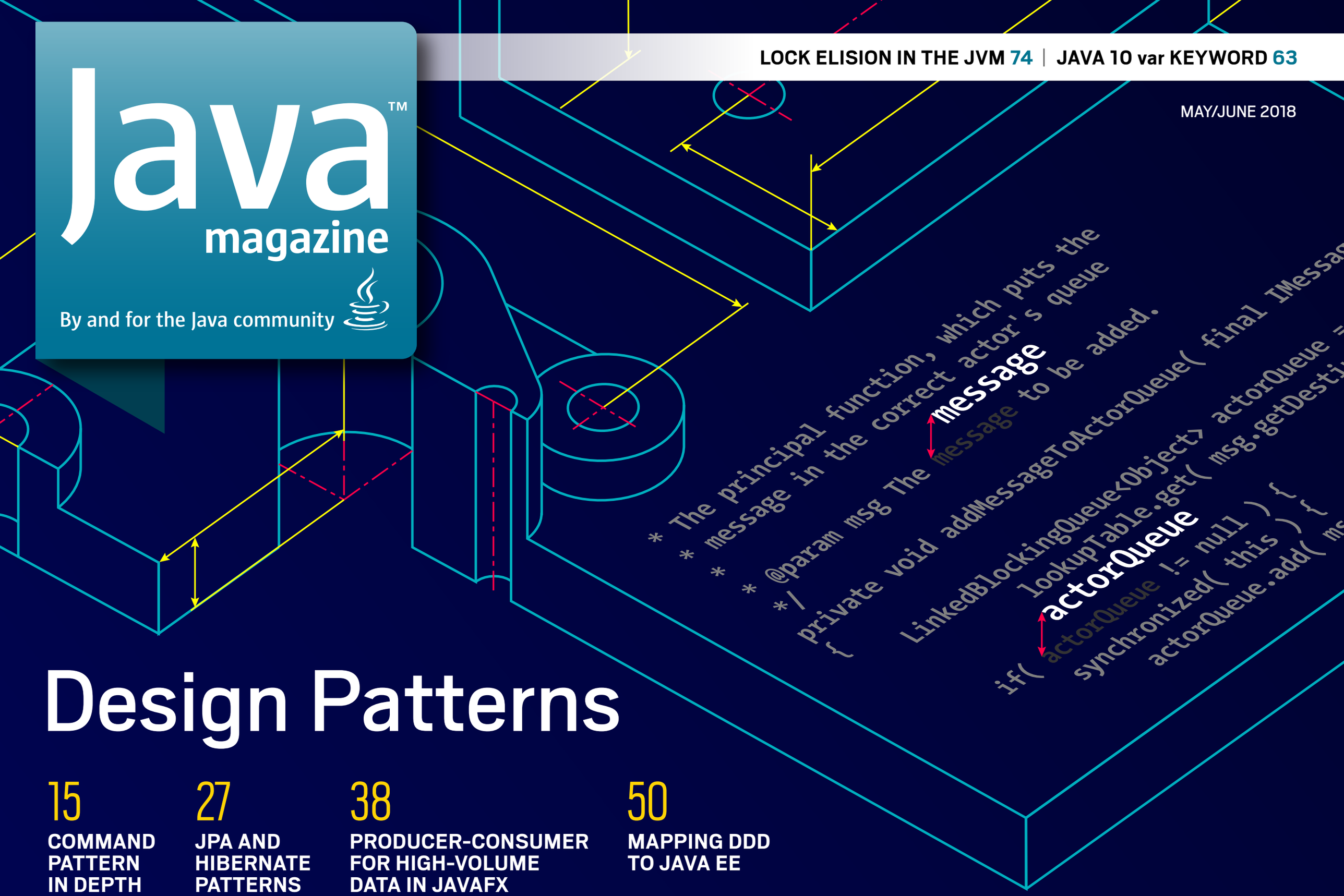
MAY/JUNE 2018

Java™ magazine

By and for the Java community



Design Patterns

15 COMMAND PATTERN IN DEPTH

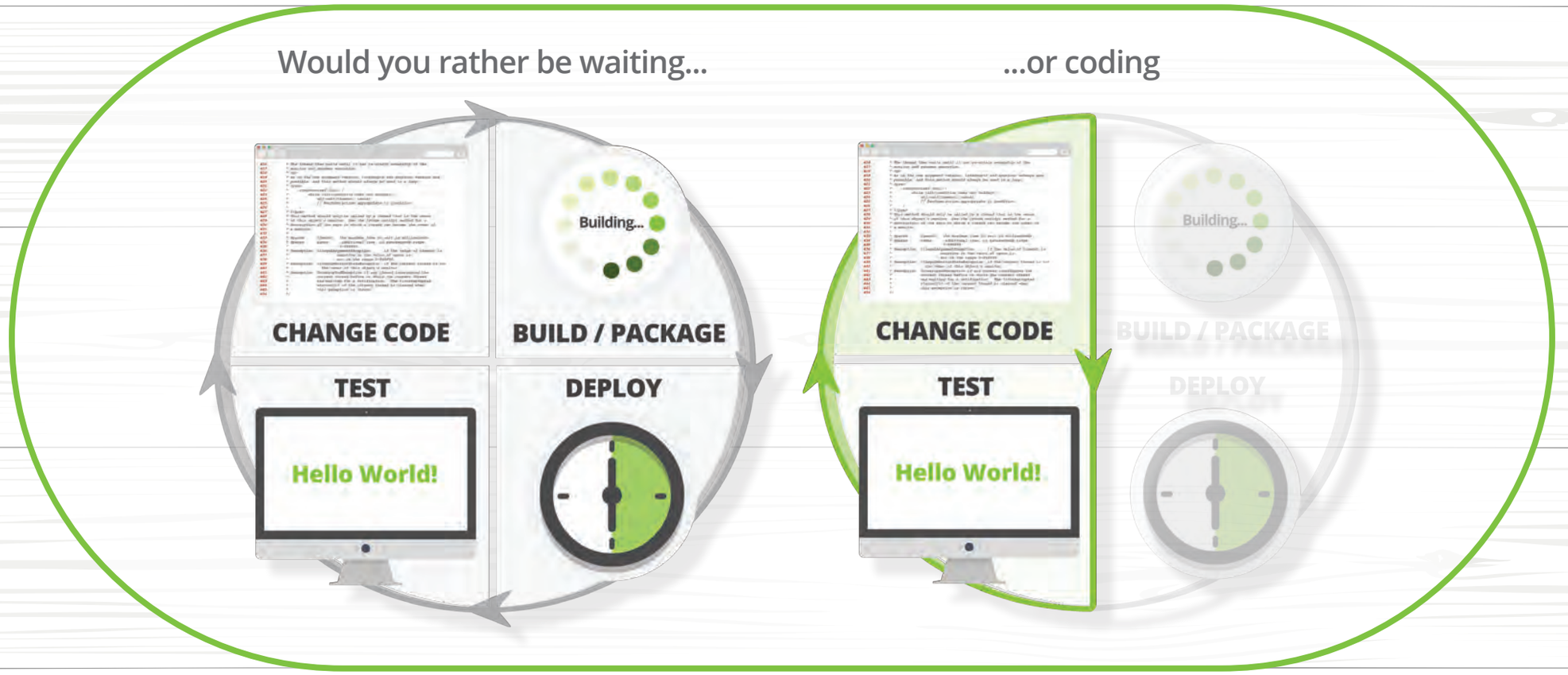
27 JPA AND HIBERNATE PATTERNS

38 PRODUCER-CONSUMER FOR HIGH-VOLUME DATA IN JAVAFX

50 MAPPING DDD TO JAVA EE

RogueWave SOFTWARE | JRebel

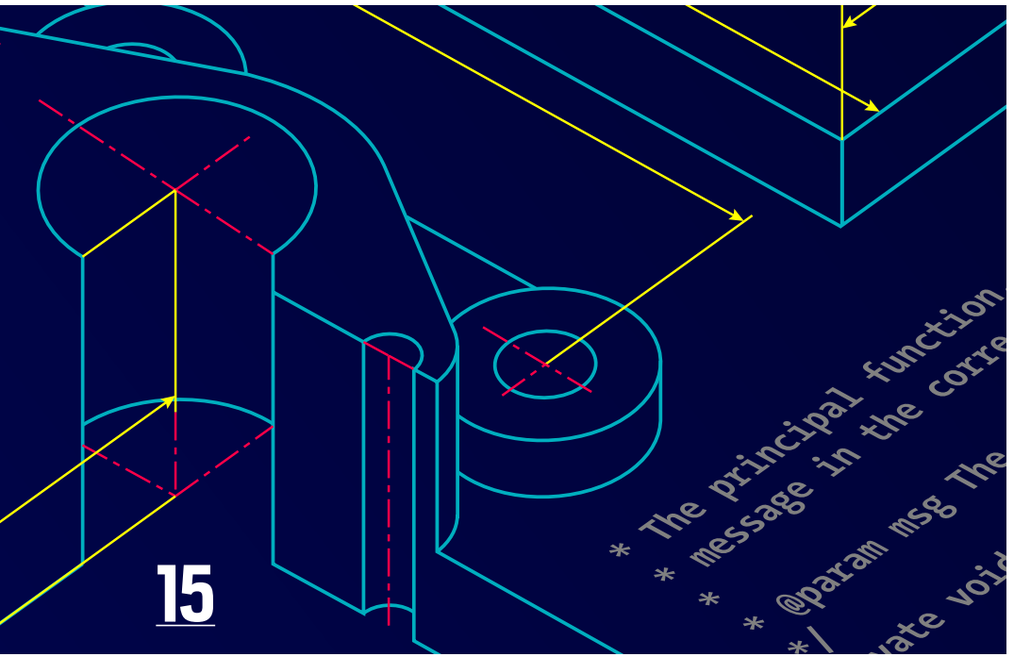
Don't let Java redeploys slow you down



Reload code changes instantly with JRebel

TRY IT NOW! **GET FREE TRIAL**

//table of contents /



THE COMMAND PATTERN IN DEPTH

By Ian F. Darwin

Packaging commands as objects and sending them to a receiver enables a clean, loosely coupled design that's easy to maintain.

COVER FEATURES

OTHER FEATURES

DEPARTMENTS

COVER ART BY WES ROWELL

EDITORIAL

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

DESIGN

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

PUBLISHING

Group Publisher
Karin Kinnear
Audience Development Manager
Jennifer Kurtz

ADVERTISING SALES

Sales Director
Tom Cometa
Mailing-List Rentals
Contact your sales representative.

RESOURCES

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

ARTICLE SUBMISSION

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

SUBSCRIPTION INFORMATION

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

MAGAZINE CUSTOMER SERVICE

java@omeda.com

PRIVACY

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

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.

IntelliJ IDEA

Level up your code
with a Pro Java IDE

jetbrains.com/idea

The Vanguard of a New Generation of JVMs

Graal leads a new ecosystem of emerging JVM technologies that herald big benefits.

Until the 2014 release of Java 8, a common complaint you heard from outside the Java community was that Java, the language, had stalled. The JVM, by comparison, was widely admired. But even though the JVM supported numerous languages (which we've covered individually in this magazine over the last four years), Java was still its principal language and, well, that was getting long in the tooth. Many of us knew this point of view was exaggerated. The Java team had widely and frequently communicated the innovations it was working on and the wave of innovations that would be coming after those. Java 8 indeed disarmed critics who had claimed that Java wasn't keeping up with other languages. And the changes in Java 9 and 10, as well as the upcoming Java 11, are demonstrating the torrid pace at which new features will be delivered. Java remains either the #1 or #2 most widely used language (depending on whose statistics you're looking at), which shows how effective the new features have been in keeping the language relevant, useful, and practical.

The many advances from the Java team have made it easy to overlook other innovation happening with the JVM. Notably, a small team within Oracle called Oracle Labs has spent the last few years researching ways to improve the JVM's performance and its ability to host other languages. In April, the group announced that a set of new technologies had reached version 1.0 and were now available for wider use. Among these were GraalVM—a different take on the current JVM—and the Truffle API, which facilitates porting languages to Graal.

GraalVM is designed with two goals in mind: performance and the ability to support many languages. Let me start with the matter of lan-

PHOTOGRAPH BY BOB ADLER/THE VERBATIM AGENCY

guages. Out of the box, GraalVM supports JavaScript, Python 3, R, and Ruby. In addition, it can run any code destined for the LLVM back end. Today, this includes C and C++, among other languages.

Looking just at the JavaScript support, you can swap out the V8 virtual machine in Node.js and replace it with GraalVM, and your programs will run unchanged. However, thanks to GraalVM's support for other languages, your project can now run more than JavaScript. For example, you can define data structures in C or C++ and use them with JavaScript. How cool is *that*?

A framework called Truffle is available to facilitate using GraalVM for execution of other languages. A tutorial, including instructions for implementing a simplified language, is available here.

The second key aspect of GraalVM is the performance improvement it delivers. Because I have seen only a few benchmarks, I am here relying on information put out by the team at Oracle Labs, in which they tout the new ways they optimize performance.

One of these ways is creating native binaries for Java and JVM programs. They do this with ahead-of-time (AOT) compilation. AOT is a term that refers to regular old compilation to native binaries (as C, C++, Go, and Rust do). To run, the programs use some memory-management and thread-scheduling abilities of the Substrate VM, which is a JVM-like virtual machine that is itself compiled to native code. In this sense, the Substrate VM can be viewed as simply a runtime library of functions needed for execution. This native option removes the JVM's long startup time. In addition, the code is not interpreted at any point, providing further performance benefit.

Another path to reduced execution time is the use of more-aggressive optimizations. For example, GraalVM makes extensive use of escape analysis—a type of analysis that recognizes certain data objects that can be treated as if they were local variables, resulting in far fewer memory allocations. Ben Evans and Chris Newland examined escape analysis in considerable detail in the previous issue. In addition, GraalVM makes expansive use of inline code, so that many calls are now replaced by direct execution. This optimization is available for Java and the languages discussed earlier.

GraalVM is an exciting project—not only because it continues to expand the possibilities of both Java and the JVM, but because it appears to have a genuine opportunity to deliver on the quest of many language infrastructure developers: a virtual machine that runs all major languages well.

Because of its importance in advancing the Java platform and because it is inherently such interesting technology, we'll be covering GraalVM extensively in future issues of the magazine. But don't wait for us: if you want to start working with the technology now, go to the website, download it, and try it out.

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

MARCH/APRIL 2018

Java Web Start

In the March/April issue of your magazine, I stumbled on news of the upcoming absence of Java Web Start from Java 11. Those of us who have used Java Web Start for some time (in my case, 15 years or so) to deploy application and configuration to company desktops are really going to miss this technology. The only real alternative that exists is IcedTea, and some of the community suggestions are patently absurd—my favorite is that "it's time to rewrite those applications in a JavaScript/HTML framework." I hope Oracle open sources Java Web Start and some group picks it up and carries it forward (I'd certainly volunteer).

—Chris Hermansen

Editor Andrew Binstock responds: I checked with the Java team, and they have no plans at present to open source Java Web Start.

What Do Containers Contain?

In the March/April issue's introductory page to the Features section, you wrote "In many cases, containers hold the entire application and all its dependencies; in other cases, only one service resides in the container. This latter model, termed microservices . . ." This is inaccurate. Containers always hold an entire application and dependencies. That is what Docker containers are designed for. You may refer to Docker's own documentation. In the section on containers, it states, "Containers are an abstraction at the app layer that packages code and dependencies together."

—Deepak Vohra

Editor Andrew Binstock responds: This question might be more of a terminological issue than anything else. A single microservice can reside in a Docker container; likewise, so can entire apps and apps that depend on microservices running in other containers. In fact, decomposition of monoliths into microservices requires precisely the ability to run services, rather than complete apps, in their own containers.

Keys Rather Than Mice

I read *Java Magazine* soon after it comes out. One thing that I find somewhat annoying is having to use the mouse to click on the page-forward button. Many interfaces like this have keyboard shortcuts for such actions. I didn't see any information about keyboard shortcuts for this interface. Are there any?

—David Karr

Limited keyboard navigation is available in the issue hosted online: The Enter key provides a zoom/unzoom facility, and the arrow keys allow pagination when the page is not zoomed.

Contact Us

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. Please write to us at javamag_us@oracle.com. For other ways to reach us, see the last page of this issue.

Oracle Press™

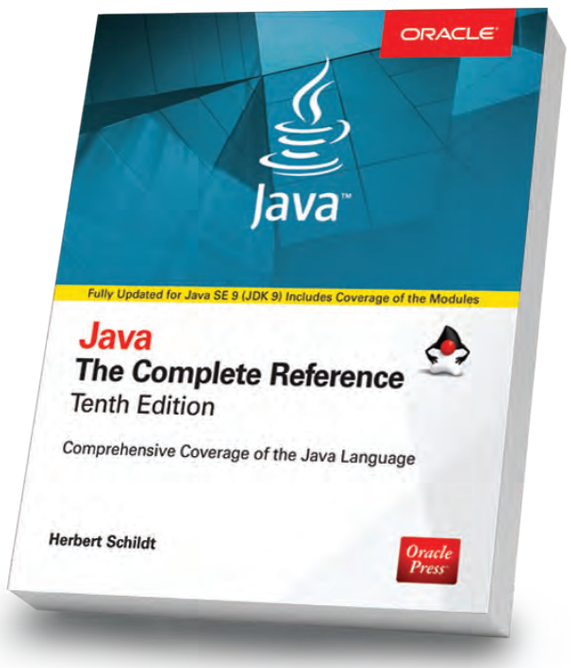
Your Destination for Oracle and Java Expertise

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

Java: A Beginner's Guide, 7th Edition

Herb Schildt

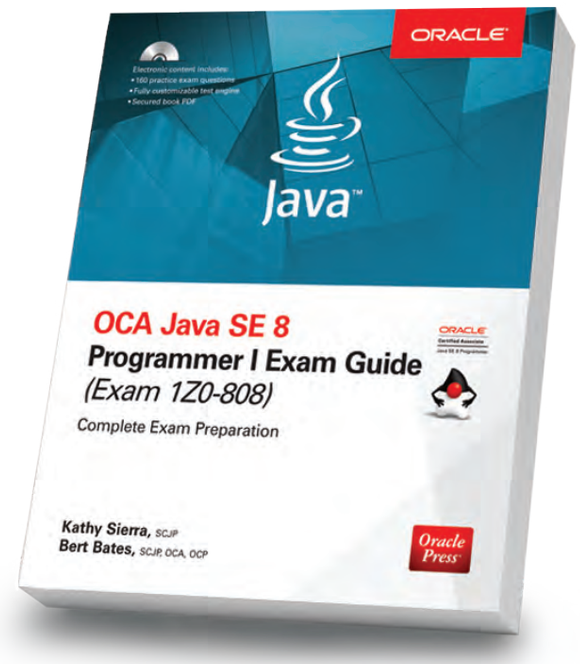
Revised to cover Java SE 9, this book gets you started programming in Java right away.



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

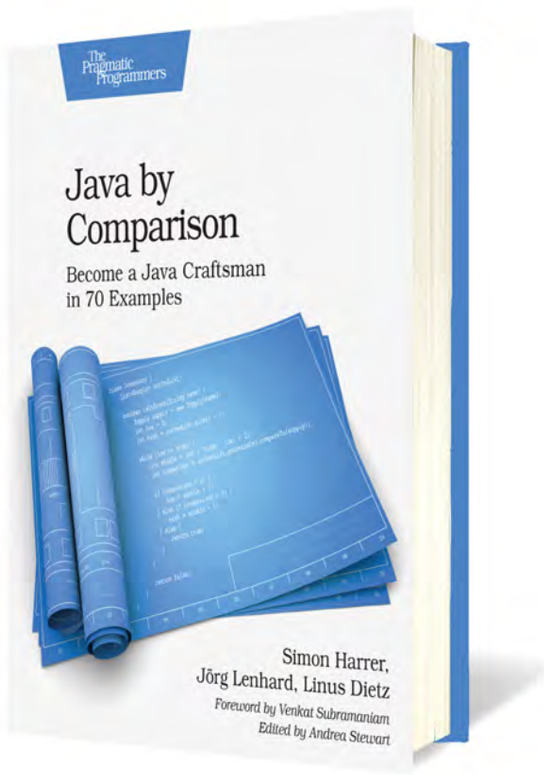
G. Venkat

Adopt a high-performance enterprise Java application modernization strategy.

Available in print and eBook formats.

Oracle Press™

www.OraclePressBooks.com • @OraclePress



JAVA BY COMPARISON

By Simon Harrer, Jörg Lenhard, and Linus Dietz

As code reviews have become the norm in programming, rather than a practice found only in vanguard development organizations, it has become apparent that the industry lacks a way to name and refer to corrections for poorly written code. In this sense, code reviews have needed the kind of terminological advance provided by Martin Fowler's book, *Refactoring*, which gave names to different ways to remediate code (extract class, pull up method, and so forth).

Java by Comparison aims to fill that gap by specifying the cures to 70 common Java programming infelicities. Each less-than-desirable coding choice is explained across its own two-page spread in clear, concise language that any beginning or intermediate developer will find approachable and instructive.

The explanations start with an example of the poor practice and then spend much of the rest of the time articulating why the practice can lead to undesirable results. And then, as you'd expect, the authors show how to correctly write the code. For this reason, the book is invaluable in code reviews for pointing out to programmers who don't understand why a particular piece of their code, which works and passes all tests, needs to be rewritten. Instead of spending time explaining the point, reviewers can simply point programmers to the appropriate entries in this book.

Some of the explanations are of well-trod advice, such as "avoid returning nulls" and "use Java naming conventions." Some move up the complexity spectrum, such as "parametrize [sic] your tests" and "favor method references over lambdas." There are a handful of even more-advanced recommendations, such as "use `collect` for terminating complex streams." And the authors have included a small set of process-oriented recommendations, such as suggestions to use static analysis, continuous integration, and so forth, which don't really fit here and probably would have been better replaced with other code-level recommendations.

Books like this one are excellent for their intended readers: beginners and intermediate developers. In fact, beginners should read the book cover to cover to understand all the conventions they need to bear in mind, whereas intermediates can use the book more as a reference volume. For these roles, I can highly recommend this slim volume, despite occasional faulty English. (For example, parentheses occasionally are referred to as brackets.)

However, I strongly suggest getting the electronic version. The hard-copy version (which is my normal preference for reference works) is unusable because the publisher printed the code *in gray on a gray background*, making it illegible in all but the brightest light. —*Andrew Binstock*

//events /

QCon

JUNE 25–26, WORKSHOPS
JUNE 27–29, CONFERENCE
NEW YORK, NEW YORK

QCon New York is a conference for senior software engineers and architects in enterprise software development. Tracks this year include "Modern Java Reloaded," "Ethics in Computing," "Blockchain Enabled," and "Container and Orchestration Platforms in Action."

PHOTOGRAPH BY HTTPS://BESTPICKO.COM/

J On The Beach

MAY 23–25
MÁLAGA, SPAIN

J On The Beach (JOTB) is an international workshop and conference event for developers interested in big data, JVM and .NET technologies, embedded and IoT development, functional programming, and data visualization.

Spring I/O

MAY 24–25
BARCELONA, SPAIN

Spring I/O focuses on the Spring Framework ecosystem and is the largest Spring-based conference held in Europe.

DevOpsCon

MAY 28–31
BERLIN, GERMANY

This conference is divided into tracks on business and culture; cloud platforms and serverless architecture; container technologies; continuous delivery; logging, monitoring, and analytics; microservices; and security. Sessions include "Become a Cloud-Native: Java Development in the Age of the Whale," "Continuous Integration and Continuous Delivery for Microservices," and "Microservice Authentication and Authorization."

jPrime

MAY 29–30
SOFIA, BULGARIA

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

Riga Dev Days

MAY 29–31
RIGA, LATVIA

The biggest tech conference in the Baltic States covers Java, .NET, DevOps, cloud, software architecture, and emerging technologies.

DevSum

MAY 30, WORKSHOPS
MAY 31–JUNE 1, CONFERENCE
STOCKHOLM, SWEDEN

DevSum focuses on the latest trends and technologies in web development, software architecture, AI and machine learning, programming languages, cloud, and collaboration. This year, the conference has added Java coverage including two ses-

sions on JDK 9 and 10 hosted by Simon Ritter.

Shift

MAY 31–JUNE 1
SPLIT, CROATIA
More than 1,300 attendees are expected at the seventh annual Shift developer conference. Scheduled highlights include speakers from GitHub and Heroku and workshops covering React, React Native, Webpack, and Symphony.

O'Reilly Fluent

JUNE 11–12, TRAINING
JUNE 12–14, TUTORIALS AND CONFERENCE
SAN JOSE, CALIFORNIA
O'Reilly Fluent is devoted to practical training for building sites and apps for the modern web. This event is designed to appeal to application, web, mobile, and interactive developers, as well as engineers, architects, and UI/UX designers. It will be collocated with O'Reilly's Velocity conference for system engineers, application developers, and DevOps professionals.

EclipseCon France

JUNE 13–14
TOULOUSE, FRANCE
EclipseCon France is the Eclipse Foundation's event for the entire European Eclipse community. The conference program includes technical sessions on current topics pertinent to developer communities, such as modeling, embedded systems, data analytics and data science, IoT, and DevOps. The Eclipse Foundation supports a community for individuals and organizations who wish to collaborate on commercially friendly open source software, and recently was given control of development technologies and project governance for Java EE. EclipseCon France attendance qualifies for French training credits.

GOTO

JUNE 18, WORKSHOPS
JUNE 19–20, CONFERENCE
AMSTERDAM, THE NETHERLANDS
This year's GOTO software development conference is devoted to digital transformation, privacy, and security. Workshops on advanced Kotlin and Java are scheduled. Phil Zimmermann, creator of Pretty Good Privacy, will give the opening keynote.

JNation

JUNE 19
COIMBRA, PORTUGAL
Organized by the Coimbra Java user group, JNation brings together technology enthusiasts from all over the world. Attendees will enjoy a full roster of rock-star speakers presenting on subjects including Java and JVM-related technologies, frameworks, tools, programming languages, the cloud, Internet of Things, and much more.

DWX Developer Week 2018

JUNE 25–28
NUREMBERG, GERMANY
This software development conference is conducted in German and will feature talks on GPU computing with Java, Kotlin, and multithreaded JavaScript.

OSCON

JULY 16–17, TRAINING AND TUTORIALS
JULY 18–19, CONFERENCE
PORTLAND, OREGON
Groundbreaking open source projects, from blockchain to machine learning frameworks, will be the focus of the 20th annual OSCON event. Live coding, emerging languages, evolutionary architecture, and edge computing are among the topics this year.

JCrete

JULY 22–28
KOLYMBARI, GREECE
This loosely structured "unconference" involves morning sessions discussing all things Java, combined with afternoons spent socializing, touring, and enjoying the local scene. There is also a JCrete4Kids component for introducing youngsters to programming and Java. Attendees often bring their families.

Java Forum Nord

SEPTEMBER 13
HANNOVER, GERMANY
Java Forum Nord is a one-day, noncommercial conference in northern Germany for Java developers and decision makers. With more than 25 presentations in parallel tracks and a diverse program, the event also provides interesting networking opportunities.

jDays
SEPTEMBER 24–25
GOTHENBURG, SWEDEN
jDays brings together software engineers from around the world to share their experiences in different areas such as Java, software engineering, IoT, digital trends, testing, agile methodologies, and security.

Strange Loop
SEPTEMBER 26–28
ST. LOUIS, MISSOURI
Strange Loop is a multidisciplinary conference that brings together the developers and thinkers building tomorrow's technology in fields such as emerging languages, alternative databases, concurrency, distributed systems, and security. Talks are generally code-heavy and not process-oriented.

KotlinConf
OCTOBER 3, WORKSHOPS
OCTOBER 4-5, CONFERENCE
AMSTERDAM, THE NETHERLANDS
This is the principal conference for the up-and-coming JVM language, Kotlin. The schedule has not yet been posted. However, last year's conference in San Francisco was sold out, and this one will probably be popular as well.

Oracle Code One (formerly JavaOne)
OCTOBER 22–25
SAN FRANCISCO, CALIFORNIA
Oracle Code One (formerly JavaOne) is the premier source of technical information and learning about Java languages and leading-edge technologies including blockchain and artificial intelligence. For four days, developers from around the world will gather to talk about all aspects of Java, JVM languages, polyglot programming, development tools, and trends in technology including cloud and containers. Tutorials on related topics are offered.

DeveloperWeek Austin
NOVEMBER 6–8
AUSTIN, TEXAS
DeveloperWeek Austin will feature tracks devoted to JavaScript, virtual reality development, microservices, and artificial intelligence development.

Devoxx Belgium 2018
NOVEMBER 12–16
ANTWERP, BELGIUM
The largest Java developer conference in Europe takes place again in Antwerp, Belgium, with tracks covering everything from Java, to the mechanics of the JVM, to JVM language. The event is held in a multiplex theater with code and slides shown on giant screens.

Topconf Tallinn
NOVEMBER 20–22
TALLINN, ESTONIA
Topconf Tallinn is an international software conference covering Java, open source, agile development, architecture, and new languages.

DevTernity
NOVEMBER 30–DECEMBER 1
RIGA, LATVIA
The DevTernity forum covers the latest developments in coding, architecture, operations, security, leadership, and many other IT topics. Venkat Subramaniam, author of *Programming Concurrency on the JVM* and *Functional Programming in Java*, is slated to speak.

Oracle Code Events
Oracle Code is a free event for developers to learn about the latest programming technologies, practices, and trends. Learn from technical experts, industry leaders, and other developers in keynotes, sessions, and hands-on labs. Experience cloud development technology in the Code Lounge with workshops as well as other live, interactive experiences and demos.

MAY 30, London, England
JUNE 12, Berlin, Germany
JUNE 20, São Paulo, Brazil
JULY 3, Paris, France

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.

DEVELOPER COMMUNITY EVENTS FROM THE DEVOXX FAMILY

COMING SOON

DEVOXX™

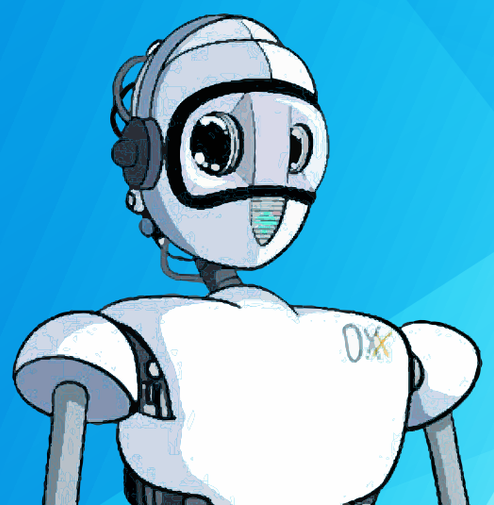
DEVOXX.COM

POLAND 20-22 JUNE

BELGIUM 12-16 NOVEMBER

UKRAINE 23 - 24 NOVEMBER

MOROCCO 27 - 29 NOVEMBER



MINSK 26 MAY

SINGAPORE 1 JUNE

ATHENS 1-2 JUNE

LUXEMBOURG 22 JUNE

TICINO 20 OCT

BRISTOL 25 OCT

BANFF 26 - 27 OCT

MICROSERVICES, PARIS 29 – 31 OCT

CLUJ-NAPOCA 22 NOV

THESSALONIKI 30 NOV - 1 DEC

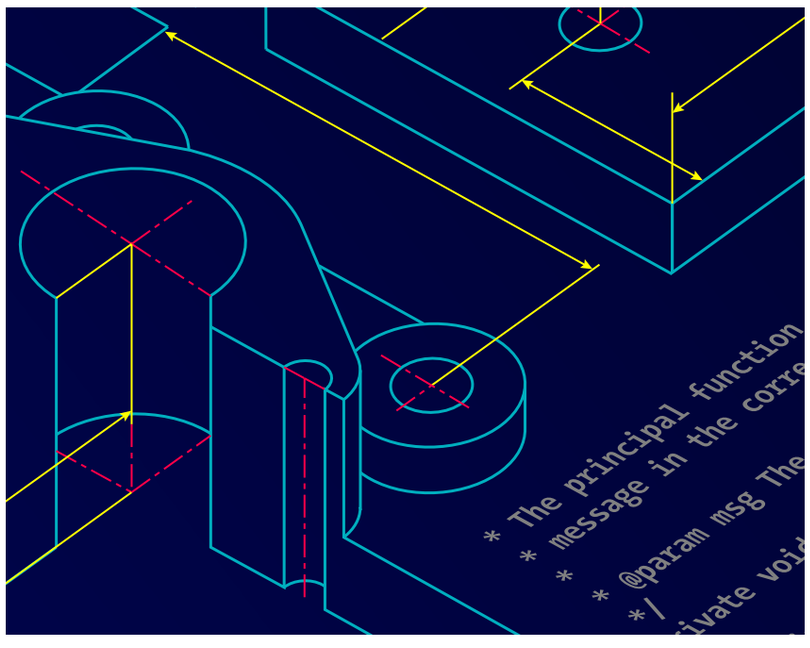
VOXXEDDAYS

VOXXEDDAYS.COM

Useful Patterns and Best Practices

When design patterns first appeared in programming via the famous "Gang of Four" book, they represented a breakthrough on two levels. The first was that they provided a prescription for implementing solutions to basic programming problems. In this sense, they defined best practices. They also provided terminology for describing certain solutions: decorators, factories, and singletons, among others.

In time, the term "patterns" expanded to cover more than the original 23 instances in that seminal book. It came to refer to best practices for solving a common problem. As a result, the use of patterns proliferated, and they now touch every aspect of computing.



ART BY WES ROWELL

In this issue, we launch a series of articles that dig deeply into the most important Gang of Four patterns. Here we start with the Command pattern (page 15) and look at multiple ways to implement it, including across disparate systems. We then look at patterns for using Hibernate and JPA (page 27) and explore the producer-consumer pattern (page 38) as a way to handle large sets of datapoints in JavaFX. Finally, we examine (page 50) how to map domain-driven design (DDD) entities to Java EE.

Separate from patterns, we look at use of the `var` keyword (page 63) introduced in Java 10. And we continue our deep dive into the inner workings of the JVM (page 74) by examining how the Java compiler and the JVM remove unneeded locks from threads—and how this explains why performance between `StringBuffer` and `StringBuilder` varies so much. Enjoy!

IAN F. DARWIN

The Command Pattern in Depth

Packaging commands as objects and sending them to a receiver enables a clean, loosely coupled design that's easy to maintain.

Orders. Commands. All developers are familiar with them in real life: one person's *request* or *demand* that another person perform (or not perform) some action is *transmitted* to another person or persons. It works the same in software: one component's request is transmitted to another in the Command pattern. In this article, I explain how this pattern works and illustrate it with several examples. I also demonstrate how it can be introduced when adding new functionality and when cleaning up existing code.

A Familiar Example of the Command Pattern

The Command pattern is one of about two dozen patterns popularized in the book *Design Patterns: Elements of Reusable Object-Oriented Software* by Gamma, Helm, Johnson, and Vlissides—known more concisely as the "Gang of Four" book or even just the "GoF." (Incidentally, a less academic, more memorable read is *Head-First Design Patterns* by Bates, Sierra, Freeman, and Robson. One other reference worthy of study is *Refactoring to Patterns* by Joshua Kerievsky.)

The Command pattern is not simply a method call (or "message" in the sense that Java's founders used that term). The request is *packaged* in some way, like putting a letter into an envelope and getting the (old school) post office or courier to *deliver* it. In software, the request can be packaged simply as executable code to be performed, it can be a string in some "little language" devised for that purpose, or it can be anything that gets the message across.

Perhaps the most familiar example to Java developers is the `ActionListener` interface used in Swing or the JavaServer Faces action handler bound to a submit button. Some code, which is often loosely called the *handler*, is packaged up and associated with the `JButton` or other control, to be acted upon when the user chooses to click the button.

In this pattern (Figure 1), the button is called the *invoker*. The `ActionListener` implementation is the Command pattern; it consists of the command or code that the application has sent to the button. The object to act upon is called the *receiver*, because it receives the action. The receiver may be passed as a constructor argument to the Command, or it may be implicit in the case of a smaller application using a field in the main class as the receiver.

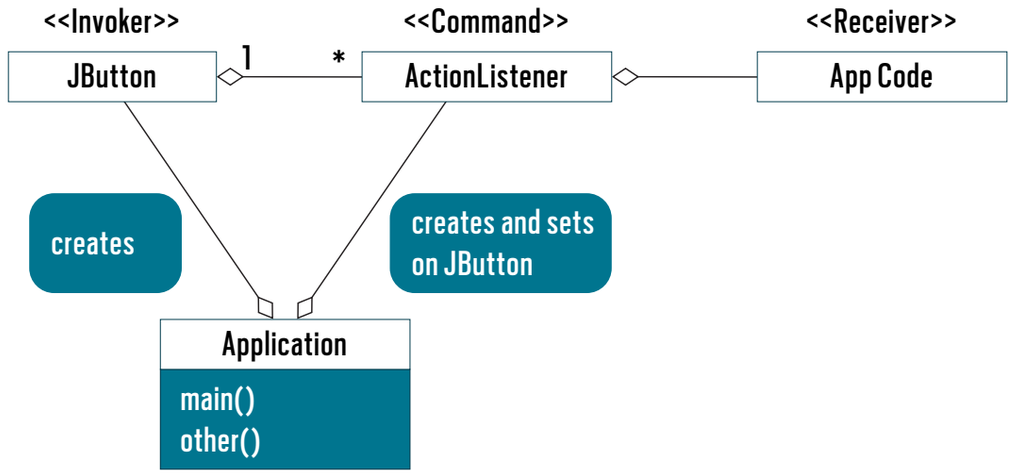


Figure 1: Key players in the Command pattern, illustrated with ActionListener

A Remote Sending Example

As another example, if you want to package some arbitrary code for execution in a different VM—perhaps on a rebel spaceship far, far away—you could package it into an instance of `Runnable`. The `Runnable` interface was designed for use in threading, but it's a perfectly fine interface to use as a Command interface: it has one method, no arguments, and a `void` return type. As my colleague Chris Mawata says, "Use standard interfaces where they serve."

To run a "hello, world" command on another VM, you could package it this way:

```
Runnable command = new Runnable() {
    public void run() {
        System.out.println("Hello, world.");
    }
};
```

Nowadays, I'd probably write that as a lambda, like this:

```
Runnable command = () -> System.out.println("Hello, world.");
```

Then, assuming all the network plumbing has been set up, there might be a method such as `submit()` to send a command to the server:

```
remoteConnection.submit(myCommand);
```

To make the code clearer, you could define `Command` as an interface that extends `Runnable`:

```
public interface Command extends Runnable, Serializable {
    // empty
}
```

`Serializable` is needed for some of the networking transports that might be used, such as remote method invocation (RMI), and it costs nothing anyway. I'd simplify the code by instantiating the lambda inline, as in the following:

```
remoteConnection.submit( () -> System.out.println("Hello, world.") );
```

The code on the other end—the "server"—could implement this method in a simple fashion:

```
public void submit(Command c) {
    c.run();
}
```

Or, the server could put the command into a batch queue, run it in a thread pool (see `java.util.concurrent.Executor`), or use any of several options. Either way, on the client side, you don't know and shouldn't care. Of course, the `Command` interface could be changed to have arguments

and a return type that is not `void`, and it could be an abstract class instead of an interface. I used `Runnable` as a base just to get started.

The code in this `Command` object could perform arbitrary (and possibly malicious) actions on the server, so the server should provide a `SecurityManager` and a policy file to control the imported code.

All the code for this remote sending example is available in my GitHub site under remotecommand. There are two subdirectories, `server` and `client`, with Maven and Eclipse files and a README file showing how to build and run each. If you're not up to speed on RMI, you might want to read my Java RMI tutorial.

One use of the Command pattern is packaging Java code to tell a remote process what to do.

An Auction House Example

The Gang of Four book describes the `Command` object as holding a reference to the receiver; that is, the object on which the work will be done. In a word processor, the receiver might be a `Document` object. In an online auction house, it might be a `Listing` or `Auction` object. My demo implementation of the auction house scenario, called bidpay, is in my patterns-demos GitHub repository. My scenario is so simplified from real life that you can bet it will forever be outbid on eBay, but it's developed enough to show some interesting aspects of the Command pattern.

In that implementation, `Command` is a top-level interface, and two implementing classes with a `Receiver` field (the `Auction` object) are given: `BidCommand` and `CancelCommand`. Here is the former:

```
public class BidCommand implements Command {
    Auction receiver;
    double amount;
    Client bidder;
    public void execute() {
        receiver.bid(amount, bidder);
    }
```

```
    // Obvious three-argument constructor not shown.
}
```

The intent is that the main program, `BidPaySite`, doesn't need to know or care what the clients are sending it. As long as the objects implement `Command`, it will be happy, and the `Auction` class will receive what's sent and process it.

```
public class BidPaySite {
    public void submitCommand(Command command) {
        // These could go into a queue to serialize them,
        // or you could make sure that all methods exposed
        // to the Command are thread-safe.
        // For now, just let the command do its thing:
        command.execute();
    }
    ...
}
```

There are times when you want multiple commands to execute as a single command (for example, something like database transactions, or batching, or reducing network traffic on a remote connection). You could create a `CompositeCommand`, which is created with an array or `List` of commands. The execute method of a `List` implementation could be something like this:

```
class CompositeCommand implements Command {
    List<Command> commands;
    public void execute() {
        commands.forEach(Command::execute);
    }
    // Obvious one-argument constructor omitted
}
```

An Undo Stack Example

I've shown that one use of the Command pattern is packaging Java code to tell a remote process what to do. A different use would be the undo stack in an editor or word processor. When you request an operation such as "insert," "move," or "delete," the editor program could create a Command object representing the operation to be performed. This object would then be passed to a "perform" method in the editor and, upon successful completion, it would be added to the undo stack.

The stack could be implemented as a simple push-down stack of objects of the `EditorCommand` type. When you request an undo operation, the top element is popped off the stack, and it is passed to an "unperform" method in the editor, which removes the inserted text if the operation was an insertion, reinserts the deleted text if the operation was a deletion, and so on. In a full implementation, you wouldn't actually pop the undoable action and drop it after use; you would keep it there for use by a redo command.

In adding base undo functionality into a simple line-editor called edj (also on my GitHub site), I took a slightly simpler approach. To provide a degree of separation between the main code and the "model" (here, the in-memory buffer-handling code), I built the editor from the start with an `interface` called `BufferPrims` between the main code and the operations on the buffer. These are primitive operations such as "add lines," "delete lines," and so on.

There are two versions of the code: `BufferPrimsNoUndo` and `BufferPrimsWithUndo`. In real life, you probably don't need these, so you might not even need the interface, but having them both makes it easier to compare them to see all the changes. In the first version of the code, there was no undo operation. So the first step was refactoring to include the undo capability in the interface, and then have the no-undo implementation, shown next, just print a message:

```
public interface BufferPrims {
    void addLines(int start, List<String> newLines);
    void deleteLines(int start, int end);
    /** Print one or more lines */
    void printLines(int i, int j);
```

```
    /** Undo the most recent operation */
    void undo();
}
public class BufferPrimsNoUndo extends AbstractBufferPrims {
    public void undo() {
        System.err.println("?Undo not written yet");
    }
    ...
}
```

Then, instead of writing code to decipher and reverse each command, I have the "with undo" version of each low-level modify operation create and push an `UndoableCommand` object that contains the exact code to undo the operation. For diagnostic purposes, I associate a String with each `Undoable`, so the `Undoable` looks like this:

```
class UndoableCommand {
    public UndoableCommand(String name, Runnable r) {
        this.name = name;
        this.r = r;
    }
    String name;
    protected Runnable r;
}
```

The two constructor arguments provide all the information you could want, because the undo actions can be simple (the undo of inserting a number of lines is just to delete the inserted range of lines) or complex (the undo of deleting some lines must include all the text of the deleted lines). For example, here is a slightly simplified look at `addLines()`:

```
public void addLines(int startLnum, List<String> newLines) {
```

```
        buffer.addAll(startLnum, newLines);
        current += newLines.size();
        pushUndo("add " + newLines.size() + " lines",
            () -> deleteLines(startLnum, startLnum + newLines.size()));
    }
```

The `pushUndo()` method is simply a convenience routine that creates the `UndoableCommand` and pushes it on the stack:

```
private void pushUndo(String name, Runnable r) {
    undoables.push(new UndoableCommand(name, r));
}
```

Now the undo implementation becomes trivial (error handling is omitted):

```
public void undo() {
    UndoableCommand undoable = undoables.pop();
    undoable.r.run();
}
```

Here is an example of the edj editor in action:

1. I run edj, telling it to start with the sample three-line file included with the source code.
2. The `,p` command prints all the lines in memory; it's short for *1,Np* where *N* is the number of lines in the buffer.
3. The `2d` command deletes the second line.
4. I print the whole thing again to show that the deletion worked.
5. I invoke the newly added undo feature using the `u` command.
6. I print the buffer again to show that line 2 was miraculously restored by the `u` command.
7. I use the command `q` to quit.

```
$ edj 3lines.txt      // 1
3L, 26C
,p                    // 2
Line One
Line Two
Line Three
2d                    // 3
,p                    // 4
Line One
Line Three
u                     // 5
,p                    // 6
Line One
Line Two
Line Three
q                     // 7
$
```

At this point, the undo operation in edj worked nicely. I had refactored the bottom layer made of buffer primitives. But when I went to hook this code into the main line code of the editor, I was reminded that that code is large and hoary. The main loop was something like this:

```
while ((line = in.readLine())  != null) {
    if (line.startsWith("e")) {
        // code to edit a new file
    } else if (line.startsWith("f") {
        // code to print or set filename
        } ...
    }
    // many more if/else statements, one per command
}
```

The book *Refactoring to Patterns* calls such code a *conditional dispatcher*, because it uses a conditional statement (a long chain of `if` statements, but a `switch` is also common). There's nothing inherently wrong with writing code this way, but it can lead to really long methods that are hard to read. You could extract each bit of code into a named method, but that leads to a lot of method names. Ideally, for a couple of reasons, conditional dispatcher code is refactored to use the Command pattern. One reason is if the code requires more flexibility. Another, as the book says, is the following:

"Some conditional dispatchers become enormous and unwieldy as they evolve to handle new requests or as their handler logic becomes ever more complex with new responsibilities."

That is exactly a description of the line editor's main loop: as more commands are implemented, the size of the code in the `if-else` chain or `switch` statement will grow larger without bound.

So I replaced the main loop with a table of Command implementations: an array, indexed by the first letter of each command, is nice and simple. This approach also forced me to provide standardized parsing of the input lines, which up to now was done on demand in the various sections. I introduced the `ParsedLine` class to hold the information about the input line and, in fact, it is a form of Command object, because it describes what to do (but not how, and the receiver is still implicitly `this`).

```
public class ParsedLine {
    char cmdLetter; // 'a' for append, 'd' for delete, etc.
    boolean startFound, commaFound, endFound;
    int startNum, endNum;
    String operands;  // The rest of the line
    public String toString() {
        return String.format("%d,%d%c%s", startNum, endNum, cmdLetter,
            operands == null ? "" : (' ' + operands));
    }
}
```

The `toString()` method is used in this version for debugging, but in a GUI editor, it would appear in the Undo menu item.

The executable Command objects—the actual code—are defined by the interface `EditCommand`:

```
public interface EditCommand {
    void execute(ParsedLine pl);
}
```

With that structure, I was able to trim the main loop to look like this (error checking omitted):

```
while ((line = in.readLine())  != null) {
    ParsedLine pl = LineParser.parse(line, buffHandler);
    EditCommand c = commands[pl.cmdLetter];
    c.execute(pl);
}
```

That is, I parse the line into a `ParsedLine` structure, use the command code from that to find the executable `EditCommand` object, and invoke that. The array of `EditCommand` objects named `commands` is initialized in a static block using assignments like this:

```
// d - delete lines
commands['d'] = pl -> {
    buffHandler.deleteLines(pl.startNum, pl.endNum);
};
```

In other words, each `EditCommand` is constructed as a lambda, passing the `ParsedLine` as a parameter to the `execute` method. As before, the receiver is implicitly the buffer handler.

I've described two uses of Command in my line editor. But most people don't use line editors anymore; they use screen-based editors. And the Swing UI framework already has support for undo operations. I have a simple notepad-style editor called TinyPad that uses this feature.

There isn't room to dissect it here, but if you want to look at its code, check out this GitHub repository. In the "before" version, a `Document Listener` was attached to the main (and only) document, so that when the `TextArea` made any changes to the model, I'd be notified, and an `unsavedChanges` boolean would be set to prompt for unsaved changes when exiting.

In the "after" version, I use Swing's `UndoableEditListener` and `UndoManager`. To see how all those pieces fit together, look at the code starting at `// Set up Undo/Redo actions` and the Command objects `UndoAction` and `RedoAction`.

The GoF book says this: "A command can have a wide range of abilities. At one extreme, it merely defines a binding between a receiver and the actions that carry out the request. At the other extreme, it implements everything itself without delegating to a receiver at all…[in between] are commands that have enough knowledge to find their receiver dynamically."

In bidpay, the command has an explicit receiver and is little more than that binding. In edj, there's only one source file, so the document is available to all code and does not need to be passed with the command. In TinyPad, the command—when coupled with the undo manager—is smart enough to know its associated document internally.

Conclusion

The Command pattern isn't just for undo stacks, of course. It's good for remote execution (as you saw in my first example) and for journaling in database-like systems and file systems to be re-executed after a crash. A composite version can be used to implement database-style transactions and batch processing.

The Command pattern is a good example of a general-purpose design pattern that has many uses and, when applied properly, it will clarify your code and make it more readable and maintainable. And that's largely what this patterns business is all about. `</article>`

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

THORBEN JANSSEN

Design Patterns for JPA and Hibernate

Best practices for an efficient and maintainable persistence layer with JPA and Hibernate

For the last several years, the Java Persistence API (JPA) specification (JSR 338) and its most popular implementation, the Hibernate object relational mapping (ORM) framework, have been widely used in Java software. They are used in Java EE, Jakarta EE, and Spring applications to persist data in relational databases. So it's no surprise that there are several well-established patterns and design principles you can follow to build efficient and maintainable persistence layers. In this article, I explain the reasons to use composition instead of inheritance, the repository and Data-Transfer Object (DTO) patterns, and the Open Session in View antipattern. These are probably the most commonly used patterns and should be known by all experienced developers. To follow along, you'll need familiarity with the concepts and terminology of database access and JPA.

Let's start with two structural patterns and principles that make your application easier to understand and maintain: the Composition over Inheritance pattern and the Repository pattern.

Composition over Inheritance Pattern

As an experienced Java developer, you are probably aware of all the discussions about composition and inheritance. Over the years, the Java world reached the consensus that, in general, you should prefer composition over inheritance for plain Java classes.

The consensus and all the arguments in favor of composition are also valid for JPA entity classes. And in addition to that, there is another important reason you should prefer composition when you model your entities, which I'll explain in a moment.

Let's start by noting that relational databases support only the concept of composition, but not the concept of inheritance. Composition models a "has a" association as foreign key associations between one or more database tables. In your domain model, you can implement composition with an annotation that defines the type of association, such as `@ManyToMany`, `@ManyToOne`, or `@OneToMany` and an attribute of the type `List` or `Set` or of the type of the associated entity. The following code snippet shows an example of a simple many-to-one association between an `Employee` and a `Department` entity:

```
@Entity
public class Employee {

    @ManyToOne
    @JoinColumn(name = "department_id")
    private Department department;
    ...
}
```

As you can see in this code, composition is not only well supported by relational databases but is also easy to map in your domain model. This makes it a great choice.

Inheritance, on the other hand, models an "is a" association. This kind of association can't be modeled in a relational table model. That's why the JPA specification defines a set of mapping strategies that enable you to map an inheritance hierarchy to one or more relational database tables. You can choose from the following:

- A mapped superclass strategy, which maps all subclasses as entities to their own, independent database table without supporting polymorphic queries.
- A table-per-class strategy, which models all classes, including the superclass, as entities and maps them to independent database tables. This strategy supports polymorphic queries.
- A joined strategy, which maps the specific attributes of each entity to its own database table but does not include the attributes of the superclass. So, whenever you want to fetch one of

the subentities, you need to join at least two tables: the one mapped by the superclass and the one mapped by the subclass.

- A single-table strategy, which maps all entities to the same database table. This table has columns for the attributes of all entities in the inheritance hierarchy.

You probably know from your own experience that bridging such a huge conceptual gap isn't an easy task and that it can't be done perfectly.

Unfortunately, that's also true for JPA's inheritance mapping strategies. All four of them have their advantages and disadvantages, but none of them provides an ideal solution. The mapped superclass strategy doesn't support polymorphic queries, which you would expect for a full-featured mapping of an inheritance hierarchy. The table-per-class strategy supports polymorphic queries, but these are very inefficient and most often too slow to be used in complex applications. The joined strategy always requires at least one additional JOIN operation to retrieve subentities. The single-table strategy uses a simple and very efficient table model that doesn't require any JOIN operations to retrieve an entity. But mapping all entities of the inheritance hierarchy to the same database table also has a disadvantage. Some of the columns are mapped by only one subclass and will be null for all records that are mapped to other classes of the inheritance hierarchy. This strategy, therefore, does not permit not-null constraints on columns that are not mapped by the superclass.

Some of the main reasons for the popularity of the Open Session in View antipattern are that it's very easy to use and it doesn't cause any problems on small development systems or test systems.

As you can see, composition does not introduce any additional mapping problems, but all inheritance mapping strategies have their trade-offs. So, when you model your next entity, be aware of these trade-offs and, if possible, avoid them by preferring composition over inheritance.

Repository Pattern

The Repository pattern is a well-established pattern in enterprise applications. The repository class contains all persistence-related code but no business logic. It provides methods to persist, update, and remove an entity or methods that instantiate and execute specific queries. The goal of this pattern is to separate your persistence-related code from your business code and to improve the reusability of your persistence-related code. It also makes your business code easier to read and write, because you can focus on solving business requirements instead of interacting with a database.

If you're using Spring Data or Apache DeltaSpike, you're probably already familiar with the Repository pattern. Both frameworks enable you to generate repositories easily for your entities. They can generate the most common create, read, update, and delete (CRUD) operations and custom queries based on interfaces and method signatures.

The following code snippet defines a repository by extending Spring Data's `CrudRepository` interface and adds a method to load `Employee` entities with a given last name. Spring Data's `CrudRepository` interface defines a set of methods for standard write operations, such as save, delete, and read operations. Spring Data generates a class that implements this interface. So, you don't need to spend any additional effort to get the required functionality.

```
public interface EmployeeRepository
    extends CrudRepository <Employee, Long> {
    List<Employee> findByLastname(String lastname);
}
```

Apache's DeltaSpike project provides you with similar functionality for Java EE and Jakarta EE applications.

In addition to the previous structural patterns, there are also several query patterns and antipatterns that you should apply or avoid when you read your data from a relational database. I want to focus on the two most popular ones: the Open Session in View antipattern and the DTO pattern.

Open Session in View Antipattern

Opening the Hibernate `Session` in your view layer is an antipattern that has been around for years. Let's start with a quick explanation. The general idea is simple: you open and close Hibernate's `Session` in the view layer instead of in the business layer of your application. That enables you to trigger some business operations in your business layer and retrieve one or more entities that you use to render the result in your view layer. During this rendering step, you keep the Hibernate `Session` open so that Hibernate can load lazily initialized entity associations without throwing a `LazyInitializationException`.

Some of the main reasons for the popularity of this antipattern are that it's very easy to use and it doesn't cause any problems on small development systems or test systems. That benefit disappears when you deploy the application to production, where the initialization of these associations requires lots of additional queries. This problem is known as the *n+1 select issue.*

You can avoid these problems by using the DTO pattern, which I explain in the next section, or by controlling the Hibernate `Session` inside your business layer.

Use cases that require a few attributes of multiple, associated entities are also common reasons to use data transfer objects.

The latter solution requires you to initialize all required associations inside your business layer. That becomes necessary because the view layer can no longer access the Hibernate `Session` because it is already closed when the view layer starts the rendering operations. Because of this, each access to an uninitialized association throws a `LazyInitializationException`. The best way to avoid this exception is to initialize lazily fetched associations in your business layer. JPA and Hibernate offer several options for doing that. Let's look at the two most popular ones:

- The use of a `JOIN FETCH` clause in a JPQL query
- The definition of a query-independent @NamedEntityGraph

Initialize associations with a JOIN FETCH clause. A `JOIN FETCH` clause is the easiest option for initializing an association and is my recommendation for all use cases. You can use the `JOIN`

`FETCH` clause within a Java Persistence Query Language (JPQL) query or a `CriteriaQuery`. It tells your persistence provider to not only join the tables of the two associated entities within your query but also to initialize the association.

The following code snippet shows a simple JPQL query that fetches `Department` entities with their associated `Employee` entities:

```
TypedQuery<Department> q = em.createQuery(
    "SELECT d FROM Department d LEFT JOIN FETCH d.employees", Department.class);
```

When you execute this query and activate Hibernate's SQL query logging, the following SQL statement is written to your log file:

```
18:25:08,666 DEBUG [org.hibernate.SQL] -
    select
        department0_.id as id1_0_0_,
        employees1_.id as id1_1_1_,
        department0_.name as name2_0_0_,
        department0_.version as version3_0_0_,
        employees1_.department_id as departme5_1_1_,
        employees1_.firstName as firstNam2_1_1_,
        employees1_.lastName as lastName3_1_1_,
        employees1_.version as version4_1_1_,
        employees1_.department_id as departme5_1_0__,
        employees1_.id as id1_1_0__
from
    Department department0_
left outer join
    Employee employees1_
        on department0_.id=employees1_.department_id
```

As you can see, Hibernate not only selects the database columns mapped by the `Department` entity but also selects all columns mapped by the `Employee` entity within the same query. That's a lot faster than executing an additional query to initialize the association for each selected `Department` entity.

Initialize associations with a NamedEntityGraph. If you load an entity via the `find` method of your `EntityManager` or if you're looking for a reusable way to define the fetching behavior, you can use a `NamedEntityGraph`. It was introduced in JPA 2.1 and provides an annotation-based approach to define a graph of entities that will be fetched from the database.

Here's a simple example of a `NamedEntityGraph` that fetches all `Employee` entities associated with a `Department` entity:

```
@NamedEntityGraph (
    name = "graph.DepartmentEmployee",
    attributeNodes = @NamedAttributeNode("employees")
)
```

After you have defined your `NamedEntityGraph`, you can use a query hint to tell your persistence provider to use it as a `fetchgraph` with your query or your call of the `EntityManager.find` method.

```
Map<String, Object> hints = new HashMap<>();
hints.put("javax.persistence.fetchgraph",
    em.getEntityGraph("graph.DepartmentEmployee"));
Department d = em.find(Department.class, 1L, hints);
```

The generated SQL statement is similar to the one generated for the previously explained JPQL query.

```
18:25:35,150 DEBUG [org.hibernate.SQL] -
    select
        department0_.id as id1_0_0_,
```

```
    department0_.name as name2_0_0_,
    department0_.version as version3_0_0_,
    employees1_.department_id as departme5_1_1_,
    employees1_.id as id1_1_1_,
    employees1_.id as id1_1_2_,
    employees1_.department_id as departme5_1_2_,
    employees1_.firstName as firstNam2_1_2_,
    employees1_.lastName as lastName3_1_2_,
    employees1_.version as version4_1_2_
from
    Department department0_
left outer join
    Employee employees1_
        on department0_.id=employees1_.department_id
where
    department0_.id=?
```

Data Transfer Object Pattern

DTO is another well-known and often used design pattern. It introduces one or more classes to model a data structure employed by a specific use case or by the API of your application. A DTO is a simple Java class that aims to transfer and provide access to its data in the most efficient way. The following code snippet shows an example of the `EmployeeWithDepartment` DTO, which stores the first name and last name of the employee and the name of the department:

```
public class EmployeeWithDepartment {

    private String firstName;
    private String lastName;
    private String department;

    public EmployeeWithDepartment(String firstName, String lastName,
```

```
            String department) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.department = department;
    }

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public String getDepartment() {
        return department;
    }

    public void setDepartment(String department) {
        this.department = department;
    }
}
```

After you've defined your DTO, you can use it as a projection in your JPQL, Criteria, and native queries. JPA supports constructor expressions in JPQL and `CriteriaQuery` queries, and you can use `@SqlResultSetMapping` to map the result of your native queries for each retrieved record. In all cases, your persistence provider selects the specified database columns and calls the constructor referenced in the constructor expression for each record of the result set.

The following code snippet shows an example of a JPQL query using a constructor expression. It consists of the keyword `new` followed by the fully referenced class name of the DTO and one or more entity attributes that define the parameters.

```
TypedQuery<EmployeeWithDepartment> q = em
    .createQuery(
        "SELECT new "
        + "org.thoughts.on.java.model.EmployeeWithDepartment("
        + " e.firstName, e.lastName, e.department.name) "
        + "FROM Employee e WHERE e.id = :id", EmployeeWithDepartment.class);
q.setParameter("id", 1L);
q.getSingleResult();
```

As you have seen, you can easily map each record of your query result to a DTO. But when should you use DTOs?

Reasons to use DTOs. If you implement and deploy your presentation and business layers independently of each other (for example, a Java microservice with a REST API and a JavaScript front end), you'll want to create a stable API that doesn't leak any internal information or design decisions. This enables you to adapt your business layer to new requirements or to improve the existing implementation without changing the API.

It also enables you to exclude some entity attributes from your API—for example, internal attributes that shouldn't be visible to any user or huge lists of associated entities. Especially in REST APIs, it's better to provide a link to another REST endpoint that provides you the requested resources instead of including them in the returned JSON document.

Use cases that require a few attributes of multiple, associated entities are also common reasons to use DTOs. You can, of course, load the entities with all their associations, but that is not as efficient as selecting only the attributes that you need for your use case.

Disadvantages of DTOs. The DTO design pattern also has a few disadvantages. DTOs introduce a lot of redundancy when they are identical to your entities. That creates additional effort whenever you need to change or remove one of these attributes.

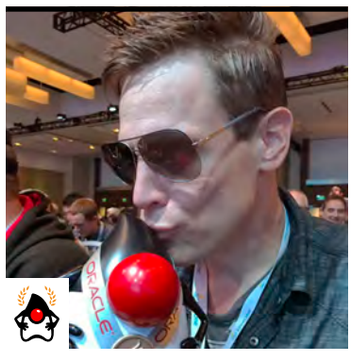
And as you probably know from your own experience, it's difficult to decouple the API of CRUD use cases from the persistence layer. Even so, DTOs enable you to change your entities without changing your API. Real-world projects show that if you change your entities, you most often also need to change your DTOs. If you find yourself in a situation in which your DTO is identical to your entity and you're always changing both of them, you should consider removing the DTO and using the entity instead.

Conclusion

Most Spring and Jakarta EE applications use JPA to implement their persistence layer. It's no surprise that you can choose from several well-established design patterns that help you to implement a robust and efficient persistence layer.

In this article, I discussed why you should prefer composition over inheritance, looked at the repository and DTO patterns, and compared two options for initializing lazily fetched associations in the business layer to avoid the Open Session in View antipattern. These are just a few of the most commonly used patterns. If you want to learn more about design patterns for JPA and Hibernate, you should also take a look at transactional patterns, such as the Session per Request pattern or the Conversation pattern. `</article>`

Thorben Janssen (@thjanssen123) is an independent consultant, a trainer, and the author of *Hibernate Tips: More than 70 solutions to common Hibernate problems*. He has been working with Java and Java EE for more than 15 years and is a member of the CDI 2.0 expert group (JSR 365). He writes about JPA, Hibernate, and other persistence-related topics on his blog, Thoughts on Java.



SEAN M. PHILLIPS

Producer-Consumer Implementations in JavaFX

Graphing high volumes of spiky data requires adaptations to the traditional pattern.

Explanations of traditional design patterns for the Java programming language often ignore the performance context of the implementation in favor of the elegance or purity of the pattern. This choice typically is made to help accelerate the rate of understanding, leaving implementation details to the software engineer. Often, however, the implementation details become problematic and performance requirements become demanding. This is especially true with graphical rendering applications, which are extremely computationally intensive by nature. To squeeze out every last drop of performance, design patterns must be significantly modified to fit within the underlying implementation paradigm.

Building visualizations in Java using the JavaFX API greatly helps with this effort. JavaFX is implemented as a hardware-accelerated scene graph with a rich high-level API to allow even beginner Java developers to make attractive graphical interfaces quickly. Experienced Java developers will find that JavaFX enables implementations that can visualize tens of thousands of data points in near real time. These implementations remain elegantly simple yet performant—even when many other languages and frameworks would fail to perform adequately.

My profession as a software engineer for space-mission analysis applications requires a large amount of complex data computations and, typically, visualizing a lot of data. JavaFX is a highly effective technology to leverage for achieving visually informative interfaces. Careful implementation patterns that balance both elegance and performance are necessary to increase the data processing rates while maintaining the smooth visual user experience. These specialized patterns are hard for beginner and intermediate developers to implement—not because the end result is supremely complex, but simply because the patterns are not quite standard.

This article examines an advanced JavaFX design pattern that addresses a scenario in which a large set of data points will be received in dense bursts. I refer to this as a *high-density data pattern*. The data must be plotted onscreen as fast as possible to be prepared for the next burst. The data is generated dynamically, and the user must see the visualization evolve graphically.

An elegant solution for these requirements would be a variant of the classic producer-consumer thread pattern utilizing Java's `ConcurrentLinkedQueue`. However, JavaFX uses a single rendering thread, and any time spent rendering screen updates will block the application. So, the base design pattern must be modified to minimize the time that code spends rendering onscreen in order to preserve the desired visual effect. This type of specialized pattern, in addition to having a pleasing visual aspect, can be applicable to certain server-side problems as well.

To demonstrate the high-density pattern, I must first construct a test setup that will both generate the data and render it. The code samples that follow were compiled using Java SE 8u161. The entire listing was developed using NetBeans 8.2 and is available as a separate download with this issue. You will need basic knowledge of JavaFX to follow along.

Demonstration of the High-Density Pattern

I'll use a JavaFX `Application` class for execution and a `Canvas` node to visualize the data, as shown below:

```
//Set up simple scene and layout for demonstration
canvas = new Canvas(1000, 1000);
BorderPane bp = new BorderPane(canvas);
bp.setBackground(
    new Background(
        new BackgroundFill(
            Color.BLACK, CornerRadii.EMPTY, Insets.EMPTY)));
Scene scene = new Scene(bp, Color.BLACK);

primaryStage.setTitle("Producer Consumer Canvas Example");
```

```
primaryStage.setScene(scene);
primaryStage.show();
```

To simplify the example and to facilitate the data generation and rendering, I define a few global variables; for example:

```
int setSize = 100000;
```

Here, `setSize` is the total number of objects generated by the producer thread with each burst. The consumer thread must receive each burst of randomly generated data points, process it asynchronously, and then visualize the points. To stress the various patterns in this article, I will increase the size of the burst by increasing the `setSize` variable. An interesting follow-up to this discussion would be connecting this variable to a JavaFX GUI control. Doing so would allow the user to dynamically increase or decrease the stress created by my implementations.

To add some complexity to the example, I create a custom Object called `PointPojo`, which has member fields for the *x* and *y* data coordinates. The `PointPojo()` constructor is smart enough to implement a few interesting geometric equations for each field to make plotting the data interesting. The data set exists within a numeric range unrelated to screen pixels. So, to graph the data on screen, I must transform the data to a numeric range compatible with the screen.

Data values will range from –1.0 to 1.0, with a total range of 2.0 in both the *x* and *y* axes. By predefining the possible data ranges using the following variables, the coordinate transformations are simplified later:

```
double totalMinX = -1.0;
double totalRangeX = 2.0;
double totalMinY = -1.0;
double totalRangeY = 2.0;
```

To assist with debugging the initial development and to assess performance effectiveness, I need a timing metric for the different methods of the application. The downloadable listing

provides an example method, `printTotalTime()`, which calculates elapsed time at the nanosecond level using `System.nanoTime()`. The output should appear like this:

```
Total elapsed time: Total ns: 128000, 0:s:0:ms:128:us:0:ns
```

High-precision timing is an important nuance of graphical programming where even one additional millisecond (ms) of time is considered expensive. I will time my methods with the goal of obtaining single-digit ms times.

For this design pattern, I want to both produce and consume a high volume of data, so I need the producer-consumer interaction to support multiple threads executing asynchronously. There are several ways to facilitate the data exchange between the producer and consumer threads. Many beginner and intermediate examples of this pattern rely on the Java `synchronized` keyword to implement an explicit locking pattern. Typically, these examples suggest that you create methods that encapsulate access to a shared collection such as a List or Queue. In these basic examples, the encapsulation methods are guarded by the `synchronized` keyword. This approach creates a blocking implementation that is thread-safe. In a high-performance situation, though, I need to minimize the time spent blocking and managing access to shared resources between threads. Rather than implementing a blocking thread-safe approach, I recommend using `java.util.concurrent.ConcurrentLinkedQueue`, which is a highly efficient non-blocking collection that is thread-safe.

To simulate a long-running service or perpetual source of incoming data, I created an asynchronous producer daemon thread using the `javafx.concurrent.Task` interface. This task, shown in the following code, is an infinite loop that periodically creates a set of objects that contain data.

```
//Define producing task to create and add data points to queue
Task<Void> producerTask = new Task<Void>() {
    @Override
    protected Void call() throws Exception {
        int count = 0; //track how many times I have produced data
```

```
//Produce data asynchronously forever
while (true) {
    //wait until the data has been processed
    if (!pointQueue.isEmpty()) {
        Thread.sleep(0, 100); //Spin the thread a bit
    } else {
        //The queue is ready to be filled again
        System.out.println("Producer Task run: " + count);
        for (int i = 0; i < setSize; i++) {
            PointPojo p = new PointPojo();
            pointQueue.add(p);
        }
        count++;
    }
}
    }
};
```

To execute my rendering approach, I now create an asynchronous consumer daemon thread—again, using the `javafx.concurrent.Task interface`. The task, shown in the following code, is an infinite loop that attempts to render points placed in the `pointQueue` by the producer thread as fast as possible.

```
//Start consumer task for rendering the data
Task<Void> consumerTask = new Task<Void>() {
    @Override
    protected Void call() throws Exception {
        resetCanvas(); //randomize current canvas fill color
        //track sets of points that are plotted
        int plotCount = 0;
        while (true) {
```

```
//drawing in batches reduces thread contention
long startTime = System.nanoTime();
int batchSize = drawNext_SimplePolling();
//int batchSize = drawNext_ArrayTransforms();
printTotalTime(startTime);
System.out.println("Drew batch size of " + batchSize);
plotCount += batchSize;
//little thread spin helps it animate
//less than 15 ms can freeze graphics
Thread.sleep(25);
//if the set is plotted let's change the colors
if (plotCount >= setSize) {
    resetCanvas();
    plotCount = 0;
    System.out.println("Reset Canvas");
                }
            }
        }
};
```

In this code, there are two custom method calls: `drawNext_SimplePolling` and `drawNext_ArrayTransforms`. These methods can use different approaches to process and render the data produced. I will start with `drawNext_SimplePolling` and then switch to `drawNext_ArrayTransforms` and compare the results. Each execution of this `Task` produces a graphical view similar to **Figure 1**, with each subsequent loop iteration graphically drawing over the previous iteration. After each data set is rendered, the fill color for the `Canvas`

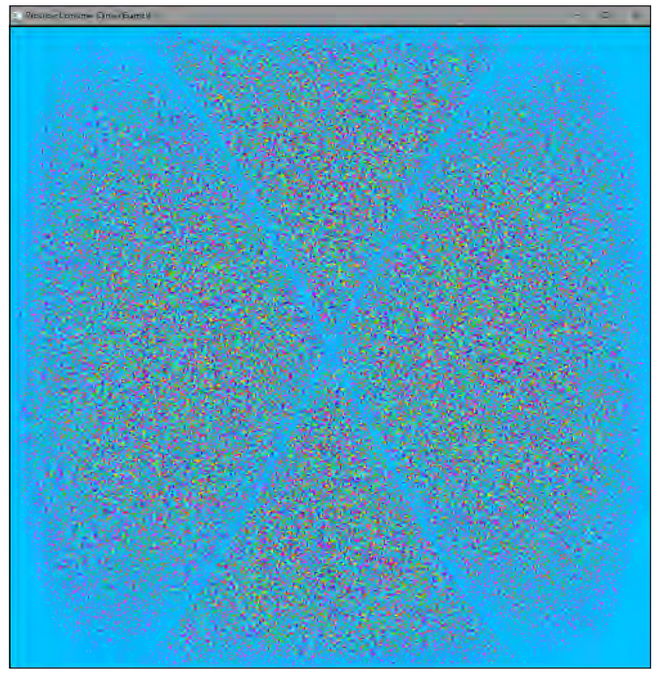


Figure 1: Sample output from the program

is randomized, which helps visually indicate when each data set has been completely rendered. Because the colors are random, sometimes subsequent renders will look similar and other times not. An interesting upgrade to this example would be to connect the color scheme to a JavaFX GUI control to allow the user to manipulate the colorations.

Recall the issue discussed earlier regarding transforming the data to screen space. Typically with data visualization scenarios, the numeric range of the data will not be in a numeric range that is workable for screen coordinates. For example, many data sets are normalized to a range of less than 1.0 and often have negative values. To avoid using hardcoded scaling values, determine the minimum, maximum, and range of the data coordinates and transform them to the screen (and, therefore, `Canvas`) coordinates using a method such as the following and the data maxima and minima I defined earlier:

```
private double transformXToScreen(double x) {
    return (((x - totalMinX) * (canvas.getWidth() - radius))
        / totalRangeX) + radius;
}
```

For coordinate transformations of graphical renderings, I recommend simplifying your rendering code by separating the coordinate transformations into separate methods for both the *x* and *y* axes. Placing these calculations in a separate method makes it easier to debug and profile the application's performance.

First Approach

In the consumer `Task`, I will investigate two small alternative patterns for retrieving and rendering the data, each defined in separate methods. The first approach, which follows, is a simple polling strategy that would look similar to a queue-based implementation:

```
    private int drawNext_SimplePolling() {
        int size = pointQueue.size();
        GraphicsContext g = canvas.getGraphicsContext2D();
```

```
//        Platform.runLater(() -> {
            while (!pointQueue.isEmpty()) {
                PointPojo point = pointQueue.poll();
                //coordinate transformation from data to canvas pixels
                double x = transformXToScreen(point.x);
                double y = transformYToScreen(point.y);
                //encourage the object to be garbage collected sooner
                point = null;
                g.fillOval(x, y, radius, radius);
            }
//        });
        return size;
    }
```

This code features a commented call to `Platform.runLater`. For those unfamiliar with this feature, the JavaFX platform provides the `runLater()` interface as a blocking but thread-safe way to run code and make changes to a JavaFX GUI. Any code placed inside a `runLater` block will be executed in a `Runnable` thread. The `Runnable` will be executed at some point later by the JavaFX platform, but we won't have control over the timing. I will examine performance for both using and not using `runLater` calls.

In this method, I continuously pull `PointPojo` objects from the concurrent queue using the `.poll()` method until the queue is empty. After each `.poll()` call, I transform the data to screen coordinates and render to the canvas using the `fillOval(x, y, radius, radius)` call. Without using the `runLater` option, I achieve the following performance:

```
Total elapsed time: Total ns: 24242446, 0:s:24:ms:242:us:446:ns
Drew batch size of 64434
```

These numbers will fluctuate, of course, but they represent a median for time and points processed within that time. However, at this pace and number of points, I quickly run into

the problem with my rendering requests, which are not thread-safe by default. This code will quickly and inevitably lock the JavaFX rendering thread due to its thread-unsafe behavior. This can be somewhat mitigated by wrapping the `while` loop in the thread-safe `Platform.runLater() Runnable`, which is commented out in the previous code. Doing so improves the performance as follows:

The important detail here is to perform the coordinate transformations *off* the rendering thread.

```
Total elapsed time: Total ns: 497896, 0:s:0:ms:497:us:896:ns
Drew batch size of 56194
```

For a similar quantity of points, total execution time has shrunk from 24 ms to less than 1 ms. Don't be fooled though: this reduced time is deceptive because what I have done is handed the workload to the JavaFX platform to manage and then execute at the next available rendering pass. This approach takes much less time than the original approach without `runLater()`, but now I am sending a large number of `Runnable`s to the JavaFX platform for processing.

This solution scales well to the 10,000-point range. However, what if you want to add an order of magnitude and make it 100,000 points? Given all the coordinate transform math and other work being done, the time spent in each `Runnable` instance is too large. Further, given the high frequency of data bursts, there are far too many `Runnable` instances being sent to the JavaFX platform for processing. This pattern could potentially throw cryptic rendering exceptions, such as `java.lang.InternalError: Unrecognized PGCanvas token: 64`, when the underlying JavaFX engine cannot keep up.

Second Approach

An alternative approach is a slight variation to the previous method. This implementation seeks to do as little as possible in the blocking `Platform.runLater() Runnable`. The code for this is shown toward the end of the following implementation of the `drawNext_ArrayTransforms()` method:

```
private int drawNext_ArrayTransforms() {
    int size = pointQueue.size();
    GraphicsContext g = canvas.getGraphicsContext2D();
    //temporary arrays to hold the transformed canvas coordinates
    //Be sure to use primitive double type to minimize memory
    double[] xArray = new double[size];
    double[] yArray = new double[size];
    //loop across all available points by polling the concurrent queue
    for (int i = 0; i < size; i++) {
        PointPojo point = pointQueue.poll();
        //coordinate transformations stored in temporary arrays
        xArray[i] = transformXToScreen(point.x);
        yArray[i] = transformXToScreen(point.y);
        //encourage finalization of the object I polled
        point = null;
    }
    //Using arrays to hold transformed coordinates allows me
    //to use a runLater() thread while minimizing blocking time
    Platform.runLater(() -> {
        //The key is minimizing time spent in this blocking thread
        for (int i = 0; i < size; i++) {
            g.fillOval(xArray[i], yArray[i], radius, radius);
        }
    });
    return size;
}
```

You will notice that I use a for-loop to poll objects from the concurrent queue. I could have used an iterator instead of the for-loop to process the available points. However, iterators take more than an order of magnitude longer than a loop that calls `poll()`, because the JVM Hotspot has

not yet "warmed up" to the iterator bytecode and so it is not JIT'ed but rather would be interpreting each command.

The important detail to mention here is that the code in this alternative method performs the coordinate transformations *off* the rendering thread. Moving these time-expensive calculations off the rendering thread minimizes the time spent blocking in the JavaFX rendering thread. By minimizing the time spent in the JavaFX rendering thread, I reduce or eliminate the chance that we will experience a threading exception. This approach can be demonstrated by commenting out the `drawNext_SimplePolling` method and uncommenting the `drawNext_ArrayTransforms` method featured in the Consumer thread code block and provided with the downloadable listing. By transforming the coordinates outside the `Platform.runLater` call, I remain asynchronous and thread-safe. Now the only work that must be done in the blocking `runLater` is a simple loop of `fillOval()` calls, which perform the actual drawing onto the `Canvas` object.

Making this change uses the slightly modified rendering pattern that minimizes the time spent executing code in the JavaFX rendering thread. With this change made, and setting the `setSize` variable to 100,000 points, I achieve the following performance:

```
Total elapsed time: Total ns: 3178141, 0:s:3:ms:178:us:141:ns
Drew batch size of 100000
```

I shifted the majority of the computation off the rendering thread (3 ms of work) and was able to process and render all 100,000 points every single pass. This improvement means that the pattern now does not risk freezing the rendering loop due to an unsafe thread operation. This is a really amazing result, which maintains a fairly elegant solution with only a slight increase in complexity. Note that this result was obtained using a nonparallel computing approach. The consumer thread is not explicitly leveraging multiple cores if they are available. An interesting extension would be to modify this pattern to explicitly perform the coordinate transformations in parallel using all available cores.

This solution continues to scale up in a predictable manner. Adding another order of magnitude, I am able to plot 1 million points without locking the rendering thread or having a rendering exception thrown. The consumer thread on my machine tended to process points almost as fast as the producer thread could add them to the `ConcurrentLinkedQueue`, creating a natural pattern of rendering batches, as shown in **Figure 2**.

According to the console output, which was taken from the NetBeans IDE, typically 1 million points could be processed and rendered in about 30 ms on my machine. This means about 30 frames of animation per second, which is faster than any human can discern. Excellent!

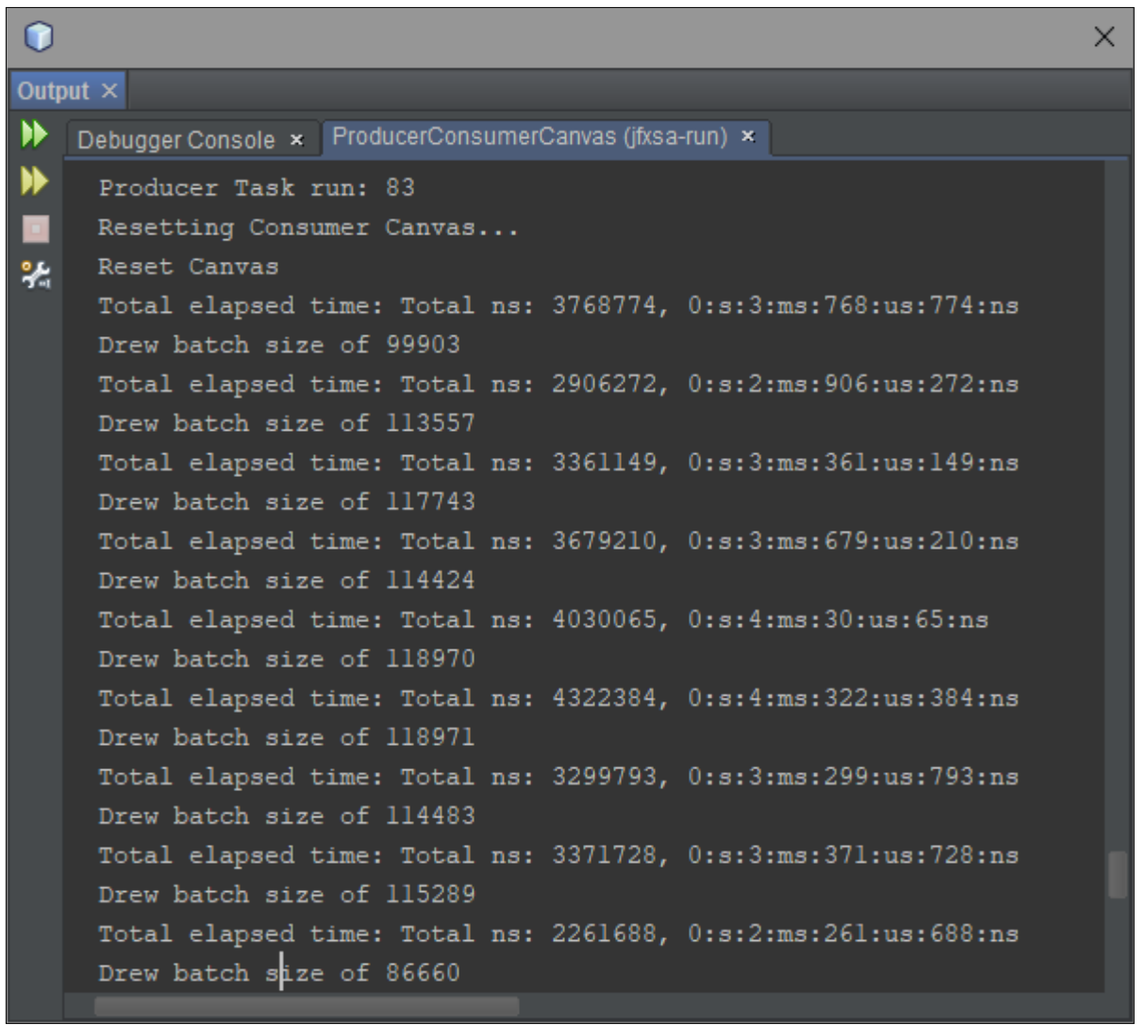


Figure 2: Sample output from second approach

Conclusion

I have shown patterns for processing and visualizing a large number of data points using JavaFX, including the JavaFX Canvas class and the Platform.runLater() interface. However, what I have really done is extend the producer-consumer design pattern through a threaded implementation that also takes into account the details of a JavaFX visualization. Although it was specialized for high performance, this example was reduced for the sake of rapid understanding. However, you can use this JavaFX design pattern as a kernel around which to build better visual interfaces. `</article>`

Sean M. Phillips (@seanmiphillips) is a recently elected Java Champion and a consulting software engineer with NASA and a.i. solutions, an aerospace industry leader that provides solutions to the USAF. His specialties are data analysis and visualization using Java and JavaFX. He created the Deep Space Trajectory Explorer software, which won a Duke's Choice Award in 2017 and is used for trajectory design to deep-space targets.

SEBASTIAN DASCHNER

Using Domain-Driven Design with Java EE

How to map DDD artifacts to Java EE code

Domain-driven design, as described in the book by Eric Evans, aims to construct software models that represent an actual business domain as accurately as possible. It especially focuses on the necessity of communicating with domain experts; sharing a common, ubiquitous domain language; refining the understanding of the underlying domain model; and gradually refactoring the model.

Domain-driven design (DDD) describes certain concepts, such as *bounded contexts*, *aggregates*, and *entities*. Is it possible to implement the concepts with Java EE or the upcoming Jakarta EE? Let's examine these concepts and see how the programming model of modern Java EE enables developers to craft proper domain models, starting with some basic definitions.

Bounded Contexts

Bounded contexts enclose the meanings and responsibilities of some part of the domain. A specific domain entity contained in a bounded context could be a customer, for example. The boundaries, responsibilities, and possible overlaps of bounded contexts are defined in a *context map* of the system.

In the model of microservices, a bounded context would typically result in a single deployable application.

Domain Entities

Entities represent the business *domain entities*. An important feature of domain entities is that they are identifiable by their nature. For entities, it matters "which" entity object is being referred to.

Let's take an example of an instrument craft shop. A crafted instrument is an identifiable entity, implemented as a plain Java class. In Java EE, it's especially interesting to see how the entities are persisted to the database via Java Persistence API (JPA).

The examples I'll use are from a music-instrument craft shop application. In the following code, the JPA annotation `@Entity` is used to map identifiable domain entities to the database. JPA requires that the entity define an identifier, mapped by `@Id`.

```
@Entity
public class ElectricGuitar {

    @Id
    private long id;

    private Model model;

    public long getId() {
        return id;
    }

    public void setId(long id) {
        this.id = id;
    }

    public Model getModel() {
        return model;
    }

    public void setModel(Model model) {
        this.model = model;
    }
}
```

The instrument model is a *value object* (discussed next), which is mapped as an embedded JPA property.

Value Objects

Value objects are business domain types that do not represent identifiable types but rather specific values. For these domain objects, it doesn't matter which instance will be used inside the business process. Examples of value objects are addresses, money values, or Java enums. Value objects ideally are immutable and, therefore, reusable.

The model of an instrument is an example of a value object. Instrument models that are defined by the same brand and name are identical and can be used interchangeably.

Value objects are mapped with JPA as embeddable objects—because entities are required to define identifiers. The database table of the enclosing entity type (here, `ElectricGuitar`) will inline all nontransient fields of the embeddable type (here, `Model`).

```
@Embeddable
public class Model {

    @Basic(optional = false)
    private String brand;

    @Basic(optional = false)
    private String name;

    protected Model() {
        // required by JPA
    }

    public Model(String brand, String name) {
        this.brand = brand;
```

```
        this.name = name;
    }

    public String getBrand() {
        return brand;
    }

    public String getName() {
        return name;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Model model = (Model) o;
        return Objects.equals(brand, model.brand)
            && Objects.equals(name, model.name);
    }

    @Override
    public int hashCode() {
        return Objects.hash(brand, name);
    }

    @Override
    public String toString() {
        return brand + ", " + name;
    }
}
```

Value objects typically implement `equals` and `hashCode` to ensure that identical instances are recognized as such. You probably noticed that the model implementation is not completely immutable. This is because of the requirement in the current JPA specification to define no-argument constructors with at least `protected` visibility. Some mapping frameworks, such as Hibernate, make it possible to further restrict the visibility by defining private no-argument constructors. This step, however, is not fully compliant with the JPA standard and leads to nonportable applications.

Services

Services are responsible for performing domain business logic that is not naturally part of an entity or value object. They are entry points of the business use cases that manage and orchestrate domain entities, and they hold together the separate steps of the business process.

In a Java EE world, services are implemented as managed beans—either Contexts and Dependency Injection (CDI) beans or EJB beans. Services that serve as the entry point for business use cases, sometimes called *boundaries*, are usually implemented as EJB beans. They already comprise often-required cross-cutting concerns, such as transactions.

The `InstrumentCraftShop` service represents the use case boundary for creating new music instruments:

```
@Stateless
public class InstrumentCraftShop {

    @Inject
    InstrumentMaker instrumentMaker;

    @PersistenceContext
    EntityManager entityManager;

    public ElectricGuitar craftInstrument() {
        ElectricGuitar instrument = instrumentMaker.build();
```

```
        return entityManager.merge(instrument);
    }
}
```

The boundary services typically delegate complex domain logic to other services. These delegates, such as `InstrumentMaker`, are injected into the beans via dependency injection.

Aggregates

Aggregates represent more-complex domain entities that consist of multiple entities or value objects. They are accessed and managed as a whole by a single root object to ensure integrity and consistency.

In JPA, persistence operations are invoked on the root entity of an aggregate. The operations cascade to the other entities of the aggregate.

In the following example, I examine a `GuitarBody` type, which will become part of an aggregated electric guitar. The `ElectricGuitar` type represents the root entity of the aggregate. The `GuitarBody` type is another entity in the instrument domain.

```
@Entity
public class GuitarBody {

    @Id
    private long id;

    @Enumerated(EnumType.STRING)
    @Basic(optional = false)
    private Material material;

    @Enumerated(EnumType.STRING)
    @Basic(optional = false)
    private Color color;
```

```
protected GuitarBody() {
}

public GuitarBody(Material material, Color color) {
    this.material = material;
    this.color = color;
}

public enum Material {
    MAPLE, MAHOGANY
}

public enum Color {
    BLACK, RED
}
```

In this domain, electric guitars comprise a single guitar body (which, for reasons of traceability, is identifiable).

The `GuitarBody` type is referenced in electric guitars and mapped appropriately via JPA. The following shows an `ElectricGuitar` type that is enhanced for persistence:

```
@Entity
public class ElectricGuitar {

    // id, model, getters & setters from previous definition

    @OneToOne(cascade = CascadeType.ALL, optional = false)
    private GuitarBody body;

}
```

The cascading `ALL` relation causes all persistence operations that are invoked on the electric guitar to be cascaded to the body and its potential relations, thus ensuring consistency of the involved entities. `ElectricGuitar` represents an aggregate type.

Repositories

All the mentioned persistence operations need to be invoked somehow. In the same way, domain entities need to be retrieved from a persistence provider in a consistent way.

DDD *repositories* are responsible for managing the persistence of domain entities. They encapsulate this functionality in a self-sufficient and consistent way to keep the rest of the domain model clear of persistence implementation details. Only entities that expose a unique identity within the business domain are persisted and managed via repositories.

In Java EE and JPA, the provided `EntityManager` type fulfills this function already. It is used to persist, retrieve, and manage domain objects that are defined as entities or object hierarchies thereof. JPA's constraint that entities are required to define an identifier property fits the idea that DDD entities are identifiable within the business domain.

The entity manager is injected and used by services as follows:

```
@Stateless
public class InstrumentCraftShop {

    @Inject
    InstrumentMaker instrumentMaker;

    @PersistenceContext
    EntityManager entityManager;

    public ElectricGuitar craftInstrument() {
        ElectricGuitar instrument = instrumentMaker.build();
        return entityManager.merge(instrument);
    }
```

```
    public ElectricGuitar retrieveInstrument(long identifier) {
        return entityManager.find(ElectricGuitar.class,
          identifier);
    }
}
```

Factories

Creating domain objects might involve more-complex logic than just calling a constructor. To address this, DDD uses *factories*. The idea is to encapsulate the creation of complex objects into separate methods or classes.

If the creation of domain objects is tightly coupled to existing objects in the domain, it makes sense to define factories as methods of domain types. For the instrument craft shop example, I'll create some music instances based on instruments. I define a `Music` value object:

```
public class Music {

    private final String description;

    public Music(String description) {
        this.description = description;
    }

    public String getDescription() {
        return description;
    }
```

The creation of the value object is tightly bound to an instrument type and will, therefore, be placed as a method of the domain object type. The same holds true if the creation requires information about the actual instance that is contained in its properties.

```
public class ElectricGuitar {
```

```
    // ...

    public Music play() {
        return new Music("Let's rock!");
    }
}
```

CDI producers are another way to implement factories that are less coupled to specific domain objects. The CDI producer method or field exposes the instances, which are then injected into managed beans. The CDI producer, therefore, represents a factory.

Domain Events

Domain events occur during the execution of the business logic. They comprise semantics that are specific to the domain and usually emerge from the business use cases. Examples of domain event types are `InstrumentCrafted` or `ArticlePurchased`.

Domain events are implemented as value objects that contain the information associated with the event. In Java, you usually create domain events as immutable plain old Java objects. Because events already happened in the past, they shouldn't change later on. The `ElectricGuitarCrafted` type represents a domain event, implemented as regular Java object:

```
public class ElectricGuitarCrafted {

    private final Instant instant;
    private final Model model;

    public ElectricGuitarCrafted(Model model) {
        this.model = model;
        instant = Instant.now();
    }
```

```
    // getters
}
```

Java EE ships with functionality that allows you to fire and observe events with loose coupling, namely CDI events. CDI events are fired during the execution of the business logic:

```
@Stateless
public class InstrumentCraftShop {

    @Inject
    InstrumentMaker instrumentMaker;

    @Inject
    Event<ElectricGuitarCrafted> instrumentCreated;

    @PersistenceContext
    EntityManager entityManager;

    public ElectricGuitar craftInstrument() {
        ElectricGuitar instrument = instrumentMaker.build();

        instrumentCreated.fire(
            new ElectricGuitarCrafted(instrument.getModel()));

        return entityManager.merge(instrument);
    }

    // retrieveInstrument() ...
}
```

CDI's `Event<T>` type is injected into managed beans and is used to fire any defined events, such

as `ElectricGuitarCrafted`. The event will be handled in a CDI observer method, decoupled from the rest of the business logic:

```
public class CraftedBrandRecorder {

    public void onCraftedInstrument(
           @Observes ElectricGuitarCrafted event) {
        Model model = event.getModel();
        System.out.println(
            "new instrument crafted for model: " + model);
    }
}
```

Since the introduction of Java EE 8, it has been possible to handle events asynchronously, directly via CDI by using the `Event#fireAsyncmethod` and the `@ObservesAsync` annotation. The event handling is then executed in a separate thread.

Conclusion

Modern Java EE makes it possible to develop enterprise applications with a focus on the business logic. The technology doesn't set many constraints on the domain logic, as it did in J2EE. Domain classes don't have to extend or implement specific Java EE types. The easiest approach is to write the business logic in plain Java. The technical cross-cutting concerns are configured via annotations.

The flexibility of the CDI and JPA specifications enable developers to focus on what adds value to the application: the business logic. Note that Jakarta EE will be based on Java EE 8, so the concepts and ideas behind it and demonstrated here will hold true in the future. `</article>`

Sebastian Daschner (@DaschnerS) is a Java Champion, consultant, author, and trainer. He wrote the book *Architecting Modern Java EE Applications*, and he serves in the JAX-RS, JSON-P, and Config Expert groups. He also collaborates on multiple open source projects and is a double JavaOne Rockstar.

ORACLE CODE

Register Now

Oracle Code is BACK! | 1-Day, Free Event

Explore the Latest Developer Trends:

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

Live for the Code

Coming to a city near you:

developer.oracle.com/code

ORACLE®

RAOUL-GABRIEL **URMA**

RICHARD **WARBURTON**

var and Java 10's Expanded Type Inference

Best practices for using local variable type inference

Java 10 introduced a new shiny language feature called local variable type inference. Its principal goal is to reduce boilerplate and enhance code readability. It enables you to replace the type in a local variable declaration with the keyword `var`—the compiler fills in the appropriate type from the variable initializer. For example, this code:

```
Map<User, List<String>> userChannels = new HashMap<>();
```

can be rewritten in Java 10 as:

```
var userChannels = new HashMap<User, List<String>>();
```

In addition to concision, this inference of the type provides several advantages, which we explore in this article. Let's look at a more involved example:

```
Path path = Paths.get("src/web.log");
try (Stream<String> lines = Files.lines(path)){
    long warningCount =
        lines.filter(line -> line.contains("WARNING"))
             .count();
    System.out.println(
        "Found " + warningCount + " warnings in the log file");
} catch (IOException e) {
```

```
    e.printStackTrace();
}
```

can be refactored as follows in Java 10:

```
var path = Paths.get("src/web.log");
try (var lines = Files.lines(path)){
    var warningCount =
        lines.filter(line -> line.contains("WARNING"))
             .count();
    System.out.println(
        "Found " + warningCount + " warnings in the log file");
} catch (IOException e) {
    e.printStackTrace();
}
```

Each expression in this code still has a static type (that is, the declared type of a value), as follows:

- The local variable `path` is of type `Path`.
- The variable `lines` is of type `Stream<String>`.
- The variable `warningCount` is of type `long`.

This means that assigning a value of a different type will fail. For example, the reassignment in the following code will produce a compilation error:

```
var warningCount = 5;
warningCount = "6";

|  Error:
|  incompatible types: java.lang.String cannot be converted to int
|  warningCount = "6"
```

There's some small cause for concern with type inference. For example, given classes `Car` and `Bike` that subclass a class `Vehicle` and given the declaration `var v = new Car();` do you declare `v` to have type `Car` or `Vehicle`?

Here, a simple explanation that the missing type is the type of the initializer (`Car`, in this case) is perfectly clear, and it can be backed up with a statement that `var` may not be used when there's no initializer.

This means, however, that a later assignment of `v = new Bike();` stops working. In other words, polymorphic code doesn't play nice with `var`.

Where Can't You Use Local Variable Type Inference?

Where does local type inference not work? For starters, it only works with local variables. You cannot use it with fields or in method signatures. For example, the following is not possible:

```
public long process(var list) { }
```

You cannot use local variable declarations without an explicit initialization. This means you cannot just use the `var` syntax to declare a variable without a value.

The following

```
var x;
```

will return a compiler error:

```
|  Error:
|  cannot infer type for local variable x
|    (cannot use 'var' on variable without initializer)
|  var x;
|  ^----^
```

You cannot initialize a `var` variable to null either. Indeed, it is not clear what the type should be, because it's probably intended for later initialization.

```
|  Error:
|  cannot infer type for local variable x
|    (variable initializer is 'null')
|  var x = null;
|  ^-----------^
```

You also cannot use `var` with lambda expressions, because they require an explicit target type. The following assignment will fail:

```
var x = () -> {}
```

and generate this error message:

```
|  Error:
|  cannot infer type for local variable x
|    (lambda expression needs an explicit target-type)
|  var x = () -> {};
|  ^---------------^
```

Weirdly, though, the following assignment is valid, because there is an explicit initializer on the right side:

```
var list = new ArrayList<>();
```

What is the static type of `list`? The type of the variable inferred is `ArrayList<Object>`, which is not particularly useful because you don't benefit from generics. So, you might want to avoid writing this type of assignment.

Type Inference with Nondenotable Types

Java has several nondenotable types—that is, types that can exist within your program, but for which there's no way to explicitly write out the name for the types. A good example of a nondenotable type is an anonymous class—you can add fields and methods to it, but you won't be able to write the name of the anonymous class in your Java code. The diamond operator can't be used with anonymous classes. `var` is less restricted and can be used to support some nondenotable types—specifically, anonymous classes and intersection types.

The `var` keyword also enables you to use anonymous classes more effectively and refer to types that would otherwise be impossible to describe. Normally, if you create an anonymous class, you can add fields to it, but you can't refer to those fields elsewhere because they need to be assigned back to a named type.

For example, the following code won't compile, because the type of `productInfo` is an `Object` and you can't access fields `name` and `total` of an `Object`:

```
Object productInfo = new Object() {
    String name = "Apple";
    int total = 30;
};

System.out.println(
    "name = " + productInfo.name + ", total = " + productInfo.total);
```

With `var`, you can overcome this limitation. When you assign an anonymous class to a `var` typed local variable, the compiler infers the type of the anonymous class, rather than the type of its parent. This means that you can refer to fields declared in the anonymous class, as illustrated in the following code:

```
var productInfo = new Object() {
    String name = "Apple";
    int total = 30;
```

```
};

System.out.println(
    "name = " + productInfo.name + ", total = " + productInfo.total);
```

This capability might initially seem like an interesting piece of language trivia that has very little use, but it can be handy in certain circumstances. Sometimes, for example, you want to return a few values as an intermediate result inside some method. Normally, you would need to create and maintain a new class for this purpose just to use it inside a single method. For example, inside the `Collectors.averagingDouble()` implementation, a small array of double values is used for this purpose.

There's a better approach that you can take with `var`: using an anonymous class as a store for intermediate values.

Consider a case where you have some products, each of which has a name, a stock count, and a per-item monetary worth (that is, a value) associated with it. You want to calculate the total cost for each item—in other words, the count multiplied by the value. If that were the only piece of information you had, you could just map each product to its cost, but to do something useful with the result, you would also want the product's name.

The following is an example of how you can do that with `var` in Java 10:

```
var products = List.of(
    new Product(10, 3, "Apple"),
    new Product( 5, 2, "Banana"),
    new Product(17, 5, "Pear"));
var productInfos = products
    .stream()
    .map(product -> new Object() {
        String name = product.getName();
        int total = product.getStock() * product.getValue();
    })
```

```
    .collect(toList());
productInfos.forEach(prod -> System.out.println(
    "name = " + prod.name + ", total = " + prod.total));
```

This code outputs the following:

```
name = Apple, total = 30
name = Banana, total = 10
name = Pear, total = 85
```

Not all nondenotable types can be used with `var`. Anonymous classes and intersection types are supported. However, wildcard captured types are not inferred so as to avoid even more-cryptic wildcard-related error messages being presented to Java programmers. The goal of supporting nondenotable types was to retain as much information as possible in the inferred type and allow people to insert local variables and refactor more code. The original intent of this feature wasn't to write code like the previous code, but simply to solve the problem of how `var` should deal with nondenotable types. Whether the use of `var` with nondenotable types will become niche trivia or commonplace is hard to predict.

Recommendations

Type inference definitely reduces the amount of time it takes to write Java code, but what about readability? Developers spend much more time reading source code than writing it, so you should definitely be optimizing for ease of reading over ease of writing. The extent to which `var` improves readability is subjective: some developers will hate `var` and some will love it. You should always focus on what helps your teammates read your code. So, if they are happy reading code with `var`, you should use it; otherwise, you should not.

Sometimes, including explicit types can impede readability. For example, when looping over the `entrySet` of a `Map`, you need to regurgitate the type parameters on the `Map.Entry` object.

The following code is an example of looping over a `Map` from a country name to the names of the cities within the country:

```
Map<String, List<String>> countryToCity = new HashMap<>();
// ...
for (Map.Entry<String, List<String>> citiesInCountry :
    countryToCity.entrySet()) {
        List<String> cities = citiesInCountry.getValue();
    // ...
}
```

You could rewrite the code above with `var` and reduce the repetition and boilerplate, as follows:

```
var countryToCity = new HashMap<String, List<String>>();
// ...
for (var citiesInCountry : countryToCity.entrySet()) {
    var cities = citiesInCountry.getValue();
    // ...
}
```

There isn't just a readability advantage here, though—there's also an advantage in terms of maintaining the code.

For example, if you take similar code that has explicit types and replace the `String` representing the name of the city with a `City` class that could contain additional information about the city, you need to rewrite all the code that is relying on that specific type being exposed.

```
Map<String, List<City>> countryToCity = new HashMap<>();
// ...
for (Map.Entry<String, List<City>> citiesInCountry :
    countryToCity.entrySet()) {
```

```
        List<City> cities = citiesInCountry.getValue();
        // ...
}
```

However, if you had used the `var` keyword and type inference, you would need to alter the first line of code, as follows, but you would not need to alter the other lines:

```
var countryToCity = new HashMap<String, List<City>>();
// ...
for (var citiesInCountry : countryToCity.entrySet()) {
    var cities = citiesInCountry.getValue();
    // ...
}
```

This example illustrates a key principle to follow when using `var`: Don't optimize for ease of writing or for ease of reading; optimize for ease of maintenance. If you optimize for ease of maintenance, that should balance readability with the amount of code that needs to be changed as your program evolves over time.

It would be foolhardy to claim that adding type inference is always a positive for your code—sometimes having explicit types in code can help readability. This is particularly the case when the type isn't obvious from the expression that generates it. In the following code, it would be better to have explicit types, because you don't know from just reading the `getCities()` method call what it is returning:

```
Map<String, List<City>> countryToCity = getCities();
var countryToCity = getCities();
```

This discussion leads to one final recommendation when it comes to readability and `var`: variable names matter! Because `var` removes the ability of the reader of your code to guess at the

code's intent simply from the type of the variable, there is more of a burden on you as the developer to provide good names for local variables. In theory, that's something Java developers should already be doing. In practice, though, many readability problems in Java code aren't related to new language features, but rather to existing practices, such as variable naming.

Type Inference and IDEs

One commonly used feature that many IDEs provide is the ability to extract a local variable, and in doing so they will infer the correct type of that variable and write it out for you. That feature has some overlap with the `var` keyword in Java 10. Both the IDE feature and `var` remove the need to write out the type explicitly, but they otherwise have different trade-offs.

The extract-local feature generates a local variable with the full, explicit type written out in your code, whereas `var` removes the need to have the explicit type written out in your code. So although they both have similar value in terms of simplifying the writing of code, `var` alters readability in a way that the extract-local feature does not. As mentioned before, `var` is mostly a readability benefit, but sometimes it can be a hindrance.

In Java 11, to be released a mere six months after Java 10, the var keyword will be allowed within the parameters of a lambda expression.

Java Compared to Other Programming Languages

Java isn't the only or even the first language to include type inference for variables. In fact, the type inference introduced in Java 10 with `var` is a very limited and restricted form of type inference. It keeps the approach simple and also ensures that compiler errors related to `var` declarations are restricted to a single statement, because the `var` inference algorithm looks only at the expression being assigned to the variable in order to deduce the type.

Conclusion

`var` is a helpful addition to the Java language in terms of productivity and readability, but the fun doesn't stop there. Future versions of Java will continue the steady evolution and modernization of the language. For example, in Java 11, to be released a mere six months after Java 10 and with long-term support, the `var` keyword will be allowed within the parameters of a lambda expression. This is useful because it allows you to have a formal parameter whose type is inferred, but onto which you can still add Java annotations, for example the following:

```
(@Nonnull var x, var y) -> x.process(y)
```

Other ideas that have been implemented in functional programming languages and are ready for the mainstream will be working their way into future Java versions—for example, pattern matching and value types. This doesn't mean that these improvements will stop Java from being the Java that developers know and love. It'll just be more flexible, readable, and concise than ever before. `</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.

BEN EVANS

CHRIS NEWLAND

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

Lock Elision in the JVM

How the compiler's escape analysis removes unnecessary locks

One of the optimizations made possible by escape analysis (EA)—an analysis done by the compiler, which we discussed in detail in our previous article—is the removal of locks. This removal can take place when it can be proved that the object on which a lock is acquired does not escape the local scope. Such a situation means that the object can be accessed only by a single thread, so there is no need to exclude other threads from accessing it. Therefore, the lock can be removed. This is known as *lock elision,* which is the topic of this article in our long-running series on the mechanics of JVM operations.

An example of lock elision can be demonstrated using a well-known thread-safe class that uses synchronized methods, `java.lang.StringBuffer`. `StringBuffer` was included in Java 1.0 to allow more efficient concatenation of immutable string objects. Each of its `append()` methods is synchronized to enable the string under construction to be created safely when multiple threads are writing to the same `StringBuffer` object.

Many programs do not need this thread safety, so in Java 5 the `java.lang.StringBuilder` class was introduced to provide an unsynchronized alternative to `StringBuffer`. Both classes inherit from the package-private `java.lang.AbstractStringBuilder` and have very similar implementations of `append()`.

The main difference is the synchronization behavior of `StringBuffer`:

```
@Override
public synchronized StringBuffer append(String str) {
    toStringCache = null;
    super.append(str);
```

```
    return this;
}
```

Compare that to the `StringBuilder` form:

```
@Override
public StringBuilder append(String str) {
    super.append(str);
    return this;
}
```

A thread calling an `append()` method on a `StringBuffer` must acquire that object's intrinsic lock before entering the method, and it must release the lock upon exit. `StringBuilder` does not need to do this work, so the class should outperform a `StringBuffer`—at least based on first appearances.

After escape analysis was added to the HotSpot JVM, calls to synchronized methods on objects such as `StringBuffer` can have their locks removed automatically. This is possible only on objects that are created within the scope of a method and that can be shown not to escape.

Timing of Java operations is universally performed using the Java Microbenchmark Harness (JMH). Let's look at a JMH benchmark to see how modern JVMs can narrow the performance difference by eliding the `StringBuffer` locks when it can be proven that only a single thread can access the `StringBuffer` object:

```
@State(Scope.Thread)
@BenchmarkMode(Mode.Throughput)
@OutputTimeUnit(TimeUnit.SECONDS)
public class StringBufferLockElision {

    private static final String[] pieces =
        new String[]{"a", "b", "c", "d", "e"};
```

```
@Benchmark
public String concatWithStringBuffer() {
    final StringBuffer buffer = new StringBuffer();

    for (String piece : pieces) {
        buffer.append(piece);
    }

    return buffer.toString();
}

@Benchmark
public String concatWithStringBuilder() {
    StringBuilder builder = new StringBuilder();

    for (String piece : pieces) {
        builder.append(piece);
    }

    return builder.toString();
  }
}
```

Lock elision has been a very successful optimization and is enabled by default as of Java 8, but it can be disabled using the `-XX:-DoEscapeAnalysis` VM switch, so that you can see the impact of the optimization. With escape analysis enabled (the default), the performance of `StringBuffer` and `StringBuilder` is almost identical. (The results are reported in operations per second. A higher score indicates better performance.)

```
concatWithStringBuffer  16280252.994 ± 17K  ops/s
concatWithStringBuilder 16479504.748 ± 34K  ops/s
```

As shown below, without escape analysis, the `StringBuffer` code is around 15% slower—and this difference is due solely to the cost of locking on the `append()` method calls.

```
.concatWithStringBuffer  12385164.076 ± 58K  ops/s
.concatWithStringBuilder 14570548.284 ± 55K  ops/s
```

Lock Coarsening

The HotSpot JVM contains additional optimizations for locks that are not technically part of the escape analysis subsystem but that also use analysis of scope to improve intrinsic lock performance. When consecutive locks on the same object are encountered, the HotSpot JVM will check whether it is possible to enlarge the locked region by combining the locked regions into a single, larger region. This aggregation can eliminate some of the locking and unlocking overhead and is called *lock coarsening*.

When the HotSpot JVM encounters a lock, it will search backward to try to find an unlock operation on the same object. If a match is found, it will consider whether the two lock regions can be joined and the paired unlock/lock actions removed.

Let's look at a program with consecutive regions that are locked by the same object's monitor:

```
public class CoarsenedLocks {
    public static void main(String[] args) {
        new CoarsenedLocks();
    }

    private java.util.Random random = new java.util.Random();
    private static final Object lock = new Object();

    public CoarsenedLocks()
    {
        long sum = 0;
```

```
            for (int i = 0; i < 1_000_000; i++) {
                synchronized (lock) {
                    sum += random.nextInt();
                }

                synchronized (lock) {
                    sum -= random.nextInt();
                }
            }
            System.out.println(sum);
        }
    }
```

The bytecode for this method is rather verbose and looks like this:

```
public optjava.CoarsenedLocks();
    descriptor: ()V
    flags: ACC_PUBLIC
    Code:
      stack=5, locals=5, args_size=1
         0: aload_0
         1: invokespecial #3                  // Method java/lang/Object."<init>":()V
         4: aload_0
         5: lconst_0
         6: putfield      #4                  // Field sum:J
         9: iconst_0
        10: istore_1
        11: iload_1
        12: ldc           #5                  // int 1000000
        14: if_icmpge     73
        17: aload_0
```

```
18: dup
19: astore_2
20: monitorenter
21: aload_0
22: dup
23: getfield      #4                  // Field sum:J
26: lconst_1
27: ladd
28: putfield      #4                  // Field sum:J
31: aload_2
32: monitorexit
33: goto          41
36: astore_3
37: aload_2
38: monitorexit
39: aload_3
40: athrow
41: aload_0
42: dup
43: astore_2
44: monitorenter
45: aload_0
46: dup
47: getfield      #4                  // Field sum:J
50: lconst_1
51: lsub
52: putfield      #4                  // Field sum:J
55: aload_2
56: monitorexit
57: goto          67
60: astore        4
```

```
62: aload_2
63: monitorexit
64: aload         4
66: athrow
67: iinc          1, 1
70: goto          11
    // Field java/lang/System.out:Ljava/io/PrintStream;
73: getstatic     #6
76: aload_0
77: getfield      #4                  // Field sum:J
    // Method java/io/PrintStream.println:(J)V
80: invokevirtual #7
83: return
```

[The comment lines near the end of the listing each refer to the succeeding line of output. —*Ed.*]

Recall that the relevant bytecodes for operating on intrinsic locks are `monitorenter` and `monitorexit`.

In the bytecode, for each `monitorenter` instruction, there are two `monitorexit` instructions, each taking a different execution path. This is because the first `monitorexit` releases the monitor upon a normal exit from the locked region and the second `monitorexit` releases the lock upon an abnormal exit from the region.

This set of bytecodes might look odd, because in the source code the only operation performed within the synchronized region is an increment to a primitive `int` variable. This action cannot throw an exception, but there is a possibility of an abnormal termination of the locked region. (That can occur if the thread receives an `InterruptedException` if, for instance, the `stop()` method were invoked on the executing thread. For this reason, there is a second path to ensure the monitor is always released, even in the event that an unchecked exception is thrown. You can read more about this in the JVM specification.) The lock coarsening optimization is enabled by default but can be disabled using the VM switch `-XX:-EliminateLocks`.

Nested Locks

One synchronized block can be nested inside another, and it is perfectly possible for both blocks to synchronize on the same object monitor. The HotSpot JVM is able to detect this case, which we refer to as *nested locks*, and it can remove the inner locks. This removal is possible because a thread will acquire the lock as it enters the outer block, and so will definitely still be holding it when the thread tries to enter the inner block.

At the time of writing, the nested lock elimination in Java 8 appears to work only with locks that are declared as `static final` or with locks on `this`.

The following example shows an inner lock that is eliminated when nested synchronized blocks are encountered:

```
public class NestedLocks {
    public static void main(String[] args) {
        new NestedLocks();
    }

    private java.util.Random random = new java.util.Random();

    private static final Object lock = new Object();

    public NestedLocks()
    {
        long sum = 0;

        for (int i = 0; i < 1_000_000; i++) {
            synchronized (lock) {
                sum += random.nextInt();

                synchronized (lock) {
                    sum -= random.nextInt();
                }
```

```
            }
        }
        System.out.println(sum);
    }
}
```

The HotSpot JVM is able to eliminate the inner nested lock, so the code will effectively become this:

```
for (int i = 0; i < 1_000_000; i++) {
    synchronized(lock) {
        sum += random.nextInt();

        sum -= random.nextInt();
    }
}
```

The nested lock optimization is enabled by default but can be disabled using the VM switch `-XX:-EliminateNestedLocks`.

Arrays and Escape Analysis

Escape analysis, like other optimizations, is subject to trade-offs because every allocation not made on the heap must happen somewhere on the stack or in CPU registers, both of which are relatively scarce resources. One limitation in the HotSpot JVM is that by default, arrays of more than 64 elements will not benefit from escape analysis. This size is controlled by the VM switch `-XX:EliminateAllocationArraySizeLimit=n`, where `n` is the number of elements.

Consider a hot code path that contains a temporary array allocation to read from a buffer. If the array does not escape the method scope, escape analysis should prevent the heap allocation. However, if the array length is more than 64 elements (even if they are not all used), it must be stored on the heap. This restores the heap allocation and defeats escape analysis for the array.

In the following JMH benchmark, the test methods allocate nonescaping arrays of 63, 64, and 65 elements. (The array size of 63 is tested to ensure that 64 is not faster than 65 simply because of memory alignment.)

In each test, only the first two array elements, `a[0]` and `a[1]`, are used. Note, however, that the limitation on escape analysis is dependent only on the array length, not on how many elements of the array are actually used.

```
@State(Scope.Thread)
@BenchmarkMode(Mode.Throughput)
@OutputTimeUnit(TimeUnit.SECONDS)
public class EscapeTestArraySize {

    private java.util.Random random = new java.util.Random();

    @Benchmark
    public long arraySize63() {
        int[] a = new int[63];

        a[0] = random.nextInt();
        a[1] = random.nextInt();

        return a[0] + a[1];
    }

    @Benchmark
    public long arraySize64() {
        int[] a = new int[64];

        a[0] = random.nextInt();
        a[1] = random.nextInt();
```

```
        return a[0] + a[1];
    }

    @Benchmark
    public long arraySize65() {
        int[] a = new int[65];

        a[0] = random.nextInt();
        a[1] = random.nextInt();

        return a[0] + a[1];
    }
}
```

The results show a large drop in performance once the array allocation cannot benefit from the escape analysis optimization. (Once again, the scores show operations per second, so higher scores indicate better performance.)

```
EscapeTestArraySize.arraySize63  49824186.696 ±  9K  ops/s
EscapeTestArraySize.arraySize64  49815447.849 ±  2K  ops/s
EscapeTestArraySize.arraySize65  21115003.388 ± 34K  ops/s
```

If you find that you need to allocate a larger array in hot code, you can instruct the VM to allow larger arrays to be optimized. Running the benchmark again with a limit of 65 elements shows that performance is restored.

Using this command line:

```
java -XX:EliminateAllocationArraySizeLimit=65 -jar target/benchmarks.jar
```

You get the following results:

```
EscapeTestArraySize.arraySize63  49814492.787 ± 2K  ops/s
EscapeTestArraySize.arraySize64  49815595.566 ± 6K  ops/s
EscapeTestArraySize.arraySize65  49818143.279 ± 2K  ops/s
```

Notice the parity in the results.

Conclusion

This article and the previous one on escape analysis show some of the magic that occurs behind the scenes in the HotSpot JVM. They also convey some of the complexity of the operations. Each major release of Java tends to add new features to the JVM.

In fact, Oracle is investing in a new generation of compiler technology. Known as Graal, it is a just-in-time (JIT) compiler that is pluggable, user-extensible, and written in Java. It is a major effort within Project Metropolis, which aims to build as much of a runtime in Java as is practical.

The Graal compiler is an experimental addition to Java 10, as described in JEP 317. The principal aim is to provide a way for developers and specialist platform owners to write their own JIT compilers to meet their specific needs. Graal is a very amenable environment in which to introduce and prototype optimization techniques.

This article and its predecessor described scope analysis that enables a variety of optimization techniques. First among these is allocation elimination (that is, scalar replacement), but related locking techniques were discussed as well. These are just some examples of JIT compilation techniques that are present in the mature C2 compiler in the HotSpot JVM. Upcoming articles will examine other techniques that the HotSpot JVM uses to improve code performance. </article>

Ben Evans (@kittylyst) is a Java Champion, a tech fellow and founder at jClarity, an organizer for the London Java Community (LJC), and a member of the Java SE/EE Executive Committee.

Chris Newland (@chriswhocodes) is a Java Champion. He invented and still leads developers on the JITWatch project, an open source log analyzer to visualize and inspect just-in-time compilation decisions made by the HotSpot JVM.

PERU JUG

The Peru Java User Group, PERUJUG, was founded in March 2006. Since then, the community has been organizing monthly meetups as well as an annual conference called Java Day Peru, in which Java Champions, JUG leaders, and developers from around the world participate.

Recent Java Day Peru topics, for example, included "Agile, DevOps, Cloud," with Eddú Meléndez of Peru; "Java EE with Apache TomEE," with César Hernández from Guatemala; "JMoordb NoSQL," with Aristides Villarreal from Panama; "Front End for Back-End Developers," with José Díaz from Peru; and "API Design," with Jorge Vargas of Mexico. Thanks to the collaboration of different JUGs and the promising initiative JEspañol (Java developers who speak Spanish) that made the event possible, Java Day Peru was awarded a Duke's Choice award in 2017.

The current conference takes place in Lima, Peru, on June 30 at the San Marcos University (UNMSM) and includes trending topics such as microservices, serverless, DevOps, and microprofile.

Today, more than 1,500 JUG members participate in Peru's Java community. For communication outside meetings, PERUJUG uses Facebook, Twitter (@perujug), Slack.perujug.org, and meetups to spread news and knowledge. Please contact PERUJUG in advance if you are a member of another JUG, a Java Champion, or a technology evangelist coming to Peru, so that the group can host you and arrange some conversations.

SIMON ROBERTS

MIKALAI ZAIKIN

Quiz Yourself

More intermediate and advanced test questions

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

These questions rely on Java 8. We'll begin covering later releases of Java in future columns, of course, and we will make that transition quite clear when it occurs.

Question 1 (intermediate). The objective is to import other Java packages to make them accessible in your code. Given the following code:

```
package sys;
// line n1
class Computer {
    public void handleOutput(Printer p) {
        printJob("Computer");
    }
}
```

And given this code:

```
package sys;
class Printer {
```

```
    public static void printJob(String s) {
        System.out.println(s);
    }
}
```

Which is true?

A. The code compiles and executes without error as written.

B. The code compiles and executes without error if the following is added at line n1:
```
import static sys.Printer.printJob;
```

C. The code compiles and executes without error if the following is added at line n1:
```
import sys.Printer;
```

D. The code compiles and executes without error if the following is added at line n1:
```
import Printer.printJob;
```

E. The code compiles and executes without error if the following is added at line n1:
```
import sys.Printer.printJob;
```

Question 2 (intermediate). Given these classes:

```
public class Base {
    Base doStuff(int val){ return null; }
}

class Sub extends Base {
    @Override
    private Sub doStuff(int val) // line n1
    { return null; }
}
```

Which of the following are true? Choose two.

A. The classes compile successfully as shown.

B. The classes would compile successfully if the method declaration at line 1 were changed to this:

```
public Base doStuff(int val) throws RuntimeException
```

C. The classes would compile successfully if the method declaration at line 1 were changed to this:

```
public Sub doStuff(int val)
```

D. The classes would compile successfully if the method declaration at line 1 were changed to this:

```
Base doStuff(int val) throws Exception
```

E. The classes would compile successfully if the method declaration at line 1 were changed to this:

```
private Base doStuff(int val)
```

Question 3 (advanced). Given this code:

```
abstract class Stuff {
    int x;
    public Stuff(int x) { this.x = x; }
    public Stuff() { this(18); }
    public abstract void doStuff();
}
```

Which of the following snippets (taken individually) compiles? Choose two.

A.

```
(new Stuff(9) {
    public void doStuff() {
```

//fix this /

```
        System.out.println("x is " + x);
}}).doStuff();
```

B.

```
(new Stuff() {
    public void doStuff() {
        System.out.println("x is " + x);
}}).doStuff();
```

C.

```
(new Stuff() {
    ()->System.out.println("x is " + x);
}).doStuff();
```

D.

```
Stuff s = ()->System.out.println("doStuff!");
```

Question 4 (advanced). The objective here is to use method references with streams. Given the following code:

```
class Converter {
    Integer inc(Integer i) {
        return i + 1;
    }
}
```

And given this code:

```
List<Integer> ls = Arrays.asList(1,2,3);
```

Which code fragment will create a list of incremented integers?

A.

```
ls.stream().map(Converter::new::inc).collect(Collectors.toList());
```

B.

```
Converter c = new Converter();
ls.stream().map(c::inc).collect(Collectors.toList());
```

C.

```
ls.stream().map(Converter::inc).collect(Collectors.toList());
```

D.

```
ls.stream().map(i -> Converter::inc(i)).collect(Collectors.toList());
```

Answer 1. The correct answer is option B. This question investigates Java's static import feature. This feature, which was introduced in Java 1.5, allows static elements from one class to be used by their unqualified, or "short," names in another source file. For example, given an import of the following form,

```
import static java.lang.Math.PI;
```

a source file can simply refer to `PI` instead of, potentially repetitively, referring to `Math.PI`.

This feature can be particularly useful when the static elements that are being imported have a meaning that is very well understood in the scope of the program, such that the meaning doesn't benefit from the scoping effect of using a class prefix. Clearly, referring to `Math.PI` is

cumbersome when compared with simply writing `PI`, and the longer form doesn't improve readability in any way; no one will be unsure what `PI` means. Of course, like any syntax feature that shortens code, it should be used judiciously. If a static import hides helpful context, it might make the code harder to understand, and in that case it should probably be avoided.

The syntax of a static import uses the two keywords, `import static`, in that order, followed by the fully qualified class name and either the static element to be imported or the asterisk, indicating that all static elements of the class are to be made available. Here are examples of both approaches:

```
import static my.package.MyClass.member;
```

or

```
import static my.package.MyClass.*;
```

Notice that although this feature is called a static import, the syntax actually has the keywords in the opposite order—that is, `import static`. Take care that this doesn't cause you to trip in a real exam situation.

Now, let's consider the options. Option A suggests that the code compiles as written. However, the `printJob` method is out of scope in the `Computer` class, and for the code presented in the question, the compiler rejects `printJob` with a complaint along the lines of "cannot find symbol." Therefore, option A is incorrect.

Option B correctly applies the syntax rules just described, and it succeeds in making the `printJob` method available by that short name. Option B is, therefore, the correct answer.

In option C, the suggested change constitutes a regular import of the `Printer` class. However, there are two problems here. First, such an import is redundant, because the two classes are in the same package and, therefore, they are visible to one another anyway. The second problem is that importing a class in the simple way does not make the static members available in their unqualified form in the file that performs the import. Instead, a simple import means

that the imported class can be used without its full qualifying package prefix. So, whether it is imported or not, the `Computer` class would compile correctly if the call to `printJob` had been qualified as `Printer.printJob`. However, the proposed change of option C does not address the problem. Therefore, option C is incorrect.

Nonstatic imports can import only a class (or, using the asterisk format, all the classes) from a package.

Option D introduces an additional compilation error, without doing anything to repair the original problem. In Java, any import statement *must* contain a fully qualified package name. Other elements may be chained on that package name also. In this case, however, option D declares an import path that starts at a class, instead of at a root package, and that's an error in itself. There's another problem in option D, too, which is that the import is not of the static variety. Nonstatic imports can import only a class (or, using the asterisk format, all the classes) from a package. Given these two problems, it's clear that option D is incorrect.

Option E attempts to use the nonstatic form of `import`. However, as just mentioned, such a statement can import only classes, not elements from inside them. Only a static import is able to import the contents of a class, rather than just the class itself. Because of this, option E is also incorrect.

Answer 2. The correct answers are option B and option C. This question investigates the nature of overriding and the syntactic requirements of the Java programming language. Essentially, three types of variations are proposed: changes in accessibility, a change of return type, and changes in the exception behavior of the method.

When a method overrides an existing method in a parent class or in another generalized form (such as an interface declaration), the basic requirement is that the replacement method must be a valid syntactic replacement for the original. This requirement is one aspect of the Liskov Substitution Principle. Essentially, a specialized object (such as an instance of the `Sub` class in this example) must be a proper substitute for the generalized object (an instance of the `Base` class in this example). This is because you could write code like the following:

```
Base b = new Sub();
```

And, if you did, the compiler expects the object referred to by the variable `b` to behave according to the rules defined for a `Base` object. The fact that the variable `b` actually refers to an instance of `Sub` is allowed, but it must not cause surprises.

In the light of this background, let's look at the three categories of change individually.

First, consider the accessibility of a method. The `Base` class gives default access to the method `doStuff`. If an override makes the method *more* accessible, no surprises will arise; the method will still be accessible in the places you expect it to be accessible. In fact, if you had a reference of `Sub` type, the `doStuff` method would be more accessible, but that is fine too. The thing referred to by the variable `b` lets you do what you expect to do. So making the method `public` (or, generally, making an overriding method *more* accessible) is acceptable.

By contrast, making the method `private` would cause a problem. Using the variable `b`, you expect to be able to invoke the method `doStuff`, but if the object to which `b` refers were actually an instance of `Sub`, the invocation would be rejected by the `Sub` object, and it would fail at run-time. This would definitely be a surprise and is, therefore, not acceptable. So the general rule is that a method may be overridden by another method that is *not* less accessible. This rule is sufficient to tell you that option A and option E are incorrect.

Considering exceptions, Java's exception mechanism requires that any checked exception that might be thrown by a method must be declared. This is required so that the compiler knows about the exception and is able to verify that the caller of the method handles the exception (either by using a `catch` block or by declaring that it, too, throws the exception). Imagine a situation based on the assignment shown in question 2 in which the overriding method in `Sub` throws a checked exception. Code of the form `b.doStuff()` would be analyzed by the compiler, using the information defined in the `Base` class. The compiler would decide that no checked exceptions are possible, and it would not require the programmer to add a `try/catch` or any other exception handling. However, given that the variable `b` actually refers to a `Sub` object, if the overriding method did in fact throw a checked exception, the compiler would have been

cheated, and the rules of checked exceptions would have been bypassed. Because of this, an overriding method (including methods that implement interface methods) is prohibited from throwing checked exceptions that are not permitted for the base method. Based on this prohibition, option D must be incorrect.

This rule, however, relates only to checked exceptions. Any piece of code might throw an unchecked exception (a `RuntimeException` or an `Error`) and as such, while it's unnecessary to declare such exceptions, it's also irrelevant whether they're declared. Therefore, it's permissible to declare them on an overriding method, even if the base method doesn't mention them.

It's not possible to use a lambda to create a concrete implementation of an abstract class, even if it has exactly one abstract method.

The return of a method will often be assigned to a variable, and the compiler, of course, ensures that the assignment is acceptable. Given that the `doStuff` method in the `Base` class declares that it returns a `Base` object, that variable would have to accept assignment from an object of `Base` type. Of course, if the method actually returns a `Sub`, that's still assignment-compatible with the `Base` type and, therefore, is permitted. Given that the return type and accessibility have changed in acceptable ways, option C is correct.

There are some side notes on this assignment compatibility of overriding method return types. First, this was not the case in the earliest versions of Java, so you will sometimes find reference material that appears to contradict it. However, it's been true for many years now.

Another side note is that this flexibility relates only to object type returns. Primitive values cannot be used in this way. The final side note is that this kind of type variation is called a *covariant* return.

In option B, the overriding method is more accessible and—redundantly—declares throwing a `RuntimeException`. These changes are OK, and no other changes are made. Therefore, option B is correct.

Answer 3. The correct answers are option A and option B. This question is essentially about things you can't do with lambda expressions. Both of the first two options successfully create objects from anonymous inner classes that subclass the abstract class `Stuff`.

Although it might be more common to create anonymous classes that are constructed with zero arguments, it's entirely possible to use arguments in the construction process, provided the parent class defines the target constructor. This requirement means that it's not possible to have constructor arguments for anonymous inner classes that are defined in terms of an interface, because no "parent" constructors can exist in an interface. Therefore, only the zero-argument form is possible. It's also not possible to define an explicit constructor in the anonymous inner class itself. However, the code in both option A and option B interacts properly with the available constructors of the `Stuff` parent class.

In fact, both option A and option B compile correctly and execute successfully, and the output `x` is 9 for option A and 18 for option B. As a result, both option A and option B are correct.

Option C attempts to use a lambda expression to define the implementation of the abstract method in the `Stuff` class. However, the syntax presented is entirely bogus, and option C is incorrect.

Option D is more plausible than option C. However, it is only permissible to define lambda expressions to satisfy the behavioral requirement of a functional interface. A functional interface is one that has exactly one abstract method—and, of course, the class `Stuff` has exactly one abstract method, which might make this option tempting. However, a functional interface is an interface, and it's not possible to use a lambda to create a concrete implementation of an abstract class, even if it has exactly one abstract method. Therefore, option D is incorrect.

Answer 4. The correct answer is option B. This question investigates the method reference syntax introduced with lambda expressions in Java 8. Method references are an alternative syntax that addresses one of four situations that occur fairly frequently in lambda expressions.

Four forms are available. **Table 1** shows how a lambda in the conventional "arrow" form may be translated to or from the method reference form.

FORM	CONVENTIONAL LAMBDA	METHOD REFERENCE FORM
1	`(a, …) -> SomeClass.aMethod(a, …)`	`SomeClass::aMethod`
2	`(a, …) -> anObject.aMethod(a, …)`	`anObject::aMethod`
3	`(a, …) -> new SomeClass(a, …)`	`SomeClass::new`
4	`(a, …) -> a.aMethod(…)`	`SomeClass::aMethod`

Table 1: Four method reference forms for lambdas

Notice that in forms 1 through 3 in Table 1, the argument list of the lambda expression is passed entirely unchanged to the argument list of the method call on the right of the arrow. In form 4, this convention changes, but only slightly; the first argument to the lambda is used as the `this` object on which the behavior is invoked, and any subsequent arguments are passed to the argument list of the invoked method.

Four points should be emphasized about this translation and the argument-list handling that goes with it. First, in no case does a method reference permit the arguments to be altered in any way, whether by calculation or by changing their order.

Second, in forms 1 through 3, the syntax can be used with lambdas that implement interface methods requiring any number of arguments, including zero arguments. In form 4, the syntax is workable only if the interface method being implemented requires at least one argument, because the first argument becomes the `this` object for the method invocation on the right of the lambda's arrow. Clearly, you can't say `x.doStuff()` unless you have an `x`.

Third, the argument list and the return type of the lambda or method reference form are entirely mandated by the interface method that is being provided. For example, if the context requires that the lambda provide a `BiFunction<Fruit, Color, Ranking>`, this demands that the lambda take two arguments, a `Fruit` first and a `Color` second, and that it return a `Ranking`. Method references do not bypass this standard rule; when you use them, you still must provide behaviors that are suitable for the context.

Finally, the method reference syntax forms are very specific and cannot be altered in any way. On the left of the double colon is some kind of object reference or class identity, and on the

right is a method name or the keyword `new`. An object identity can be any expression that results in a single object, and a class name might be fully qualified. So sometimes there's an expression or package dot syntax on the left of the double colon, but none of these variations allows two sets of double colons to be strung together in the way shown in option A. As a result, option A is incorrect.

In option C, the form `Converter::inc` could match either a static method in the `Converter` class or an instance method in that class, if the method is called `inc`. However, no such static method exists, and, for the instance method form, the first argument to the lambda would have to be of type `Converter`. However, in this stream code, the `map` operation requires a `Function<T,U>` where `T` (the first argument to the lambda) is the type of the stream data. In this code, the stream contains `Integer` objects, not `Converter` objects. As a result, you can determine that option C is also incorrect.

Option D proposes a syntax that involves parentheses after the method name in the reference. No such syntax exists and, therefore, option D is incorrect.

In option B, everything works as needed. The `map` method requires a `Function<T,U>` where `T` is the upstream data type (`Integer`, in this case) and `U` defines the downstream type after the `map` operation (also `Integer`, in this case). Because `c` is an instance of `Converter`, the syntax `c::inc` is of form 4. Therefore, it translates to `x -> c.inc(x)`. This fits the required form `Function<Integer, Integer>` and, therefore, option B is correct. `</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 publishes recorded and live video training through Pearson InformIT (available direct and through the O'Reilly Safari Books Online service). He remains involved with Oracle's Java certification projects.

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

//contact us /

Comments

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

Article Proposals

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

Customer Service

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

Where?

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

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

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