Java™ magazine

By and for the Java community

JULY/AUGUST 2018

LIBRARIES

CHOOSING THE RIGHT ONE

ORACLE.COM/JAVAMAGAZINE

ORACLE®

RogueWave SOFTWARE | JRebel

Would you rather be waiting... or coding?

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

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

Productivity bottlenecks lead to tradeoffs
in quality, performance & security

Download the ebook!

roguewave.com/java-bestpractices

Top 5 best practices for streamlining Java development

//table of contents /

BUILD COMPLEX PDFs EASILY

By Bruno Lowagie and Joris Schellekens

How to use the popular Java library iText to create and manipulate PDF files

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

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

Giving Expert Users What They Need

The case for including embedded scripting engines in your apps

Much of the attention today on the "user experience" focuses on the preferences of consumers and nonexpert users. When vendors deal with expert users, usability of applications takes a different form: can experts make the software do what they need it to do? For many applications and packages, the answer is wrapped in all kinds of limitations driven by the vendor's perception of expert needs. But there is one undervalued option that can in many cases guarantee the expert's ability to get work done: an embedded scripting engine. Applications with a user-accessible scripting option are expert-friendly; those lacking them are not.

Scripting is the province of skilled users because it requires some grounding in programming and time invested in learning the details of a tool's internal structure. The benefit of application programmability has long been recognized. The most common instance is surely Visual Basic for Applications (VBA), which enables the writing of sophisticated macros in Microsoft Excel (and other Microsoft desktop apps). Other software, too, has relied on scripting languages of greater sophistication. For example, Tcl is the primary scripting tool for electronic design automation (EDA) and CAD tools. In other spheres, such as UI design, the embeddable scripting language Lua is widely popular.

In fact, for many years, the concept of embedded scripting was sufficiently common that the original "Gang of Four" book on design patterns included the Interpreter pattern—which today seems like a positively odd inclusion. The popularity of this solution crested roughly 10 years ago in the form of domain-specific languages (DSLs), which had a prolonged moment in the sun until developers realized that the benefit

PHOTOGRAPH BY BOB ADLER/THE VERBATIM AGENCY

of domain specificity (rather than a general-purpose solution) imposed difficult requirements.

Today, the turn from DSLs toward general-purpose embeddable languages has made it much easier to embed scripting options in apps, as no proprietary language needs to be defined and implemented. Since the release of Java 6 (when DSLs were flourishing), the JVM has included a scripting capability (under JSR 223). The JDK-provided scripting engine was Rhino, an implementation of JavaScript. Not long thereafter, scripting engines for Groovy, JRuby, and Jython became available—the latter two being implementations of Ruby and Python, respectively.

Between JavaScript, Ruby, Python, and Groovy, nearly every important scripting language became available for Java developers to offer their users. But users are not the only beneficiaries of the scripting option—so are developers.

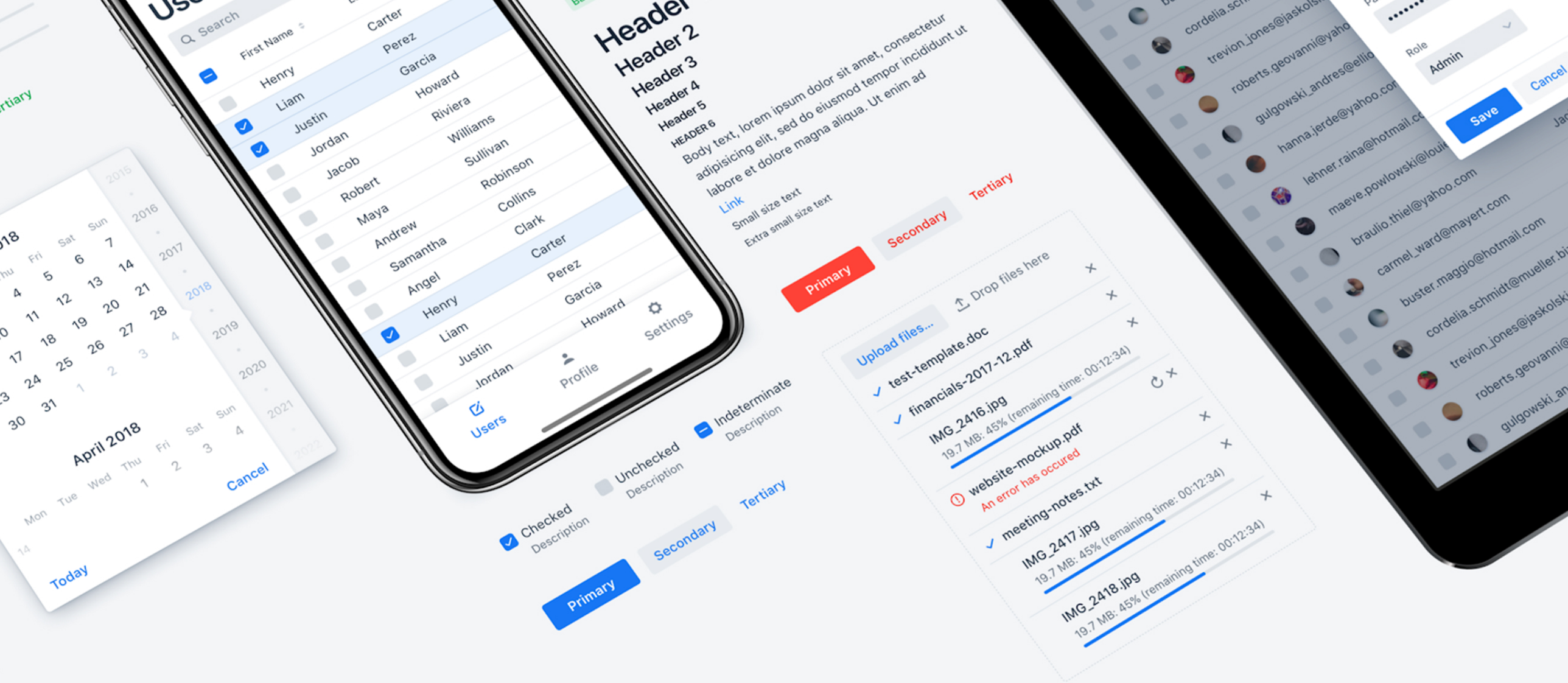
For example, I spend time writing typesetting software. (Everyone has peculiarities, no?) I'd like to offer my users the ability to specify column widths as a percent of the page width, rather than locking them into using absolute units. But I don't want to limit them to a single value; I'd like to let them represent the width as a computation. For example: half the remaining width after margins of *x* width and a single other column of *y* width. However, I have no desire whatsoever to write a parser and calculator for the whole range of possible expressions users might employ. So, I use an embedded scripting engine: I let users declare variables, assign them values, and write as complex an expression as they want. I pass the whole thing to the scripting engine, which executes the code and returns a value, which I then pass to the Java code. With very little work, I've helped my users and signaled to the experts that their needs are understood. I've also distinguished my software from other simpler solutions. And the scripting language (JavaScript) is not proprietary. Instead, it's fully documented in hundreds of books and tutorials. Not bad, eh?

In Java 8, Oracle shipped Nashorn, a faster implementation of JavaScript, which replaced Rhino. Today, however, as described on page 87, there is a proposal to discontinue active maintenance of Rhino, which is intended to sound out developers on their interest in Nashorn. The engine currently supports JavaScript 5.1 fully. The question is whether the cost of updating it to be fully compliant with recent updates to JavaScript is worthwhile, or whether it should remain fixed at its current release and eventually be removed from the JDK (but still available for developers as a downloadable engine).

I am convinced of Nashorn's value and impressed by how simple it is to use (the use case I've described here takes about two dozen lines of code). I will continue using Nashorn because the experts I'm targeting are likely to write only short snippets of JavaScript and version 5.1 of the language is entirely sufficient.

However, your needs might be more extensive, in which case I urge you to look at all the scripting engines. By including them in your applications—both server-based and user-facing—you'll save yourself a lot of time and a lot of work, and you'll gladden your customers.

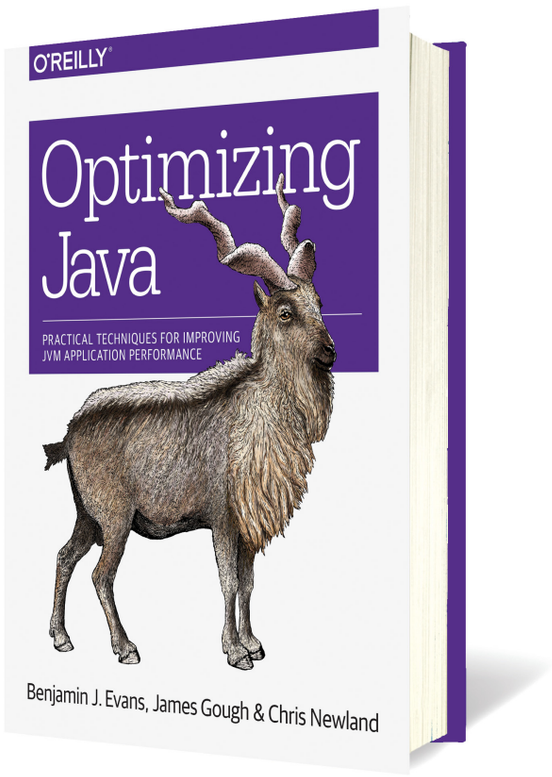
Andrew Binstock, Editor in Chief
javamag_us@oracle.com
@platypusguy



Next generation web framework and UI components for the JVM

https://vaadin.com

vaadin



OPTIMIZING JAVA

By Ben Evans, James Gough, and Chris Newland

Books on Java performance are few and far between. From top publishers, we have Scott Oaks' *Java Performance* (O'Reilly, 2014) as well as *Java Performance Companion* (Addison-Wesley, 2016), by multiple authors, which I reviewed in the September/October 2016 issue. The latter book was a descent into the workings of the G1 garbage collector and the Java Serviceability Agent. *Optimizing Java* is like the Oaks book in that it is an exhaustive explanation of the workings of the JVM, with a principal focus on the HotSpot implementation and OpenJDK-based variants.

The authors start with explanations of what performance is, how it's measured, and the terminology used to refer to different aspects of performance. A quick overview of the JVM is then followed by a detailed discussion of hardware. This is necessary, because at numerous points in the rest of the book the authors refer to concepts that are explained in this hardware section: the CPU, I/O, memory paging, and other aspects of today's systems. I admire the decision to anchor future discussions in the hardware, rather than generalizing in the abstract.

The book takes one additional step in preparing the reader for the forthcoming analysis: a detailed examination of how to run benchmarks (primarily using the Java Microbenchmark Harness) and how to understand the results obtained. The next three chapters are deep dives into the garbage collection (GC), including an excellent technical overview of GC in general, followed by analysis of how G1 and the parallel GC work in the present JVM. There follows a chapter on logging GC performance, and only then suggestions for how to tune the garbage collector for specific loads and execution situations. By the time you get to the tuning advice, you are fully capable of understanding the principles behind the suggestions.

The next four chapters explore how the JVM executes code. Authors Evans and Newland have been writing about code execution for this magazine for the last several issues. If you like their lucid explanations, you'll love these chapters. They're nitty-gritty nerd stim and highly readable. Then come two chapters on concurrency and parallel execution, one chapter on logging and efficient message processing, and one on profiling. A final section discusses the changes wrought by Java 9 and their effects on the previous material.

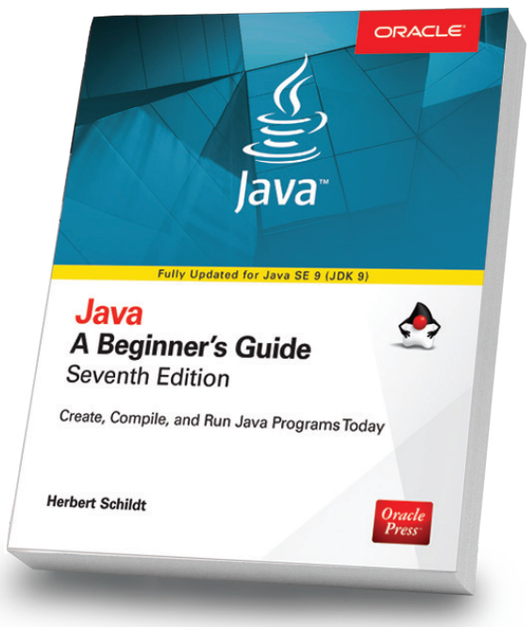
Optimizing Java is one of the most informative explanations of the internals of the JVM. Even if you're not looking to solve a performance problem, you'll benefit from reading it. And if you believe—as I do—that good programmers become better by knowing how exactly their code is executed, then you owe it to yourself to read this excellent volume.

—Andrew Binstock

Oracle Press™

Your Destination for Oracle and Java Expertise

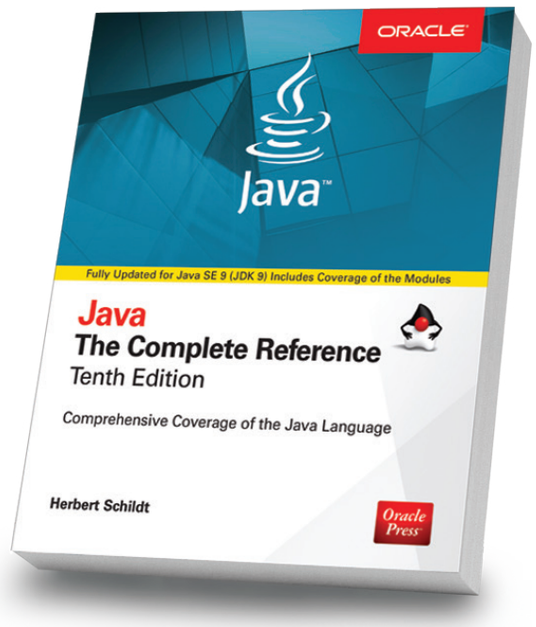
Written by leading experts in Java, Oracle Press books offer the most definitive, complete, and up-to-date coverage of Java available.



Java: A Beginner's Guide, 7th Edition

Herb Schildt

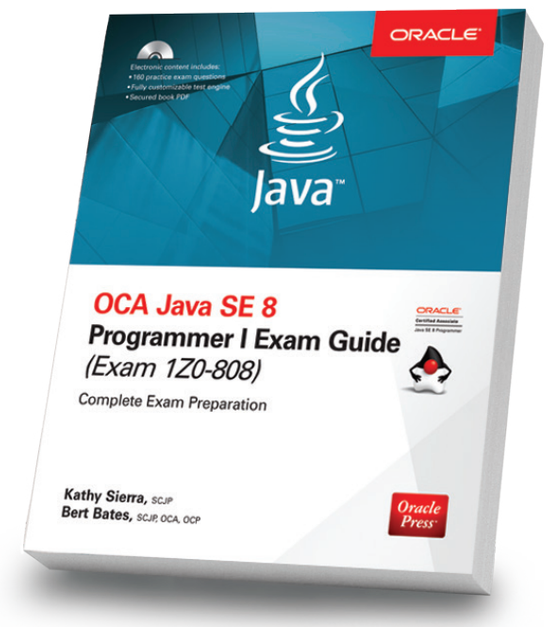
Revised to cover Java SE 9, this book gets you started programming in Java right away. Free online supplement covering key new features in JDK 10 available for download on the book's page on OraclePressBooks.com



Java: The Complete Reference, 10th Edition

Herb Schildt

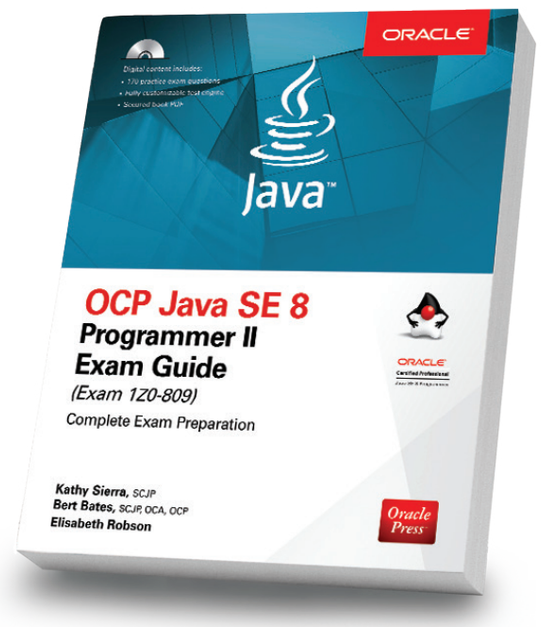
Updated for Java SE 9, this book shows how to develop, compile, debug, and run Java programs. Visit the book's page on OraclePressBooks.com to download free supplements on JDK's key new features.



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

Kathy Sierra, Bert Bates

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



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

Kathy Sierra, Bert Bates, Elisabeth Robson

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

Available in print and eBook formats.

Oracle Press™

www.OraclePressBooks.com • @OraclePress

//events /

Oracle Code One

OCTOBER 22–25
SAN FRANCISCO, CALIFORNIA
The annual JavaOne event has been reimagined as Oracle Code One, a new developer conference that includes more languages, technologies, and developer communities. Look for talks on Go, Rust, Python, JavaScript, and R, along with the great Java technical content that developers expect. Topics will include microservices, containers, AI, chatbots, blockchain, and databases. A Java keynote and community keynote will remain, and all of the Java-focused community activities are being carried forward including the kids event, IGNITE sessions, community day (now as a track), Java Champion briefings, and Duke's Choice Awards.

PHOTOGRAPH BY SFWHALEWATCHING-24/FLICKR

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.

JVM Language Summit

JULY 30–31, CONFERENCE
AUGUST 1–2, WORKSHOP
SANTA CLARA, CALIFORNIA
The JVM Language Summit is an open technical collaboration among language designers, compiler writers, tool builders, runtime engineers, and VM architects. Presenters and attendees will share their experiences as creators of both the JVM and programming languages for the JVM. Organizers also welcome non-JVM developers of similar technologies to attend or speak about their runtime, VM, or language of choice. The conference will be followed immediately by a two-day OpenJDK Committers' Workshop with a focus on the JDK technical roadmap and discussion of both technical and community issues.

NFJS Central Iowa Software Symposium

AUGUST 3–4
DES MOINES, IOWA
This conference will focus on the latest technologies and best practices emerging in the modern software development and architecture space. Scheduled topics include modern Java frameworks for building microservices and migrating to Java 9 with the Jigsaw module system. Team attendance is encouraged.

O'Reilly Artificial Intelligence Conference

SEPTEMBER 4–5, TRAINING
SEPTEMBER 5–7, CONFERENCE AND TUTORIALS
SAN FRANCISCO, CALIFORNIA
This conference centers on learning how to implement AI in real-world projects. Topics include image classification models in TensorFlow, deep learning with time-series data, and trustless machine learning contracts on Ethereum.

//events /

JavaZone

SEPTEMBER 11, WORKSHOPS
SEPTEMBER 12–13, CONFERENCE
OSLO, NORWAY

JavaZone is a conference for Java developers organized by javaBin, the Norwegian Java User Group. This year the 17-year-old conference boasts approximately 200 speakers and seven parallel tracks over two days in addition to a day of hands-on workshops.

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. (Website in German.)

jDays

SEPTEMBER 25
GOTHENBURG, SWEDEN

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

Strange Loop

SEPTEMBER 26–28
ST. LOUIS, MISSOURI

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

NFJS New England Software Symposium

SEPTEMBER 28–30
FRAMINGHAM, MASSACHUSETTS

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

KotlinConf

OCTOBER 3, WORKSHOPS
OCTOBER 4–5, CONFERENCE
AMSTERDAM, THE NETHERLANDS

This is the principal conference for the up-and-coming JVM language, Kotlin. Keynotes by Kotlin Project Lead Andrey Breslav and Purple Evolution CEO Alicia Carr are slated.

JAX London

OCTOBER 8 AND 11, WORKSHOPS
OCTOBER 9–10, CONFERENCE
LONDON, ENGLAND

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

Desert Code Camp

OCTOBER 13
CHANDLER, ARIZONA

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

Java Enterprise Summit

OCTOBER 17–19
DÜSSELDORF, GERMANY

Java Enterprise Summit is a Java EE training event exploring new paradigms such as microservices, API design, and state-of-the-

PHOTOGRAPH BY JENS-PETTER SALVESEN/FLICKR

art enterprise Java applications. (Website in German.)

EclipseCon Europe

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

O'Reilly Software Architecture Conference

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

VOXXED DAYS MICROSERVICES

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

QCon San Francisco

NOVEMBER 5–7, CONFERENCE
NOVEMBER 8–9, WORKSHOPS
SAN FRANCISCO, CALIFORNIA
Although this year's content had not yet been announced at press time, recent QCon conferences have offered several Java tracks along with tracks related to web development, DevOps, cloud computing, and more.

W-JAX

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

DeveloperWeek Austin

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

J-Fall 2018

NOVEMBER 7–8
J-Fall is organized by and for the Dutch Java community. With 1,500 Java professionals attending, J-Fall is the biggest Java conference of the Netherlands, boasting more than 40 sessions and four hands-on labs, more than 60 top speakers from all around the world, and a preconference day with in-depth workshops and the Masters of Java contest. Speakers include Apache Maven Project Chair Robert Scholte, Google Developer Advocate Ray Tsang, and AWS Senior Solutions Architect Brian Hammons.

Devoxx Belgium 2018

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

Codemotion Berlin

NOVEMBER 20–21
BERLIN, GERMANY
Organizers were still accepting speaker submissions at press time, but last year's event featured talks on reinforcement learning, microservices testing, and building domain-specific languages. The event is open to all languages and technologies and features coding lectures and workshops.

Topconf Tallinn

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

Are you hosting an upcoming Java conference? Send us a link and a description of your event at least 90 days in advance at javamag_us@oracle.com.

DEVELOPER COMMUNITY EVENTS FROM THE DEVOXX FAMILY

COMING SOON

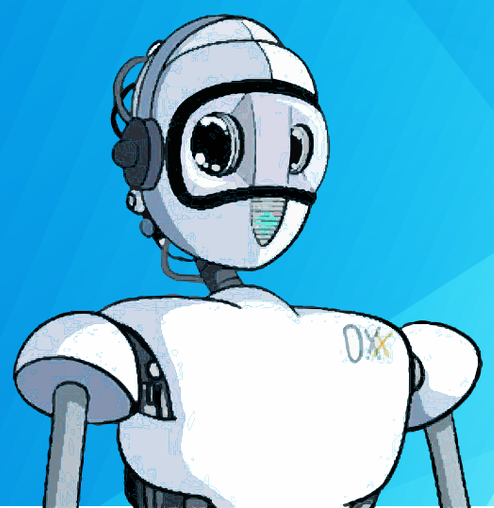
DEVOXX™

DEVOXX.COM

BELGIUM 12-16 NOVEMBER

UKRAINE 23 - 24 NOVEMBER

MOROCCO 27 - 29 NOVEMBER



TICINO 20 OCT

BRISTOL 25 OCT

BANFF 26 - 27 OCT

MICROSERVICES, PARIS 29 – 31 OCT

THESSALONIKI 19 - 20 NOV

CLUJ-NAPOCA 22 NOV

VOXXEDDAYS

VOXXEDDAYS.COM

Finding and Using the Good Libraries

When developers speak with admiration of the Java ecosystem, they're referring especially to two things: the abundance of excellent development tools and the vast number of third-party libraries. Libraries are available today to do almost anything that is required, and for the most part, they're open source and freely available. A quick look at Maven Central—one of the principal repositories for Java artifacts—lists more than 3 million entries, of which nearly 300,000 are unique. That's a lot of choices!

To help you navigate such a wide body of work, we regularly cover libraries in *Java Magazine*, and once a year we dedicate an entire issue to them. In this issue, we include an annotated list (page 65) of the libraries we've covered over the years—it contains everything from cryptocurrency to JVM internals. We also explain the mechanics of library operations. This issue, for example, includes a hands-on discussion of how to convert pre-Java 9 libraries (page 53) to Java modules. In earlier issues, we examined how the JVM finds and loads libraries (PDF), and we explained in depth how best to write libraries.

ART BY WES ROWELL

On the following pages we look at the most popular library for creating PDF files (page 15), explain how to create HTML on the fly (page 27) without using templates, and examine ways to transform Java bytecodes (page 39) in useful ways. We've also included another deep dive into a design pattern—this time the State pattern (page 66)—and, of course, we've bundled our quiz (page 76) and book review (page 8).

BRUNO LOWAGIE

JORIS SCHELLEKENS

Build Complex PDFs Easily

How to use the popular Java library iText to create and manipulate PDF files

iText is an open source library that is used to create and process PDF documents in web and other applications. It's licensed under the AGPL and is available in both Java and C# versions. The latest major release is iText 7.1, which is the first version that supports the new PDF 2.0 standard.

iText supports many different flavors of PDF, including PDF/A, which is the standard for archiving, and PDF/UA, which is the standard defining how to make PDF documents accessible for the blind and visually impaired. With it, you can create invoices in the PDF format, fill out PDF forms, assemble PDF documents into portfolios, remove private data from PDF documents, and perform many other activities associated with document creation and management.

In this article, we introduce some of iText's functionality:

- We create a PDF document from scratch, using simple objects such as `Paragraph`, `List`, and `Image`.
- We then process a data set stored in a comma-separated value (CSV) file and render that data to a PDF file.
- Finally, we take a data set presented in a browser using HTML and CSS, and we convert that web page into a PDF document.

Creating a PDF Document

We created the PDF document shown in **Figure 1** from Java code, using iText. You can see that it has a paragraph saying "Hello World," which is followed by a block of text in a different font consisting of a paragraph and a list. On the right side of the text, you also see an image of a hooded developer.

The code in **Listing 1** shows how we created this document.

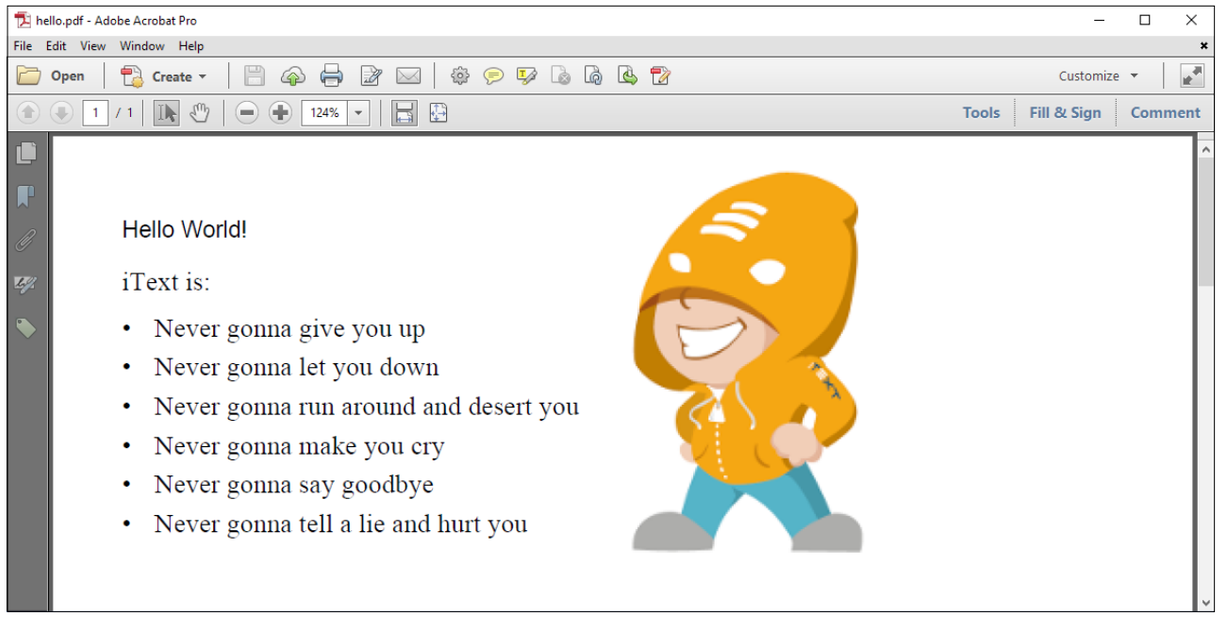


Figure 1: "Hello World" PDF document created with iText

Listing 1: Hello.java code

```
public void createPdf(String dest) throws IOException {
    PdfDocument pdf = new PdfDocument(new PdfWriter(dest));
    Document document = new Document(pdf);
    document.add(new Paragraph("Hello World!"));
    PdfFont font = PdfFontFactory
        .createFont(StandardFonts.TIMES_ROMAN);
    Div div = new Div().setFont(font).setFontSize(14);
    div.add(new Paragraph("iText is:"));
    List list = new List()
        .setSymbolIndent(12)
        .setListSymbol("\u2022");
    list.add(new ListItem("Never gonna give you up"))
        .add(new ListItem("Never gonna let you down"))
        .add(new ListItem("Never gonna run around and desert you"))
        .add(new ListItem("Never gonna make you cry"))
```

```
            .add(new ListItem("Never gonna say goodbye"))
            .add(new ListItem("Never gonna tell a lie and hurt you"));
        div.add(list);
        document.add(div);
        Image img = new Image(ImageDataFactory.create(IMG))
            .setFixedPosition(300, 625);
        document.add(img);
        document.close();
    }
```

Let's examine this code step by step.

Low-level document and writer objects. First, we created a `PdfDocument` instance, using a `PdfWriter` as a parameter. These are low-level objects. The `PdfWriter` is responsible for writing the PDF bits and bytes to a destination. In this case, the destination is a String defining the path to a file. Alternatively, the `PdfWriter` accepts a file or an `OutputStream`. For instance, if you want to create a document that exists only in memory, you can create a `PdfWriter` with a `ByteArrayOutputStream`. This is typically done when the file is created on a server but served to a client, as is the case with a web application.

You can use the many features available through the iText API to program your document so that it looks exactly the way you want it to look, but you can also use shortcuts.

You could now use the `PdfDocument` instance to create a `PdfPage` instance, and then draw content on that page using the low-level operators and operands that are described in the PDF ISO 32000 standard. However, that would be a very tedious job. Fortunately, you can keep all this heavy lifting under the hood by using iText's high-level options.

High-level objects. We used the `PdfDocument` instance to create a `Document` object. We can then create objects such as `Paragraph`, `List`, and `Image`, and add these objects to the `Document`.

1. We added a new `Paragraph` containing the String `Hello World!`
2. We created a `Div` object, for which we set a different font that was obtained from a `PdfFontFactory`. We also set the font size to 14. The `Div` object held a `Paragraph` and a `List`. Both inherited the font and font size from their parent. We created a `List` with an indentation of 12 points and a bullet as the list symbol. We added several `ListItems` to this list. We added the `Paragraph` and the `List` to the `Div`, and then added the `Div` to the document.
3. We then created an `Image` object using image data obtained from an `ImageDataFactory`. In this case, `IMG` is the path to an image. We also defined a pair of *x* and *y* coordinates for the image. When we added the image to the document, the lower-left corner of the image corresponded with this coordinate.

`Paragraph`, `Div`, `List`, `ListItem`, and `Image` are just a few of the building blocks that are available in iText; other building blocks include `Text`, `Link`, `Tab`, `AreaBreak`, `LineSeparator`, `Table`, and `Cell`.

Closing the document. Once we finished adding content, we closed the document. Closing the document automatically closes the `PdfDocument`, the `PdfWriter`, and the `OutputStream` used by `PdfWriter`.

In this example, all the content was hardcoded in our source code. In real-world applications, you can obtain the data from an external source, such as a database. In the next example, we used the `Table` and `Cell` classes to publish data stored in a CSV file in tabular form.

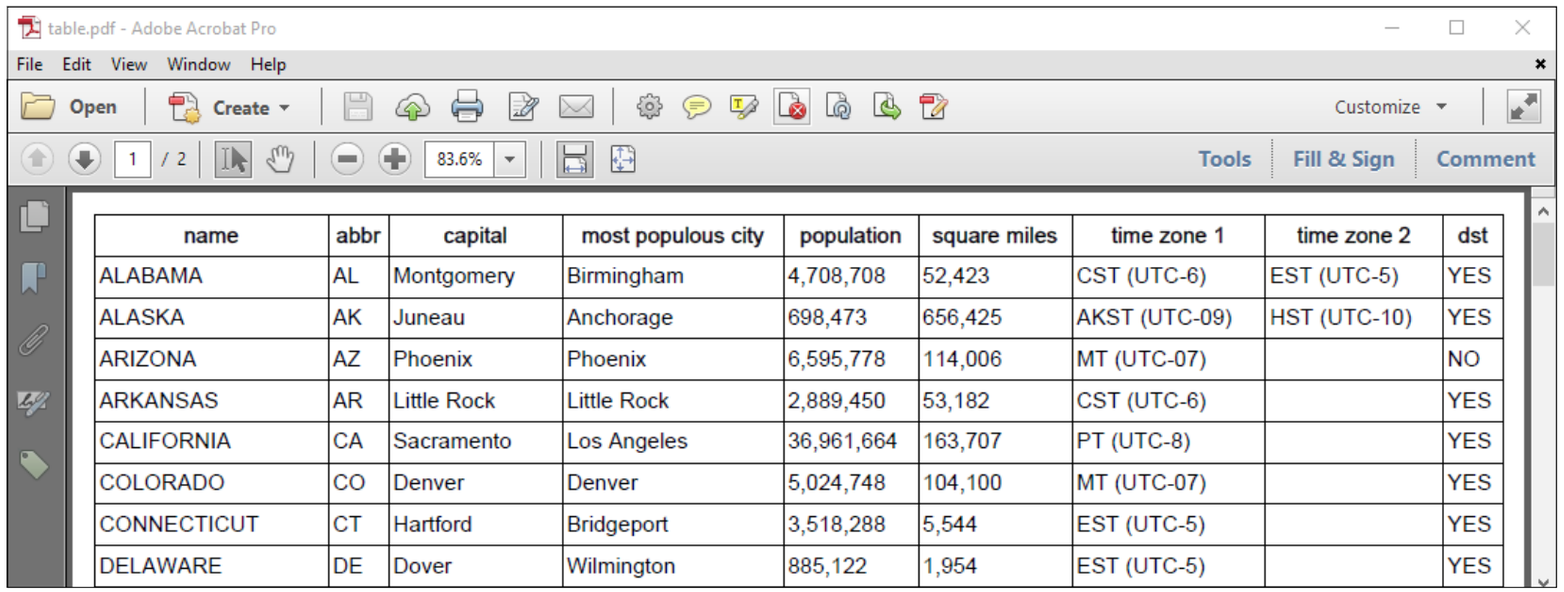
Publishing Data to a PDF File

Suppose that you have a CSV file listing all the states of the US, along with each state's abbreviation, capital, most populous city, population, square miles, and times zone(s) as well as whether it uses daylight saving time (DST).

An example of an entry in such a file might be:

```
CALIFORNIA;CA;Sacramento;Los Angeles;36,961,664;163,707;PT (UTC-8); ;YES
```

Let's see how to take this CSV file and convert it into the PDF file shown in **Figure 2**.



name	abbr	capital	most populous city	population	square miles	time zone 1	time zone 2	dst
ALABAMA	AL	Montgomery	Birmingham	4,708,708	52,423	CST (UTC-6)	EST (UTC-5)	YES
ALASKA	AK	Juneau	Anchorage	698,473	656,425	AKST (UTC-09)	HST (UTC-10)	YES
ARIZONA	AZ	Phoenix	Phoenix	6,595,778	114,006	MT (UTC-07)		NO
ARKANSAS	AR	Little Rock	Little Rock	2,889,450	53,182	CST (UTC-6)		YES
CALIFORNIA	CA	Sacramento	Los Angeles	36,961,664	163,707	PT (UTC-8)		YES
COLORADO	CO	Denver	Denver	5,024,748	104,100	MT (UTC-07)		YES
CONNECTICUT	CT	Hartford	Bridgeport	3,518,288	5,544	EST (UTC-5)		YES
DELAWARE	DE	Dover	Wilmington	885,122	1,954	EST (UTC-5)		YES

Figure 2: Data set rendered to a PDF table

In the code of **Listing 2**, we created a `PdfDocument` instance just like we did in **Listing 1**. Then, we did the following:

- The page size of the page we created in our first example was `A4`, which is the standard size for documents outside of the US. In this example, we wanted to create a document with pages in the `Letter` size in landscape orientation. We passed this page size information as the second parameter when we created the high-level `Document` object. We also reduced the default margins to 12 points.
- We organized all the content into a `Table` with nine columns. We defined the width of each column using a relative width. For instance, the first column is four times as wide as the second column. The total width of the table is defined as 100% of the available width on the page, taking into account a margin of 12 points to the right and 12 points to the left. The table has a header for which we wanted to use a different style. Therefore, we created a `Style` object that caused text to be written in bold and to be aligned in the center of its container.
- We read the CSV file line by line using a `BufferedReader`. We processed the first line (the header) using the header style. We looped over the rest of the lines and processed them without defining a style.

All of this was done in the `createPdf()` method.

Listing 2: TableExample.java (part 1)

```
public void createPdf(String dest) throws IOException {
    PdfDocument pdf = new PdfDocument(new PdfWriter(dest));
    Document document = new Document(pdf, PageSize.LETTER.rotate());
    document.setMargins(12, 12, 12, 12);
    Table table = new Table(new float[]{4, 1, 3, 4, 3, 3, 3, 3, 1});
    table.setWidth(UnitValue.createPercentValue(100));
    Style header = new Style()
        .setBold().setTextAlignment(TextAlignment.CENTER);
    BufferedReader br = new BufferedReader(new FileReader(DATA));
    String line = br.readLine();
    process(table, line, header);
    while ((line = br.readLine()) != null) {
        process(table, line, null);
    }
    br.close();
    document.add(table);
    document.close();
 }
```

In the `createPdf()` method, we processed each line using the `process()` method. This method is shown in **Listing 3**.

Listing 3: TableExample.java (part 2)

```
public void process(Table table, String line, Style style) {
    StringTokenizer tokenizer = new StringTokenizer(line, ";");
    while (tokenizer.hasMoreTokens()) {
        Cell cell = new Cell()
```

```
                .add(new Paragraph(tokenizer.nextToken()));
            if (style == null) {
                table.addCell(cell);
            }
            else {
                cell.addStyle(style);
                table.addHeaderCell(cell);
            }
        }
    }
```

We used a `StringTokenizer` to split a CSV record into fields, and we created a `Cell` instance for each field. Then we did the following:

- If no style was defined, we just added the cell to the table, using the `addCell()` method.
- If a style was defined, we added the style to the cell, and we added the cell to the table as a header cell using the `addHeaderCell()` method.

By making the distinction between header cells and data cells, we told iText what to do if the table didn't fit the page.

As you can see in **Figure 3**, header cells are repeated on every page. There's also an `addFooterCell()` method if you want repeating footer rows, but we didn't need any footers in this simple example.

At this point, you could adapt the code to do the following (among other options):

- Introduce different styles
- Introduce background colors for the cells
- Add page numbers

You can use the many features available through the iText API to program your document so that it looks exactly the way you want it to look, but you can also use shortcuts, as we illustrate next.

table.pdf - Adobe Acrobat Pro

File Edit View Window Help

name	abbr	capital	most populous city	population	square miles	time zone 1	time zone 2	dst
ALABAMA	AL	Montgomery	Birmingham	4,708,708	52,423	CST (UTC-6)	EST (UTC-5)	YES
ALASKA	AK	Juneau	Anchorage	698,473	656,425	AKST (UTC-09)	HST (UTC-10)	YES
ARIZONA	AZ	Phoenix	Phoenix	6,595,778	114,006	MT (UTC-07)		NO
ARKANSAS	AR	Little Rock	Little Rock	2,889,450	53,182	CST (UTC-6)		YES
CALIFORNIA	CA	Sacramento	Los Angeles	36,961,664	163,707	PT (UTC-8)		YES
COLORADO	CO	Denver	Denver	5,024,748	104,100	MT (UTC-07)		YES
CONNECTICUT	CT	Hartford	Bridgeport	3,518,288	5,544	EST (UTC-5)		YES
DELAWARE	DE	Dover	Wilmington	885,122	1,954	EST (UTC-5)		YES
FLORIDA	FL	Tallahassee	Jacksonville	18,537,969	65,758	EST (UTC-5)	CST (UTC-6)	YES
GEORGIA	GA	Atlanta	Atlanta	9,829,211	59,441	EST (UTC-5)		YES
HAWAII	HI	Honolulu	Honolulu	1,295,178	10,932	HST (UTC-10)		NO
IDAHO	ID	Boise	Boise	1,545,801	83,574	MT (UTC-07)	PT (UTC-8)	YES
ILLINOIS	IL	Springfield	Chicago	12,910,409	57,918	CST (UTC-6)		YES
INDIANA	IN	Indianapolis	Indianapolis	6,423,113	36,420	EST (UTC-5)	CST (UTC-6)	YES
IOWA	IA	Des Moines	Des Moines	3,007,856	56,276	CST (UTC-6)		YES
KANSAS	KS	Topeka	Wichita	2,818,747	82,282	CST (UTC-6)	MT (UTC-07)	YES
KENTUCKY	KY	Frankfort	Louisville	4,314,113	40,411	EST (UTC-5)	CST (UTC-6)	YES
LOUISIANA	LA	Baton Rouge	New Orleans	4,492,076	51,843	CST (UTC-6)		YES
MAINE	ME	Augusta	Portland	1,318,301	35,387	EST (UTC-5)		YES
MARYLAND	MD	Annapolis	Baltimore	5,699,478	12,407	EST (UTC-5)		YES
MASSACHUSETTS	MA	Boston	Boston	6,593,587	10,555	EST (UTC-5)		YES
MICHIGAN	MI	Lansing	Detroit	9,969,727	96,810	EST (UTC-5)	CST (UTC-6)	YES
MINNESOTA	MN	Saint Paul	Minneapolis	5,266,214	86,943	CST (UTC-6)		YES
MISSISSIPPI	MS	Jackson	Jackson	2,951,996	48,434	CST (UTC-6)		YES
MISSOURI	MO	Jefferson City	Kansas City	5,987,580	69,709	CST (UTC-6)		YES

name	abbr	capital	most populous city	population	square miles	time zone 1	time zone 2	dst
MONTANA	MT	Helena	Billings	974,989	147,046	MT (UTC-07)		YES
NEBRASKA	NE	Lincoln	Omaha	1,796,619	77,358	CST (UTC-6)	MT (UTC-07)	YES
NEVADA	NV	Carson City	Las Vegas	2,643,085	110,567	PT (UTC-8)	MT (UTC-07)	YES
NEW HAMPSHIRE	NH	Concord	Manchester	1,324,575	9,351	EST (UTC-5)		YES
NEW JERSEY	NJ	Trenton	Newark	8,707,739	8,722	EST (UTC-5)		YES
NEW MEXICO	NM	Santa Fe	Albuquerque	2,009,671	121,593	MT (UTC-07)		YES

Figure 3: Rows distributed over different pages with repeating header

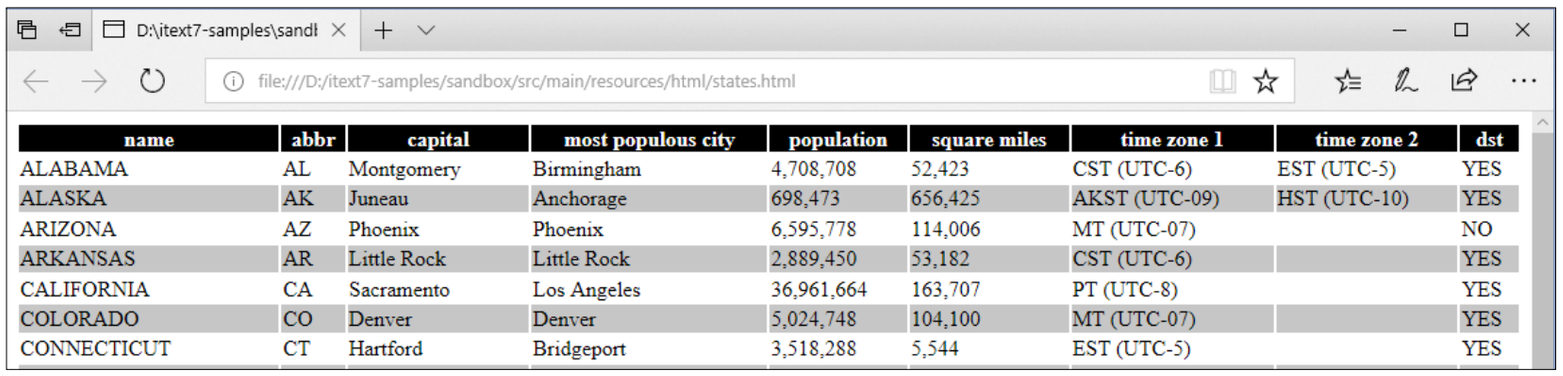
Converting HTML Data to PDF

Suppose that you already have in the form of an HTML file all the data you want to render as a PDF file. **Figure 4** shows a web page in a browser listing all the states of the US, with white text on a black background for the header row and an alternating background for the data rows.

If you look inside the HTML file that we used in **Figure 4** (shown in **Listing 4**), you can see that text and background colors are defined using CSS. You can also see that the page size, orientation, and a footer are defined using an `@page` rule. When printed, each page would have a footer labeled "Page X of Y," where "X" is the current page number and "Y" is the total number of pages.

Listing 4: HTML and CSS to render data in a browser

```
<html>
<head>
  <meta http-equiv="Content-Type" content="text/html;
        charset=windows-1252">
 <style>
       th { background: black; color: white; }
       tr:nth-child(even) { background: #C0C0C0; }
       @page {
          size: A4 landscape;
          @bottom-right {
```



name	abbr	capital	most populous city	population	square miles	time zone 1	time zone 2	dst
ALABAMA	AL	Montgomery	Birmingham	4,708,708	52,423	CST (UTC-6)	EST (UTC-5)	YES
ALASKA	AK	Juneau	Anchorage	698,473	656,425	AKST (UTC-09)	HST (UTC-10)	YES
ARIZONA	AZ	Phoenix	Phoenix	6,595,778	114,006	MT (UTC-07)		NO
ARKANSAS	AR	Little Rock	Little Rock	2,889,450	53,182	CST (UTC-6)		YES
CALIFORNIA	CA	Sacramento	Los Angeles	36,961,664	163,707	PT (UTC-8)		YES
COLORADO	CO	Denver	Denver	5,024,748	104,100	MT (UTC-07)		YES
CONNECTICUT	CT	Hartford	Bridgeport	3,518,288	5,544	EST (UTC-5)		YES

Figure 4: US states shown in a browser

```
                content: "Page " counter(page) " of " counter(pages);
            }
        }
  </style>
</head>
<body>
<table width="100%">
  <thead>
    <tr>
    <th>name</th>
    <th>abbr</th>
    <th>capital</th>
    <th>most populous city</th>
    <th>population</th>
    <th>square miles</th>
    <th>time zone 1</th>
    <th>time zone 2</th>
    <th>dst</th>
    </tr>
</thead>
<tbody>
  <tr>
  <td>ALABAMA</td>
  <td>AL</td>
  <td>Montgomery</td>
  <td>Birmingham</td>
  <td>4,708,708</td>
  <td>52,423</td>
  <td>CST (UTC-6)</td>
  <td>EST (UTC-5)</td>
  <td>YES</td>
  </tr>
```

```
    <tr>
...
```

If you take a closer look at the HTML code, you'll notice that the header is in a `<thead>` section to make sure it will be repeated on every page. The 50 data rows are inside the `<tbody>` section.

The code to convert this HTML file to PDF is very simple: it consists of the single line shown in **Listing 5**.

Listing 5

```
public void createPdf(String html, String dest) throws IOException {
    HtmlConverter.convertToPdf(new File(html), new File(dest));
}
```

Figure 5 shows the result after executing the `createPdf()` method. It looks exactly like the HTML that was rendered and then shown in **Figure 4**, but now there are also page numbers.

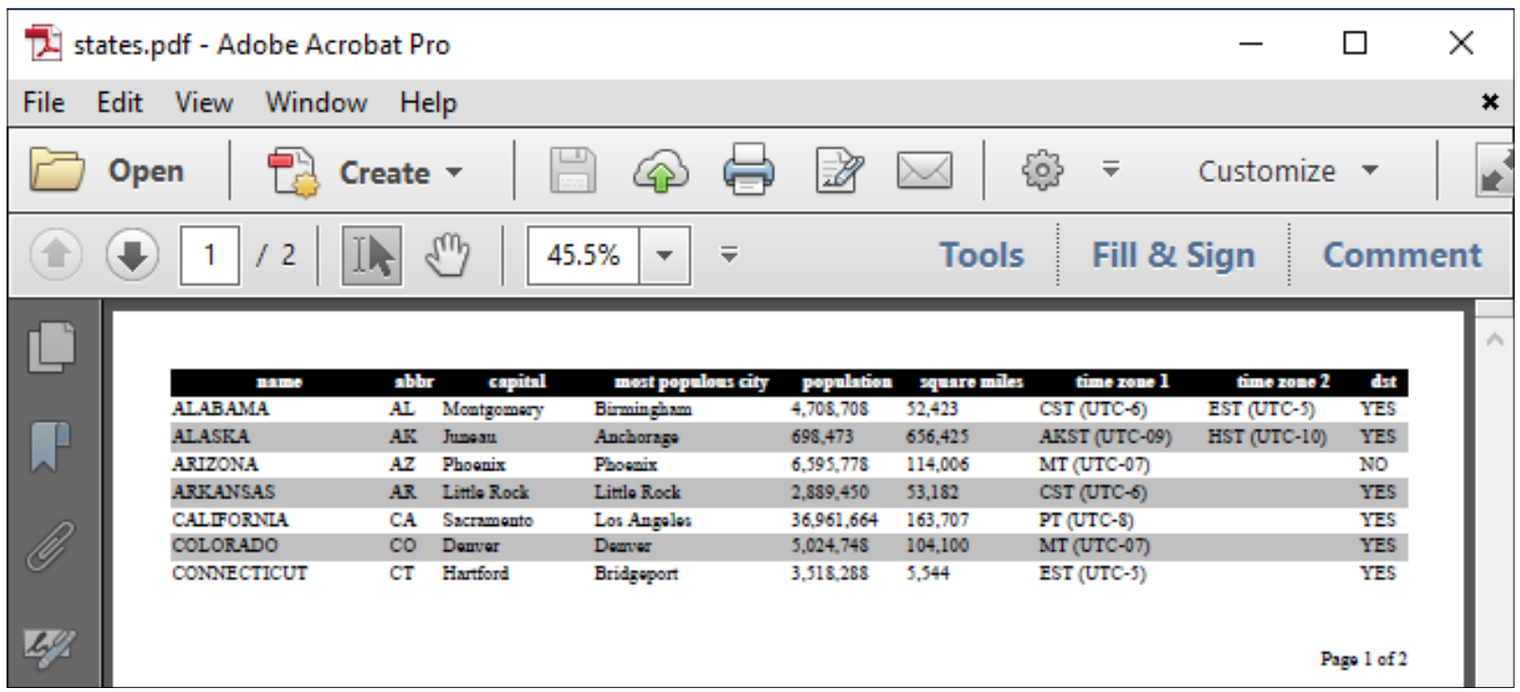


Figure 5: HTML rendered to PDF with iText and pdfHTML

The code becomes slightly more complex if you want to introduce special fonts, add bookmarks, and perform other customizations—all of which are illustrated in the tutorial on the iText website.

Conclusion

In this article, we created two PDF documents from scratch using the iText core libraries. With these libraries, you can also fill out interactive forms, digitally sign documents, split and merge existing PDF files, and reuse and extract content from existing documents. On top of the iText core libraries, you can use different add-ons, including pdfHTML—the add-on we used in our example when we converted an HTML file to PDF—and pdfSweep, which can be used to physically remove content from documents. (For instance, using a regular expression that matches a social security number [SSN], you can physically remove the SSN from a batch of PDF documents.) iText has also been active combining PDF with blockchain technology. With the pdfChain add-on, you can register documents in a blockchain instead of digitally signing them. By adding metadata such as a status and the document location, you can automate your document workflow and ensure the long-term validity of your PDF files. </article>

Bruno Lowagie (@bruno1970) is the original developer of iText. He is an active member of the ISO and PDF communities and has authored several books about iText. When he is not working in the PDF world, Lowagie spends much of his time with his wife and two sons.

Joris Schellekens (@Joris1989BE) is a research engineer with iText who focuses on disruptive technologies such as machine learning and natural language processing. He is passionate about new technology and finding ways to solve challenges with PDF. When Schellekens is not researching or coding for iText, you can find him working on his own coding projects, working on math projects, or listening to music.

MERT ÇALIŞKAN

j2html: An HTML5 Generator Library

Easily write small, dynamically generated web applications and get the benefits of Java's type safety and tools ecosystem.

j2html is a small but powerful library that enables you to generate type-safe HTML code with its fluent API. The idea behind it is to allow developers to write declarative Java code with a one-to-one mapping to HTML code.

Note that j2html is not a templating engine, but it provides an alternative way to dynamically create and reuse UI code in a type-safe way. It was created and is actively maintained by David Åse. In this article, I show you the bits and pieces you need to get started with the library and then move on to advanced use cases. The source for the examples is available for download.

Getting Started

You can obtain the latest version of j2html from Maven's Central Repository with the following dependency definition. The latest available version at the time of this writing is 1.3.0.

```
<dependency>
    <groupId>com.j2html</groupId>
    <artifactId>j2html</artifactId>
    <version>1.3.0</version>
</dependency>
```

If you are using Gradle, the dependency can be added as follows:

```
compile 'com.j2html:j2html:1.3.0'
```

JDK 8 is a minimum requirement as of version 1.2.2 of j2html.

Now, let's generate your first HTML code. Creating a body with a heading inside is simple, as shown in **Listing 1**.

Listing 1.

```
html(
    body(
        h1("Hello, Java Magazine Readers!")
    )
);
```

That will render the HTML output shown in **Listing 2**.

Listing 2.

```
<html>
   <body>
      <h1>Hello, Java Magazine Readers!</h1>
   </body>
</html>
```

The `body()` and `h1()` methods are statically imported. They come from one of the most important classes of the library, the `TagCreator` class. It can be statically imported with the syntax shown in **Listing 3**.

Listing 3.

```
import static j2html.TagCreator.*;
```

Let's continue with the more complex example shown in **Listing 4**, where I create a table with a list of employees provided as rows. The page will have a stylesheet file referenced in the head section and a `div` that acts as a header on top of the employee table.

Listing 4.

```
List<ContainerTag> rows = new ArrayList<>();
rows.add(tr().with(
       td().withText(("Mert")),
       td().withText(("Caliskan"))
));

String output =
   html(
      head(
          title("Java Magazine Examples"),
          link().withRel("stylesheet").withHref("my.css")
      ),
      body(
          div().withId("header").with(h1("Employees")),
          table().withClass("tableClass").with(
              thead(
                  tr(
                      th("Name"), th("Last Name")
                  )
              )
          )
          .with(tbody().with(rows)),
             footer().withClass("footerClass")
    )
).render();
```

In this code, I am defining the list of rows as a list of `ContainerTag`s (more about the class hierarchy after the next listing), which will map to the content that will be rendered inside the table. I move on with an HTML container tag definition created with the `html()` method, and then I set the header of the page with the `head()` method. I created a stylesheet file inside the head with

the `link()` method to demonstrate including a resource. The output is shown in **Listing 5**.

Listing 5.

```
<html>
  <head>
     <title>Java Magazine</title>
     <link rel="stylesheet" href="my.css">
   </head>
   <body>
     <div id="header">
       <h1>Employees</h1>
     </div>
     <table class="tableClass">
        <thead>
           <tr>
              <th>Name</th>
              <th>Last Name</th>
           </tr>
        </thead>
        <tbody>
           <tr>
              <td>Mert</td>
              <td>Caliskan</td>
           </tr>
        </tbody>
     </table>
     <footer class="footerClass"></footer>
   </body>
</html>
```

The important part of the class hierarchy is shown in **Figure 1**. All the tags referenced with

methods, such as `head()`, `body()`, and `div()`, are represented with an instance of `ContainerTag` within the library. `ContainerTag` stores the name of the tag along with its children referenced as `List<DomContent>`. `DomContent` is the abstract class that refers either to text or tags used inside the HTML document. `ContainerTag` implements the interface `Renderable` as well, which defines the content that can be represented as a string, and it provides a default implementation for the `render()` methods. The `Attribute` class defines the attributes of HTML elements, and they can be implemented by the `with*()` method syntax—for example, `withRel()`, `withHref()`, and `withClass()`, as I did in **Listing 4**.

Deep Dive

j2html provides various approaches for HTML generation, and I will try to address some of them in this section. Appending texts and tags to each other is easy with the `join()` method, with which you can concatenate input elements, in a given order, separated by a space. The method also removes spaces before periods and commas. An example is shown in **Listing 6**.

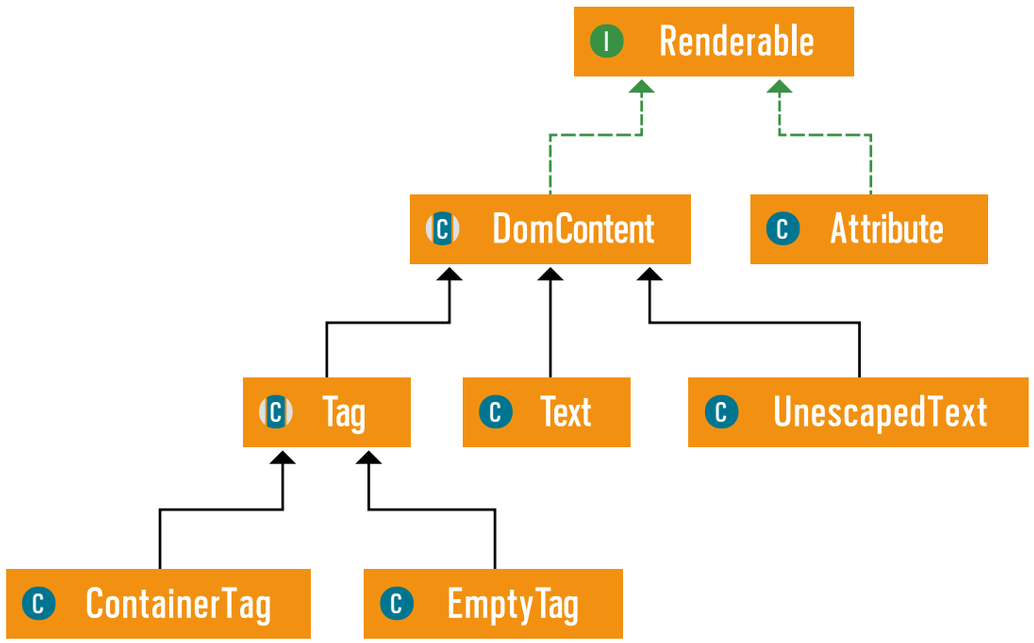


Figure 1: Important part of the class hierarchy

Listing 6.

```
body(
    p(join(
            i("Java Magazine"),
            "is a bimonthly deep dive into",
            b("Java"),
            "and the",
            b("JVM.")
      )
    )
);
```

The snippet returns the HTML output shown in **Listing 7**.

Listing 7.

```
<body>
    <p>
        <i>Java Magazine</i> is a bimonthly deep dive into
             <b>Java</b> and the <b>JVM.</b>
    </p>
</body>
```

`id` and `class` attributes can be obtained from the string provided by parsing it using CSS selectors. In **Listing 8**, the `attrs()` method that defines the `id` attribute of the body is set to `main`, and the `class` attribute is set to `content`.

Listing 8.

```
. . .
body(attrs("#main.content"),
    h1("Heading!")
);
```

In this code, the `attrs()` method creates an instance of the `Attr.ShortForm` class, which is a wrapper class that stores two string properties inside: `id` and `classes`. The `Attr` class contains the type-safe enumerations of the HTML attributes, so they can be used as `Attr.ID`, `Attr.HEIGHT`, `Attr.WIDTH`, and so on.

Conditional generation is also possible with the `iff()` and `iffElse()` methods shown in **Listing 9**.

Listing 9.

```
. . .
div().withClasses("menu",
    iff(isActive, "active")
);

div().withClasses("item",
    iffElse(isSelected, "selected", "not-selected")
);
```

By setting the value of `isSelected` to `true`, the output will be rendered as shown in **Listing 10**.

Listing 10.

```
<div class="menu active"></div>

<div class="item selected"></div>
```

You can create partial code snippets to reuse them for generating HTML snippets that you use often. **Listing 11** creates a login form by reusing the code to generate the HTML content for the username and password input fields and the Submit button of a form. The `genericInput()` method defines a way for creating an input field with the given type, name, and placeholder values.

Listing 11.

```
form().withMethod("post").with(
      genericInput("text","uname", "Enter Username"),
      genericInput("password", "psw", "Enter Password"),
      submitButton()
);

private static Tag genericInput(String type,
                                String name,
                                String placeholder) {
     return input()
             .withType(type)
             .withId(name)
             .withName(name)
             .withPlaceholder(placeholder)
             .isRequired();
};

private static Tag submitButton() {
    return button("Login").withType("submit");
};
```

The HTML output of the login form is shown in **Listing 12**.

Listing 12.

```
<form method="post">
    <input type="text" id="uname" name="uname"
           placeholder="Enter Username" required>
    <input type="password" id="psw" name="psw"
           placeholder="Enter Password" required>
```

```
    <button type="submit">
        Login
    </button>
</form>
```

By applying filtering rules, you can generate conditional HTML with the `each()` and `filter()` methods provided by j2html or with the `stream()` and `filter()` methods shipped with Java 8. The code in **Listing 13** filters an employee list with the employee `id` set to an even number, first by using the j2html methods and then by using Java 8's stream and filtering approach.

Listing 13.

```
String j2htmlFilter = ul().with(
    each(filter(employees, e -> e.id % 2 == 0),
        employee -> li(
            h2(employee.name),
            p(employee.title)
        )
    )
).render();

String javaFilter = ul().with(
        rawHtml(employees.stream()
            .filter(e -> e.id % 2 == 0).map(
                    employee -> li(
                        h2(employee.name),
                        p(employee.title)
                    )
            ).map(DomContent::render)
             .collect(Collectors.joining()))
).render();
```

If you want to generate a custom HTML snippet inside a `<div>` generated by the `div()` method, you can use the `rawHtml()` method, as shown in **Listing 14**.

Listing 14.

```
. . .
div(
     rawHtml("<p>I like <em>HTML</em></p>")
);
```

The method creates an instance of the `UnescapedText` class, which is a wrapper class for the text representation (see **Figure 1**).

Handling JavaScript and CSS Resources

`TagCreator` provides `style()` and `script()` methods to create `<style>` and `<script>` HTML tags, respectively. A sample that includes resources is shown in **Listing 15**.

Listing 15.

```
script(
   rawHtml("alert('ok')")
);

style(
    rawHtml("body {background-color: blue}")
);
```

`TagCreator` also provides static methods for loading JavaScript and CSS resources directly from files. I show this in **Listing 16**. The methods suffixed with `_min` provide the ability to minify the content while transforming it into a string representation.

Listing 16.

```
scriptWithInlineFile("/test.js");
styleWithInlineFile("/test.css");

scriptWithInlineFile_min("/test.js");
styleWithInlineFile_min("/test.css");
```

The default minifier used for JavaScript simply strips out whitespace and newlines. CSSMin is the default minifier for CSS. Keep in mind that the content loaded with these methods is going to be represented as inline content in the generated HTML code. The actual resource loading is implemented within the `InlineStaticResource` class, where content read from the given file is used as input to create a container tag with the `script()` or `style()` methods.

Conclusion

If you need to generate HTML content on the server side in a declarative, type-safe way, j2html is a solution that you'll definitely want to have in your class path. Writing declarative Java code with one-to-one HTML component mapping in a builder fashion has never been this easy.

The HTML generation is fast compared to existing templating engines, so it's worth trying j2html if you have performance concerns. But j2html is not a template engine, and it doesn't have the capability to compete with one. If you were building a static website, I would advise not using j2html, because you'll probably end up having to generate all the content that you insert into the coding. If you are using a CSS framework (for example, something like Bootstrap), j2html is not going to play nicely with the framework. So, use j2html at your own risk.

However, j2html is definitely the Swiss Army knife that you'll want to have in your pocket if you are dealing with small, dynamically created web applications. `</article>`

Mert Çalişkan (@mertcal) is a Java Champion and the director of OpsGenie Academy. He is a coauthor of *PrimeFaces Cookbook* (Packt Publishing, 2013) and *Beginning Spring* (Wiley Publications, 2015). He is working on his latest book, *Java EE 8 Microservices*, while he develops Payara Server inside the Payara Foundation.

ORACLE CODE ONE

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

ORACLE CODE ONE

The Most Inclusive Developer Conference

- Discover the Latest on Java—from the Source
- Experience Leading-edge Technology Sessions
- Connect with Your Global Community

REGISTER NOW – **Save $400 by Aug. 11***

ORACLE®

// Silver Sponsor

redhat.

*Discount based on the onsite registration price.

BEN EVANS

Real-World Bytecode Handling with ASM

Scan, inspect, generate, and transform bytecodes on the fly with the ASM library.

The ASM library is a production-quality open source library for reading, writing, and manipulating JVM bytecode. It is used as part of many projects (including Gradle and the Kotlin compiler) and is used in shaded form (that is, as copied code with renamed packages to avoid namespace collisions) inside the JDK. In fact, it is used as the code-generation engine to enable runtime support of lambda expressions. Note that when you are working with ASM, you should use the external version, *not* the shaded version present inside the JDK.

In this article, I explain how to use ASM to perform some useful operations. In what follows, I assume that the reader is already familiar with some basics of JVM bytecode and the structure of class files. You can find the code from this article on the *Java Magazine* download page.

A "Hello World" Example

Let's take a look at a very traditional example, namely creating a class that will print "Hello World!" I will use ASM's `ClassWriter` API for this exercise. It is a simple API that makes heavy use of the Visitor pattern to achieve its goals.

My example produces a new class file, `HelloWorld.class`, completely from scratch. This class will not have any Java source code representation—that is, it will exist only as a compiled class.

The `HelloWorld.class` file will be created by another class, `MakeHelloWorld`, which will use the ASM libraries to assemble `HelloWorld.class` as output. However, the generated output class will run completely standalone and will not need ASM or any other JAR as a runtime dependency.

PHOTOGRAPH BY JOHN BLYTHE

Within `MakeHelloWorld`, the overall structure of the class creation is to use a `ClassWriter` field, referred to as `cw`, to build up the class by visiting these aspects of the class in turn:

- Overall metadata
- Constructor body
- Definition of the `main` method and its bytecode

After all aspects of the class have been visited, you can make the writer object ready for serialization by calling `visitEnd()` and then convert it to a byte array that can be written to disk.

In code, this overall driver method looks like the following, and it only needs to be called with the name of the output class:

```
public byte[] serializeToBytes(String outputClazzName) {
    cw.visit(V1_8, ACC_PUBLIC + ACC_SUPER, outputClazzName,
        null, "java/lang/Object", null);
    addStandardConstructor();
    addMainMethod();
    cw.visitEnd();
    return cw.toByteArray();
}
```

The serialization method starts by visiting the top-level metadata (class file version, flags, class name, and superclass name) and then calls methods to add a constructor and the main method, before finishing the class and converting it to a frozen byte array.

You create the constructor like this:

```
void addStandardConstructor() {
    MethodVisitor mv =
        cw.visitMethod(ACC_PUBLIC, "<init>", "()V", null, null);
    mv.visitVarInsn(ALOAD, 0);
    mv.visitMethodInsn(
        INVOKESPECIAL, "java/lang/Object", "<init>", "()V", false);
```

```
        mv.visitInsn(RETURN);
        mv.visitMaxs(1, 1);
        mv.visitEnd();
    }
```

This code works with a `MethodVisitor` that is created from the `ClassWriter` field before visiting each instruction in turn. After that, you must finish the method by noting how many stack slots the code uses. You do this by calling `visitMaxs()`.

The main method is added using another `MethodVisitor`:

```
void addMainMethod() {
    MethodVisitor mv =
        cw.visitMethod(ACC_PUBLIC + ACC_STATIC,
            "main", "([Ljava/lang/String;)V", null, null);
    mv.visitCode();
    mv.visitFieldInsn(GETSTATIC, "java/lang/System",
        "out", "Ljava/io/PrintStream;");
    mv.visitLdcInsn("Hello World!");
    mv.visitMethodInsn(INVOKEVIRTUAL, "java/io/PrintStream",
        "println", "(Ljava/lang/String;)V", false);
    mv.visitInsn(RETURN);
    mv.visitMaxs(3, 3);
    mv.visitEnd();
}
```

The code here is a little more complex, because objects need to be retrieved from static fields (via a `GETSTATIC` opcode and then the method must be called).

When I run `MakeHelloWorld`, I see `HelloWorld.class` appear in the file system. I can run the generated class in the usual way—`java HelloWorld`—and when I do, I see the familiar message appear.

The visitor API for ASM is easier to understand than some of the alternative APIs offered by the library. The general principle is that the different sections of the class file must be visited in the correct order (or skipped if there's nothing required for that section). The `MethodVisitor` interface is quite general.

For the case of `MakeHelloWorld`, I've obtained a visitor from the `ClassWriter`, and the actual implementation of the interface is `MethodWriter`. This keeps a reference back to the `ClassWriter` that created it and allows metadata about the method to be built up as the various `visit` methods are called.

The method represented by a `MethodWriter` needs to be sealed up when it is completed, and so `mv.visitEnd()` is called as the final action of the methods that create the methods in `HelloWorld`.

Let's decompile the generated class via `javap -c HelloWorld.class` and look at the bytecode that results from the ASM class generation:

```
public class HelloWorld {
  public HelloWorld();
    Code:
       0: aload_0
       1: invokespecial #8   // Method java/lang/Object."<init>":()V
       4: return

  public static void main(java.lang.String[]);
    Code:
       0: getstatic     #16  // Field
                             //  java/lang/System.out:Ljava/io/PrintStream;
       3: ldc           #18  // String Hello World!
       5: invokevirtual #24  // Method
                      //  java/io/PrintStream.println:(Ljava/lang/String;)V
       8: return
}
```

The correspondence between the Java bytecode instructions and the calls to the visitor API is clear, especially in the `main()` method. On the whole, ASM tries to stay very close to the bytecode format, while still providing enough of a high-level API to allow you to be productive.

Let's consider two more examples with a bit more complexity:

- Exploring the "lost update" problem, as illustrated by trying to increment a counter safely
- Exploring a prototype of a "safe class loader" that tries to prevent any user code from executing any native methods

Defeating Lost-Update Protection

I'll start with some simple code to demonstrate the lost-update effect, which I'll describe in a moment. One of the classic ways to introduce the effect is via an incrementing class:

```
public class Counter {
    private int i = 0;

    public int increment() {
        return i = i + 1;
    }
}
```

This class needs a driver:

```
int MAX_INC = 10_000_000;
Counter c = new Counter();
Runnable r = () -> {
    for (int i = 0; i < MAX_INC; i++) {
        c.increment();
    }
};
Thread t1 = new Thread(r);
```

```
Thread t2 = new Thread(r);
t1.start();
t2.start();
t1.join();
t2.join();
int disc = 2 * MAX_INC - c.increment() + 1;
System.out.println("Discrepancy: "+ disc);
```

The code is incrementing 10 million times on each thread, so `increment()` is called a total of 20 million times. However, when you run this code, you can clearly see that the code reports a discrepancy—not all calls to `increment()` appear to have been recorded in `c`.

This is the Lost Update antipattern, and even code as simple as `increment()` can exhibit it. This pattern is one of the classic pitfalls of concurrent programming in modern environments.

The lost update is caused by the operating system scheduler running both threads on CPU cores at the same time. Each thread increments the value of `i` as it sees it in the local CPU cache but does not flush the result to main memory. This results in an indeterminate number of updates being performed by both threads before the CPU flushes the cache line to main memory. These cache-only writes are then lost from the overall total being recorded in main memory.

The solution, of course, is to add the `synchronized` keyword to `increment()`, and then the discrepancy is always zero; all updates to `i` are flushed to main memory before being reread.

To see this, let's start with a synchronized counter and write a tool using ASM that switches off all synchronization in a class. Then, the transformed class will suffer from the lost-update problem even though the original code was safe.

The `OfflineUnsynchronizer` code will operate in the following way:

- Read in the class file using a `ClassReader`.
- Walk through the ASM representation of the class, using a custom `ClassVisitor`.
- Write the Java bytecode back out as a `byte[]`, using a `ClassWriter`.
- Save the bytecode as a transformed class file.

I need to know some details of Java bytecode to carry out the transformation.

For example, in Java bytecode, a `synchronized` method is represented by a flag called `ACC_SYNCHRONIZED` on the method, so I need to remove that flag from any method that I visit. However, to be really sure that all the synchronization is gone, I also need to know that the block form of synchronization is represented slightly differently. If I have some code like this:

```
Object o = ...
synchronized (o) {
    // ...
}
```

It will be turned into a sequence of bytecodes that looks a bit like this:

```
[Sequence that leaves o on top of the stack]
monitorenter
// ...
[Reload o]
monitorexit
```

Both `monitorenter` and `monitorexit` bytecodes consume the top of the stack and lock or unlock the object that they find there. So if these opcodes were replaced with a basic `pop`, this would strip the synchronization out of any method body that is encountered.

The resulting code is represented by the following two simple classes: an `UnsynchronizingClassVisitor` and an `UnsynchronizingMethodVisitor`, both of which extend ASM framework classes:

```
public class UnsynchronizingClassVisitor extends ClassVisitor {

    public UnsynchronizingClassVisitor(int api, ClassVisitor cv) {
```

```
            super(Opcodes.ASM5, cv);
        }

        @Override
        public MethodVisitor visitMethod(int flags, String name,
          String desc, String signature, String[] exceptions) {
            int maskedFlags = flags & (~ACC_SYNCHRONIZED);

            MethodVisitor baseMethodVisitor =
                super.visitMethod(maskedFlags, name, desc,
                                  signature, exceptions);

            return new UnsynchronizingMethodVisitor(baseMethodVisitor);
        }
    }
```

The `UnsynchronizingClassVisitor` class uses a Decorator pattern: it takes the `baseMethodVisitor` and wraps it by adding functionality that is called only when a no-argument opcode is encountered in the body of the method, as shown in this code:

```
public class UnsynchronizingMethodVisitor extends MethodVisitor {
    public UnsynchronizingMethodVisitor(MethodVisitor mv) {
        super(Opcodes.ASM5, mv);
    }

    @Override
    public void visitInsn(final int opcode) {
        switch (opcode) {
            case Opcodes.MONITORENTER:
            case Opcodes.MONITOREXIT:
                super.visitInsn(Opcodes.POP);
```

```
                return;
            }

            super.visitInsn(opcode);
        }
    }
```

I use the following bit of code to drive this transformation:

```
try (InputStream in =
        Files.newInputStream(Paths.get(fName))) {
    ClassReader classReader = new ClassReader(in);
    ClassWriter writer =
       new ClassWriter(classReader, ClassWriter.COMPUTE_FRAMES);

    ClassVisitor unsynchronizer =
       new UnsynchronizingClassVisitor(writer);
    classReader.accept(unsynchronizer,
       ClassReader.SKIP_FRAMES | ClassReader.SKIP_DEBUG);

   Path newClazz = Paths.get(transformName(fName));
   Files.write(newClazz, writer.toByteArray());
} catch (Exception ex) {
    System.err.println(
        "Exception whilst reading class: " + fName);
    ex.printStackTrace(System.err);
}
```

Now, if I take a synchronized version of the `Counter` class, I can run it through the `Offline-Unsynchronizer`, and the resulting transformed class will suffer the lost-update problem even though the original code was safe.

Ruling Out Native Code

Java bytecode is platform-independent, so it cannot call operating system libraries directly (for example, to handle I/O operations). Instead, Java programs (including the JDK) call out to native methods (written in C) that in turn call the relevant parts of the operating system.

Suppose you have a use case where you want to allow users to execute unknown code as part of a framework or container. Such a capability has obvious security concerns, so you might want to reduce the risk by disallowing certain actions—such as running native methods—in the users' classes. Fortunately, the Java security model relies on class loading, and it allows you to hook into the loading process to customize how (and whether) new code is loaded. [For more on how class loading works, see the article "How the JVM Locates, Loads, and Runs Libraries" by Oleg Selajev, which you can download as a PDF. —*Ed.*]

The overall scheme could look like this:

- Write a custom class loader.
- During class loading, inspect every "call site" where a method is called.
- Check to see whether the metadata for the method indicates that the method is native.
- If it is, reject the class and fail class loading.
- If you reach the end without failing, the class is good and can be loaded.

Here's how to write a class loader that will reject any non-pure Java classes it is asked to load:

```
public final class PureJavaClassLoader extends ClassLoader {
    private final List<String> auxClasspath = new ArrayList<>();

    public PureJavaClassLoader(ClassLoader parent) {
        super(parent);
    }

    public void setupClasspath(final String auxiliaryClassPath) {
        for (String entry : auxiliaryClassPath.split(":")) {
            if (entry.startsWith("/")) {
                auxClasspath.add(entry);
```

```
            } else {
                System.err.println(
                    "Bad classpath entry seen: " +
                     entry + ", ignoring");
            }
        }
    }

    Path findClassFile(String qualifiedClassName)throws IOException {
        final String fileName =
            qualifiedClassName.replaceAll("/", "\\.") +".class";
        for (String s : auxClasspath) {
            Path trial = Paths.get(s, fileName);
            if (trial.toFile().exists())
                return trial;
            }

            throw new IOException("Class "+ qualifiedClassName +
                                  " not found on classpath");
        }
    }
```

For simplicity, I'll manage an *auxiliary class path* of directories that I want to search for classes to load, rather than using the main class path. The `findClassFile` method is a helper that locates the file corresponding to a qualified class name. The real action is in the `findClass` method to which class loaders delegate from `loadClass()`. This is where I implement the check for native code:

```
@Override
public Class<?> findClass(final String qualifiedClassName) throws
  ClassNotFoundException {
```

```
Class<?> cls = null;
try {
    return super.findClass(qualifiedClassName);
} catch (ClassNotFoundException ignored) {
    try (final InputStream in = Files.newInputStream(
            findClassFile(qualifiedClassName))) {
                final byte[] allClassBytes = in.readAllBytes();
                final ClassReader classReader =
                    new ClassReader(allClassBytes);
                final PureJavaCheckingClassVisitor
                    classVisitor =
                      new PureJavaCheckingClassVisitor();

                // If there's debug info in the class,
                // don't look at it
                classReader.accept(
                    classVisitor, ClassReader.SKIP_DEBUG);

                if (classVisitor.containsNative()) {
                    throw new ClassNotFoundException(
                   "Class cannot be loaded - contains native code");
                } else {
                    return defineClass(null, allClassBytes, 0,
                                       allClassBytes.length);
                }
            } catch (IOException e) {
                throw new ClassNotFoundException(
                    "Error finding and opening class", e);
            }
        }
    }
```

In the previous code, I call `in.readAllBytes()` directly, rather than passing `in` to the `ClassReader` constructor. This is because the ASM class `ClassReader` consumes input streams, so I can't reuse `in` after it's been used to create a class reader.

Next, I create an instance of our custom class visitor, `PureJavaCheckingClassVisitor`. This visitor simply visits the metadata for each method in the class being considered and records whether any method is native. It is defined as the following:

```
public class PureJavaCheckingClassVisitor extends ClassVisitor {
    private boolean containsNative = false;

    public PureJavaCheckingClassVisitor() {
        super(Opcodes.ASM5);
    }

    @Override
    public MethodVisitor visitMethod(int flags, String name,
      String desc, String signature, String[] exceptions) {
        if ((flags & ACC_NATIVE) > 0) {
            containsNative = true;
        }

        return new MethodVisitor(Opcodes.ASM5) {};
    }

    public boolean containsNative() {
        return containsNative;
    }
}
```

If the class visitor ever sees a native method, it sets a flag. The flag is read by the `PureJava-ClassLoader`, which rejects the class with a `ClassNotFoundException` if the flag has been set. This

exception is used, rather than the alternative natural choice (`SecurityException`), because the contract of `ClassLoader` (which is the supertype of this class) uses the checked exception `ClassNotFoundException`. In this circumstance, use of a runtime exception (such as `SecurityException`) could violate some expectations of clients of the classloader.

Assuming that an exception has not been thrown, the bytes of the class file are fed to `defineClass()`, which is a protected method defined on `ClassLoader` so it is accessible only to subclasses—effectively custom class loaders. This returns the `Class<?>` object that I return from `findClass()`, and the class is successfully loaded.

Conclusion

A word of caution: the previous example will indeed prevent any classes with native methods from being loaded. However, in a real environment, you would also have to take into account other cases, such as the following:

- Code that calls a native method of an already-loaded class (the transitive case)
- Reflective access to native methods
- Invocation of native methods via the `MethodHandles` interface

Not only that, but some native methods are essential for proper functioning of virtually all Java programs (such as `getClass()` or `Object::hashCode`).

A full discussion of what would be required to fully restrict native code from running is too far afield for this article. In practice, some sort of approved list of core native methods within the JDK would have to be used. Nevertheless, note the things I did with ASM in the example: I read through bytecodes for a given release of Java, skipped over debugging data, and identified specific bytecodes. And earlier, I transformed bytecodes on the fly. `</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. He has written four books on programming, including the recent *Optimizing Java* (O'Reilly).

NICOLAI PARLOG

Migrating Your Library to Java Modules

Migration requires careful planning and diligent execution, while sidestepping several "gotchas."

Two issues all library writers will have to face sooner or later is how to make their library work with Java modules and how to convert the library into the Java modules introduced in Java 9. The process for doing these things is not entirely straightforward and requires careful planning, the right tools, and some engineering work.

In this article, I go through the necessary steps for taking a Java 8 library all the way to modules. I begin by discussing tools, the likely migration challenges, and running parallel builds for multiple Java versions. I then look at configuring your library for use as an automatic module, and finally I provide guidance on how to modularize your library. With this information, you can plot your own migration and modularization strategy. It will be very helpful if you know the basics of the module system, particularly about strong encapsulation of module internals—but if you don't, you will still be able to follow most of what's covered here.

I should point out that modules are not required for programs to run on Java 9 or later, but you do need to know how libraries work as modules.

The Right Tools

For the best Java 9 integration into your favorite IDE, you should use that IDE's most current version, because Java 9 support is constantly improved. If being on the cutting edge isn't for you, you should at least use IntelliJ IDEA 2017.2 or Eclipse Oxygen.1a (before that version, Eclipse needed Java 9 support plugins; they are obsolete now). And if you use NetBeans, you need at least version 9.0.

Similarly, use a current version of your build tool. In the case of Maven, this should at least be version 3.5.0 of the application itself and version 3.7.0 of the compiler plugin. For Gradle, use at least version 4.2.1.

Migration Challenges for Moving to Java 9

Java 9 introduced modules to the ecosystem, and with it came many changes. Some were caused by the presence of modules. These changes caused subtle differences in Java's behavior—and a few could be considered incompatibilities. Here's a brief summary of what you can expect and what can be done about it. Note that you're unlikely to encounter all of these issues, so don't worry too much.

Failing access to JDK-internal APIs. The most obvious problem you might encounter is access to internal APIs. If your code depends on classes from `sun.*` packages or most `com.sun.*` packages, you're bound to see compilation errors like this:

```
error: package com.sun.java.swing.plaf.nimbus is not visible
import com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel;
                              ^
    (package com.sun.java.swing.plaf.nimbus is declared
    in module java.desktop, which does not export it)
1 error
```

The proper fix is to stop using such classes, but if you can't do that yet, you can make them available at compile time and runtime by using the command-line flag `--add-exports`. If you're accessing JDK-internal APIs at runtime via reflection, take a look at the flags `--add-opens` and `--illegal-access`.

A word of warning regarding command-line flags: if you use flags to fix any of the problems I'm describing here, your users will usually need to apply the same flags when running their application with your library. That can be a serious inconvenience, so try to avoid that if at all possible. If you absolutely must use flags, make sure you document them well.

Dependencies on unresolved Java EE modules. Historically, the JDK contained a few APIs that belonged to Java EE—for example, JavaBeans Activation Framework (JAF), JTA, JAXB, JAX-WS, and CORBA. To more clearly separate Java SE and Java EE and in preparation for Java EE's move away from Oracle's aegis, these APIs were deprecated in Java 9 and will be removed in Java 11.

If your library depends on these APIs being present in the JDK, you will get errors like this when building your code on Java 9 or later releases:

```
error: package javax.xml.bind is not visible
import javax.xml.bind.JAXBException;
                ^
    (package javax.xml.bind is declared in module java.xml.bind,
        which is not in the module graph)
1 error
```

Although it is possible to fix the error with the command-line flag `--add-modules`, that will help you only until Java 11. The long-term solution is to pick an implementation of the API you need and add it as a regular dependency.

Failing casts to URLClassLoader. Some libraries interact with the class path—for example, to examine its content or add additional JAR files. They often do that by casting the system class loader to `URLClassLoader`, because it has the needed methods. But Java 9 changed the class-loading strategy, and it uses a different class loader. Therefore, such casts will fail with an error similar to this:

```
Exception in thread "main" java.lang.ClassCastException:
    java.base/jdk.internal.loader.ClassLoaders$AppClassLoader
    cannot be cast to java.base/java.net.URLClassLoader
        at monitor.Main.logClassPathContent(Main.java:46)
        at monitor.Main.main(Main.java:28)
```

The solution depends on what exactly you want to do with the class loader, including the following:

- If you only want to examine the class path content, look at the system property `java.class.path`.
- To add new JAR files to the running application, you need to create a new class loader (you can use `URLClassLoader` for this) that delegates to the existing system class loader.
- For other use cases, look at `ClassLoader`; a few methods have been added, and perhaps you'll find what you need by consulting the Javadoc.

New Java version string format. From Java 9 on, the system property `java.version` and its siblings no longer start with `1.x` but instead start with `x`. Therefore, on Java 9 you get `9`, `9.0.1`, and `9.0.4` back, and you'll get something similar for Java 10 and later. If you're tired of parsing that string, check out the new type `java.lang.Runtime.Version`, which was introduced in Java 9 and provides easy access to version information.

There are a few more details, such as the changed JDK folder structure, that could theoretically go wrong when you migrate to Java 9, but as a library developer, you're unlikely to encounter them.

If users start modularizing their own project, they have to declare each dependency—including declaring your library—with a requires directive.

Building on Multiple Java Versions

As you've seen, Java 9 contains several changes that might affect your library's behavior. It stands to reason that while you are updating your code to adapt to these changes, you need a continuous integration (CI) build that compiles and tests on Java 9 and later. This step usually means setting up an additional build that runs in parallel to your existing one.

You might be tempted to avoid a parallel CI build by raising your project's baseline to Java 9 or Java 10. However, that would considerably reduce your user pool, at least for now, because many sites have not yet migrated to Java 9 or later. Even more important, with new releases coming out every six months, you're likely to need parallel builds for future releases of your library anyway. It pays to be able to set up multiple CI builds on various Java versions. So how do you go about that?

Configuring the CI server. I generally recommend running the entire build on the desired Java version (as opposed to just compiling and testing with it). The first step, then, is to configure

your CI server to actually execute the build several times, each time with a different Java version. Jenkins and Travis CI, among others, make this fairly easy.

Even if your build takes a long time and resources are scarce, don't put this off. You should build with the nonbaseline versions at least every night. While this practice allows problematic commits to stay undetected for an entire day, you will at least find out by next morning that something went wrong.

Configuring the build steps. Running the build is the easy part; configuration can be the difficult part. It is possible that, depending on the Java version running the build, you'll need to apply command-line flags to compilation or test runs, change a dependency's version, or edit a build step's configuration. To be able to do this, you need to familiarize yourself with your build tool's support for conditional configuration.

For Maven, profiles are what you're looking for. The following block creates a profile that automatically activates itself when the build runs on Java 9 or later:

```
<profiles>
  <profile>
    <!-- automatically activate the profile if running on Java 9 -->
    <activation>
      <jdk>[9,)</jdk>
    </activation>
    <!-- version-specific build configuration goes here-->
  </profile>
</profiles>
```

You can then apply a version-specific configuration inside that `<profile>` block. For example, to export an internal package from a platform module that Google Guice depends on, you could do this:

```
<profile>
  <id>java9+</id>
```

```
<activation>
  <jdk>[9,)</jdk>
</activation>
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-failsafe-plugin</artifactId>
      <configuration>
          <!-- deny illegal access to detect new problems -->
          <!-- Google Guice 4.1 uses the internal method
               java.lang.ClassLoader::defineClass -->
          <argLine>
             --illegal-access=deny
             --add-opens java.base/java.lang=ALL-UNNAMED
          </argLine>
      </configuration>
    </plugin>
  </plugins>
</build>
</profile>
```

Configuring the build process. If you're truly unlucky, you need to configure not only individual build steps but the entire build process. This happens, for example, if a plugin you're using as part of your build relies on Java EE modules. In this case, the build process itself needs command-line flags to work on Java 9.

For Maven, the solution is to create a `.mvn/jvm.config` file in your project's root folder and put the required command-line options in there. When a Maven command is executed, the command will look in that file and apply the options to the Java process it launches to run Maven.

The problem is that Maven also does this on Java 8, which then fails due to unknown command-line options such as `--add-modules`. As a solution, I've named the configuration files `jvm9.config`, so they are ignored by default, and then I let the CI server call a script that renames them all to `jvm.config` before launching the build. It's not exactly beautiful, but it works.

The final step into the future that Java 9 brought is to turn your project's JAR file into a modular JAR file.

I never dealt with these Maven features before building on Java 9, and in the beginning, I wasn't enthused to dig through them. But I found it to be a great opportunity to get to know Maven a little better, and that has paid off many times over.

Your Library as an Automatic Module

Congratulations! If you've gotten this far, your project is built and tested on Java 9. Your users will be thankful and will start using your library—at first on the class path, but soon they'll want to take the next step and use it as a module. How does that work? After all, you have not created a module yet.

Automatic module crash course. If users start modularizing their own project, they have to declare each dependency—including declaring your library—with a `requires` directive. For that to work, though, they need a module name to use with `requires`. And they need to place that on the module path.

If the Java runtime encounters a JAR file without a module descriptor (`module-info.class`) on the module path, it will create a so-called *automatic module* for it. It's just like a regular module, but there are certain assumptions about its properties: an automatic module exports all packages and can read all other modules.

The more interesting aspect is the name. Ideally, the automatic module name is defined with the manifest entry `Automatic-Module-Name`. If that's not the case, the module system derives a name from the JAR file's name. That fallback is obviously unstable across development environments and causes additional problems if your library ever changes its module name (for example, because you modularize it and give it a proper name).

If your users work on an application, however, it's not too bad: they can simply update all their `requires` directives and use your library under its new name. If they also work on a library, things are much more complicated. Chances are, they released a version of their project that requires your JAR file by its filename, and now your newer JAR files cannot fulfill that dependency because your module has a different name. Even worse, if a user transitively depends on your library twice, once under each name, that user is in serious trouble.

Defining the automatic module's name. Consequently, users will be wary of depending on your plain JAR file as a module if you don't set the `Automatic-Module-Name` entry in your JAR manifest. So, once you've made sure your library works well and you don't expect any major refactoring where packages get moved between JAR files (if you ship more than one), you should pick a module name (see the next section for more on that) and set the manifest entry.

With the Maven JAR plugin, you could do that as follows:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-jar-plugin</artifactId>
  <configuration>
    <archive>
      <manifestEntries>
        <Automatic-Module-Name>$name</Automatic-Module-Name>
      </manifestEntries>
    </archive>
  </configuration>
</plugin>
```

Just replace `$name` with the actual name. As soon as you've done that, other modules can start requiring your JAR file as an automatic module under a stable name. Make sure you advertise that fact and the chosen module name prominently in your documentation.

Modularizing Your Library

The final step into the future that Java 9 brought is to turn your project's JAR file into a modular JAR file. That's actually fairly simple: create a module declaration (a file `module-info.java`) in your project's root source folder and use it to define your library's module name, dependencies, exports, and services, for example:

```
module $name {
    requires $module;
    exports $package;
    uses $service;
    provides $service with $provider
}
```

I'll go through the different properties shortly, but before I do, I want to discuss what else you need to do: nothing. If your tools are up to date, they will automatically detect that you're building a module and do the right thing. IDEs and build tools will include the module declaration in the list of files to compile, put dependencies on the module path, and package the resulting `module-info.class` into the JAR file they're building. There usually isn't anything explicit you need to do.

With that said, I'll turn to the details of modularizing your library. Note that this is not a module system tutorial; I expect you to know how the module system works. [If you don't, read up on it in this article. —*Ed.*]

Picking a module name. Just like package names, module names should be globally unique. The easiest way to achieve that is to take the same approach as with package names: pick a domain that is associated with the project and reverse it. If this results in your module name being a prefix of your package names, you did it right.

Declaring dependencies. By default, each nontest dependency in your build configuration should result in a `requires` directive in the module declaration. Note that you can also have

optional dependencies (with `requires static`) and dependencies that are exposed to modules requiring your library (with `requires transitive`).

One option is to depend on automatic modules. As I implied when discussing automatic modules, it is technically feasible and generally acceptable to depend on nonmodular JAR files, so not all of your dependencies need to be modularized for you to modularize your library. But be aware that it is extremely careless to require nonmodular JAR files that do not define an automatic module name. As pointed out earlier, you will expose your users to serious problems if you do that—so please don't.

Consequently, you should not publish modular JAR files for your library unless all of your dependencies are either a module or at least define their automatic module name with a manifest entry. In fact, Maven explicitly warns that you should not publish the artifact if you require a module by its filename. I would even go a step further and implore you to check all of your transitive dependencies as well—none of them should be required by their filename.

Be aware that it is extremely careless to require nonmodular JAR files that do not define an automatic module name.

Handling split packages. Another consideration is how to handle split packages. Once you start treating dependencies as modules, the module system exposes them to more checks, which they might fail. One example is that some JAR files *split packages*. This refers to a situation in which a package—say, `org.lib`—and two different JARs each have a type in that package—say, `org.lib.Foo` in `foo.jar` and `org.lib.Bar` in `bar.jar`. The module system does not allow that and complains loudly when these JARs end up on the module path.

Although there are ways to make such a configuration work (look up `--patch-module` if you need to), they require extensive build tool configuration, which your users would have to repeat. My recommendation is not to put that task onto them. Therefore, if your dependencies split packages, work with them to fix the problem before publishing your project as a module.

Exporting packages. The next step after naming your module and declaring your dependencies is to define your public API. The technical aspect is simple: just name in an `exports` directive all

the packages whose public types should be accessible. The devil is in the details, though.

Before the module system was introduced, there was no way to mark classes that needed to be public but that you didn't consider a public API. You could always back out of supporting them by adding a comment or having your documentation say which classes were supported. That meant a package could contain some supported and some unsupported public classes—the new module system takes that option away.

By exporting a package, you're not only making all public types in it accessible; you're also making a promise that you're going to support all of them. *If you don't want to support a class, that class needs to be nonpublic or in a nonexported package.*

This means that defining your public API might require some shuffling around of classes that you do not support. The most common solution is to move them to a different package, which is subsequently not exported.

Using and providing services. If your library interacts with other JAR files by providing or using services, you need to include this information in your module declaration, as follows:

- For every service you use—meaning, for every type that you call `ServiceLoader::load` with—you need to add a `uses` directive to the declaration.
- For every class name in a file in `META-INF/services`, you need to add the line `provides $file-name with $class-name` to the declaration.

Note that your users are not forced to use your modular JAR file as a module, and if they don't, the module declaration is ignored. That means you need to keep the files in `META-INF/services` around and in sync with your module declaration.

Delivering multiple modules. I want to point out two details for the situation in which you deliver your library in more than one JAR file. The first is that, as I've discussed, split packages are a problem. Therefore, make sure your JAR files don't split packages between them. If they do now, you need to move classes around or rename packages before modularizing your library.

The second point concerns the moving around of classes. With the class path, it didn't matter which JAR file a class came from. As long as some JAR file contained a needed class, the application worked just fine. That made it possible to move classes between JAR files without

breaking downstream code (if it was done carefully). Modules make this much harder. For a module to access types in one of your library modules, it needs to read that exact module, which usually means it needs to require it. If an application requires only one of your several library modules and you move a class out of that module and into another one, the application will not be able to access the class, and its compilation and execution will fail.

So, moving types between modular JAR files has a higher risk of causing compatibility problems than with plain JAR files. Be careful when you have to do it.

Conclusion

Going from Java 8 to full modules on Java 9 and later releases can be quite a lot of work, but for most projects it isn't. The smaller your library and the more plain Java code it contains, the less effort it typically requires. `</article>`

Nicolai Parlog (@nipafx) has found his passion in software development. He constantly reads, thinks, and writes about it, and codes for a living as well as for fun. He wrote the just-published book *The Java Module System* (Manning). He blogs about software development at codefx.org and is a long-tail contributor to several open source projects.

JEP 296: Consolidate the JDK Sources into a Single Repository

This proposal, which was implemented in Java 10, illustrates some of the difficulties in maintaining a codebase that has the special requirements of the JDK. Specifically, Java Enhancement Proposal (JEP) 296 addresses the following problem: many code commits span several of the individual repositories in the source base. Consequently, it's not possible to make atomic commits to the source code manager. (When speaking of a commit, *atomic* means that the action is carried out in its entirety as a single operation—or not at all.) Clearly, if a commit must be done separately to several repositories to be complete, there is no possibility of atomicity. This means that every commit is labor-intensive and fragile until completed.

In the JDK, there are eight principal repositories, and roughly 1,100 defects and requests span more than one of them. So this JEP proposed merging these repositories into a single repository.

The advantages of a central, single repository are increasingly being recognized by companies that have large codebases. For example, Google and Facebook have frequently broadcast their use of a single, monolithic repository for all their products' code. The Google repository, which contains more than 2 billion lines of code, has reportedly generated several advantages: unified versioning, simplified code sharing, and greater reuse.

Invariably, this JEP will require changes to internal development practices and DevOps. Should those changes have value to Java developers at large, we'll report on them in a future issue.

A Wealth of Libraries

The many Java libraries we have covered in the last few years

Blockchain

▶ **web3j:** Create Ethereum transactions and smart blockchain contracts. Jan/Feb 2017

Bytecode Operations

▶ **Byte Buddy:** Generate bytecode on the fly, write and deploy agents, and more. Nov/Dec 2015

▶ **ASM Library:** The most widely used Java bytecode manipulation library. Page 39 of this issue

Command-Line Parsing

▶ **JCommander:** Simplified parser for complex command lines. Nov/Dec 2015

Control Flow

▶ **JDeferred:** Asynchronous processing using promises and futures. May/June 2017

Core Java SE

▶ **Java 9 library updates:** Collections and Streams. July/Aug 2017

▶ **Java 9 library updates:** Optionals and CompletableFutures. Sept/Oct 2017

Data

▶ **JSON-P:** How to use the JSON library in Java EE. July/Aug 2016

Document Processing

▶ **iText:** Generating and manipulating PDF files. Page 15 of this issue

JSF

▶ **OmniFaces:** A single, integrated, multipurpose utility library for JSF. Jan/Feb 2016

HTML

▶ **j2html:** Generating HTML on the fly. Page 27 of this issue

▶ **jsoup:** HTML parser and data extractor. May/June 2017

Reactive Programming

▶ **Vert.x and RxJava:** The popular, wide-ranging library for reactive programming. Jan/Feb 2018

And several articles on the mechanics of Java libraries

▶ Designing a library properly: The developer of Joda-Time explains good library design. May/June 2017

▶ How the JVM locates, loads, and runs libraries. Nov/Dec 2015 (PDF)

▶ Building libraries with Java 9 modules. Page 53 of this issue

IAN DARWIN

The State Pattern

Elegantly manage state transitions without large switches or numerous if statements.

We're all familiar—even nonprogrammers, implicitly—with the notion of a state machine. The machine is anything that can be in one of several discrete states with defined transitions. A simple coffee machine can be only on or off. The Start button turns it on. Another button, or a sensor of some kind, turns it off. It cannot be 72% on, nor can it be in a state of making popcorn or playing music. A media player, whether implemented in hardware or software, can be in any one of several states: stopped, paused, playing, or rewinding. The Play button transitions from the stopped state to the playing state. The Pause button transitions from the playing state to the paused state. The Stop button transitions from either the playing or paused state to the stopped state.

The behavior of the buttons (or Java methods) is contextually dependent upon what state the machine is in. Pressing Stop when the machine isn't doing anything is ignored. Pressing Start when the machine is running also will usually be ignored.

It's common to diagram these transitions by using state diagrams similar to the one in **Figure 1**.

Figure 1 is for a stateful device such as a media player, but you can imagine the diagram would be similar for, say, 3D printer firmware and many other kinds of devices. And the same general idea would apply for a graphics application, which might have edit and preview modes/states, or for a computer game with player-play, computer-play, and won/lost states.

The notion of a state machine gave rise to—and gave its name to—the *State pattern*. In this pattern, an object can be in exactly one of a fixed number of states, and transition between them causes changes in the object's behavior.

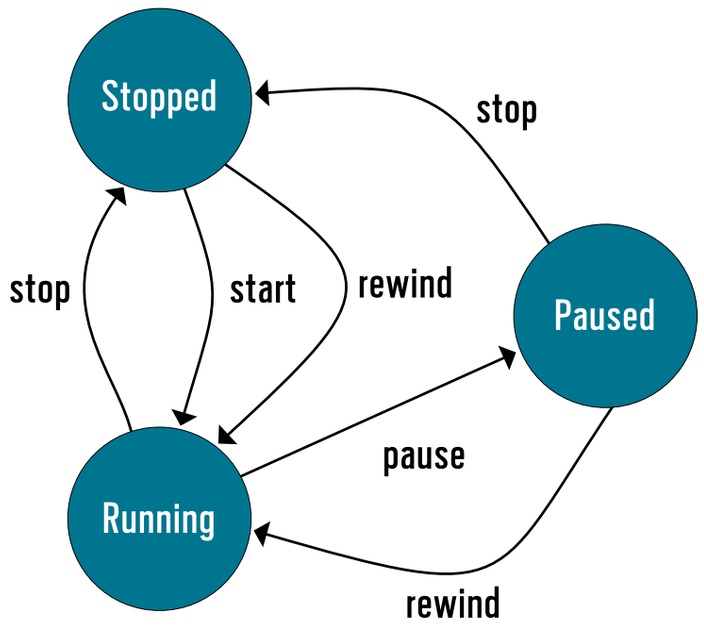


Figure 1: A typical state diagram

State in an Application

You may have noticed that Java documentation refers to "the state of an object," meaning the contents of the instance fields in an object. This is a related meaning of the word *state,* and in fact, the two meanings overlap in large applications. The app needs to keep track of the state it's in, and that information is typically stored in the state of objects. But there is a very specific meaning of the term *State pattern*, as defined in the "Gang of Four" book *Design Patterns: Elements of Reusable Object-Oriented Software*, where it was first popularized. The pattern will "allow an object to alter its behavior when its internal state changes. The object will appear to change its class."

Before exploring this definition of the State pattern and its implementation, though, let's look at a less-well-structured way to manage the state of a media player. In **Figure 1**, which is a simplified representation of the solution I'll implement, notice that the transitions are labeled start, stop, pause, and rewind. I could implement those as methods. I'd also need a field to keep track of what state the object is actually in, using a Java `enum`. (In older code, you might see a bunch of `final static ints` or `chars` defined to track the state.)

```
enum StateName { STOPPED, PLAYING, PAUSED, REWINDING };
StateName currentStateName;
```

Then, I might define each of the four methods, using logic like the following code. Because lots of things work only in one or another state, I need to check what state the object is in. So, I end up with a lot of `if` statements or a giant `switch` statement in each method.

```
public void start() {
    if (currentStateName == StateName.STOPPED) {
        currentStateName = StateName.PLAYING;
        startPlay();
    } else if (currentStateName == StateName.PAUSED) {
        currentStateName = StateName.PLAYING;
        resumePlay();
    } else if (currentStateName == StateName.PLAYING) {
        System.out.println("Already playing!");
    } else if (currentStateName == StateName.REWINDING) {
        System.out.println("Wait a while, OK?");
    }
}
```

I also need the same amount of conditional code in each of the four methods. This becomes a serious maintenance issue when you need to add or change functionality. And seriously, who's ever worked on a project for a few months and not had to add a feature? If you're not getting feature-add requests, you probably have no users!

The State pattern suggests a cleaner way to organize the code. To refactor the previous approach using the State pattern, I'll start by creating an interface called `State`, and make four instances of it, one for each state the player can be in. First, here's the interface:

```
interface State {
    void stop();
```

```
    void start();
    void pause();
    void rewind();
    default enterState() {
        // Only some states will need this
    }
}
```

In **Figure 2**, I show a generic class diagram for this. The `process()` method corresponds to the various processing methods such as `stop()`, `start()`, and so on. It is common, but not required, to provide a method such as `enterState()` to be called upon entry into the state, to configure things appropriately.

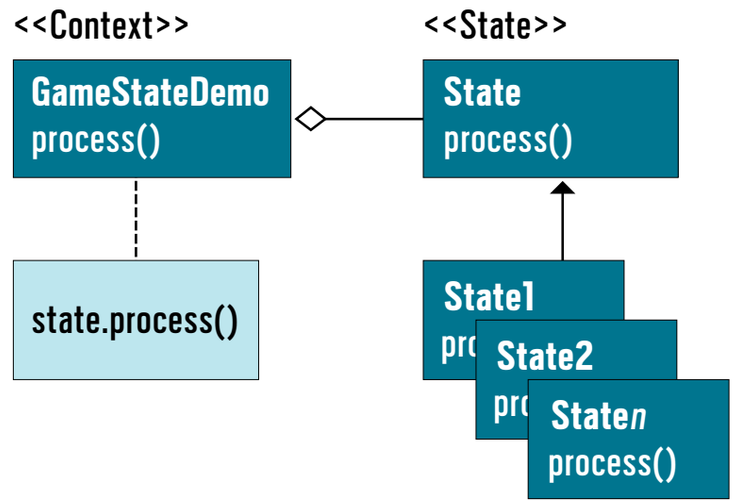


Figure 2: A typical class diagram in the State pattern

The class *using* the `State` classes is called the `Context` class, and the `State` classes are frequently (but not necessarily) written as *inner classes* inside the `Context` class. Note that the term *Context class* is an important concept in the following explanations.

(By the way, all the code for this article is in my GitHub repository. The file `PlayerStateDemo.java` is the one I'm discussing here.)

With the `State` defined, I now define the four states by subclassing. Because the app needs only one instance of each `State` class, the classes are instantiated as anonymous classes and saved as fields.

However, if your program changes state only rarely, you could create the `State` subclasses as named classes, configure them in the constructor, and lazily instantiate the `State` subclass only when switching into it.

Here is the code for the "Stopped" `State` subclass, which defines how the various operations will be performed when invoked while the player is in the stopped state. In this example, `stop()` is some low-level hardware control method that might stop a motor moving on a DVD player. The commented-out `setIcon()` call symbolizes some update to the GUI.

```
State stoppedState = new State() {
    @Override
    public void enterState() {
        stop();
        // setIcon(Icon.stopped);
    }

    @Override
    public void stop() {
        // Do nothing - already stopped
    }

    @Override
    public void start() {
        currentState = playingState;
        currentState.enterState();
    }
```

```
    @Override
    public void pause() {
        // Do nothing - already stopped
    }

        @Override
        public void rewind() {
            currentState = rewindingState;
            currentState.enterState();
        }
    };
```

The last two statements in `rewind()` show how an object of the `Context` class can "appear to change its class." Let's look at the main program's methods, such as `stop()` and `start()`:

```
PlayerStateDemo context = new PlayerStateDemo();
System.out.println("Initial state: " + context.getState());
// User presses the Start button
context.start();
System.out.println("Current state: " + context.getState());
// User presses the Stop button
context.stop();
System.out.println("Current state: " + context.getState());
```

These methods invoke or "delegate to" the current `State` object's methods. Just by having a new value assigned to the `currentState` variable, the program appears to change its class, because different code will be invoked from the same method call based on whichever `State` object is current. For example, if you call `stop()` while playing, the player will stop, but if you call the very same method a second time, it will do nothing, because the first invocation invokes the `stop()` method in `playingState` while the second calls the `stop()` method in `stopState`.

In this example, invalid but harmless operations are silently ignored, because that's how real media players work. Obviously, there will be some cases where it makes sense to signal invalid usage—for example, by logging or throwing an exception.

You need to ensure that the variable containing the state can never be null; in my example, I use a field initialization to ensure that this is set from the beginning:

```
State currentState = stoppedState;
```

While this way of doing things may seem like more code than the maze of `if` statements shown earlier (in fact it is slightly longer, by line count), it is worthwhile in terms of more-maintainable code. You can more clearly see what goes where, and—just as important—the compiler will probably tell you if you forgot to write a required method for a particular state, due to the interface requirement that all methods be implemented.

The State pattern sorts out the state-specific behavior, making it easier to read and maintain the code and easier to add new states.

There are several possible variations to how you implement the pattern. Depending on the scope of the application, it might be useful to move the interface to be a noninner class of the Context, and have the `Context` class also implement the interface, so the compiler will check that all the delegation methods are present with the correct arguments and return types. The `State` classes themselves don't have to be inner classes at all. If it made sense, they could be separate, package-level visibility classes; the `Context` class could create them. Here, the `Context` would typically pass a reference to itself into the constructor, for example:

```
State shuttingDownState = new ShuttingDownState(this);
```

This is because the `State` classes are likely to need access to nonpublic methods and fields in the `Context` class.

Client code using the `Context` class does not need access to the actual `State` object, and typically it should not have access, so as to enforce encapsulation. If necessary, the `Context` class could provide generic information methods such as `isPlaying()`, `isStopped()`, and so on to export information, where it makes sense, without exporting implementation details. But if you don't need or want encapsulation at this boundary, you could have a method such as this in the `Context` class:

```
public State getState() { return currentState; }
```

For an in-between version (a sort of mild encapsulation), you could use this:

```
public String getState() { return currentState.getClass().getSimpleName(); }
```

However, if you have used anonymous classes, note that `Class.getSimpleName()` returns the empty string on at least the standard JVM, so use `getName()`.

In this example, I've used the `State` code to perform the transitions. The `Context` class could as easily handle the transitions itself. As with all design patterns, it is just a pattern. Therefore, how you implement it is up to you, as long as you follow the general guidelines.

A Bit of History

The State pattern, like most good patterns, is language-agnostic. I came close to figuring out this pattern a long time ago in a job far, far away, before there were pattern catalogs. Remember that patterns are not designed *a priori*, but are extracted from working code. One weekend at that long-ago job, I started writing a text-based adventure game in C. I made up a giant C struct—a data type that foreshadowed Java's class mechanism—and simply assigned a new one whenever the player changed locations. Assigning the struct controlled the outcome of operations such as enter, exit, take, and so on. It had both strings and pointers to functions to handle behaviors; there was even a preprocessor that took a simple text file (this was long before JSON) and generated the C files containing the states. The point here is that the State pattern can be

implemented in C or in most modern procedural languages; it's certainly not limited to C++ and Java.

I've partly re-created this in the file `GameStateDemo.java` (in the same GitHub repository), although it doesn't have the actual room descriptions. The program is just a demo; the game is playable, but the average time to boredom is on the order of 4.2 seconds. Here is the `State` class, which covers actions regarding rooms:

```
abstract class State {
    public abstract void lookAround();
    public abstract void goInside();
    public abstract void goOutside();
    public void quitGame() {
        display("Goodbye!");
        System.exit(0);
    }
}
```

This simple version of `State` does not have an activation method, but it does have `quitGame()`. In a trivial game, it makes sense to allow the user to exit from any state, so I allow that in the default `quitGame()` method. States could override this, too—for example, by prompting users if they're holding any valuables or if they want to save the state of the game.

The states in this game are `inRoom`, `inHallway`, and so on. Here is the `inHallway` state:

```
public State inHallwayState = new State() {
    public void lookAround() {
        display("You are in a hallway. There is a door here");
    }
    public void goInside() {
        display("You are in a room");
        state = inRoomState;
```

```
    }
    public void goOutside() {
        display("You are already in the hallway");
    }
};
```

To see the other states, check out the online codebase.

Conclusion

The State pattern is a good one. It sorts out the state-specific behavior, making it easier to read and maintain than having a long conditional statement in each method. It's also easier to add new states. Finally, it makes the state transitions explicit inside the `Context` class and the States—and they can be completely invisible to the outside. Transitions are also atomic, because only a single variable in the `Context` class, typically with a name like `currentState`, is changed on a transition.

Try the State pattern the next time you find yourself writing parallel `if` statements in several methods, or anytime an object's behavior has to change significantly based on what state it's in. You'll like the improvement in readability and maintainability that it brings. `</article>`

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

SIMON ROBERTS

MIKALAI ZAIKIN

Quiz Yourself

More intermediate and advanced test questions

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

Answer 1
page 79

Question 1 (intermediate). Given the following code:

```
public class Calculator {
    public static void main(String[] args) {
        int i = 0;
        Calculator c = new Calculator();
        System.out.print(i++ + c.operation(i));
        System.out.println(i);
    }

    public int operation(int i) {
        System.out.print(i++);
        return i;
    }
}
```

What is the result? Choose one.

A. 121
B. 123
C. 234
D. 345
E. Three digits, but the exact values are platform- and implementation-dependent

Answer 2
page 81

Question 2 (intermediate). Given this:

```
public class CustomException extends Exception {}
```

And this:

```
public class BadCalculator {
    public static void main(String[] args) {
        try {
            new BadCalculator().divide();
        } catch (Error err) {
            System.out.println("main catch");
        }
    }

    void divide() throws Error {
        try {
            int i = 1 / 0;
        } catch (RuntimeException re) {
            System.out.println("catch");
            throw new CustomException();
        } finally {
            System.out.println("finally");
        }
```

//fix this /

```
    }
}
```

What is the result? Choose one.

A.
```
catch
finally
main catch
```

B.
```
catch
finally
```
followed by an exception stack trace

C.
```
catch
finally
```

D. Compilation fails.

Answer 3
page 83

Question 3 (advanced). Which two of the following statements are true?

A. An anonymous class may specify an abstract class as its base type.

B. An anonymous class may specify an interface as its base type.

C. An anonymous class may specify both an abstract class and an interface as base types.

D. An anonymous class can always be replaced with a lambda expression.

E. An anonymous class requires a zero-argument constructor for its parent type.

Answer 4
page 85

Question 4 (advanced). Given the following code:

```
public class Car {
    int speed;
    public Car () { speed = 90; }
    abstract void accelerate(int deltaSpeed);
}
```

And this code:

```
public class RacingCar extends Car {
```

//fix this /

```
    public RacingCar() { speed = 180; }
    public void accelerate(int deltaSpeed) {
        speed += deltaSpeed;
        System.out.println("The new speed is : " + speed);
    }
}
```

And this code:

```
Car c = new RacingCar(); // line n2
c.accelerate(50);        // line n3
```

What is the result? Choose one.

A. The `Car` class fails to compile.
B. Line n2 fails to compile.
C. Line n3 fails to compile.
D. `The new speed is : 140` is printed.
E. `The new speed is : 230` is printed.

Question 1
page 76

Answer 1. Option A is correct. This question investigates the evaluation order of expressions and how values are passed in method calls.

Java Language Specification section 15.7 tells us, "The Java programming language guarantees that the operands of operators appear to be evaluated in a specific evaluation order, namely, from left to right." Because of this, the result of the calculations must be predictable and option E must be incorrect.

//fix this /

Given that the evaluation order is specified as left to right, you can begin to determine the behavior. In the first `print` call, the argument expression is `i++ + c.operation(i)`.

You know from the left-to-right rule that this code must evaluate `i++` before making the call to `c.operation(i)`. Because the value of `i` prior to this line is zero (due to the unambiguous initialization two lines prior), the value passed into the call to the `operation` method will be 1. Also, the value of `i++` used in the calculation will be zero, because the value of a post-increment expression is the value prior to incrementing the variable.

The first `print` that executes, and therefore the first to produce output, is actually the last one in the order of the codebase. That's the one in the method `operation()`. You've established that the argument value is 1. The `print` statement in the `operation` method prints the value of the expression `i++`, which is another post-increment expression. So, the first digit printed will be 1. This, by itself, means that options C and D must be incorrect.

Next, `c.operation` returns the value 2 to its caller. This is because the value of `i` in the call has been incremented by this point. So, the value 2 is added to the original value of `i` in the `main` method. That value was zero, and remember that with `i++`, you're still using the zero value, not the incremented value. Therefore, the next digit printed will be 2. Unfortunately, this doesn't eliminate any other options, so you have to proceed to the final `println` to decide what the correct answer is.

The third digit is output by the `println` that is the last statement in the `main` method. That will show the value of `i` after the `i++` operation, which is 1. The call to `c.operation` did not affect that value, because Java uses a call-by-value method invocation model. Specifically, this means that `i` inside the call to `c.operation` is a *different* variable from the one of the same name in the caller, and the increment that happened inside the `operation` method does not affect the value with which you currently are concerned.

So, the final value printed is 1 again, because that is the result of incrementing zero in the `i++` behavior on the left side of `i++ + c.operation(i)`.

As a result, you should see that the total output is 121. That's sufficient to identify that option B is incorrect, and the correct answer is option A.

Question 2
page 77

Answer 2. Option D is correct. Typically, the exam creators try to avoid creating questions that use "compilation fails" in the unqualified manner used in option D. It's considered too open-ended to expect a candidate to be absolutely sure that not a single semicolon has been omitted. (Indeed, there's a very small, but nonzero, chance that such a trivial syntax error could creep into a question during production, and then "compilation fails" would be unintentionally correct.) However, even though an answer such as option D would likely identify a particular line in the real exam, we are willing to cheat a little in the interest of creating this question for the purpose of mental exercise. In general, though, "compilation failure" remains a legitimate basis for a question.

Here, the question is investigating Java's rules about checked exceptions and the distinctions therein relating to runtime exceptions, regular checked exceptions, and errors.

The background to this is that for years, programmers got themselves into trouble by writing code that dealt with only the "happy path" and ignored things that might fail. Later, their customers would complain that the code crashed in strange ways. Investigation would often show that something environmental (such as a file not being found or a network cable being unplugged) had gone wrong, and the programmers hadn't thought about what to do in that situation.

For years, programmers got themselves into trouble by writing code that dealt with only the "happy path" and ignored things that might fail.

Java's "declare or handle" rule makes it rather harder to sweep such unpleasant realities under the carpet, and it tends to push a programmer to do something about the error path when the code is first created. We've greatly oversimplified the philosophy here, particularly because we've said nothing about how to create *good* designs using exceptions (and checked exceptions do nothing to improve poor designs), but the checked exception rule has always seemed like a really good idea to us.

Philosophy and background aside, the essence of Java's "declare or handle" rule is that if any exception might arise that is neither a `RuntimeException` nor an `Error` (or a subclass of those,

of course), the source code must do something about the problem. The options available in the "do something" category are to catch the exception or to declare that the method throws the exception. This suggests that the `catch` block that prints the message `catch` would be invoked to handle the problem of the `ArithmeticException` that will arise from the division by zero (because `ArithmeticException` is a `RuntimeException`).

But here is where the trouble starts. That `catch` block then attempts to throw a `CustomException`. Given that there is no related `catch` block for that `CustomException` (it's not thrown in a `try` block, anyway), the `CustomException` will propagate out of the method. `CustomException` is a subclass of `Exception`, not a subclass of `Error`. These two classes are siblings in the class hierarchy, as shown in **Figure 1**.

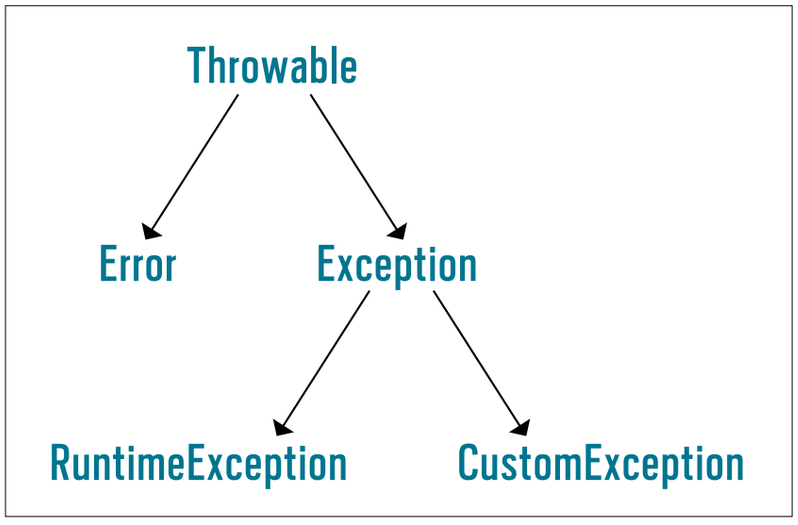


Figure 1: Class hierarchy

The `divide` method declares that it throws `Error`, not `CustomException`, and because `CustomException` is a checked exception, the "declare or handle" rule has been broken and the code does not compile. Given this, the correct answer is option D, and options A, B, and C are all incorrect, because if the code doesn't compile, it surely doesn't generate any output.

To cure the problem, a likely approach would be to change the two occurrences of `Error` (one in the `throw` clause of the `divide` method and the other in the `catch` block of the `main` method) to `CustomException`. The code in its current form never actually throws an `Error`, and `Error` is not a

checked exception anyway, so it need not be declared or handled. If these changes were made, the output would match option A:

```
catch
finally
main catch
```

Question 3
page 78

Answer 3. The correct answer is option A and option B. An anonymous class is one for which the programmer does not specify a name. Instead, the context is such that it's enough to say "an object that has this parentage, with these particular features" (where "particular features" are usually, but not exclusively, overriding/implementing methods).

Because the class does not have a name of its own, the approach is similar to asking a builder to "make this" while handing over a stack of plans. There's no need to specify the name of the thing to be built; it's to be built "according to these specifications." Of course, this means that the declaration and instantiation must be coincident; you clearly cannot refer to this nameless class by any means other than the code of its class specification.

Anonymous classes have existed since Java 1.1, and they were the first syntactic support for simplifying object creation in a way that allows focusing the code on what's deemed "important"—generally *what* the object does—rather than what might be called "syntactic scaffolding." In this case, the important aspects are the parent class or interface, the generalization that describes it, and the unique methods of the implementation.

Lambda expressions have another restriction not applicable to anonymous classes: an anonymous class can override (or implement) many methods, but a lambda expression provides behavior for exactly one method.

Another consequence of using an anonymous class is that it reduces clutter in the class namespace and, consequently, in the documentation space. In a way, this takes the view "who

cares what it's called; it just does exactly what it says on the can!"

The syntax of an anonymous class can create an object that is derived from a class (abstract or concrete) or that implements an interface. Because of this trait, both option A and option B are correct.

Regardless of whether a class is being extended or an interface is being implemented, if the parent is called `BaseType`, the syntax looks like this:

```
BaseType anon = new BaseType() {
  // implementation specific code here, usually overriding
  // method(s)
}
```

This syntax, unlike the more general `implements` clause that may be used when declaring a regular class, permits only a *single* base type to be specified. Consequently, option C is incorrect.

If the anonymous class is to serve any useful purpose, some code must be provided in the body. If `BaseType` is an interface, it presumably declares an abstract method (or methods) that must be implemented. If `BaseType` is a class, you must modify some behavior; otherwise, you might as well have simply instantiated `BaseType` directly.

In Java, a constructor is defined using the name of the class that it initializes. If no such name exists, then clearly no explicit constructor is possible. Despite this, however, the anonymous form allows parameters to the construction (for example, `new BaseType(1,2)`), provided that those arguments match an *existing* constructor in `BaseType` (and, of course, `BaseType` must be a class because interfaces do not have constructors).

We mentioned earlier that the anonymous class form can build an object based on a class—either concrete or abstract—or an interface. A lambda expression, however, can be created only to implement an interface. Therefore, there are things that anonymous classes can do that lambdas cannot do, which tells you that option D is incorrect.

In fact, lambda expressions have another restriction not applicable to anonymous classes: an anonymous class can override (or implement) many methods, but a lambda expression pro-

vides behavior for exactly one method. Further, that method must be one abstract method of the interface that is being implemented. As a result, it's clear that anonymous classes still have a purpose in the language, even with the advent of the lambda syntax.

In the sample syntax, the construction is performed using zero arguments. That mandates that the parent class—`BaseClass` in the sample—must have a zero-argument constructor (it could also have other constructors). However, in the more general case, arguments are permitted, provided that they match an available constructor in the type being extended. Consequently, option E is incorrect.

Question 4
page 78

Answer 4. Option A is correct. This question investigates abstract classes and methods. A class may be declared abstract by adding the keyword `abstract` before the keyword `class`, like this:

```
public abstract class MyClass { ...
```

Java Language Specification section 8.1.1.1 describes abstract classes. There are two key consequences when a class is declared to be abstract. First, creating an instance of that class is prohibited. Subclasses of it may be defined and, provided those classes are not abstract, those subclasses can be instantiated. However, you can never have an instance of an abstract class itself.

The second thing that changes is that an abstract class is permitted to declare abstract methods. Concrete classes (nonabstract ones, if you like) are prohibited from doing this. And that pinpoints the problem in this question. The `Car` class is *not* abstract and, therefore, it is prohibited from containing an abstract method. As a result, option A is correct and options B, C, D, and E are incorrect.

If the `Car` class had been declared abstract, how would things have been different? Immediately, the `Car` class would compile successfully. At that point, you have new considerations. First, is it permitted to create an instance of `RacingCar`? Yes; absolutely it is. `RacingCar` at that point would be a correct and complete concrete class. A concrete class is not permitted to have any abstract methods, but the `accelerate` method that's defined in `RacingCar` is a correct imple-

mentation of the abstract method of the same signature declared in `Car`. The implementation is public, where the abstract method has default access, but that's OK. An overriding or implementing method must not be less accessible than the method it overrides, but it is fine for it to be more accessible. In other words, there are zero abstract methods in `RacingCar`, so line n2 would not cause any problems.

You cannot instantiate an abstract class, but you can use an abstract class or interface as a variable type, and you can assign objects that are built from subclasses or implementations to that variable.

The use of the abstract base class `Car` as the type of the variable `c` in line n3 would also be completely correct. You cannot instantiate an abstract class, but you can use an abstract class or interface as a variable type, and you can assign objects that are built from subclasses or implementations to that variable. This kind of generalization is one of the reasons that abstract classes exist: to allow you to write code in terms of generalizations (the abstract class or interface) that will work correctly with any specialization (that is, with an object created from a concrete subclass or implementation). At this point, you can be confident that the code would not have compilation issues at either line n2 or line n3.

Therefore, if the code with the abstract version of `Car` would have compiled correctly, how would it have behaved? To answer this, you need to consider how the construction process works, and that aspect is fairly simple. The constructors run from the top of the class hierarchy (`Object`) downward. This means that the `RacingCar` constructor would run after the `Car` constructor. Because of that, the initial value of speed would be 180 rather than 90. (These speeds are in kilometers per hour, presumably.) If the initial speed is 180, the speed printed after the `accelerate` method is called would be 230.

There's a potentially interesting side discussion about how exam questions are usually phrased. As a rule, questions are intended to be specific about the line containing the problem that causes compilation to fail. Recently written questions generally don't suggest a cause.

In this example, it's not abundantly clear whether the "problem" is that there's an abstract method declared in a concrete class or that a nonabstract class surrounds an abstract method. Of course, the perspective you take on this determines where you think the error is located: either in `public class Car` or in the declaration of the abstract `accelerate` method.

In its current form, the compiler actually reports the problem in terms of the class being in error because it fails to provide concrete implementations for all its methods. However, for this question, it seemed easier to simply offer the suggestion that the class doesn't compile. Such a form isn't very likely in the real exam, so be aware that if two separate elements of syntax are mutually exclusive, you should be ready to consider either of them as the cause of the problem. `</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.

JEP 335: Deprecate the Nashorn JavaScript Engine

This proposal, made in May 2018, puts forth the idea of deprecating the engine added in Java 8 that provides application users a way to script inside Java apps. In this way, Nashorn (pronounced "nas-horn") provides functionality similar to what Lua provides to C programs.

You might recall that Nashorn was added to Java 8 as a replacement for its slower predecessor, the Rhino engine. Both products are implementations of the Java scripting interface, which defines a mechanism that enables scripting and specifies how data is moved between the engine and the host Java application. Although Nashorn and Rhino support JavaScript, scripting engines exist for other languages, such as Groovy and Ruby.

JEP 335 does not propose deprecating the scripting engine technology, but rather just the JavaScript implementation. The primary driver is that JavaScript itself is evolving, and to provide full support for the language requires considerable developer resources for apps that might not need the new features. To wit, Nashorn currently supports the ECMAScript version 5.1. However, ECMAScript 2016 and 2017 await implementation.

JEP 335 currently is only a proposal. According to the authors, part of the rationale for this proposal is to see whether developers outside of Oracle might be interested in working on Nashorn. In any event, all code that presently works with Nashorn would continue to do so for the foreseeable future. More on the status of JEP 335 will be posted as comments are received.

//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