Java™ magazine

By and for the Java community

MARCH/APRIL 2019 EXPANDED ISSUE

Lightweight Frameworks

Fast services without all the baggage

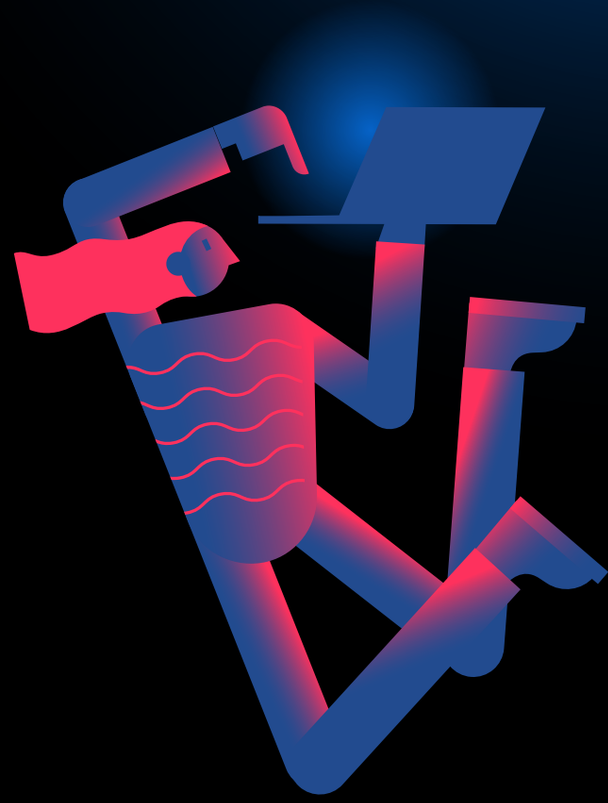
ORACLE.COM/JAVAMAGAZINE

ORACLE®

IntelliJ IDEA

Capable and Ergonomic IDE for JVM

Indepth coding assistance
Cross-language refactorings
Clever error analysis
and much more.



Download at jetbrains.com/idea

THE DRIVE
TO DEVELOP

JET
BRAINS

//table of contents/

COVER FEATURES

OTHER FEATURES

DEPARTMENTS

COVER ART BY WES ROWELL

EDITORIAL

Editor in Chief
Andrew Binstock

Senior Managing Editor
Leslie Steere

Copy Editors
Lea Anne Bantsari, Karen Perkins

Contributing Editors
Deirdre Blake, 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.

Size Still Matters

Despite servers with terabytes of RAM, executable size still matters.

We now live in an age in which servers can host more than 1 TB of RAM and where so-called spindle farms are populated with hard drives weighing in at 12 TB or more of capacity. For the average consumer, smartphones with 128 GB of storage are no longer uncommon. You would think with all this space (at amazingly low price points) that we could finally let go of concerns about the size of executable programs. But we can't. While the capacity has grown, the problem of executable size remains frustratingly constant.

Take, for example, the theme of this issue: lightweight frameworks. The three frameworks we cover (Javalin, Micronaut, and Helidon) all tout their small size, which is presented as a proxy for speed and simplicity of programming. But in fact, the main benefit of the small size is to fit within the architectural metaphor of microservices. Those microservices are universally deployed in containers, which themselves were created in response to the considerable size of virtual machines.

It's important to note here that the argument for microservices is not the usual one: less complexity. Microservices bring many benefits, but it is far too early—in my opinion and that of many others—to know whether in fact they deliver an equivalent computing experience with less complexity. If you ever have tried to locate an intermittent bug in a microservices implementation, you have firsthand knowledge of the complexity problem.

But returning to the question of size. The size of executables has been an issue for programmers since the beginning of business computing. Memory arrays for IBM computers in the early 1960s cost $1 a byte (which would be $8.57 a byte in today's dollars). Because of this cost, most

PHOTOGRAPH BY BOB ADLER/GETTY IMAGES

development was done in assembly language and programmers had to be exceedingly knowledgeable about the effect of their code on the binary size. They worked much like embedded programmers—equally skilled at coding and shoehorning.

Two decades later, the problem still existed but without such tight strictures. For example, UNIX workstations for much of the 1980s and early 1990s were considered top of the line if they had 16 MB of RAM. And software that used a lot of that memory was heavily criticized because it greatly slowed other operations. For example, emacs, the editor upon which millions of developers depended, was widely mocked for its memory consumption as being an acronym for "eight megabytes and continually spooling." The irony is that today, no editor is small enough to use only 8 MB of memory. In those days, memory was a hard limit—if you blew the limit, performance crawled, or on some systems, like the early Macs, the system would simply hang.

Today, with hundreds of gigabytes of RAM on high-end workstations and servers, the limits are soft. This is especially true in the cloud, where servers of more than 1 TB are commonly available. Executable size still matters, though, because in a microservice architecture, you could have hundreds of instances of services running in the same memory space. In such a situation, you want those instances to be small so that the remaining RAM can be dedicated to the data itself. And if that data's needs decline, you want to be able to shrink your memory usage to lower your cloud infrastructure bill. So for these soft reasons, economy of resources and of spending, executable size remains an important aspect.

What is not entirely clear is whether microservices do in fact lower memory usage. Containers hold duplicated code—both user and system code. Run enough containers and you are likely to surpass the execution footprint of the monolithic server that those services aim to replace. However, services deliver a greater scalability that servers cannot match, and this benefit alone might offset the potential for excess memory consumption.

But regardless of whether your architecture reaches that tipping point, you'll always be well served by true economy in executable size.

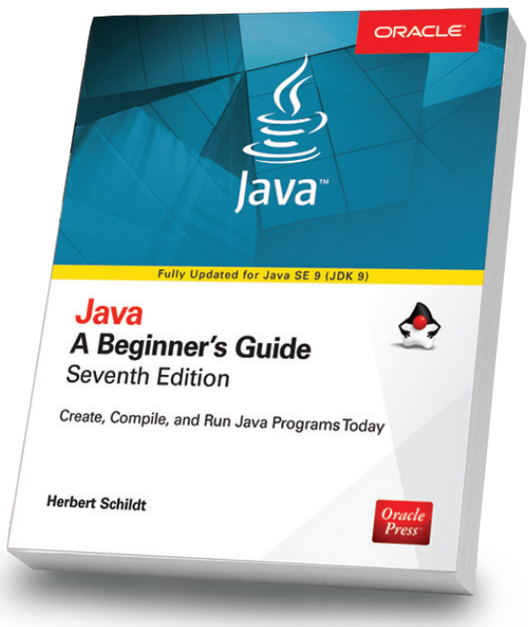
Andrew Binstock, Editor in Chief
javamag_us@oracle.com
@platypusguy

Note: We did not publish a January/February issue due to being greatly tied up with a project we have wanted to deliver for a long time: a full website in which *Java Magazine* articles are presented in responsive HTML. We expect to roll that out to you around midyear and are confident it will greatly improve your experience.

Oracle Press™

Your Destination for Oracle and Java Expertise

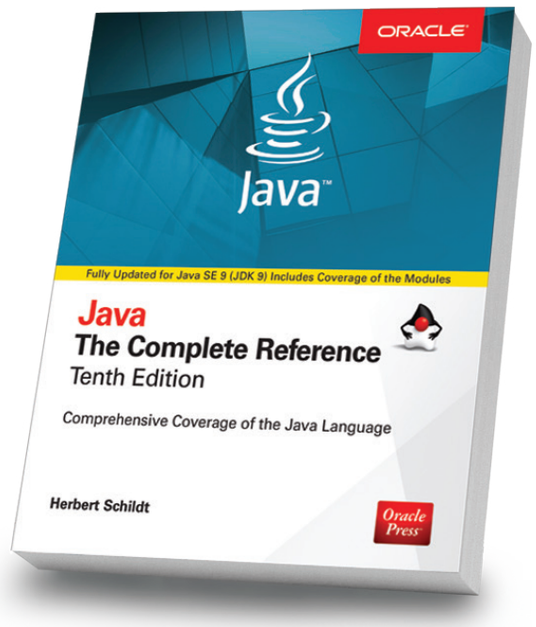
Written by leading experts in Java, Oracle Press books offer the most definitive, complete, and up-to-date coverage of Java available.



Java: A Beginner's Guide, 7th Edition

Herb Schildt

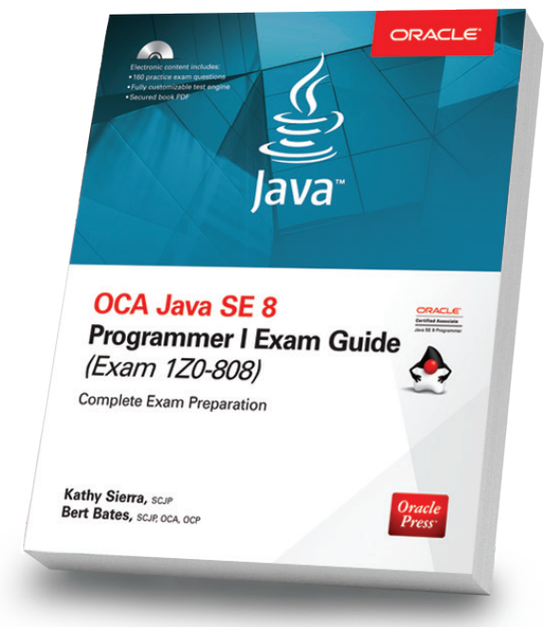
Revised to cover Java SE 9, this book gets you started programming in Java right away. Free online supplement covering key new features in JDK 10 available for download on the book's page on OraclePressBooks.com



Java: The Complete Reference, 10th Edition

Herb Schildt

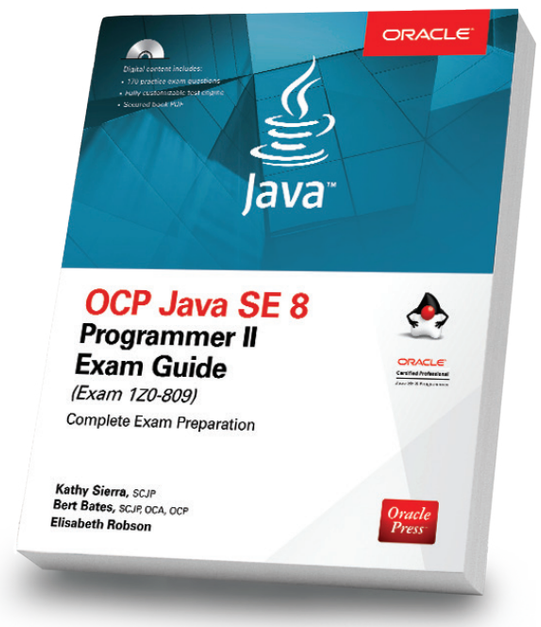
Updated for Java SE 9, this book shows how to develop, compile, debug, and run Java programs. Visit the book's page on OraclePressBooks.com to download free supplements on JDK's key new features.



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)

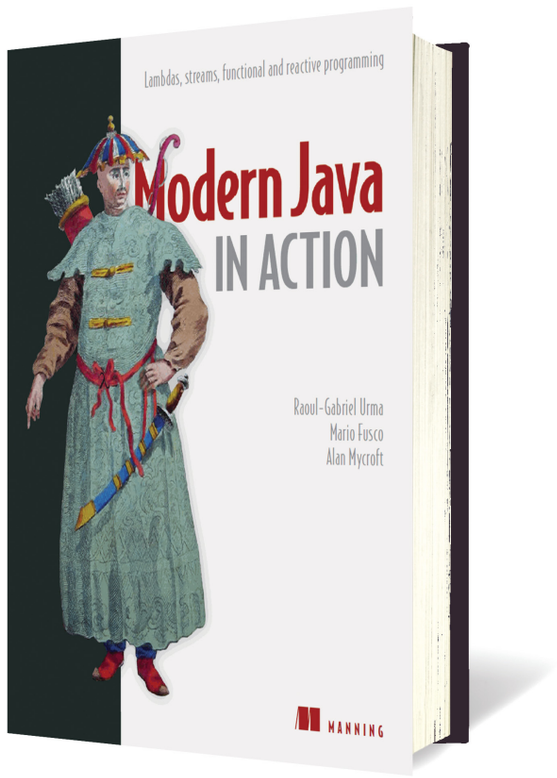
Kathy Sierra, Bert Bates, Elisabeth Robson

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



MODERN JAVA IN ACTION

By Raoul-Gabriel Urma, Mario Fusco, and Alan Mycroft

The new six-month cadence for Java releases has become a real quandary for the book industry. It takes more than six months to update a book, so if you're originally targeting Java 11 in your new volume, by the time it comes out Java 12 will already be released—with Java 13 imminent. This book finds an innovative way to cover all the versions since Java 8 by referring to them as "Modern Java." (The term *modern* was also used by the C++ community several years ago to refer to releases of C++ that enabled strong typing throughout. But use of the term there was political—an indication that you should abandon the former non-modern ways of coding. Such is not the case here.)

This book is the sequel to the authors' previous work, the well-regarded *Java 8 in Action*. Whereas that volume introduced lambdas and how they might be used for functional Java, this book more deeply embraces functional Java and shows how to use this flow style of programming beneficially. As you'd expect, the authors pay thorough attention—across more than 150 pages—to the effective use of lambdas and streams together, including coverage of refactoring, testing, and debugging lambdas as well as a systematic review of most of the major design patterns in which the implementations are performed using lambdas.

Any good work on functional programming in Java needs to treat reactive programming as well. *Modern Java in Action* is the first major book on Java to understand this. It digs into reactive development across 130 pages that are filled with numerous demonstrations of how to implement your own reactive programming with `CompletableFuture`—as well as how to do the same with the RxJava library. This functional/reactive approach is the core of the volume, although other post-Java 8 topics are covered as well: modules, improvements to the date and time library, using Optionals, and so on.

The authors are well-known experts (Urma has cowritten several articles in *Java Magazine*), and their style is engaging and lucid. Their illustrations are deep and avoid snippet-size examples by exploring larger projects, such as writing a DSL or creating an asynchronous API and pipelining its actions.

This book bears the same cover as *Java 8 in Action*, which underscores that it is a second edition of that volume. In my estimation, the new material does indeed warrant the cost of the upgrade: the sections on reactive programming alone justify the price. For readers not familiar with Urma, Fusco, and Mycroft's earlier volume, it is one of the best books for getting a firm grip on the cognitive load required by the new features in Java 8 and subsequent releases.

—Andrew Binstock

//events/

DevNexus

MARCH 6–8
ATLANTA, GEORGIA
DevNexus is an international open source developer conference. Its stated goal is to connect developers from all over the world, provide affordable education, and promote open source values. Session topics include building evolutionary architectures; design patterns; becoming the first Java 11 certified developer; hands-on cloud native Java with MicroProfile, Kubernetes, and Istio; Java security; and a Spring 2.x workshop.

PHOTOGRAPH BY MATT KIEFFER / CC BY-ND 2.0

QCon London

MARCH 4–6, CONFERENCE
MARCH 7–8, WORKSHOPS
LONDON, ENGLAND
QCon is designed for technical team leads, architects, engineering directors, and project managers who influence innovation in their teams. Topics include evolution of Java and the JVM, JavaScript frameworks, operationalizing microservices, advances in FinTech security, and AI and machine learning methods.

Twin Cities Software Symposium

MARCH 8–10
MINNEAPOLIS, MINNESOTA
Topics covered at this developer conference include microservices migration patterns, the evolution of Java, migrating to Java modules, and more. Speakers include author Venkat Subramaniam, author and Spring expert Craig Walls, and author and software architect Mark Richards.

ConFoo Montreal

MARCH 13–15
MONTREAL, CANADA
This multi-technology conference for web developers promises 155 presentations and typically features targeted sessions for Java and JVM developers on topics such as CI/CD, Spring, and cloud native development.

Oracle Code Bengaluru

MARCH 15
BENGALURU, INDIA
Oracle Code is a free event aimed at helping developers explore the latest developer technologies, practices, and trends and includes educational sessions for developing software using technologies such as containers, microservices, machine learning, intelligent bots, and blockchain.

JavaLand 2019

MARCH 19–21
BRÜHL, GERMANY
This annual conference features sessions on subjects including core Java and JVM languages, microservices architecture, front-end development, and much more.

Voxxed Days Zürich

MARCH 19
ZÜRICH, SWITZERLAND
Voxxed Days Zürich shares the Devoxx philosophy that content comes first; it draws internationally renowned and local speakers discussing topics such as cloud development, containers, machine

learning, and programming languages. Scheduled speakers include Venkat Subramaniam, Lukas Eder, and Anton Arhipov.

Voxxed Days Bucharest

MARCH 20–22
BUCHAREST, ROMANIA
This developer conference brings together in-demand speakers and practitioners of popular open source technologies and features insights into topics such as Java, infrastructure, real-world architectures, analytics, the modern web, and programming languages.

O'Reilly Strata Data Conference

MARCH 25–26, TRAINING AND TUTORIALS
MARCH 27–28, KEYNOTES AND SESSIONS
SAN FRANCISCO, CALIFORNIA
Data science, machine learning, and data engineering are the core focus of Strata. Training courses include hands-on data science with Python, machine learning in TensorFlow, and professional Kafka development.

Desert Southwest Software Symposium

MARCH 29–30
PHOENIX, ARIZONA
This No Fluff Just Stuff–hosted conference will focus on the latest technologies and best practices emerging in the Java and JVM software development space. Speakers are authors, consultants, open source developers, and recognized industry experts.

Voxxed Days Milan

APRIL 13
MILAN, ITALY
This one-day developer conference will cover topics such as cloud, big data, AI, robotics, Java, security, and architecture. A keynote will be delivered by Java Champion Holly Cummins, who leads IBM's Open Liberty project.

O'Reilly Artificial Intelligence Conference

APRIL 15–16, TRAINING
APRIL 16–18, KEYNOTES, SESSIONS, AND TUTORIALS
NEW YORK, NEW YORK
This conference is devoted to the latest developments and techniques in AI, including specialized hardware for sensing, model training, and model inference; cloud and on-premises tools for building AI applications online and on the edge (including mobile); new architectures and pipelines; proven best practices and detailed case studies; and ethics and security guidelines.

Devoxx France

APRIL 17–19
PARIS, FRANCE
Devoxx France will take place at the Palais des Congrès this year, with an estimated 3,000 participants and 235 presentations and hands-on labs focused on topics including Java, alternative JVM languages, architecture, IoT, and cloud computing.

JAX 2019

MAY 6–10, CONFERENCE
MAY 7–9, EXPO
MAINZ, GERMANY
This year, JAX focuses especially on Java Enterprise technologies, the Spring ecosystem, JavaScript, continuous delivery, and DevOps. More than 200 internationally renowned speakers will give practical and performance-oriented lectures.

Devoxx UK

MAY 8–10
LONDON, ENGLAND
Devoxx UK invites developers and architects to come together and explore the latest technology advancements. The conference offers more than 120 sessions covering a range of topics, including Java, cloud, big data,

PHOTOGRAPH BY MATT KIEFFER / CC BY-ND 2.0

AI, robotics, interesting programming languages, security, architecture, methodologies, and developer culture.

jPrime
MAY 28–29
SOFIA, BULGARIA
jPrime is a conference with talks on Java, various languages on the JVM, mobile, web, and best practices. Its fourth edition, run by the Bulgarian Java User Group, will be held in the Sofia Event Center and is backed by the biggest companies in the city. The conference features a combination of international speakers along with presenters from Bulgaria and the Balkans.

O'Reilly Velocity Conference
JUNE 10–11, TRAINING
JUNE 11–13, TUTORIALS, KEYNOTES, AND SESSIONS
SAN JOSE, CALIFORNIA
New topics at the 11th annual Velocity conference for web and systems engineers include cloud, cloud native, and infrastructure as well as machine learning, AI, and blockchain. There also will be sessions devoted to serverless computing, containers, Kubernetes, microservices, DevOps, and security.

ÜberConf
JULY 16–19
DENVER, COLORADO
This conference for software developers and architects will cover Java 12, Docker, cloud native architecture, reactive programming, JVM internals, Apache Spark, distributed systems, Gradle, machine learning, and more.

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

TRANSYLVANIA JUG

The Transylvania Java User Group (TJUG) was founded in Cluj Napoca, Romania, in May 2008 at the initiative of Gabriel "Gabi" Pop. Over the past 10 years, the community has grown to more than 1,000 members and has organized more than 60 events. The audience and the number of members present at the events has steadily increased, bringing the average current number of attendees for most of these gatherings to more than 200 people. Recent meetings have hosted presentations by Java luminaries such as Adam Bien, Venkat Subramaniam, Peter Lawrey, Raoul-Gabriel Urma, and Richard Warburton—all of whom have written for *Java Magazine*. The TJUG has also hosted Romanian speakers, including Vlad Mihalcea and Victor Rentea, whose presentations enjoyed considerable popularity. In addition, some members of the community worked on a Java Advent Calendar (one article for every day of Advent) dedicated to the JVM ecosystem.

The TJUG is exploring ways to diversify its existing roster of events to grow the community and the expertise of the individuals within it. New event types currently under consideration include code katas, blitz talks every semester, and workshops. In addition to all these activities, as of last year, the TJUG became a community partner of Voxxed Days Cluj-Napoca, the Voxxed conference in Romania.

To participate in the TJUG activities, check the upcoming events on Meetup.

DEVELOPER EVENTS FROM THE DEVOXX & VOXXED FAMILY
COMING IN 2019

DEVOXX™

DEVOXX.COM

FRANCE 17-19 APRIL

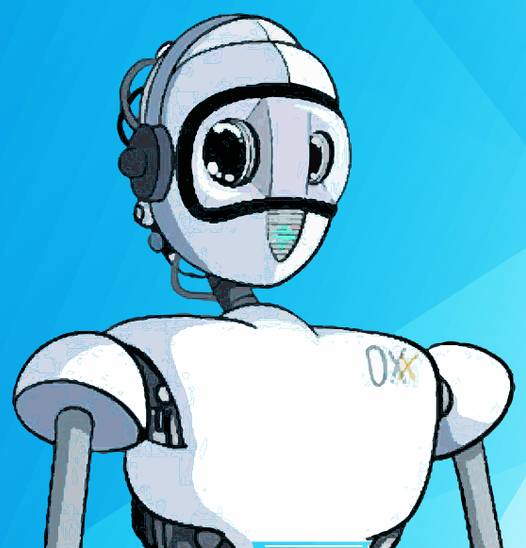
UK 8-10 MAY

POLAND 24-27 JUNE

UKRAINE 1-2 NOVEMBER

BELGIUM 4-8 NOVEMBER

MOROCCO 12-14 NOVEMBER



ZURICH 19 MARCH

BUCHAREST 20-22 MARCH

MILAN 13 APRIL

CERN 1 MAY

MELBOURNE 13-14 MAY

SYDNEY 16-17 MAY

FRONTEND, BUCHAREST 21-22 MAY

MINSK 24-25 MAY

SINGAPORE 30-31 MAY

ATHENS 7-8 JUNE

LUXEMBOURG 20-21 JUNE

BANFF TBC

TICINO TBC

MICROSERVICES, PARIS TBC

CLUJ NAPOCA TBC

VOXXEDDAYS

VOXXEDDAYS.COM

Running Fast and Light Without All the Baggage

The emergence of microservices as the new architecture for applications has led to a fundamental change in the way we use frameworks. Previously, frameworks offered an omnibus scaffolding that handled most needs of monolithic applications. But as microservices have gained traction, applications now consist of orchestrated containers, each performing a single service. As such, those services require far less scaffolding—favoring instead lightweight frameworks that provide basic connectivity and then get out of the way.

In this issue, we examine three leading frameworks for microservices: Javalin (page 13), which is a very lightweight, unopinionated Kotlin-based web framework; Micronaut (page 23), which handles all feature injection at *compile* time and so loads extremely fast; and Helidon (page 34), which is a cloud native framework that generates a pure Java SE JAR file that can be run as a service or a complete app. Helidon comes in two flavors: a minimal framework and a slightly heftier one for developers wanting additional services.

In addition to these articles, we continue with the final installment of our series on Java design patterns—this time covering the Proxy pattern (page 53), with practical examples and coverage of the rarely discussed dynamic proxy feature in a little-used corner of the Java language.

Ben Evans examines a common optimization in VMs, loop unrolling (page 62), and explains the subtle reason why loops on the JVM will execute more slowly if they're indexed by `long`s rather than `int`s.

And of course we have our quiz—somewhat expanded for this issue (page 81)—and our book review (page 7).

ART BY WES ROWELL

DAVID ÅSE

Javalin: A Simple, Modern Web Server Framework

Building web apps with a fast, lightweight, unopinionated framework that creates tiny executables

Javalin is a very lightweight web framework for Java 8 (and later) and Kotlin. It supports modern features such as HTTP/2, WebSocket, and asynchronous requests. Javalin is servlet-based, and its main goals are simplicity, a great developer experience, and first-class interoperability between Java and Kotlin.

In this article, I explain what Javalin is and how easily it enables you to write web applications quickly. You'll need some experience with the basics of web applications to follow along.

Many developers would say Javalin is a library rather than a framework. This is because in Javalin, unlike in most frameworks, you never extend anything; it sets no requirements for your application structure; and there are no annotations, no reflection, and no other magic—just code. The "Hello World" example is just four lines and an `import` statement:

```
import io.javalin.Javalin;

public static void main(String[] args) {
    Javalin app = Javalin.create().start(7000);
    app.get("/", ctx -> ctx.result("Hello World"));
}
```

This snippet creates a new Javalin instance and starts it on port 7000. It then attaches a `Handler` that is triggered by GET requests to the root path. You can build and package this application as a JAR file. If you use Maven, just add this to your build:

```
<dependency>
    <groupId>io.javalin</groupId>
    <artifactId>javalin</artifactId>
    <version>2.5.0</version>
</dependency>
```

Run the output JAR file like any other Java program (`java -jar myjar.jar`).

Getting Started

All Javalin programs require the creation of a Javalin instance (`Javalin.create()`), which creates a web server to which you can attach `Handler` objects. The `Handler` interface has a single method, `handle`, which is void and takes a `Context` as its only parameter. This `Context` contains everything you need for operating on the HTTP request and response.

```
@FunctionalInterface
public interface Handler {
    void handle(Context ctx) throws Exception;
}
```

A `Handler` is attached to the `Javalin` instance with a verb and a path:

```
app.get("/hello-get", ctx -> ctx.result("Hello GET"));
```

Responses are set on the `Context` instance (`ctx`) through the `ctx.result()` method. As mentioned previously, the `Context` contains all the methods required to deal with both requests and responses.

You could also write the previous code snippet by creating a class that implements `Handler`:

```
class MyGetHandler implements Handler {
    @Override
```

```
    public void handle(Context ctx) {
        ctx.result("Hello GET")
    }
}
```

Then, you need to add an instance of `MyGetHandler` to the `Javalin` instance:

```
app.get("/hello-get", new MyGetHandler());
```

Although it's possible to write Javalin applications this way, it's recommended that you use lambda syntax instead. If you need to split up your code, the best approach is to create method references:

```
app.get("/hello-get", helloController::myGetHandler);
```

This approach makes it easier to group common functionality and puts fewer restrictions on how you build your application. I'll present more information on handlers later in this article. Now let's look at how to handle common operations with Javalin.

JSON responses. A common use case for Javalin is to serve a JSON object. This can be done easily by calling `ctx.json(myObject)`:

```
app.get("/json", ctx -> ctx.json(myObject));
```

This code transforms `myObject` to JSON by using Javalin's JSON plugin and sets the content type of the response to `application/json`. The JSON plugin is fully configurable, so any JSON library can be used with Javalin—be it Jackson, Gson, or another choice.

Note that Javalin defines two interfaces for mapping to and from JSON and includes a Jackson implementation for both of these interfaces. But you're free to provide your own implementation to replace the ones that ship with Javalin.

Handling input. All client input is available through the `Context`. You get parameters from the path, the query string, or the request body. The request body can contain either the form parameters or a string (usually JSON). Javalin handles all cases in a consistent way:

```
app.get("/:path-param", ctx -> {
    String qp = ctx.queryParam("query-param");
    String pp = ctx.pathParam("path-param");
    String fp = ctx.formParam("form-param");
    String body = ctx.body();
    MyObject mo = ctx.bodyAsClass(MyObject.class);
});
```

Getting input as a string is great for quick prototyping and debugging, but usually you'll want a specific type of object. For that, you can use the `Validator` class:

```
int index = ctx.validatedQueryParam("index").asInt().getOrThrow();
```

In this code, the call to `getOrThrow()` tells Javalin to convert the string to the specified type or throw an exception. These exceptions are automatically mapped to standard HTTP responses, and they provide helpful debug messages to the client. For example, if the `index` query-param is `abc`, the client will be sent the following error:

```
Query parameter 'number' with value 'abc' is not a valid int
```

The `Validator` class also supports an `asClass` method that you can use to validate any type. This enables you to do powerful things, such as validating two `Instant` types relative to each other:

```
Instant fromDate = ctx.validatedQueryParam("from")
    .asClass(Instant.class)
    .getOrThrow();
```

```
Instant toDate = ctx.validatedQueryParam("to")
    .asClass(Instant.class)
    .check(to -> to.isAfter(fromDate), "'to' has to be after 'from'")
    .getOrThrow();
```

If you were to use `asClass(UnfamiliarType.class)`, Javalin will ask you to register a converter for that particular class.

Filters and mappers. Sometimes you need to apply the same logic for multiple endpoints, or you need to handle errors in a consistent way. These kinds of problems are solved in Javalin by filters and mappers. Just like HTTP endpoints, the filters in Javalin use the `Handler` interface. Filters can be attached to the Javalin instance with or without specifying a path. For example:

```
app.before("/some-path", ctx -> {
    // runs before requests to /some-path
});

app.after(ctx -> {
    // runs after all requests
});
```

The `before` filters run before endpoint handlers. If you want to prevent an endpoint handler from doing something in certain cases, you can throw an exception in a `before` filter. The `after` filter runs after the endpoint handlers (even after exceptions have been handled).

Exception mappers. It's common to throw exceptions when writing controllers for web applications. If a resource is not found, or if a user isn't authorized to view a resource, you throw an exception and handle it elsewhere. Javalin has an exception mapper that lets you map any exception, and it has a set of premapped exceptions for your convenience, such as `BadRequestResponse`, `NotFoundReponse`, and `UnauthorizedResponse`. Like HTTP endpoint handlers and filters, the exception mapper has access to the `Context`:

```
app.exception(NullPointerException.class, (exception, ctx) -> {
    // handle null pointers here
});
```

WebSocket. Javalin offers a high-level lambda-based WebSocket API:

```
app.ws("/websocket/:path", ws -> {
    ws.onConnect(session -> System.out.println("Connected"));
});
```

The `ws` object is a `WsHandler` and supports the most common WebSocket events:

```
onConnect(WsSession session)
onMessage(WsSession session, String msg)
onMessage(WsSession session, Byte[] msg, int offset, int length)
onClose(WsSession session, int statusCode, String reason)
onError(WsSession session, Throwable throwable)
```

The `WsSession` object contains methods for getting path-params and query-params, as well as methods for sending data to the client.

Configuring the server. Javalin doesn't require an application to run, because it runs on top of an embedded Jetty server. Javalin provides several helpful configuration options, all of which are programmatic (there are no configuration files). The following snippet shows some of the options:

```
Javalin.create()
    .contextPath("/context-path")
    .enableAutogeneratedEtags()
    .enableCorsForOrigin("*")
    .enableDebugLogging()
    .enableStaticFiles("/public")
    .start();
```

If you need more control than what Javalin exposes through its configuration API, you can supply Javalin with your own Jetty `Server` object:

```
app.server(() -> {
    Server server = new Server(); //org.eclipse.jetty.server.Server
    // configure server
    return server;
});
```

You can use this option if you want to run Javalin on an HTTP/2 server. (The code required to set up Jetty to run with HTTP/2 is too long to include in this article, but there is a working example on GitHub.)

The process is similar for configuring a Jetty `SessionHandler`, and an extensive tutorial is available on the Javalin website.

Advanced Concepts

Handler groups. When you build a larger application, you often end up with routes that share the same path. For example, consider a standard CRUD API for users:

```
app.get("/users/", UserController::getAll)
app.post("/users/", UserController::create)
app.get("/users/:user-id", UserController::getOne)
app.patch("/users/:user-id", UserController::update)
app.delete("/users/:user-id", UserController::delete)
```

To reduce the amount of noise in these kinds of apps, Javalin has the concept of handler groups, which define a block scope where the `app` object is the receiver and thereby allows you to write tighter code:

```
app.routes {
    path("users") {
        get(UserController::getAll)
        post(UserController::create)
        path(":user-id") {
            get(UserController::getOne)
            patch(UserController::update)
            delete(UserController::delete)
        }
    }
}
```

Handler groups improve readability and significantly reduce the potential for programming errors. Instead of repeating `users` five times and `:user-id` three times, each string is now used only once. This eliminates the need to extract strings into variables, leaving the code more readable and less error-prone.

Asynchronous responses. Asynchronous request handling is simple in Javalin. If you set the `Context` result to be a `CompletableFuture`, Javalin will remove the request from the thread pool and finish it asynchronously. This option improves performance by freeing up the thread pool to deal with new requests instead of waiting for database calls or HTTP requests to finish. Several libraries return `CompletableFuture` in Java, which makes things even simpler. Here is an example using Java 11 and jasync-sql, a database driver for MySQL and PostgreSQL:

```
app.get("/", ctx -> {
    var futureResult =
        connection.sendPreparedStatement("select 0")
                  .thenApply(...)
    ctx.result(futureResult);
});
```

As you can see, there is no difference between an async handler and a blocking one, except for the type of object you give to `ctx.result()`.

Access management. Most production applications eventually need some sort of access management. Security is not something for which Javalin is responsible, but the framework does give you the tools to easily create your own implementation. Every HTTP request in a Javalin application is run through an `AccessManager`. The default implementation is to allow every request, so developers are responsible for defining their own security. Consider the following snippet:

```
get("/secured", ctx -> ctx.result("!"), roles(MY_ROLE));
```

Here, the code defines a GET handler for the path `"/secured"` and attaches the role `MY_ROLE` to it. So how do you get Javalin to respect this role? As with most other concepts in Javalin, `AccessManager` is a functional interface:

```
app.accessManager((handler, ctx, permittedRoles) -> {
    handler.handle(ctx); // handle the request
});
```

The parameter `permittedRoles` is of type `Set<Role>`, and it contains the roles attached to the endpoint. If you have not attached any roles to your endpoint, you can ignore it. If you do have roles attached to your endpoint, you can use it to determine whether the user should have access to the endpoint:

```
app.accessManager((handler, ctx, permittedRoles) -> {
    if (permittedRoles.contains(getUserRole(ctx)) {
        handler.handle(ctx);
    } else {
        ctx.status(401);
    }
});
```

There are no predefined roles in Javalin. The recommended approach is to create an enum that implements `Role`, which is a marker interface—that is, an empty interface.

When to Use Javalin

Javalin is simple and unopinionated, which makes it a good choice if you need to get started quickly. Its abstraction layer is thin, which makes it easy to understand what's going on under the hood. Javalin also is fast, serving 1.1 million requests per second (rps) in the October 2018 TechEmpower benchmarks, which is significantly faster than most heavier frameworks and many lightweight frameworks.

Javalin works well with GraalVM (there's a tutorial on the website). The final binary is only 22 MB (everything included) and starts instantly.

The simplicity of Javalin comes at a cost. Because Javalin does only web applications, developers need to solve database setup, dependency injection, command-line parsing, and other important aspects of an application. The Javalin website has numerous tutorials that show how to approach many of these tasks.

Conclusion

This article presents just a quick overview of Javalin's functionality. As you can see, the scope of Javalin is narrow and limited to the web layer. The codebase is small, and tests make up the majority of it (6,000 out of 10,000 lines of code). If you're interested in contributing, please visit the project on GitHub. Otherwise, consider using Javalin for your projects, both commercial and personal, whenever you need a fast, lightweight web framework. `</article>`

David Åse is a software engineer at Working Group Two, a telecommunications startup. He graduated from the Norwegian University of Science and Technology with a master's degree in computer science in 2014 and immediately joined the open source project Spark Java. He is now an open source enthusiast and the creator and maintainer of two popular open source Java projects: Javalin and j2html.

JONAS HAVERS

Building Microservices with Micronaut

A lightweight framework designed from the ground up for microservices and serverless computing

Micronaut is a microservices framework that is especially designed for the development of modular, easy-to-test applications that embrace the 12-factor design orientation. It can be used to build self-contained systems, microservices, and serverless functions as well as command-line applications with Java, Kotlin, and Groovy on the JVM or on GraalVM.

The Micronaut project was begun in 2017 by Object Computing, Inc.—the same team that developed the Grails framework, which recently celebrated its 10th anniversary. In late May 2018, the Micronaut project was published under Apache License 2, and Micronaut 1.0 GA was released in October 2018.

Opinionated Framework

Micronaut joins a series of frameworks such as Spring Boot, Grails, Jakarta EE, and MicroProfile that follow an opinionated approach using an annotation-driven programming model that enables fast results. The popularity of these frameworks in the Java community—according to the largest survey ever of Java developers (question 17)—is a testament to the fact that developers overwhelmingly prefer an opinionated framework—that is, one that provides autoconfigurations with reasonable defaults and support for different technologies without requiring developers to put all the pieces together from various components. This programming model distinguishes Micronaut and the others from unopinionated frameworks such as Ratpack, Spark, Vert.x, and Javalin.

The goals for Micronaut were to create a framework that is designed from the ground up for

microservices and serverless computing. Micronaut focuses on—and optimizes for—the following aspects of applications for the JVM:

- Fast application startup time
- Low runtime memory footprint
- Minimal use of reflection and proxies
- Few external dependencies
- Simple and fast application tests

To meet these goals, the team performed an analysis of the Spring and Grails frameworks and the challenges of using them to develop microservice applications today. Both of those frameworks were launched at a time when monolithic applications predominated, which is why their design was different.

The problem with most of today's monolithic Java frameworks that provide a large set of out-of-the-box features is that they come with performance and memory-consumption compromises.

One of Micronaut's primary differences from these frameworks is that it performs dependency injection, aspect-oriented programming (AOP) proxying, and configuration management at *compile time*. Micronaut processes annotation metadata into ASM-generated code that is used to glue components together. The JVM JIT compiler additionally optimizes the generated bytecode. Other frameworks use reflection to perform these tasks at runtime—generally in the application startup phase—producing runtime annotation metadata and storing the information in memory. Micronaut does not use the Java Reflection API. Instead, it uses a trio of technologies: the Annotation Processor API for Java, the Kapt Kotlin compiler plugin for annotation processors, and the Groovy AST transformations for metaprogramming. Scala is not supported at the time of this writing.

With Micronaut, the startup time and memory consumption of applications are not tied to the size of the codebase. Other reflection-based inversion-of-control frameworks scan the classpath and then load and cache the reflection metadata for each field, method, and constructor. This can create significant overhead. In addition, the more reflection-based microservices are in operation, the more resources they need and the greater the operational costs.

In comparison, Micronaut's ahead-of-time (AoT) compilation makes it possible to package a minimum Micronaut application in a 10 MB JAR file that can be run with a heap of 10 MB maximum size while enjoying a startup time of about one second. For developers, this benefit is most noticeable during development and the execution of integration tests. Moreover, the fast startup and low memory consumption is a huge advantage for running cloud functions, which is why Micronaut is particularly suitable for the development of serverless applications.

Features

Micronaut has many features that are tailor-made for microservices, including the following:

Reactive streams. Micronaut supports any framework that implements the Reactive Streams standard, including RxJava and Reactor. Web-based nonblocking reactivity is enabled by using Netty. Reactive programming is integral to the design of Micronaut, which consistently supports reactive types to allow efficient use of system resources. For example, Micronaut provides its own service-discover client implementation for Consul that uses Micronaut's reactive HTTP client—while the majority of existing Consul and Eureka clients provide only blocking access. Also included in the framework are reactive database drivers for SQL databases, such as PostgreSQL, and NoSQL databases, including Neo4j, Redis, MongoDB, and Cassandra.

Cloud native features. Micronaut offers the typical features that you'd expect from a cloud native microservice framework. These include resilience mechanisms (retryables, fallbacks, circuit breakers), service discovery, client-side load balancing, distributed tracing, configuration sharing, and so forth. Default implementations and alternative libraries can be easily integrated into applications by declaring compile-time dependencies from the Micronaut ecosystem and runtime external dependencies. These features are covered in the Micronaut documentation.

Message-driven microservices. Message-driven microservices can be easily implemented with Micronaut's support for Kafka by using the compile-time AOP annotations `@KafkaClient`, `@KafkaListener`, `@Topic`, `@Body`, `@Header`, and `@KafkaKey` and a few lines of YAML configuration. Equivalent support is planned for RabbitMQ.

Serverless functions. Micronaut provides support for the development, testing, and deployment of serverless functions for AWS Lambda and any framework-as-a-service (FaaS) system,

such as OpenFaaS and Fn, that supports running functions in the form of containers. Registered functions can be addressed by using a configured service-discovery service (Consul, Eureka, Kubernetes, Google Cloud Platform, or Amazon Route 53). They can be accessed easily using a `@FunctionClient`-annotated client and tested in isolation or via an HTTP server.

OpenAPI documentation. API documentation can be done using OpenAPI (or Swagger). Micronaut creates a Swagger 2.x-compliant YAML file at compile time, which is based on regular Micronaut annotations and Javadoc comments. The YAML file can then be added to the application as a static resource and imported into the Swagger user interface.

GraalVM-ready. Due to its AoT compilation, a Micronaut application can be compiled into a native GraalVM image, which further reduces the already-short startup time of a Micronaut application. With a native GraalVM image, startup time drops from about one second to fewer than 100 milliseconds. The memory consumption of about 60 MB for a regular Java app drops to roughly 20 MB for the native process. You can find more information about using GraalVM with Micronaut in the Micronaut Guide.

Getting Started with the Micronaut CLI

Micronaut offers a CLI application. After installing the Micronaut CLI with SDKman, you can generate project setups from the console. You can also use the CLI to create basic (web) applications, command-line applications, serverless functions, and federations (that is, services that share a profile and its features). Features (such as database drivers) can be applied in the CLI, which will add the dependencies to the project and to templates for code and configuration. You can find the complete list of such features online.

Once a project has been created, singleton beans, scheduled jobs, HTTP clients, and controllers as well as WebSocket clients and servers can be scaffolded (see **Listing 1**).

Listing 1. Micronaut CLI capabilities

```
$ mn -h
Usage: mn [-hnvVx] [COMMAND]
Micronaut CLI command line interface for generating projects and services.
```

```
Commonly used commands are:
  create-app NAME
  create-cli-app NAME
  create-federation NAME --services SERVICE_NAME[,SERVICE_NAME]...
  create-function NAME

Options:
  -h, --help           Show this help message and exit.
  -n, --plain-output   Use plain text instead of ANSI colors and styles.
  -v, --verbose        Create verbose output.
  -V, --version        Print version information and exit.
  -x, --stacktrace     Show full stack trace when exceptions occur.

Commands:
  create-app         Creates an application
  create-cli-app     Creates a command line application
  create-federation  Creates a federation of services
  create-function    Creates a serverless function application
  create-profile     Creates a profile
  help               Prints help information for a specific command
  list-profiles      Lists the available profiles
  profile-info       Display information about a given profile
```

The Micronaut CLI provides autocompletion and help information in its interactive mode, which provides a smooth user experience.

Hands-on Exercise: Creating a Catalog Service for Books

To illustrate the development of a simple Micronaut application, I will create a catalog service for books. You can find the complete codebase on Github, but I will highlight the important parts here.

I start by running the Micronaut CLI's `create-app` command in the console (see **Listing 2**).

Listing 2. Creating a book-catalog project via the CLI

```
$ mn create-app catalogservice.book-catalog --features mongo-reactive
| Generating Java project...
| Application created at .../github.com/JonasHavers/book-catalog
```

[Note: A | at the start of a line indicates screen output from the command. —*Ed.*]

This command creates a Gradle project (by default) called `book-catalog` in a folder with the same name. The base package, `catalogservice`, is created too. The project structure also contains an `Application.java` class with the `main` method for starting the application, an application configuration file called `application.yml`, a `logback.xml` file for logging via Logback, and a Dockerfile for building an application Docker image that uses Java 8 (as of Micronaut 1.0.3). To make the project ready for Java 11, I changed the base image in the Dockerfile to a JDK 11 image.

The feature `mongo-reactive`, which is set via the features option, adds the required dependencies to make use of the MongoDB reactive driver. It also configures a default MongoDB connection URI in the `application.yml` file with default values for the host and port.

To access configuration values in general, such as the database name with the key `mongodb.database`, I can use type-safe configuration and validation with Micronaut. That includes referencing and binding configuration values (`@Value`) at compile time, type-safe mappings to beans by declaring a class annotated with `@ConfigurationProperties`, validation of properties in Hibernate Validator (for example, `@NotNull`, `@Size`, `@Min`, and `@Max`), bean factory configurations (`@Configuration`), and conditional beans (`@Requires`) with Bean Validation 2.

Now, I can make use of the CLI to scaffold some parts of the application. I create `@Singleton` beans for the books repository and an HTTP API controller (see **Listing 3**). There are various annotations for the creation of beans with different scopes (`@Singleton`, `@Prototype`, `@Refreshable`, and so on). These are supported for the different forms of dependency injection with the annotations from JSR 330. Bean injection via the constructor can even be done without the `@Inject` annotation.

Listing 3. Creating use-case and repository beans and a web controller via the CLI

```
$ cd book-catalog/
$ mn
| Starting interactive mode
| Enter a command name to run. Use TAB for completion:
mn> create-bean catalogservice.application.FindBooksUseCase
| Rendered template Bean.java to destination
src/main/java/catalogservice/application/FindBooksUseCase.java
mn> create-bean catalogservice.adapter.mongodb.MongoBooksRepository
| Rendered template Bean.java to destination
src/main/java/catalogservice/adapter/mongodb/MongoBooksRepository.java
mn> create-controller catalogservice.adapter.web.BooksApiController
| Rendered template Controller.java to destination
src/main/java/catalogservice/adapter/web/BooksApiController.java
```

[Note: Screen ouput that does not start with a | is continued from the previous line. —*Ed.*]

Controllers take an RFC-6570 URI template for their `@Controller` and `@Get`, `@Post`, and other mappings. I added a MongoClient dependency to the `MongoBooksRepository` constructor and implemented the port adapter method to fulfill the purpose of the `FindBooksUseCase`.

Let's look at an API test. In an integration test inside the `BooksApiControllerTest` class (see **Listing 4**), I send a GET request to the API controller with Micronaut's `RxHttpClient`, which will access the use case that will then access the database through the repository to return the books data.

Listing 4. Implementing an API test in the BooksApiControllerTest

```
@Test
public void shouldReturnAllBooks() throws Exception {
  try (EmbeddedServer server = ApplicationContext
    .run(EmbeddedServer.class)) {
    try (RxHttpClient client = RxHttpClient.create(server.getURL())){
      // given
```

```
        HttpRequest<?> request = HttpRequest
          .GET("/api/books/");
        // when
        HttpResponse<?> response = client
          .toBlocking()
          .exchange(request, Argument.of(List.class, Book.class));
        // then
        HttpStatus responseStatus = response.status();
        assertEquals(HttpStatus.OK, responseStatus);
        // ...
      }
    }
}
```

If the MongoDB server is not available on the configured port for the test environment, an embedded MongoDB will be bootstrapped and made available for testing. Alternatively, I can replace the `MongoBooksRepository` with a test stub. To do this, I need to create another bean that acts as the stub implementation (see **Listing 5**).

Listing 5. Creating a bean for a books repository test stub

```
$ mn create-bean catalogservice.adapter.test.StubBooksRepository
| Rendered template Bean.java to destination
src/main/java/catalogservice/adapter/test/StubBooksRepository.java
```

In **Listing 6**, I add the `@Replaces` annotation to define the bean I want to replace as well as the `@Requires` annotation with the `Environment.TEST` value to replace the `MongoBooksRepository` with the `StubBooksRepository` when I run the application with the `test` profile.

Listing 6. Creating a bean as a test stub for the books repository

```
@Singleton
@Replaces(bean = MongoBooksRepository.class)
```

```
@Requires(env = Environment.TEST)
public class StubBooksRepository ...
```

I could create other use cases that can be invoked through the API controller and make use of the repository.

To showcase another feature, I want to display the books by using a view. To do this, I can add another controller, `BooksViewController` (see **Listing 7**).

Listing 7. Creating the BooksViewController

```
$ mn create-controller catalogservice.adapter.web.BooksViewController
| Rendered template Controller.java to destination
src/main/java/catalogservice/adapter/web/BooksViewController.java
| Rendered template ControllerTest.java to destination
src/test/java/catalogservice/adapter/web/BooksViewControllerTest.java
```

Although Micronaut is primarily designed for the exchange of pure data (that is, in JSON format), the template engines Thymeleaf, Velocity, and Handlebars are supported for server-side view rendering. The rendering itself is done in the I/O thread pool to avoid blocking the Netty event loop. For this example, I will choose Handlebars.

After creating the controller class with the corresponding CLI command in Listing 7, I edit its source file. I use an instance of Micronaut's `ModelAndView` class to render a view template with dynamic data as in **Listing 8**. You might be familiar with the `ModelAndView` class from the Spring Web MVC framework. I could have used an `HttpResponse` or my own data transfer object (DTO) class as a return value, thus requiring me to specify the view name as the value of the `@View` annotation. To stay in the reactive loop, the return value is encapsulated in a reactive type, in this case an instance of `io.reactivex.Single`.

Listing 8. Implementing the BooksViewController

```
@Controller("/")
@RequiredArgsConstructor
```

```
public class BooksViewController {
  private final FindBooksUseCase findBooksUseCase;
  @Get
  @View
  public Single<ModelAndView> booksView() {
    return findBooksUseCase.invoke()
      .toList()
      .map(bookList -> {
        Map<String, ?> model = Map.of(
          "books", bookList,
          "numberOfBooks", bookList.size()
        );
        return new ModelAndView("booksView", model);
      });
  }
```

For this example to work, I need to add Project Lombok to the project as well as the Handlebars library (see **Listing 9**). By default, the Handlebars renderer will process the `booksView.hbs` template file from the directory `src/main/resources/views/`.

Listing 9. Adding Project Lombok and Handlebars to build.gradle

```
compileOnly "org.projectlombok:lombok:1.18.4" // added
annotationProcessor "org.projectlombok:lombok:1.18.4" // added
annotationProcessor "io.micronaut:micronaut-inject-java"
…
runtime "com.github.jknack:handlebars:4.1.2" // added
```

What's Next?

I could now add configurations to communicate in a microservices architecture. I could use authentication strategies such as Basic Auth, Session, LDAP, and JSON Web Tokens to secure

the service. I could also implement use cases that require WebSockets or server-sent events by using Micronaut's Event API.

In addition, I could dive into managing and monitoring with restrictable endpoints as well as publishing metrics with the Micrometer library to supported services such as Atlas, Graphite, Prometheus, and StatsD. But I will leave all that to you and your curiosity. In the Micronaut guide, which I have frequently referenced in this article, you will find all the nitty-gritty details for doing these things.

Conclusion

Due to its consistent focus on cloud native computing and the reactive paradigm, Micronaut is particularly suitable for the development of microservices and serverless functions, but it is universally applicable as well. The already-fast startup time of about one second for a Micronaut application can be reduced further by using GraalVM instead of the JVM.

Micronaut is worthy of serious consideration for enterprise applications. If you have an existing Spring Boot application, you can try out Micronaut for Spring to benefit from Micronaut's AoT compilation.

To learn more about Micronaut, take a look at the official Micronaut guide to familiarize yourself with all the features. There are also official guides for different topics and projects, which are accompanied by code examples. In addition, the examples repository contains a complete pet store application that has been implemented as a federated microservice architecture. Whatever path you choose, I believe you'll find Micronaut to be a useful lightweight but powerful framework for cloud apps and microservices. `</article>`

Jonas Havers (@JonasHavers) is a freelance full-stack software engineer and lecturer on software engineering from Germany. He develops web applications predominantly in ecommerce projects with a mix of Java, Kotlin, Groovy, TypeScript, and JavaScript. He is also an advocate for remote work, and he and blogs frequently.

TODD SHARP

Helidon: A Simple Cloud Native Framework

Create container-friendly microservices with a minimum of code running straight Java SE.

For a good portion of the internet's early years, web applications were "monolithic." That is, they were single, self-contained applications that encapsulated the entire API and front-end display code. Business logic, validation, data retrieval and manipulation, persistence, security, and the UI were all wrapped up in a single bundle and deployed on application or web servers, such as Tomcat, Apache, or Microsoft IIS. This approach worked, and still works, but it leads to challenges as your application grows in scope, among them the following:

- Deployment: Checking out source code, compiling, testing, bundling, and deploying monoliths takes a long time.
- Dependencies, frameworks, and language: Choices and versions are locked in for the entire application, which leads to difficulties in upgrading when new versions are released.
- Single point of failure: Monoliths are brittle; if the web server goes down, the whole application is down.
- Scaling: The application must be scaled in its entirety—even if only a single portion of the application is the cause of increased load.

There are certainly other challenges that come with monoliths, but these tend to be the ones that cause the most pain to developers, project managers, and operations-minded folks. And for a long time, everyone just dealt with them.

In part, because of those limitations, we're now in the microservice era. This approach, which uses individual services that typically serve a single, distinct purpose and usually are

deployed in some sort of container, is growing in popularity and adoption. It's easy to see why, too. Let's briefly look at the issues I raised earlier and see how microservices address each of them.

- Deployment: Each service can be tested, compiled, and deployed independently.
- Dependencies, frameworks, and language: Each service is free to use the language, framework, dependencies, and versions as necessary and desired.
- Single point of failure: Each service is typically deployed in a container and managed by an orchestration tool, which means outages can be isolated and do not affect the entire application.
- Scaling: Services can be scaled independently of one another, which means the high-load services can scale up while the lower-demand services remain scaled down.

> You've done nothing more than run a few Maven commands and launched the JAR file from the command line, **and you've obtained a fully scaffolded, running application without touching a single line of code.**

Microservices are not a silver bullet and they don't solve all problems, but in many applications, they make a lot of sense. Now that I've established the "why" when it comes to microservices, let's look at the "how."

There are many microservice frameworks available right now, and although creating a new one might seem misguided, that's what Oracle has done with Project Helidon. You don't need to look much further than the framework name to understand Oracle's reasons for creating it: *Helidon* is a Greek word for the swallow—a small, highly maneuverable bird that fits naturally in the clouds. With this in mind, Helidon's creators strove to develop a lightweight set of libraries that didn't require an application server and could be used in Java SE applications.

Helidon comes in two flavors: SE and MP. Helidon SE is simple, lightweight, functional, and reactive. It runs on an embedded Netty web server and falls into the microframework category. It can be compared with Javalin and Micronaut (both of which are covered in this issue of the magazine) or Spark Java. Helidon MP is a MicroProfile-based framework that uses familiar

annotations and components that Java EE/Jakarta EE developers should be familiar with, such as JAX-RS/Jersey, JSON-P, and CDI. Think of Helidon MP in the same league as Open Liberty, Payara, and Thorntail (formerly WildFly Swarm). Let's take a look at Helidon, starting with Helidon SE.

Getting Started with Helidon SE

There's nothing worse than learning about a new tool and hitting a brick wall with a lack of tools or documentation to get you started. That is not an issue here. To get started with Helidon SE, make sure you've got a few prerequisites installed and ready to go: JDK 8 or later and Maven 3.5 or later. If you're using Docker and Kubernetes, you'll get some handy files generated to help you create your containers and deploy them. To take advantage of that, make sure you also have Docker 18.02 or later and Kubernetes 1.7.4 or later. (You can use Minikube or the Kubernetes support in Docker Desktop to run Kubernetes on your desktop.)

Verify your versions like so:

```
$ java --version
$ mvn --version
$ docker --version
$ kubectl version --short
```

Once you've met the prerequisites, it's time to generate a project using the Helidon quickstart Maven archetype. If you're not familiar with archetypes, they are project templates that you can use to scaffold out a basic starter project so you can quickly begin working with a framework. Oracle provides two archetypes: one for Helidon SE and one for Helidon MP.

Here's a basic example you can use from your favorite terminal to generate a Helidon SE project:

```
$ mvn archetype:generate -DinteractiveMode=false \
    -DarchetypeGroupId=io.helidon.archetypes \
    -DarchetypeArtifactId=helidon-quickstart-se \
```

```
    -DarchetypeVersion=0.10.2 \
    -DgroupId=[io.helidon.examples] \
    -DartifactId=[quickstart-se] \
    -Dpackage=[io.helidon.examples.quickstart.se]
```

The archetype is documented at Maven Central, which is where you can always check the latest released version to make sure it's available to use. The items bracketed in the previous snippet are project-specific, and you can edit them to apply to your project. Here's an example I put together for this article:

```
$ mvn archetype:generate -DinteractiveMode=false \
    -DarchetypeGroupId=io.helidon.archetypes \
    -DarchetypeArtifactId=helidon-quickstart-se \
    -DarchetypeVersion=0.10.2 \
    -DgroupId=codes.recursive \
    -DartifactId=helidon-se-demo \
    -Dpackage=codes.recursive.helidon.se.demo
```

Once complete, a fully scaffolded sample application is available in a new directory that matches the value used for the `artifactId`. The example is complete and ready to run, so to see it in action, you can compile the application with

```
$ mvn package
```

This command will run all the generated tests, build the application JAR file, and place that file in the `target/libs` directory. Because the framework includes an embedded web server, you can now run the application by using the following command:

```
$ java -jar target/helidon-se-demo.jar
```

You'll see the application start up and confirm that it's running on port 8080:

```
[DEBUG] (main) Using Console logging
2018.10.18 14:34:10 INFO io.netty.util.internal.PlatformDependent Thread[main,5,main]:
Your platform does not provide complete low-level API for accessing direct buffers
reliably. Unless explicitly requested, heap buffer will always be preferred to avoid
potential system instability.
2018.10.18 14:34:10 INFO io.helidon.webserver.netty.NettyWebServer
Thread[nioEventLoopGroup-2-1,10,main]: Channel '@default' started:
[id: 0x3002c88a, L:/0:0:0:0:0:0:0:0:8080]
WEB server is up! http://localhost:8080
```

But trying to view the root path will result in an error, because the archetype doesn't declare a routing for the root path. Instead, go to `http://localhost:8080/greet`, and you'll see a simple "Hello World" message returned as JSON.

At this point, you've done nothing more than run a few Maven commands and launched the JAR file from the command line, and you've obtained a fully scaffolded, running application without touching a single line of code. Obviously, you will need to dig into the code at some point, but before that, let's see what Helidon provides for Docker support.

Helidon MP and Helidon SE both provide a low barrier to entry for teams looking to adopt a new microservices framework.

Before moving forward, stop the application by pressing Ctrl+C. Now look inside the `target` directory, and you'll notice a few extra files that were created when you ran `mvn package`. You'll see that Helidon has created both a Dockerfile for building a Docker container from your application and an `application.yaml` file for creating a Kubernetes deployment. The files themselves are basic, but they give you out of the box all you need to run them.

Here's the Dockerfile for the demo project (excluding the license information, for brevity):

```
FROM openjdk:8-jre-alpine

RUN mkdir /app
COPY libs /app/libs
COPY helidon-se-demo.jar /app

CMD ["java", "-jar", "/app/helidon-se-demo.jar"]
```

If this is the first time you've seen a Dockerfile, this file declares a base image on the first line. In this case, I am using the `openjdk` image tagged with `8-jre-alpine`, which includes the Java 8 JRE in a very lightweight image based on Alpine Linux. Two lines later, the Dockerfile creates an `app` directory to store the application. The next line copies the output in the `libs` directory into `app/libs`, and the following line copies the JAR file into the `app` directory. The final line tells Docker to run the `java -jar` command at startup, which launches the application.

Let's test out this Dockerfile by running the following command from a terminal in the project root directory:

```
$ docker build -t helidon-se-demo target
```

This instructs Docker to build an image tagged with `helidon-se-demo`, using the Dockerfile located in the `target` directory. You should see output similar to the following after running the `docker build` command:

```
Sending build context to Docker daemon  5.231MB
Step 1/5 : FROM openjdk:8-jre-alpine
 ---> 0fe3f0d1ee48
Step 2/5 : RUN mkdir /app
 ---> Using cache
```

```
 ---> ab57483b1f76
Step 3/5 : COPY libs /app/libs
 ---> 6ac2b96f4b9b
Step 4/5 : COPY helidon-se-demo.jar /app
 ---> 7d2135433bcc
Step 5/5 : CMD ["java", "-jar", "/app/helidon-se-demo.jar"]
 ---> Running in 5ab71094a72f
Removing intermediate container 5ab71094a72f
 ---> 7e81289d5267
Successfully built 7e81289d5267
Successfully tagged helidon-se-demo:latest
```

To confirm all is well, run this command:

```
docker images helidon-se-demo
```

You'll see a container file, `helidon-se-demo`, in your directory. My file from this demo is 88.2 MB. To run this container, use the following command:

```
$ docker run -d -p 8080:8080 helidon-se-demo
```

The `docker run` command uses the `-d` switch to run the container in detached mode (in the background) and exposes the container port using `-p`. The final part of the `docker run` command tells Docker which image to run, which in this case is the image name `helidon-se-demo` that I used in the `docker build` command.

To view the running containers on your system, execute this command:

```
$ docker ps -a
```

Alternatively, you can use a GUI tool such as Kitematic or Portainer. I'm partial to Portainer, so I verified the running container with it, as shown in **Figure 1**.

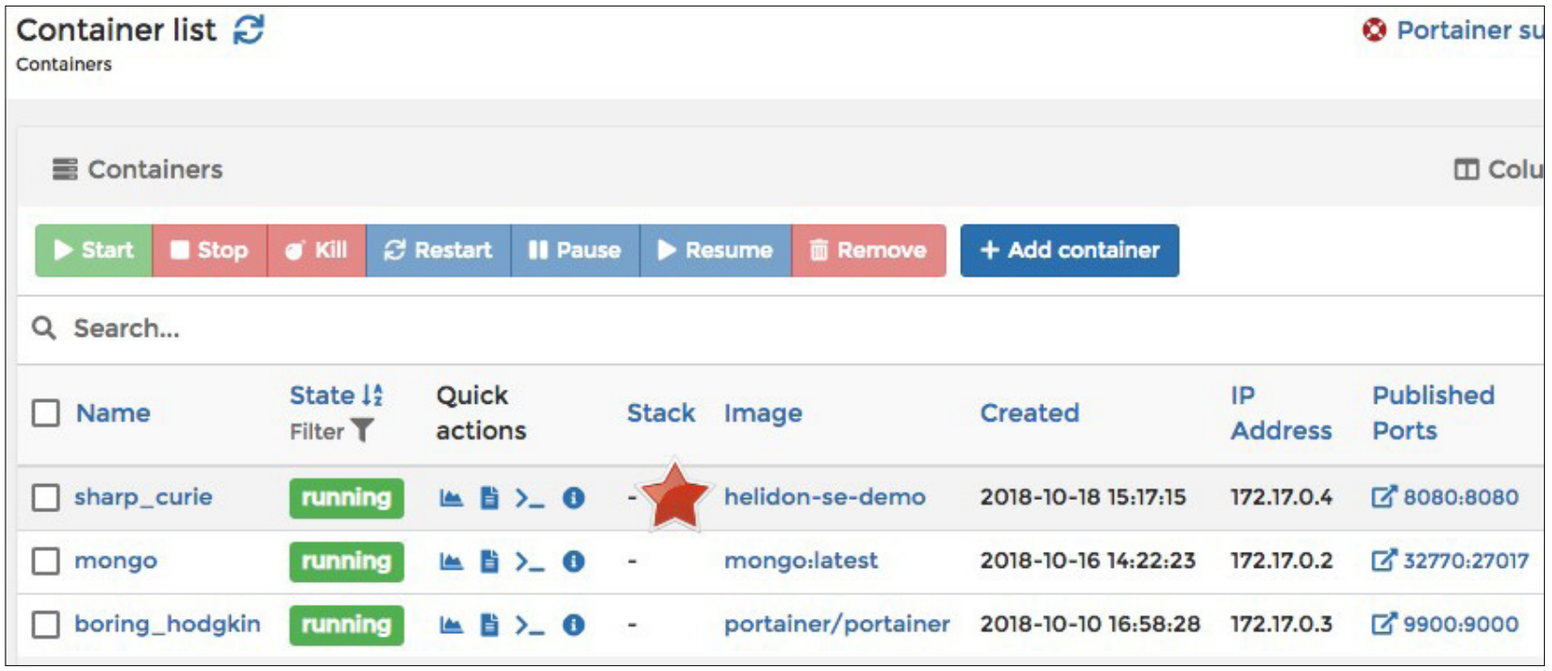


Figure 1. The Helidon application running in a container (see starred entry)

Of course, you could simply go to `http:localhost:8080/greet` again to confirm that the application is running locally (only this time, it's running via Docker).

Running on Kubernetes

Now that you've tested out Helidon's Docker support, let's see what the framework gives you for Kubernetes support. First, kill the running Docker container (via either the command line or the graphical interface of your choice). Then, take a look at the generated file located at `target/app.yaml`. It contains the following:

```
kind: Service
apiVersion: v1
metadata:
  name: helidon-se-demo
  labels:
    app: helidon-se-demo
```

```
spec:
  type: NodePort
  selector:
    app: helidon-se-demo
  ports:
  - port: 8080
    targetPort: 8080
    name: http
---
kind: Deployment
apiVersion: extensions/v1beta1
metadata:
  name: helidon-se-demo
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: helidon-se-demo
        version: v1
    spec:
      containers:
      - name: helidon-se-demo
        image: helidon-se-demo
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 8080
---
```

I won't go over the details of this configuration file, but it gives you the ability to quickly deploy the application to Kubernetes, which provides container management and orchestration. To

deploy the application to a running Kubernetes cluster, enter the following command (again, from the root of the project; otherwise, modify the path to `app.yaml` accordingly):

```
$ kubectl create -f target/app.yaml
```

Assuming everything was created properly, you'll get a reply like this:

```
service/helidon-se-demo created
deployment.extensions/helidon-se-demo created
```

You can confirm the deployment with `kubectl get deployments`, and you can check the service with `kubectl get services`:

```
NAME              TYPE     CLUSTER-IP     EXTERNAL-IP PORT(S)
helidon-se-demo   NodePort 10.105.215.173 <none>      8080:32700/TCP
```

[This line is slightly truncated to fit here. —*Ed.*] As you can see, the service is running on port 32700, which you can verify by visiting the service in the browser.

So far, you have scaffolded an application, built it into a Docker container, and deployed that container to Kubernetes—and yet you have not written a single line of code.

Let's switch gears and examine the code. Open up `src/main/java/Main.java` and take a look at the `startServer()` method to see how Helidon SE initializes the built-in Netty web server:

```
protected static WebServer startServer() throws IOException {

    // load logging configuration
    LogManager.getLogManager().readConfiguration(
        Main.class.getResourceAsStream("/logging.properties"));

    // By default this will pick up application.yaml from
```

```
        // the classpath
        Config config = Config.create();

        // Get web server config from the "server" section of
        // application.yaml
        ServerConfiguration serverConfig =
            ServerConfiguration.fromConfig(config.get("server"));

        WebServer server =
            WebServer.create(serverConfig, createRouting());

        // Start the server and print some info.
        server.start().thenAccept(ws -> {
            System.out.println(
                "WEB server is up! http://localhost:" + ws.port());
        });

        // Server threads are not demon. NO need to block. Just react.
        server.whenShutdown().thenRun(()
            -> System.out.println("WEB server is DOWN. Goodbye!"));

        return server;
    }
```

The comments that are included when this code was generated do a fairly good job explaining what's going on here, but the following steps summarize what it's doing:

1. Initializing logging: Grabbing the configuration from the generated `application.yaml` file (additional application configuration variables can be added here)
2. Creating an instance of `ServerConfiguration` and passing it the host/port info from the application configuration

3. Creating and starting an instance of the `WebServer` and passing it the necessary routing info returned from `createRouting()`

The `createRouting()` method registers any necessary services like this:

```
private static Routing createRouting() {
    return Routing.builder()
              .register(JsonSupport.get())
              .register("/greet", new GreetService())
              .build();
}
```

That's where you register a single endpoint, `"/greet"`, which points at the `GreetService`, which I will break down here. You'll notice a few class variables that use the `Config` class to obtain values from the `application.yaml` file I discussed earlier.

```
private static final Config CONFIG =
    Config.create().get("app");
private static String greeting =
    CONFIG.get("greeting").asString("Ciao");
```

The `GreetService` implements `Service` and overrides the `update()` method to define subpaths under the `/greet` endpoint like this:

```
@Override
public final void update(final Routing.Rules rules) {
    rules
        .get("/", this::getDefaultMessage)
        .get("/{name}", this::getMessage)
        .put("/greeting/{greeting}", this::updateGreeting);
}
```

In this code, `update()` receives an instance of Routing.Rules, which has methods corresponding to each HTTP verb—`get()`, `post()`, `put()`, `head()`, `options()`, and `trace()`—as well as some useful methods such as `any()`, which acts as a catchall and can be used for things such as logging and security.

I have registered three endpoints: `/greet/`, `/greet/{name}`, and `/greet/greeting`. Each endpoint has a method reference pointing to a service method. Each service method registered as an endpoint will receive two arguments: `request` and `response`. This design allows you to pull arguments out of the request scope, such as headers and parameters, and to set elements such as headers and body in the response. Here's what the `getDefaultMessage()` method looks like:

```
private void getDefaultMessage(final ServerRequest request,
                               final ServerResponse response) {
    String msg = String.format("%s %s!", greeting, "World");

    JsonObject returnObject = Json.createObjectBuilder()
            .add("message", msg)
            .build();
    response.send(returnObject);
}
```

It's a bare-bones example, but it illustrates the basic structure of a service method. The `getMessage()` method shows an example of a dynamic path parameter (the `{name}` element within the path that was registered), which allows you to grab that value from the URL.

```
private void getMessage(final ServerRequest request,
                        final ServerResponse response) {
    String name = request.path().param("name");
    String msg = String.format("%s %s!", greeting, name);

    JsonObject returnObject = Json.createObjectBuilder()
```

```
                .add("message", msg)
                .build();
        response.send(returnObject);
    }
```

Calling `http://localhost:8080/greet/todd` would result in the expected output shown in **Figure 2**.

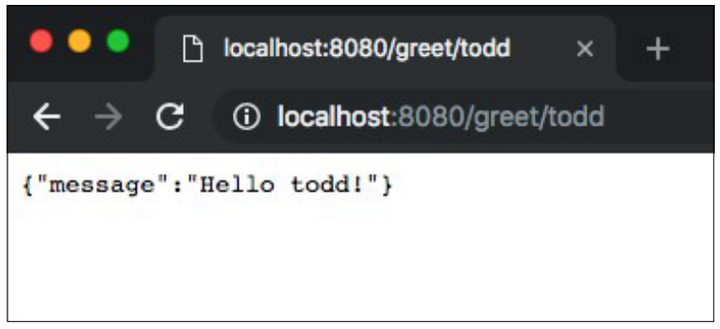


Figure 2. Expected output

The `updateGreeting()` method, shown next, isn't much different from `getMessage()`, but note that it must be called with `PUT` instead of `GET` because I registered it that way in `update()`.

```
private void updateGreeting(final ServerRequest request, final ServerResponse response)
{
    greeting = request.path().param("greeting");

    JsonObject returnObject = Json.createObjectBuilder()
            .add("greeting", greeting)
            .build();
    response.send(returnObject);
}
```

There's much more to Helidon SE, from error handling and static content to metrics and health

support. I highly recommend reading the project documentation to learn about those features and others.

Getting Started with Helidon MP

Helidon MP is the MicroProfile variant of Helidon. If you've been working with Java EE for any amount of time, you'll probably find that it looks pretty familiar. As I mentioned earlier, you'll see the usual things such as JAX-RS/Jersey, JSON-P, and CDI.

To get started quickly, use the Helidon MP archetype just like I did earlier with Helidon SE:

```
$ mvn archetype:generate -DinteractiveMode=false \
    -DarchetypeGroupId=io.helidon.archetypes \
    -DarchetypeArtifactId=helidon-quickstart-mp \
    -DarchetypeVersion=0.10.2 \
    -DgroupId=codes.recursive \
    -DartifactId=helidon-mp-demo \
    -Dpackage=codes.recursive.helidon.mp.demo
```

Take a look at the `Main.java` class, and you'll see that it's even easier than Helidon SE to get the embedded web server running:

```
protected static Server startServer() throws IOException {

    // load logging configuration
    LogManager.getLogManager().readConfiguration(
        Main.class.getResourceAsStream("/logging.properties"));

    // Server will automatically pick up configuration from
    // microprofile-config.properties
    Server server = Server.create();
    server.start();
```

```
        return server;
    }
```

The application is defined in the `GreetApplication` class, which has a `getClasses()` method that is used to register resources that represent routes in the application:

```
@ApplicationScoped
@ApplicationPath("/")
public class GreetApplication extends Application {

    @Override
    public Set<Class<?>> getClasses() {
        Set<Class<?>> set = new HashSet<>();
        set.add(GreetResource.class);
        return Collections.unmodifiableSet(set);
    }
}
```

The `GreetResource` in Helidon MP performs the same tasks as the `GreetService` from Helidon SE, but instead of registering routes individually, you use annotations on the class and methods to represent the endpoints, HTTP verbs, and content-type headers:

```
@Path("/greet")
@RequestScoped
public class GreetResource {

    private static String greeting = null;

    @Inject
    public GreetResource(@ConfigProperty(name = "app.greeting")
      final String greetingConfig) {
```

```
        if (this.greeting == null) {
            this.greeting = greetingConfig;
        }
    }

    @Path("/")
    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public JsonObject getDefaultMessage() {
        String msg = String.format("%s %s!", greeting, "World");

        JsonObject returnObject = Json.createObjectBuilder()
                .add("message", msg)
                .build();
        return returnObject;
    }

    @Path("/{name}")
    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public JsonObject getMessage(@PathParam("name") final String name){
        String msg = String.format("%s %s!", greeting, name);

        JsonObject returnObject = Json.createObjectBuilder()
                .add("message", msg)
                .build();
        return returnObject;
    }

    @Path("/greeting/{greeting}")
    @PUT
```

```
    @Produces(MediaType.APPLICATION_JSON)
    public JsonObject updateGreeting(@PathParam("greeting")
                                     final String newGreeting) {
        this.greeting = newGreeting;

        JsonObject returnObject = Json.createObjectBuilder()
                .add("greeting", this.greeting)
                .build();
        return returnObject;
    }
}
```

Conclusion

There are a few other differences between Helidon MP and Helidon SE, but both versions provide a low barrier to entry for teams looking to adopt a new microservices framework. Helidon is a versatile framework that will help your team quickly develop microservice applications. If containers aren't your preference, you can choose to forgo them altogether and deploy the JAR as you would any traditional JAR. But if your team has adopted containers, the built-in support gives your team the ability to quickly deploy to any cloud-based or on-premises Kubernetes cluster. Because Helidon is being developed by Oracle, the Helidon team will continue developing the framework with some planned enhancements focused on integrating applications with Oracle Cloud. If you're currently hosting your applications in Oracle Cloud, or you plan to migrate to it soon, Helidon might be the right framework for your next microservices application. </article>

Todd Sharp is a developer advocate for Oracle focusing on Oracle Cloud. He has worked with dynamic JVM languages and various JavaScript frameworks for more than 14 years, originally with ColdFusion and more recently with Java/Groovy/Grails on the server side. He lives in the Appalachian mountains of north Georgia (in the United States) with his wife and two children.

Join the World's Largest Developer Community

Oracle Groundbreakers

Download the latest software, tools, and developer templates

Grow your network with the Groundbreaker Ambassador and Oracle ACE Programs

Get exclusive access to hands-on trainings and workshops

Publish your technical articles—and get paid to share your expertise

ORACLE GROUNDBREAKERS developer.oracle.com
Membership Is Free | Follow Us on Social:

@groundbreakers **facebook.com/OracleDevs**

ORACLE®

IAN DARWIN

The Proxy Pattern

A good solution when you need to enable or mediate access to objects, either local or remote

A *proxy* is a stand-in for something else. Corporate shareholders appoint proxies to vote for them at business meetings. Climate scientists use averaged temperature as a proxy for having a thermometer in every square meter of the world. Developers use proxies to substitute for objects that are remote, require protection, or otherwise need mediated access.

The basic approach is that a client object requests a *service object* of a given type from a third party such as a factory, and what it gets is a proxy object that can stand in for, and usually pass control to, the service object. This arrangement requires that the proxy implement the same interface or extend the same class as the one that was requested, so the proxy can be assigned to a variable of the correct type. For this set of examples, I'll use a simple "inspirational quote of the day" `QuoteService` interface with just two methods, which are used as follows:

```
// Normal use
System.out.println("The quote of the day is: " + quoteServer.getQuote());

// Admin use
quoteServer.addQuote("Only the educated are free -- Epictetus");
```

All the code samples for this article are in my GitHub repository.

In an application, I might use a factory method to obtain the instance of the server, instead of calling the constructor directly. This step allows more flexibility, and it removes tight coupling or dependence of the client code on a particular class implementing the interface.

```
QuoteService x = getQuoteService(); // Not = new QuoteServiceImpl();
```

The factory method might simply instantiate a fixed class. But more likely, it will use some configuration to determine which implementation class to create (see the Factory patterns), or it will wrap the known implementation class in a proxy object. I say "wrap" advisedly, because the proxy's main job is to mediate access to the target, so it must maintain a reference to the target.

For these demos, I will use a simple logging proxy, because it's easy to see what the code is doing. The implementation of the `getQuoteService` method might create and return a subclass of the existing `QuoteServerImpl` implementation class, overriding its methods and adding some functionality to the original. This example is short enough that I just use an anonymous class.

```
public static QuoteServer getQuoteServer() {
    final QuoteServer target = new QuoteServerImpl();
    QuoteServer proxy = new QuoteServer() {
        public String getQuote() {
            System.out.println("Calling getQuote()");
            return target.getQuote();
        }
        public void addQuote(String newQuote) {
            System.out.println("Calling addQuote()");
            target.addQuote(newQuote);
        }
    };
    return proxy;
}
```

This example shows a logging proxy where I know the class is being proxied. But it is, in fact, tightly coupled to the target class. What if you want to apply proxying to a variety of classes?

Dynamic Proxy

Java SE provides a mechanism called *dynamic proxy*, which allows you to synthetically create a proxy for a list of arbitrary interfaces—that is, you can set up a proxy at runtime instead of at

compile time. This capability has been around practically forever, since the days of Java 1.3. It does require you to create an object that subclasses `InvocationHandler`. This object will act as the go-between from the caller to the objects being proxied. You can think of the `InvocationHandler` as basically being the proxy. In fact, if you print out the call stack in the target, using either a debugger or `new RuntimeException().printStackTrace()`, you will see that other than some reflection classes, the overall structure is basically the same as in **Figure 1**.

The `InvocationHandler` class contains a convenience method, `newProxyInstance(ClassLoader, Class<?>[], InvocationHandler)`, which, as the name says, gets you a proxy instance for the interfaces given as class descriptors and the given `InvocationHandler`. The `InvocationHandler` interface that you must implement has only one method in it, `invoke`:

```
public interface InvocationHandler {
  abstract Object invoke(Object obj,
    Method method, Object[] args) throws Throwable;
}
```

The arguments passed to your `InvocationHandler`'s `invoke()` method are the proxy object (which the method often doesn't need, but it's there for the times you do), a `java.lang.reflect.Method`

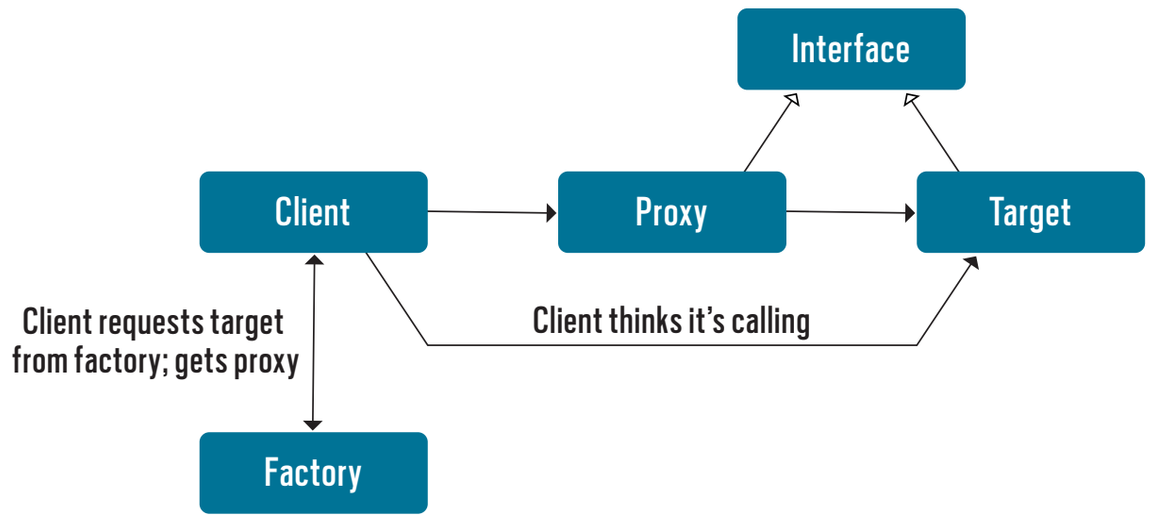


Figure 1. Proxy pattern

descriptor for the method being called, and the list of arguments being passed to that method. Because the API has no way of knowing ahead of time what kinds of objects you will be using, the parameters, the return, and the `throws` clause are written to be as general as possible, using `Object` for the first two and `Throwable` for the third.

Note especially that unless there is a reason not to, the `invoke()` method of the `InvocationHandler` *must* do the actual invocation of the real target. This is the call to `method.invoke()` in the middle of my demo handler's `invoke()` method—the same name and the same arguments, minus the method descriptor itself, which is the object on which you call `invoke()`.

Here is a version of the logging proxy done as an `InvocationHandler`:

```
class MyInvocationHandler implements InvocationHandler {

    private Object target;

public MyInvocationHandler(Object target) {
    super();
    this.target = target;
}

/**
 * Method that is called for every call into the proxy;
 * this must invoke the method on the real object.
 * This method demonstrates both logging and security checking.
 */
public Object invoke(Object proxy, Method method, Object[] argList)
  throws Throwable {
    String name = method.getName() + "()";
    System.out.println("Proxy got request for " + name);
    // Could put security checking here
    Object ret = method.invoke(target, argList);
```

```
        System.out.println("Proxy returned from " + name);
        return ret;
    }
}
```

Here is the `getQuoteServer()` method for the client program:

```
// from DynamicProxyDemo.java

public static QuoteServer getQuoteServer() {
    QuoteServer target = new QuoteServerImpl();
    InvocationHandler handler = new MyInvocationHandler(target);
    return (QuoteServer) Proxy.newProxyInstance(
        QuoteServer.class.getClassLoader(),
        new Class[] { QuoteServer.class }, handler);
}
```

If you examine the object returned from this method by calling `getClass().getName()` on it, you can see that it is a synthetic class, as indicated by its generated name:

```
QuoteServer object is com.sun.proxy.$Proxy0
```

In my online example of dynamic proxy, in my version of the `InvocationHandler`, I added a few lines as a proxy for real security protection. In place of the code comment "could put security checking here," I wrote this:

```
final String userName = System.getProperty("user.name");
if (name.startsWith("add") && !userName.equals("ian"))
    throw new SecurityException(
        "User " + userName + " not allowed to add quotes.");
```

The net result of all this coding is that we have a proxy made up of the dynamic proxy (class `$Proxy0` in this example) and the `InvocationHandler`. The dynamic proxy is generated for us, and the `InvocationHandler` doesn't need to know anything about the actual target, although in more complicated cases it might.

Proxies for Remote Access

Here's one last example from standard APIs: remote access. There's a general term, *remote procedure call* (RPC), for which there are dozens of examples throughout the history of networked computing. The basic idea is that after some setup (such as getting an object from a factory), you invoke an object by using what looks like a local method call, but the object is actually a network proxy that communicates over the network to a peer proxy on the server side, which in turn calls the real service; and the return value is passed back over the same channel. Older examples of RPC include Sun RPC, DCE RPC, and Microsoft Windows RPC. Standard Java APIs that use the RPC paradigm include RMI, CORBA (which was removed from Java 11), JAX-RS, and JAX-WS.

There is not room here to give a full working example, but see my RMI tutorial online for an example.

Proxies in Enterprise Java

The dynamic proxy mechanism works nicely for situations where you know the class or classes to be proxied; however, the `InvocationHandler` itself does not need to be written in a target-specific way. There are cases where you might not know the target class in advance, but you still want to provide services to it. A common example from enterprise Java is the provision of transactional control to business-tier objects controlling data access objects. Both Java EE (now Jakarta) and the Spring Framework provide annotations that are normally placed on business-tier classes and cause a proxy object to begin or join a transaction when a given method begins executing. The proxy will either commit the transaction when the method returns normally or roll it back if the method returns abnormally (for example, by throwing an exception). Here is some pseudocode for a persistent shopping cart using this approach:

```
public class ShoppingService {
    private ShoppingCart cart;
    private Dao dao;

    @Transactional(TransactionType.REQUIRED)
    public void addToCart(Product p) {
        // do validation/calculation work here
        dao.saveCart(cart);
    }
    ...
}
```

The important thing to note is that, in this scheme, *you don't need to write the proxy* or even know its class name for common operations such as transactional control, because these common proxies are provided by the enterprise framework (CDI/EJB or Spring) in response to the annotations. Nor do you need to modify your code to use the proxy (other than annotating it), which means you don't have any runtime dependencies. This design makes the services and data layers easier to unit test (unit testing, after all, means testing each unit in isolation).

However, CDI and Spring give you the option to provide additional proxies of your own. For example, the CDI mechanism supports a form of proxying that uses `Interceptors` that can be applied to enterprise components via annotations (usually) or XML configuration.

Here is how a CDI implementation of the logging interceptor might be used in a business method (the curly braces around the class descriptor remind you that it's an array, in case you want to apply multiple interceptors to the same method). This annotation also can be applied at the class level.

```
@Interceptors({CdiLoggingInterceptor.class})
public void validateCredit() {
    // do some work here
}
```

Here is the code for the logging interceptor or proxy:

```
import javax.interceptor.AroundInvoke;
import javax.interceptor.Interceptor;
import javax.interceptor.InvocationContext;

/**
 * A logging interceptor for CDI.
 */
@Interceptor
public class CdiLoggingInterceptor {

    // @AroundInvoke applies to business method; there are
    // also annotations for constructors, timeouts, etc.
    @AroundInvoke
    public Object log(InvocationContext ctx) throws Throwable {
        Object[] parameters = ctx.getParameters();
        String firstArg = (parameters.length > 0) ?
            "First is: " + formatArg(parameters[0]) : "(empty)";
        String methodName = ctx.getMethod().getName();
        log(String.format("About to call %s with %d arg(s): %s",
                methodName, parameters.length, firstArg));
        Object o = ctx.proceed();   // The actual call!
        log("Returned " + format(o) + " from method " + methodName);
        return o;
    }
    ...
}
```

Unlike the dynamic proxy API, in this code a single parameter, an `InvocationContext`, is passed. It contains the method descriptor, the arguments, and so on. The `InvocationContext` has a

`getMethod()` call that returns the standard `Method` descriptor and a `getParameters()` call that provides the argument list if you want to examine or modify it. The `format()` and `log()` methods aren't shown here but are in the online source code. The context `proceed()` method takes the place of the `invoke()` method.

You might think this approach is a Decorator rather than a Proxy because you are naming the implementation class. However, as an advanced topic, CDI does allow you to use an interface in the `@Interceptors` and resolve the implementation class at runtime by using other annotations. See the official documentation for more details on the `javax.interceptor` package.

Proxy Versus Decorator

As I mentioned in a previous article on the Decorator pattern, Proxy and Decorator both allow you to wrap extra functionality around an object, so the implementation code can look similar. Although there is often overlap, the primary differences are:

- Proxy is primarily about mediating access, whereas Decorator is about adding functionality.
- Proxy is normally hidden from the client (by some kind of creational method), but the client is aware that it is using a Decorator because it must do so explicitly.

Conclusion

Proxy is a good pattern when you need to control access to objects for any purpose, and it can be used for a wide variety of purposes, including enforcing security restrictions, auditing method calls and parameters, hiding the complexity of access (such as with remote objects), or transparently adding behavior (such as logging). `</article>`

Ian Darwin (@Ian_Darwin) is a Java Champion who 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.

BEN EVANS

CHRIS NEWLAND

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

Loop Unrolling

An elaborate mechanism for reducing loop iterations improves performance but can be thwarted by inadvertent coding.

In previous articles in this series on the inner workings of the JVM, you have seen some of the Java HotSpot VM's just-in-time (JIT) compilation techniques, including escape analysis and lock elision. In this article, we discuss another automatic optimization, known as *loop unrolling*. This technique is used by the JIT compiler to make loops (such as Java's `for` or `while` loops) execute faster.

Because we'll be delving deep inside the JVM here, you will at times encounter C code and even some assembly language for the purpose of illustration, so hold on to your hats!

Let's start by considering the following piece of C code, which allocates space for 1 million longs and fills the space with 1 million long random numbers:

```
int main(int argv, char** argc) {
    int MAX = 1000000;

    long* data = (long*)calloc(MAX, sizeof(long));

    for (int i = 0; i < MAX; i++) {
        data[i] = randomLong();
    }
}
```

C can be thought of as a high-level language, but is that really the case? On an Apple Macintosh, the Clang compiler (with the `-S` switch to dump the assembly language in Intel format) produces the following output for the previous code:

```
_main:                                  ## @main
## BB#0:
    pushq   %rbp
    movq    %rsp, %rbp
    subq    $48, %rsp
    movl    $8, %eax
    movl    %eax, %ecx
    movl    $0, -4(%rbp)
    movl    %edi, -8(%rbp)
    movq    %rsi, -16(%rbp)
    movl    $1000000, -20(%rbp)   ## imm = 0xF4240
    movslq  -20(%rbp), %rdi
    movq    %rcx, %rsi
    callq   _calloc
    movq    %rax, -32(%rbp)
    movl    $0, -36(%rbp)
LBB1_1:                                 ## =>This Inner Loop Header: Depth=1
    movl    -36(%rbp), %eax
    cmpl    -20(%rbp), %eax
    jge LBB1_4
## BB#2:                                ##   in Loop: Header=BB1_1 Depth=1
    callq   _randomLong
    movslq  -36(%rbp), %rcx
    movq    -32(%rbp), %rdx
    movq    %rax, (%rdx,%rcx,8)
## BB#3:                                ##   in Loop: Header=BB1_1 Depth=1
    movl    -36(%rbp), %eax
    addl    $1, %eax
    movl    %eax, -36(%rbp)
    jmp LBB1_1
LBB1_4:
```

```
    movl    -4(%rbp), %eax
    addq    $48, %rsp
    popq    %rbp
    retq
```

Looking at the code, you can see that there is one call to the `calloc()` function at the start and only one call (per loop iteration) to the `randomLong()` function. There are two separate jumps, and the produced machine code is essentially the same as that produced from the following variant C code:

```
int main(int argv, char** argc) {
    int MAX = 1_000_000;

    long* data = (long*)calloc(MAX, sizeof(long));
    int i = 0;
    LOOP: if (i >= MAX)
        goto END;
    data[i] = randomLong();
    ++i;
    goto LOOP;
    END: return 0;
}
```

In the case of Java, the equivalent code would be something like this:

```
public class LoopUnroll {
    public static void main(String[] args) {
        int MAX = 1000000;

        long[] data = new long[MAX];
        java.util.Random random = new java.util.Random();
```

```
        for (int i = 0; i < MAX; i++) {
            data[i] = random.nextLong();
        }
    }
}
```

When it is compiled into bytecode, the code becomes:

```
public static void main(java.lang.String[]);
    Code:
       0: ldc           #2    // int 1000000
       2: istore_1
       3: iload_1
       4: newarray       long
       6: astore_2
       7: new           #3    // class java/util/Random
      10: dup
      11: invokespecial #4    // Method java/util/Random."<init>":()V
      14: astore_3
      15: iconst_0
      16: istore        4
      18: iload         4
      20: iload_1
      21: if_icmpge     38
      24: aload_2
      25: iload         4
      27: aload_3
      28: invokevirtual #5    // Method java/util/Random.nextLong:()J
      31: lastore
      32: iinc          4, 1
      35: goto          18
      38: return
```

These programs are very similar in the overall shape of the code. They all perform one operation on the `data` array per loop. However, real processors have pipelines of upcoming instructions, so if the program keeps moving forward linearly, the pipeline can be used efficiently because the next instruction to be executed is always immediately at hand.

But, if a jump instruction is encountered, the benefit of the instruction pipeline is typically lost, because the pipeline contents need to be dumped and reloaded from main memory with new opcodes starting from the jumped-to address. The performance penalty in such a case will be similar to a cache miss—an additional fetch from main memory.

Native code disassembly into readable assembly language is performed directly after the JIT thread emits the compiled method. It is an expensive operation that should not be used on production processes.

For a *back branch*—a jump to a previous point—as seen in a `for` loop, the effect on performance depends on the precise form of the branch prediction algorithm provided by the CPU. Section 3.4.1 of the *Intel 64 and IA-32 Architectures Optimization Reference Manual* [PDF] has details about branch prediction optimization for the specific chips it covers.

However, in the case of Java programs, there is more to this story because of HotSpot's JIT compiler. The JIT compiler contains several optimizations that can produce very different compiled code under favorable circumstances.

In particular, there are optimizations for counted loops (for example, `for` loops) that use an `int`, `short`, or `char` variable as the loop counter. The body of the loop is unrolled, and it is replaced by multiple copies of the loop body, arranged one after the other. This reworking of the loop reduces the number of back branches needed. In addition, it can generate a significant performance improvement over the assembly language generated by the compiled C code, because the instruction pipeline cache needs to be discarded less often.

Let's examine some simple methods that execute loops in different ways. You can look at the assembly language to spot when a loop has been unrolled so that several loop body operations can be executed within a single loop iteration.

Before diving into the assembly language, we should note that the previous Java code needs to be slightly modified for JIT compilation to take effect, because the HotSpot VM compiles only whole methods. Not only that, but methods are not compiled until they have been executed in interpreted mode a certain number of times (typically 10,000 times for fully optimized compilation) before the compiler considers them. Using only a single `main()` exactly as shown would mean that JIT compilation would never be invoked and the optimization would not be performed.

A Java method that is essentially equivalent to the earlier example and that you can use for benchmarking is

```
private long intStride1()
{
    long sum = 0;
    for (int i = 0; i < MAX; i += 1)
    {
        sum += data[i];
    }
    return sum;
}
```

The example method performs summation of data fetched sequentially from an array, and then it returns the total. This is similar to earlier examples, but we chose to return the total to ensure that the JIT compiler does not combine loop unrolling with escape analysis to optimize even further, which would obscure the effect of unrolling.

You can spot a key access pattern in the assembly language that helps you understand what's going on. It shows up as a triple consisting of `[base, index, offset]` made up of registers and offsets, where

- `base register` contains the start address of data in the array
- `index register` contains the loop counter (which gets multiplied by the data type `size`)
- `offset` is used for offsetting each unrolled access

The actual assembly language will look something like this:

```
add rbx, QWORD PTR [base register + index register * size + offset]
```

Considering an array of type `long`, let's look at the conditions under which the loop will be unrolled. Note that loop unrolling behavior can vary between HotSpot VM versions and is dependent on the details of the CPU architecture, but the overall concept remains the same.

To get the disassembled native code produced by the JIT compilers, you will need a disassembly library (the standard choice is hsdis, the HotSpot Disassembler), which you need to install in the `jre/lib` directory of your Java installation.

hsdis can be built from the OpenJDK source code, and instructions for doing so can be found in the JITWatch wiki. Alternatively, Oracle's GraalVM project ships hsdis as part of the downloadable binaries—and the file can simply be copied from the GraalVM installation into the main Java installation location.

Once you have installed hsdis, you need to instruct the VM to output the method's assembly language. To achieve this, you need to add some additional VM switches, including `-XX:+PrintAssembly`.

Note that native code disassembly into readable assembly language is performed directly after the JIT thread emits the compiled method. It is an expensive operation that can affect the performance of your program and should not be used on production processes.

To see the disassembly in action, execute the program with the following VM switches, which output the assembly language for just the named method:

```
java -XX:+UnlockDiagnosticVMOptions \
     -XX:-UseCompressedOops         \
     -XX:PrintAssemblyOptions=intel \
     -XX:CompileCommand=print,javamag.lu.LoopUnrolling::intStride1 \
     javamag.lu.LoopUnrolling
```

This command produces the corresponding assembly language for an `int` loop counter with a constant stride of 1.

Note that we use `-XX:-UseCompressedOops` here only to simplify the assembly language output by switching off the arithmetic for pointer address compression. This saves some memory usage in a 64-bit JVM, but we don't recommend you do this in normal VM use. You can learn all about compressed ordinary object pointers (oops) in the OpenJDK wiki.

The accumulating `long` sum is stored in the 64-bit register `rbx`. Each `add` instruction loads the next value from the `data` array and adds it to `rbx`. The constant offset into the array increases by 8 bytes (which is the size of a Java `long` primitive) with each load.

When the unrolled section branches back to the `main` loop start, the offset register will be incremented by the amount of data processed in this loop iteration:

```
//=============================
// SETUP CODE
//=============================

// MOVE ADDRESS OF data ARRAY INTO rcx
0x00007f475d1109f7: mov rcx,QWORD PTR [rbp+0x18]  ;*getfield data

// MOVE SIZE OF data ARRAY INTO edx
0x00007f475d1109fb: mov edx,DWORD PTR [rcx+0x10]

// MOVE MAX INTO r8d
0x00007f475d1109fe: mov r8d,DWORD PTR [rbp+0x10]  ;*getfield MAX

// LOOP COUNTER IN r13d, COMPARE WITH MAX
0x00007f475d110a02: cmp r13d,r8d

// JUMP TO EXIT IF COUNTER >= MAX
0x00007f475d110a05: jge L0006
```

```
0x00007f475d110a0b: mov r11d,r13d
0x00007f475d110a0e: inc r11d
0x00007f475d110a11: xor r9d,r9d
0x00007f475d110a14: cmp r11d,r9d
0x00007f475d110a17: cmovl r11d,r9d
0x00007f475d110a1b: cmp r11d,r8d
0x00007f475d110a1e: cmovg r11d,r8d

//==============================
// PRE-LOOP
//==============================

// ARRAY BOUNDS CHECK
            L0000: cmp r13d,edx
0x00007f475d110a25: jae L0007

// PERFORM A SINGLE ADDITION
0x00007f475d110a2b: add rbx,QWORD PTR [rcx+r13*8+0x18]  ;*ladd

// INCREMENT THE LOOP COUNTER
0x00007f475d110a30: mov r9d,r13d
0x00007f475d110a33: inc r9d  ;*iinc

// JUMP TO MAIN LOOP IF FINISHED PRE-LOOP
0x00007f475d110a36: cmp r9d,r11d
0x00007f475d110a39: jge L0001

// CHECK LOOP COUNTER AND BACK BRANCH IF NOT FINISHED
0x00007f475d110a3b: mov r13d,r9d
0x00007f475d110a3e: jmp L0000
```

```
//==============================
// MAIN LOOP SETUP
//==============================
             L0001: cmp r8d,edx
0x00007f475d110a43: mov r10d,r8d
0x00007f475d110a46: cmovg r10d,edx
0x00007f475d110a4a: mov esi,r10d
0x00007f475d110a4d: add esi,0xfffffff9
0x00007f475d110a50: mov edi,0x80000000
0x00007f475d110a55: cmp r10d,esi
0x00007f475d110a58: cmovl esi,edi
0x00007f475d110a5b: cmp r9d,esi
0x00007f475d110a5e: jge L000a
0x00007f475d110a64: jmp L0003
0x00007f475d110a66: data16 nop WORD PTR [rax+rax*1+0x0]

//==============================
// MAIN LOOP START (UNROLLED SECTION)
// PERFORMS 8 ADDITIONS PER LOOP ITERATION
//==============================
             L0002: mov r9d,r13d
             L0003: add rbx,QWORD PTR [rcx+r9*8+0x18]   ;*ladd
0x00007f475d110a78: movsxd r10,r9d
0x00007f475d110a7b: add rbx,QWORD PTR [rcx+r10*8+0x20]  ;*ladd
0x00007f475d110a80: add rbx,QWORD PTR [rcx+r10*8+0x28]  ;*ladd
0x00007f475d110a85: add rbx,QWORD PTR [rcx+r10*8+0x30]  ;*ladd
0x00007f475d110a8a: add rbx,QWORD PTR [rcx+r10*8+0x38]  ;*ladd
0x00007f475d110a8f: add rbx,QWORD PTR [rcx+r10*8+0x40]  ;*ladd
0x00007f475d110a94: add rbx,QWORD PTR [rcx+r10*8+0x48]  ;*ladd
0x00007f475d110a99: add rbx,QWORD PTR [rcx+r10*8+0x50]  ;*ladd
```

```
// INCREMENT LOOP COUNTER BY 8
0x00007f475d110a9e: mov r13d,r9d
0x00007f475d110aa1: add r13d,0x8  ;*iinc

// CHECK LOOP COUNTER AND BACK BRANCH IF NOT FINISHED
0x00007f475d110aa5: cmp r13d,esi
0x00007f475d110aa8: jl L0002
//================================

0x00007f475d110aaa: add r9d,0x7  ;*iinc

// IF LOOP COUNTER >= MAX JUMP TO EXIT
             L0004: cmp r13d,r8d
0x00007f475d110ab1: jge L0009
0x00007f475d110ab3: nop

//================================
// POST-LOOP
//================================

// ARRAY BOUNDS CHECK
             L0005: cmp r13d,edx
0x00007f475d110ab7: jae L0007

// PERFORM A SINGLE ADDITION
0x00007f475d110ab9: add rbx,QWORD PTR [rcx+r13*8+0x18];*ladd

// INCREMENT THE LOOP COUNTER
0x00007f475d110abe: inc r13d  ;*iinc
```

```
// CHECK LOOP COUNTER AND BACK BRANCH IF NOT FINISHED
0x00007f475d110ac1: cmp r13d,r8d
0x00007f475d110ac4: jl L0005
//============================
```

(To make things easier, we've included some comments in the assembly code so that the separate sections are clear. For brevity, we show only one exit block, but usually there will be multiple exit blocks in the assembly language to handle the different ways the method can end. The setup section is included for comparison to other operations later in this article.)

When the loop accesses the array, HotSpot VM eliminates array bounds checks by splitting the loop into three sections:

- Pre-loop: This performs initial iterations with bounds checking.
- Main loop: The loop stride (the amount the loop counter is increased on each iteration) is used to calculate the maximum number of iterations that can be performed without requiring a bounds check.
- Post-loop: This performs the remaining iterations with bounds checking.

In Java 10, a more advanced technique called loop strip mining was introduced to further balance the effects of safepoints on throughput and latency.

You can see the practical effect of this approach by looking at the ratio of add operations to jumps. In the unoptimized C case we examined earlier, this ratio was 1:1, but the Java HotSpot VM's JIT compiler has increased this ratio to 8:1, reducing the number of jumps by 87% for this section. Because the effect of a jump is typically to consume from 2 to 300 cycles waiting for a refill of code from main memory, this improvement is potentially significant. (To learn more about how the HotSpot VM eliminates bounds checks when iterating loop-invariant arrays, see the online documentation.)

The HotSpot VM can also unroll loops with an `int` counter and a regular stride of 2 or 4. For example, with a stride of 4, the body is unrolled 8 times and the address offset increases by

0x20 (32) bytes for each access. The compiler can also unroll loops with a counter of type `short`, `byte`, or `char`, but not of type `long`, as we explain in the next section.

Safepoints

The Java code for a method with a `long` loop counter seems very similar to the `int` case:

```
private long longStride1()
{
    long sum = 0;
    for (long l = 0; l < MAX; l++)
    {
        sum += data[(int) l];
    }
    return sum;
}
```

However, with the loop counter of type `long`, the assembly language produced is completely different from the setup section in the previous assembly language listing—no loop unrolling occurs even with a constant stride of 1:

```
// ARRAY LENGTH INTO r9d
0x00007fefb0a4bb7b: mov    r9d,DWORD PTR [r11+0x10]

// JUMP TO END OF LOOP TO CHECK COUNTER AGAINST LIMIT
0x00007fefb0a4bb7f: jmp    0x00007fefb0a4bb90

// BACK BRANCH TARGET - SUM ACCUMULATES IN r14
0x00007fefb0a4bb81: add    r14,QWORD PTR [r11+r10*8+0x18]
```

```
// INCREMENT LOOP COUNTER IN rbx
0x00007fefb0a4bb86: add    rbx,0x1

// SAFEPOINT POLL
0x00007fefb0a4bb8a: test   DWORD PTR [rip+0x9f39470],eax

// IF LOOP COUNTER >= 1_000_000 THEN JUMP TO EXIT CODE
0x00007fefb0a4bb90: cmp    rbx,0xf4240
0x00007fefb0a4bb97: jge    0x00007fefb0a4bbc9

// MOVE LOW 32 BITS OF LOOP COUNTER INTO r10d
0x00007fefb0a4bb99: mov    r10d,ebx

// ARRAY BOUNDS CHECK AND BRANCH BACK TO LOOP START
0x00007fefb0a4bb9c: cmp    r10d,r9d
0x00007fefb0a4bb9f: jb     0x00007fefb0a4bb81
```

There is now only one add instruction per loop body iteration—the ratio of add to jump instructions is back to 1:1, and the benefit of loop unrolling has disappeared. Not only that, but a *safepoint* poll has been added to the loop.

A safepoint is a place in code at which the executing thread knows that it has completed all modifications to internal data structures (such as objects in the heap). It is an ideal time to check and see whether the JVM needs to halt all threads executing Java code. By checking at safepoints and safely suspending execution, application threads provide an opportunity for the JVM to perform operations that might change memory layout and modify internal data structures, such as stop-the-world (STW) garbage collection.

In the case of interpreted code, a very natural location for safepoint checks already exists: after a bytecode has finished executing and just before the next bytecode is executed.

The “in-between bytecodes” safepoint check for interpreted code is very useful, but in the

case of JIT-compiled methods, additional checks must be synthesized and inserted into the code emitted by the compiler.

Without these checks, a thread could continue to run while other threads had already stopped at their safepoints. This could lead to a pathological VM state in which almost all application threads are paused but some continue to run for a substantial amount of time.

HotSpot has several heuristics for inserting a safepoint check into compiled code. The two most common are just before a back branch (as in this case), and just after a method has exited and before control returns to the caller.

As a virtual machine, Java HotSpot VM has advanced loop unrolling capabilities to reduce or remove the overhead of back branches.

However, the appearance of the safepoint check in the example of a `long` counter also points out another feature of the `int` counted loops: They do not contain safepoint checks. This means that the entirety of an `int` counted loop (with constant stride) will run without encountering any safepoint checks, which may be a considerable length of time in extreme cases.

However, consider a loop with an `int` counter and a stride that is not constant, for example one where the stride can be different on each method invocation:

```
private long intStrideVariable(int stride)
{
    long sum = 0;
    for (int i = 0; i < MAX; i += stride)
    {
        sum += data[i];
    }
    return sum;
}
```

This code will indeed force the JIT compiler to emit a safepoint check on each back branch.

If you are concerned about latency pauses introduced by long-running counted `int` loops holding other threads at a safepoint until the loop completes, you can use the VM switch `-XX:+UseCountedLoopSafepoints`. This option adds a safepoint check before the back branch of the unrolled loop. So, in the long assembly code listing, the test would occur every eight additions.

As with every performance-related command-line switch, you should not activate it until you have proved in a performance test that it will provide a significant benefit. Very few applications will see any benefit from activating this switch, so it should not be switched on blindly. In Java 10, a more advanced technique called *loop strip mining* was introduced to further balance the effects of safepoints on throughput and latency.

Let's conclude by looking at a JMH benchmark to compare the performance of iterating the same array using either an `int` counter or a `long` counter. As we explained earlier, the body of a loop with a `long` counter will not be unrolled, and the loop will also contain a safepoint poll.

```
package optjava.jmh;

import org.openjdk.jmh.annotations.*;
import java.util.concurrent.TimeUnit;

@BenchmarkMode(Mode.Throughput)
@OutputTimeUnit(TimeUnit.SECONDS)
@State(Scope.Thread)
public class LoopUnrollingCounter
{
    private static final int MAX = 1_000_000;

    private long[] data = new long[MAX];

    @Setup
    public void createData()
    {
```

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

        for (int i = 0; i < MAX; i++)
        {
            data[i] = random.nextLong();
        }
    }

    @Benchmark
    public long intStride1()
    {
        long sum = 0;
        for (int i = 0; i < MAX; i++)
        {
            sum += data[i];
        }
        return sum;
    }

    @Benchmark
    public long longStride1()
    {
        long sum = 0;
        for (long l = 0; l < MAX; l++)
        {
            sum += data[(int) l];
        }
        return sum;
    }
}
```

The output shows the following:

```
Benchmark                            Mode  Cnt     Score   Error  Units
LoopUnrollingCounter.intStride1     thrpt  200  2423.818 ± 2.547  ops/s
LoopUnrollingCounter.longStride1    thrpt  200  1469.833 ± 0.721  ops/s
```

This means that the loop with the `int` counter performs nearly 64% more operations per second.

Conclusion

The HotSpot VM can perform more-complex loop unrolling optimizations—for example, on a loop containing multiple exit points. In this case, the loop is unrolled, and each unrolled iteration contains a test for the exit condition.

As a virtual machine, the HotSpot VM has advanced loop unrolling capabilities to reduce or remove the overhead of back branches. However, the majority of Java programmers do not need to know about this capability—it's just one more transparent performance optimization that the runtime provides. `</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).

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

Java™

developer.oracle.com/java

DEVELOP WITH THE
GLOBAL STANDARD

#1 Developer Choice for the Cloud

12 Million Developers Run Java
21 Billion Cloud-Connected Java Virtual Machines
38 Billion Java Virtual Machines are in the Cloud

ORACLE®

SIMON ROBERTS

MIKALAI ZAIKIN

Quiz Yourself

More intermediate and advanced test questions

If you're a regular reader of this quiz, you know 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 87

Question 1 (intermediate). The objective is to create methods with arguments and return values, including overloaded methods. Given the following classes:

```
class GenericEngine { public String engType="GE-001"; }

class CombustionEngine extends GenericEngine {
    public String engType="CE-002"; }

class JetEngine extends CombustionEngine {
    public String engType="JE-003"; }

public class Car {
    public void setEngine(Object o) {
        System.out.print("I have unknown engine");
    }

    public void setEngine(GenericEngine ge) {
```

```
        System.out.printf(
            "I have generic engine: %s", ge.engType);
    }

    public void setEngine(CombustionEngine ce) {
        System.out.printf(
            "I have combustion engine: %s", ce.engType);
    }
}
```

And this code fragment:

```
JetEngine e = new JetEngine();
new Car().setEngine(e);
```

What is the result? Choose one.

A. `I have unknown engine`

B. `I have generic engine: GE-001`

C. `I have combustion engine: CE-002`

D. `I have generic engine: CE-002`

E. `I have combustion engine: JE-003`

Answer 2
page 90

Question 2 (advanced). The objective is to use the `Path` interface to operate on files and directory paths. Given this code fragment:

```
Path defaultRoot =
    Paths.get(System.getProperty("user.dir")).getRoot();
Path path = Paths.get("tmp", "john", "..", "doe");
path = defaultRoot.resolve(path);
System.out.print(path.getName(2)); // line n1
```

Assume that the file system containing the current working directory has an empty directory, `tmp`, in its root.

What is the result? Choose one.

A. `john`
B. `..`
C. `doe`
D. Execution completes without exceptions, producing output that depends on the host operating system.
E. An exception is thrown at line n1.

Answer 3
page 93

Question 3 (advanced). The objective is to create and use lambda expressions. Given the following:

```
@FunctionalInterface // line n1
interface SwissKnife {
    default int compare(int i1, int i2) {
        return i1 - i2;
    }

    static void run() {
        System.out.println("Running !");
    }

    String welcomify(String name);
}
```

Which is true? Choose one.

A. The following code compiles:

```
SwissKnife sni = (int a, int b) -> a - b;
```

B. The following code compiles:

```
SwissKnife sni = () -> System.out.print("Running fast !");
```

C. The following code compiles:

```
SwissKnife sni = (a) -> "Welcome, " + a;
```

D. Compilation fails at line n1.

Answer 4 page 95

Question 4 (intermediate). The objective is to define the scope of variables. Given the following two methods, which are declared in the same class:

```
public static float divide(float arg1, float arg2)
    throws ArithmeticException { // line n1
        return arg1/arg2;
}

public static void main(String[] args) {
    try {
        int arg1 = 10;
        int arg2 = 0;
        System.out.printf("Result: %f", divide(arg1, arg2));
    } catch (RuntimeException e) {
        System.out.printf(
            "Bad arguments: %d and %d", arg1, arg2); // line n2
    }
}
```

What is the result? Choose one.

A. `Result: Infinity`

B. `Result: NaN`

C. `Bad arguments: 10 and 0`

D. Compilation fails at line n1.
E. Compilation fails at line n2.

Answer 5
page 97

Question 5 (advanced). The objective is to create and manage date-based and time-based events, including creating a combination of date and time in a single object using `LocalDate`, `LocalTime`, `LocalDateTime`, `Instant`, `Period`, and `Duration`. A senior Java developer is traveling from Chicago O'Hare (ORD) in the United States to Warsaw (WAW) in Poland. She is trying to calculate the duration of the flight but thinks that Poland might be shifting to daylight saving time on the weekend of her journey. The information printed on her ticket is as follows:

Depart Chicago O'Hare
Scheduled: March 24, 2018 5:30 PM (17:30)
Arrive Warsaw
Scheduled: March 25, 2018 9:35 AM (09:35)

She has written the following incomplete code:

```
ZonedDateTime ord =
    ZonedDateTime.of(2018, 3, 24, 17, 30, 0, 0,
                     ZoneId.of("America/Chicago"));

ZonedDateTime waw =
    ZonedDateTime.of(2018, 3, 25, 9, 35, 0, 0,
                     ZoneId.of("Europe/Warsaw"));

// line n1

System.out.print("Time in plane is: " + timeInPlane);
```

What code does she insert at line n1 to calculate the elapsed time expected between departure and arrival? Choose one.

A.
```
long timeInPlane = ChronoUnit.HOURS.between(ord, waw);
```

B.
```
Period timeInPlane = Period.between(
        ord, ZoneId.of("America/Chicago"),
        waw, ZoneId.of("Europe/Warsaw"));
```

C.
```
Period timeInPlane = Period.between(ord, waw);
```

D.
```
Duration timeInPlane = Duration.between(ord, waw);
```

Answer 6
page 99

Question 6 (intermediate). The objective is to explain an object's lifecycle (creation, dereference by reassignment, and garbage collection). Given the following `GCDemo` class:

```
public class GCDemo {
    public static ArrayList<Object> l = new ArrayList<>();
    public void doIt() {
        HashMap<String, Object> m = new HashMap<>();
        Object o1 = new Object(); // line n1
        Object o2 = new Object();
        m.put("o1", o1);
        o1 = o2; // line n2
        o1 = null; // line n3
        l.add(m);
        m = null; // line n4
        System.gc();// line n5
    }
}
```

And given this code fragment:

```
GCDemo demo = new GCDemo();
```

```
demo.doIt();
demo = null;  // line n6
```

When does the object created at line n1 become eligible for garbage collection?

A. At line n2
B. At line n3
C. At line n4
D. At line n5
E. At line n6
F. None of the above

Question 1
page 81

Answer 1. The correct option is C. This question investigates how methods are selected for invocation and also how variables are resolved. The relevant rules for overloaded methods and for field access are documented in *Java Language Specification* sections 15.12.2 and 15.11.1.

The code fragment constructs a `JetEngine` and initializes a variable of that same type to refer to the object. It then calls a `setEngine` method, using the variable as the argument. There are three overloaded methods called `setEngine`, and although none of them takes an argument of exactly the type `JetEngine`, the argument of each overload is a parent of `JetEngine`. Therefore, any of them could accept the argument. But one method must be selected, and the compiler performs that selection. The specification describes how the method is selected from the candidate overloads in as many as three stages.

The first stage (which happens to be where the decision is made in this example) is to try to identify the target method based on the provided argument types, without using any autoboxing/unboxing or variable argument list–handling rules.

If the first stage were to fail, a second stage would look for a target method by applying autoboxing/unboxing, and finally, the third stage would look for a match by applying variable argument list-handling rules.

So, the first stage selects the "most specific" method based on the types of the parameters. *Java Language Specification* section 15.12.2.5 says the following:

> If more than one member method is both accessible and applicable to a method invocation, it is necessary to choose one to provide the descriptor for the run-time method dispatch. The Java programming language uses the rule that the *most specific* method is chosen.

There's a fairly long and detailed definition of "most specific" in the specification, but in this context, it simply means "nearest to the actual argument type." Given a `JetEngine` as an actual parameter, the `CombustionEngine` is the "most specific" formal parameter type, and `Object` is the least specific. Therefore (and given that no `setEngine(JetEngine e)` method is defined), the compiler will generate code to invoke the `setEngine(CombustionEngine ce)` method.

In light of this discussion, you know that the output will start with the message `I have combustion engine`. Consequently, options A, B, and D are incorrect.

Next, let's consider which engine type message is printed. Each of three engine variants has the same `engType` variable, and each subclass "hides" the parent's class variable. This represents highly dubious style, and this question illustrates fairly convincingly why it's considered bad.

It might seem as if the three variables called `engType` are "overrides" in the same sense that occurs in the definitions of methods with the same name in a hierarchy of classes; however, this is *not* the case. A single instance of the `JetEngine` object actually contains three independent variables called `engType`, each with different values and each visible from different scopes. *Java Language Specification* section 15.11.1 states the following:

> Note that only the type of the *[p]rimary* expression, not the class of the actual object referred to at run time, is used in determining which field to use.

The "primary expression" is the part that comes before the last dot. In the following `setEngine` method, that means the primary expression is `ce`, and its type is `CombustionEngine`. Consequently, the `engType` variable that is printed is the one embedded in the `CombustionEngine` part of the object, and that has the value `CE-002`. Because of this, you can see that option C is correct and option E is incorrect:

```
public void setEngine(CombustionEngine ce) {
    System.out.printf("I have combustion engine: %s", ce.engType);
}
```

This behavior might be surprising, but the central point is that the late-binding effect applies only to the invocation of a nonprivate, nonfinal instance method on an object, not to direct field access. The behavior can perhaps be improved in several ways:

- You could render this behavior less surprising if you simply avoided using the same variable name.
- You could change this behavior to print a more expected message if you avoided making direct reference to the variable `engType` and instead invoked a method `getType()` and ensured that this method is overridden in all three classes. However, this is a cumbersome solution, duplicating an identical `getType` method in every class.
- Another possibility would be to have a single `engType` variable, defined in the base `GenericEngine` class. This variable is configured appropriately via a chain of constructors in the three classes. The following example implements this approach by using a `final` field.

```
class GenericEngine {
    public final String engType;
    protected GenericEngine(String engType) {
        this.engType = engType; }
    public GenericEngine() { this("GE-001"); }
}
```

```
class CombustionEngine extends GenericEngine {
    protected CombustionEngine(String engType) { super(engType); }
    public CombustionEngine() { this("CE-002"); }
}

class JetEngine extends CombustionEngine {
    protected JetEngine(String engType) { super(engType); }
    public JetEngine() { this("JE-003"); }
}
```

However, the best solution might be simply to avoid using class inheritance in this situation entirely. If the protected constructor of the `GenericEngine` class in the last code block were made public, that would allow all three engine types to be handled directly by that class. Of course, there might be other constraints on a more complete design, but a common mantra in modern software engineering is to "prefer delegation over inheritance." Using delegation for code reuse is an approach you should understand, but it's more complex than can be discussed in the context of this question.

Question 2
page 82

Answer 2. The correct option is B. The `Path` class represents the idea of a path on the file system—that is, an optional sequence of hierarchical directory names, perhaps ending in a filename. A `Path` object itself is not directly linked with the physical file system. Such a connection is created when some action is taken using the `Path`—for example, listing the contents of a directory or creating a file. The reason for avoiding such a hard connection is fairly compelling: If you could use a `Path` only to represent something that already exists, a `Path` could not be used in the process of creating a new directory or file.

Given this background, the question revolves around four inquiries. What do the `Paths.get` operations do? What does the `resolve` operation do? Can you extract `getName(2)` from the path after `resolve` has done its work? And if `getName` doesn't throw an exception, what value does it provide in this situation?

Java's APIs give you the tools to work with hierarchical directories without ever making explicit literal references to root directories and path separators.

You could be forgiven for wondering what the first two lines do in general, and to be fair, some of that code is beyond the scope of the exam. That first line in particular is likely beyond the scope of the real exam, but we left it in to show how you can answer a question even if you don't necessarily have a perfect understanding of everything. Given that none of the options admits the possibility of code other than the last line failing in any way, you can safely assume that these first lines compile and run without crashing. It's also reasonable—and accurate—to assume that this opening code does what it seems to suggest.

The first line extracts a `Path` object that represents the root of the file system that contains the user's current working directory. So, for example, if the current working directory is `C:\users\simon\javaprojects\examproj1`, the extracted `Path` object represents `C:\`.

The second line extracts another `Path`—which is likely to be a relative path—representing a path hierarchy that might be represented in a UNIX-like format as either `tmp/john/../doe` or `tmp/doe`.

Which of those two relative paths do you get? From the perspective of `Paths.get`, the `..` is just an element of the path; it is not treated as anything special in the basic creation process. Therefore, the effective path has four elements: `tmp`, `john`, `..`, and `doe`.

Why do you make the path this way rather than by simply specifying a `String` such as `tmp/john/../doe`? The reason is that Java seeks to allow you to create platform-independent programs. It's important to be able to describe and manipulate paths on a file system without hard-coding things such as forward slashes or backslashes. Allowing a variable-length argument list of `Strings` and joining them at runtime in whatever way is appropriate for the current execution platform is much more flexible than defining literal strings with separators. You can access the system property `file.separator` to find the local character if you want, but why bother? So yes, this approach does work, and it looks as if it should. Therefore, option D is incorrect.

As a side note, in literal path strings, forward slashes (UNIX style) work properly on Microsoft Windows–based JVMs, but backslashes (Windows style) *fail* on UNIX-based JVMs. The flexibility on Windows JVMs is possible because a forward slash is not a legal character at the operating system level in Windows paths or filenames. Therefore, if the JVM sees the forward slash, it can safely swap that character for a backslash before handing the path to the Windows system. But if a backslash shows up in a literal path string in UNIX, it's a legal—if rather odd—character in a UNIX pathname, so simply swapping would change a valid meaning.

This entire question of forward slashes and backslashes is made even more interesting because there are other systems that use different path schemes entirely. On the OpenVMS operating system, local paths have a structure such as `sys$disk:[dir.path.elements]myfile.txt;1`. Clearly, making simple assumptions about slashes would not work with this. The lesson here is that Java's APIs give you the tools to work with hierarchical directories without ever making explicit literal references to root directories and path separators. Clearly, it's a good habit to use them, because it will protect your code if it's ever run on unfamiliar operating systems.

The next question is what does `resolve` do? There are a couple of corner cases, but the most commonly used behavior is the one used here. If the invocation path is not empty and the argument path is not absolute, the effect is to concatenate the two paths. As a result, this takes the relative path (`tmp/john/../doe` in UNIX-like format) and anchors it to the root of the current file system. Again in UNIX-like format, this becomes `/tmp/john/../doe`. Note that `resolve` still does not eliminate the `..` part; that is the job of the method `normalize`.

So, now you need to know whether `getName(2)` is successful and, if it is, what it returns. It turns out that `Path` effectively treats the elements of a path (in this case `tmp`, `john`, `..`, and `doe`) like a list with a zero-based indexing system. Further, the root part of the path is not included in that list nor in its indexing scheme. Importantly, this exclusion of the root is consistent even if the path were on a Windows system where this example would likely represent `C:\tmp\john\..\doe`. So, in a platform-independent way, the indexing starts at zero with `tmp` (and never with `C:\` or something similar). This again shows that option D is incorrect.

Further, you can see that the index 2 should be within the valid range, so option E is incorrect, because no exception is thrown. Further, you will get `..` regardless of the host operating system. This tells you that option B is correct and options A and C are both incorrect.

Question 3
page 83

Answer 3. The correct option is C. In the functional programming style, functions can be arguments and return values of other functions. Many languages support this idea directly, but most object-oriented programming (OOP) languages allow passing only objects to and from functions (and, of course, you usually call functions in your object system's methods).

To address this seeming limitation, OOP design patterns (such as the Command pattern) suggest creating an object (function or method—or whatever you want to call it) that contains the desired behavior and passing that around. This works perfectly well, but the syntax tends to be cumbersome, because all the "syntactic scaffolding" necessary to define a class, and then create and instantiate an object from it, really has no immediate relevance to the point of the source code—which is, in such a situation, simply to describe a function.

To ameliorate this, Java 1.1 provided anonymous classes, which reduce the syntactic scaffolding a little, but more importantly allow the definition, instantiation, and usage to be done all in the same place—for example, as a parameter to a method invocation. Passing an object that implements `SwissKnife` might look like this when written using the anonymous syntax:

```
doStuffWithASwissKnife(new SwissKnife() {
    public String welcomify(String a) {
        return "Welcome, " + a;
    }
});
```

Note that while the anonymous syntax shown here is consistent with any version of Java from 1.1 onward, the `static` and `default` methods in the `SwissKnife` interface require at least Java 8.

As stated earlier, the behavior definition is colocated with the use, which is good, but the syntax is still very cluttered.

Java 8 addressed this clutter and, in specific situations, allows a better syntax, known as lambdas. Lambdas are essentially expressions that define a class and instantiate an object from that class, but they do it in a way that allows you to write code that addresses only the definition of a single function. Because of this restriction of defining a single function, the syntax can be used only to implement interfaces that have a single abstract method (SAM). Java's documentation generally refers to this type of interface as a *functional interface,* and the annotation with the same name can be used to verify that an interface does indeed have exactly one abstract method.

So, the `SwissKnife` interface is a functional interface, because it has a SAM named `welcomify`. The other two methods provide implementations, so they are not abstract and don't present a problem for the lambda syntax rules. The job of the `@FunctionalInterface` annotation is to cause a compiler error if it is attached to an interface that has zero abstract methods or more than one. Because the code given compiles correctly, option D is incorrect.

By now, it might be clear that the relationship between an interface and a lambda expression that implements it centers on the SAM in that interface. The lambda provides an implementation for that method, and the lambda's arguments and return type must match those of the interface's abstract method. By the way, the Java compiler's ability to determine the applicable type for a particular context is called *type inferencing*.

Therefore, a lambda that implements `SwissKnife` must take a single parameter of type `String`, and it must return a `String`. The only matching lambda is option C, so C is the correct answer. Options A and B are incorrect because their parameter lists do not match the requirement of the `SwissKnife`'s single abstract method.

As a side note, lambda expressions may include or omit the types of all their parameters; it is probably better, in general, to omit parameter types and allow them to be inferred from the context. However, in some situations, perhaps with overloaded methods, the target interface might be ambiguous. In such a situation, providing the argument type might be necessary to allow compilation to succeed. Sometimes, including the argument type might make a program easier to read. Option A includes the argument types, but it fails because the argument list is

incorrect in type (`int`, rather than `String`) and in number (two instead of one), not because the argument types are specified. For illustration, the following would also be correct:

```
SwissKnife sni = (String a) -> "Welcome, " + a;
```

Question 4
page 84

Answer 4. The correct option is E. This is one of those uncomfortable questions. Because compilation fails and in daily coding the problem would be reported immediately by the development environment, it can seem like an unreasonable question to ask. But, exams are *not* daily coding; they attempt to probe your knowledge. As such, a little care and attention to detail should lead you to the right answer and, in the process, allow you to demonstrate an element of core knowledge that is being legitimately tested. If you're still uneasy by the end of the discussion, know that the exam creators try hard to limit the number of questions that fall into this category, and the information in the question—specifically "fails at line n2"—should be used to help you spot the right answer.

Consider what happens if you perform a division by zero. It turns out that it depends on the type of the expression.

Let's look at the question. The setup has all the hallmarks of being about the way Java performs arithmetic and, in particular, how it handles division by zero—but it's not. It's about the scope of variables.

In general, a local variable, such as `arg1` and `arg2` in this sample, is visible from the point of declaration to the end of the immediately enclosing block; that's the region bounded by curly braces. As a result, `arg1` and `arg2` are not accessible in the `catch` block, and line n2 fails to compile. This—along with the assurance that there's only one correct answer—tells you that option E is correct.

An important note is that the description of visibility just given isn't complete and, therefore, isn't fully correct. Formal parameters (such as the variables in the argument list of a method) will be visible from the point of declaration to the end of the block that is associated

with whatever those variables are formal parameters to. By way of examples, the variable `args`, which is the formal parameter of the `main` method, is visible throughout the `main` method body. The variable `e`, which is the formal parameter of the `catch` block, is visible throughout that `catch` block. Similar rules apply to similar situations, including variables declared in the resources section of a try-with-resources structure, and those declared in `for` loops.

So, you could fix this particular compilation error simply by moving the declarations of the two variables further up in the source code so that they are directly above the `try` keyword. In that case, the code would compile and run. Now, to make this question and its discussion more interesting, consider what would happen if that were the case; after all, the distractors (the wrong answers) were chosen to be at least tempting, which should be true of all multiple-choice exam questions.

Perhaps the best starting point is to consider what happens if you perform a division by zero. It turns out that it depends on the type of the expression. If an integer division expression has zero in the divisor (the bottom part of a fraction), the code throws an `ArithmeticError`. In practice, this means that both the divisor and the dividend (the top) must be of integral types; if either has a floating-point type, the expression has a floating-point type, and no exceptions are possible. In fact, floating-point expressions with division by zero produce one of three special values: `Infinity`, `-Infinity`, and `NaN` (“Not a Number”). If the dividend is nonzero, and it has the same sign as the zero divisor (floating-point arithmetic distinguishes positive and negative zero), you get `Infinity`. If the signs are different, you get `-Infinity`, and if both the dividend and divisor are zero, you get `NaN`. We hope this information is interesting, but it’s not needed for either of the Java exams. However, because the code could not possibly produce `NaN`, option B must be incorrect.

The next consideration is that the variables `arg1` and `arg2` are declared as `int`, but the `divide` method takes two `float` arguments. So, would this division be handled in floating-point or integer arithmetic format? The arguments are promoted to `float` for the method invocation, so a floating-point expression is evaluated and, again, no exception will be thrown. That tells you that option C cannot be correct, even if the scope problem were fixed. Also, from the previous

discussion, you can tell that if the variable scope issue were fixed and the code were compiled, the output would be in the form of option A.

You could then ask, if the expression cannot throw an exception, is it an error that the method declares an exception that will definitely not arise? The answer is no, and in fact, it's a general rule that methods are permitted to declare exceptions that they never throw. One reason this is important is that overriding methods are not permitted to throw checked exceptions that are not permissible from the method being overridden. On this basis, abstract methods in interfaces regularly declare exceptions that, given that they have no implementation, they obviously cannot throw. It's also worth noting that `ArithmeticException` is an unchecked exception, so there's never a requirement to declare it on any method. However, it's also perfectly permissible to do so, even if it's unusual and not recommended style. From this, you can determine that line n1 does not cause a compilation error and option D is incorrect.

Question 5
page 85

Answer 5. The correct answer is option D. In fact, the intrepid programmer is correct: Poland did move its clocks forward an hour at 02:00 on the morning of March 25, 2018. However, the passenger's task is actually easy. The `ZonedDateTime` class specifically addresses the representation of not only time zone but also of legally prescribed changes of time, such as daylight saving time. Consequently, the various means of calculating the offset between two `ZonedDateTime` objects take any such shifts into account automatically.

Since Java 8 introduced the Date and Time API, Java contains a database of all the historical daylight saving time changes from governments around the world, and it is updated with new information as it becomes available.

Let's review the options and compare what they produce. Option A is incomplete; it will print `Time in plane is: 9` to the console, which is the correct number of full hours, but it will entirely ignore the minutes. The departure time ends with 30 minutes and the arrival time ends with 35 minutes, so the number of hours is not exact. If the arrival time were at the same

number of minutes past the hour, the output might appear correct, but it would be by luck, not because of technical validity. Consider also if our developer's flight had been to India or another country that has time zones that vary from those of other countries by an amount that is not a whole numbers of hours. Because of this, option A is incorrect.

You might improve this option by rewriting the code like this, which would give a result that is a bit "better"—`9.083333333333334`—although the format is not very helpful:

```
double timeInPlane =
    ChronoUnit.MINUTES.between(ord, waw) / 60.0;
```

The `Period` class measures time differences in days, months, and years, and its `between` method accepts only two arguments, which must be `LocalDate` objects. Because of this, both options B and C are incorrect.

Option D is correct; it accurately denotes the time spent in the plane, and it takes into account the time zones the flight passes through and the daylight saving time change in Poland that occurs midflight. The output is in the rather odd form `PT9H5M`. This cryptic presentation specifies a "period of time of 9 hours and 5 minutes," which is, in fact, the scheduled time for the flight.

The `PT` part of the output is hardcoded in the `Duration.toString()` method, and as you might expect, it's possible to extract the hours, minutes, and so on for a nicer presentation. The Java Date and Time API is very full featured, and it's worth spending a little time browsing Oracle's tutorial on the topic as well as the Javadoc so you can get the best out of this powerful API.

As a side note, since Java 8 introduced the Date and Time API, Java contains a database of all the historical daylight saving time changes from governments around the world, and it is updated with new information as it becomes available.

There's a final observation to make here about the nature of test questions and what you need to answer them. While we were preparing this question, we pondered whether options B and C were bad because they appeared to require rote learning of the API. Questions that test the simple learning of facts, particularly facts that an IDE will tell you, are considered to be a

bad idea (although a few exist in the real test, and you'll see many on the free sample tests that are abundant on the web). However, we decided to keep these options because although they can be rejected by rote knowledge, they are best answered from a position of understanding. So, let's explore how you would address options B and C from a position of understanding the API, rather than from simple learning of that API's contents.

It's a key feature of the API as a whole that the `Duration` and `Period` classes represent different concepts. Specifically, `Duration` represents "physics time" and `Period` represents a human-calendar type of time in days—strictly in days, months, and years, but expressly without hours, minutes, seconds, and the like. From that, it's a simple deduction that `Period` cannot be appropriate to the task of extracting hours and minutes. Therefore, you can eliminate options B and C immediately, without concern about whether the specific methods exist or not.

With a little luck, this persuades you that understanding will beat, or at least greatly augment, knowledge, even in a multiple-choice test.

Question 6
page 86

Answer 6. The correct option is F. This question investigates how objects become eligible for garbage collection and, at the risk of spoiling the story by giving away the outcome, one way that memory leaks are possible in Java.

It turns out that the object created on line n1 never becomes eligible for garbage collection unless a new value is written to the static variable `l` or the class `GCDemo` remains loaded in the JVM. However, class unloading is not a topic of either the exams, so from an exam perspective, the object is effectively collectable only if more code is added. If the object never becomes eligible for collection, the memory is simply reclaimed by the operating system after the JVM process exits. But because the question never addresses when the JVM exits, and that is not an option you can select, the correct answer is F: "None of the above."

By the way, a while ago, the Oracle exams adopted a general policy of avoiding "None of the above" and similar variations as options. However, because this rule has not always been in place, we can't be completely certain you won't ever see it, and it suits the learning purpose of this question to use it here.

The garbage collector never makes anything eligible for collection; all it ever does is collect things that are already eligible.

So, how does an object become eligible for garbage collection? This happens when there are no live references to the object left in the program. What does that mean? Well, when the object of interest is created on line n1, the variable `o1` is assigned to point at it. The variable `o1` is a reference (a reference is a kind of pointer, but one to which you cannot make arbitrary changes). In other words, `o1` is not the data; it's how to find the data. Because `o1` is a variable that the thread can use, it's a live reference and the object is not eligible for garbage collection, because the program can still find the object.

Importantly, anytime you take a copy of the value of `o1`, you duplicate the instructions on how to find the object (you don't duplicate the data). For the sake of a colorful analogy, imagine the object is buried treasure. If a pirate has a treasure map (a reference), he can find the treasure and use it if he wants to. Further, if another pirate makes a copy of that treasure map, either pirate can find the treasure. Now, if the first pirate's map sinks with his ship, the second pirate can still find the treasure and use it. But if all the copies of the map go down in sinking ships, nobody can find the treasure. (The assumption is that the pirates don't remember the map details in their heads.) This is equivalent to the situation in which an object becomes eligible for garbage collection.

Now, let's get back to real life. The line right before line n2—`m.put("o1", o1)`—puts a copy of the reference to the object into a `Map` (the data structure `Map`, not the pirate map, although the analogy holds there, too). This means that the `Map` structure can be used to reach the object.

Next, line n2 overwrites the pointer value in `o1` with the value of `o2`. This action, in effect, turns the pirate map for finding the object into a pirate map for finding another object. But the original object can still be reached by using the variable `m`. The variable `m` lets you find the `Map` data structure, and the `Map` data structure still lets you find the original object. So, at this point, the original object is still not lost and is not eligible for collection. That means that option A is incorrect.

At line n3, the value of the `o1` reference variable is changed again, but because it no longer refers to the original object, that doesn't change the picture. You still can reach the object, so option B is also incorrect.

The next line (between lines n3 and n4) makes a copy of the reference value currently in variable `m`. The copy is placed in the `List` referred to by the static variable `l`. That means that you could follow the reference in `l` to find the `Map` (which at this point is still also referred to by the variable `m`), and then from the `Map` you can find the object. So, you now have another route for finding that object. Importantly, the variable `l` is `static`, so unlike method local variables `m`, `o1`, and `o2`, which cease to exist when the method `doIt` returns to its caller, the variable `l` (that is, the reference variable `l`, which is distinct from the `List` to which it refers) will not disappear unless the class `GCDemo` is unloaded (or the JVM shuts down).

Next, line n4 nulls out the direct reference to the `Map` (that's the variable `m`). But it's still possible to find the `Map` because you have the reference to it stored in the `List` from the previous line. By following the chain from that `List` to the `Map`, and then from the `Map` to the object, you can still reach the object. So, even now, the object is still reachable and still not eligible for garbage collection. Therefore, option C is incorrect.

At line n5, the code invokes the `System.gc()` method, which encourages the garbage collector to invest some time cleaning up. This method has been the subject of much commentary regarding why it might be best avoided, but that isn't relevant here. The garbage collector never makes anything eligible for collection; all it ever does is collect things that are *already* eligible. As you saw, the object of interest wasn't eligible on line n4, so the call on line n5 changes nothing, and option D is also incorrect.

After the `doIt` method returns to its caller, the local variables `o1`, `o2`, and `m` cease to exist. Those pirate maps go down with the sinking ship that is the `doIt` method. But the static variable `l` in the `GCDemo` class still exists, it is still accessible, and the transitive series of references still leads to the object. So, there is no change at this point.

Line n6 nulls out the reference `demo` that refers to an instance of the `GCDemo` class, incidentally rendering that object eligible for collection. But the variable `l` that still lets you reach the object is `static`, so unless something changes the value of `l` or the class `GCDemo` is unloaded, the

object remains reachable. From this you can see that option E is also incorrect and, by elimination, option F must be correct.

As we hinted at the start, this kind of behavior is a good candidate for creating a memory leak. Of course, it's possible this reference chain was kept deliberately, and there might be other code that cleans up the `List` contents, and the contents of the `Map`s that are in that `List`, at intervals. In that case, everything would be OK, but if this were overlooked, you would likely find that the program consumes ever more memory as it runs, creating a memory leak.

What actions could you take to avoid having this become a memory leak? Several options exist, and the right one depends on the real purpose of the code. Note that there are two potential leaks in the current code. Every call to `doIt` puts another `Map` into the `List`, and in that `Map` there's another `Object`. These instances of `Map` and `Object` must be kept under control. Actions that might be involved in keeping memory allocation under control include the following:

- Explicitly removing object references from the `Map`(s)
- Explicitly removing `Map` references from the `List`
- Explicitly overwriting the value of the `static` variable `l`, perhaps with `null` </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.

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