Java™ magazine

By and for the Java community

MARCH/APRIL 2018

GROOVY 3.0 61 | JVM ESCAPE ANALYSIS 73 | REST API WITH SPRING 81

MICROSERVICES and CONTAINERS

15 CREATING MICROSERVICES WITH MICROPROFILE AND DOCKER

32 DEVOPS PIPELINES FOR CONTAINERS

42 MIXING JAVA MODULES AND OSGi

53 WOOKIEE: MICROSERVICES WITHOUT THE CONFIG HASSLES

ORACLE.COM/JAVAMAGAZINE

ORACLE®

//table of contents /

COVER FEATURES

OTHER FEATURES

DEPARTMENTS

COVER ART BY WES ROWELL

EDITORIAL

Editor in Chief
Andrew Binstock

Managing Editor
Claire Breen

Interim Managing Editor
Leslie Steere

Copy Editors
Lea Anne Bantsari, Karen Perkins

Technical Reviewer
Stephen Chin

DESIGN

Senior Creative Director
Francisco G Delgadillo

Design Director
Richard Merchán

Senior Designer
Arianna Pucherelli

Designer
Jaime Ferrand

Senior Publication Designer
Sheila Brennan

Production Designer
Kathy Cygnarowicz

PUBLISHING

Publisher and Audience Development Director
Karin Kinnear

Audience Development Manager
Jennifer Kurtz

ADVERTISING SALES

Sales Director
Tom Cometa

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.

The Evolving Desktop Metaphor

JavaFX and other desktop technologies adapt to a changing world.

It's no secret that the role of desktop computing has changed considerably over the last 10 years. Before the advent of web applications and the widespread popularity of mobile devices, desktops were the prevailing metaphor for user-facing apps. Major languages all had specialized libraries that delivered the ability for rich UI experiences. Java had Swing (later JavaFX) and the Eclipse Standard Widget Toolkit (SWT). C had the GTK toolkit, and C++ had Qt. And of course Microsoft had a variety of platform-specific UI toolkits.

The decline of the PC market during the last decade has been matched by a surge in mobile-oriented design. Many UI metaphors today come straight from mobile devices rather than being desktop designs rejiggered for mobile platforms. Microsoft Windows 10 is a canonical example of this trend.

Meanwhile, apps that require an elaborate or complex UI have been steadily opting for browser-based presentation, in which UIs are developed with the combination of HTML5, CSS, and JavaScript. Only a few applications have UI needs or local processing requirements that exceed the browser's ability to deliver a satisfactory experience. These are the ones that steadfastly remain desktop applications. They include programming environments, productivity software such as Microsoft Office, and visually intensive tools, like those from Adobe. In addition, a variety of scientific software relies on desktop-style UIs.

While the design of the UI front end has been evolving, so has the back end. Whereas desktop applications used to be delivered as large executables (Microsoft Office is gigabytes in size; IDEs

PHOTOGRAPH BY BOB ADLER/THE VERBATIM AGENCY are hundreds of megabytes), there is an emerging trend to deliver applications as complete containerized software packages in which the runtime is bundled. This derives in part from mobile and, especially, from the cloud paradigm, where microservices and applications are delivered in Docker containers. (The first two feature articles in this issue discuss how to do this very thing.)

The evolution of the front and back ends of the desktop metaphor as well as the PC's long-term decline have led to reconsideration of the role of JavaFX and other desktop-based technologies in the Java ecosystem.

In early March, Oracle announced that JavaFX would be unbundled from the JDK and JRE as of Java 11, which is slated to ship in September. JavaFX is already open source, but by splitting it off from the JDK, Oracle enables evolution of the tools and library separately from core Java. This is likely to be good news for interested parties, especially in Europe, where JavaFX has a particularly dedicated following.

Several pundits and analysts have wondered whether this move means Oracle is pulling back its commitment to JavaFX. I should point to an announcement made at a conclave of Java Champions just prior to the 2016 JavaOne conference, in which Oracle executives stated that they view the future of the UI to be based on web technologies, scripted with JavaScript. So, while I don't speak for Oracle, it's clear that it believes the future of the UI does not run through the desktop metaphor.

To this end, Oracle also announced that Java Web Start and Java applets would be slowly phased out. The discontinuation of applets was previously announced, but now we know the timeline, as we do for Java Web Start: they'll both be supported in Java SE 8 through March 2025. However, Java Web Start will not be included in Java 11 or later releases.

While JavaFX will be spun off, the underlying graphical subsystem consisting of AWT and Swing will continue to be part of the JDK for the foreseeable future, and they'll be supported far into the next decade.

The megatrend that is squeezing the desktop has had an unfortunate consequence, which is the diminution of rich application frameworks. While JavaFX survives and will likely advance in its new separated status, the disappearance of other frameworks and toolkits, such as Adobe Flash, Microsoft Silverlight, and Mozilla Prism, point to a future of lessened competition for excellence in advanced UI presentation. In this sense, I am heartened that we still have JavaFX, with its rich media, multimedia, and 3D graphics, but I wish it were not so lonely in its mission.

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

Your Destination for Oracle and Java Expertise

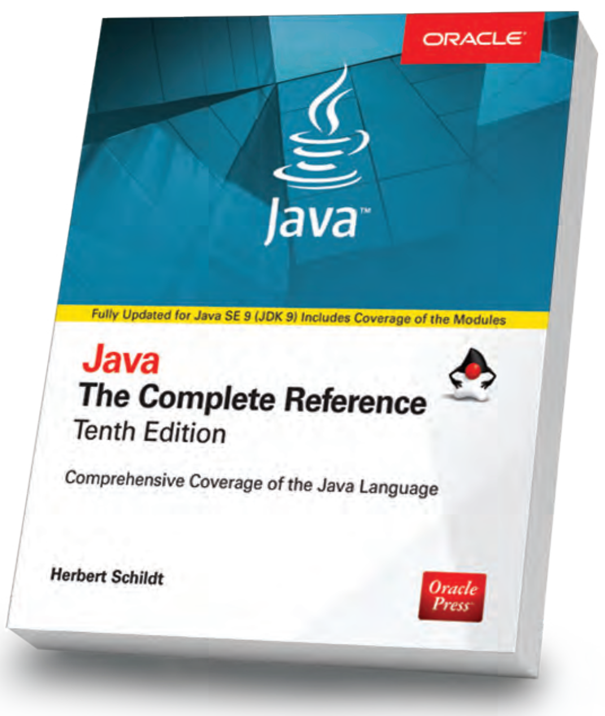
Written by leading experts in Java, Oracle Press books offer the most definitive, complete, and up-to-date coverage of Java available.

ORACLE
Java
Fully Updated for Java SE 9 (JDK 9)
Java
A Beginner's Guide
Seventh Edition
Create, Compile, and Run Java Programs Today
Herbert Schildt
Oracle Press

Java: A Beginner's Guide, 7th Edition

Herb Schildt

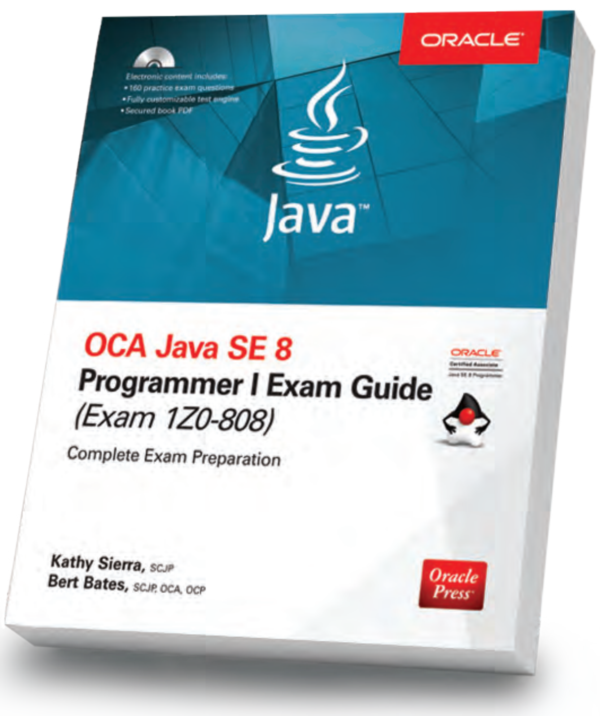
Revised to cover Java SE 9, this book gets you started programming in Java right away.


ORACLE
Java
Fully Updated for Java SE 9 (JDK 9) Includes Coverage of the Modules
Java
The Complete Reference
Tenth Edition
Comprehensive Coverage of the Java Language
Herbert Schildt
Oracle Press

Java: The Complete Reference, 10th Edition

Herb Schildt

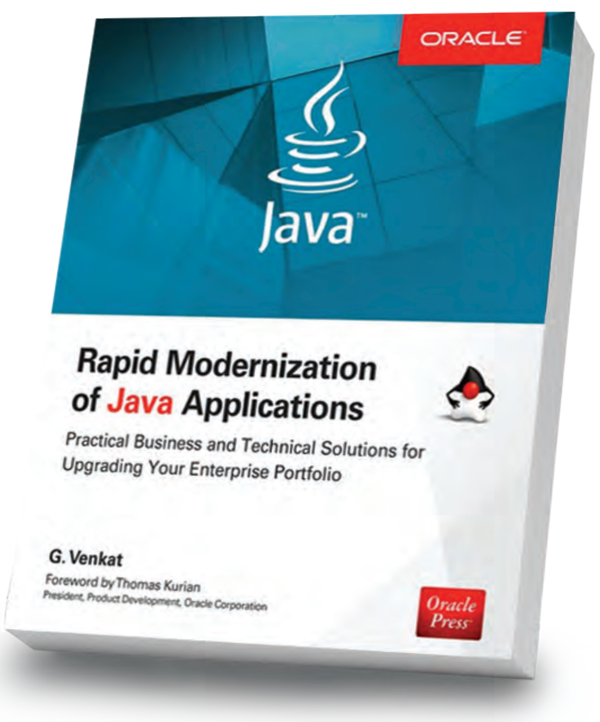
Updated for Java SE 9, this book shows how to develop, compile, debug, and run Java programs.



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

Kathy Sierra, Bert Bates

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


ORACLE
Java
Rapid Modernization
of Java Applications
Practical Business and Technical Solutions for Upgrading Your Enterprise Portfolio
G. Venkat
Foreword by Thomas Kurian
President, Product Development, Oracle Corporation
Oracle Press

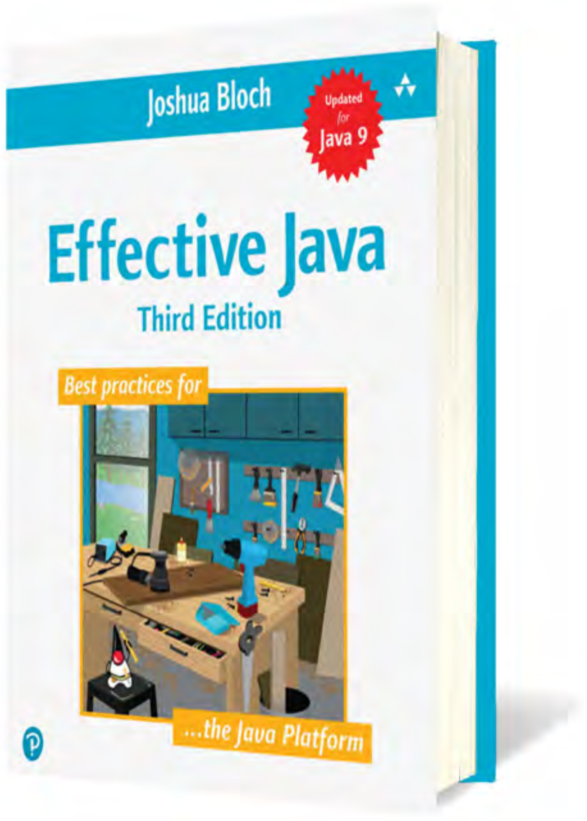
Rapid Modernization of Java Applications

G. Venkat

Adopt a high-performance enterprise Java application modernization strategy.

Available in print and eBook formats.

www.OraclePressBooks.com • @OraclePress



EFFECTIVE JAVA, THIRD EDITION

By Joshua Bloch

I can finally review the long-anticipated third edition of the classic book *Effective Java,* by Joshua Bloch. Its release at the end of 2017 brought the book's content up to date with Java 9. Given that the previous edition covered only through Java 6, you can see how long this edition has been anticipated by *Effective Java*'s many fans—fans who justifiably appreciate the clear and elegant execution of the book's central promise: a discussion of best practices in Java programming. Bloch presents these practices through a series of 90 short essays (up from 78 in the previous edition), each of which elaborates a useful recommendation. For example, here are some of the best practices added in this volume:

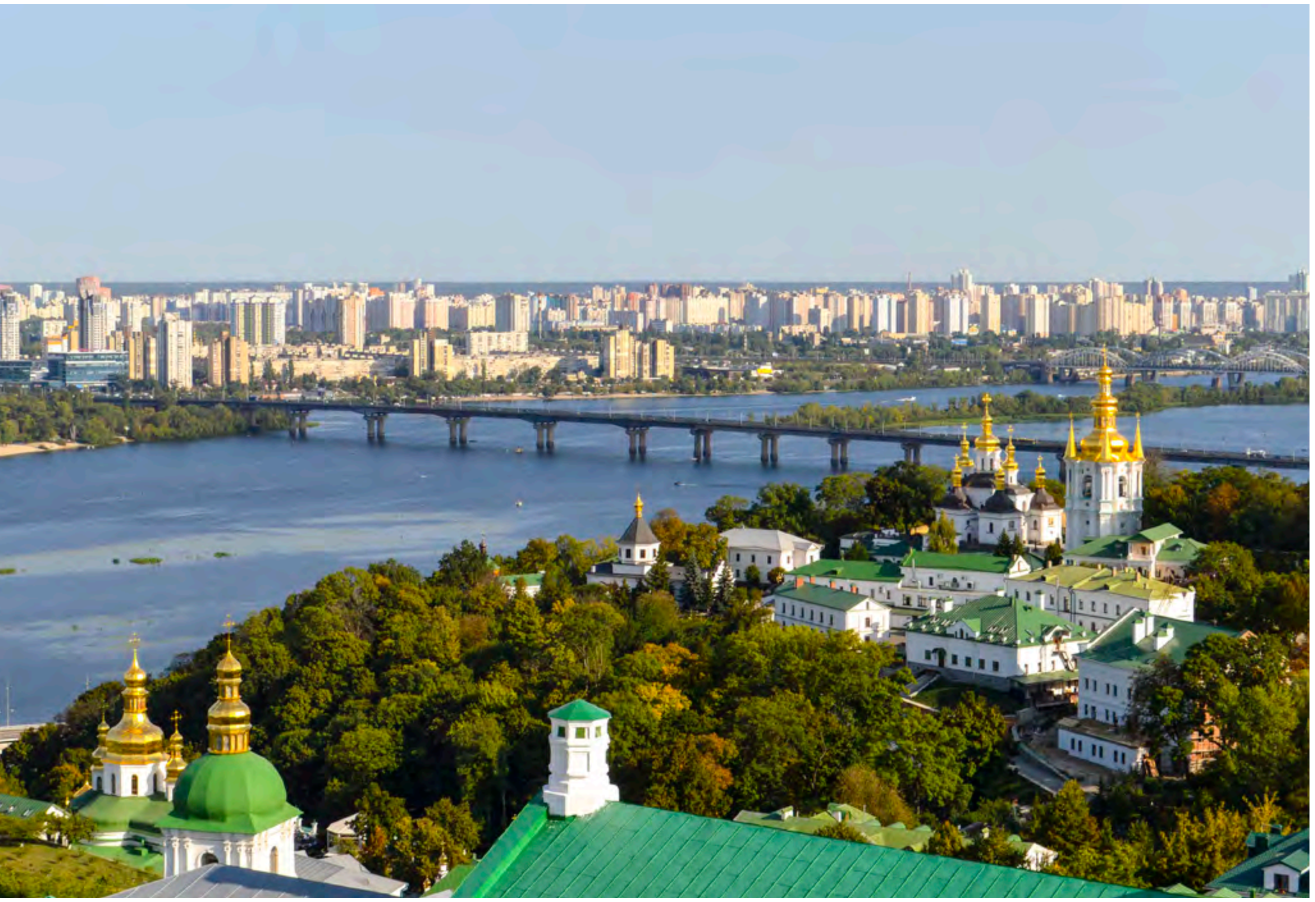
- Prefer method references to lambdas
- Prefer Collection to Stream as a return type
- Use Streams judiciously

As you can see, these recommendations are true best practices. That is, they are not intended as tutorials on the language, but rather as good things to do once you've learned the language and are using it daily. The first example on the list above will likely elicit from some readers a "Huh, I never thought about that" response. And that is precisely what gained this book praise and attention in its original release—the many recommendations that readers simply had not considered, or if they'd considered them, had not explored fully.

Other recommendations such as "Use Streams judiciously" seem banal in their formulation. "Use x judiciously" is always good advice, so what makes this suggestion important? Bloch's eight-page explanation details how Streams should be used. His principal thesis is that Streams provide little benefit if they don't implement a functional-style operation. To facilitate this, Bloch dives into the Collectors API and explains its tight relationship with operations for which many developers misuse Streams. The explanation contains examples of badly used Streams and the correct use. In the process, you learn a lot about how Streams and Collectors operate, the intended use of Streams, and how to apply that knowledge to write better and clearer code. Not bad for eight pages!

The recommendations stretch across many topics, from tricky topics such as the Serializable interface, to the more mundane, such as when to omit accessors on data-only classes.

The book is supremely readable: the style is concise and clear, and the code examples are short and to the point. As a result, *Effective Java* is a pleasant volume to read through from beginning to end—learning to refine your coding skills as you go. It is one of the very few books I recommend without reservation to all Java programmers who are past the beginner stage. *—Andrew Binstock*



PHOTOGRAPH BY FRANCISCO ANZOLA/FLICKR

JEEConf

MAY 18–19
KIEV, UKRAINE

JEEConf, the largest Java conference in Eastern Europe, focuses on practical experience and development. Topics include modern approaches for developing distributed, highly loaded, scalable enterprise systems with Java and innovations and new directions in application development using Java.

JAX DevOps

APRIL 9 AND 12, WORKSHOPS
APRIL 10–11, CONFERENCE
LONDON, ENGLAND

This event for software experts highlights the latest technologies and methodologies for accelerated delivery cycles, faster changes in functionality, and increased quality in delivery. More than 60 workshops, sessions, and keynotes will be led by international speakers and industry experts. There's also a two-in-one conference package that provides free access to a parallel conference, JAX Finance.

JAX Finance

APRIL 9–12
LONDON, ENGLAND

JAX Finance is a Java event focused on core Java and the specific technological needs of the financial industry, including low latency, messaging, and exchange architecture.

React Amsterdam

APRIL 13
AMSTERDAM, THE NETHERLANDS

This event draws front-end and full-stack developers from across the globe to discuss the React JavaScript library for building user interfaces.

Devoxx France

APRIL 18–20
PARIS, FRANCE

Devoxx France offers more than 220 presentations on topics including Java, alternative JVM languages, web programming, and architecture.

JAX

APRIL 23 AND 27, WORKSHOPS
APRIL 23–26, CONFERENCE
MAINZ, GERMANY

This German developer conference focuses on Java, architecture, and software innovation. Topics this year include microservices, Spring Framework 5, JDK 10, and property-based testing.

Voxxed Days Melbourne

MAY 2–3
MELBOURNE, AUSTRALIA

Voxxed Days is heading down under to Melbourne, Australia. The event will feature insights into cloud, containers and infrastructure, real-world architectures, data and machine learning, the modern web, and programming languages.

PHOTOGRAPH BY NOWICIEL/FLICKR

Java Day Istanbul

MAY 5
ISTANBUL, TURKEY
Java Day Istanbul is a one-day conference that includes 36 sessions presented in three parallel tracks, covering Java 9 through Java 11, DevOps, big data, microservices, and more.

Devoxx UK

MAY 9, WORKSHOPS
MAY 10–11, CONFERENCE
LONDON, ENGLAND
Devoxx UK is a Java-focused technology conference by developers, for developers. Topics include functional languages, Java SE, JDK, Ansible, Kubernetes, Istio, PaaS, serverless architecture, Java EE, EE4J, Spring, neural networks, TensorFlow, and encryption.

GeeCon

MAY 9–11
KRAKÓW, POLAND
The 10th anniversary of this conference gathers more than 1,000 participants and more than 75 speakers to discuss Java and JVM-based technologies, dynamic languages, enterprise architectures, patterns, distributed computing, software craftsmanship, mobile, and more.

Gluecon

MAY 16–17
BROOMFIELD, COLORADO
Gluecon is a developer-oriented conference focused on cutting-edge tools and platforms. Topics include serverless architectures, containers, microservices, APIs, DevOps, mobile, analytics, performance monitoring, and blockchain applications.

WeAreDevelopers World Congress

MAY 16–18
VIENNA, AUSTRIA
Billed as the largest developer congress in Europe, WeAreDevelopers expects more than 8,000 participants and more than 150 speakers for keynotes, panel discussions, workshops, hackathons, contests, and exhibitions. The program covers talks and sessions on front-end and back-end development, artificial intelligence, robotics, blockchain, security, and more.

J On The Beach

MAY 23–25
MÁLAGA, SPAIN
J On The Beach (JOTB) is an international workshop and conference event for developers interested in big data, JVM and .NET technologies, embedded and IoT development, functional programming, and data visualization.

Spring I/O

MAY 24–25
BARCELONA, SPAIN
Spring I/O focuses on the Spring Framework ecosystem and is the largest Spring-based conference held in Europe.

jPrime

MAY 29–30
SOFIA, BULGARIA
jPrime will feature two days of talks on Java, JVM languages, mobile and web programming, and best practices. The event is run by the Bulgarian Java User Group and provides opportunities for hacking and networking.

Riga Dev Days

MAY 29–31
RIGA, LATVIA
The biggest tech conference in the Baltic States covers Java, .NET, DevOps, cloud, software architecture, and emerging technologies. This year, Java Champion Simon Ritter is scheduled to speak.

O'Reilly Fluent

JUNE 11–12, TRAINING
JUNE 12–14, TUTORIALS AND CONFERENCE
SAN JOSE, CALIFORNIA

The O'Reilly Fluent conference is devoted to practical training for building sites and apps for the modern web. This event is designed to appeal to application, web, mobile, and interactive developers, as well as engineers, architects, and UI/UX designers. It will be collocated with O'Reilly's Velocity conference for system engineers, application developers, and DevOps professionals.

EclipseCon France

JUNE 13–14
TOULOUSE, FRANCE

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

QCon

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

Although the content has not yet been announced, QCon conferences typically offer several Java tracks along with tracks related to web development, DevOps, cloud computing, and more.

OSCON

JULY 16–17, TRAINING AND TUTORIALS
JULY 18–19, CONFERENCE
PORTLAND, OREGON

Groundbreaking open source projects, from blockchain to machine learning frameworks, will be the focus of the 20th annual OSCON event. Live coding, emerging languages, evolutionary architecture, and edge computing are among the topics this year.

JCrete

JULY 22–28
KOLYMBARI, GREECE

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

Oracle Code Events

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

APRIL 4, Hyderabad, India
APRIL 10, Bengaluru, India
APRIL 17, Boston, Massachusetts
APRIL 24, Bogotá, Colombia
MAY 8, Shenzhen, China
MAY 11, Warsaw, Poland
MAY 15, Buenos Aires, Argentina
MAY 17, Singapore
MAY 30, London, England

Java Forum Nord

SEPTEMBER 13
HANNOVER, GERMANY

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

jDays

SEPTEMBER 24–25
GOTHENBURG, SWEDEN

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

Strange Loop

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

JavaOne

OCTOBER 22–25
SAN FRANCISCO, CALIFORNIA
Whether you are a seasoned coder or a new Java programmer, JavaOne is the ultimate source of technical information and learning about Java. For five days, the world's largest collection of Java developers gather to talk about all aspects of Java and JVM languages, development tools, and trends in programming. Tutorials on numerous related Java and JVM topics are offered.

Devoxx Belgium 2018

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

Topconf Tallinn

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

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

THE POLISH JUG

The Polish Java User Group, founded in December 1999, is Poland's first JUG. It was started by Adrian Nowak and hosted Poland's first Java conference in November 2000, which set the tone for the huge popularity of the platform and language in the country.

Currently, PJUG is associated primarily with Kraków, Poland's second-largest city; many other Polish cities have since started their own JUGs—often via the direct involvement of prominent PJUG members. PJUG has also grown over the years and through its activities and events, with more than 2,300 developers now part of its meetup.

Since its inception, PJUG has been constantly active in the Java community, organizing meetups, hackathons, and the annual GeeCON conference (in friendly collaboration with the Poznań JUG). GeeCON is celebrating its 10th anniversary edition this year, so if you're not a GeeCON geek already, check out the conference site.

PJUG also organizes GeeCON 4 Kids with Kraków's Hackerspace and hosts developer events with JUGs from other cities. It cofounded the local Google Developers Group and regularly collaborates with the Kraków Ruby User Group, Software Craftsmanship Kraków, and Kraków Scala. PJUG experiments with meeting formats, and for some time it has been running a small conference every month, working with Kraków's universities and local companies.

To learn more about PJUG, visit its website or official meetup page, where you'll also find photos and coverage of past events.

VOXXED DAYS

The New Way to Build and Ship Software

Application development is continually in the process of moving to new paradigms, either at a high level (mainframe, PC, server, web, mobile, cloud) or a low level, such as the endless succession of web frameworks. While the higher-level trends endure, it's not always easy to gauge what will survive among the lower-level technologies. In this issue, we examine the high-level trend, containers, and their low-level use, running microservices.

Containers are the natural evolution of virtual machines and will surely endure long-term. The natural fit between containers and the cloud drives the popularity of both technologies. Microservices could well evolve quickly into something else. The benefits of loosely coupled services are balanced by the complexity of coding, testing, debugging, and deploying them. As the trade-offs are better understood and computing needs evolve, microservices could well morph into a refined version of current models—that are still likely to be housed in containers on the cloud.

We start by showing how to develop a microservice and deploy it in Docker containers (page 15). We then examine the DevOps pipeline for containerized apps (page 32). We compare and combine Java modules and OSGi as intra-application containers (page 42), and finally, we look at Wookiee (page 53), a framework that eliminates a lot of the grunt work in developing microservices.

In addition, we examine what's new in Groovy 3.0 (page 61), continue our series on the mechanics of the JVM (page 73), and look at building an API using Spring (page 81). Enjoy!

ART BY WES ROWELL

JOSH JUNEAU

Creating Microservices with Payara Micro

How to build small, lightweight services with Java EE

Microservices have become a very trendy architecture over the past few years. Because the HTML5, JSON, and RESTful web service technologies have matured, they've made it easier to construct applications as small, simple services that communicate with one another to perform a specific task. When orchestrated with other services, they come together to create powerful, decoupled applications.

A design that uses these services together can work well in a single application server environment. However, to truly separate each service from the others, the services need to stand alone within separate application server containers. Payara Micro provides a fully functional Java application server container at a fraction of the size of a standard application server container—a mere 70 MB. In addition, Payara Micro provides several different ways to deploy applications and services, from standard WAR file deployment to executable JAR packaging.

This article covers deployment of microservices from the ground up using Payara Micro. In it, I demonstrate how to get started with Payara Micro by deploying a simple service. I then explain how to create an "Uber JAR" (a JAR file with the JAR you've created plus all its dependencies) and how to deploy to Docker containers. Lastly, I'll cover some custom configuration options for Payara Micro.

The Payara Micro server is compatible with the Java EE MicroProfile, which was described in detail in the November/December 2017 issue of this magazine. Payara Micro provides a more optimized set of APIs for targeting enterprise Java microservices, and it offers portability across multiple MicroProfile runtimes. The following APIs are currently supported in Payara Micro: Bean Validation, Contexts and Dependency Injection (CDI), Concurrency, EJB Lite, JAX-RS, JBatch, JCache, Java Persistence API (JPA), Java Transaction API (JTA), JavaServer Faces (JSF), Servlets (with JSP Standard Tag Library [JSTL], Expression Language [EL], and JSP), and WebSocket.

Getting Started with Payara Micro

To get started with Payara Micro, download the latest distribution from the Payara website. The download comes in the form of an executable JAR file. Once the JAR file has been downloaded, it can be executed using a locally installed Java runtime. In this case, I am using Payara Micro 4.1.2.174, so I can start an instance of the server by executing the following from the command line or terminal:

```
java -jar payara-micro-4.1.2.174.jar
```

As the server is started, output will be generated; once the startup is complete, a message will be displayed to indicate that. Also displayed should be a host, the host port(s), and the HTTPS port(s). These can be used to determine the URL that should be typed into the browser in order to access the server. In most cases, the URL `http://localhost:8080` can be used to access the Payara Micro instance. Note that if any applications have been deployed to the instance, a section of output denoted by "Payara Micro URLs" should contain the URLs for accessing those applications. To stop the instance, simply press CTRL+C together.

At this point, Payara Micro has been started, and it can be used as a simple Java EE application server container by deploying services upon instantiation, which I'll cover next. There is no web-facing administration console per se, but several configuration options can be specified in the terminal to customize the configuration of an instance. Other configurations, such as database access, can be done within the web applications themselves, leaving very little configuration required for the instance.

Developing and Deploying a Simple Service

In this section, I walk through the development of a very basic web service that can be deployed to Payara Micro. Services such as these are typically deployed along with other services and

A true microservice should be self-contained, meaning the application server container is packaged with the service in a portable manner.

work together to power an over-arching application. In this example, I utilize very few APIs, because one of the keys to developing successful microservices is to minimize dependencies and the overall footprint. Thus, this is indeed a simplistic Java EE application, and from this point forward I will refer to this application as a service. This particular service will be used to query a database table and serve the results in XML or JSON format.

To begin, create a new Maven web project using an IDE. My IDE of choice is Apache NetBeans, but there are many other fine choices for Java EE development. I named the service `EmployeeService`, because it will be used to serve employee database records. First, I configure the Maven POM file. The entire project code can be found on GitHub. If you prefer, you can download just the source files for this article from the *Java Magazine* download area.

This particular service will connect to an Apache Derby database, so that dependency has been included, and the database driver will be registered with the Payara Micro instance from within the service configuration. That dependency is shown in this fragment from **Listing 1**:

Listing 1.

```
...
<dependencies>
   ...
  <dependency>
     <groupId>org.apache.derby</groupId>
     <artifactId>derbyclient</artifactId>
     <version>10.14.1.0</version>
   </dependency>
</dependencies>
...
```

Then I choose a database schema to use for the application, create the database objects, and load records with the SQL code in **Listing 2**. [This listing is not shown here, but it is available in the download area or on GitHub. *—Ed.*]

The web.xml file should contain the database configuration, as shown in **Listing 3**.

Listing 3.

```
<?xml version="1.0" encoding="UTF-8"?>

 <web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
      http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
      version="3.1">
   <session-config>
     <session-timeout>30</session-timeout>
   </session-config>
   <data-source>
     <name>java:global/DerbyDataSource</name>
     <class-name>
        org.apache.derby.jdbc.ClientDriver
     </class-name>
     <server-name>localhost</server-name>
     <port-number>1527</port-number>
     <url>jdbc:derby://localhost:1527/acme</url>
     <user>acmeuser</user>
     <password>yourpassword</password>
  </data-source>
</web-app>
```

Because the service utilizes JPA, an entity class needs to be created to map to the `EMPLOYEE` database table. Create the package `org.employeeservice.entity` and then create a class named

Employee. The code in **Listing 4** contains abbreviated source code for the Employee entity.

Listing 4.

```
@Entity
@Table(name = "ACME_EMPLOYEE")
@XmlRootElement
@NamedQueries({
   ...)})
public class AcmeEmployee implements Serializable {

   private static final long serialVersionUID = 1L;
   @Id
   @Basic(optional = false)
   @NotNull

   @Column(name = "ID")
   private Integer id;
   @Size(max = 50)

   @Column(name = "FIRST_NAME")
   private String firstName;
   @Size(max = 50)

   @Column(name = "LAST_NAME")
   private String lastName;

   @Column(name = "START_DATE")
   @Temporal(TemporalType.DATE)
   private Date startDate;

   @Column(name = "AGE")
   private Integer age;
   @Column(name = "JOB_ID")
   private Integer jobId;

   @Size(max = 20)
   @Column(name = "STATUS")
   private String status;

   public AcmeEmployee() {
   }
  // Getters and Setters
}
```

Now, create an ApplicationConfig class inside of the org.employeeservice package and place the REST configuration from **Listing 5** into it. This class bootstraps JAX-RS by setting up an application path of rest, meaning that RESTful web services will be invoked if a URL contains the path /rest/. The class also makes JAX-RS resource classes available for use by the application by returning them via the getClasses() method.

Listing 5.

```
import java.util.Set;
import javax.ws.rs.core.Application;

@javax.ws.rs.ApplicationPath("rest")
public class ApplicationConfig extends Application {
   @Override
   public Set<Class<?>> getClasses() {
      Set<Class<?>> resources = new java.util.HashSet<>();
      resources.add(
         org.employeeservice.AcmeEmployeeFacadeREST.class
      );
      return resources;
```

```
    }
}
```

Create another class in the same package named `AcmeEmployeeFacadeREST`, and add the source code in **Listing 6**. AcmeEmployeeFacadeREST contains RESTful web service methods. The `@Path` annotation makes the services contained within this class available via the path `/rest/acmeemployee/`.

Listing 6.

```
@Stateless
@Path("acmeemployee")
public class AcmeEmployeeFacadeREST {

    @PersistenceContext(unitName = "EmployeeService_1.0PU")
    private EntityManager em;

    public AcmeEmployeeFacadeREST() {

    }

    @GET
    @Produces({MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
    public List<AcmeEmployee> findAll() {
        List<AcmeEmployee> employeeList = null;
        try {
            employeeList =
              em.createQuery("select object(o) from AcmeEmployee o")
                .getResultList();
        } catch (NoResultException e){

        }
        return employeeList;
    }
}
```

The final piece of the puzzle is the `persistence.xml` file, which should be placed into the `resources/META-INF` folder of the project. I've shown it in **Listing 7**. This configuration file makes the database available to the application. In this case, the data source is pointing to an Apache Derby database.

Listing 7.

```
<persistence-unit name="EmployeeService_1.0PU" transaction-type="JTA">
   <jta-data-source>
      java:global/DerbyDataSource
   </jta-data-source>
   <exclude-unlisted-classes>
     false
   </exclude-unlisted-classes>
   <properties/>
</persistence-unit>
```

That's all there is to the service, and it can now be deployed and tested. Payara Micro provides the ability to deploy the service via the command line. To begin, compile the project into a WAR file. I'll refer to the compiled file as `EmployeeService-1.0.war`. At this point, the service can be deployed to Payara Micro at startup by traversing to the same directory that contains the `EmployeeService-1.0.war` file and starting up the server, specifying the `--deploy` option as follows:

```
java -jar payara-micro-4.1.2.174.jar
   --deploy EmployeeService-1.0.war
```

Once started, the service should be available here:

```
http://localhost:8080/EmployeeService-1.0/rest/acmeemployee
```

If the URL is entered into the browser, the RESTful web service method containing a @GET annotation and no specified @Path will be invoked by default.

Deployment Options

More than one service can be deployed by passing the --deploy option as many times as needed to accommodate the number of services being deployed, for example:

```
java -jar payara-micro-4.1.2.174.jar
   --deploy Service-1.war
   --deploy Service-2.war
```

It might be beneficial to deploy an exploded WAR file at times, and it is possible to do that by specifying the root directory path of the WAR file in the --deploy option. In this situation, a file named .reload can be placed into the exploded WAR file's root directory, and the time stamp of the file can be updated to cause a redeployment to occur.

To deploy multiple instances of Payara Micro on a single machine, simply pass the --port option and specify a different port number for each instance. For example, to deploy the EmployeeService to multiple instances on the same host, deploy the service as outlined in the previous section for the initial deployment, and add the --port option for each subsequent deployment:

```
java -jar payara-micro-4.1.2.174.jar
   --deploy EmployeeService-1.0.war
   --port 8082
```

If a service or application has been added to a Maven repository, it can be deployed directly from there. This can be done by specifying the --deployFromGAV option and listing the Maven repository contained within double quotes. If there is more than one repository, separate them with commas:

```
java -jar payara-micro-4.1.2.174.jar
    --deployFromGAV "my-repository, EmployeeService-1.0"
```

By default, when more than one instance is started on the same machine, the instances are clustered via Hazelcast. Clustering means that the session data will be replicated between the two instances. Hazelcast is an in-memory data grid, which Payara uses to implement caching. The output will display a list of the members that have been clustered together:

```
Members [2] {
  Member [192.168.1.32]:5900 -9f8c400c-...-9af203bbeec9
  Member [192.168.1.32]:5901 -32b2bb1f-...-88eb47273d5b this
}
```

[The output was truncated to fit. —*Ed.*] To see a full list of Payara Micro options, specify the `--help` option when executing the Payara Micro JAR file.

Java EE Application as an Executable JAR

A true microservice should be self-contained, meaning the application server container is packaged with the service in a portable manner. One such way to package a microservice is to create an executable JAR file that can be ported across environments, as needed. Payara Micro enables you to create such an Uber JAR by simply specifying the option `--outputUberJar` when executing the Payara Micro JAR, as follows:

```
java -jar payara-micro-4.1.2.174.jar
    --deploy EmployeeService-1.0.war
   --outputUberJar EmployeeService.jar
```

Executing this command will package the application with an instance of Payara Micro. This command does not run the service, but it packages it into a JAR file such that it can be executed via the `java -jar` command. As such, the resulting JAR file is now a portable, executable instance of the service. The resulting EmployeeService.jar file can then be executed by running this command:

More than one service can be deployed by passing the *--deploy* option as many times as needed to accommodate the number of services being deployed.

```
java -jar EmployeeService.jar
```

The Payara Micro instance will then start up, and the application will be deployed to the Payara Micro instance. The application will then be available on port 8080 by default. To change the default port number, there are several options. The first way is to specify the `--port` configuration option, along with any of the other Payara Micro configuration options, when you build the Uber JAR. The second way is to specify the `--port` option when you execute the Payara Micro JAR file, because all of the standard Payara Micro options can be specified at JAR startup.

To package more than one service with a container, simply specify the `--deploy` option multiple times, once for each service, when you create the Uber JAR:

```
java -jar payara-micro-4.1.2.174.jar
    --deploy Service-1.war
    --deploy Service-2.war
    --outputUberJar EmployeeService.jar
```

The deployment of multiple WAR files in a single JAR is the ideal way to package a small application made of multiple services. An Uber JAR can also be created via Maven by using the exec plugin in the package phase. This would be useful for automatically creating a Payara Micro Uber JAR when you build an application within an IDE.

Deploying to Docker Containers

Docker plays an important role in the microservices universe, because it allows you to encapsulate an entire environment inside a portable container. Using this ability with Payara Micro, it is possible to create portable, self-contained microservices at a fraction of the size of Docker containers that contain full-sized application server containers.

Docker is far too broad a topic to cover in this article; to learn more about this, please see one of the many tutorials online. In this section, I will briefly demonstrate how to deploy a Payara Micro container and service to Docker.

One of the most useful ways to deploy Payara Micro containers to Docker images is to utilize a Dockerfile. Because a Dockerfile enables a step-by-step build of an image, it is possible to reuse instructions from an existing image to obtain an operating system and Payara Micro instance, then specify any additional libraries, and finally perform the WAR deployment(s). Listing 8 shows the Dockerfile for this example, which I will explain in detail.

Listing 8.

```
# Using the Payara Micro 5 snapshot build.
FROM payara/micro:5-SNAPSHOT

# Maintainer of the Image
MAINTAINER Josh Juneau "myemail@mycompany.com"

# Downloads the Apache Derby Client library
RUN wget -O
 /opt/payara/deployments/database-connector.jar
  http://central.maven.org/maven2/org/apache/
   derby/derbyclient/10.14.1.0/derbyclient-10.14.1.0.jar

# Sets database connection environment variables
ENV DOCKER_HOST docker.for.mac.localhost
ENV DB_NAME ACME
ENV DB_USER acmeuser
ENV DB_PASSWORD yourpassword

# Adds an application
COPY EmployeeService-1.0.war /opt/payara/deployments

# Default command to execute
ENTRYPOINT ["java", "-jar",
  "/opt/payara/payara-micro.jar", "--addJars",
   "/opt/payara/deployments/database-connector.jar",
   "--deploy",
  "/opt/payara/deployments/EmployeeService-1.0.war"]
```

To begin writing a Dockerfile, simply create a file named `Dockerfile` with no extension within a directory where you will place any required files to be loaded. A Dockerfile generally consists of one or more instructions for building a Docker image. The first instruction must be a `FROM` instruction, specifying the base image from which to build. In this case, I am building from the Payara Micro 5 base; so I use this:

```
FROM payara/micro:5-SNAPSHOT
```

Next, it is a good idea to indicate the maintainer of the file using the `MAINTAINER` instruction. The next instruction (`RUN`, in this example) obtains the Apache Derby JAR file and copies it into a JAR file in the `/opt/payara/` deployments directory of the image. Following that, a few environment variables are specified for connecting to the database in the host environment using the configurations that were placed within the `web.xml` file. It is worth noting that this Dockerfile is set up for use on an Apple Mac running "Docker for Mac," because the environment variable for the host machine is specified for a Mac. If you are deploying to a different OS host, modify the host IP address accordingly.

Along those lines, you should modify the web.xml data-source element shown in Listing 9 to specify environment variables for the pertinent JDBC fields, rather than hardcoding them.

Listing 9.

```
<data-source>
   <name>java:global/DerbyDataSource</name>
   <class-name>
     org.apache.derby.jdbc.ClientDriver
   </class-name>
   <server-name>${ENV=DOCKER_HOST}</server-name>
   <port-number>1527</port-number>
   <url>
     jdbc:derby://${ENV=DOCKER_HOST}:1527/${ENV=DB_NAME}
   </url>
   <user>${ENV=DB_USER}</user>
   <password>${ENV=DB_PASSWORD}</password>
</data-source>
```

Returning to the Dockerfile, there is a COPY instruction to copy the EmployeeService-1.0.war file to the image deployment directory. The final step in the Dockerfile is to start up the Payara Micro instance, using an ENTRYPOINT instruction to deploy the service.

To create the image and start it up, first build an image using the Dockerfile by opening a terminal, traversing to the directory containing the Dockerfile, and issuing the following command:

```
docker build  -t employeeservice:1.0 .
```

Note that there is a trailing dot, which tells Docker that the Dockerfile is in the current directory. Once the image has been built, it can be started by issuing the following command:

```
docker run -d -p 8082:8080 --name employeeservice employeeservice:1.0
```

This command runs the Docker image with the name `employeeservice` and the tag `1.0`. The `-d` option specifies that the image should be run in detached mode, and the `-p` option maps the container's port 8080 to the host port of 8082. The Docker images that are currently built and ready to run can be listed by using the `docker images` command, and the containers can be listed by using the `docker ps` command. To remove a Docker container, issue the command `docker rm <<containerId>>`, and to remove a Docker image, issue the command `docker rmi <<imageId>>`, where the commands within double arrowheads indicate a dynamic variable.

Introduced in Payara Micro 4.1.2.161, the HealthCheck API provides a self-monitoring capability to automatically report issues or future problems so that they can be acted upon.

The `EmployeeService` should be running in the Docker container after issuing the `run` command. Therefore, it can be accessed by entering the following URL into the browser: `http://localhost:8082/EmployeeService-1.0/rest/acmeemployees`. (Note: You must have an Apache Derby database running on port 1527 of the local host if you are using the example service for this article; otherwise, the startup of the JAR file might throw errors.)

Health Checking

Introduced in Payara Micro 4.1.2.161, the HealthCheck API provides a self-monitoring capability to automatically report issues or future problems so that they can be acted upon or prevented. By default, the HealthCheck API is disabled, but it can be enabled programmatically by configuring it within a separate application that is deployed to the Payara Micro instance along with any other applications or services. Ideally, the configuration application should be packaged as a singleton EJB, which will start up upon deployment.

To enable HealthCheck, develop the configuration application by creating a Maven web application that includes the following dependency:

```
<dependency>
   <groupId>fish.payara.extras</groupId>
   <artifactId>payara-micro</artifactId>
   <version>4.1.2.174</version>
   <scope>provided</scope>
</dependency>
```

The PayaraMicroRuntime object can be used to perform the configuration, and an instance of it can be obtained by calling the `PayaraMicro.getInstance().getRuntime()`. Thus, the PayaraMicroRuntime object can then be used to configure options, such as how often to perform and log health checks to the system log, which metrics to monitor (memory, CPU, and so on), and specifying logging levels. In this case, I wish to enable health checking. To do so, I specify the following configuration within the singleton EJB class:

```
final PayaraMicroRuntime pmRuntime =
  PayaraMicro.getInstance().getRuntime();
pmRuntime.run("healthcheck-configure",
  "--enabled=true", "--dynamic=true");
```

The sources shown in **Listing 10** (available online) demonstrate what a simple Payara Micro configuration class may look like. In this case, the configuration will be set up to enable health checking and machine memory logging every 30 seconds. Simply including this class as the sole class of a Maven web application will allow for external configuration of a Payara Micro instance. You can then deploy the configuration WAR and subsequent WAR files by invoking the standard Payara Micro deployment sequence:

```
java -jar payara-micro-4.1.2.174.jar
   --deploy MicroConfig.war
   --deploy EmployeeService-1.0.war
```

Note that the configuration WAR is specified first in this example. Doing so allows for configuration to take place prior to deployment of any applications or services.

External Libraries

If external libraries are needed to execute services, the `--addJars` option can be specified, as of Payara Micro 5 as well as Payara Micro 4.1.2.173. To package one or more JAR files with a Payara Micro container, specify one or more colon-separated JAR files or directories after the `--addJars` option. If a directory is specified, all JAR files within that directory will be added. This option can be combined with the `--outputUberJar` option to package everything up into an Uber JAR.

Conclusion

Payara Micro packs a lot of power into a small package. By taking advantage of its versatility and ease of use, you can develop powerful microservices that can be deployed just about anywhere. </article>

Josh Juneau is a Java Champion, application developer, system analyst, and database administrator. He writes regularly for *Java Magazine* and the Oracle Technology Network and is the author of several books on Java and Java EE published by Apress. He was a member of the JCP Expert Group for JSR 372 and JSR 378 and is a member of the NetBeans Dream Team.

MICHAEL HÜTTERMANN

DevOps with Container-Based Delivery Pipelines

A real-world pipeline for automating delivery of a containerized Java EE app to the cloud with a Jenkins-based toolchain

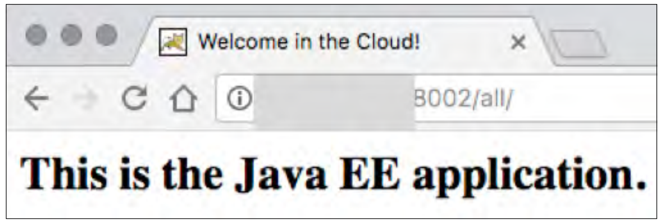
In this article, I examine how to implement DevOps using the widely used Jenkins tool, Docker containers, and a cloud-hosted instance of Java EE. *DevOps* is a term invented in 2009 to emphasize the cooperation of developers and operations personnel in building and deploying applications. DevOps is often focused on shortening *cycle time;* that is, making it easier to push out new product releases quickly.

How does DevOps relate to other disciplines? *Continuous delivery* is the capability to make changes to a product available frequently, whereas *continuous deployment* refers to the process of effectively bringing those changes to the end user. *Continuous inspection* and *continuous integration* are building blocks of continuous delivery. Continuous inspection emphasizes that high quality is always mandatory. Continuous integration is the practice of checking in changes to a version control system several times a day and ensuring that the code in version control can be checked out anytime and builds successfully. Continuous delivery is the prerequisite for continuous deployment and, in turn, DevOps is the prerequisite for continuous delivery.

The Basic Use Case

Because my colleagues and I do not want to reinvent the wheel, we try to share good practices and synergies on toolchains. So, centralized tools—above all, the automation engine Jenkins—are the foundation of our DevOps initiative.

This means that from the moment we push a change to our version control system (in our case, Git), the overall process of testing, packaging, containerization, staging, and promotion is automated with Jenkins. While one of the building blocks of DevOps is automation, manual steps do not conflict with DevOps, and they are often meaningful and sometimes mandatory.



[redacted]8002/all/

This is the Java EE application.

Figure 1: The start page of the simple application used in this article

In this example, I deploy a Java EE web application, running OpenJDK on Alpine Linux, bundled with the widely used Apache Tomcat, and deployed to the cloud. While this application could be deployed on many clouds, I've chosen to run it on Oracle Cloud. As expected, I use Docker containers driven by a Dockerfile.

Running the application displays the start page shown in Figure 1.

I organize the processing of the upcoming changes with pipelines, which I discuss next.

Pipelines to Workflows

On its way toward production, the code changes will go through different stages. *Staging* (also called *promotion*) is the activity of consistently transferring a defined baseline of the software with all its configuration items from one stage to another. Software is staged over different environments by configuration, without rebuilding. A *delivery pipeline* is a set of stages together with transition rules between those stages. From a DevOps perspective, a pipeline bridges multiple functions in organizations, above all development and operations.

A change typically waits to be pulled to a succeeding stage for further processing according to the transition rules, which are aligned with defined requirements that must be met. These are the *quality gates*. In bigger and more-complex setups, multiple pipelines form a *workflow*. Parts of the workflow are glued together and run automatically, and other pipelines are triggered manually.

Figure 2 shows a typical workflow that maps a code change to a release build. The workflow is made up of different pipelines with quality gates in between (the lock in the figure). The approved change request enters the workflow (illustrated top left) and leaves the workflow by delivering the change to the end user (bottom right). The code in the change is built continuously, and the dev build promotes it to be a defined development version, a release candidate

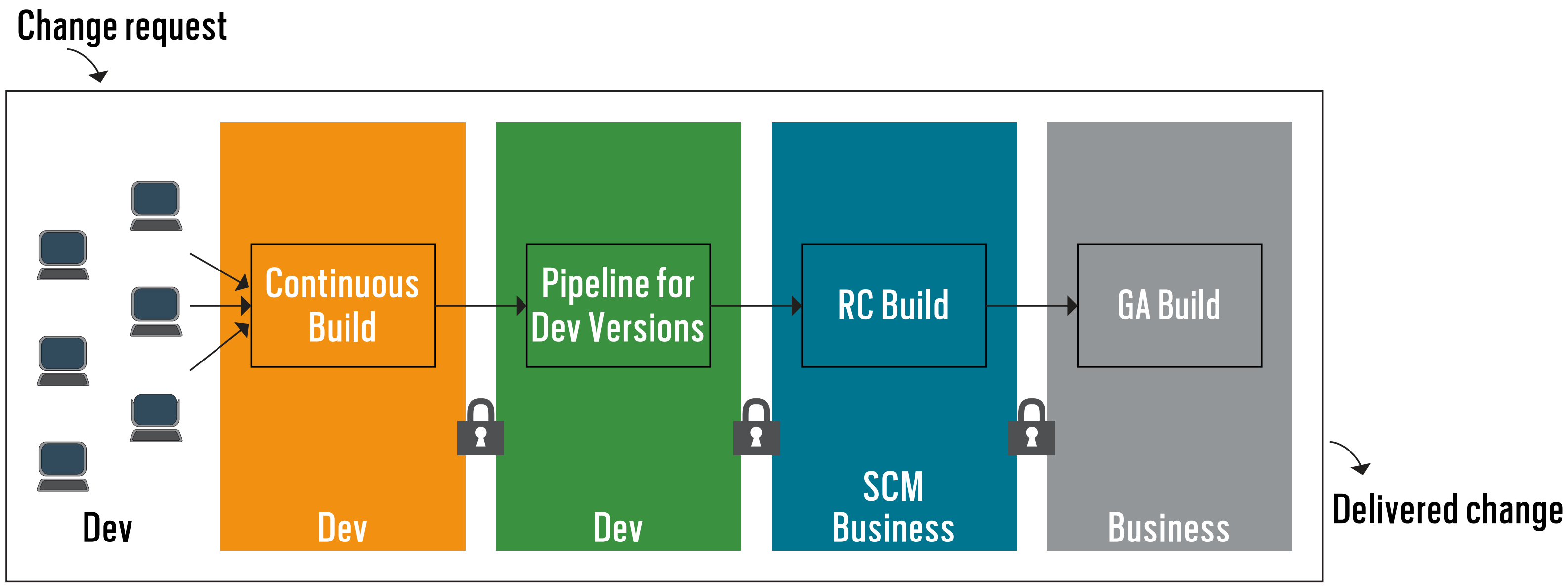


Figure 2: The workflow pipelines that move a code change to a release build

(RC), and finally a general availability (GA) version. Major stakeholders include development (*Dev* at the bottom of the first three columns), a business representative (*Business*), or a software configuration management function (*SCM*). I discuss this scenario in more detail in the following sections.

The first stage is the developer workspace, where developers design, write, and test code before checking the code changes into a version control system. After checking the code in, downstream Jenkins pipelines are triggered. The continuous build of the second stage is just a fast-running job to check whether the changes, together with the existing code, compile and whether some basic unit tests run successfully. Continuous integration refers to these first two stages.

The next stage is the "dev build" for producing development versions. This stage takes a meaningful set of commits and builds a working implementation of the product. It passes through all the stages of build, verification, and creation of a deliverable in a container.

In my case, this pipeline takes a Maven snapshot and derives a Maven-based release, packages the WAR file, inspects the code (I use SonarQube for static code analysis), checks whether the generated WAR file is deployable, packages the WAR file into a Docker image, and checks whether the Docker container can be run and, thus, start the web application. If so, it transfers the WAR file as well as the Docker image to my company's binary repository manager, which is hosted with JFrog Artifactory. Artifactory serves as a Docker registry to manage our Docker images. The WAR file and Docker image are placed in the repository on Artifactory, and I can navigate from Jenkins to Artifactory and back for traceability.

Listing 1 shows a snippet from a pipeline script I wrote in Groovy that shows how to bring binaries from Jenkins to JFrog Artifactory, while adding context information. See the Jenkins pipeline for details.

Listing 1: Excerpt of a pipeline script to move Docker images to JFrog Artifactory and label them with metadata

```
def artDocker =
  Artifactory.docker server:server, host: "tcp://127.0.0.1:1234"
artDocker.addProperty("eat", "pizza").addProperty("drink", "beer")

def dockerInfo =
  artDocker.push(
     "$ARTI3REGISTRY/michaelhuettermann/alpine-tomcat7:${version}",
     "docker-local")
buildInfo.append(dockerInfo)
server.publishBuildInfo(buildInfo)
```

In DevOps, *left shifting* and *right shifting* describe moving activities from the classic software development approaches to earlier in the pipeline (left shift) or later in the pipeline (right shift). In my case, the left shift means packaging production-ready containers early in the process, and the right shift is putting Dockerfiles and other configuration items into version control as part of the same baseline business application.

The Jenkins pipeline is set up with a Jenkins pipeline visualization and creation feature called Jenkins Blue Ocean and its domain-specific language (DSL). This strongly overlaps with what is often called continuous integration. SonarQube is also triggered by DSL, and defining a SonarQube-based quality gate is shown in **Listing 2**.

Listing 2: Quality gate as part of the SonarQube processing

```
timeout(time: 2, unit: ‘MINUTES’) {
  def qg = waitForQualityGate()
  if (qg.status != ‘OK’) {
    error "Pipeline aborted ... ${qg.status}"
  }
}
```

My pipeline also contains a test of the Docker image, before it is pushed to Artifactory. For that, the Docker image is run, and the resulting Docker container makes available the bundled web application—exactly the one I want to promote to production later. I can now utilize Selenium to check whether the web application started correctly and has the expected behavior. I have posted a visualization of the pipeline run online.

Now let’s move to the next step: deriving a release candidate (RC) for the artifacts that have been created.

Release Candidate Pipeline

The next pipeline is the pipeline to create the release candidate. This is the last stage before release, so here I am concerned with making sure the binaries are packaged, well-tested, and containerized. This is done by cherry-picking an existing development version. Typically this is the most recent version created in the previous pipeline, but often there are reasons to not take that one—for instance, if a regression was introduced. (In other words, not every single dev version must be promoted to be a release candidate.) On big or complex projects, at this stage you often have multicomponent setups or framework development.

In this example, the RC pipeline promotes the WAR file and the Docker images to dedicated staging repositories in Artifactory. Context information is often added to the binaries as part of this stage—for example, ticket numbers or a new owner of the binary expressing changed responsibilities. You can find my Groovy script for this pipeline on GitHub.

General Availability Pipeline

The next stage in the overall workflow is the pipeline to derive general availability (GA) versions. Also, the idea is to promote the binaries, apply different quality gates, add context information, and do further testing. In practice, this is often done by a software configuration management team or release team. In my example, the original binaries are promoted again, this time from JFrog Artifactory to JFrog Bintray.

This is an example of how to utilize a different Docker registry. I've included it to demonstrate a heterogeneous setup. In practice, many platforms and tools are used, reflecting different functions, ownership, and responsibilities in the company. This example stresses that I use JFrog Artifactory as the Docker registry for my release candidates and JFrog Bintray as the tool that serves as a Docker registry for my general-availability binaries (see **Figure 3**, taken from my Jenkins Blue Ocean workflow). The script for this pipeline, again in Groovy, is available online.

Once the Docker image is pushed to JFrog Bintray, consumers can apply native Docker commands on those images, such as for pulling the Docker images for further usage. The consumer in this example is Oracle Cloud, because I want to run and orchestrate containers in the cloud, which I cover in the next section.

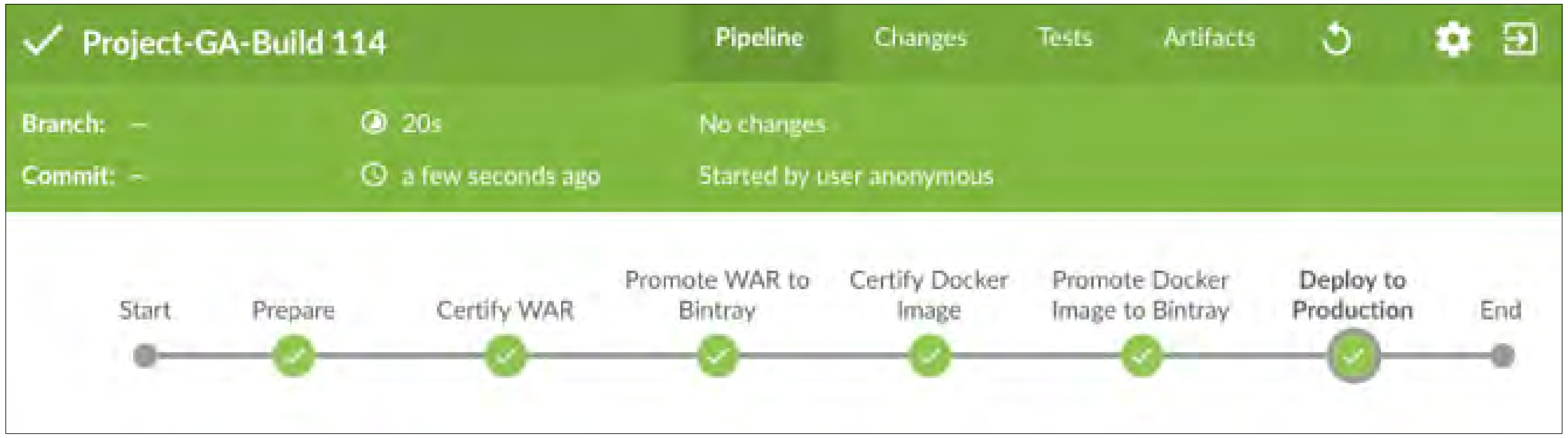


Figure 3: Pipeline for GA versions

Provisioning Docker Containers in the Cloud

Docker is the de facto standard for moving applications from development to production in a reliable way, because it enables you to ship isolated, well-defined services across boundaries, such as from on-premises data centers to the cloud. In addition, Docker can be easily integrated with configuration management systems, such as Puppet (see my previous article). The emerging computing model today, however, is not only about moving around Docker images and containers, but about orchestrating containers (including making up solutions by grouping single containers, such as an application server and a load balancer), managing resources (including hosts and network), providing lifecycle operations on a set of containers (including "self-healing" by automatic restart upon failure), and using considerably more functionality (including scaling; service discovery; and a full-fledged, well-documented API).

An easy-to-use platform for achieving this and hosting environments, including production environments, is Oracle Cloud Infrastructure Container Service Classic. This platform-as-a-service (PaaS) software is part of the Oracle Cloud offering. Its foundation is *manager nodes* and *worker nodes.* The former orchestrate the deployment of containers to the latter. Container services run on *hosts*. Hosts are further organized into *resource pools*, which are a collection of hosts on which to place the containers for a service. An Oracle Cloud Infrastructure Container Service Classic container service defines a Docker service together with the necessary configuration settings for running a Docker image and its deployment rules. With the Oracle Cloud service, you can construct a command to run a service with a graphical wizard or by using a plain-text field for entering the Docker command. The Oracle service internally uses a YAML configuration, which can be edited as well. Services can be linked together and started as a semantic group called a *stack*. The Oracle Cloud Infrastructure Container Service Classic console provides many default services and stacks, so getting started quickly is easy.

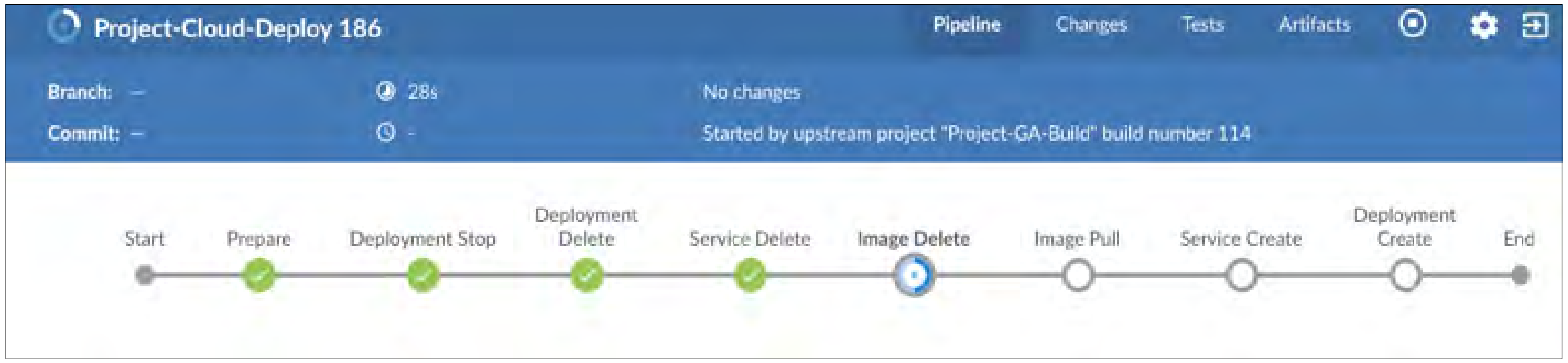
A good software solution should be decoupled from the underlying platform.

In this context, a service is a manageable, deployable unit on which cross-functional teams can work across its lifecycle. A good software solution should be decoupled from the underlying platform. That is the reason that using the Oracle Cloud service as a platform can serve as the vehicle with which to manage and orchestrate all your Docker containers and solutions composed with them.

In my scenario here, the packaged and tested Docker container is pushed as a GA version to a Docker registry on JFrog Bintray. After defining this Docker registry inside Oracle Cloud Infrastructure Container Service Classic, the Oracle service can easily pull images from there.

For demo purposes, I stop a possibly existing demo deployment, delete it, and delete the corresponding services and Docker image in Oracle Cloud Infrastructure Container Service Classic. It is easy to enhance the solution by just providing respective new versions and retiring the running ones. Afterwards the pipeline can pull the image, in its new version; create a new service; and deploy the service. **Figure 4** shows the sequence of included stages (captured while the image deletion is in progress).

Jenkins is the automation engine; it is not the Docker orchestrater. Thus, I use the Oracle Cloud API to work on the respective goals. The sequence of stages is secured by Jenkins' credentials handling and partly uses centralized scripts, managed as Jenkins *shared libraries*. The calls to Oracle Cloud are done with curl, according to the defined API. The code snippet shown in **Listing 3** creates a new service on Oracle Cloud Infrastructure Container Service Classic, for


Project-Cloud-Deploy 186

Branch: –

Commit: –

28s

No changes

Started by upstream project "Project-GA-Build" build number 114

Figure 4: Pipeline for cloud deployment: the streamlined sequence of automation steps for Oracle Cloud

the given parameterized version, based on the newly created image.

■ **Listing 3:** Creating a new service (indented lines are continuations of the previous line)

```
curl -ski -X "POST"
  -H "Authorization: Bearer ${BEARER}"
  "https://${CLOUDIP}/api/v2/services/"
   --data "@${WORKSPACE}/new-service.json"
```

The command uses a JSON file as input, and is available online.

After the service is created, I need to create a new deployment from the service. The deployment is a running instance of the service. The command for creating a new deployment on the Oracle Cloud service is shown in **Listing 4**.

■ **Listing 4:** Creating a new deployment (indented lines are continuations of the previous line)

```
curl -ski -X "POST"
  -H "Authorization: Bearer ${BEARER}"
  "https://${CLOUDIP}/api/v2/deployments/" \
   --data "@${WORKSPACE}/create-deployment.json"
```

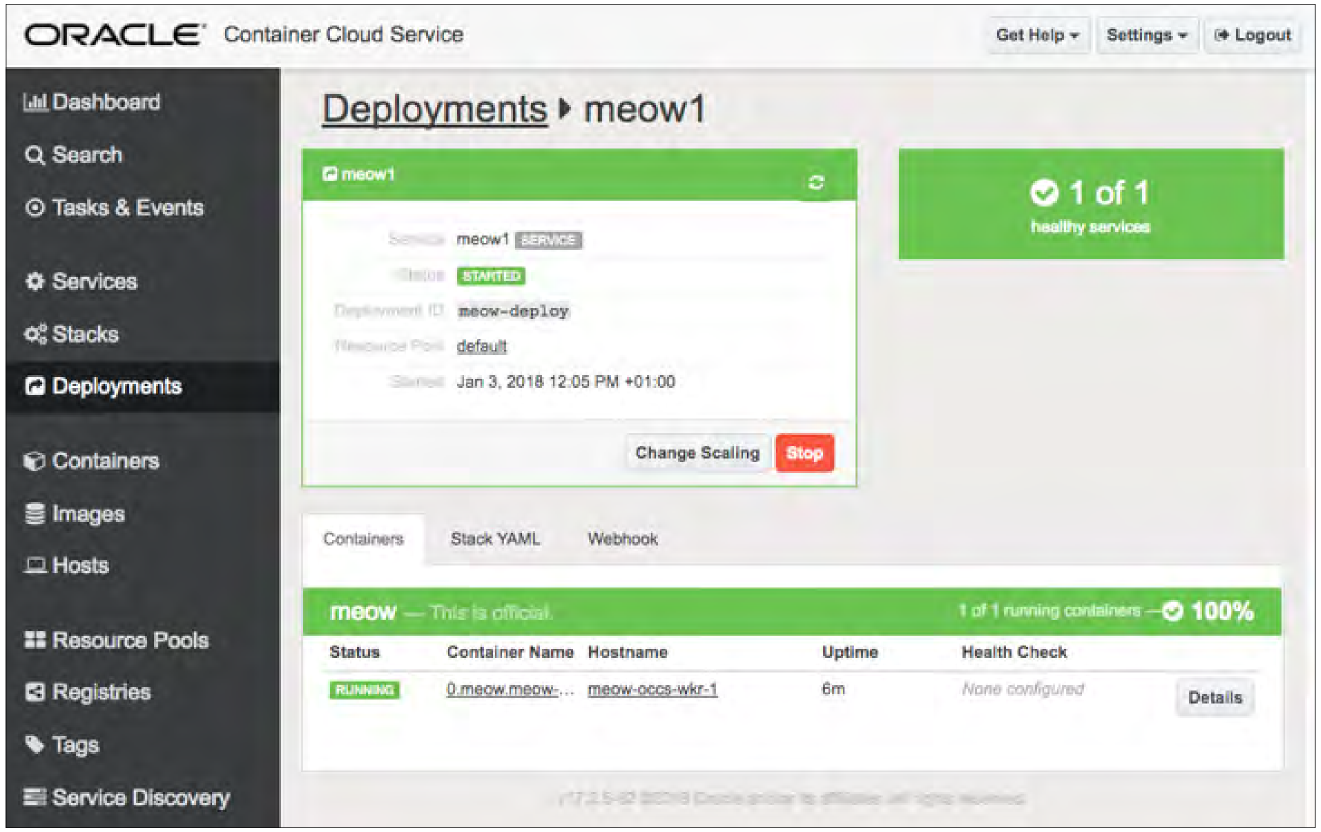
The command again uses a generic JSON file as input.

After the pipeline is completely processed (you can find the complete pipeline file here), the container based on the new version of the Docker image runs and can be inspected in the cloud console (see **Figure 5**; don't be surprised by the names—I like cats).

The change is now live and available in the cloud. Now you have to check the public web page to determine whether the new content is really available. **Figure 6** shows that it is working correctly.

Conclusion

This completes the journey of bringing code changes into production—based on tools such as Jenkins and Docker while emphasizing DevOps concepts and tools. This article shows how



Deployments ▸ meow1

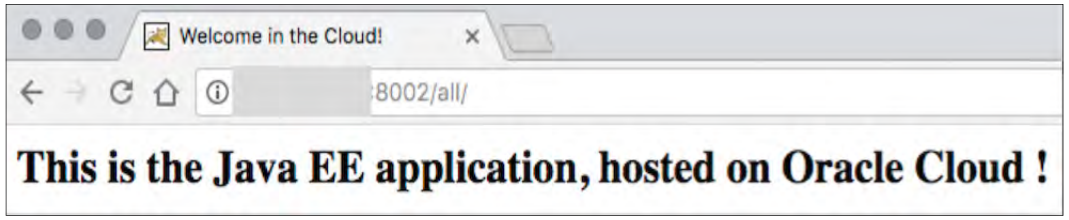
1 of 1 healthy services

Service	meow1 SERVICE
Status	STARTED
Deployment ID	meow-deploy
Resource Pool	default
Started	Jan 3, 2018 12:05 PM +01:00

meow — This is official. 1 of 1 running containers — 100%

Status	Container Name	Hostname	Uptime	Health Check
RUNNING	0.meow.meow-...	meow-occs-wkr-1	6m	None configured

Figure 5: Visualizing and working on deployments in the Oracle Cloud Infrastructure Container Service Classic console



[redacted]8002/all/

This is the Java EE application, hosted on Oracle Cloud !

Figure 6: Checking whether the new content is available

deployment via pipelines and stages transitions the changes and deploys them to the cloud following typical best practices. </article>

Michael Hüttermann is a Java Champion and an expert in continuous delivery, DevOps, software configuration management, and application lifecycle management. He has written four books, including *DevOps for Developers* (Apress) and *Agile ALM* (Manning Publications). In 2017, he was named an Oracle Developer Champion.

ERIC J. BRUNO

Working with OSGi and Java 9 Modules

Integrating two module systems whose benefits complement each other

As many developers have learned, code modularity is more than just defining Java packages. It's about specifying precisely which code is imported, which is shared publicly, and which is to be kept hidden. Proper modules also define understandable contracts that give insight into how shared code will behave and interact (Figure 1).

With Java SE 9, you can now modularize both Java applications and the Java platform itself. Proponents of the Open Service Gateway Initiative (OSGi) have been modularizing Java applications since the early 2000s, but there are key differences between the two approaches. In this article, I'll compare OSGi and Java 9 modularity, enumerate the strengths of each, and conclude with an example of how to use them together.

Although Java 9 implements much of the modularity that OSGi offers, there are cases where OSGi fits especially well. Examples include some private cloud implementations, Internet of Things (IoT) solutions (especially for device-side IoT gateways), applications that fit a

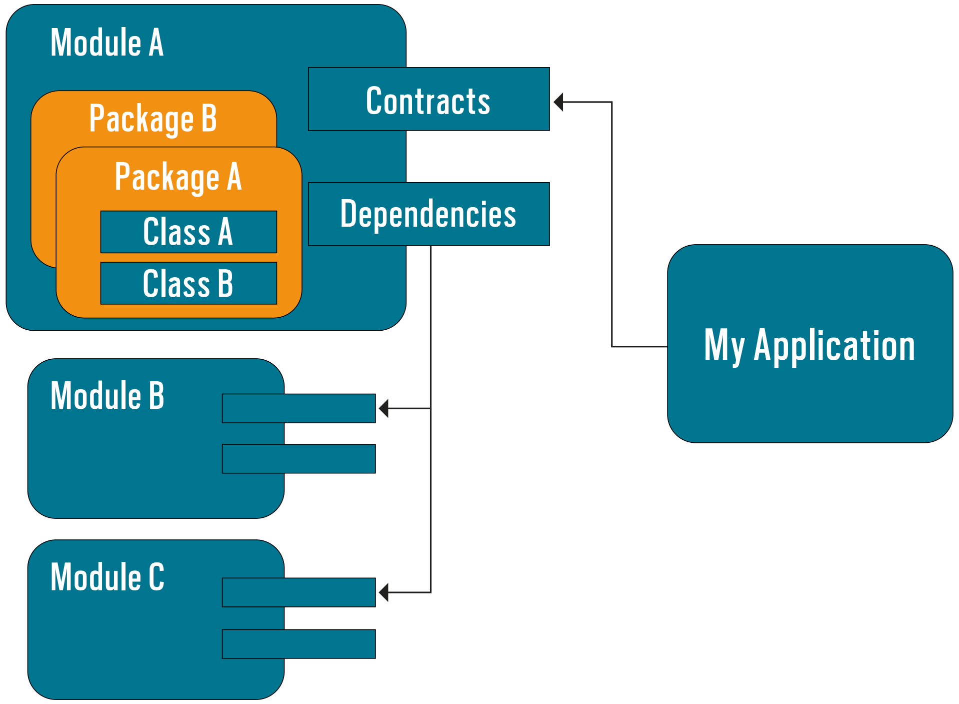


Figure 1: Modularity defines precisely how components are published and used, along with contracts for proper usage.

plugin model (such as the Eclipse IDE), and those designed for ongoing extensibility. Still, the Java 9 module system has an advantage in that it benefits from Java compiler support, which OSGi does not.

Instead of debating the two approaches to modularity, in my opinion it's better to focus on the isolation each allows from the component, application, and JVM perspectives. This article explores integrating OSGi and Java 9 modules into an ideal solution that uses the best features of both options.

What Is OSGi?

OSGi began as JSR 8 in the late 1990s and has seen its share of political and technical challenges. OSGi is not just a module system. It's an entire platform and dynamic component model for Java. With it you can remotely install, activate, deactivate, and upgrade Java components or entire Java applications without requiring OS reboots or even a restart of the JVM. It goes so far as supporting the remote download and installation of Java components and supports management policies that, taken together, form a full application lifecycle model.

Although dynamic control of components might not be necessary for enterprise applications, it is useful in the world of IoT and embedded systems. However, you're free to ignore the dynamic parts of OSGi and just take advantage of its modularity. Modules in OSGi are defined by bundles, each of which comes with a custom classloader. As a result, private

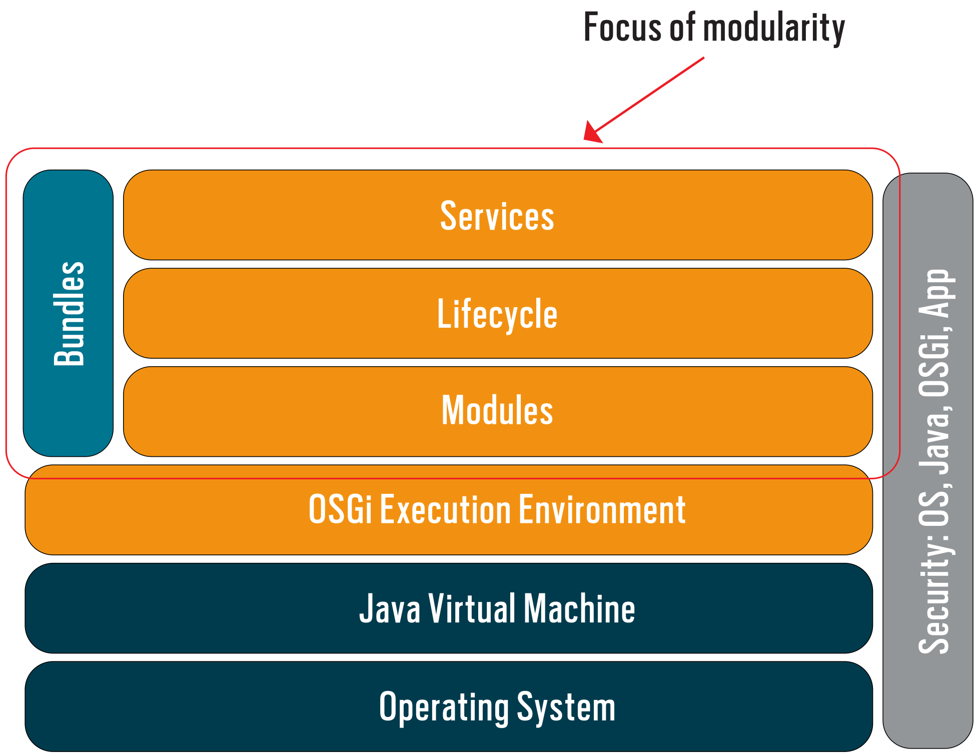


Figure 2: OSGi provides a layered platform for module definition and dynamic component control.

internal classes are not directly visible outside the bundle, thereby increasing true isolation.

Further, objects within bundles are shared through services (**Figure 2**), defining strict contracts for component usage and communication while further hiding internal implementation. The result of this isolation is reduced complexity, increased transparency into the running system, and greater component reuse.

In **Figure 2**, which purposely differs from the typical OSGi layer illustration, the main section where modularity is defined is outlined in red. Security is shown as a shared responsibility involving the OS, the JVM, the OSGi environment, and your application code. No single component can be completely responsible for security. **Listing 1** shows a sample OSGi bundle for a partial math library to be used in a simulator application (four files are shown in the listing).

Listing 1: Simplified sample OSGi bundle implementation

```
package example.simulation.math;
public interface SimulationMath {
   double degreesToRadians(double degrees);
   double getTargetAngle(double centerOffset,
                 double targetDistanceZone);
   //...
}

package example.simulation.math.impl;
import example.simulation.math.SimulationMath;
public class MyMathImpl implements SimulationMath {
   public double degreesToRadians(double degrees) {
      //...
      return radians;
   }

   public double getTargetAngle(double centerOffset,
                      double targetDistanceZone) {
      //...
```

```
     return angle;
   }
   //...
}

package example.simulation.math.impl;
import example.simulation.math.SimulationMath;
import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;
import org.osgi.framework.ServiceRegistration;

public class MathActivator implements BundleActivator {
   SimulatorMath mathImpl = new SimulatorMathService();
   ServiceRegistration registration;

   public void start(BundleContext bc) throws Exception {
     registration =
       bc.registerService(
          mathImpl.class.getName(),
         requestResponse, null);
   }

   public void stop(BundleContext bc) throws Exception {
     registration.unregister();
   }
}
```

```
Manifest-Version: 1.0
Bnd-LastModified: 1516396255825
Build-Jdk: 9.0.1
Built-By: ericjbruno
```

```
Bundle-Activator: example.simulation.math.MathActivator
Bundle-ManifestVersion: 2
Bundle-Name: math
Bundle-SymbolicName: simulation.math
Bundle-Version: 1.0.0
Created-By: Apache Maven Bundle Plugin
Export-Package: example.simulation.math;version="1.0.0"
Import-Package: example.simulation.math;version="[1.0,2)",
   org.osgi.framework;version="[1.5,2)"
Require-Capability: osgi.ee;filter:="(&(osgi.ee=JavaSE)(version=1.6))"
Tool: Bnd-3.5.0.20170929184
```

With OSGi, you code mostly through plain old Java interfaces and objects, as the `SimulationMath` and `MyMathImpl` illustrate in this set of listings. The notable addition is an OSGi bundle activator specific to the simple math library bundle. The OSGi framework creates instances of the `MathActivator` class to load the bundle, start it, and later stop it from running, as dictated by the OSGi lifecycle. The addition of a bundle activator and a short manifest file is all that's needed to plug into the OSGi framework.

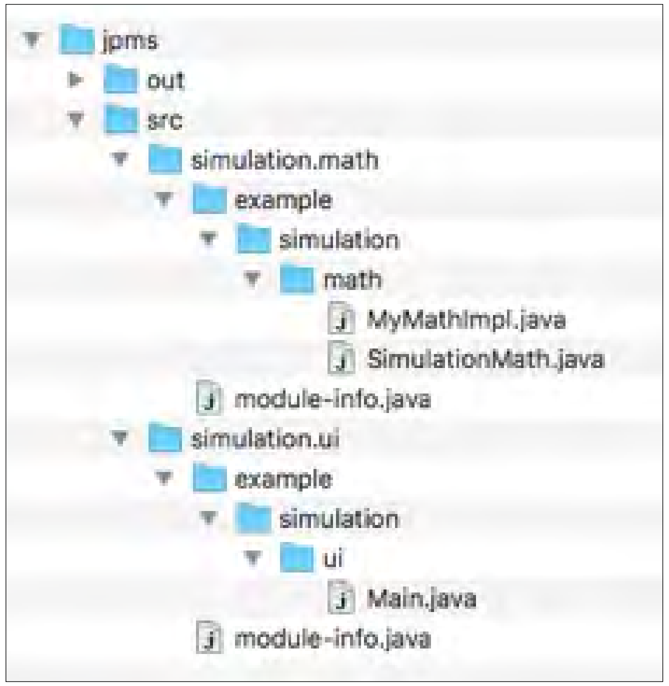
Systems can be formed and orchestrated securely by dynamically assembling and controlling sets of components. If OSGi offers a solution for Java modularity, how does Java SE 9 fit in?

An Overview of Java Modules

Although there's more to Java SE 9 than just modularity, Java modularity is certainly the highlight of the release. Its use was described in detail in a previous issue of this magazine. Java SE 9 comes with built-in modularity for the Java class libraries—broken into layers—and the ability to modularize your own application. You can break your applications into modules, presumably to increase reuse across applications or to support future extensibility. This increases robustness, because the JVM validates the classpath to ensure all dependencies are present at compile time.

As part of defining application modules and specifying which Java class libraries to import, you can modularize the JVM itself. The result is a custom Java runtime that strips out everything except what your application requires, reducing the overall JVM and application footprint.

As with OSGi, modularity, from a size perspective, might not be much of an issue for enterprise applications, but it's a big issue in the world of embedded and IoT development—both very important business topics today.



```
jpms
  out
  src
    simulation.math
      example
        simulation
          math
            MyMathImpl.java
            SimulationMath.java
      module-info.java
    simulation.ui
      example
        simulation
          ui
            Main.java
      module-info.java
```

Figure 3: Example showing Java modules available in the sample simulation application

Similar to OSGi, the Java module system requires you to specify the modules your application is dependent upon (via the new `requires` keyword) and those your application or module exports for others to use (Figure 3). How it differs is in its implementation. With the Java module system, you define a module-info.java file that outlines all the dependencies, and the Java compiler enforces the modularity and validates that all of the dependent modules are present at compile time.

Listing 2 shows the module-info.java files where modularity requirements are defined for the sample application, which consists of a user interface and math library implementation, matching what's shown in Figure 3.

Listing 2: Two sample module source files—one for the math package and a second for the application's UI

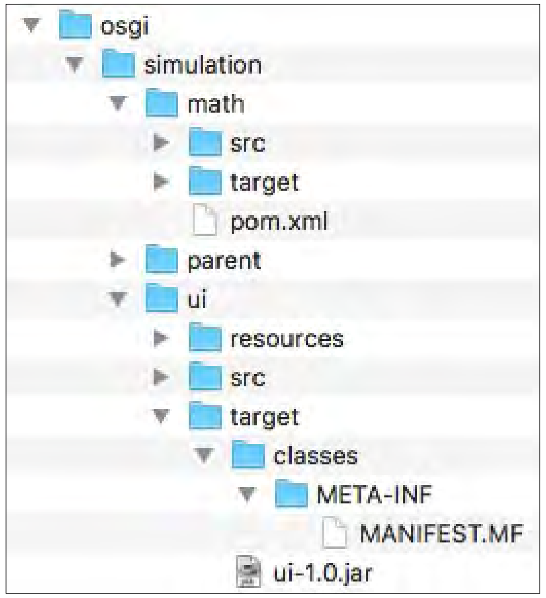
```
// simulation.math/module-info.java
module simulation.math {
   exports example.simulation.math;
}

...
```

```
// simulation.ui/module-info.java
module simulation.ui {
   requires simulation.math;
}
```

For comparison, **Figure 4** shows the OSGi file structure for an application similar to the Java module version shown in **Figure 3**.

Given this overview of both OSGi and Java modules, let's examine their strengths and differences.



```
▼ osgi
  ▼ simulation
    ▼ math
      ▶ src
      ▶ target
        pom.xml
    ▶ parent
    ▼ ui
      ▶ resources
      ▶ src
      ▼ target
        ▼ classes
          ▼ META-INF
              MANIFEST.MF
          ui-1.0.jar
```

Figure 4: Example showing OSGi modules available in the sample simulation application

OSGi and Java Modules Compared

OSGi creates a plugin model for applications—one such common application is the Eclipse IDE. OSGi's success is largely due to its support for dynamic component control. In this case, plugins or components (modules defined as OSGi bundles) are loaded dynamically and then activated, deactivated, and even updated or removed as needed. Presently, this dynamic module lifecycle is not available with Java modules.

Additionally, compared with Java modules, OSGi supports improved versioning. Other OSGi advantages are related to isolation. For example, bundle changes require only the direct dependencies to be recompiled, whereas a Java module's entire layer—along with all child layers—needs to be recompiled if just one module changes (**Figure 5**).

Enforcing proper coding practices, Java modules help to hide internal implementations (private classes) from usage, and even from reflection, which is an improvement over earlier versions of Java. OSGi cannot enforce this. However, although the ability to use internal classes within OSGi bundles using reflection can be considered a weakness, it does allow for dependency injection of private classes, which is more difficult, if not impossible, with Java modules.

The downside is that OSGi bundles still suffer from classpath issues, such as runtime exceptions for missing dependencies, or arbitrary class loading for packages with the same name (also referred to as *split packages*). Additionally, OSGi requires a classloader per module, which can affect some libraries that expect only a single classloader. Java modules don't allow split packages—which is considered a big improvement in Java overall—and don't have similar classloader requirements or restrictions.

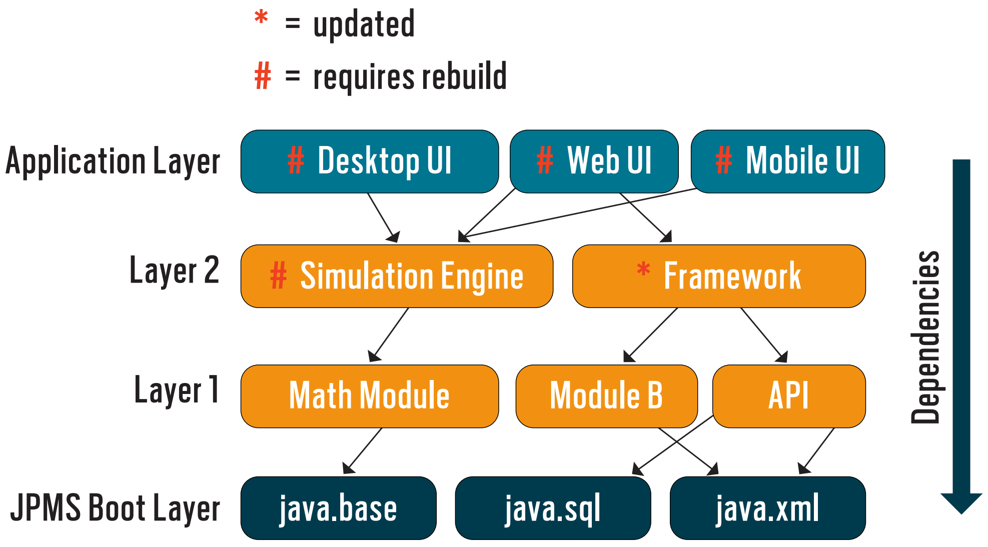


Figure 5: Java module layers and dependencies propagate changes, whereas OSGi isolates them.

One big advantage Java modules have over OSGi is compiler support, as illustrated previously in **Listing 1**. Because the JDK is now modularized, it can reveal classpath issues at compile time and can be used to create custom JVMs and runtimes to reduce the total footprint and deployment size. OSGi provides no control over the JVM or its libraries.

IoT support is a strength of *both* OSGi and Java modules, but for different reasons:

- OSGi offers excellent versioning and lifecycle support for modules: they can be remotely installed, turned on and off, and later updated or removed altogether.
- Java modules allow you to create custom, modular JDKs and JVMs for reduced footprint and deployment size.

For IoT, mobile applications, and embedded development support, to name a few, it makes sense to integrate OSGi and Java modules.

Integrating OSGi and Java Modules

As I explained earlier, OSGi and Java modules complement one other, even if they weren't explicitly designed to. The overall strategy is to use Java modules to modularize libraries (either imported or exported) and the JVM itself, and use OSGi on top to handle application modularity and dynamic lifecycle control. Neither technology can offer as complete a solution on its own, and the combination extends modularity support from top to bottom (Figure 6).

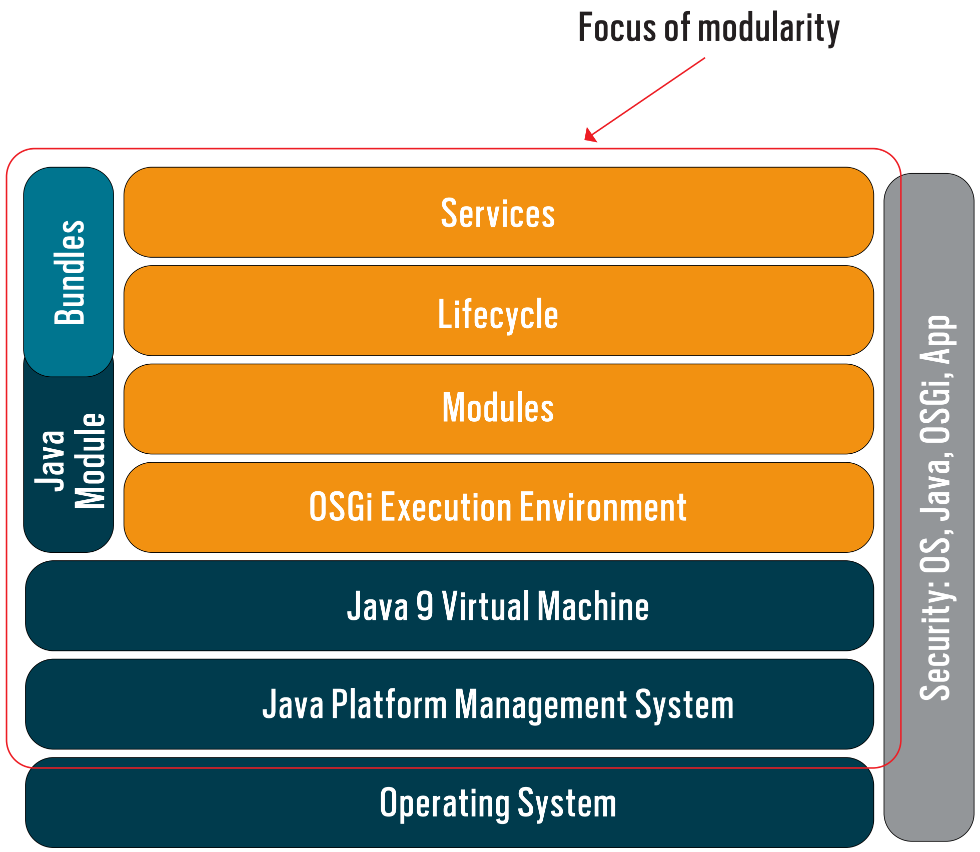


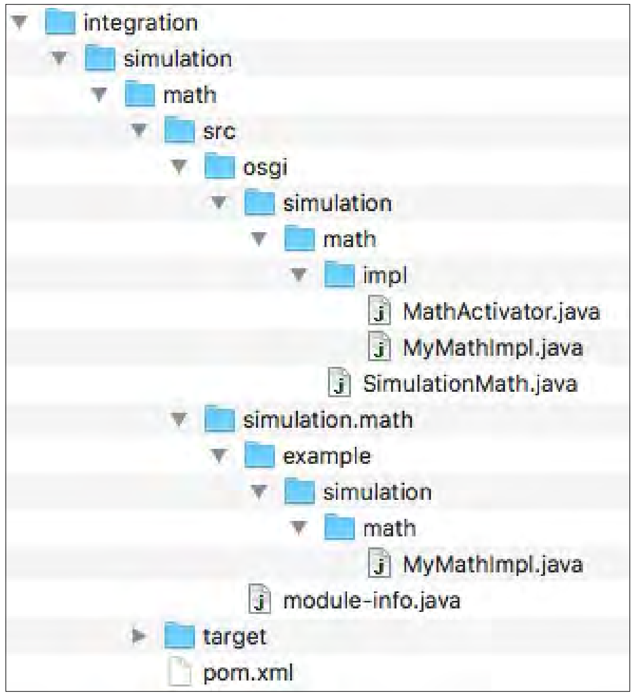
Figure 6: When OSGi is layered on top of JPMS, modularity is extended from the application, to libraries, through to the JVM itself, with full lifecycle control.

To ensure that existing Java applications can run with Java 9 as is—that is, without explicitly defining modules for existing code—the JVM by default runs in compatibility mode. It does this by creating an unnamed module in which to place nonmodular application code. When OSGi is run on Java SE 9 in compatibility mode, the JVM automatically creates an unnamed module per OSGi bundle, service, and execution environment. However, explicit Java modules (those designed to be real Java 9 modules *not* in compatibility mode) cannot import unnamed modules. Therefore, to import OSGi bundles, all OSGi bundles in your application and the execution environment must be loaded within a matching Java 9 module. Doing this requires the following:

- Bundle `imports` match module `requires`.
- Bundle names match module names.
- Bundle versions match module versions.
- Private classes might need to be made accessible to all.
- A module layer must be defined per bundle dependency in the graph (in which nodes are modules, and edges define dependencies between modules). Java 9 layers allow new modules to be added to a running application, just as OSGi bundles allow new implementations to be loaded, started, stopped, and even updated within a running application.

The advantages of integration include

- Increased isolation: changes to a single OSGi bundle within a JPMS layer likely will not require that layer, or dependent layers, to be rebuilt.
- Bundles inherit Java module restrictions on split packages and circular dependencies.
- Bundles inherit Java module compile-time presence of all dependencies.
- Migration: OSGi bundle definition and resolution is maintained "as is," on top of Java modules.
- Compatibility: OSGi bundles will still work independently on Java SE 8 and earlier JVMs.



```
▼ integration
  ▼ simulation
    ▼ math
      ▼ src
        ▼ osgi
          ▼ simulation
            ▼ math
              ▼ impl
                  MathActivator.java
                  MyMathImpl.java
                SimulationMath.java
        ▼ simulation.math
          ▼ example
            ▼ simulation
              ▼ math
                  MyMathImpl.java
            module-info.java
      ▶ target
        pom.xml
```

Figure 7: The simulation.math Java module extends the OSGi math bundle implementations.

To integrate this way, the Java `math` module code in the earlier example will simply extend the OSGi `math` bundle built earlier. The package names in the OSGi bundle code need to be changed slightly, resulting in the combined directory structure shown in **Figure 7**.

The code in **Listing 3** contains the two implementations for the `math` module. The actual implementation is in the OSGi bundle, while the Java module simply extends it.

Listing 3: Integrating the OSGi math bundle and the Java math module

```
package osgi.simulation.math.impl;
import osgi.simulation.math.SimulationMath;
public class MyMathImpl implements SimulationMath {
   // Actual implementation here...
}

package example.simulation.math;
//
// JPMS Module extends OSGi Bundle
```

```
//
public class MyMathImpl extends osgi.simulation.math.impl.MyMathImpl {
}
```

The compile script from the Java module sample application needs to be modified slightly to include the updated module source path and the path to the OSGi math bundle file (in the classpath):

```
javac -d out -classpath ./math/target/*.jar --module-source-path
    ./math/src -m simulation.math
```

The application layer modularity remains the same, because it's simply loading the Java module wrappers around the OSGi bundles.

Conclusion

Cooperation beats rivalry. When combined, Java modules and OSGi offer a more complete and robust modularity solution than either offers on its own. For example, without Java modules, OSGi cannot support compile-time enforcement of dependencies. Likewise, application-level modularity isn't as complete and lifecycle control isn't as robust without OSGi.

Combining both also helps existing OSGi applications leverage a growing number of Java 9 modules, and it offers a clean integration (and, possibly, migration) path to build OSGi bundles and services into Java 9 applications.

I'll close with a final thought: when polarizing concepts arise in computer science (for example, the OS wars, the C++ versus Java debate, and framework differences), it's often better to avoid being religious and instead find ways to work together. </article>

Eric J. Bruno is a lead real-time engineer at Perrone Robotics, where he's teaching cars to drive themselves. He has 25 years' experience in the information technology community as an enterprise architect, developer, and analyst with expertise in large-scale distributed software design.

SPENCER WOOD

Wookiee: Reducing Microservice Configuration

An open source framework to set up microservices quickly and with little fuss

Being a developer today means dividing products into interconnected microservices—standalone, always-up programs designed to address a few key functions in an assembly line of such services. The trouble with this pattern, however, is that much of your setup code is reinvented based on the last service you wrote. By the time everything is up and running, you find you're likely to have burned as many cycles coaxing your program to work and play nicely with others as you've burned actually writing the key functions.

The open source framework Wookiee reduces the time and clutter of building a communicative microservice architecture. It takes the error-prone process of initial setup out of the picture. When I go to set up my next project, I take it for granted that I will be ready for the main logic with only a few lines of setup. I can also depend on the base Wookiee framework for hosting health checks, recording metrics, and loading and updating global configurations.

On top of that, Wookiee also has multiple components that seamlessly incorporate the latest technologies. Java-based frameworks can fill a similar niche, but the flexible and extensible nature of Wookiee has allowed me to apply it to every type of service I need.

I'll be walking you through the primary interfaces and key concepts of Wookiee. All code is in Scala, but it should be understandable if you know Java. In some places, I have simplified it to avoid concepts specific to Scala or to Akka, the open source messaging framework. If you implement Wookiee on your own, it will be critical to understand the Akka framework, because Wookiee provides/manages the Actor System and its respective routing. If you see the word *Harness* in class or package names, it is because *Harness* was the original name of Wookiee.

The Wookiee Service

The Service is the root of your application; it is your access point to add your key functions, and it lets you branch out into using Wookiee's features:

```
class MyMicroService extends Service
```

As you can see, implementing the `Service` interface is the only required code integration. Now, you point to a configuration file that lets you tweak your Service or Wookiee itself by specifying the VM option `-Dconfig.file`:

```
wookiee-system {
   services {
     internal = "MyMicroService"
   }
}
```

In this configuration, I tell Wookiee that there is one Service to initialize. The Java `main()` method is actually wrapped by another internal class, `HarnessService`, and on initialization it will instantiate `MyMicroService` along with any Components (which I will cover later). Upon running the new Service, you'll see the following on your console:

```
INFO  MyMicroService - The service MyMicroService started
INFO  ServiceManager - Service Manager started
INFO  ServiceManager - Wookiee Started, Let's Go
```

You are now set up and ready to start coding. You now have logging access everywhere, HTTP-accessible health checks and pings, configuration loading/watching, local Akka messaging, and helpful utility functions. You can even connect a companion test library that will let you spin up a mock Wookiee Service for build-time integration testing.

Next, I will examine the extra functionality you can add with Components.

Wookiee Components

Components are modular *living libraries.* That is, they spin up before your Service and allow you to set up and manage other libraries, features, or technologies. A Service can have any number of Components, and a Component can be designed to do anything. The Wookiee community has already open sourced a few Components that add helpful features, such as metrics. In the Component code below, I create a connection to a database and receive queries with it:

```
class MyDBQueryComponent(name: String) extends Component(name) {
   val dbConnection = DBConnection(config).start()

   def receive = {
      case DBQuery(queryString) =>
      // run the query that was sent against the
     // database connection that was initialized
      val dbQueryResult = dbConnection.query(queryString)
      sender ! dbQueryResult
   }
}
```

In this code, I establish a connection to the database, `dbConnection`, and then I am ready to handle queries from the Akka Actor `receive` block (which inflows messages). Results return to their original sender, which could exist anywhere in your Wookiee Service. There are many other possible applications of Components, which begin to become recognizable as you envision Wookiee-based architectures. **Figure 1** demonstrates how a normal Wookiee architecture incorporates a set of Components that load in parallel before it starts the Service.

To imagine your next Component, think back to a time you went to build a service that was accessible externally via HTTP. First, you selected a library that fit your needs. Next, you set up and configured an HTTP server class that needed to be passed around. And most HTTP libraries would require you to compose complex routing trees that are difficult to split across classes.

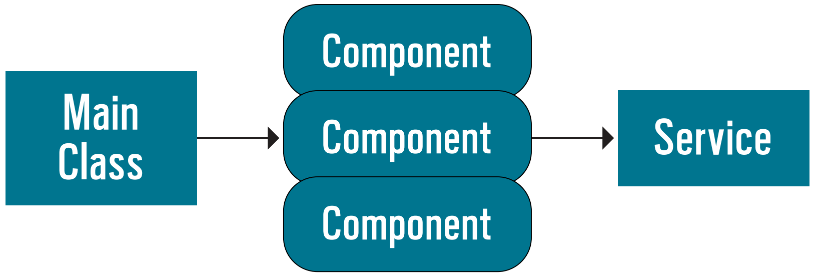


Figure 1: Startup of a Wookiee application

Once that was done, you still had other services to write before your product worked end to end, some of which needed HTTP servers of their own. This is a perfect situation for putting all of your interactions with the chosen library into its own Component.

Components can also be built to do things that are normally reserved for the service-specific startup logic—for instance, starting up technologies or connections to servers. They can even exist as their own pseudo-service running alongside your service and, for example, accepting messages to process and respond to independently.

There are numerous possibilities for Components, such as a supervised connection to a database with APIs for querying; managed access to a health-monitoring service; possibly a cluster Component that messages instances of itself on other servers; or a metrics coordinator that records timings and counts or one that connects to your big data stores, allowing access from any Service. As the next evolution of libraries, Components seek to widen your applications.

Wookiee Commands

When you are communicating using Wookiee, the first step is the Command. As with Services and Components, you make your own Command by extending an interface. The snippet below is simplified (the actual version allows more flexible marshaling and asynchronous processing):

```
class MyStringCommand extends Command {

  def execute(bean: CommandBean): CommandResponse = {
    CommandResponse("Replying")
    }
}
```

MyStringCommand, like all other Commands, is set up to handle messages sent to it. The messages are processed in the execute method, and a response is returned to the original sender of the message. These messages can come from an external HTTP or WebSocket entity, another Wookiee Service, or even from any other internal class (Figure 2). Replying is optional, and Commands can process more than one request at a time—acting like a thread pool for parallel processing.

Communication between Services depends on Zookeeper and Akka Remote—tools for registering cluster state and sending messages between server nodes. Wookiee wraps these tools using the Discoverable Command interface, which enables Commands to be seen by other Services and exchange messages. You can think of it as the ability to call a function in a class running on a different server. Getting this functionality is as easy as using the following extends statement:

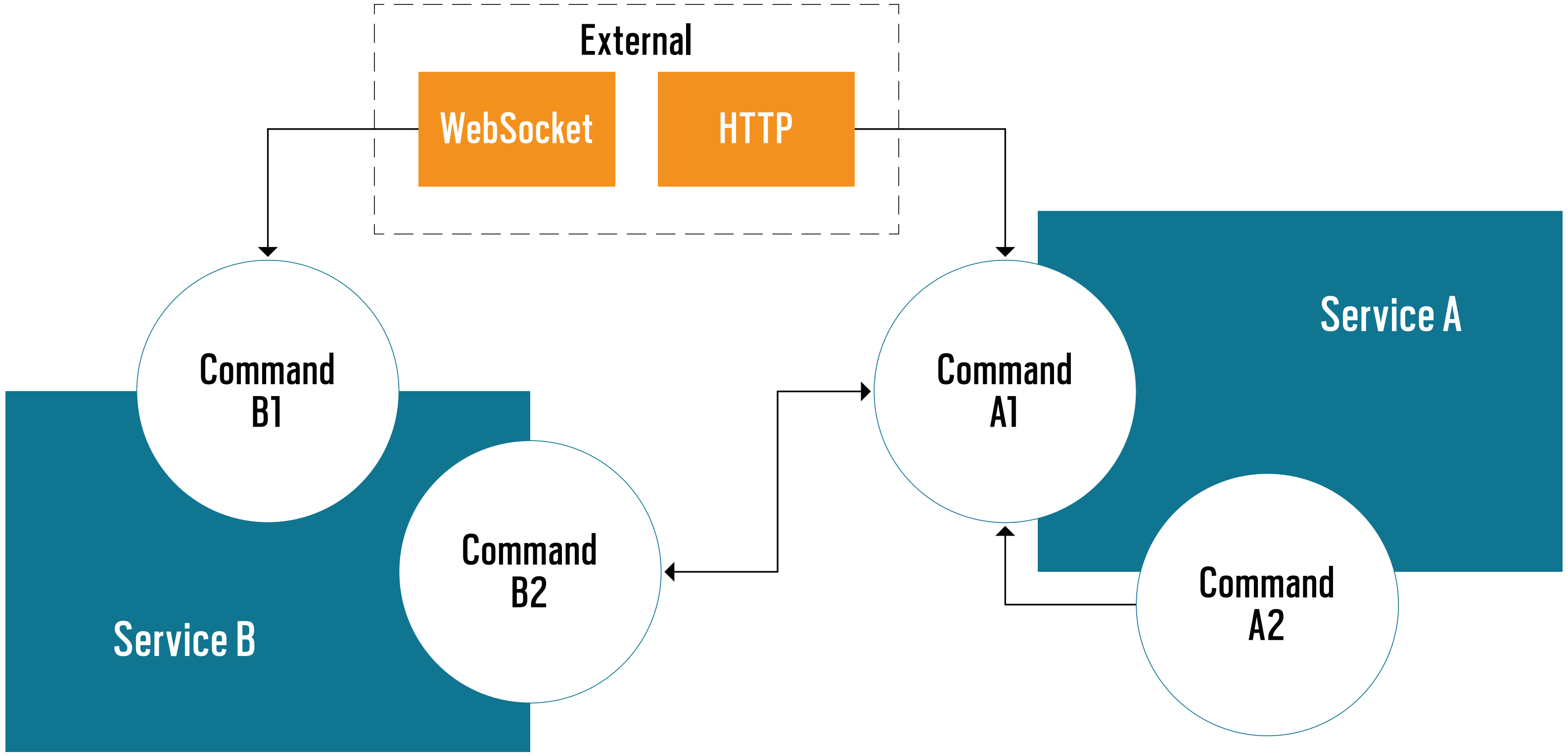


Figure 2: Messages can come from various external and internal sources.

```
class MyStringCommand extends Command with Discoverable
```

The Command is then ready to receive from other Services on which it must act. With the setup complete, you are clear to start sending messages to your new MyStringCommand. The CommandHelper and its extension, the DiscoverableCommandExecution (from the Wookiee Zookeeper Component), enable you to make calls to remote and local Commands, respectively:

```
class MyStringCommandCaller extends
     Actor with DiscoverableCommandExecution {
  ...
   executeCommand("MyStringCommand", bean)
   executeRemoteCommand("/other/service/path", "MyStringCommand", bean)
  ...
```

In addition, Commands themselves are accessible through HTTP or WebSockets using one of the premade Components that support HTTP. (Colossus, Akka HTTP, Spray, and Socko are currently supported.) Using supported premade Components is usually as easy as adding one more layer of inheritance onto your class and specifying the endpoint that the Command should match. In the following example, you would be able to reach the execute method of the MyHttpAccessibleGetCommand by sending an HTTP request to your Service on the /endpoint/to/match/over/http path.

```
class MyHttpAccessibleGetCommand extends Command with AkkaHttpGet {
   override def path: String = "endpoint/to/match/over/http"
   ...
```

Using a Service with Commands is a great way to connect your microservices. Commands take advantage of the flexibility of Akka Actors to route requests and messages cluster-wide, and they are usually the external point of entry for any processing on the Service.

A Wookiee About the Galaxy

The Wookiee Service can be shrunk into a single JAR file for hosting anywhere with all its dependencies, such as in a Docker container. Using a series of DevOps tools, my organization's vision has been to raise entire environments of Wookiee Services on Docker containers at build time to allow for automated testing and verification of code changes in a productionlike ecosystem. This enables QA to sign off on code changes soon after their completion, followed by a release to production right after—thereby delivering on the promise of continuous delivery.

In speaking about Commands, I addressed communication between Wookiee Services through Remote Commands. It is possible to go a step further in connecting our applications by using the Cluster Component. This Component allows you to send messages on topics that broadcast to all Services that subscribed to that topic. The result could be dozens of interconnected Services constantly sending notifications and state changes between each other. You could even use such a framework to create a reactive architecture that waits for state changes and propagates them through a cluster of Wookiee applications rather than using traditional method calls.

Letting the Wookiee Win

In my organization, I face diverse engineering challenges, from many-user application administration to demanding big data collection and querying. In every case, my organization has been able to describe code solutions using a common vocabulary of Service, Component, and Command. With this model, we have avoided many of the pitfalls inherent in creating setup logic for each new application. With the help of tools such as Wookiee, developers eventually will be able to take for granted that their next microservice can go from an empty repository to operational quickly—possibly even in minutes. </article>

Spencer Wood is the big data democratizer at newly acquired Oracle Infinity, the next-generation streaming infrastructure for Oracle Marketing Cloud. For the last five years, he has been building performant, robust, infinite-parameter streaming big data products. He has maintained Wookiee since its inception.

KEN KOUSEN

Groovy 3.0: What's Coming

One of the most popular JVM languages adds handy new features.

Groovy has always been easy for Java developers to learn, partly because the language syntax is a natural extension and simplification of Java, and partly because virtually all Java syntax also compiles in Groovy. In fact, it's sometimes an amusing demo to simply change a Java class to Groovy by changing the file name from `.java` to `.groovy` and watching it compile without problems. While the result might not be idiomatic Groovy, the demo shows how easy it is to port code from one language to the other.

This situation changed with the introduction of JDK 8, because Java added syntax for lambda expressions and method references that don't fit the Groovy model. While Groovy supplies closures where Java expects lambdas, it was considered unfortunate that the complete set of Java syntax no longer can be compiled in Groovy.

Along with other improvements in Groovy 3.0, with the introduction of the new Parrot parser, Java syntax once again works in Groovy. Groovy 3.0 is in alpha at the moment (specifically, 3.0.0-alpha-1). The release notes show the direction the core team is taking, and feedback is welcome. This article reviews many of the new features and examines some existing features of Groovy that you might not know about.

Version Numbers

Groovy is an Apache project, which means it follows a semantic versioning scheme. That is, the first number is the major version, the number after the first dot is the minor version, and bug fix versions come after the second dot. As of early 2018, the current stable version of Groovy is 2.4 and the latest point release is 2.4.13. The 2.4 line is stable and effective, but there are major changes coming soon.

The next planned release in the 2.x line is 2.5, which is currently in beta. The biggest addition in 2.5 is the new macro capability, discussed below, which converts simple methods into abstract syntax tree (AST) transformations.

Except as I explain in a moment, the plan is to jump from version 2.5 to 3.0. Groovy 3.0 will include the Parrot parser and support the functional syntax in the Java 8 release. The Java syntax for lambda expressions and method references will then work in Groovy without modification.

Version 3.0 will also be the first version of Groovy that requires JDK 8 as the minimum JDK level. Because some developers still might not be able to move to JDK 8, the Groovy team decided to back-port as many of the new features as possible into Groovy 2.6, which will run on JDK 7. That means 2.6 will be, at best, a temporary solution, but better than nothing for those developers who need it. The specific list of features to be back-ported to version 2.6 is still under discussion.

Let's look at some new features. In the rest of this article, I presume you have some working familiarity with Groovy.

Functional Groovy

Groovy 2.x includes several functional features. Because most of the features were developed prior to the release of JDK 8, they use a slightly different syntax than that found in Java.

For example, the simple map/filter/reduce paradigm is implemented in Groovy using methods defined directly on collections. The Java `map` method is called `collect` in Groovy, while `filter` is `findAll`, and `reduce` is `inject`.

```
List nums = [3, 1, 4, 1, 5, 9, 2, 6, 5]
assert nums.collect { it * 2 }   // [6, 2, 8, 2, 10, 18, 4, 12, 10]
     .findAll { it % 3 == 0 }    // [6, 18, 12]
     .sum() == 36
```

(Unfortunately, the `collect` method name was established for mapping operations long before Java used the same name for the operation in the `Stream` interface that converts streams into collections.)

As a reminder, the closure syntax in Groovy uses braces to wrap the entire expression, and if you are using a single-argument closure without defining a dummy name, the variable it is used by default.

These days, Java 8 and above can do this same procedure using streams. The analogous code in Java would be this:

```
List<Integer> nums = Arrays.asList(3, 1, 4, 1, 5, 9, 2, 6, 5);
int sum = nums.stream()
        .mapToInt(n -> n * 2)
        .filter(n -> n % 3 == 0)
        .sum();
System.out.println("The sum is " + sum);
```

The mapToInt method on Stream takes a lambda expression representing a java.util.function.Function interface and produces an IntStream. The filter method takes a java.util.function.Predicate as an argument and returns a new IntStream, which includes the sum method to get the final result.

If you want to use streams in Groovy, the only difference is that where the Java methods expect functional interfaces, you provide Groovy closures, as in the following example:

```
assert nums.stream()
       .mapToInt { it * 2 }
       .filter { it % 3 == 0 }
       .sum() == 36
```

Here, the arguments to the Java stream methods mapToInt and filter are Groovy closures.

If you use Groovy versions that support the new Parrot parser, you can supply Java lambdas instead, in any of the legal forms. For example:

```
assert nums.stream()
   .mapToInt((n) -> n * 2) // Expression lambda
   .filter(n -> {            // Block lambda
     return n % 3 == 0
   })
   .sum() == 36
```

Note the use of the expression lambda syntax, which does not require either braces or a `return` statement, in the `mapToInt` intermediate method on `stream`. Just to show the alternative, a block lambda was used in the `filter` method.

You can also use Java method references. Groovy uses the ampersand operator (`&`) as a way to refer to methods, but Java uses a double colon (`::`). For example, one of the additions to the API in Java 8 was the static `sum` method in the `Integer` class, which can be used as a `BinaryOperator` argument to the `reduce` method. The following code is written in Groovy, but uses the standard Java syntax:

```
assert nums.stream()
      .mapToInt(n -> n * 2)
      .filter(n -> n % 3 == 0)
      .reduce(0, Integer::sum) == 36     // BinaryOperator
```

The point is that all the Java syntax works with the new Groovy parser. Groovy adds the ability to add default parameter values to lambdas, too.

Groovy goes well beyond what Java provides. For example, Groovy has AST transformations that generate useful code at compile time. Consider the classic Fibonacci calculation, which is easy to do inefficiently.

```
@Memoized
long fibonacci(long n) {
   if (n < 2) 1
   else fibonacci(n - 1) + fibonacci(n - 2)
}
```

The key is the @Memoized annotation, which triggers a corresponding AST transformation. Any evaluation of the fibonacci method defined this way will involve many repeated calculations, even for small arguments. For example, fibonacci(5) = fibonacci(4) + fibonacci(3) = fibonacci(3) + fibonacci(2) + fibonacci(2) + fibonacci(1) and so on.

To prevent the combinatorial explosion in the number of repeated calculations, the @Memoized AST transform generates a cache of values, in which the keys are the arguments and the values are the results of evaluating the operation. That means each computation of fibonacci(n) is stored in a map of n to the result. With the AST transformation applied, it's fast and easy to compute higher values, such as

```
assert fibonacci(100) == 1298777728820984005
```

Another AST transformation provided in the Groovy standard library is @TailRecursive. If you can write an algorithm in such a way that the last evaluated expression is a recursive call with different arguments, the transform will convert the recursive calls into iterative ones.

For example, see the following factorial calculation:

```
import groovy.transform.*

@TailRecursive
def fact(n, acc = BigInteger.ONE) {
   n < 2 ? acc : fact(n - BigInteger.ONE, n * acc)
}

def result = fact(70000)
println "$result".size() // 308,760 digits
```

Note the use of Groovy's optional arguments as well—another underrated feature.

Groovy also has closure currying, closure composition using the shift operators, and more. Before going on to some of the other additions to Groovy, I will show you what happens beneath the covers with AST transformations, using macros as an example.

Macros

First, a note on terminology. Groovy uses regular Java annotations to trigger code modifications done by the parser at compile time. The compile-time changes are called AST transformations, and for each annotation, such as @ToString, there is a corresponding AST transformation class, org.codehaus.groovy.transform.ToStringASTTransformation. The transformation class visits the various nodes of the syntax tree and modifies them as needed. Writing a transformation class involves writing the node-manipulation code (creating nodes, adding them to the tree, and so on), and thus requires deep knowledge of how the compiler generates the AST in the first place. (Although there are various utility and builder classes available to help, writing an AST transformation is a tedious process.)

What Groovy macros enable you to do is to write the transformation code as a regular Groovy method wrapped inside a macro block. The compiler then converts the provided methods into the detailed node-manipulation code for you.

Using an AST transformation is easy. For example, here the @ToString annotation is added to a plain old Groovy object (often referred to as a POGO).

```
import groovy.transform.*
@ToString
class Person {
   String first
   String last
}
```

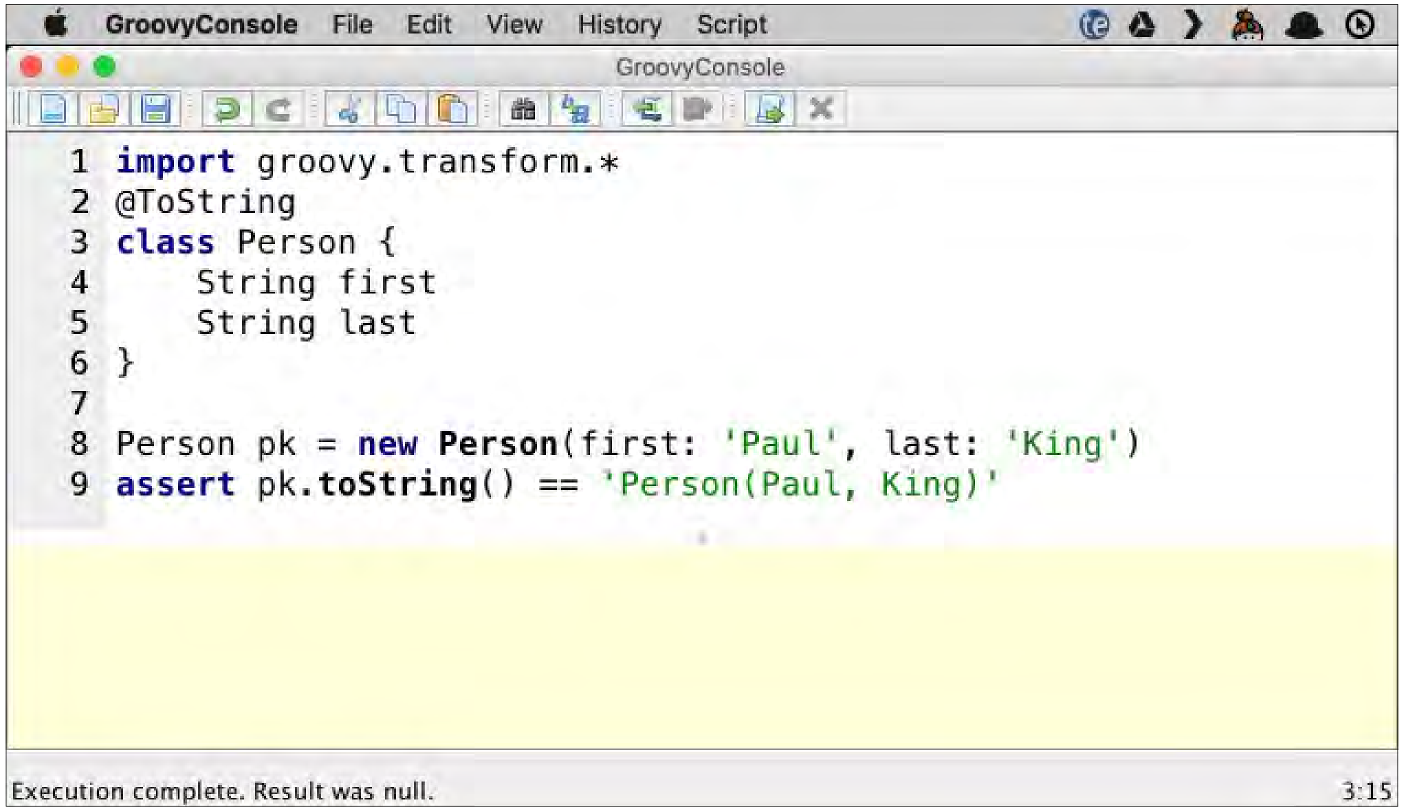
The @ToString annotation triggers an AST transformation (via the ToStringASTTransformation class) that generates a toString method during the compilation process and adds it to the compiled bytecodes. In this case, the generated code returns the fully qualified class name, followed by the values of the attributes from the top down, wrapped in parentheses:

```
Person pk = new Person(first: 'Paul', last: 'King')
assert pk.toString() == 'Person(Paul, King)'
```

(Note: Paul King is one of the Groovy core team members, as well as one of the lead coauthors of the book on Groovy 2.x, Groovy in Action.)

To see some of the details, load the POGO into the Groovy Console, which comes with the Groovy SDK, as I've done in **Figure 1**.

Under the Script menu of the Groovy Console, there is an entry called "Inspect AST." This opens the Groovy AST Browser, which shows the generated nodes in the tree, some of which are shown in **Figure 2**.

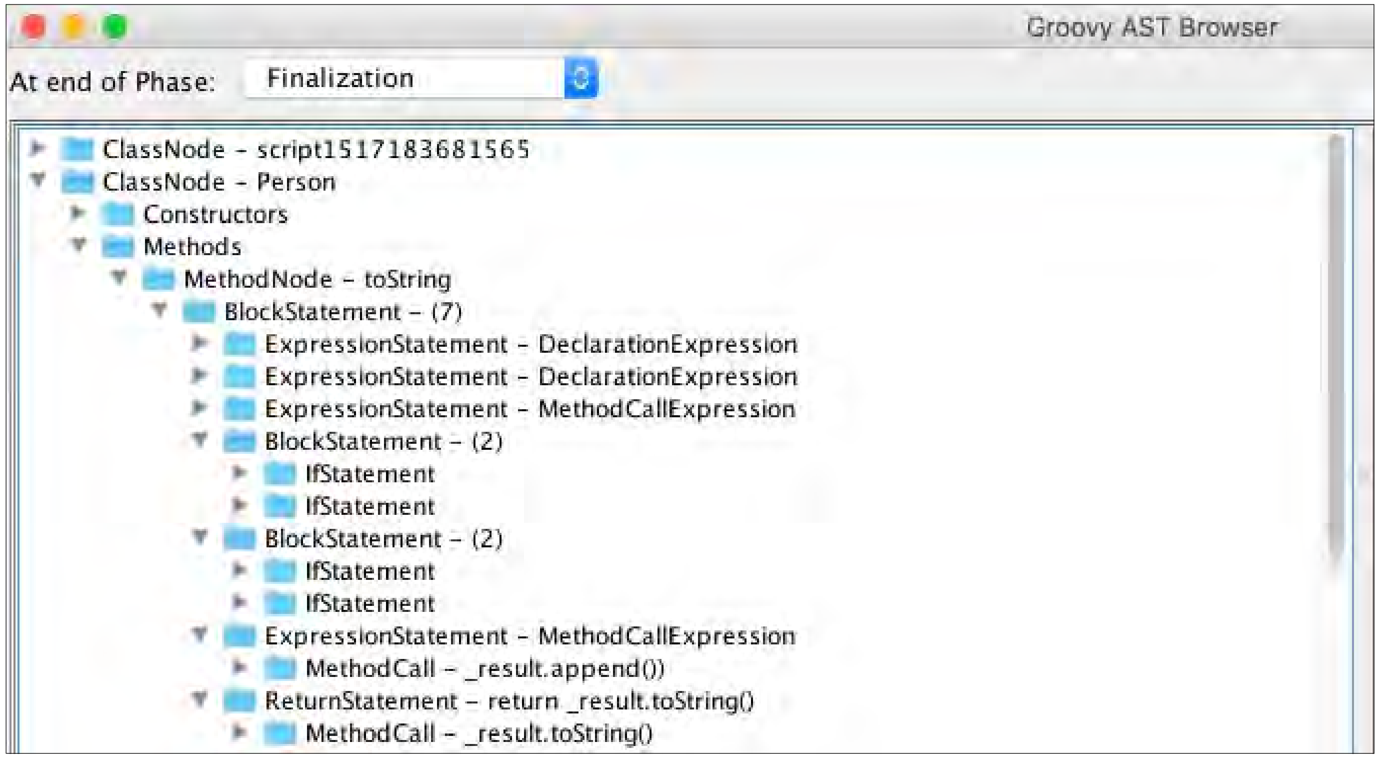


```
import groovy.transform.*
@ToString
class Person {
    String first
    String last
}

Person pk = new Person(first: 'Paul', last: 'King')
assert pk.toString() == 'Person(Paul, King)'
```

```
Execution complete. Result was null.                                3:15
```

Figure 1: Code that will be converted into ASTs in Figure 2



At end of Phase: Finalization

```
▸ ClassNode - script1517183681565
▾ ClassNode - Person
    ▸ Constructors
    ▾ Methods
        ▾ MethodNode - toString
            ▾ BlockStatement - (7)
                ▸ ExpressionStatement - DeclarationExpression
                ▸ ExpressionStatement - DeclarationExpression
                ▸ ExpressionStatement - MethodCallExpression
                ▾ BlockStatement - (2)
                    ▸ IfStatement
                    ▸ IfStatement
                ▾ BlockStatement - (2)
                    ▸ IfStatement
                    ▸ IfStatement
                ▾ ExpressionStatement - MethodCallExpression
                    ▸ MethodCall - _result.append())
                ▾ ReturnStatement - return _result.toString()
                    ▸ MethodCall - _result.toString()
```

Figure 2: AST tree for the code in Figure 1

The source code for the transformation class, which can be found on Github, shows the details of the transformation and is far too long to include here. The source consists of more than 200 lines that work with instances of classes such as `VariableExpression`, `MethodCallExpression`, `FieldNode`, `PropertyNode`, `BlockStatement`, and so on, which combine to generate the tree shown in Figure 2.

The beauty of the new macro approach is that much of the code in the transformation class can be rewritten as normal Groovy statements wrapped in a block called `macro`. The Groovy language documentation shows a trivial example where AST transformation code such as

```
ReturnStatement code = new ReturnStatement(new ConstantExpression("42"))
```

can be replaced with

```
ReturnStatement simplestCode = macro { return "42" }
```

Life isn't usually that simple, but I hope this gives you an idea how much easier it will be to work with macros. It's not a capability that is likely to change what Groovy developers do every day, but it will make the creation of AST transformations easier.

Groovy already includes a wealth of useful AST transformations, triggered by annotations such as @Canonical, @Delegate, @Immutable, @InheritConstructors, @Slf4j, @TypeChecked, and @CompileStatic. The new macro capability will make it much easier to create many more.

Traits

Another major addition to Java 8 was the ability to define default and static methods inside interfaces. While the Parrot parser will support them as well, at least in some form, Groovy already has something that does this and more: traits.

Groovy traits are used like interfaces, but their methods can contain implementations, just like Java's default methods. Traits can also have state, however, which Java interfaces cannot.

The Groovy documentation contains several simple examples, along with the rules and restrictions on their usage, but here is an example from the Grails framework. Grails 3 is based on Spring Boot, and it provides a powerful object-relational mapping API called GORM that works with both relational and NoSQL databases. The testing framework has recently been refactored to use traits.

Here is the beginning of a test of a Grails controller. The controller receives HTTP requests by mapping a URL to a controller operation, known as an action. The declaration of a controller test, which is auto-generated by Grails when you define a controller, looks like this:

```
class QuestControllerSpec extends Specification
    implements ControllerUnitTest<QuestController>, DomainUnitTest<Quest> {
```

Both ControllerUnitTest and DomainUnitTest are traits, and are implemented by the class the same way interfaces are. The source code for ControllerUnitTest, for example, starts this way:

```
@CompileStatic
trait ControllerUnitTest<T> implements
  ParameterizedGrailsUnitTest<T>, GrailsWebUnitTest {

   static String FORM_CONTENT_TYPE = MimeType.FORM.name
   static String XML_CONTENT_TYPE = MimeType.XML.name
   static String JSON_CONTENT_TYPE = MimeType.JSON.name
   static String HAL_JSON_CONTENT_TYPE = MimeType.HAL_JSON.name
   static String HAL_XML_CONTENT_TYPE = MimeType.HAL_XML.name
   // … other constants ...

   private T _proxyInstance // instance attribute
```

It then has various methods such as

```
@CompileStatic(TypeCheckingMode.SKIP)
Map getModel() {
   request.getAttribute(
     GrailsApplicationAttributes.CONTROLLER)?.modelAndView?.model ?: [:]
}
```

Note the use of the `@CompileStatic` AST transformation, which triggers the corresponding AST transformation and uses compile-time checks in the style of Java to perform static compilation, thus bypassing the Groovy meta object protocol. The `getModel` method uses Groovy's safe navigation operator, `?`, and the so-called Elvis operator, `?:`. Other methods in the trait let you retrieve the view, the controller, and more. By using traits, the test class includes many powerful capabilities at compile time. In Groovy 3.0, default methods in Java interfaces are implemented using traits.

Miscellaneous Features

Groovy 3 includes a wide range of smaller changes that are interesting. Some are trivial extensions used to make it more Java-friendly, including

- A do/while loop
- Java-style for loops with multiple looping variables
- Java-style array initialization
- Java's try-with-resources syntax

Each of these could be done more idiomatically with Groovy alternatives, but it's convenient to be able to use the Java versions if you're still learning.

Some of the new additions are simple. For example, you can use an exclamation point on the `in` and `instanceof` operators to test for negation:

```
assert 45 !instanceof Date
assert 4 !in [1, 3, 5, 7]
```

The Elvis operator has been in Groovy for a long time. It is a minimal form of Java's ternary operator, which uses a supplied value if it's true according to the Groovy truth, or a default if not. The new Elvis assignment operator, `?=`, lets you check for the Groovy truth and use the result in an assignment, all in a single statement.

```
@groovy.transform.Canonical
class User {
   String name
}

User u = new User()
// u.name = u.name ?: 'default'  // Elvis operator
u.name ?= 'default'              // Elvis assignment

assert u.toString() == 'User(default)'
```

```
u = new User('Guillaume Laforge')
u.name ?= 'default'              // Elvis assignment

assert u.toString() == 'User(Guillaume Laforge)'
```

(Guillaume Laforge is the head of the Groovy project.)

Groovy has always supported operator overloading, in that every operator in Groovy actually invokes a method. For example, the + sign calls the `plus` method, `*` calls the `multiply` method, and so on. One interesting ramification of this design is that the `==` operator does not check that two references point to the same object, but rather it invokes the `.equals` method instead. That intuitive approach means Groovy equivalence is represented by `==`. If you want to see if two references are the same, use the `is` method.

In Groovy 3.0, however, the `===` and `!==` operators now delegate to the `is` method, so you can use those operators instead.

There are a few additional features and changes. See the release notes for details.

Conclusion

Groovy is an active, evolving language. Since its move to the Apache Software Foundation, the number of monthly downloads has more than doubled. The changes coming in Groovy 3.0 include native support for lambda expressions, method references, default methods in interfaces, and more. These changes and others will continue to make Groovy a natural way to integrate features to existing Java projects as well as improve new Groovy development. `</article>`

Ken Kousen is a Java Champion and the author of the books *Modern Java Recipes* (O'Reilly Media), *Gradle Recipes for Android* (O'Reilly Media), and *Making Java Groovy* (Manning), as well as more than a dozen video courses at Safari Books Online on topics ranging from Java to Groovy to Spring to Android. He holds BS degrees in both mechanical engineering and mathematics from MIT, a PhD in aerospace engineering from Princeton, and an MS in computer science from Rensselaer Polytechnic Institute.

BEN EVANS

CHRIS NEWLAND

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

Escape Analysis in the HotSpot JIT Compiler

Complex analysis of variables' scope enables a variety of subtle optimizations.

In previous issues of *Java Magazine*, we introduced the basic theoretical concepts of just-in-time (JIT) compilation as well as the Java Microbenching Harness and the JITWatch open source tool for visualizing and understanding the basic mechanisms provided in the Java HotSpot VM. In this article, we dive into *escape analysis* (EA), which is one of the more interesting forms of optimization that takes place in the JVM. EA is an automatic analysis of the scope of variables performed by the JVM to enable certain kinds of special optimizations, which we'll also examine. To follow along, you need only basic familiarity with how the HotSpot JVM works.

To understand the basic idea behind EA, let's look at the following buggy C code—which is impossible to write in Java, of course:

```
int * get_the_int() {
   int i = 42;
   return &i;
}
```

This C code creates an `int` on the stack and then returns a pointer to it as the return value of the function. This is *incorrect*, because the stack frame where the `int` was stored is destroyed as `get_the_int()` returns, so you have no way of knowing what is in the memory location if it is accessed at some later time.

Completely eliminating the possibility of these types of bugs was a major safety goal in the design of the Java platform. By design, the JVM does not have a low-level "read memory at location indexed by value" capability. All heap access is done by field name (or array index) relative to a base object. The relevant JVM bytecodes corresponding to these operations include `getfield` and `putfield`.

Now consider the following bit of Java code:

```
public class Rect {
   private int w;
   private int h;

   public Rect(int w, int h) {
      this.w = w;
      this.h = h;
   }

   public int area() {
      return w * h;
   }

   public boolean sameArea(Rect other) {
      return this.area() == other.area();
   }

   public static void main(final String[] args) {
      java.util.Random rand = new java.util.Random();

      int sameArea = 0;

      for (int i = 0; i < 100_000_000; i++) {
         Rect r1 = new Rect(rand.nextInt(5), rand.nextInt(5));
         Rect r2 = new Rect(rand.nextInt(5), rand.nextInt(5));
```

```
                if (r1.sameArea(r2)) {
                    sameArea++;
                }
            }

            System.out.println("Same area: " + sameArea);
        }
    }
```

This code creates 100 million pairs of rectangles of random size and counts how many pairs are of equal size. During each iteration of the `for` loop, a new pair of `Rect` objects is allocated. You would therefore expect 200 million `Rect` objects to be allocated in the `main` method: 100 million each of `r1` and `r2`.

However, if an object is created in one method and used exclusively inside that method—that is, if it is not passed to another method or used as the return value—the runtime can potentially do something smarter. You can say that the object does not *escape* and the analysis that the runtime (really, the JIT compiler) does is called *escape analysis*.

If the object does not escape, then the JVM could, for example, do something similar to an "automatic stack allocation" of the object. In this case, the object would not be allocated on the heap and it would never need to be managed by the garbage collector. As soon as the method containing the stack-allocated object returned, the memory that the object used would immediately be freed.

In practice, the HotSpot VM's C2 JIT compiler does something more sophisticated than stack allocation. Let's have a look.

Within the HotSpot VM source code, you can see how the EA analysis system classifies the usage of each object:

```
typedef enum {
  NoEscape = 1,     // An object does not escape method or thread and it is
                    // not passed to call. It could be replaced with scalar.
  ArgEscape = 2,    // An object does not escape method or thread but it is
              // passed as argument to call or referenced by argument
              // and it does not escape during call.

  GlobalEscape = 3 // An object escapes the method or thread.
}
```

The first option suggests that the object can be replaced by a scalar substitute. This elimination is called *scalar replacement*. This means that the object is broken up into its component fields, which are turned into the equivalent of extra local variables in the method that allocates the object. Once this has been done, another HotSpot VM JIT technique can kick in, which enables these object fields (and the actual local variables) to be stored in CPU registers (or on the stack if necessary).

One of the major challenges of the Java platform is the sophistication of the execution model. In this case, just by looking at the Java source code, you might naively conclude that the object `r1` does not escape the `main` method but that `r2` is passed as an argument to the `sameArea` method on `r1` and so it escapes the scope of the `main` method.

Using the previous classifications, it would appear at first sight that `r1` should be treated as a `NoEscape` and `r2` should be treated as an `ArgEscape`; however, this would be a dangerous conclusion for several reasons.

First of all, recall that method calls in Java are replaced by the Java compiler with `invoke` bytecodes. These operate by setting up the stack with the destination of the call (known as the receiver object) and with any arguments before the call of the appropriate method is looked up and dispatched (that is, executed).

This means that the receiver object is also passed to the method being called (it becomes the `this` object in the method that is called). So receiver objects also escape the current scope; in this case, that would mean that both `r1` and `r2` would be classified as `ArgEscape` if EA were to be applied to the code as it appears in the Java source code.

If this were the whole story, it would seem that the feature of allocation elimination is extremely limited. Fortunately, the Java HotSpot VM can do better than this. Let's look at the detail of the bytecode and see what can be observed.

The method `sameArea()` is both small (17 bytes of bytecode) and frequently called in the example, thereby making it an ideal candidate to be inlined:

```
public boolean sameArea(Rect);
 Code:
    0: aload_0
    1:    invokevirtual #4                  // Method area:()I
    4: aload_1
    5:    invokevirtual #4                  // Method area:()I
    8: if_icmpne        15
   11: iconst_1
   12: goto             16
   15: iconst_0
   16:    ireturn
```

The method makes two further calls to another (easily inlineable) method `area()`:

```
public int area();
  Code:
   0: aload_0
   1: getfield       #2                   // Field w:I
   4: aload_0
   5: getfield       #3                   // Field h:I
   8: imul
   9: ireturn
```

Using JITWatch or `PrintCompilation`, you can see that the calls to `area()` are indeed inlined into their caller `sameArea()` and that method is inlined into its callsite in the loop body of the `main()`

method. JITWatch provides a useful graphical representation of which methods will be inlined (illustrated in **Figure 1**, which due to its size is available only online).

Remember that the order in which the Java HotSpot VM applies its JIT compiler optimizations is important. Method inlining is one of the first optimizations and is known as a *gateway optimization*, because it opens the door to other techniques by first bringing related code closer together.

Now that the call to `sameArea()` and the calls to `area` have been inlined, the method scopes no longer exist, and the variables are present only in the scope of `main()`. This means that EA will no longer treat either `r1` or `r2` as an `ArgEscape`: both are now classified as a `NoEscape` after the methods have been fully inlined.

This might seem like a counterintuitive result, but you need to bear in mind that the original source code is not what the JIT compiler will use as a starting point. Without this knowledge, it's easy to draw the wrong conclusion about what is eligible for EA.

In the previous example, both of these object allocations can avoid using the heap and instead their fields will be treated as individual values. The register allocator will normally place the broken-up object fields directly into registers, but if not enough free registers are available, the remaining fields will be placed on the stack. This situation is known as a *stack spill*.

To illustrate the power of eliminating heap allocations inside tight loops of code, run this program with and without EA enabled and inspect the activity of the garbage collector.

Because EA is enabled by default in modern JVMs, to do this, you need to disable EA by using the JVM switch `-XX:-DoEscapeAnalysis`.

Here is the garbage collection log with EA enabled (with some extraneous detail removed):

```
java -XX:+PrintGCDetails Rect
Same area: 18073993
Heap
 PSYoungGen      total 95744K, used 13462K
  eden space 82432K, 16% used
  from space 13312K, 0% used
  to   space 13312K, 0% used
 ParOldGen       total 218624K, used 0K
  object space 218624K, 0% used
 Metaspace       used 2664K, capacity 4490K, committed 4864K, reserved 1056768K

  class space    used 286K, capacity 386K, committed 512K, reserved 1048576K
```

The log shows that there were no GC events at all—instead, the log just contains the heap summary as the process exits. If you look at the GC Log from a run without escape analysis enabled, then things look quite different:

```
java -XX:+PrintGCDetails -XX:-DoEscapeAnalysis Rect
[GC (Allocation Failure) [PSYoungGen: 82432K->480K(95744K)] 82432K->488K(314368K),
0.0008348 secs] [Times: user=0.01 sys=0.00, real=0.00 secs]
[GC (Allocation Failure) [PSYoungGen: 82912K->464K(95744K)] 82920K->480K(314368K),
0.0007404 secs] [Times: user=0.00 sys=0.00, real=0.01 secs]

[Many minor GC collections]

[GC (Allocation Failure) [PSYoungGen: 56352K->0K(55808K)] 56720K->368K(274432K),
0.0004405 secs] [Times: user=0.00 sys=0.01, real=0.00 secs]
[GC (Allocation Failure) [PSYoungGen: 55296K->0K(54784K)] 55664K->368K(273408K),
0.0004537 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
Same area: 18080278
Heap
 PSYoungGen      total 54784K, used 46674K
  eden space 54272K, 86% used
  from space 512K, 0% used
  to   space 512K, 0% used
 ParOldGen       total 218624K, used 368K
  object space 218624K, 0% used
 Metaspace       used 2665K, capacity 4490K, committed 4864K, reserved 1056768K
  class space    used 286K, capacity 386K, committed 512K, reserved 1048576K
```

In this case, you can clearly see the GC events that are caused by allocation failure as the Eden area of memory fills up and needs to be collected.

Conclusion

The addition of EA to the Java HotSpot VM is a useful improvement. When EA was in development, an additional 3% to 6% performance increase in real-world tests was seen that was directly attributable to it.

However, for the developer who is also interested in the how and why of platform features, EA provides an interesting insight: it is a feature that depends upon another optimization (automatic inlining) and is essentially useless without it.

The low-level details and the source code of the JVM's implementation can be found in `opto/escape.hpp` in the Java HotSpot VM source code. It is a modified form of the algorithm presented in the "Escape Analysis for Java" proceedings of the ACM SIGPLAN Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) conference in November 1999 by Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff.

In addition to allocation elimination, within the Java HotSpot VM there are several other optimization techniques that depend upon similar scope analysis to that which is used for allocation elimination. These mostly work with the intrinsic locks that Java provides for each object. We'll discuss those optimizations in the next issue. </article>

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

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

JOSH LONG

Reactive Spring: Setting up a REST API

In part 1, I built a reactive app with the Spring Framework. Now I'll provide access to it by quickly implementing a REST API.

In the previous issue of this magazine, I presented part 1 of this two-part series. In it, I explained the reactive components that are available in Spring Framework 5.0, and I built a simple project to serve up book data. Now that I've got data in the data source, I will stand up a REST API. I'll use Spring WebFlux, a brand-new reactive web runtime and component model. Spring WebFlux does not depend on the Java Servlet specification. It can work independently, with a Netty-based web server. It is designed, from the bottom up, to work with `Publisher<T>` instances.

To follow along, you'll need just a little Spring background, although you'll be well served by quickly reading or reviewing the previous article in this series.

Spring WebFlux

With Spring WebFlux, I can use Spring model-view-controller (MVC)-style controllers, as shown in Listing 1.

Listing 1: A Spring MVC-style REST API

```
package com.example.libraryservice;
import org.springframework.context.annotation.Profile;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RestController;
import reactor.core.publisher.Flux;
```

```
@Profile("mvc-style")
@RestController
class BookRestController {
   private final BookRepository bookRepository;

   BookRestController(BookRepository bookRepository) {
      this.bookRepository = bookRepository;
   }

   @GetMapping("/books")
   Flux<Book> all() {
      return this.bookRepository.findAll();
   }

   @GetMapping("/books/{author}")
   Flux<Book> byAuthor(@PathVariable String author) {
      return this.bookRepository.findByAuthor(author);
   }
}
```

Spring has the concept of *profiles*. Profiles are essentially labels, or tags, for Spring beans. Beans in a given profile don't exist unless that profile is activated. The easiest way to activate a profile is to use a command-line argument when you run the `java` command. For example, if you want to activate all the beans under the `profile1` and `profile2` profiles, you'd use a command line like this:

```
java -Dspring.profiles.active=profile1,profile2 -jar ...
```

The benefit of the profile, in this case, is that you can have the same HTTP endpoints implemented three different ways in the same codebase and activate only one at a time. The controller in Listing 1 should look familiar to anyone who's ever used Spring MVC. It might look familiar, but it is *not* Spring MVC. I am using a new reactive runtime called Spring WebFlux. The annotations are the same, but the rules are sometimes different.

Functional Reactive Endpoints

Listing 1 demonstrates a controller. Spring WebFlux controllers define endpoint handlers and endpoint mappings through declarative annotations. The annotations describe how the routing for a given endpoint is to be handled. The annotations are sophisticated, but ultimately limited to whatever the framework itself can do with those annotations. If you want more-flexible request-matching capabilities, you can use Spring WebFlux functional reactive endpoints, as shown in Listing 2. You can run this code using the `frpjava` profile.

Listing 2: The same endpoints as in Listing 1 reworked as functional reactive endpoints in Java

```
package com.example.libraryservice;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Profile;
import
 org.springframework.web.reactive.function.server.RouterFunction;
import static
  org.springframework.web.reactive.function.server.RequestPredicates.GET;
import static
  org.springframework.web.reactive.function.server.RouterFunctions.route;
import static
  org.springframework.web.reactive.function.server.ServerResponse.ok;

@Profile("frp-java")
@Configuration
class BookRestConfigurationJava {
   @Bean
   RouterFunction<?> routes(BookRepository br) {
     return
     route(GET("/books"),
        req -> ok().body(br.findAll(), Book.class))
              .andRoute(GET("/books/{author}"),
        req -> ok().body(br.findByAuthor(req.pathVariable("author")),
          Book.class));
   }
}
```

The functional reactive style lets you express HTTP endpoints as request predicates mapped to a handler class. The handler class implementation is easily expressed as concise Java lambdas. You can use the default request predicates or provide your own to gain full control over how requests are matched and dispatched.

In Listing 2, I produce a result and pass it to the `body(Publisher<T>)` method, along with a class literal. I need the class literal to help the engine figure out what type of message framing it should do. Remember that `Publisher<T>` might produce billions of records—it might never stop! The producer can't afford to wait until all records have been produced and only then marshal the record from an object to JSON. So, it marshals each record as soon as it gets it. I need to tell it what kind of message to look for. In Spring MVC–style controllers, the return value (a `Publisher<T>`) in the handler methods encodes its generic parameter, `T`, and the engine can retrieve that generic parameter using reflection. The engine cannot do the same thing for the instance variable passed into the `body` method as a parameter, because there is no easy way to retrieve the generic signature of instance variables. This limitation is called *type erasure*. The type literal gets you past this restriction. If you're using the Kotlin language, things are even more concise thanks to a

The Spring Security framework supports a rich set of integrations with all manner of identity providers.

Kotlin-language DSL that also ships as part of Spring Framework 5. The Kotlin DSL requires less code and also supports retrieving the generic parameter, thanks to runtime reification of inline methods.

Listing 3 shows the same endpoints reimplemented using the Kotlin-language DSL.

Listing 3: The same endpoints using the Kotlin-language DSL

```
package com.example.libraryservice
import org.springframework.context.annotation.Bean
import org.springframework.context.annotation.Configuration
import org.springframework.context.annotation.Profile
import
 org.springframework.web.reactive.function.server.ServerResponse.ok
import org.springframework.web.reactive.function.server.body
import org.springframework.web.reactive.function.server.router

@Profile("frp-kotlin")
@Configuration
class BookRestConfigurationKotlin {
   @Bean
   fun routes(br: BookRepository) = router {
      GET("/books") { r -> ok().body(br.findAll()) }
      GET("/books/{author}") { r ->
         ok().body(br.findByAuthor(r.pathVariable("author")))
      }
   }
}
```

Spring Security

Even with this, though, the code is not quite ready for production. I need to address security. The Spring Security framework supports a rich set of integrations with all manner of identity providers. It supports authentication by propagating a security context so that application-level code (method invocations, HTTP requests, and so on) have easy access to the context. The security context historically has been implemented with ThreadLocal. Thread-local state doesn't make a lot of sense in a reactive world. Spring Reactor, which I explored in depth in the first article, provides a Context object, which acts as a sort of dictionary. Spring Security 5.0's reactive support propagates its security context using this mechanism. Parallel, reactive-type hierarchies have been introduced to support nonblocking authentication and authorization. You don't have to worry about much of this. All you need to know is that in the reactive world, authentication is handled by an object of type ReactiveAuthenticationManager that has a simple job: given an Authentication attempt, return a Mono<Authentication> indicating whether the authentication attempt succeeded; otherwise, throw an exception.

One implementation of the ReactiveAuthenticationManager supports delegating to a user-provided object of type MapReactiveUserDetailsService. The MapReactiveUserDetailsService connects your custom username and password store to Spring Security's authentication. You might have a database table called USERS or just a hardcoded Map<K,V> of users. By default, Spring Security locks down the whole application and installs HTTP BASIC authentication. Any attempt at calling any endpoint will fail unless you provide credentials. By default, all authenticated principals can access all endpoints.

Let's introduce a handful of users with various roles. All users will have the USER role, but only a privileged few will have the ADMIN role. In this newly secured world, let's say that all users will be able to view the books they've written, but only those with the ADMIN role will be able to see all the books. **Listing 4** shows how this is done. (Let's ignore for now whether this domain makes any sense!)

Listing 4: Adding users with the Spring Security configuration

```
package com.example.libraryservice;
import org.springframework...

@Profile("security")
```

```
@Configuration
@EnableWebFluxSecurity
class SecurityConfiguration {
   @Bean
   ReactiveUserDetailsService authentication() {
   User.UserBuilder builder = User.withDefaultPasswordEncoder();
   return new MapReactiveUserDetailsService(
     builder.username("rjohnson")
         .password("pw").roles("ADMIN").build(),

     builder.username("cwalls")
         .password("pw").roles().build(),

     builder.username("jlong")
         .password("pw").roles().build(),

     builder.username("rwinch")
         .password("pw").roles("ADMIN").build());
   }

   @Bean
   @Profile("authorization")
   SecurityWebFilterChain authorization(ServerHttpSecurity http) {
     ReactiveAuthorizationManager<AuthorizationContext> am =
        (auth, ctx) ->auth.map(authentication -> {
          Object author = ctx.getVariables().get("author");
          boolean matchesAuthor =
            authentication.getName().equals(author);
          boolean isAdmin =
            authentication
               .getAuthorities()
```

```
                    .stream()
                    .anyMatch(ga ->
                      ga.getAuthority().contains("ROLE_ADMIN"));
                return (matchesAuthor || isAdmin);
              })
              .map(AuthorizationDecision::new);

          return http.httpBasic()
                .and()
                .authorizeExchange()
                .pathMatchers("/books/{author}").access(am)
                .anyExchange().hasRole("ADMIN")
                .and()
                .build();
          }
        }
```

This code installs some rules for authentication and authorization. Spring Security can talk to any number of different identify providers, but for our example I use a hardcoded map of usernames and passwords and associated roles. The ReactiveUserDetailsService bean handles username and password-based authentication.

The authorization bean uses the ServerHttpSecurity builder DSL to say that all requests have the ADMIN role unless the request is to the /books/{author} endpoint. In this case, I defer to some custom business logic (captured in the ReactiveAuthorizationManager) that inspects the path variable in the request and allows the request to proceed if the author in the path variable matches the currently authenticated user or if the currently authenticated principal is an admin (that is, it has the role ROLE_ADMIN).

In **Listing 5**, I'll try making an HTTP BASIC authenticated call to the service. I'll use curl to make the first request as jlong, a regular USER.

Listing 5: Using `curl` to access the endpoint as `jlong`

```
curl -ujlong:pw http://localhost:8080/books/jlong
```

It won't work if I try to access `http://localhost:8080/books/rwinch`, as I do in **Listing 6**. Only `ADMIN` role users can access other endpoints.

Listing 6: Using `curl` to access the endpoint as `rwinch`

```
curl -urwinch:pw http://localhost:8080/books
```

Deployment

The application is secure and observable. *Now* I can deploy it. This kind of app is a natural thing to run in a cloud provider such as Cloud Foundry, an open source cloud platform released under version 2.0 of the Apache License, which is optimized for the continuous management of applications. It sits at a level (or two) above cloud infrastructure. It is infrastructure-agnostic, running on local cloud providers such as OpenStack and vSphere and on public cloud providers such as Amazon Web Services, Google Cloud, Microsoft Azure, and Oracle Cloud. No matter where Cloud Foundry is installed, its use is basically the same. You authenticate and then tell the platform about your application workload using the `cf` command-line interface (CLI) and the `cf push` command, as shown in **Listing 7**.

Listing 7: Using the `cf` CLI to push the application

```
cf login -a $CF_API_ENDPOINT -u $CF_USER -s $CF_SPACE -o $CF_ORG
cf push -p library-service-0.0.1-SNAPSHOT.jar java-magazine-library-service
```

Once the application is up and running, you can access its public HTTP endpoints. You can provision backing services—such as message queues, databases, and caches—using `cf create-service`. You can scale the application up to multiple load-balanced instances by using `cf scale`. You can interrogate the application's metrics, its Spring Boot Actuator endpoints, its health, and much more, all from the Pivotal Apps Manager dashboard. The application is up and running now and the clients can talk to it in a secure fashion. Let's look at that client.

A (Reactive) Client

I have now stood up a REST API. I need to connect a client to the service. While I could use the Spring Framework RestTemplate, the general workhorse HTTP client that has served developers well for the better part of a decade, it's not particularly suited to potentially unbounded streams of data. The RestTemplate takes a whole payload, reading until the end of a document or file, and converts it all in one go. This isn't going to work if the client is using server-sent events or even just a really large JSON response. Instead, let's use the new Spring WebFlux WebClient, as shown in Listing 8.

Listing 8: Configuring and using an authenticated WebClient

```
package com.example.libraryclient;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import org.springframework...

@SpringBootApplication
public class LibraryClientApplication {

   @Bean
   WebClient client(
      @Value("${libary-service-url:http://localhost:8080/}")
        String url) {
           ExchangeFilterFunction basicAuth =
             ExchangeFilterFunctions
                .basicAuthentication("rwinch", "pw");
           return WebClient
             .builder()
             .baseUrl(url)
             .filter(basicAuth)
```

```
            .build();
       }
   @Bean
   ApplicationRunner run(WebClient client) {
     return args -> client.get().uri("/books")
                  .retrieve()
                  .bodyToFlux(Book.class)
                  .subscribe(System.out::println);
   }

   public static void main(String[] args) {
     SpringApplication.run(LibraryClientApplication.class, args);
   }
}

@Data
@AllArgsConstructor
@NoArgsConstructor
class Book {
   private String id;
   private String title;
   private String author;
}
```

In the code in **Listing 8**, I configure the `WebClient` and preconfigure a `baseUrl` as well as an `ExchangeFilterFunction` that authenticates the client with the service. The `WebClient` gives me a `Publisher<T>` for the response, which I then print to the console. The client lives in a separate process, so I reproduced the `Book` class definition here so that the `WebClient` can bind the JSON to it as a client-side data-transfer object. In this case, it doesn't really matter; I've got only four records in the endpoint! This web client is designed to process potentially unbounded data.

Although we're looking at this code for four records, there's no reason it shouldn't handle an unlimited amount of data.

Conclusion

In this pair of articles, I have demonstrated briefly how to build a web service with Spring Boot. I looked at Reactor, Spring Data Kay, Spring Framework 5 and Spring WebFlux, Spring Security 5, and Spring Boot 2. Spring Boot 2 makes it easy to assemble the various reactive Spring projects into an application. I didn't examine the Spring Boot Actuator, but it surfaces operational data such as metrics, application health, and more. It also has been updated to work seamlessly in a reactive world.

Spring Boot 2 sets the stage for the upcoming Spring Cloud Finchley. Spring Cloud Finchley builds on Spring Boot 2.0 and updates several different APIs to support reactive programming, service registration, and discovery works in Spring WebFlux–based applications. Spring Cloud Commons supports client-side load balancing across services registered in a service registry (such as Apache Zookeeper, HashiCorp Consul, and Netflix Eureka) for the Spring Framework `WebClient`. Spring Cloud Netflix Hystrix circuit breakers have always worked naturally with RxJava, which in turn can interop with Spring `Publisher<T>` instances.

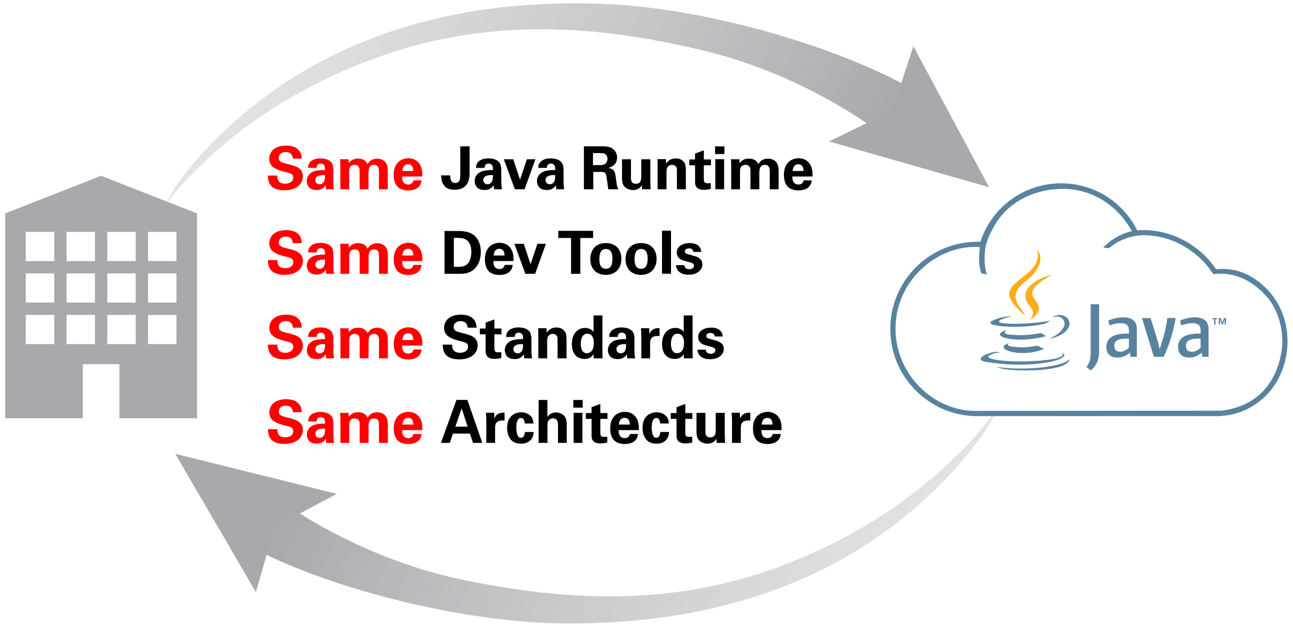
Spring Cloud Stream supports working with `Publisher<T>` instances to describe how messages arrive and are sent to messaging layers such as RabbitMQ, Apache Kafka, and Redis.

Begin your journey building reactive applications and services with Spring Boot using the Spring Initializr. The complete code for this article is on GitHub. If you have questions, find me on Twitter (@starbuxman) or by email at josh@joshlong.com. </article>

Josh Long (@starbuxman) is a Java Champion and a Spring developer advocate at Pivotal. He is the author of several books on Spring programming, and he speaks frequently at developer conferences.

Push a Button

Move Your Java Apps to the Oracle Cloud



Same Java Runtime
Same Dev Tools
Same Standards
Same Architecture

… or Back to Your Data Center

ORACLE®

SIMON **ROBERTS**

MIKALAI **ZAIKIN**

Quiz Yourself

Intermediate and advanced test questions

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

These questions rely on Java 8. We'll begin covering Java 9 and 10 in future columns, of course, and we will make that transition clear when it occurs.

Question 1 (intermediate). The `main` method that represents the entry point of a typical Java program must match a particular form.

Which of the following are true? Choose two.

A. The method must be `public`.
B. The method may be any accessibility type except `private`.
C. The method may be either an instance method or `static`.
D. The argument list must be declared exactly as `String [] args`.
E. The argument list may be declared as `String… args`.

Question 2 (intermediate). Which are true of a Java program or the JVM? Choose two.

A. The program runs substantially more slowly than an equivalent program written in a language that compiles to machine binary code, because Java bytecode is interpreted.

B. Java's `private` keyword can be used to help make programs easier to maintain by supporting the concept of encapsulation.

C. The JVM garbage collector system ensures that the programmer has zero control over and zero responsibility for releasing allocated memory.

D. The JVM's "write once, run anywhere" goal allows creating programs that produce the same results on differing hardware and under different operating systems. However, achieving this goal imposes some requirements on the programmer and the configuration of the host environment.

E. The multithreading features of the Java programming language and the JVM ensure that programs written with threads always produce the same output, even when they are run on different hardware.

Question 3 (advanced). Given this code:

```
import static java.lang.System.out;
// line n1
interface Something {
   void execute();
   default void speak() { out.println("Hello!"); }
}
public class TryThis {
   public void speak() { out.println("Bonjour!"); }
   public void go() {
      Something s = () -> this.speak(); // line n2
      s.execute();
   }
   public static void main(String[] args) {
      new TryThis().go();
   }
}
```

Which is true?

A. The output is `Hello!`

B. The output is `Bonjour!`

C. Compilation fails, but if line n1 is altered to `@FunctionalInterface` then the output is `Hello!`

D. Compilation fails, but if line n2 is altered to `Something s = () -> Something.super.speak();` then the output is `Hello!`

E. Compilation fails, but if line n2 is altered to `Something s = () -> TryThis.this.speak();` then the output is `Bonjour!`

Question 4 (advanced). Given this code:

```
StringBuilder sb = IntStream.iterate(0, x->(x+1)%26)
   .mapToObj(x->new StringBuilder("" + (char)(x+'A')))
   .parallel() // line n1
   .limit(52)
   .collect(
      ()->new StringBuilder(),
      (x,y)->y.append(x),
      (x,y)->y.append(x)
   );
System.out.println(sb);
```

What is the result?

A. Compilation fails because of an error at line n1.

B. The code throws a runtime exception because of the position of line n1.

C. The code prints out one line containing the sequence of capital letters A through Z repeated twice.

D. The code prints out one line containing the capital letters A through Z with two of each letter, but they are not necessarily in order.

E. The code prints only a single, empty line and then exits.

Answer 1. The correct answers are options A and E. The form of Java's `main` entry point has been refined a bit over the years of the language's history. For example, at one time the method was not required to be `public`. However, section 12.1.4 of the Java 8 version of the *Java Language Specification* says the following:

> "The method `main` must be declared `public`, `static`, and `void`. It must specify a formal parameter whose declared type is array of `String`."

Given these requirements, it's clear that option A must be correct, and options B and C must be incorrect. However, options D and E are yet to be resolved.

The specification demands that the formal parameter's type must be array of `String`. (If the term *formal parameter* is unfamiliar, it simply refers to the argument listed in the method's declaration; the term distinguishes that argument from the actual parameter, which is the value passed in by an invocation.) Clearly the form presented in option D defines an array of `String`, and it is actually the usual form. But option D demands this argument must be exactly as shown. It's reasonable to question this, because the name of the formal parameter (`args`, in this case) isn't usually syntactically critical. In fact, it turns out that the ellipsis (`...`) form, which defines a variable-length argument list, really causes the formal parameter to be of array type. This means that option E, while unconventional with respect to

A running Java program is unlikely to be slower than a program in a compiled language that uses the same data structures and algorithms.

the use of the ellipsis form of array declaration, is entirely valid, and we can say that not only is option D incorrect, but that option E is correct.

Answer 2. The correct answers are options B and D. This question addresses some descriptions and assumptions about the Java language and execution environment. These kinds of topics are often where marketing statements get misconstrued into misunderstandings that later contribute to program errors.

The first statement suggests that Java is slow. This is a comment we still hear quite a bit, and it's worth addressing. Benchmarks are typically difficult to write, and precise comparisons are often hard to come by. However, the origin of this comment is twofold. First, the earliest versions of Java did, in fact, interpret the bytecode, and they were significantly slower than languages that compiled directly to machine binary code. However, for a long time now, the JVM has used a mechanism that compiles regions of bytecode into native machine binary code, while optimizing it for the platform and the manner of use. Consequently, a running Java program is unlikely to be slower than a program in a compiled language that uses the same data structures and algorithms. Of course, the compilation to native binary code happens after the program has started—the system that performs this is called the just-in-time (JIT) compiler, so if the execution time is fairly short (in the region of a few seconds, for example), you won't see the benefit and the machine-specific compiled language will be faster.

> **The incorrect idea that the garbage collector renders Java programs** magically immune to memory problems has been around for a long time.

Another part of the puzzle is that starting a Java program involves first starting the JVM, and the JVM is a large and complex program that takes significant time to load up. Many programs are much smaller than the JVM. Again, this means that if you take a small machine-specific compiled program and compare it with a similar small Java program, the time from startup to completion will differ. But for a program of any real consequence, the startup time is typically lost in the overall execution time.

Given these observations, option A is incorrect.

The private keyword can indeed be used to implement encapsulation. This can be used to build a system in which an object is responsible for protecting its own "structural integrity." For example, in a Gregorian calendar, a month might have 28, 29, 30, or 31 days. If you ever find February 31 in a system, something went wrong; that would be a failure of structural integrity (which, by the way, is our made-up term, not something with formal academic significance). Anyway, if you build a Date class (one of your own making, not the ones already provided) with private day, month, and year fields, you can protect the values and ensure they are never invalid in this way. So, if the method setDayOfMonth is invoked with the value 31, it should not set the day to 31 if the month is February (what it does instead won't be discussed here). Then, if you find an invalid date, you're able to say that there is a bug in the Date class, instead of having no idea where in the entire program the problem is. The simple expedient of having a better idea of where to look makes maintenance easier, or perhaps we should say it makes maintenance less difficult. Because of this, option B is correct.

The idea that the garbage collector renders Java programs magically immune to memory problems has been around for a long time. Unfortunately, it's substantially exaggerated, and the programmer does, in fact, have some influence over the garbage collection mechanism. That influence can cause undesirable effects, including memory leaks, if mishandled.

Specifically, the garbage collector does not reclaim the memory of an object until that object is "unreachable." This means that if there is any way the program could use the object—if any usable reference to it exists in the system—the object is not collected. So, for example, if a program has a static variable that refers to a list, and it uses that list to store references to large arrays that it will never again use, the program is causing memory leaks and eventually may fail. The easiest solution is not to store the unwanted objects. In the more general case, the programmer can write null over a reference to indicate that the object referred to is no longer needed.

Given these observations, the programmer does have some control over and some responsibility for releasing objects (or at least not preventing their timely release). Therefore, option C is incorrect.

Java's "write once, run anywhere" promise is an important feature of the language's popularity. Java bytecode is a machine-like language that is not specific to any particular CPU hardware but is easy and reasonably efficient to execute on any hardware. By compiling the source language to bytecode instead of to native machine language, the result can be executed on any computer equipped with a JVM.

In addition to the design and availability of the JVM, the Java system as a whole includes extensive libraries, and these have been carefully designed to allow for equivalent behavior on different operating systems. However, it's possible for the programmer to request some actions that will not work properly across all operating systems and all hardware environments. For example, while the method `java.nio.file.Paths.get` allows you to access files and directories regardless of the format of the paths (notably, their separators) on differing hosts, it's also possible to try to access files in a way that would work on one operating system, but fail on another.

The meaning of names and the *this* and *super* keywords appearing in a lambda body, along with the accessibility of referenced declarations, are the same as in the surrounding context (except that lambda parameters introduce new names).

On the topic of paths and the differences in file system behavior among operating systems, it's possible to get into trouble because Java is (almost) entirely case-sensitive, so class `A` and class `a` would properly be two distinct classes, and they could coexist in the same package. However, in an operating system that does not distinguish case in its file system, this would fail.

More platform variations that can cause trouble if approached clumsily relate to the screen. Different systems will have different screen resolutions, and creating windows of fixed sized could make a program unusable on a small screen. Similarly, different hosts have different fonts available, with different geometries, and these issues, too, can cause trouble if the programmer fails to follow some established guidelines. The classic, but tempting, error is to position graphical items in a window using absolute coordinates, rather than using one of Java's layout managers. By doing this, critical elements can become obscured and inaccessible on some hosts.

Given these differences, it's clear that although "write once, run anywhere" generally works well, allowing you to create platform-independent programs quite easily, there are a few things you can do that would prevent code from working properly. As a result, option D is correct.

The final option considers whether Java's multithreading system can guarantee the same output on different systems. In fact, one issue that arises with threaded code is that a deliberate design effort is necessary if a programmer wants to ensure that the same output is reliably presented—even with the same machine running the same program several times. One fundamental reason is that running two threads concurrently offers no guarantee that the threads proceed at the same speed each time they run. Therefore, at the very least, messages output by these threads might appear interleaved in differing orders. Because the threading system does not intrinsically guarantee exactly consistent output even on the same machine, it certainly cannot be guaranteed on different machines. Of course, the threading libraries provide tools that allow a programmer to deliberately create such guarantees when they are needed, but this requires deliberate design by the programmer and is not simply a result of the language or the JVM. Because of this, option E must be incorrect.

Answer 3. The correct answer is option B. The essence of this question is that a lambda expression does not create a new scope for names. In particular, *Java Language Specification* section 15.27.2 notes the following:

"Unlike code appearing in anonymous class declarations, the meaning of names and the `this` and `super` keywords appearing in a lambda body, along with the accessibility of referenced declarations, are the same as in the surrounding context (except that lambda parameters introduce new names)."

In other words, the value of `this` in the body of the lambda in the code shown in question 3 does not refer to the lambda itself, but instead refers to the enclosing instance of the `TryThis` class. As a result, when the lambda invokes `this.speak()`, it calls the method defined in the `TryThis` class, not the default method in the `Something` interface. As a result, the code prints the output `Bonjour!`. Because of this, option B is correct, and option A is incorrect.

Option C suggests that the lambda cannot be created unless the target interface (Something) is annotated with @FunctionalInterface. While it is a good idea to annotate an interface that's created specifically for the purpose of supporting lambdas with this annotation, it's only a means of getting a more helpful error report from the compiler. Specifically, if an interface carries this annotation, the annotation will report an error if the interface contains more than a single abstract method. On the other hand, if the annotation is not present, the errors will show up whenever any attempt is made to create a lambda expression using the interface. It's generally more helpful to have an error reported as close to its cause as possible, rather than being reported when a consequential problem arises. However, because the existing code does not fail to compile, and because it prints Bonjour! rather than Hello!, option C is incorrect.

The syntax suggested in option D will not compile. This is an attempt to resolve ambiguous access to default methods in interfaces. However, this form cannot be used in this situation, so option D is incorrect.

Option E employs a syntax that is normally used to access shadowed elements of an enclosing class, although in this case, the class of this is already TryThis. The syntax compiles and does result in the output of Bonjour!. However, option E is incorrect because the original code does not fail to compile.

Answer 4. The correct answer is option E. This is one of those questions that tend to annoy people. It requires you to spot a subtle programming error. However, this is an error that the compiler cannot spot and that does not cause any visible problems at runtime (other than a wrong answer), and you would be hard-pressed to look up a solution in a reference document. Unless you "just know" the relevant detail, you run the risk of falling into this trap. So, let's discuss the various options.

First, the code compiles and executes without any errors being reported. The call to parallel() is merely a distraction in this question; it is completely correct but entirely irrelevant. It doesn't matter where in the sequence of the stream operations this call is placed; the call has the same effect regardless. Further, the call affects the entire stream, not merely the parts of the pipeline that follow the call. It's worth noting that the stream methods don't really execute the processing; rather they configure the various pipes and processing elements, and this is why the position of the parallel() call is not significant unless it's followed by a call to sequential(), (which would be confusing, but still not an error; such a call merely overrides the effect of the call to parallel()). Because of these observations, both options A and B are incorrect.

Using parallel() to run in parallel mode can sometimes cause the order of the items arriving at the collector to be altered by concurrency interleaving.

Having established that the code runs, you must determine what it does. The form of the stream appears ready to print out the sequence of A through Z—that is, all the capital letters in order, with the sequence repeated a second time. Certainly, the stream creates these values internally.

One question is whether the letters show up in order or not. Using parallel() to run in parallel mode can sometimes cause the order of the items arriving at the collector to be altered by concurrency interleaving, and that might call into question whether option C is correct. In fact, parallel mode isn't the same as unordered mode and in this case, the letters should not be jumbled by this effect.

Superficially then, it looks like you should expect a bunch of capital letters, but closer inspection shows the real root of this question. It turns out that the second and third arguments to the collect method are incorrectly formed.

The three-argument collect method (there's also a single argument overload) requires that the second and third arguments work to mutate a "bucket" of intermediate/incremental result data with additional input. But the bucket that is collecting the result is always the first of the two arguments to the operation, and the second argument is the data that should be merged into that bucket. In this case, the first argument is merged into the second, which will guarantee that the final result is empty. That might seem like a "tricky" question, but this mistake is easy to make if you're unfamiliar with the requirements of the collect operation, and it's hard to debug, because the operations are performed internally to the collect method implementation. Some things just have to be learned, and once you have learned about this issue (particularly if you learn it the hard way), you become pretty sensitive to the order of those arguments. Therefore, options C and D are incorrect, and option E is the correct answer.

As an aside, this is the correct form of the collect call:

```
.collect(
  ()->new StringBuilder(),
  (x,y)->x.append(y),
  (x,y)->x.append(y)
)
```

</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 the 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