Java™ magazine

By and for the Java community

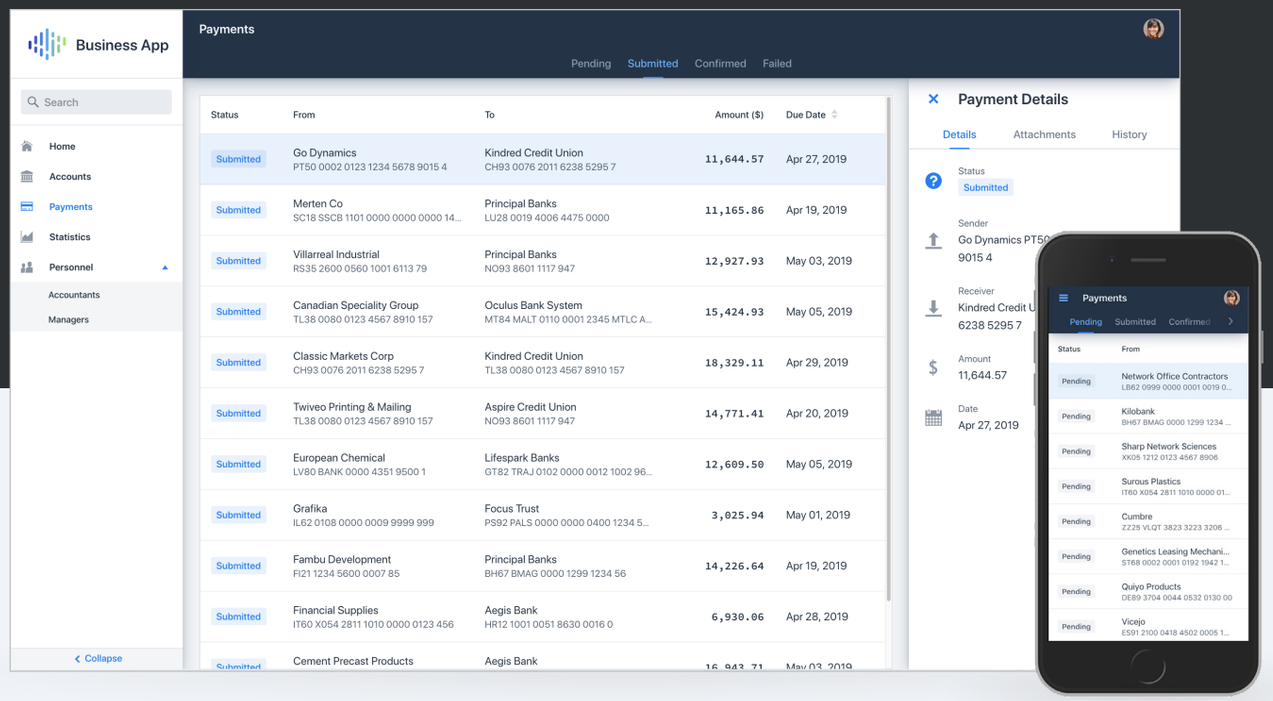
MAY/JUNE 2019

CONTAINERS

ORACLE®

vaadin

THE EFFICIENT WAY TO BUILD BUSINESS WEB APPS IN JAVA




```
@Route("payments")
public class Payments extends SplitViewFrame {
    private Grid<Payment> grid;
    private ListDataProvider<Payment> dataProvider;
    private DetailsDrawer detailsDrawer;
    @Override
    protected void onAttach(AttachEvent attachEvent) {
        super.onAttach(attachEvent);
        initAppBar();
        setViewContent(createContent());
        setViewDetails(createDetailsDrawer());
```

Create Progressive Web Apps with a familiar Java API

- ✓ Run on any device, in any browser, and at any resolution
- ✓ Create an appealing UI with a large collection of components
- ✓ Easily connect to existing Java backend code

Find out more at:

vaadin.com

//table of contents/

COVER FEATURES

OTHER FEATURES

DEPARTMENTS

COVER ART BY I-HUA CHEN

EDITORIAL

Editor in Chief Andrew Binstock
Senior Managing Editor Kimberly Brinson
Copy Editors Lea Anne Bantsari, Karen Perkins
Contributing Editors Deirdre Blake, Simon Roberts, Mikalai Zaikin
Technical Reviewer Stephen Chin

PUBLISHING

Group Publisher Karin Kinnear

ADVERTISING SALES

Tom Cometa
Mailing-List Rentals Contact your sales representative.

RESOURCES

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

ARTICLE SUBMISSION

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

SUBSCRIPTION INFORMATION

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

MAGAZINE CUSTOMER SERVICE

java@omeda.com

PRIVACY

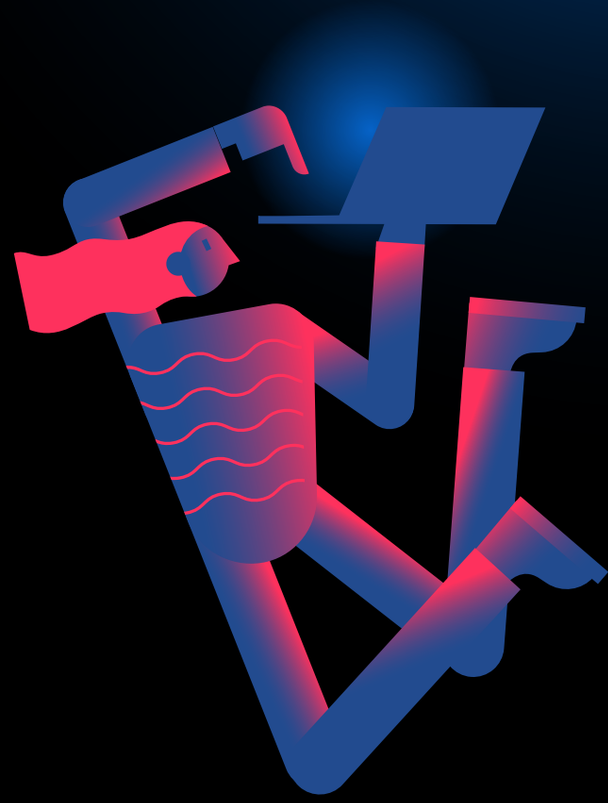
Oracle Publishing allows sharing of its mailing list with selected third parties. If you prefer that your mailing address or email address not be included in this program, contact Customer Service.

Java Magazine is published bimonthly and made available at no cost to qualified subscribers by Oracle, 500 Oracle Parkway, MS OPL-3A, Redwood City, CA 94065-1600.

IntelliJ IDEA

Capable and Ergonomic IDE for JVM

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



Download at jetbrains.com/idea

THE DRIVE
TO DEVELOP

JET
BRAINS

Improving the Reading Experience

After months of preparation, *Java Magazine* is moving to a new, responsive web format.

In many domains that serve a large audience, there is the view that "if no one is complaining, it is because you're irrelevant." In programming, this view is echoed in C++ inventor Bjarne Stroustrup's oft-quoted maxim, "There are only two kinds of languages: the ones people complain about and the ones nobody uses." I'm not a partisan of such a dismissive view. Complaints should not be taken as validation nor dismissed as irrelevant. Rather, they are useful conduits to improvement. You will note that almost all my emails and editorials as well as the back page of this magazine solicit feedback, including criticisms, because my team and I read this feedback and actively use it to improve the magazine.

By far the most common critique we receive concerns the difficulty of reading the articles in the present hosted-PDF format. The text is hard to read (especially on mobile devices) and the flipping of pages back and forth to study code is quite annoying. Although we've improved the reading experience over the years (look at one of our issues from several years ago to see how much has changed), we've not been able to create an entirely new reading experience—until now! After much preparation, we're announcing that *Java Magazine* will be published in responsive HTML hosted on a website. The design will make it far easier to read both text and code on any device and to scroll through an article rather than click through page after page.

The new format will also enable us to post content more frequently and link to it more easily—helping you find articles in searches and on social media. We will continue the subscription model by sending you notifications whenever an issue's worth of content has been posted and by providing benefits available only to subscribers. I will give you more details once the rollout is complete.

We hope you'll enjoy this new platform when it debuts this summer and, as always, I hope you'll send me suggestions and critiques if you see ways to improve the experience. Until then!

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

PHOTOGRAPH BY BOB ADLER/GETTY IMAGES

//events/

JCrete
JULY 14–19
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 on July 20. Attendees often bring their families.

PHOTOGRAPH BY STEPHEN HUGHES/ALAMY STOCK PHOTO

GOTO
JUNE 17–20
AMSTERDAM, THE NETHERLANDS
The lead singer of the rock band Iron Maiden is slated to give the opening day keynote at this enterprise software development conference. Talks on REST and API design, microservices, migrating Spring Boot apps with Kotlin, quantum computing, cloud native architecture, and Java 11 are scheduled.

DeveloperWeek NYC
JUNE 17, HIRING MIXER
JUNE 19–20, HACKATHONS
JUNE 19–20, CONFERENCE AND EXPO
More than 3000 developers, DevOps professionals, and executives are expected to gather to discuss app development, blockchain, IoT, microservices, quantum computing, AI, and more.

NFJS Central Ohio Software Symposium
JUNE 21–23
COLUMBUS, OHIO
This conference focuses on the latest technologies and best practices emerging in the modern software development and architecture space. Scheduled topics include machine learning, the evolution of Java, and microservices.

DWX Developer Week
JUNE 24–27
NUREMBERG, GERMANY
This software development conference is customarily conducted in German, but this year several talks will be conducted in English. Topics include artificial intelligence, microservices, real-time web application programming, and enterprise-grade NodeJS.

QCon
JUNE 24–26, CONFERENCE
JUNE 27–28, WORKSHOPS
NEW YORK, NEW YORK
QCon New York is a conference for senior software engineers and architects in enterprise software development. Topics this year include modern Java, machine learning for developers, data engineering, and high-performance computing.

JConf Dominicana 2019
JUNE 29
SANTIAGO, DOMINICAN REPUBLIC
JConf Dominicana is a community-

driven conference organized by the Dominican Republic Java User Group (Aka JavaDominicano). It is designed to discuss the latest about the Java ecosystem, including alternatives JVM languages, Web and Mobile development, and newest Cloud tendencies.

JConf Colombia 2019

JULY 5-6
MEDELLIN, COLOMBIA
JConf Colombia is a Java technologies conference, organized by the JUGs of Barranquilla, Bogota, Cali and Medellin. Its goal is to help developers, entrepreneurs and companies to learn new technologies related with Java, Cloud and Containers.

JConf Peru 2019

JULY 13
LIMA, PERU
JConf Peru is a conference organized by the Peru JUG, its goal is to connect developers and companies interested in Java-related technologies. Local and international speakers will share their experiences with topics such as Java and Jakarta EE.

OSCON

JULY 15–16, TRAINING AND TUTORIALS
JULY 17–18, CONFERENCE
PORTLAND, OREGON
The O'Reilly Open Source Software Conference (OSCON) returns with tutorials on Java, AI assistants, and cloud-native databases, as well as talks on blockchain, data science, and integrating security into modern software development.

ÜberConf

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

Open Source Summit Japan

JULY 17–19
TOKYO, JAPAN
Software developers, programmers, open-source maintainers, and IT professionals will gather for keynotes and talks at this conference hosted by the Linux foundation.

NFJS Lone Star Software Symposium

JULY 26–28
AUSTIN, TEXAS
This conference will focus on the latest technologies and best practices emerging in the modern software development and architecture space. Scheduled topics include the evolution of Java, Java Modules migration, and asynchronous programming in Java.

O'Reilly Artificial Intelligence Conference

SEPTEMBER 9–10, TRAINING
SEPTEMBER 10–12, CONFERENCE AND TUTORIALS
SAN FRANCISCO, CALIFORNIA
This conference centers on learning how to implement AI in real world projects. Topics include ethics, cloud and on-premise tools, new architectures and AI business implementations.

JavaZone

SEPTEMBER 11–12
OSLO, NORWAY
JavaZone is a conference for Java developers organized by javaBin, the Norwegian Java User Group. The conference has been running since 2001 and now consists of around 200 speakers and seven parallel tracks over two days. Last year's event drew 3,100 participants and 200 speakers.

Strange Loop

SEPTEMBER 12–14
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, distributed systems, and security. Talks are generally code-heavy and not process-oriented.

Oracle CodeOne

SEPTEMBER 16–19
SAN FRANCISCO, CALIFORNIA
Tutorials, sessions, keynotes, and hands-on labs will cover the future of Java and other software development concerns. There will also be discussions on Go, Rust, Python, JavaScript, SQL, R, and more.

Java Forum Nord

SEPTEMBER 24
HANNOVER, GERMANY
Java Forum Nord is a one-day, noncommercial conference in northern Germany for Java developers and decision makers. Typically featuring more than 25 presentations in parallel tracks and a diverse program, the event also provides interesting networking opportunities.

JAX London

OCTOBER 7 AND 10, WORKSHOPS
OCTOBER 8–9, CONFERENCE AND EXPO
LONDON, ENGLAND
JAX London is a four-day conference for software engineers and enterprise-level professionals, bringing together the world's leading innovators in the fields of Java, microservices, continuous delivery, and DevOps.

EclipseCon Europe

OCTOBER 21–24
LUDWIGSBURG, GERMANY
EclipseCon 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.

O'Reilly Software Architecture Conference

NOVEMBER 4–5, TRAINING
OCTOBER 5–7, CONFERENCE AND TUTORIALS
LONDON, ENGLAND
For four days, expert practitioners share new techniques and approaches, proven best practices, and technical skills. Topics include microservices, application architecture, distributed systems, domain-driven design, and more.

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

PHOTOGRAPH BY TODD VAN HOOSEAR/FLICKR

DEVELOPER EVENTS FROM THE DEVOXX & VOXXED FAMILY
COMING IN 2019

DEVOXX™

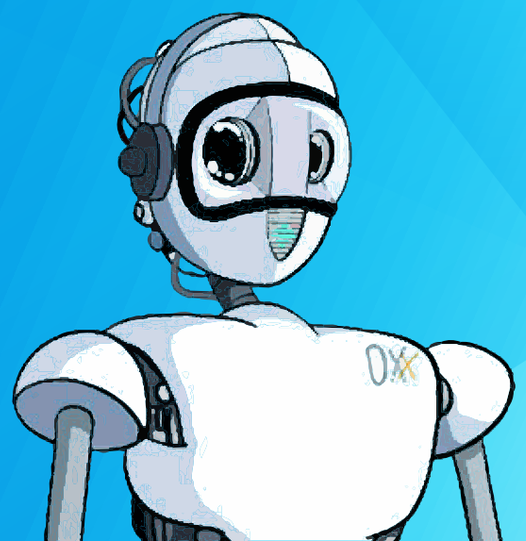
DEVOXX.COM

POLAND 24-27 JUNE

UKRAINE 1-2 NOVEMBER

BELGIUM 4-8 NOVEMBER

MOROCCO 12-14 NOVEMBER



SINGAPORE 30-31 MAY

ATHENS 7-8 JUNE

LUXEMBOURG 20-21 JUNE

BANFF 20-21 SEPTEMBER

TICINO 5 OCTOBER

MICROSERVICES, PARIS 21-23 OCTOBER

CLUJ-NAPOCA 30-31 OCTOBER

VOXXEDDAYS

VOXXEDDAYS.COM

A Developer's Guide to Containers

In the last few issues, we've presented extensive coverage of the cloud's effects on software development. In our previous issue, we explored the use of lightweight frameworks—Javalin, Micronaut, and Helidon—to create microservices, which typically are deployed in the cloud. In that issue's article on Helidon, we also showed how to package a service into a Docker container for deployment.

In this issue, we continue the theme by examining how to build apps with containers in mind and how to deploy containers. For straight Java apps, the jlink and jdeps tools are excellent solutions for creating modularized, small, self-contained apps. We discuss how to use those tools on page 25.

If very fast startup time is a concern, then consider the GraalVM platform. It is written in Java but compiles Java code to an executable format. We've discussed GraalVM in past issues, but this article focuses on the latest features and their use in creating small executables with native-code startup speed. Finally, if you're straddling the Dev and Ops sides of DevOps, you surely have seen that most containers are managed with the open source Kubernetes platform. In our lead feature, we give you a full introduction to Kubernetes and all the information you need to start working with managing your containerized apps.

In addition, we explore what's new in the recent release of Java 12, and we examine a major upgrade to Java Card, which in all senses is the very smallest container for a Java app. In addition, we have our usual quiz and our events calendar. Finally, future issues of this magazine will look materially different from what you're used to. Please see the editorial in this issue for details. Thank you!

ART BY I-HUA CHEN

JESSE BUTLER

Getting Started with Kubernetes

Automate the deployment, scaling, and management of containerized applications and services.

Today, monolithic applications are considered an antipattern and, for most use cases, the cloud is the deployment platform of choice. These two changes equate to far more than booting virtual machines on other people's computers. Effectively leveraging the power and scalability of the cloud means departing from yesterday's monoliths and adopting new architectures and development practices.

A microservices architecture is the emerging standard for delivering applications and services in the cloud. Applications are broken down into loosely coupled discrete programs, each of which has a specific responsibility. Adopting this sort of architecture allows teams to work more independently as they push out their specific features—without being tied to a cumbersome whole-organization roadmap. In addition, with discrete software components, testing can be simplified and deployments streamlined.

Microservices adoption comes with its own set of challenges. Creating and deploying a few services on a virtual machine is a first step, but how do you manage the full software lifecycle? Container adoption has been primarily driven by this need. Using containers addresses several concerns for a microservices architecture, such as the following:

- Application software is decoupled from the host environment, providing great portability.
- Containers are lightweight and fairly transparent, thereby enabling scalability.
- Software is packaged along with its dependencies.

Given these benefits, containers are an excellent choice for the packaging and deployment of microservices. But containers are not magic. Under the covers, it's all still software, and you need a way to deploy, manage, and maintain your containers at scale. Where developers once had a single monolith to monitor and maintain, they now might have dozens or hundreds of

services. This is where Kubernetes comes into play.

Kubernetes is an open source platform for automating the deployment, scaling, and management of containerized applications and services. It was developed in response to the challenges of deploying and managing large fleets of containers at Google, which open sourced the project and donated it to the Cloud Native Computing Foundation (CNCF). That foundation fosters the cloud native computing ecosystem. Kubernetes was the first graduated project for CNCF, and it became one of the fastest growing open source projects in history. Kubernetes now has more than 2,300 contributors and has been widely adopted by companies large and small, including half of the Fortune 100.

Getting Started with Kubernetes

How to get started? Kubernetes has a large ecosystem of supporting projects that have sprung up around it. The landscape can be daunting, and looking for answers to simple questions can lead you down a rabbit hole, which can easily make you feel like you're woefully behind. However, the first few steps down this path are simple, and from there you can explore more-advanced concepts as your needs dictate. In this article, I demonstrate how to:

- Set up a local development environment with Docker and Kubernetes
- Create a simple Java microservice with Helidon
- Build the microservice into a container image with Docker
- Deploy the microservice on a local Kubernetes cluster
- Scale the microservice up and down on the cluster

To follow this tutorial, you will need to have some tools installed locally:

- Docker 18.02 or later
- Kubernetes 1.7.4 or later
- JDK 8 or later
- Maven 3.5 or later

You can install the latest version of each of these requirements on macOS, Linux, or Windows.

If you don't have Docker installed already, refer to the *Getting Started with Docker* guide and follow the instructions for your platform. You'll want to be familiar with Docker basics.

You can use any Kubernetes cluster available to you, but use Minikube to follow along with this article. Minikube runs a single-node Kubernetes cluster inside a virtual machine on your local system, giving you just enough to get started. Follow the Minikube installation documentation if you don't have Minikube installed already.

In Kubernetes, a service is an abstraction that defines a way to access a pod or a set of pods.

Now that you have Docker ready to go and can spin up a local Kubernetes cluster with Minikube, you will need an example microservice to work with. Inspired by the Helidon article on microservices frameworks in the March/April issue of *Java Magazine*, I use Helidon in this tutorial to create a simple microservice.

Build a Basic Microservice

Get started quickly with Helidon by creating a new project that uses the Helidon quickstart Maven archetype. The following will get you up and running with a basic starter project:

```
$ mvn archetype:generate -DinteractiveMode=false \
    -DarchetypeGroupId=io.helidon.archetypes \
    -DarchetypeArtifactId=helidon-quickstart-se \
    -DarchetypeVersion=1.0.1 \
    -DgroupId=io.helidon.examples \
    -DartifactId=helidon-quickstart-se \
    -Dpackage=io.helidon.examples.quickstart.se
```

Change into the `helidon-quickstart-se` directory and build the service:

```
$ cd helidon-quickstart-se
$ mvn package
```

That's it—you now have a working example of a microservice. The project will have built an application JAR file for your sample microservice. Run it to check that everything is working properly:

```
$ java -jar ./target/helidon-quickstart-se.jar
[DEBUG] (main) Using Console logging
2019.03.20 12:52:46 INFO io.helidon.webserver.NettyWebServer
Thread[nioEventLoopGroup-2-1,10,main]: Channel '@default'
started: [id: 0xbdfca94d, L:/0:0:0:0:0:0:0:0:8080]
WEB server is up! http://localhost:8080/greet
```

Use curl in another shell and put the service through its paces:

```
$ curl -X GET http://localhost:8080/greet
{"message":"Hello World!"}
$ curl -X GET http://localhost:8080/greet/Mary
{"message":"Hello Mary!"}
$ curl -X PUT -H "Content-Type: application/json" -d '{"greeting" : "Hola"}'  /
http://localhost:8080/greet/greeting
$ curl -X GET http://localhost:8080/greet/Maria
{"message":"Hola Maria!"}
```

This isn't a very exciting service, but it serves the purpose of an example to build a container with. Before you carry on to Kubernetes, build a Docker image of your microservice. Helidon provides an example Dockerfile; you can build your image simply via

```
docker build -t helidon-quickstart-se target
```

You can now see what Kubernetes can do for you.

Moving to Kubernetes

With Docker, you can build container images, create containers, and manage images and containers locally. Kubernetes comes into play when it is time to deploy containers in production and at scale.

In Kubernetes, containers are deployed in groups of related containers called *pods*. A pod is a deployable unit and may contain more than one container. For example, a pod might contain two containers: one running a web server and the other being a logging service for the server. Later, you will create a pod for your microservice and it will have just one container: an instance of your `helidon-quickstart-se` image.

Part of the role of Kubernetes is to ensure your application services are up and running. You describe what should be run and monitored by defining a *deployment*. Kubernetes will monitor the health of pods within a deployment, and it will restart the containers within a pod if they stop running. Deployments are the best way to deploy and scale pods on Kubernetes.

Let's start by testing your local Kubernetes cluster with a simple example. First, to get your local cluster spun up, invoke the `minikube start` command. This command provides a running status of its progress. Wait for the completion message before moving on.

```
$ minikube start
  minikube v0.35.0 on darwin (amd64)
  ...
  Done! Thank you for using minikube!
```

The Minikube installation comes with `kubectl`, the primary interface to Kubernetes. Like the Docker client, it is a powerful multipurpose tool for interacting with all aspects of your cluster and the containers deployed to it.

Use `kubectl create` to create a simple deployment with a prebaked "hello world" example. Then use `kubectl get` to show your deployment and the pods within it. You'll see something like this:

```
$ kubectl create deployment hello-node \
  --image=gcr.io/hello-minikube-zero-install/hello
deployment.apps/hello-node created
$ kubectl get deployments
NAME         READY   UP-TO-DATE   AVAILABLE   AGE
hello-node   0/1     1            0           27s
$ kubectl get pods
NAME                          READY   STATUS    RESTARTS   AGE
hello-node-64c578bdf8-5b7jm   1/1     Running   0          10m
```

By default, Kubernetes assigns a pod an internal IP address that is accessible only from within the cluster. To enable external access to the containers running within a pod, you will expose the pod as a *service*. In Kubernetes, a service is an abstraction that defines a way to access a pod or a set of pods.

Create a simple `LoadBalancer` service with `kubectl expose`. This will enable external access to your service through a load balancer.

```
$ kubectl expose deployment hello-node --type=LoadBalancer --port=8080
service/hello-node exposed
$ kubectl get services
NAME         TYPE           CLUSTER-IP      EXTERNAL-IP PORT(S)
hello-node   LoadBalancer   10.104.108.47   <pending>   8080:30631/TCP
kubernetes   ClusterIP      10.96.0.1       <none>      443/TCP
```

[The output includes other fields that are omitted to fit on this page. This is also true of some subsequent output listings. —*Ed.*]

On a cloud provider or other managed Kubernetes platform, this action would result in the allocation of a load balancer resource, and its IP address would be shown in the `EXTERNAL-IP` column. In Minikube, use the `service` command, which will open a browser and show you that the service is working:

```
$ minikube service hello-node
```

With your sanity test done, you can tear things down and clean up. To do this, use the `kubectl delete` command:

```
$ kubectl delete service hello-node
service "hello-node" deleted
$ kubectl delete deployment hello-node
deployment.extensions "hello-node" deleted
```

Now that you have a local Kubernetes cluster and know that it is working, it's time to put your own image to work.

Deploying to Kubernetes

In the previous section, you created a simple deployment with the command-line arguments to `kubectl create`. Typically, you'll need to describe your deployments in more detail, and for that you can pass a YAML file to `kubectl`. This step allows you to define all aspects of your Kubernetes deployments. Another benefit of defining your deployments in YAML is that the files can be kept in source control along with your other project source code.

The Helidon starter project includes some boilerplate configuration for both a deployment and a service in the `target/app.yaml` file. Open this file in your favorite editor and let's examine its contents.

```
$ cat target/app.yaml
kind: Service
apiVersion: v1
metadata:
  name: helidon-quickstart-se
  labels:
    app: helidon-quickstart-se
```

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

This content might appear at first to be just a wall of metadata, but by reading through it, you can see that it defines a deployment that will consist of a pod with a `helidon-quickstart-se`

container in it and a service that makes it available externally. Notice the service uses an `app` by the same name as the deployment's `app` label.

Note the image name in the deployment specification calls for your local image name: `helidon-quickstart-se`. When you deploy this in Kubernetes, the runtime will look for this image locally first. You *are* using a local image, in that Docker is running on your local machine. But the issue is that Minikube is running its own instance of Docker locally in its virtual machine. The image you built earlier will not be found there.

Minikube has a handy solution for this issue: the `docker-env` command. This command prints a set of shell environment variables that will direct a Docker client to use the Docker server where Minikube is running. Simply invoke this command inside an `eval` on Linux, UNIX, or Mac (consult the documentation for the Windows equivalent), and it will set the variables in your current shell environment.

```
$ eval $(minikube docker-env)
```

Now when you invoke the `docker` client, it will connect to Docker running in the Minikube virtual machine. Note that this will be configured only in your current shell session; it's not a permanent change.

Now you can build the image as you did above, but this time it will be built in Minikube's virtual machine, and the resulting image will be local to it:

```
$ docker build -t helidon-quickstart-se target
Sending build context to Docker daemon  5.877MB
Step 1/5 : FROM openjdk:8-jre-slim
8-jre-slim: Pulling from library/openjdk
f7e2b70d04ae: Pull complete
05d40fc3cf34: Pull complete
b235bdb95dc9: Pull complete
9a9ecf5ba38f: Pull complete
91327716c461: Pull complete
```

```
Digest: sha256:...
Status: Downloaded newer image for openjdk:8-jre-slim
 ---> bafe4a0f3a02
Step 2/5 : RUN mkdir /app
 ---> Running in ec2d3dad6e73
Removing intermediate container ec2d3dad6e73
 ---> a091fb56d8c5
Step 3/5 : COPY libs /app/libs
 ---> a8a9ec8475ac
Step 4/5 : COPY helidon-quickstart-se.jar /app
 ---> b49c72bbfa4c
Step 5/5 : CMD ["java", "-jar", "/app/helidon-quickstart-se.jar"]
 ---> Running in 4a332d65a10d
Removing intermediate container 4a332d65a10d
 ---> 248aaf1a5246
Successfully built 248aaf1a5246
Successfully tagged helidon-quickstart-se:latest
```

With your image built local to your Kubernetes cluster, you can now create your deployment and service:

```
$ kubectl create -f target/app.yaml
service/helidon-quickstart-se created
deployment.extensions/helidon-quickstart-se created

$ kubectl get pods
NAME                                    READY   STATUS    RESTARTS
helidon-quickstart-se-786bd599ff-n874p   1/1     Running   0
$ kubectl get service
NAME             TYPE       CLUSTER-IP    EXTERNAL-IP PORT(S)
helidon-quick... NodePort   10.100.20.26  <none>      8080:31803/TCP
```

```
kubernetes       ClusterIP  10.96.0.1     <none>       443/TCP
```

There is one last thing to note before you test your newly deployed Kubernetes service. Earlier, you created a service with type `LoadBalancer`. You can access your pod through the service's external IP address, which is configured on a load balancer. This is one type of service providing external access to services running within your cluster. Another type is `NodePort`, which this service is using. A node port exposes the service on a mapped port across all of the cluster nodes.

Minikube once again comes through with a handy command. You can use the `service` command again, this time with the `--url` option to retrieve a service access URL. Invoke this command and then test your service with the URL it returns:

```
$ minikube service helidon-quickstart-se -url
http://192.168.99.101:31803
$ curl -X GET http://192.168.99.101:31803/greet
{"message":"Hello World!"}
```

You have now deployed a microservice to Kubernetes. Fantastic! Before you head off to do amazing things with containers, let's examine a few more basics.

Keeping an Eye on Things

When you ran your service at the command line, you saw the output of the web server starting up. This output is also captured by the container runtime and can be seen with the `kubectl logs` command. If things don't appear to be running correctly, this is a good place to check first.

```
$ kubectl logs helidon-quickstart-se-786bd599ff-n874p
[DEBUG] (main) Using Console logging
2019.03.23 01:00:53 INFO io.helidon.webserver.NettyWebServer
  Thread[nioEventLoopGroup-2-1,10,main]: Channel '@default'
```

```
  started: [id: 0x9f01de18, L:/0.0.0.0:8080]
WEB server is up! http://localhost:8080/greet
```

This command is especially useful when you are doing development and working with a local cluster. At scale, there are several ways to collect, store, and make use of log data. Open source projects as well as fully automated commercial offerings exist, and an online search will show many options to explore.

Scaling to Meet Demand

Suppose your service is part of an application and is responsible for saying "Hello" to users. Your team anticipates a spike in use tomorrow, so you will scale up your service to handle more users. Scaling your deployments up and down is a core feature of Kubernetes. To scale, simply modify the number of replicas defined in your deployment specification and apply the changes via `kubectl apply`.

Edit your `target/app.yaml` file and apply the changes now. Start by scaling the number of replicas up from one to five.

```
$ grep replicas target/app.yaml
replicas: 5
$ kubectl apply -f target/app.yaml
service/helidon-quickstart-se unchanged
deployment.extensions/helidon-quickstart-se configured
```

You should now see five pods deployed where there was only one before. Note that because Kubernetes configuration is declarative, only the changes that are required are committed to the cluster. In this case, four pods are added to the deployment.

```
$ kubectl get pods
```

```
NAME                                     READY   STATUS    RESTARTS
helidon-quickstart-se-786bd599ff-5gm29   1/1     Running   0
helidon-quickstart-se-786bd599ff-fkg8g   1/1     Running   0
helidon-quickstart-se-786bd599ff-g7945   1/1     Running   0
helidon-quickstart-se-786bd599ff-h6c5n   1/1     Running   0
helidon-quickstart-se-786bd599ff-n874p   1/1     Running   0
```

You can just as easily scale your deployment back down; once again edit your deployment specification and apply the change:

```
$ grep replicas target/app.yaml
replicas: 2
$ kubectl apply -f target/app.yaml
service/helidon-quickstart-se unchanged
deployment.extensions/helidon-quickstart-se configured
$ kubectl get pods
NAME                                     READY   STATUS    RESTARTS
helidon-quickstart-se-786bd599ff-h6c5n   1/1     Running   0
helidon-quickstart-se-786bd599ff-n874p   1/1     Running   0
```

There are more-advanced scaling features in Kubernetes, such as the Horizontal Pod Autoscaler, as well as features in managed cloud platforms to automatically scale up node resources on demand.

Next Steps

While it is beyond the scope of this article, there are several more-advanced Kubernetes features that are now within your grasp. The Kubernetes documentation is a great resource to start with.

When you are done working with your service, you can remove it from your cluster by using `kubectl delete`:

```
$ kubectl delete service helidon-quickstart-se
service "helidon-quickstart-se" deleted
$ kubectl delete deployment helidon-quickstart-se
deployment.extensions "helidon-quickstart-se" deleted
```

Along with the extensive documentation, there are several free online training options to take advantage of. The Kubernetes community is very welcoming and helpful as well. In addition, a great source for learning, ideas, and support can be found in the Kubernetes Slack channel. Issues and pull requests are always welcome in the project on GitHub.

Conclusion

Containers are a powerful abstraction for microservices deployment. They allow you to decouple your service from the host environment, making it portable and scalable. As you can see from this brief introduction, Kubernetes makes containers manageable at scale.

Of course, these examples are just the beginning. Dig into the documentation and learn about more-advanced concepts. Welcome to Kubernetes development! </article>

Jesse Butler (@jlb13) is a cloud native advocate with Oracle Cloud Native Labs. He's been working with containers for several years, first in Solaris and later in Linux. Most recently, he's been focused on Kubernetes, containers, serverless, and other cloud native technologies.

NICOLAS FRÄNKEL

Containerizing Apps with jlink

A JDK utility greatly facilitates containerizing your applications.

If you have tried working with Java modules, you might have realized that modularization is not easy. The first hurdle might be modularizing your own application, but many issues are due to the current state of modularization of third-party libraries. This is unfortunate, because once an application is modularized, it can be distributed as a standalone executable with a trimmed-down version of the JDK. In this era of containerization, that means small Docker images.

In this article, I explain how to use jlink, which is a command-line utility available since Java 9, to create easy-to-containerize Java executables. I'll start with a quick overview of modules. Then I'll demonstrate how to use jlink to create standalone executables and how beneficial jlink can be when used with Docker containers.

The complete source code and files for this article are available on GitHub. The source code and configuration files for the larger project in this article can also be found on GitHub.

Modularization and jlink

A class `A` may make use of other classes such as `java.util.List` at compile time (the compiler checks the dependency is available on the compiling classpath) or at runtime (in which case the JRE tries to resolve the same dependency on the runtime classpath).

One issue that occurs is that the JRE delivers many classes; some of them might not be used by the application, but they are bundled anyway. For example, application servers that run in headless mode still bundle graphical packages such as `javax.swing`.

Another issue that arises from dependence on the JRE is how visibility is managed in Java. For class `A` in package `ch.frankel.a` to be visible from class `B` in package `ch.frankel.b`, class `A` must have public visibility. With that in mind, it's impossible for third-party JAR libraries to cleanly separate their API classes and their internal classes into different packages.

Historically, packages that were meant to be internal relied upon implicit naming, such as `ch.frankel.c.internal`. However, there was no technical way to enforce this constraint.

Java 9 tried to address this problem by providing another way to manage visibility: modules. There are several kinds of modules:

- System modules: These modules are provided by the JVM.
- Application modules: An application can be made into an application module by providing a module-info class at its root.
- Automatic modules: By adding an `Automatic-Module-Name` entry in a JAR's `MANIFEST.MF`, the specified module will be treated as an automatic module.
- Unnamed module: A JAR that is not a system module, an application module, or an automatic module is an unnamed module.

A Java 9 (or later) application that has been modularized makes use of a `module-info` file located at the root. This file is a manifest that contains the module name and declares which module dependencies are required. At runtime, the loader reads that manifest to load only the modules that are necessary.

To reduce the overall size of an image, you can take advantage of the module system and distribute only the required modules of the JDK.

With this design, it's possible to eliminate unnecessary modules that are part of the JDK, which is the mission of jlink. As the official documentation states, "You can use the jlink tool to assemble and optimize a set of modules and their dependencies into a custom runtime image."

jlink enables you to use the underlying module configuration of an application to deliver a custom JRE along with the application. Using the same mechanics, it also allows you to create an executable out of the application, so the deliverable is completely self-sufficient and doesn't rely on the target system having a compatible JRE.

Laying the Foundation

Let's examine jlink starting with the simplest possible application: Hello World. Here it is in all its glory:

```
//Main.java
public class Main {

    public static void main(String[] args) {
        System.out.println("Hello world");
    }
}
```

It is hard to argue that Docker is not the most popular container distribution channel nowadays. To distribute this Hello World application, it would be a huge benefit to use Docker. Because my intention is to both create a single Dockerfile and keep the final image as small as possible, a multistage build is needed.

As a reminder, a multistage Docker build allows you to chain stages in such a way that a later stage can reuse build results from previous stages. In addition, each stage can inherit from different base images and you can name each stage, because it is easier to reference a stage by name than by index. The main benefit of multistage builds is the ability to use the most relevant image in each stage, so you can have the smallest resulting image at the end of the build process.

Here's an example Dockerfile showing how to create an image for Hello World by using Maven. I assume the project has a Maven-compatible structure:

```
Dockerfile
FROM maven:3.6-jdk-12-alpine as build

WORKDIR /app

COPY pom.xml .
COPY src src

RUN mvn package
```

```
FROM openjdk:12-alpine

COPY --from=build /app/target/jlink-ground-up-1.0-SNAPSHOT.jar \
                  /app/target/jlink-ground-up.jar

ENTRYPOINT ["java", "-jar"]
CMD ["/app/target/jlink-ground-up.jar"]
```

In this file, the second line identifies the first stage of the multistage build, which uses a Maven image and is labeled `build`. The `mvn package` command generates the JAR, using the default name that is `jlink-ground-up-1.0-SNAPSHOT.jar`.

In the line that begins with `FROM`, you see the second and last stage of the build. That line uses one of the smallest images possible, an Alpine distribution of Linux. There's no Alpine image for Java 11, but there is one for Java 12. Unfortunately, no JRE is available, only a JDK. The `COPY` statement that's next reuses the JAR file that is the output of the first stage.

Then, you create the image:

```
$ docker build -t jlink:1.0 .
```

The main issue with this approach is that the Docker image is huge, because it embeds the whole JDK, even for a Hello World application. In fact, the size of the application is negligible compared to the size of OpenJDK 12:

```
$ docker images
REPOSITORY    TAG          IMAGE ID       CREATED              SIZE
openjdk       12-alpine    8f180304fad9   7 days ago           336MB
jlink         1.0          7c612235f308   About a minute ago   336MB
```

To reduce the overall size of an image, you can take advantage of the module system and distribute only the required modules of the JDK.

Distributing a Custom Launcher

The Hello World application is so simple that no module except `java.base` is required. This module is automatically required, just as `java.lang` packages are implicitly imported.

To distribute a custom, smaller executable, the first step is to migrate the application to the module system. As stated earlier, jlink can work only with modularized applications, because it relies on the `module-info` file.

Because this app requires only the `java.base` module, creating a `module-info.java` module descriptor is straightforward:

```
// module-info.java
module ch.frankel.jlink {
    exports ch.frankel.blog.jlink;
}
```

jlink's main feature is to optimize an application to keep only the modules that will be used. Furthermore, it can be used to create a standalone executable out of the optimized version of the application. Because our app is now taking advantage of the module system, it becomes possible to create this dedicated launcher.

However, jlink requires an existing JAR to work its magic:

```
$ mvn clean package
```

Everything is now set, so it's time to use `jlink`. Be aware that like the `java` or `javac` commands, jlink requires options to be specified. Here's the command to create a custom executable for Hello World:

```
$ jlink --add-modules ch.frankel.blog.jlink \
    --module-path ${JAVA_HOME}/jmods:target/jlink-ground-up-1.1.jar \
    --output target/jlink-image \
    --launcher hello=ch.frankel.jlink/ch.frankel.blog.jlink.Main
```

Let's examine the options.

- The first line defines the modules that are required via their names. The module's name of our application should be set.
- The second line specifies the module path. While pre-Java 9 applications use the classpath, module-compatible applications use the module path. Just as with the classpath, the module path references path elements to search for dependent modules. For now, the path to every module including those provided by the JDK and the JAR file, must be referenced.
- The third line specifies the output folder.
- The final line specifies the entry point of the custom distribution. Its format consists of several parts: the final executable name, an = sign, the module name, and a / followed by the fully qualified class name of the `Main` class.

The good news is that there's an existing Maven plugin to help you declaratively manage the module path: the ModiTect plugin.

Once you have created the distribution, you can launch it using this command:

```
$ target/jlink-image/bin/hello
```

As expected, this command prints `Hello world` to standard out.

Just as before, the goal is to wrap this custom distribution in a Docker image. Let's adapt the Dockerfile accordingly, as follows:

```
FROM maven:3.6-jdk-12-alpine as build

WORKDIR /app
COPY pom.xml .
COPY src src
RUN mvn package && \
    jlink --add-modules ch.frankel.jlink \
     --module-path ${JAVA_HOME}/jmods:target/jlink-ground-up-1.1.jar \
```

```
    --output target/jlink-image \
    --launcher hello=ch.frankel.jlink/ch.frankel.blog.jlink.Main

FROM alpine:3.8

COPY --from=build /app/target/jlink-image /app
ENTRYPOINT ["/app/bin/hello"]
```

The next question is whether this extra step had any effect on the size of the resulting Docker image. Let's compare it with the previous build that was created without jlink:

```
REPOSITORY      TAG     IMAGE ID        CREATED              SIZE
jlink           1.0     7c612235f308    About a minute ago   336MB
jlink           1.1     e590fb6697e7    14 hours ago         53MB
```

Wow; that's a whopping 283 MB savings from the previous unmodularized application! If 53 MB seems a bit much for a Hello World application, remember that the distribution contains all the power of the JVM, including the just-in-time (JIT) compiler and garbage collection management. If necessary, pause to savor this victory, and when you are ready, proceed to the next section.

Industrializing the Generation of the Custom Launcher

The previous jlink command is verbose. Moreover, it will become more verbose when the number of module dependencies increases. The situation is quite similar to that of the Java compiler: In day-to-day life, developers rarely invoke the compiler directly; instead they favor the use of a build tool such as Maven. With Maven, dependencies are declared in the POM file, and the Java compiler plugin manages the classpath for you. It would be very awkward to manage the classpath manually at every compilation.

The good news is that there's an existing Maven plugin to help you declaratively manage the module path: the ModiTect plugin. I use this plugin in the rest of this article.

Among other features, the plugin offers a `create-runtime-image` goal that creates the launcher. Here is a POM snippet that creates the custom launcher as described earlier but in a repeatable way:

```
<plugin>
  <groupId>org.moditect</groupId>
  <artifactId>moditect-maven-plugin</artifactId>
  <version>1.0.0.Beta2</version>
  <executions>
    <execution>
      <id>create-runtime-image</id>
      <phase>package</phase>    <!-- comment 1 -->
      <goals>
        <goal>create-runtime-image</goal>  <!-- comment 2 -->
      </goals>
<configuration>
  <modulePath>            <!-- comment 3 -->
    <path>                <!-- comment 4 -->
      ${project.build.directory}/${project.artifactId}-
      ${project.version}.${project.packaging}
    </path>
  </modulePath>
  <modules>
    <module>ch.frankel.jlink</module>   <!-- comment 3 -->
  </modules>
  <launcher>
    <name>hello</name>                  <!-- comment 3 -->
    <module>
      ch.frankel.jlink/ch.frankel.blog.jlink.Main
    </module>                           <!-- comment 3 -->
  </launcher>
```

```
        <outputDirectory>
            ${project.build.directory}/jlink-image
        </outputDirectory>                    <!-- comment 3 -->
      </configuration>
    </execution>
  </executions>
</plugin>
```

The line that contains `comment 1` binds execution to the package phase. The line that contains `comment 2` calls the `create-runtime-image` goal. The lines that contain `comment 3` will be translated to a jlink command-line option. And the line that contains `comment 4` is followed by two lines that you should enter as a single line (including the hyphen); the two lines are shown separately so they fit on the page.

Table 1 shows how the declarative configuration maps to the jlink command-line options:

XML	JLINK OPTION
`<modulePath>`	`--module-path`
`<modules>`	`--add-modules`
`<name>`	FIRST PART OF `--launcher`
`<module>`	SECOND PART OF `--launcher`
`<outputDirectory>`	`--output`

Table 1. How the configuration maps to the jlink options

With that information added to the POM, the Dockerfile can be simplified further:

```
Dockerfile
FROM maven:3.6-jdk-12-alpine as build

WORKDIR /app
```

```
COPY pom.xml .
COPY src src
RUN mvn package

FROM alpine:3.8

COPY --from=build /app/target/jlink-image /app
ENTRYPOINT ["/app/bin/hello"]
```

At this point, you have a repeatable build process for creating custom distributions.

Adding Module Dependencies

Let's beef up the application and improve the code. As you probably know, it's unwise to use `System.out.println()` statements: they are not configurable, so if they are used for debugging purposes, they will be written even in production. Let's replace this log statement with a call to a proper logging framework.

As my logging framework, I will use the Simple Logging Facade for Java (SLF4J), which is modularized. This choice requires me to add two dependencies: the API and a single implementation.

To do that, add the following to the POM file:

```
<dependencies>
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-api</artifactId>
        <version>1.8.0-beta2</version>
    </dependency>
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-simple</artifactId>
```

```
            <version>1.8.0-beta2</version>
            <scope>runtime</scope>
        </dependency>
    </dependencies>
```

And update the `module-info.java` file:

```
module ch.frankel.jlink {
    requires org.slf4j;
    exports ch.frankel.blog.jlink;
}
```

To use jlink, you also need to add SLF4J to the `--module-path`, as follows:

```
<configuration>
  <modulePath>
    <path>  <!--enter the next two lines as a single line -->
      ${project.build.directory}/${project.artifactId}-
      ${project.version}.${project.packaging}
    </path>
    <path> <!--enter the next two lines as a single line -->
      ${settings.localRepository}/org/slf4j/slf4j-api/
        1.8.0-beta2/slf4j-api-1.8.0-beta2.jar
    </path>
  </modulePath>
</configuration>
```

As the number of dependencies grows, this approach can quickly become tedious. It's easy to forget a dependency, and it's cumbersome to upgrade the version number in the `<dependencies>` section and here as well.

A better alternative is to copy every dependency into a dedicated directory, and use that directory as a part of the module path. The following code shows how you can configure the build process in the POM file to automatically do that during each run:

```
<plugin>
  <artifactId>maven-dependency-plugin</artifactId>
  <version>3.1.1</version>
  <executions>
    <execution>
      <id>copy</id>
      <phase>package</phase>
      <goals>
        <goal>copy-dependencies</goal>
      </goals>
      <configuration>
        <outputDirectory>
          ${project.build.directory}/modules
        </outputDirectory>
      </configuration>
    </execution>
  </executions>
</plugin>
```

Note that you can configure the module path once and for all, as follows, because every dependency will be copied to the `modules` folder:

```
<modulePath>
  <path> <!--enter the next two lines as a single line -->
   ${project.build.directory}/${project.artifactId}-
   ${project.version}.${project.packaging}
  </path>
```

```
  <path>${project.build.directory}/modules</path>
</modulePath>
```

Adding Nonmodule Dependencies

Unfortunately, jlink works only with modules and it will fail if not all dependencies are modules. The only way to fix that is to craft a custom `module-info.java` file, compile it, and update the dependent JAR with it. You can use the jdeps utility, which is part of the JDK, to determine the contents of the module file, namely the dependencies that need to be declared.

Here's an example usage:

```
$ jdeps --generate-module-info \
        . \
        $M2_REPO/org.apache…/commons-lang3-3.8.1.jar
```

The first line specifies that jdeps should generate the `module-info.java` file, the second line is the output directory, and the third line is the target JAR file to be analyzed. [The path was shortened to fit the page. —*Ed.*]

This command generates the following file:

```
module org.apache.commons.lang3 {
    requires transitive java.desktop;

    exports org.apache.commons.lang3;
    exports org.apache.commons.lang3.arch;
    exports org.apache.commons.lang3.builder;
    exports org.apache.commons.lang3.concurrent;
    exports org.apache.commons.lang3.event;
    exports org.apache.commons.lang3.exception;
    exports org.apache.commons.lang3.math;
    exports org.apache.commons.lang3.mutable;
```

```
    exports org.apache.commons.lang3.reflect;
    exports org.apache.commons.lang3.text;
    exports org.apache.commons.lang3.text.translate;
    exports org.apache.commons.lang3.time;
    exports org.apache.commons.lang3.tuple;
}
```

Note that the folder's name is not randomly chosen: It's taken from the `Automatic-Module-Name` attribute in the JAR's manifest. This design allows a JAR to have a stable module name. If the attribute is missing, the module system will automatically infer a module name based on the JAR's name, which might not be suitable.

jdeps doesn't handle the compilation, but the ModiTect plugin provides a goal for achieving that. Let's update the POM file accordingly:

```
<plugin>
<groupId>org.moditect</groupId>
<artifactId>moditect-maven-plugin</artifactId>
<version>1.0.0.Beta2</version>
<executions>
  <execution>
    <id>add-module-info</id>
    <phase>package</phase>           <!-- comment 1 -->
    <goals>
      <goal>add-module-info</goal>  <!-- comment 2 -->
    </goals>
    <configuration>
      <outputDirectory>
        ${project.build.directory}/modules
      </outputDirectory>
      <modules>
        <module>
```

```
            <artifact>                <!-- comment 3 -->
              <groupId>org.apache.commons</groupId>
              <artifactId>commons-lang3</artifactId>
            </artifact>
            <moduleInfo>              <!-- comment 4 -->
              <name>org.apache.commons.lang3</name>
            </moduleInfo>
          </module>
        </modules>
        <overwriteExistingFiles>true</overwriteExistingFiles>
      </configuration>
    </execution>
</plugin>
```

In this file, the line with `comment 1` binds plugin execution to the package phase; the line with `comment 2` calls the `add-module-info` goal; the section that starts with the line that has `comment 3` specifies the target dependency to update; and the section that starts with the line that has `comment 4` shows that in the additional module information section, only the name is required.

The build process now modularizes the `commons-lang3` dependency. This snippet is compatible with the previous snippet, so every dependency is copied to the `modules` folder, and it will be overwritten by the modularized JAR if it's not a module already.

The `commons-lang3` dependency is simple in two regards: It has an `Automatic-Module-Name` in its manifest and it has no external dependencies. For that reason, it's pretty easy to make use of it.

If you were to replace `commons-lang3` with the Guava library, for example, you would simply change the library name in the plugin. However, jdeps would explore the whole dependency tree and all of Guava's dependencies would need to be modularized as well, just as Guava itself would. This configuration would be quite verbose, but unfortunately it would be necessary. I've posted a copy of the pom.xml file.

Conclusion

In this article, I used jlink to create a custom launcher for a simple application. As we saw, jlink enables you to create launchers that contain only the required modules. To use jlink, an application itself must be modularized.

At that point, the complexity of the process depends on the compatibility of dependencies regarding modularization. If dependencies are modules themselves, all is straightforward. If not, they need to be transformed into modules before going further. Fortunately, the ModiTect plugin offers such a feature.

I hope that this article will help you to create smaller distributions of your apps, suitable for containerization. `</article>`

Nicolas Fränkel (@nicolas_frankel) has many years of experience consulting for various customers in a wide range of contexts. Usually he works with the Java/Java EE and Spring technologies, and he also focuses on interests such as rich internet applications, testing, continuous integration and continuous delivery, and DevOps. He currently works for Exoscale and has authored several books on programming and testing.

THE CONNECTICUT JUG

The Connecticut Java User Group is a technical community of more than 500 members that holds monthly meetings for Java software developers, engineers, managers, software architects, and computer science students across the state of Connecticut. Members work in many industries, including insurance, financial services, aerospace, and defense.

The Connecticut JUG was founded in 1999 by a group of employees from Computer Sciences Corporation (CSC). Initially, the JUG was a special interest group of the Connecticut Object Oriented Users Group (COOUG), which was the oldest object-oriented user group. COOUG had several thousand members and regularly attracted internationally known speakers. The Connecticut JUG continued to thrive after the dissolution of COOUG in the mid 2000s.

In October 2008, the Connecticut JUG organized its first conference. More than 110 developers from companies around the greater Hartford area attended the event, which featured two concurrent sessions and exhibits from local companies.

The Connecticut JUG meets monthly throughout the year in Hartford to discuss Java-related topics. Meeting information is posted to Meetup, Twitter, and soon to an upcoming website for the JUG, which at press time was in the process of migration. The JUG is currently sponsored by NEOS, a data science company based in Hartford.

OLEG ŠELAJEV

GraalVM: Native Images in Containers

Put Java apps into containers, run them as native apps, and get faster startup times and lower memory overhead.

GraalVM is a high-performance virtual machine for running programs in different languages. Currently, it supports JVM languages such as Java, Scala, Kotlin, and Groovy. It also supports JavaScript and Node.js, Ruby, R, Python, and the native languages that use LLVM. It's a very versatile project. However, one GraalVM capability is perhaps the most exciting for cloud deployments and the containers world. GraalVM can compile the JVM bytecode to the native executable or a shared library ahead of time in a way in which the resulting binary does not depend on the JVM for the execution.

This executable can be placed as a standalone application in a container and started really, really fast. Besides the quick startup time, GraalVM native images have low runtime memory overhead, which makes them even more attractive for use in the cloud.

Getting Started

Let's start at the beginning and create a GraalVM native image from an example application. First, you need a GraalVM distribution; download one from the GraalVM website. Both the community edition and the enterprise edition can create native images.

Unpack the archive and set `$GRAALVM_HOME` to point to the GraalVM directory; you can also point `$GRAALVM_HOME/bin` (or `$GRAALVM_HOME/Contents/Home/bin` on macOS) to the path for convenience. Once this is done, the utility for producing native images, called native-image, is available to you. Check the setup with `$GRAALVM_HOME/bin/native-image --version`.

Let's use GraalVM on a small example application. Clone `https://github.com/graalvm/graalvm-demos/` and navigate to the `native-list-dir` directory. There you'll find the `ListDir.java` class, which is a simple utility that traverses the filesystem and prints some information about what it finds. The code is straightforward:

```
public class ListDir { public static void main(String[] args)
         throws java.io.IOException {
    String root = ".";
    if(args.length > 0) {
        root = args[0];
    }
    System.out.println("Walking path: " + Paths.get(root));
    long[] size = {0};
    long[] count = {0};
    try (Stream<Path> paths = Files.walk(Paths.get(root))) {
        paths.filter(Files::isRegularFile).forEach((Path p) -> {
            File f = p.toFile();
            size[0] += f.length();
            count[0] += 1;
        });
    }
    System.out.println("Total: " +
        count[0] + " files, total size = " + size[0] + " bytes");
}
```

Compile this code to a `.class` file, because native-image operates on the bytecode level, which allows it to support other JVM languages, too.

After running `javac ListDir.java`, run `native-image ListDir`.

You also can use native-image on a collection of JAR files; you just need to specify the classpath and the main class for the executable. The native-image utility will analyze your application, statically determine which other classes it uses (both in your dependencies and the

JDK library), and create a map of reachable classes and method calls. It does this analysis statically and depends on a "closed universe" premise—making sure that all bytecode files ever to be executed in the resulting executable are present at the native image generation time.

One important feature of GraalVM native images is that the generation process can evaluate the static initializers of classes at generation time and store the preinitialized data structures in the resulting image heap.

Moments after the analysis, you can find a `listdir` file. For me, on macOS, it's a native macOS executable. It is linked to the operating system libraries directly without the JVM.

The file itself is a few megabytes. It contains the sample program compiled ahead of time and the JDK classes it uses, such as the `java.lang` classes or `Exception` classes, which can be thrown at any time. However, even with all the required classes, the size of these native executables is often smaller than the full distribution of the JDK that would otherwise be needed to run the program.

Let's run the Java version and the native binary and then time the execution using the UNIX `time` command in the Java directory:

```
$ time java ListDir
Walking path: .
Total: 7 files, total size = 8366834 bytes
java ListDir  0.22s user 0.06s system 51% cpu 0.555 total
```

Now let's run the Graal native implementation:

```
$ time ./listdir
Walking path: .
Total: 7 files, total size = 8366834 bytes
./listdir  0.00s user 0.00s system 66% cpu 0.011 total
```

You can see they produce the same result, and although the time of the Java version isn't bad, the time used by the native image version is almost zero.

One important feature of GraalVM native images is that the generation process can evaluate the static initializers of classes at generation time and store the preinitialized data structures in the resulting image heap. It's a configurable option, but it's useful for shaving off the last milliseconds of startup time.

This design, however, poses an interesting challenge for native images: What if the program you try to compile ahead of time uses production instance initialization in the class initializers, for example, creating thread pools, opening files, or mapping memory? It would not make any sense to perform these actions during the image generation phase, which usually won't be done in the production environment but on a continuous integration server, for example. The native-image utility will back down and refuse to compile your app if class initializers perform actions that don't make sense at image generation time. And you'd need to configure which classes should be initialized at runtime by using the `--delay-class-initialization-to-runtime=classname,list` option.

Handling Special Cases

There are a few more things that require configuration at native image generation time. The most obvious is, perhaps, reflection. Java code can inspect the class data, load additional classes, or invoke methods using the Reflection API. Because the Reflection API allows fully dynamic access to the classes and objects, static analysis cannot resolve all classes that must be included in the native image. This doesn't mean GraalVM native images cannot process any code that uses reflection. You just need to list ahead of time the classes and methods that will be used reflectively. The format of the configuration is a JSON file listing the classes and the files. Imagine you have two classes like the following, where one calls into the other via reflection:

```
package org.example;
class ReflectionTarget {
```

```
    public String greet();
}
```

and

```
import java.lang.reflect.Method;
public class Main {

    public static void main(String[] args) throws Exception {
        System.out.println(
            getResult(Class.forName("org.example.ReflectionTarget")));
    }
    private static Object getResult(Class<?> klass) throws Exception {
        Method method = klass.getDeclaredMethod("greet");
        return method.invoke(
                    klass.getDeclaredConstructor().newInstance());
    }
}
```

To compile them as a native image, you provide the following JSON file and specify it on the command line using the `-H:ReflectionConfigurationFiles=` command-line parameter:

```
[
  {
  "name" : "org.example.ReflectionTarget",
  "methods" : [
     {
       "name" : "<init>",
       "parameterTypes" : []
     },
     {
```

```
      "name" : "greet",
      "parameterTypes" : []
    }
  }
]
```

This file specifies which classes, methods, and constructors will be accessed reflectively. In a similar manner, you would typically need to configure Java Native Interface (JNI) access if the application you are compiling to a native image uses JNI.

You might imagine that providing such a configuration could become annoying, especially if the code that uses reflection is not yours but comes from a dependency. In such a case, you can use the configuration javaagent that GraalVM provides. Run your application with the agent attached, and it will record all uses of reflection, JNI, and anything else that you need to configure for the native image:

```
$ /path/to/graalvm/bin/java \
    -agentlib:native-image-agent=trace-output=/path/to/trace-file.json
```

You can run it multiple times, producing different trace files to ensure that all relevant code paths are executed at least once and the native-image utility has the full picture of the code you want to run.

You can run the tracing agent when you execute tests. Tests usually cover the most important code path. (If not, perhaps you should correct that first.) When traces are collected, you can turn them into a native-image configuration file:

```
$ native-image --tool:native-image-configure
$ native-image-configure process-trace \
  --output-dir=/path/to/config-dir/ /path/to/trace-file.json
```

The commands above will process the trace file and output the required configuration JSON files: `jni-config.json`, `reflect-config.json`, `proxy-config.json`, and `resource-config.json`.

After this preparation, using the generated configuration is pretty straightforward. The following command would take the configuration into account:

```
$ native-image -H:ConfigurationFileDirectories=/path/to/config-dir/
```

Another important configuration option to know is the `--allow-incomplete-classpath` option. Java applications often check for the existence of a class on the classpath and behave differently based on its availability. The classic example of such behavior is perhaps the logging configuration, which might state that if the `logback` library classes are available, then configure `logback`; otherwise, check for `log4j2` and configure it if it's available; if it's not, then fall back to `log4j`, and so on. How can native-image—which requires that all the classes be present for the analysis and which eagerly follows all the code paths—deal with such code? The answer is simple: By default, it currently refuses to compile code that uses this pattern, but if you explicitly say that an incomplete classpath is not a problem, it can compile the code without including those code paths.

There are plenty of configurable options that influence the behavior of the native-image generation, such as the ones we looked at earlier. As a developer, you have access to all the configurations that can make GraalVM native images successfully process more programs.

Performance

Let's consider the performance of native images. You saw in the earlier example that a native image can start in milliseconds. What about its throughput capabilities? After all, you know that just-in-time (JIT) compilers typically target peak performance rather than the speed of the startup or warm-up. Native images are not sluggish, but a warmed-up JIT compiler would be preferable performance-wise for long-running workloads. With this in mind, let's take a look at a sample Netty-based web service application.

First, build and produce the native image binary with the following commands:

```
$ mvn clean package
$ native-image -jar target/netty-svm-httpserver-full.jar \
   -H:ReflectionConfigurationResources=\
netty_reflection_config.json \
   -H:Name=netty-svm-http-server \
   --delay-class-initialization-to-runtime=\
io.netty.handler.codec.http.HttpObjectEncoder \
   -Dio.netty.noUnsafe=true
```

Now you can start the native file and check how fast it works for a single request and for some load.

For this test, I used the wrk2 benchmarking tool to generate the load and measure the latencies of the responses from the service. On my MacBook I ran the following, which specifies 2 threads and 100 simultaneous connections to keep a stable request rate of 2,000 per second for 30 seconds:

```
$ wrk -t2 -c100 -d30s -R2000 http://localhost:8080/
```

Here are the results. I'll first show the bytecode version of the Netty sample application and then the native version.

Java bytecode version:

```
Running 30s test @ http://127.0.0.1:8080/
  2 threads and 100 connections
  Thread calibration: mean lat.: 1.386ms, sampling interval: 10ms
  Thread calibration: mean lat.: 1.362ms, sampling interval: 10ms
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency     1.30ms  573.88us   3.34ms   65.01%
```

```
    Req/Sec     1.05k   181.18     1.67k    78.84%
  59802 requests in 30.00s, 5.70MB read
Requests/sec:   1993.21
Transfer/sec:    194.65KB
```

Native image version, which produces a very similar result:

```
$ wrk -t2 -c100 -d30s -R2000 http://127.0.0.1:8080/
Running 30s test @ http://127.0.0.1:8080/
  2 threads and 100 connections
  Thread calibration: mean lat.: 1.196ms, sampling interval: 10ms
  Thread calibration: mean lat.: 2.788ms, sampling interval: 10ms
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency     1.43ms  715.90us   5.78ms   70.34%
    Req/Sec     1.07k     1.37k    5.55k    89.40%
  58898 requests in 30.01s, 5.62MB read
Requests/sec:   1962.88
Transfer/sec:    191.69KB
```

[Note the output for both results has been slightly truncated to fit the page. —*Ed.*]

This is not a rigorous benchmark, of course, but these numbers demonstrate that for a short span, a native image can show similar performance to the JDK version of an application.

If you want even better throughput of native images, you can consider the Oracle GraalVM enterprise version of GraalVM, which is a proprietary product and includes additional performance enhancements. For native images, it includes profile-guided optimizations among other optimizations, which means you can build an instrumented image, gather the profile data by applying load, and then build the final image with performance optimizations that are tailored to the application's specific needs. This brings performance almost to the levels of the warmed-up JIT.

Memory Consumption

Let's talk about memory consumption. One of the typical complaints about using the JVM for serverless workloads is that it takes quite a bit of memory, even for a one-off task such as processing an individual request. (If you're interested in how GraalVM native images perform in that regard, the previous Netty application on my machine consumes 30 MB of memory total—including the heap.)

A native image also has garbage collection. It runs your program and collects unused objects at runtime to create the illusion of infinite memory. This is not new; any JVM does the same. Moreover, the JVM usually offers you a choice of garbage collection algorithms tuned for low latencies, minimal CPU consumption, or anything in between.

The garbage collector in native images is not the one you run in the JVM. Rather, it's a special implementation of a garbage collector written in Java that is a nonparallel generational scavenger. For simplicity, you can think of it as a somewhat simpler implementation of the parallel garbage collector that is the default in JDK 8. It splits the heap into generations; new objects are created in the so-called eden (see **Figure 1**) and then they are either collected or promoted to the old generation.

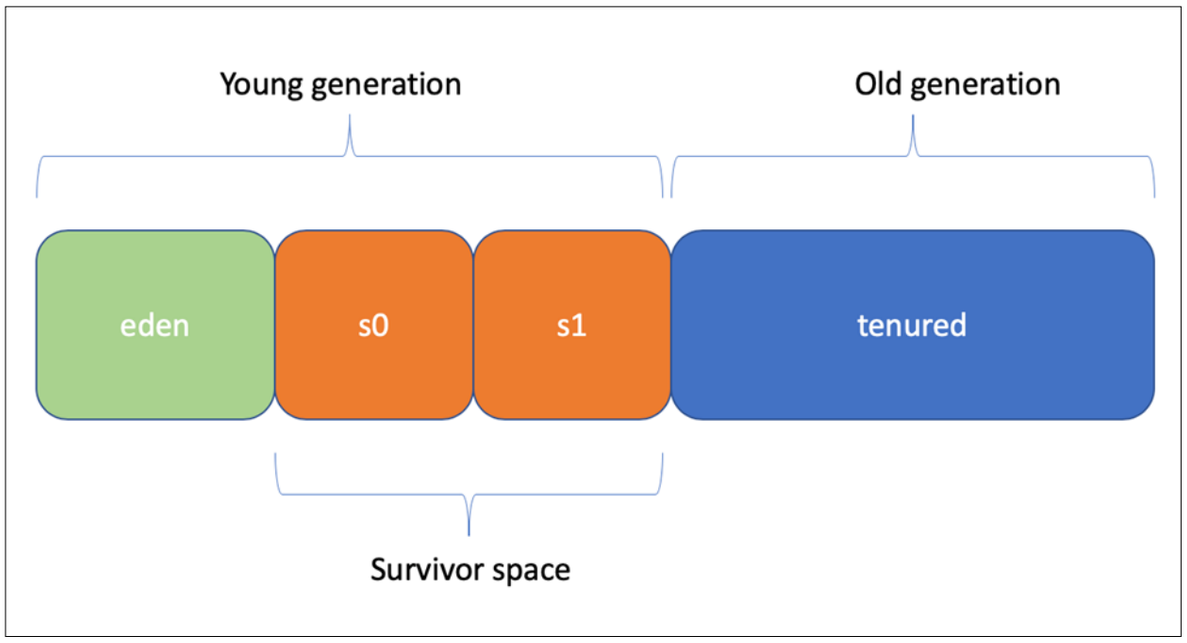


Figure 1. Garbage collector in native images

You can tune the garbage collector options for native images. Typically, you might want to adjust the maximum heap size. You can configure it with the `-Xmx` command-line parameter. If you'd like to better understand the garbage collector patterns of your native image, you can use the `-R:+PrintGC` or `-R:+VerboseGC` flags to get a summary of the garbage collector information before and after each collection. Native images often require less memory; one reason is that they do not need or include the machinery to load new classes dynamically, store their metadata for possible reflection, or compile them at runtime.

Conclusion

All in all, GraalVM native images offer a great opportunity to run Java applications in containers without loading the Java runtime. They also offer almost instantaneous startup and very low runtime memory overhead. This can be very important for cloud deployments where you want to autoscale your services or you have compute and memory constraints, such as in a function as a service (FaaS) environment.

Native images are an experimental feature of GraalVM, and today you can find applications that won't work with them out of the box. But many nontrivial apps work, and there are frameworks that accept GraalVM native images as a deployment target to simplify their usage. If you're deploying your apps in containers and you value startup performance and low runtime memory consumption, you'll likely find GraalVM native images very useful. `</article>`

Oleg Šelajev (@shelajev) is a developer advocate at Oracle Labs working on GraalVM—the high-performance embeddable polyglot virtual machine. He organizes the Virtual JUG (vJUG)—an online Java user group—and a Google developers' group (GDG) chapter in Tartu, Estonia. In 2017, he became a Java Champion.

RAOUL-GABRIEL **URMA**

RICHARD **WARBURTON**

New switch Expressions in Java 12

A new preview feature makes switch statements friendlier and less error-prone.

JDK 12 was released in March 2019. It's the third release using the six-month release cycle that was announced with the release of Java 9. What's in store this time? This article focuses on the new language feature available in preview mode: switch expressions. Our next article will cover the changes in the JDK, including the garbage collectors G1 and Shenandoah.

Switch Expressions

For those who get excited about new language features, you are going to be pleased with Java 12, which contains enhanced switch expressions. However, this feature is available only in preview mode currently. This means that simply running the Java compiler as usual won't work if you use the new switch expression syntax. To enable this feature, you'll need to use the flags `--enable-preview` and `--release 12` when you compile your code. To compile the code snippets in this article, make sure you have JDK 12 installed and use the following command:

```
javac --enable-preview --release 12 Example.java
```

To run the generated class file, you'll need to pass the `--enable-preview` flag to the Java launcher:

```
java --enable-preview Example
```

Before looking at the new feature, let's review what preview mode is. According to the official documentation, "A preview language or VM feature is a new feature of the Java SE Platform that is fully specified, fully implemented, and yet *impermanent*. It is available in a JDK feature release to provoke developer feedback based on real-world use; this may lead to it becoming permanent

in a future Java SE Platform. In other words, it can still be refined or even removed." [Emphasis added. —*Ed.*]

So what's wrong with the switch statement as you currently know it? There are four improvements that we discuss: fall-through issue, compound form, exhaustiveness, and expression form.

Fall-through issue. Let's start with the fall-through behavior. In Java, you typically write a switch as follows:

```
switch(event) {
    case PLAY:
        //do something
        break;
    case STOP:
        //do something
        break;
    default:
        //do something
        break;
}
```

Note all the `break` statements within each block that handles a specific `case` clause. The `break` statement ensures that the next block in the `switch` statement is not executed. What happens if you miss the `break` statement, though? Will the code still compile? Yes it will. As a quiz, try to guess the console output of the following code:

```
var event = Event.PLAY;

switch (event) {
    case PLAY:
        System.out.println("PLAY event!");
```

```
        case STOP:
            System.out.println("STOP event");
        default:
            System.out.println("Unknown event");
    }
```

The code prints the following:

```
PLAY event!
STOP event
Unknown event
```

This behavior in a switch is called *fall through*. As described in Oracle's Java SE documentation, "All statements after the matching `case` label are executed in sequence, regardless of the expression of subsequent `case` labels, until a `break` statement is encountered."

The fall-through behavior can lead to subtle bugs when you simply forget to include a `break` statement. Consequently, the behavior of the program could be incorrect. In fact, the Java compiler warns you of suspicious fall through if you compile with `-Xint:fallthrough`. The issue is also picked up by code checkers, such as Error Prone.

This issue is also mentioned in JDK Enhancement Proposal (JEP) 325 as a motivation for the enhanced form of `switch`: "The current design of Java's `switch` statement follows closely languages such as C and C++, and supports fall-through semantics by default. Whilst this traditional control flow is often useful for writing low-level code (such as parsers for binary encodings), as `switch` is used in higher-level contexts, its error-prone nature starts to outweigh its flexibility."

This new switch form uses the lambda-style syntax introduced in Java 8 consisting of the arrow between the label and the code that returns a value.

Now in Java 12 (with `--enable-preview` activated), there's a new syntax for `switch` that has no

fall through and, as a result, can help reduce the scope for bugs. Here's how you'd refactor the previous code to make use of this new `switch` form:

```
switch (event) {
    case PLAY -> System.out.println("PLAY event!");
    case STOP -> System.out.println("STOP event");
    default -> System.out.println("Unknown event");
};
```

This new `switch` form uses the lambda-style syntax introduced in Java 8 consisting of the arrow between the label and the code that returns a value. Note that these are not actual lambda expressions; it's just that the syntax is lambda-like. You can use single-line expressions or curly-braced blocks just as you can with the body of a lambda expression. Here's an example that shows the syntax of mixing single-line expressions and curly-braced blocks:

```
switch (event) {
    case PLAY -> {
        System.out.println("PLAY event!");
        counter++;
    }
    case STOP -> System.out.println("STOP event");
    default -> System.out.println("Unknown event");
};
```

Compound cases. Next is dealing with multiple case labels. Before Java 12, you could use only one label for each case. For example, in the following code, despite the fact that the logic for `STOP` and `PAUSE` is the same, you'd need to handle two separate cases unless you use fall through:

```
switch (event) {
    case PLAY:
```

```
            System.out.println("User has triggered the play button");
            break;
        case STOP:
            System.out.println("User needs to relax");
            break;
        case PAUSE:
            System.out.println("User needs to relax");
            break;
    }
```

A typical way to reduce the verbosity is to use the fall-through semantics of `switch` as follows:

```
switch (event) {
    case PLAY:
        System.out.println("User has triggered the play button");
        break;
    case STOP:
    case PAUSE:
        System.out.println("User needs to relax");
        break;
}
```

However, as discussed earlier, this style can lead to bugs because it's not clear whether the `break` statement is missing or intentional. If there were a way to specify that the handling is the same for the cases `STOP` and `PAUSE`, that would provide more clarity. That's exactly what is now possible in Java 12. Using the arrow-syntax form, you can specify multiple `case` labels. The previous code can be refactored like this:

```
switch (event) {
    case PLAY ->
```

```
            System.out.println("User has triggered the play button");
        case STOP, PAUSE ->
            System.out.println("User needs to relax");
};
```

In this code, the labels are simply listed consecutively. The code is now more concise and the intent clear.

Exhaustiveness. Another benefit of the new `switch` form is exhaustiveness. This means that when you use `switch` with an enum, the compiler checks that for any possible value there is a matching `switch` label.

For example, if you have the following `enum` type:

```
public enum Event {
    PLAY, PAUSE, STOP
}
```

And you create a switch that covers some but not all the values, such as the following:

```
switch (event) {
    case PLAY -> System.out.println("User has triggered the play button");
    case STOP -> System.out.println("User needs to relax");
}; // compile error
```

Then, in Java 12, the compiler will generate this error:

```
error: the switch expression does not cover all possible input values.
```

This error is a useful reminder that a default clause is missing or that you've forgotten to deal with all possible values.

Expression form. The expression form is the other improvement of the old `switch` statement: To understand what an "expression form" means, it's worthwhile reviewing the difference between a statement and an expression.

Statements are essentially "actions." Expressions, however, are "requests" that produce a value. Expressions are fundamental and simple to understand, which leads to better code comprehension and easier maintenance.

In Java, you can clearly see the distinction as it exists between an `if` statement and the ternary operator, which is an expression. The following code highlights this difference.

```
String message =  "";
if (condition) {
    message = "Hello!";
}
else {
    message = "Welcome!";
}
```

This code could be rewritten as the following expression:

```
String message = condition ? "Hello!" : "Welcome!";
```

Before Java 12, `switch` was a statement only. Now, though, you can also have switch expressions. For example, take a look at this code that processes various user events:

```
String message = "No event to log";
switch (event) {
    case PLAY:
        message = "User has triggered the play button";
        break;
    case STOP:
```

```
        message = "User needs a break";
        break;
}
```

This code can be written as a concise switch expression form that better indicates the intent of the code:

```
var log = switch (event) {
    case PLAY -> "User has triggered the play button";
    case STOP -> "User needs a break";
    default -> "No event to log";
};
```

Note how the `switch` now returns a value—it's an expression. This expression form for a switch also gives you the ability to use a `return` statement within a `case` block. However, we recommend extracting complex code logic into a private helper method with a meaningful name if you feel that things are becoming difficult to understand. You can then simply call this method using the expression-style syntax.

Toward Pattern Matching

You may be wondering what the motivation is for yet another language feature. Since Java 8, functional programming has clearly influenced the evolution of Java:

- Java 8 brought lambda expressions and streams.
- Java 9 included the Flow API to support reactive streams.
- Java 10 introduced local variable type inference.

All these ideas have been available in functional programming languages such as Scala and Haskell for a long time. So what's the latest idea for Java? It's (structural) *pattern matching*, which should not be confused with regular expressions.

Pattern matching, as it's meant here, is the idea that you test whether an object is of a particular structure before extracting certain parts of that structure to do some processing. This is typically accomplished in Java by using a combination of `instanceof` checks and cast expressions. A common situation in which to do this is when you need to write a parser for a domain-specific language. For example, the following code checks whether an object is of a particular type before casting it so that information can be extracted from it.

Switch expressions are a helpful addition that will enable you to write code that is a bit more concise and less error-prone.

```
if(o instanceof BinOp) {
    var binop = (BinOp) o;
    // use specific methods of the BinOp class
}
else if (o instanceof Number) {
    var number = (Number) o;
    // use specific methods of the Number class
}
```

Java doesn't yet support a language feature to provide full pattern matching, but that is currently being discussed as a potential addition, as described in JEP 305.

As an initial step in that direction, Java 12 needed to enhance the switch functionality that has existed since the first version of Java by making it an expression form. Through the introduction of this new feature, Java augments the switch syntax to enable future enhancements.

Conclusion

Java 12 doesn't bring any new language feature that you can readily use. However, it brings switch expressions, which are available as a preview language feature. Switch expressions are a helpful addition that will enable you to write code that is a bit more concise and less error-

prone. In particular, switch expressions provide four improvements: fall-through semantics, compound form, exhaustiveness, and expression form. Finally, and perhaps more exciting, the syntax available to `switch` has now become richer.

At the moment, in Java 12, the switch cases support only switching on enum, String, byte, short, char, int, and their wrapper classes. However, in the future there may well be more sophisticated forms and support for structural pattern matching on arbitrary "switchable" types. `</article>`

Acknowledgments. The authors wish to thank Oracle's Java langtools team for providing feedback on this article.

Raoul-Gabriel Urma (@raoulUK) is the CEO and cofounder of Cambridge Spark, a leading learning community for data scientists and developers in the UK. He is also chairman and cofounder of Cambridge Coding Academy, a community of young coders and students. Urma is the lead author of the best-selling program-ming book *Modern Java in Action*, (Manning 2018). He holds a PhD in computer science from the University of Cambridge.

Richard Warburton (@richardwarburto) is a software engineer, teacher, author, and Java Champion. He is the cofounder of Opsian.com and has a long-standing passion for improving Java performance. He's worked as a developer in HFT, static analysis, compilers, and network protocols. Warburton also is the author of the best-selling *Java 8 Lambdas* (O'Reilly Media, 2014) and teaches via Iteratr Learning and at Pluralsight. He holds a PhD in computer science from the University of Warwick.

NICOLAS **PONSINI**

Java Card 3.1 Unveiled

The major new release tunes the popular Java platform for IoT.

With close to 6 billion Java Card–based devices deployed each year, Java Card is the leading software platform for running security services on secure chips. These are used to protect smartphones, banking cards, government services, and other business needs.

Earlier this year, Oracle released Java Card 3.1—a major version of the Java Card specifications and the Java Card Development Kit. It is the most extensive update to the technology in several years, and it introduces new features to address the IoT market and new secure-elements hardware.

Two articles will detail the content of the 3.1 release. This first article presents the release and focuses on its extensions and services for IoT. An upcoming article will cover features that benefit payment, identity, and cellular connectivity markets.

What Is Java Card 3.1?

Java Card enables secure elements, such as smartcards and other tamper-resistant security chips, to host applications that employ Java technology. Java Card 3.1 is a major release updating all the components for developing Java Card products and applications, including the Java Card specifications and the development and compliance tools.

The specifications provide the basis for cross-platform and cross-vendor application interoperability:

- The Virtual Machine Specification provides the instruction set of the Java Card Virtual Machine, which is a supported subset of the Java language, and the file formats for installing applications and libraries into Java Card devices.
- The Runtime Environment Specification defines the necessary behavior of the runtime environment in any implementation of the technology. The Java Card APIs complement the Java Card

Runtime Environment Specification and describe the framework exposed by the technology.

- The Java Card Platform Specification facilitates the development and deployment of secure applications and introduces new functionality to support IoT security.

The Java Card Development Kit is a standalone environment for developing applications. It also includes a simulator and integration with the Eclipse IDE to facilitate the testing. Both the Java Card Platform Specification and the development kit are freely available to developers.

Oracle licenses a Java Card Reference Implementation and the Java Card Technology Compatibility Kit (TCK) to its commercial customers. The TCK ensures application interoperability for a Java Card implementation on a particular platform.

A key goal of version 3.1 is to ensure the availability of security services on a large range of secure hardware, including smartcards, embedded chips, secure enclaves within microprocessor units (MPUs) and microcontroller units (MCUs), and removable SIMs. It was designed to support the growth of existing Java Card markets, such as payment, identity, and connectivity markets, while enabling new IoT use cases with dedicated features. This article covers those features.

IoT Extensions

New features mirror the extended role that a Java Card secure element plays in a connected device. Specifically, the features include a new extensible I/O model and a range of security services to facilitate the design of new security applications. Let's examine these.

Secure elements embedded within a device or integrated within the system on a chip (SoC) of a device have recently been bringing security directly into the heart of devices. This enables use cases that establish a direct channel between a secure element and device peripherals. Making use of this capability requires specialized protocols at the application layer. To that end, Java Card 3.1 introduces an I/O framework, including the `javacardx.framework.event` package and the `javacardx.framework.nio` package, that allows applications to have logical access to device peripherals.

The `javacardx.framework.event` package is the base framework used by platform implementers to extend their platform with specialized APIs defining new I/O protocols or interfaces with peripherals. It contains:

- An `EventSource` that represents any peripheral or I/O interface in the device host or the secure element itself. Sources of events could be, for example, a GPIO pin or port, a UART interface, memory-mapped I/O, an I2C bus, a watchdog timer, and so on.
- An `EventListener` that enables code to handle peripheral or I/O events coming from a given source. The specification provides the default base interface, which is extended by platform implementers.
- An `EventRegistry`, which is a class used by applications to register listeners with a source of events.

The `javacardx.framework.nio` package contains classes for parsing and extracting structured information from raw data in an efficient way. These classes enable access to those data items from the heap and also from external memory (such as from a peripheral).

Numerous use cases can benefit from this I/O framework. For example, a Java Card application can directly read and verify fingerprint data from a biometric sensor. There is no need to go through the host device to transfer data from the biometric sensor to the main processor of the secure element, nor is there a need to tunnel data into application protocol data unit (APDU) commands to overcome associated constraints such as bandwidth, timing, ordering, priorities, execution context, and so on.

In IoT solutions, the enforcement of security policies can benefit from access to device peripherals and from collection of their data for decision-making at the edge. For example, the secure-element application in a smart meter could use localization or a motion sensor to detect abnormal situations and react accordingly. The application could also be used to securely configure attached peripherals and ensure the integrity of the control plane.

Security Services

Security services in Java Card 3.1 include the Certificate API, the Key Derivation API, the Monotonic Counter API, and the System Time API. Let's look at these in more detail.

Certificate API. Cryptographic certificates are critical for security and serve as a basis in a public key infrastructure (PKI) to establish trust between different entities. A notable example of a

protocol using cryptographic certificates is the Transport Layer Security (TLS) protocol. Based on certificate chains, a client (such as an IoT gateway) and a server (for example, an IoT cloud service) can authenticate each other.

The `javacardx.security.cert` package is an efficient way to manage cryptographic certificates such as X.509 certificates for memory- and resource-constrained devices.

With Java Card's Certificate API, it is possible to verify a certificate signature, select and check some of its fields and extensions, and access its public key—without needing to create a dedicated certificate object that is potentially useless in the future. You can also build a certificate object (for example, for root certificates) that will be reused later, while deciding on fields and extensions that need to be associated with this certificate object and storing only useful components of the certificate.

With these mechanisms, an application has an efficient way to verify a certificate chain, check sensitive certificate fields, and keep track of trusted public keys.

Key Derivation API. Pseudorandom functions (PRFs) and key derivation functions (KDFs) are widely used in cryptography to derive sensitive data such as a secret key. They make it possible to stretch a secret or to derive multiple keys from it.

A typical usage is the derivation of a password to store a derived value without needing to store the initial password value. Another common example is the derivation of the shared secret established by a key-agreement operation in Diffie-Hellman (DH) or Elliptic-Curve Diffie Hellman (ECDH) key exchange. Another example is the TLS handshake protocol, which uses a PRF applied to a shared secret in between a client and a server to generate the cryptographic block material used during a TLS session between the two peers.

Java Card 3.1 introduces the `javacardx.security.derive` package. Its class `DerivationFunction` permits the management of both PRF and KDF algorithms, and it is easily extensible. Currently, eight algorithms are proposed that enable support for the International Civil Aviation Organization (ICAO) or TLS protocols, among others.

In addition, the Key Derivation API guarantees both the security of the derivation keys and the derived keys by encapsulating them into trusted objects.

Monotonic Counter API. To prevent replay attacks, numerous security protocols use monotonic counters, which are counters whose value can only increase. Once the value of a monotonic counter has been used, the counter is incremented (typically by 1). Thus, if a counter value has been attached to a given protocol payload at a certain time, it is guaranteed that the same counter value cannot be reused and attached to the same protocol payload later. This allows a third party consuming the protocol payload to know whether it has already been used by checking the counter value.

Device attestation (also known as *remote attestation*) is an example of a payload that needs to be protected against replay attacks. A remote attestation is a signature of software measurements running on a given device, and the attestation is sent to a third party. The third party can check whether a device is running unaltered software and can make sure the attestation is current and not replayed from the past.

External secure storage is another example requiring a monotonic counter. DRM licenses or the number of PIN entry tries are typical sensitive data protected by a Java Card secure element. In some hardware architectures, like the ones with an integrated secure element, such sensitive data might be stored in the memory flash of the host device; that is, in storage external to the secure element hardware itself. Thus, untrusted and rogue applications from the device host might be able to save and restore later such sensitive data. In the case of a DRM license, this risk would mean that access to a content item is granted an unlimited number of times instead of the initial limited times granted originally. Hence, in addition to integrity and confidentiality, the secure element must guarantee an anti-replay protection to sensitive data stored in external secure storage.

The `javacardx.security.util` package and its `MonotonicCounter` class enable the creation and management of multiple monotonic counters of up to 64 bits each. The Monotonic Counter API guarantees the atomicity of the update of the counter value.

System Time API. Time stamping and time interval calculation are important security operations. Time stamps enable you to record or check the time at which an event occurred. Estimating a time interval allows you to limit the duration of a transaction, for instance.

Java Card 3.1 introduces the package `javacardx.framework.time`, which has two classes:

- `SysTime` serves to retrieve the uptime; that is, the time elapsed since system boot. It does not require an internal clock.
- `TimeDuration` represents a time duration with microsecond resolution. Several operations, such as comparisons and conversions as well as plus and minus operations, are supported.

Java Card's System Time API can support a variety of use cases related to device security. For example, consider an IoT device managing a temperature sensor in the chemical industry. This monitoring system is critical and needs to react to unexpected temperature variations. With the new I/O mechanism introduced in Java Card 3.1 and the System Time API, an application has the ability to retrieve a temperature value securely and to assess the elapsed time since the beginning of the measurement. If the time is too short or too long compared with the average expected value, this is reported to the monitoring system, which will trigger corrective operations such as changing the sensor or checking for a corruption of the IoT device software.

Conclusion

In this article, I described the major new IoT-oriented features of Java Card 3.1 and I detailed some of the related use cases in IoT security. For example, through trusted peripherals, Java Card can secure the "last yard" between devices, gateways, and attached peripherals, enabling trust and the exchange of sensitive data at the very edge. A secure channel can be established between peripherals and security chips to allow out-of-band communication for sensitive data (for example, biometric information or the provisioning of root-of-trust credentials).

Similarly, through device attestation, a Java Card secure element in an IoT device can support multiple proprietary or standard secure boot and device attestation mechanisms without requiring a dedicated security chip. This enables a single secure chip to be used in multiple attestation ecosystems and ensures compatibility with future standards.

In an upcoming article, I will describe the enhanced deployment model and core features as well as cryptographic extensions proposed by the 3.1 release. `</article>`

Nicolas Ponsini is a security solutions architect at Oracle, where he works on Java and the Internet of Things. He is an expert in security, cryptography, and IoT, and he owns 10 patents in related areas.

SIMON ROBERTS

MIKALAI ZAIKIN

Quiz Yourself

Try more intermediate and advanced test questions.

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

Answer 1
page 74

Question 1 (intermediate). The objective is to apply the `static` keyword to methods and fields. Given the following classes:

```
public class Test {
    int result = -1;
}

public class CodeTest extends Test {
    public static void main(String[] args) {
        // line n1
    }
}
```

What two lines added independently at line n1 will make the code compile successfully?

Choose two.

A. `result = 0;`

B. `this.result = 0;`

C. `super.result = 0;`

D. `new Test().result = 0;`

E. `new CodeTest().result = 0;`

Answer 2
page 79

Question 2 (intermediate). The objective is to use abstract classes and interfaces. Given the following:

```
interface Text {
    default String getContent() { return "Blank"; }
    void setContent(String txt);
    void spellCheck() throws Exception;
}

abstract class Prose {
    public abstract void setAuthor(String name);
    public void spellCheck() {
        System.out.print("Do generic prose spellcheck");
    }
}

Class Novel extends Prose implements Text {
  // line n1
}
```

Which two fragments added simultaneously at line n1 allow the following code to compile and run? Choose two.

```
Novel n = new Novel();
n.spellCheck();
```

A. `public void spellCheck() throws Exception {  }`

B. `String getContent() { return "Novel"; }`
C. `public String getContent() { return "Novel"; }`
D. `public void setAuthor(String a) {  }`
E. `public void setContent(String txt) {  }`

Answer 3
page 81

Question 3 (intermediate). The objective is to declare and initialize variables (including casting of primitive data types). Given the following primitive variable declarations:

```
char c = '1';
int i = 2;
long l = 3L;
float f = 4.0F;
double d = 5.0D;
```

Which of the following compile? Choose two.

A. `f = d;`
B. `l = f;`
C. `i = c;`
D. `f = l;`
E. `i = f;`

Answer 4
page 87

Question 4 (advanced). The objective is to submit queries and read results from the database, including creating statements, returning result sets, iterating through the results, and then properly closing result sets, statements, and connections. Given the following `WAREHOUSE` table:

ID	TITLE	QUANTITY
0	Cell Phone	10
1	Computer	20
2	TV	30

And the following code fragment:

```
Connection conn = … // properly initialized connection
Statement stmt = conn.createStatement
(ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_UPDATABLE);
ResultSet results = stmt.executeQuery("SELECT * FROM WAREHOUSE");
while (results.next()) {
    int id = results.getInt(1);
    if (id == 1) {
         results.updateInt(3, 15); // line n1
         results.next();
         break;
    }
}
results.previous(); // line n2
int qty = results.getInt(3);
System.out.print(qty);
```

Assume the database connection is properly initialized, any JDBC features used are supported by the driver and database, and any modes not explicitly shown are default.

What is the result? Choose one.

A. Runtime exception at line n1
B. Runtime exception at line n2
C. 15
D. 20

Answer 5
page 89

Question 5 (advanced). The objective is to use `java.util.concurrent` collections and classes, including `CyclicBarrier` and `CopyOnWriteArrayList`. Given the following `CBTest` class:

//fix this/

```
import static java.lang.System.out;
public class CBTest {

    private List<Integer> results =
      Collections.synchronizedList(new ArrayList<>());

    class Calculator extends Thread {
        CyclicBarrier cb;
        int param;
        Calculator(CyclicBarrier cb, int param) {
            this.cb = cb;
            this.param = param;
        }

        public void run() {
            try {
                results.add(param * param);
                cb.await();
            } catch (Exception e) {
            }
        }
    }

    void doCalculation() {
        // add your code here
    }

    public static void main(String[] args) {
        new CBTest().doCalculation();
    }
}
```

Which code fragment when added to the `doCalculation` method independently will make the code reliably print `13` to the console? Choose one.

A.

```
CyclicBarrier cb = new CyclicBarrier(2, () -> {
    out.print(results.stream().mapToInt(v ->  v.intValue()).sum());
    });
new Calculator(cb, 2).start();
new Calculator(cb, 3).start();
```

B.

```
CyclicBarrier cb = new CyclicBarrier(2);
out.print(results.stream().mapToInt(v -> v.intValue()).sum());
new Calculator(cb, 2).start();
new Calculator(cb, 3).start();
```

C.

```
CyclicBarrier cb = new CyclicBarrier(3);
new Calculator(cb, 2).start();
new Calculator(cb, 3).start();
cb.await();
out.print(results.stream().mapToInt(v -> v.intValue()).sum());
```

D.

```
CyclicBarrier cb = new CyclicBarrier(2);
new Calculator(cb, 2).start();
new Calculator(cb, 3).start();
out.print(results.stream().mapToInt(v -> v.intValue()).sum());
```

Question 1
page 68

Answer 1. The correct options are D and E. This question investigates a small part of the rules by which names are resolved, the distinction between static and nonstatic variables, and access to variables from a static context. Here, we present a simplified description of some of the rules laid out in *Java Language Specification* sections 6 and 15. Our discussion will be nowhere near as complete as those sections, in either scope or detail, but we aim to present a perspective that has sufficient scope to answer this question and others like it. The approach presented is sound as far as it goes, but it does have limitations. Undoubtedly some readers will have knowledge way beyond the intended audience of this question, and they will likely be able to see exceptions to this description.

Every variable in Java exists in a context. The same is true of methods, although this discussion will mostly focus on variables. That context can be a method, an object (an instance of a particular class), or a class.

Variables declared inside methods have a *method context* and are often called either a *method-local variable* or an *automatic variable*. This question isn't really concerned with variables in method contexts.

Variables declared inside classes—but outside of methods—are in either a class or an instance context. If such variables carry the `static` modifier, they exist in a *class context*. If such variables do not have the `static` modifier, they exist in an *object context*. With the object context, a different variable that has the same name exists in every object of the same type. With the class context, just one variable exists and it's considered to be part of the class.

Given this, it's perhaps not a surprise that to address any variable, the context in which it exists must also be identified. This identification can always be done explicitly, but it is often done implicitly.

If you have a class such as:

```
public class Car {
    public static int MAX_SPEED = 105;
    public int speed = 10;
}
```

And two objects created from this class:

```
Car c1 = new Car();
Car c2 = new Car();
```

Then the variable `speed` is an instance variable, and it must be referred to in the context of an instance of the class `Car`. Therefore, provided the variables `c1` and `c2` are in scope, you can write code in terms of `c1.speed` or `c2.speed`, but you cannot refer to `Car.speed`. This is because both `c1` and `c2` refer to contexts that contain a variable called `speed`, whereas the `Car` context does not. It's also worth noting that the contexts referred to by `c1` and `c2` each contain independent variables that simply share the same basic name. This, of course, allows you to model multiple cars that each has its own speed.

By contrast, the variable `MAX_SPEED` exists in the context of the class `Car`. Therefore, the proper way to refer to it is `Car.MAX_SPEED`. Perhaps unfortunately, because the compiler knows that `c1` and `c2` are of the type `Car`, it's also possible to refer to the exact same variable—that is, `Car.MAX_SPEED`—as either `c1.MAX_SPEED` or `c2.MAX_SPEED`. The syntax of those latter two is widely discouraged, but it is likely to show up on an exam precisely because it's potentially confusing and a competent programmer must not be confused by it. Notice that `c1.MAX_SPEED` and `c2.MAX_SPEED` look like different variables, but they're not; they're just aliases for `Car.MAX_SPEED`, and that ambiguity is a bad thing when it comes to creating understandable code.

In a few paragraphs, we'll discuss the context called `this`. At the risk of getting ahead of ourselves, know that if `this` exists, it too can be used as a prefix for a `static` variable. This is the same as using an instance prefix, and it's at least as ugly. If this paragraph confused you, ignore

it for now; keep going and revisit this paragraph after you've read the discussion on `this` that comes later.

Up to this point, *explicit contexts* have been discussed. *Implicit contexts* and the meaning of `this` and `super`, which bridge explicit and implicit contexts, also need to be considered.

Let's start with implicit contexts. An implicit context means there's no prefix in front of a variable name. In this case, the compiler must work out where to find the variable from one of three possible contexts.

First, the compiler checks if there's a method local variable in scope with that name. If one is found, that's what's accessed. Local variables always win.

If there's no local variable, the compiler looks for class or instance variables, starting at the most specific class definition and working up through the class hierarchy. Only one or the other can possibly exist in one class declaration; otherwise, the code won't compile.

If this search finds a `static` variable, that variable is used. Implicitly, the compiler has determined that the context is that of the enclosing class.

However, if the search finds an instance variable, the context *must* be the value known as `this`. However, `this` exists only in an instance (nonstatic) method. Instance methods must be invoked with a context, and that context becomes the value of `this` inside the method. Consider the following code fragment:

```
public class Car {
    private int speed;
    public void setSpeed(int s) {
        speed = s;
    }
}
```

And the following code:

```
Car c1 = new Car();
c1.setSpeed(55);
```

Inside the code of `setSpeed`, when the call to `c1.setSpeed(55)` is executed, the object referred to by the variable `this` is the same object that was referred to by `c1` at the moment of the call. That is, the context of the method call is `c1`, and that same context has been embedded inside `this` in that particular method invocation.

A static method cannot access an instance variable using this prefix, **either implicitly or explicitly.**

Now, the use of an implicit context—the implicit use of `this`—makes sense only if there is a value for `this`. And because that value comes from the instance prefix used for the method invocation, it's perhaps not a surprise that there is no value called `this` in a `static` method. As a result, a `static` method cannot access an instance variable using the `this` prefix, either implicitly or explicitly.

As a side note, you could be forgiven for thinking that the ugly syntax mentioned above—using an object prefix to invoke a `static` method—might create a `this` context in the `static` method, but it does not. This is another reason the syntax is considered ugly and to be avoided. When a static method is invoked in this way, the compiler simply takes the *type* of the variable prefix and discards the value. Of course, the syntax used to call the method is not known at the time the method is written, nor could you guarantee that all invocations would use this ugly form. Therefore, the call format cannot sensibly affect what is or is not permitted inside the method. The bottom line is that a `static` method does not have the use of `this` either implicitly or explicitly.

Another note is that it's common to hear people say "`static` methods cannot access instance variables." That's not accurate; they absolutely *can* do so, provided they have an explicit object context and that context is not `this`. (The normal rules of access control apply too; so, if the variable in question is in an instance of another class, it can't be `private` and it might need to be `public`.)

Moving on, there's a variation on the explicit context `this`, which is `super`. The `super` keyword means "the object referred to by `this`, but viewed with the parent type." Logically then, if `this` doesn't exist, neither does `super`. And if `this` exists (in other words, if you're inside an

instance method), `this` and `super` are equal in value, but different in type. The type of `super` is the type of the immediate superclass of the type of `this`.

Now that you've reviewed the background, you should be in a good position to evaluate the options and decide on the right answers.

The method that surrounds line `n1` is a `static` method. Because of that, you know that no `this` context is available—and neither is a `super` context. This fact is key in answering this question.

The code of option A makes an unqualified reference to a variable called `result`. The compiler will look for a method local variable of that name but fail to find one. Because of that failure, the compiler proceeds to look for a `static` variable in the context of the class `CodeTest` but again fails. Because the method is `static`, no `this` context is available, so the search in the class `CodeTest` has failed. The search then repeats up the parent class hierarchy (in the class `Test` and then in `Object`), but, of course, it fails again. Even though `Test` has a field called `result`, there's still no `this` context from which to access an instance. Therefore, the code fails to compile and option A is incorrect.

As a side note, paying close attention to the way that search was just described reveals that it's possible to have a `static` variable and an instance variable with the same name in the same object, but only if they are declared at different points in the hierarchy. Of course, it is probably evident that this is very bad practice, because it will cause confusion that will make maintenance harder. Just because syntax is legal doesn't make it good.

In option B, the code tries to make *explicit* use of the `this` context. But the enclosing method is still `static`, so no such context exists, the code cannot compile, and option B is also incorrect.

Option C trades the `this` explicit context for the `super` explicit context, but it was already established that the code is in a `static` method, so neither `this` nor `super` contexts exist, and the code also fails to compile. Therefore, option C is also incorrect.

In option D, the code creates an object of the type `Test`. This is successful, because the class `Test` is accessible and the class has an accessible zero-argument constructor (that's the compiler-generated default constructor, in this case). This newly created object satisfies the

need for an explicit object context for the variable `result`, and the code compiles. Therefore, option D is correct.

Option E does almost the same thing as option D, but it creates an instance of `CodeTest` rather than the base class `Test`. What matters, however, is that the `CodeTest` object forms a context that does contain a variable called `result`. Consequently, the code compiles, and option E is also correct.

As a final note, let's consider the accessibility of the variable `result` in this question. It's not specified explicitly and, therefore, it is the default—which means accessible from code within the same package. The code shown doesn't indicate a package, so how can you know if the access succeeds? There are three considerations that might be applied here.

First, you could consider that you see all the code and, therefore, both classes are in the default package. In this case, the access would succeed.

Second, see the exam guidelines and expand the "Review Exam Topics" section, which explicitly states the following:

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

Third, there's no possible way for the first three options (A, B, and C) to compile, but if you assume that the two classes are in the same package, they could provide an answer that's believable. Given a question that requires two selections, along with two believable answers and three impossible ones, you have no meaningful choice but to select the two believable ones.

Question 2
page 69

Answer 2. The correct options are D and E. The class `Novel` does not carry the modifier `abstract`. It is therefore a concrete class, and this requires that all the abstract methods it inherits, whether directly or indirectly, must have concrete implementations or the code cannot compile.

All the methods defined in an interface (in Java 8) are public, and they are `abstract`, `default`, or `static`. Any method that is not explicitly marked as either `default` or `static` is `abstract`. This means that the interface declares two abstract methods, which are `setContent` and `spellCheck`. These, therefore, must be implemented before the `Novel` class satisfies the requirements for being concrete. The interface additionally provides a default method called `getContent`. Although this is not considered a concrete method, it is not abstract either, and it's acceptable for a class that claims to be a concrete implementation of this interface to have no mention of this method.

At this point, you know that `Novel` must still provide or obtain concrete implementations for `setContent` and `spellCheck` (and that these implementations must provide the correct argument lists). Let's move on and consider what the abstract class brings to `Novel`.

A related question is whether a method that does not throw exceptions can satisfy a requirement for a method that does.

The abstract class `Prose` declares one abstract method, `setAuthor`. The `Novel` class must obtain or provide a concrete implementation for this. At this point, you're looking for concrete implementations of three methods to satisfy the needs of `Novel`. However, `Prose` also provides a concrete implementation of `spellCheck`, and the argument list exactly matches the same-named abstract method in the interface. This method satisfactorily resolves the requirement that the `Novel` class obtain or provide an implementation of the abstract `spellCheck` method declared in the interface. That leaves you with a need for two concrete methods to be implemented in the `Novel` class. These are `setAuthor` and `setContent`. These are the two answers in options D and E, which are the correct answers.

Now, let's explore why the incorrect answers are incorrect. Let's start by considering why option A is incorrect. The interface declares an abstract method `spellCheck` that throws `Exception` in its signature. Is the `spellCheck` defined in the `Prose` class sufficient for this? It turns out that, yes, it is. It's completely OK for an implementation of an interface method to be

inherited from elsewhere in the class's hierarchy, so the method can come from a parent class or from a default method in another interface.

Another related question is whether a method that does not throw exceptions can satisfy a requirement for a method that does. The answer to this, too, is yes. Just because a method declares an exception does not mean it will throw the exception; it means only that it might. Something that never actually throws the exception is OK. The converse, however, is not OK. A concrete method that declares checked exceptions that are not declared (either exactly or using a superclass) by an abstract method cannot correctly provide the implementation of that abstract method.

Option B will not compile. The method that it attempts to override is declared (and given a default implementation) in the interface, and even though no access modifier is provided, it is implicitly public. An overriding method may not *reduce* the accessibility of the method it overrides. Given that the method declared in option B has default access, which is less accessible than public access, compilation fails. Therefore, option B is incorrect.

A further observation on option B, from an exam-taking perspective, is that it's not *necessary* to make the code compile correctly. So even if it compiled, selecting it would "use up" one of the two options that you can select, and because both options D and E are necessary, if you chose option B, you'd end up missing one of the options that are required.

Finally, option C presents code that would compile but is not necessary. The method `getContent` is already provided by the default implementation in the interface. And as with the discussion on option B, if you selected option C, that would prevent you from selecting option D or option E; therefore, you should not select option C and it is incorrect.

As a side note, in Java 9, it's permissible to provide a private concrete method in an interface. However, for abstract methods, the rules have not changed; they are always public. Keep in mind that for the time being, the certification exams are still written for Java 8.

Question 3
page 70

Answer 3. The correct options are C and D. Generally, Java permits assignments of primitive types based on whether they'll work reliably. Therefore, if the entire range of values that can be represented by the type of a given expression can also be represented by the type of a given

destination type, the assignment from the expression to the destination type is permitted. The converse also applies: If an expression can represent values that are outside the range of the type of a destination, such an assignment is rejected.

Let's look at the ranges of the primitive types:

TYPE	EFFECTIVE STORAGE	RANGE OF VALUES
`boolean`	1 BIT	`true` AND `false`
`byte`	1 BYTE	-128 TO +127
`short`	2 BYTES	-32,768 TO +32,767
`char`	2 BYTES	0 TO 65,535
`int`	4 BYTES	PLUS OR MINUS ABOUT 2 BILLION
`long`	8 BYTES	PLUS OR MINUS ABOUT 9 * 1018
`float`	4 BYTES	PLUS OR MINUS ABOUT 3.4 * 1038
`double`	8 BYTES	PLUS OR MINUS 1.8 * 10308

Some things should be pretty clear:

- `boolean` cannot be assigned to or from a numeric expression.
- For the main integer numeric types, assignment is possible from smaller to larger; that is, from `byte` to `short` to `int` to `long`.
- For the floating-point types, assignment from `float` to `double` is possible.
- Assignment from any of the integer types to either of the floating point types looks good also.
- `float` and `double` do some magic to be able to store a vastly bigger range in the same amount of storage.
- Perhaps a little surprising is the fact that assignment between `short` and `char` cannot be performed in *either* direction. This is because a `char` can represent values greater than the maximum value of a `short` (65535 > 32767) and a `short` can represent negative values, which are not representable by a `char`.

These rules are more than sufficient to answer this question, although before we dive into the answers, we'll mention that the formal specification of these rules (and more) is described *Java Language Specification* sections 5.1.2 and 5.1.3.

Let's look at the options. Option A tries to assign a `double` value to a `float`. From our table, it's clear that the `double` can represent a much greater range than the `float`, so the assignment is not permitted, compilation fails, and you can conclude that option A is incorrect.

Assignments that might result in a completely wrong value—based on the ranges representable by the types—will be rejected by the compiler, but you can force the compiler's hand by using a cast.

Of course, you know that the actual value stored in the `double` is small enough to be represented properly by the `float` and, in general, if you are confident that the value about to be assigned does not overflow the capacity of the destination, you can use a cast to persuade the compiler to let you perform the assignment. In this case, the resulting code would look like this:

```
f = (float)d;
```

However, no cast is included in the code shown, so this does not alter the fact that option A fails to compile.

Option B tries to assign a `float` to a `long`, but it's clear from the table that the range of values representable by a `float` vastly exceeds that of a `long`; therefore, the assignment is refused, compilation fails, and option B is incorrect.

Note that the objection in option B is the loss of gross value, not the potential for loss of precision. You just saw that the `float` can represent a much greater range than the `long`, and because of that, the assignment is rejected. But imagine that you have a `float` that contains 3.14, and you cast and assign that to a `long`. The result would be 3, and the fractional part would be lost entirely. This, and some situations like it, might be called "loss of precision." Although it's

not evident or even possible here (because the `float`-to-`long` assignment—and, actually, assignment of any Java floating-point primitive type to any Java integer primitive type—is always rejected due to loss of range), you'll see later that Java permits assignments that might simply lose some precision.

Option C assigns a `char` value to an `int`. Every value that can be represented by a `char` can be represented perfectly by an `int`. Therefore, the result is reliably accurate, the compiler permits it, and option C is correct.

Option D assigns a `long` to a `float`. The range of a `float` is much greater than the range of a `long`, and the compiler allows this. Therefore, option D is also correct. It might seem odd that a `long` has 8 bytes of effective storage and a `float` has only 4 bytes. That might seem like option D should be rejected. But again, the issue here relates to loss of precision, not loss of gross value. This will be investigated further in a moment.

Option E attempts to assign a `float` to an `int`, but as the table shows, the range of a `float` is vastly greater than that of an `int` and, therefore, the attempt is rejected by the compiler and option E is incorrect.

At this point, you've seen that assignments that might result in a completely wrong value—based on the ranges representable by the types—will be rejected by the compiler, but you can force the compiler's hand by using a cast. A cast operation is safe if you're sure the value will fit.

You've also seen that an assignment that risks losing only *precision*, *not* gross value, is permitted. The rules described in *Java Language Specification* sections 5.1.2 and 5.1.3 are concerned with what's permitted and what might happen, not with understanding the consequences. At this point, you've not seen how an assignment could cause the issue raised here. So now let's look closer at it and as a side thought consider how a 4-byte `float` value can store a larger range than an 8-byte `long` value.

It's easy to understand losing the fractional part when you assign (through a cast) a `float` to an `int`, but what does it mean to "lose precision" when you assign, for example, a `long` to a `float`? The answer lies in how the `float` manages to have a wider range than the `long`, despite using half the storage.

Floating-point numbers in Java (in fact, in most computer languages) don't actually count "units" in the way that integer values do. Instead, they count in a variable chunk size. Although it's possible a value is counted in ones, it might be counted in sixteenths or five-hundred-and-twelfths. This behavior is how these types of numbers handle fractions. But similarly, the number might be counted in chunks of two hundred and fifty-six. This is how these numbers can have much greater ranges than might be expected.

In essence, the floating-point number's storage is split into two parts. One part stores the number of "chunks" (this is called the *significand*) and the other part indicates the size of each chunk (this part is called the *exponent*). Wait—isn't an exponent the thing after the *E* in a number such as 3E+12? Yes, in effect, it is. When you write 3E+12, you're saying "three chunks" where each chunk is worth one followed by twelve zeros. You could equally say 314159E-5, which would be the same as 3.14159 without having a "fraction" and with having only the information that each unit (of which there are 314,159) has a value of a one-hundred-thousandth of 1.

With that background on the floating-point number representation, let's get back to the idea of loss of precision. It turns out that for a `float` value in Java, the significand (the "chunk count") uses 24 bits, which means that the largest value it can represent is 224, which is 16,777,216. Notice this is much less than the largest value of an `int` and very much less than the upper limit for a `long`. If you try to use a `float` value to store a number bigger than this, the chunk size must be increased. Initially the count is in twos, and then after 16,777,216 twos, the count is by fours.

There are a couple of immediately demonstrable consequences to this. First, if you try to assign to a `float` the literals 16,777,216; 16,777,217; 16,777,218; and 16,777,219, you see that it takes on the values 16,777,216 (accurate); 16,777,216 (rounded down by one); 16,777,218 (correct); and 16,777,220 (rounded up by one). You can try this easily by casting the literals to `float`, and then casting them back to `int`s, like this:

```
System.out.println((int)((float)(16777216)));
System.out.println((int)((float)(16777217)));
```

```
System.out.println((int)((float)(16777218)));
System.out.println((int)((float)(16777219)));
```

Another effect that is perhaps even more significant is what happens in this loop. Try to guess how many times this prints `incrementing`. Then copy the following code and find out if you were right.

```
float count = 33554432;
while (count < 33554435) {
    System.out.println("incrementing");
    count = count + 2;
}
```

Of course, `double` numbers have essentially the same behavior, although the boundary numbers are different and, in fact, a `double` can accurately represent the full range of values of an `int` (it has a 53-bit significand, which has much greater range than a 32-bit `int`).

The point here is that loss of precision means just that: When you assign an `int` to a `float` or a `long` to either a `float` or a `double`, you might end up with an approximation of your original number. And this approximation might have practical consequences for your code.

If you're interested in more detail, Java (at least in `strictfp` mode) uses IEEE 754 floating-point representations and a Wikipedia page provides more details.

This discussion is now complete, right? Well, no. Look back at that table of data type ranges. The second column is entitled "effective storage." Why is that significant? Why not simply use the title "storage"? It's significant because the storage space used for a variable isn't specified by the language or the virtual machine specification.

For example, on modern 64-bit processors, it's possible that the machine cannot efficiently address single bytes or perhaps even 4-byte words. In this situation, a particular implementation is free to allocate more storage than is strictly needed for the data.

What the specifications *do* mandate is that the integral data types must behave as if they

are "two's complement binary numbers" with the specified amount of storage. For floating-point, the specification mandates that the behavior must be exactly compliant with the IEEE 754 specification only if the class or method carries the modifier `strictfp`. The bottom line is that although the previous table describes what the numerical behavior will be, you cannot assume that allocating 1,000 `int` variables will reliably allocate 4,000 bytes of memory. This discrepancy can be even more startling with `boolean` values. It's possible that an array of 64 `boolean`s might be packed into a single 8-byte word, but it's also possible (though perhaps not very likely) that each individual value might take 8 bytes. The reality is likely somewhere in between; but crucially, neither the language specification nor the virtual machine specification mandate this behavior.

Question 4
page 70

Answer 4. The correct option is D. This question investigates several aspects of the behavior and the use of result sets. One aspect is the operations that are permitted based on the configuration of the statement that produces the result set. Another is the sequence of operations necessary to update a table through a `ResultSet`.

The `createStatement(...)` method exists in three overloaded forms. The form used in this question accepts two `int` parameters that define the "result set type" and the "concurrency," respectively.

The result set type parameter is selected from three possible values:

- `ResultSet.TYPE_FORWARD_ONLY`: This specifies that the result set may be traversed only in a forward direction and traversed only once. (This is the default that's implied if the zero-argument overload `createStatement()` is used.)
- `ResultSet.TYPE_SCROLL_INSENSITIVE`: This specifies that the result set permits arbitrary traversal, backward and forward, and absolute positioning. With this mode, if some other process performs updates on the underlying table, they will not be visible through this result set object.
- `ResultSet.TYPE_SCROLL_SENSITIVE`: This specifies the same traversal freedom of the previous mode, but if data in the underlying database is updated during navigation, those changes will be visible through this result set object.

It is true that not all JDBC drivers support all types of result sets, but the question tells you to assume that everything used is supported. Therefore, there's no reason to expect the mode configuration to cause a problem. And, as noted, the `ResultSet.TYPE_SCROLL_SENSITIVE` type permits forward and reverse navigation through the result set, so line n2 can be expected to complete without raising an exception. Because of this, you can see that option B is incorrect. (If the mode had been configured as `ResultSet.TYPE_FORWARD_ONLY`, option B would have been correct.)

The second parameter passed into the `createStatement(...)` method defines the concurrency type of the future result set. It may accept two values:

- `ResultSet.CONCUR_READ_ONLY`: This is the default mode (when you create a statement without customization), and it allows only reading of the data from the result set.
- `ResultSet.CONCUR_UPDATABLE`: This allows updating of the database through the result set.

As with the previous parameter, in practice, a driver or database might not support the updatable concurrency mode, but the question specifies that you can assume all modes are supported. Therefore, you can expect the update call at line n1 to complete without an exception. Because of this, you can see that option A is incorrect. (Again, note that if the mode had been configured as `ResultSet.CONCUR_READ_ONLY`, line n1 would throw a `java.sql.SQLException` exception, and option A would have been correct.)

Having determined that the code runs successfully and, therefore, generates some output from the `print` statement, let's consider what will be printed. It seems clear that the programmer's intention was that the output should be 15; however, it won't be that because the code fails to use the `ResultSet` object properly.

Often, when a row in a result set is updated, multiple individual fields are changed, and with the result set, this must be done using multiple calls to `update`*Xxx* methods. If each of these calls committed a change to the persistent storage, it would be expensive in terms of multiple writes and it would unnecessarily create transactionally inconsistent partial changes. Because of this, the `update`*Xxx* methods modify only the row data in memory—even though the connection is in auto-commit mode—and the accumulated changes must be written to persistent storage under the explicit control of the programmer. The process of updating a row through a result set is completed by using a call to the `updateRow()` method on that result set, like this:

```
results.updateRow();
```

If this call were included right after line n1, the change to the table would be persisted and the result would be 15 (and, consequently, option C would have been correct).

However, in the absence of the call to `updateRow()`, the changes to the result set are not persisted, and option C is incorrect. As a result, the output is 20 and option D is correct.

One final note is that in this question it does not matter whether you use `ResultSet.TYPE_SCROLL_INSENSITIVE` or `ResultSet.TYPE_SCROLL_SENSITIVE` because the changes are not being performed by another thread or process. With the question as written, no changes are made anyway, and even with the addition of the call to `updateRow()`, the changes are visible to this result set, because they're made through it.

Question 5
page 71

Answer 5. The correct option is A. This question investigates some of the core features of the `java.util.concurrent.CyclicBarrier` class, including its purpose and use.

The `CyclicBarrier` class is a feature of the `java.util.concurrent` package and it provides timing synchronization among threads, while also ensuring that data written by those threads prior to the synchronization is visible among those threads (this is the so-called "happens-before" relationship). These problems might otherwise have been addressed using the `synchronized`, `wait`, and `notify` mechanisms, but they are generally considered low-level and harder to use correctly.

If multiple threads are cooperating on a task, two problems must commonly be addressed. (Note that other issues might exist, too). One problem is that the data written by one thread must be read correctly by another thread when the data is needed. Another problem is that the other thread must have an efficient means of knowing when the necessary data has been prepared and is ready to be read. The major feature of the `CyclicBarrier` is to provide timing synchronization using a *barrier point*.

The operation of the `CyclicBarrier` might be likened to a group of colleagues at a conference preparing to go to a presentation together. They get up in the morning and go about their routines individually, getting ready for the presentation and their day. When they're ready, they

go to the lobby of the hotel they're staying in and wait for the others. When all of the colleagues are in the lobby, they all leave at once.

Similarly, a `CyclicBarrier` is constructed with a count of "parties" as an argument. In the analogy, this represents the number of colleagues who plan to go to the presentation. In the real system, this is the number of threads that need to synchronize their activities. When a thread is ready, it calls the `await()` method on the `CyclicBarrier` (in the analogy, this is arriving in the lobby). At this point, one of two behaviors occurs. Suppose the `CyclicBarrier` was constructed with a "parties" count of 3. The first and second threads that call `await()` will be blocked. Being blocked means their execution is suspended, using no CPU time, until some other occurrence causes the blocking to end. When the third thread calls `await()`, the blocking of the two threads that called `await()` before is ended, and all three threads are permitted to continue execution. (This is the second behavior mentioned earlier.)

A second effect of the `CyclicBarrier` is that after the block is ended, data written by any of the threads prior to calling `await()` will be visible (unless it is perhaps subsequently altered, which can confuse the issue) by all the threads that called `await()` on this blocking cycle of this `CyclicBarrier`. This is the happens-before relationship and it addresses the visibility problem.

After the threads are released, the `CyclicBarrier` can be reused for another synchronizing operation—that's why the class has *cyclic* in the name. Note that some other synchronization tools in the `java.util.concurrent` API cannot be reused in this way.

The `CyclicBarrier` provides two constructors. Both require the number of parties (threads) they are to control, but the second also introduces a new behavior called the *barrier action*:

```
public CyclicBarrier(int parties, Runnable barrierAction)
```

The barrier action defines an action that is executed when the barrier is tripped; that is, when the last thread enters the barrier. This barrier action will be able to see the data writes by the awaiting threads, and any writes by the barrier action will be visible to the threads after they resume. The API documentation states, "Memory consistency effects: Actions in a thread prior to calling `await()` *happen-before* actions that are part of the barrier action, which in turn *happen-*

before actions following a successful return from the corresponding `await()` in other threads."

Now that you know what the `CyclicBarrier` does in general, let's review the code. Each of the options creates a `CyclicBarrier` and passes it to a thread (created from the `Calculator` class). Each thread—or each calculator, if you prefer—performs a calculation and then adds the result of that calculation to a thread-safe `List` that's shared between the two threads. Note that the `ArrayList` itself isn't thread-safe, but the `Collections.synchronizedList` method creates a thread-safe wrapper around it. After adding the result to the `List`, the calculator thread calls the `await()` method on the `CyclicBarrier`. Subsequently, the intention is to pick up the data items that have been added to the `List` and print their sum.

For this to work correctly, the summing operation must see all the data written by the calculators and not occur until after the calculated values have been written. The code should achieve this using the `CyclicBarrier`.

In option B, the attempt to calculate the sum and print the result precedes the construction and start of the two calculator threads. As a result, option B is incorrect.

Option B might occasionally print the right answer; the situation is what's called a *race condition*. Although unlikely, it's not impossible that the JVM might schedule its threads in a way that the calculations are completed before the summing and printing starts. It's also possible that the data written in this situation might become visible to the thread that performs the summing and printing. However, it's unlikely at best and certainly not reliable. The output is most likely to be 0 (because the list is empty), but the values 4, 9, and 13 are all possible.

Option D is a variation of option B with swapped lines. Although this looks like the calculations might be executed before the summing and printing operations, the same uncertainty exists. So, although this option is more likely than option B to print 13 on any given run, for the same reasons, all the values are possible. Therefore, option D is incorrect.

Option C is almost correct, but not quite. The `CyclicBarrier` is created with a "parties" count of three. Three calls to `await` are made, so the main thread would not proceed until the two calculations are complete. The timing would be correct and the visibility issue would be correctly addressed such that the last line of the option—the line that computes and prints the sum—would work reliably if it were not for one remaining problem with the implementation shown.

Blocking behaviors in the Java APIs are generally interruptible, and if they are interrupted, they break out of their blocked state and throw an `InterruptedException`, which is a checked exception. Because neither the code of option C nor the body of the `doCalculation` method into which the code is inserted include code to address this exception, the option fails to compile.

In fact, the `await()` method throws another checked exception, `BrokenBarrierException`, and this is also unhandled. However, while you might not know about the `BrokenBarrierException`, you should definitely know about the `InterruptedException` because it's a fundamental and pervasive feature of Java's thread management model. Because these checked exceptions are unhandled and the code does not compile, option C is incorrect.

In option A, the `CyclicBarrier` is created with two "parties" (which will be the two calculator threads) and also with a barrier action. The barrier action is the lambda expression that aggregates results from working parties. The two `Calculator` threads invoke `await()` after they have written the result of their calculations to the list. This behavior ensures that both writes have occurred before, and the data is visible to, the barrier action. Then, the barrier action is invoked to perform the summing and printing. As a result, it's guaranteed that both 4 and 9 are in the list before the summing and printing and that they are visible to that operation. Consequently, the output must be 13, and you know that option A is correct. `</article>`

Simon Roberts joined Sun Microsystems in time to teach Sun's first Java classes in the UK. He created the Sun Certified Java Programmer and Sun Certified Java Developer exams. He wrote several Java certification guides and is currently a freelance educator who publishes recorded and live video training through Pearson InformIT (available direct and through the O'Reilly Safari Books Online service). He remains involved with Oracle's Java certification projects.

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

//contact us/

Comments

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

Article Proposals

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

Customer Service

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

Where?

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

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

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